Effective Use of Non-blocking Data Structures in a Deduplication Application

Steven Feldman[1], Akshatha Bhat[2], Pierre LaBorde[1], Qing Yi[3], Damian Dechev[1,4]
[1]University of Central Florida, [2]University of Texas at San Antonio
[3]University of Colorado at Colorado Springs [4]Sandia National Laboratories
Feldman@knights.ucf.edu, akshathab@gmail.com, pierrelaborde@knights.ucf.edu
qyi@uccs.edu,dechev@eecs.ucf.edu

Abstract

Efficient multicore programming demands fundamental data structures that support a high degree of concurrency. Existing research on non-blocking data structures promises to satisfy such demands by providing progress guarantees that allow a significant increase in parallelism while avoiding the safety hazards of lock-based synchronizations. It is well-acknowledged that the use of non-blocking containers can bring significant performance benefits to applications where the shared data experience heavy contention. However, the practical implications of integrating these data structures in real-world applications are not well-understood. In this paper, we study the effective use of non-blocking data structures in a data deduplication application which performs a large number of concurrent compression operations on a data stream using the pipeline parallel processing model. We present our experience of manually refactoring the application from using conventional lock-based synchronization mechanisms to using a wait-free hash map and a set of lock-free queues to boost the degree of concurrency of the application. Our experimental study explores the performance trade-offs of parallelization mechanisms that rely on a) traditional blocking techniques, b) fine-grained mutual exclusion, and c) lock-free and wait-free synchronization.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

Keywords lock-free synchronization, parallel data structures, concurrent data deduplication, multiprocessor software design, C/C++ multithreading

1. Introduction

As modern architectures evolve to feature an increasingly large number of CPU cores, a key challenge in developing multi-threaded applications is correctly synchronizing shared data while avoiding excessive performance penalties. Unsafe low-level synchronization mechanisms can easily introduce errors, e.g. race conditions and deadlock, which are extremely difficult to debug. At the same time, application performance and scalability are frequently compromised due to inefficient implementations of synchronous operations on shared data.

Recent advances in the design of lock-free and wait-free data structures bring the promise of practical and highly scalable library containers for concurrent programming [1, 2]. Existing results indicate that these non-blocking data structures can avoid the safety hazards of blocking synchronization and outperform their lock-based counterparts in many scenarios [1–6], especially when the shared data experience high degrees of contention. As demonstrated by Tsigas et al. [7], large scale scientific applications using non-blocking synchronization generate fewer cache misses, exhibit better load balancing, and show better scalability when compared to their blocking counterparts.

In spite of the increasing demand for non-blocking data structures and the existing design of a variety of non-blocking containers, the practical implications of integrating these data structures into existing applications are not well-understood. In this paper we present a study on integrating non-blocking data structures within a data deduplication application from the PARSEC benchmark suite [8]. Data deduplication algorithms are highly concurrent and extremely important in new-generation backup storage systems to compress storage footprints, and in bandwidth-optimized networking applications to compress communication data transferred over the network.

The algorithm implementation applies a large number of concurrent operations to a data stream using the pipeline parallel processing model, in a scenario different from the typical use case scenarios explored in previous studies of non-blocking algorithms. By demonstrating the use of non-blocking synchronization within this implementation of data deduplication, we expect our results to be applicable to other applications of a similar behavior pattern and to potentially enable better implementations of such algorithms in terms of performance, scalability, and progress guarantees.

We have manually refactored the original *Dedup* application from PARSEC from using conventional lock-based synchronization mechanisms to using a wait-free hash map and several lock-free queues to enhance the degree of concurrency within the application. We seek a better understanding of two important issues: 1) the transformations required to modify an existing multi-threaded application from using lock-based synchronizations to using non-blocking synchronizations; 2) the use case scenarios where non-blocking data structures provide significant performance benefits. Our contributions include the following:

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10. . . $15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2508075.2508431

- We analyze the design of two non-blocking data structures, a hash map and a queue, present detailed steps necessary to incorporate them within an existing C/C++ multi-threaded application, and study their behavior when used in combination with blocking algorithms and data structures.
- We use a full-scale application and several micro benchmarks to study the performance trade-offs of traditional blocking mechanisms, fine-grained mutual exclusion, and the up-and-coming non-blocking techniques.
- We outline semantic differences between the use of traditional shared containers and their non-blocking counterparts and outline important tactics in the design and utilization of existing non-blocking data structures to enhance their usability in real-world applications.

The rest of the paper is organized as follows. Section 2 introduces some basic concepts of non-blocking synchronization. Section 3 describes the overall structure of Dedup. Section 4 outlines our methodology for refactoring the application and the alternative concurrent data structures that we have integrated with the refactored *Dedup* code. Section 5 presents experimental results to evaluate the performance and scalability of the various synchronization approaches. Finally, Section 10 discusses related work, and Section 11 summarizes our findings.

2. Background

In multi-threaded programming, shared data among multiple threads need to be protected to ensure thread safety. A commonly used protection mechanism is to use a global lock to enforce mutual exclusion, so that at most one thread can update a shared data item at any time. However, when many threads are contending for a single lock, the sequential ordering of updates becomes a serious performance bottleneck that limits the scalability of parallel applications.

Existing research has sought to ameliorate this problem by redesigning data structures to support fine-grained locking [9, 10], which uses different locks for disjoint components of the data structure, e.g. one lock per bucket in a hash map, allowing operations that modify disjoint items of a compound data structure to proceed concurrently. Besides potential performance scalability issues, lock-based synchronization requires meticulous attention to details when threads need to acquire and release multiple locks, to avoid unsafe situations such as deadlock, livelock, starvation, and priority inversion [2].

Using atomic primitives supported by hardware, a large collection of *non-blocking algorithms* has been developed to support lock-free and wait-free synchronizations over shared data, where in a finite number of steps, a subset of the participating threads is guaranteed to make progress [2, 11] (lock-free), or *all* participating threads are guaranteed to make progress [2, 11] (wait-free). These guarantees are typically based on constructing shared data structures that support *linearization* of concurrent operations as a main correctness invariant; that is, they support seemingly instantaneous execution of every operation on the shared data, so that an equivalent sequential ordering of the concurrent operations can always be constructed [2].

The practical design of non-blocking data structures is known to be difficult. In particular, non-blocking algorithms are known to vary widely, and there is no general recipe for their design. A typical non-blocking algorithm generally includes three phases: 1) determining the current state of the shared object, 2) deciding whether to assist a pending operation or proceed with own operation, 3) attempting own operation, retrying (return to step 1) if necessary.

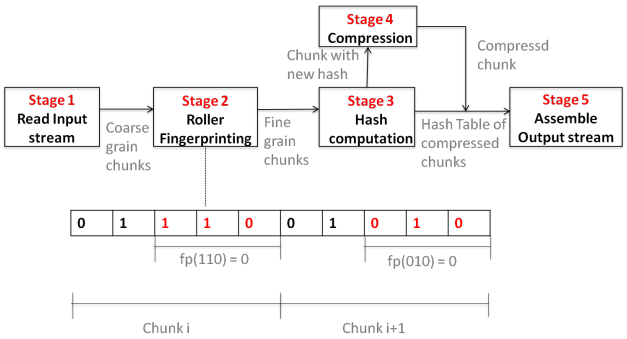


Figure 1: Pipeline stages of Dedup

While widely adopted, this approach is susceptible to the ABA[1] problem [12], which occurs when the value of a memory location changes from A, to B, and then back to A. An erroneous solution to the ABA problem occurs when a thread does not realize the intermediate nature of the update to B and proceeds to incorrectly change the final value of the entire update sequence from A to B.

3. The Dedup Application

To study the impact of using differently synchronized data structures on the performance and scalability of real-world applications, we selected *Dedup*, a data deduplication application which applies several global and local compressions to an input data stream, from the PARSEC benchmark suite [8]. The algorithm uses a pipelined programming model to parallelize the compression, where the input stream is first broken into coarse-grained chunks and then divided into fine grained segments, with each segment hashed and compressed before stored into a hash map. Finally, an output stream is assembled from the hash values and compressed data segments. The algorithm is widely used in new-generation backup storage systems, e.g., the IBM ProtecTIER Deduplication Gateway, to compress storage footprints, and in bandwidth-optimized networking applications, e.g., the CommVault Simpana 9, to compress communication data.

3.1 Computation Structure

Fig. 1 illustrates the overall computational structure of *Dedup*, which is comprised of five pipeline stages. The three stages in the middle (`Roller Fingerprinting, Hash computation, and Compression`) are parallelized using multiple threads, while the first (`Read Input stream`) and last (`Assemble Output stream`) stages are single-threaded and mostly serve to handle program I/O. The first stage reads the input stream and breaks it up into coarse-grained chunks as independent work units to be operated on by the threads of the second stage. The second stage further fragments the chunks into finer-grained segments via rolling fingerprinting, which uses the Rabin-Karp fingerprinting technique to break up coarse-grained chunks into finer-grained ones. A unique sequence number is associated with each chunk so that they can be ordered correctly when they are reassembled. The third stage computes a SHA-1 hash value for each fine-grained chunk and stores the chunks into a hash map; if any chunk already exists in the hash map, the chunk is considered a duplicate and omitted by the later stages. Next, the fourth stage compresses each remaining chunk before storing it back into the hash map. Finally, the last stage assembles the deduplicated output stream by combining the compressed data segments in the correct order based on their unique

[1] ABA is not an acronym; it refers to a value in shared memory changing from A to B and then to A again.

sequence numbers while replacing the duplicate chunks with their hash values (fingerprints) in the final result.

The algorithm uses two types of concurrent data structures, a global hash map and four shared queues, one per pipeline stage, to store the data generated from each pipeline stage before they are further processed. In the original implementation of the algorithm, both the hash map and the local queues are synchronized via global locks to support concurrent access of their data. Our study focuses on replacing the implementations of these lock-based concurrent data structures with alternative non-blocking data structure implementations, to observe the impact of non-blocking containers on the overall application performance.

3.2 Execution Configurations

The *Dedup* application supports a number of ways to reconfigure the concurrent execution of its pipeline stages. e.g. by using different numbers of threads or local queues within each pipeline stage. In particular, it maintains a separate thread pool for each of the middle three pipeline stages so that an arbitrary number of threads can be assigned to work on each stage. The default configuration uses a single queue to maintain the data stream between each pair of pipeline stages. However, to reduce contention among the threads, the application can be configured to automatically use multiple queues so that only a small group of threads are sharing each queue.

Overall, the *Dedup* application can be configured with two parameters: *n*, the number of threads to be allocated to each of the middle three pipeline stages; and Max_{th}, the maximum number of threads allowed to share a single local queue (by default, Max_{th} = 4). Based on these two parameters, the application adjusts the number of local queues per pipeline stage as $\lceil n/Max_{th} \rceil$. For example, if the user specifies that 32 threads should be used per pipeline stage and a maximum of 4 threads are allowed to share a single queue, then 8 local queues are created per stage.

The strategy within *Dedup* to dynamically adjust the number of local queues per pipeline stage serves as a scheduling mechanism within the application to ensure proper load-balancing among the threads and to reduce contention on the local queues. Section 5 presents our results when experimenting with different configurations of these two parameters to determine their impacts on the overall performance of the application.

4. Refactoring Dedup

To investigate the potential of automatically refactoring existing applications to use non-blocking concurrent data structures and the overall impact of such application transformations in terms of promoting stronger progress guarantees, better scalability, and improved overall performance, we have manually replaced the lock-protected data structures within *Dedup* with several alternative implementations and have modified the externally synchronized operations on the original shared data with new internally synchronized APIs. The following subsections detail our modifications to the application and their semantic implications.

4.1 Application Transformations

Modifying existing parallel codes to use non-blocking concurrent data structures requires a number of considerations, including semantic differences of the alternative data structures, the API's used to access them, modifications of synchronization schemes to protect shared data, and the implications of making them lock-free. We have taken the following three steps to systematically modify the *Dedup* application to address these issues.

1. Identify shared data structures protected by global locks. Through manual examination of the application source code and its profiling runs, we have identified two critical shared data structures within *Dedup*: the hash map used to track duplications of data items, and the local queues used to store the data stream between pipeline stages. The use of these data structures within the application is illustrated in Fig. 1.
2. Analyze the use of the shared data structures and identify equivalent concurrent containers that can be used as alternative implementations. Determine a common API, e.g., similar to that of the unordered maps and queues from the C++ Standard Template Library (STL), for the alternative concurrent containers.
3. Rewrite the application to organize their shared data using the alternative containers and to access the shared data using a designated common API for the containers. In particular, to substitute the original lock-protected hash map and queue operations in *Dedup*, we manually identified all instances of blocking synchronizations within the application and replaced them with equivalent alternative operations to the new concurrent data structures.

We have explored several alternative implementations of the hash maps and local queues. The following subsections summarize each of these implementations, the progress guarantees they provide, and their expected performance characteristics.

4.2 Substituting The Hash Map

The original hash map used within *Dedup* has a similar interface as that of the *unordered_map* in the C++ Standard Template Library (STL) except that it allows the insertion of duplicate keys. Additionally, a special function, *hashtable_getlock(key)*, is provided that returns a lock to support mutual exclusion for all operations related to the designated hash key. While a thread holds this lock, no other thread can make progress on the hash map with a key that hashes to the same position.

Fig. 2a shows an example code fragment that operates on the *Dedup* hash map. Here the first instruction at line 1 acquires the lock for some hash key, line 2 calls the search function, lines 3-6 insert a new entry to the map if the key is not found, lines 7-10 process the returned result, and finally line 11 releases the lock. This fine-grained locking approach supports the parallelism of operations that operate on disjoint positions in the map. However, if two or more threads are operating on the hash map with keys that hash to the same position, these operations will be serialized. A linked list is used to store an arbitrary number of keys at each location to accommodate hash collisions. As a result, while a typical search operation is expected to take constant-time, it could take O(n) time, where n is the number of entries already in the hash map, in the worst case. The size of the hash map is a hard-coded constant that must be chosen wisely to avoid lengthy linked lists.

To explore the performance trade-offs of varying concurrent hash maps, we substituted the original *Dedup* hash map with the following alternative implementations.

- The Concurrent Hash Map from Intel's Thread Building Blocks (TBB) library. The TBB hash map incorporates both lock-free and fine-grained synchronization techniques to provide a high degree of concurrency and scalability [9]. All operations on this hash map, except for the resize, are performed in O(1) time. It utilizes internal locks to maintain correctness and provides the user the option of holding read or write locks on each specific key value [9]. It differs from the *Dedup* hash map in that a thread needs to acquire a lock only if it needs to update the value associated with a key. If a thread is simply inserting a new key or reading the value of a key, no explicit locking is required.
- The Wait-Free Hash Map implemented by [13, 14]. This implementation provides a high degree of concurrency and scalabil-

```
1  lock=hashtable_getlock(key);
2  hash_entry *res= hashtable_search(hashtable,key);
3  if(!res){
4       ....
5       hashtable_insert(hashtable,key, value);
6  }
7  else{
8       ....
9       res.value=...
10 }
11 lock.unlock();
```

(a) Blocking Hash Map

```
1
2  void *res= hash_get(hashtable,key,thread_id);
3  if(!res){
4       ...
5       bool res2=hash_insert(hashtable,key,value,thread_id)
6       if(!res2)
7            goto found;
8  }
9  else{
10 found:
11      ...
12      atomic_update(res,value)
13 }
```

(b) Wait-free Hash Map

Figure 2: Hash Map Code Transformation Example

ity in addition to guaranteeing that all operations complete in a constant number of steps. In particular, it uses a bounded number of linked arrays to store the inserted key-value pairs. Each entry of an array may hold nothing, a reference to a key-value pair, or a reference to another array. Each hash key is interpreted as a sequence of bits, where pre-determined sub-sequences of these bits are used to determine the path to follow to find a key-value pair in the linked arrays. The first sub-sequence of bits determines the entry on the root array where the key-value pair would be placed. If the desired entry already holds a reference to another array, the next sub-sequence of bits is used to determine the entry on the next array to place the key-value pair. The process repeats until the key-value pair has been placed successfully, and the number of repetitions is bounded by a constant value (the maximum length of the key). New arrays are added incrementally when hash collisions occur, with the total number of referenced arrays limited by the number of bits in each hash key.

It is also important to consider Doug Leas java implementation of a concurrent hash map [15] which is similar in design to `Dedups` concurrent hash table, in that they both resolve hash collisions by using linked lists. However, Lea's concurrent hash map allows for get (search) operations to concurrently execute without the need to block or synchronize with other operations unless an inconsistent state is detected. Additionally, it supports the remove operation by cloning the portion of the linked list before the node to be removed and joining it to the portion of the linked list pointed to by that node. Allowing for the get function to "detect that its view of the list is inconsistent or stale. If it detects that its view is inconsistent or stale, or simply does not find the entry it is looking for." In the event that it does detect this "it then synchronizes on the appropriate bucket lock and searches the chain again." The design of Lea's concurrent hash map allows for non-conflicting operations to proceed independently, and serializes modification operations that operate on the same segment of data. Since his design is based on Java's garbage collection and synchronization functions it is not compatible with `dedup` and was not included in the implementations tested.

Fig. 2 illustrates the program transformations we made in order to substitute *Dedup's* hash map with the above alternative concurrent hash maps. The main difference in the transformed code is that by using a concurrent hash map, explicit locking is no longer required when performing an update operation, as the APIs are already thread-safe when working concurrently. In Fig. 2a, *Dedup* uses locks to ensure that its hash map only contains unique keys. These locks are not necessary when using a thread-safe hash map. In the event that between the search and insert operations of the hash map at lines 2 and 5 of Fig. 2b, a different thread inserts the same key, the insert operation at line 5 would return false, and the execution would proceed as if the search function had found the key in the data structure.

4.3 Substituting The Queues

Fig. 3a outlines the original implementations of the three queue operations in *Dedup*, where a global lock is used to enforce mutual exclusion of operations on each queue container. In contrast to the typical API of queues in the C++ STL, a key difference in *Dedup's* queue implementation is that each enqueue and dequeue operation performs a batch operation of multiple elements to increase throughput every time a thread acquires the global lock of a queue. Additionally, multiple queues could be employed at each pipeline stage to reduce the number of threads operating on the same queues, further reducing contention among the threads and wait time induced by the locks.

A shared queue is a fundamentally sequential data structure where a high degree of contention can be experienced by its head and tail pointers, leading to increased contention and reduced parallelism even when using fine-grained synchronization on the few heavily-contended data of interest. As a result, using non-blocking methods may not deliver as large of a performance boost as that provided by fundamentally concurrent containers such as the hash maps. To investigate such contention issues, we have replaced the original local queue implementation within *Dedup* with the following non-blocking alternative implementations.

- The non-blocking concurrent queue from Intel's Thread Building Blocks library is implemented using a sequence of arrays called micro-queues [9]. Each queue operation is assigned to one of the micro-queues, with the Kth operation assigned to the $K\%N$th micro-queue, where N is the number of micro-queues (by default, N = 8). In essence, the micro-queues are used to remove the contention from a single head or tail at the cost of relaxing the FIFO property of a queue.
- The lock-free queue based on the popular Michael-Scott queue algorithm [16] from the Amino Concurrent Building Blocks library is fundamentally a linked list in which each value is stored in a distinct node. A back-off scheme was added to the queue to allow contending threads to sleep for a pre-defined amount of time in hopes that the scheduler will provide a more favorable ordering of the threads in the future, so that they no longer impede each other's progress. ABA-prevention [12] has been added to the Amino implementation based on Michael's safe memory-reclamation algorithm [17].

To substitute the original queue implementation in Fig. 3a with the alternative lock-free implementations, we have modified each implementation of the three batch operations with alternative im-

```
1  function enqueue(queue q, int chunk_size, void *buf){
2     q.lock();//Enter Critical Region
3     while( q.isFull() ) //Block until queue is full
4       q.wait();
5
6     for( item=0; item<chunk_size; item++)//Enqueue items
7         q.data[queue.head++] = buf[item];
8     q.signal();//Signal enqueue
9     q.unlock();//Leave Critical Region }
10
11
12 function dequeue(queue q, int chunk_size, void *buf){
13     q.lock();//Enter Critical Region
14
15     //Block if queue is empty and if other threads
16     //are still working on the queue
17     while( (q.head==q.tail) && (q.thCmpl < q.threads) ){
18       q.wait();}
19
20      //Exit condition
21      if( (q.tail == q.head) && (q.thCmpl == q.threads) ) {
22          q.signal();
23          q.unlock();
24          return; }
25
26      for( item=0; item<chunk_size; item++){//Dequeue items
27          buf[item] = q.data[q.tail++];
28          if(q.tail == q.head) //Queue is Empty
29             break; }
30      q.signal();
31      q.unlock(); //Leave Critical Region
32 }
33
34
35 function terminate(queue q){
36     lock(); //Enter Critical Region
37     q.thCmpl++;
38     unlock(); } //Leave Critical Region
```

(a) Using blocking queues

```
1  function enqueue(queue q, int chunk_size, void *buf){
2     //No locking required
3
4
5
6     for( item=0; item<chunk_size; item++)//Enqueue items
7         q.concq.enqueue( buf[item] );
8
9  }
10
11
12 function dequeue(queue q, int chunk_size, void *buf){
13     //No locking required
14
15
16     //Conditional wait replaced with Busy Wait
17     while( q.concq.empty() && (q.thCmpl < q.threads) )
18     {  }
19
20      //Exit condition
21      if( q.concq.empty()&& (q.thCmpl == q.threads) ) {
22
23
24          return; }
25
26      for( item=0; item<chunk_size; item++){//Dequeue items
27          r = q.concq.dequeue( buf[item] );
28          if( !r ) //If Dequeue fails
29             break; }
30
31
32 }
33
34
35 function terminate(queue q){
36     //No locking required
37     //Shared variable q.thCmpl is replaced with atomic counter
38     _sync_add_and_fetch( q.thCmpl, 1 ); }
```

(b) Using lock-free queues

Figure 3: Code Transformation for Queues in Dedup

plementations that operate on a non-blocking concurrent queue data structure, outlined in Fig. 3b. In essence, the `decorator` design pattern is used to adapt a concurrent queue to the original batch operation interface of `Dedup`. Compared with the the original queue implementations, the new implementation in Fig. 3b has removed from Fig. 3a the locking and unlocking operations at lines 2, 9, 23, and 31; and the waiting-while-full block at lines 3–4, as the enqueue and dequeue functions of a lock-free queue are thread-safe and have unbounded capacities. Further, the locking around the variable that tracks the number of completed threads (lines 36–38) has been replaced with a single atomic increment of the *thCmpl* value. Finally, the enqueue and dequeue APIs of the lock-free queue are called a specified number of times to support the original batch enqueue and dequeue operations of *Dedup*.

4.4 Semantic Implications

As discussed in Section 4.1 and illustrated in Figs. 2 and 3, a key challenge in refactoring *Dedup* to use alternative hash map and queue implementations is to recognize the uses of these data structures and their respective programming interfaces within the application and to identify equivalent concurrent data structures to substitute the original APIs. In many cases, equivalent data structures provide slightly different APIs due to their different internal synchronization mechanisms. For example, using lock-based synchronization, an application can first examine the head of a queue before popping it off. However, such operations may not be possible using a non-blocking queue designed without support for a linearizable "check-and-pop" method. Similarly, the application may need to check whether an element is present in a hash map before insertion, which is not easily implemented in a non-blocking hash map. Fortunately both TBB's concurrent hashmap and the wait-free hash map we used support such combination of operations, as shown in Fig. 2.

To automate the process of refactoring *Dedup*, a full understanding of the semantics of the involved hash map and queue implementations is required. In particular, a data structure annotation language could be used to convey their APIs and expected use patterns, so that an automated source-to-source translation framework could be developed to identify the relevant code fragments and perform the necessary program transformations. However, due to the highly complex and dynamic nature of C/C++ applications, we expect that user interaction will be required to ensure correct and complete coverage of the necessary transformations.

5. Experimental Configurations

To understand the performance implications of integrating non-blocking data containers within existing well-tuned multi-threaded applications, we studied the performance variations of *Dedup* when using the alternative implementations for its two shared data structures, the hash map and local queues. To gain a deeper insight into the observed performance differences, we additionally designed a number of synthetic tests to evaluate the relative efficiencies of these data structures independently of the *Dedup* application.

Each synthetic test starts with a main thread that initializes a set of global variables, creates an instance of each data structure to be studied, and then spawns a predetermined number of threads to evaluate a mixture of different operations on the data structure of

interest. The main thread records the elapsed time between signaling the threads to begin execution and after all threads have completed execution. Care has been taken to remove all unnecessary work between operations.

Input files (small to larger)	Simsmall (10.10mb), Simmedium (30.70mb), Simnative (671.58mb)
Threads per Pipeline Stage (TPPS)	1,2,4,8,16,32,48,64
Maximum Number of Threads per Queue (MTPQ)	1,2,**4**,8,16,32,48,64
Items per En/Dequeue (EDITEMS)	1,5,10,15,**20**,...60

Figure 4: Execution Configurations

Fig. 4 summarizes the different execution configurations (see Section 3.2) of *Dedup* that we have evaluated for each combination of hash map and queue implementation. All experimental evaluations were performed on a 64-core SuperMicro server, with four sixteen-core AMD Opteron (Model 6272) 2.1 Ghz processors and 64GBs of memory. The machine runs 64-bit Ubuntu Linux version 11.04. All programs were compiled with g++ version 4.7 with the -O3 optimization flag. The default configurations of these parameters are printed in bold in Fig. 4 and are used when comparing the hash map and queue implementations in Sections 6 and 7. Each evaluation has been repeated ten times, and the average of the ten runs are reported as the result of the evaluation. Since we have a dedicated machine for our evaluations, the average variation across different runs of the same evaluation is about 0.3%.

6. Comparing Hash Map Implementations

This section provides a study of the performance trade-offs of the original *Dedup* hash map, TBB's unordered map, and the wait-free hash map discussed in Section 4.2. Fig. 5 shows the performance comparison of *Dedup* when using the three different hash map implementations and when operating on a small (Simsmall), a medium (Simmedium), and a large (Simnative) size input data streams. The key differences between these implementations are the progress guarantees they provide and how they handle hash collisions and resizes. We expect the wait-free hash map to perform the best when the number of keys is relatively large and hash collisions occur often, and the original *Dedup* hash map to degrade in performance as the number of keys increases due to the unbounded linked lists that must be searched during every operation.

From Fig. 5a and 5b, we see that when using the small or medium-sized input, the performance differences from using different hash map implementations are minor, with the wait-free hash map performing slightly better than the other two implementations. Since only a limited amount of work is available at each pipeline stage, all versions reached their best performance when using 4 threads per stage, and adding more threads produces either no benefit or leads to performance degradation (e.g., when using more than 32 threads per stage to operate on the *Simsmall* input).

When operating on the much larger *Simnative* input, for all different thread configurations, the wait-free hash map consistently produces a 7-21% performance boost over the alternative hash map implementations. Here, since a much larger amount of work is available at each pipeline stage, the performance peak is not reached until at least 16 threads are used per stage. The increased work leads to the possibility for a higher degree of contention when multiple threads attempt to simultaneously update the same entries of the shared hash map. Further, since a large amount of data need to be stored in the global hash map, many of these data may incur hash collisions. To achieve scalable performance, an implementation with an efficient collision management methodology is needed. Intel TBB's unordered map performs similarly to the original *Dedup* map for the majority of cases, with about a 2-6% improvement over *Dedup's* hash map when a large number of threads are used per pipeline stage (e.g., 64 threads/stage in Fig. 5c).

We hypothesize that the wait-free hash map performs better than *Dedup's* hash map and Intel TBB's hash map because: 1) it provides better collision management by guaranteeing a constant bound on both the search and insert operations, and 2) it supports a higher degree of concurrency when multiple threads are competing to update the hash map. To validate this hypothesis, we designed two synthetic tests and present the results in Fig. 6.

Since operations from different threads are executed sequentially only when they need to operate on a common position within the hash map (that is, when they need to operate on the same hash key or when their keys incur a collision within the hash map), our first synthetic test aims to maximize these occurrences. By artificially increasing the total number of operations to 15 million (an 18x increase), with similar distributions of search and insert operations as those in `Dedup`, we expect to trigger a much larger degree of contention in the hash maps and to observe their relative efficiencies of handling the contention and their ability to scale.

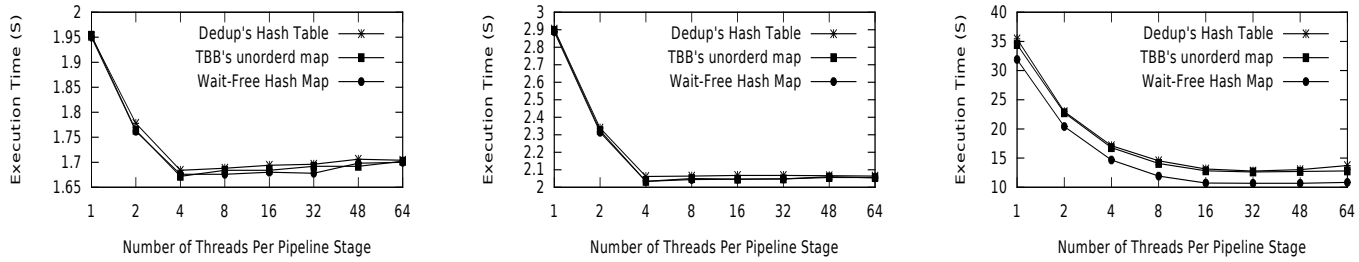
Fig. 6a compares the performance of the different hash map implementations when each thread uses a list of randomly generated SHA1 values (the hash key data type used by *Dedup*) to perform its share of the 15 million operations, where 16% of the operations are insert operations and 84% are search operations. Note that in *Dedup*, there is only a total of about 810,000 operations on the hash map when using the *Simnative* input. From Fig. 6a, we see the much larger number of operations indeed result in more contention, which in turn severely degrades the performance of the TBB and *Dedup* hash maps so that the wait-free hash map now performs orders of magnitude better (41.72x - 838.81x).

To isolate the effect of hash collision management from the overall scalability issues of the *Dedup* application, our next synthetic test aims to significantly reduce the chances of hash collisions in all three hash map implementations without affecting the probability of multiple threads contending to operate on the same hash key. Our study revealed that the SHA1 key data type used by Dedup imposes an artificial dependence among the different keys so that overall a much higher number of collisions results, as many keys are hashed to a small number of clustered positions within the TBB and *Dedup* hash maps. This defect in the Dedup hashing function, however, may have a much smaller impact on the wait-free hash map as it is more resilient to poor hashing functions.

To validate this hypothesis, Fig. 6b compares the performance of the hash maps when using a hash function with a more even distribution. Here, since the keys are much better distributed, the number of hash collisions is significantly reduced. Consequently, the performance differences are mostly caused by the different degrees of concurrency when multiple threads are operating on the same key. When up to 32 threads are used, the wait-free hash map outperforms the alternative approaches. In addition, both the wait-free and the TBB hash maps perform significantly better when compared to the original Dedup hash map due to their use of fine-grained synchronization. Note that the performance differences between the TBB and the wait-free hash maps are much smaller, as both of them support a high degree of concurrency when multiple threads need to work on the same data.

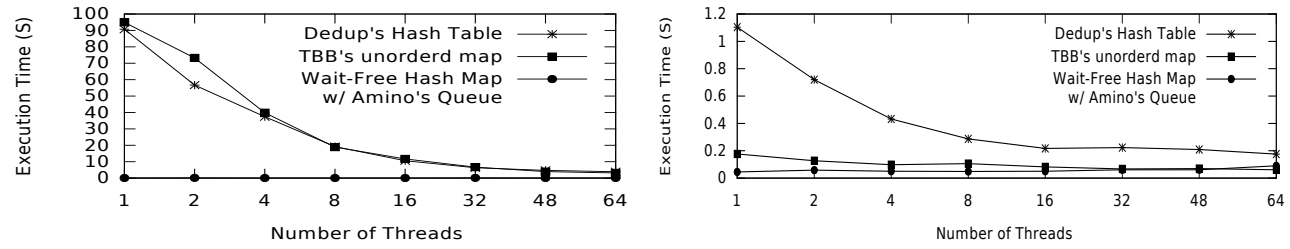
7. Comparing Local Queue Implementations

In contrast to the concurrent hash map, where operations with distinct hash keys can proceed in parallel independently of each other, all operations on a shared queue occur at either the head or the tail of the queue. Because of its inherently sequential First-In First-Out (FIFO) property, the degree of concurrency that a queue can support is limited. Hence, concurrent operations on a shared queue typically lead to a high degree of contention and thus performance degrada-



(a) Input: Simsmall (b) Input: Simmedium (c) Input: Simnative

Figure 5: Performance of Dedup when using different hash map implementations



(a) 20 Byte SHA1 Key (b) 4 Byte Integer Key

Figure 6: Synthetic Hash Map Tests

tions. As a result, we expect low impact on the overall performance of *Dedup* when replacing its coarse-grained lock-protected queue with two non-blocking concurrent queues, the Intel TBB's queue and the Amino Concurrent Building Block's (CBB) queue (see Section 4.3). Fig. 7 shows our experimental results of comparing the performance impact of the different queue implementations. These results confirm that the integration of queues that support a higher degree of concurrency does not have a significant impact on the overall performance of *Dedup*.

The *Dedup* application can be reconfigured with two key parameters, the maximum number of threads per queue and the number of items to be enqueued or dequeued per operation (see Section 3.2). In particular, when the maximum number of threads operating on each queue is decreased, so is the contention on the head and tail of the queue. Increasing the number of items enqueued or dequeued per operation has two effects: the amount of work performed between calls to enqueue and dequeue is increased, and the number of times a queue operation is invoked is decreased, further reducing the contention. These two parameters can therefore be adjusted to help ameliorate any negative performance impact associated with the use of course-grained locking in the application's shared queue.

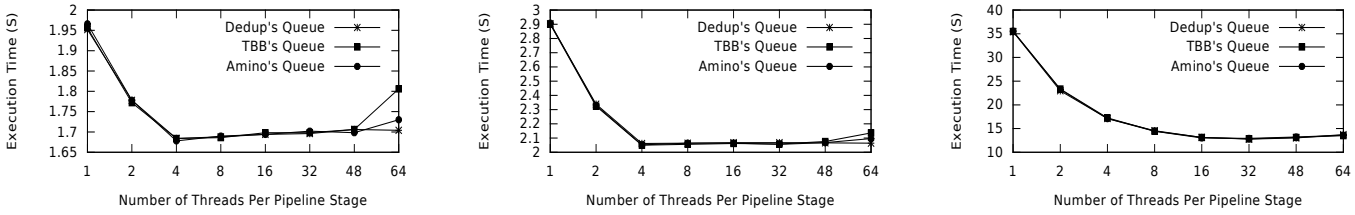
Fig. 8 shows how the application performance scales with the change of these configuration parameters when using 64 threads per pipeline stage and the *Simnative* input file. Fig. 8a confirms that as the number of threads operating on the queues increase, so does the contention among the threads, leading to a longer execution time for the whole application. Fig. 8b illustrates the importance of performing batch operations, showing a steady performance increase when more items are grouped in a single enqueue/dequeue operation, with the best performance reached when enqueuing/dequeuing 50 items at a time. By default, *Dedup* uses a batch mode of 20 items per operation, providing a performance boost of 9% compared with the non-batching mode where each operation manipulates a single item. Another observation from Fig. 8 is that the original Dedup queue has consistently outperformed the other two lock-free queues when the batch mode reaches 20 items per en/dequeue operation and when the max number of threads to share a queue is greater than 4 (the default Dedup configuration), showing better scalability than the other queue implementations.

While our evaluations in both Figs. 7 and 8 demonstrate that using the coarse-grained locking approach employed by *Dedup's* queue provides the best overall performance, we attribute this result to the fact that among the three evaluated approaches, *Dedup's* queue is the only shared queue that implements an inherent support for batching operations. As shown in Fig. 3b and discussed in Section 4.3, due to the lack of internal support for batch operations from the two alternative lock-free queues, we implemented the *Dedup* batch operations by simply invoking their single-item enqueue/dequeue operations multiple times, without taking advantage of the added concurrency.

To validate our speculation, in Fig. 9, we devised a synthetic test for the three concurrent queues by splitting all threads into two equal groups: a set of enqueuers and a set of dequeuers. Each enqueuer thread adds its share of elements, and each dequeuer attempts to dequeue until the queue is empty and the enqueuers have completed all of their work. Fig. 9a shows our results when 20 items are enqueued or dequeued at a time by each operation, and Fig. 9b shows the results when the number of elements per operation is decreased to 1.

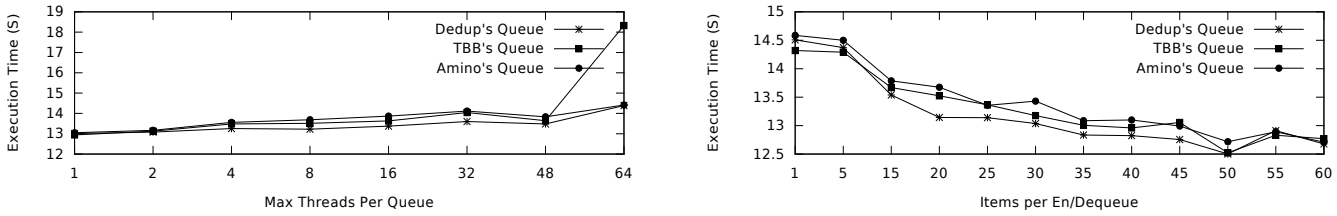
Note that the performance of the TBB and CBB's queues did not change at all in these two graphs, while the performance of Dedup's original queue degraded significantly in Fig. 9b, where Intel TBB's queue outperformed the other approaches as the original queue of *Dedup* became a victim of lock contention and reduced parallelism due to the use of mutually exclusive locks. Compared with Amino CBB's non-blocking implementation, Intel TBB's non-blocking queue provides an improved and more recent design which results in higher efficiency.

In summary, our performance analysis of the shared queues shows that due to the distribution of contention to multiple queues



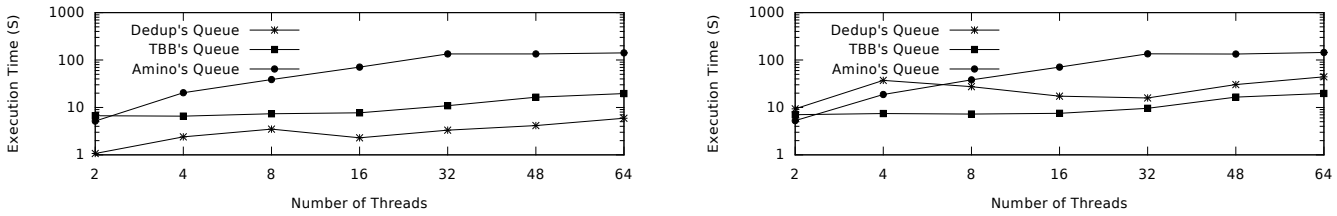
(a) Input: SimSmall (b) Input: Simmedium (c) Input: Simlarge

Figure 7: Performance of Dedup when using different queue implementations



(a) Maximum Number of Threads per Queue (b) Items Per En/Dequeue

Figure 8: Performance of different Dedup configurations. All evaluations use 64 threads per pipeline stage and the *Simnative* input file



(a) 20 Items per En/Dequeue (b) 1 Item per En/Dequeue

Figure 9: Synthetic Queue Tests

and the batch mode of enqueueing and dequeueing, the queue's effect on the overall performance of *Dedup* is minimal. Extending the functionality of Intel TBB's queue to support a batch mode of element insertion and deletion may lead to improved performance and scalability when shared queues are used in the pipeline parallel processing model.

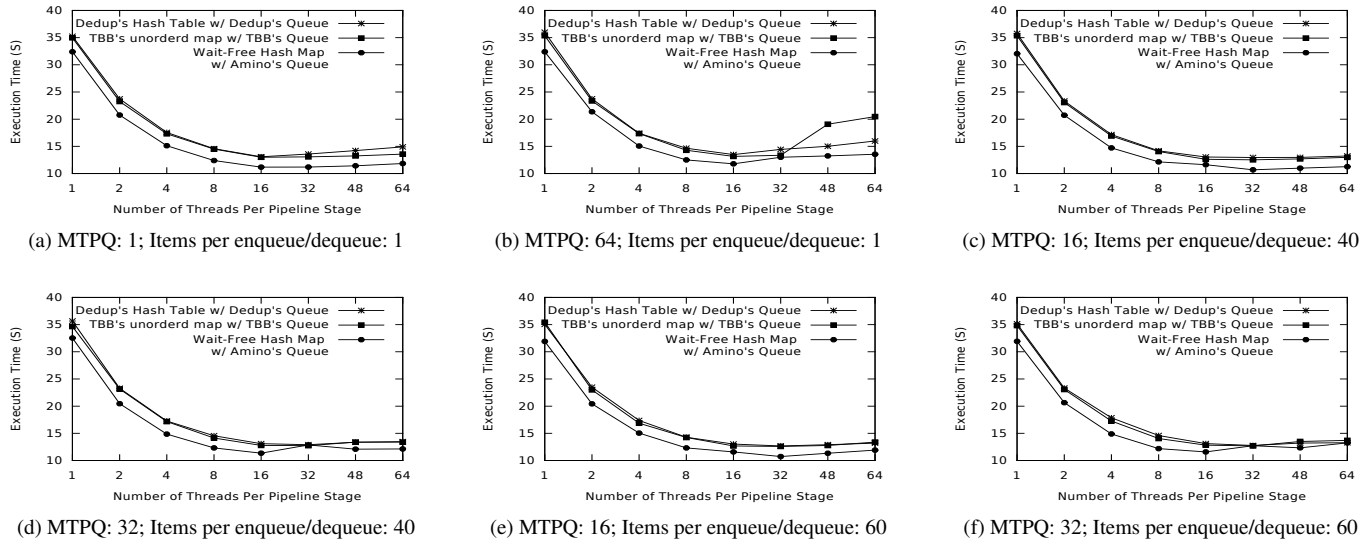
8. Application Scalability

Fig. 10 presents our results of studying how the overall performance of *Dedup* scales when using representative combinations of the hash map and queue implementations with varying execution configurations including the number of threads per pipeline stage, the maximum number of threads that can share a single queue (MTPQ), and the number of items per batch enqueue/dequeue operation. All evaluations use the largest input size, *Simnative*.

Most of the graphs in Fig. 10 are similar to that of Fig. 5c, as the performance impact of the hash map implementations dominates the minor performance differences caused by the varying queue implementations. However, when the maximum number of threads per queue (MTPQ) is increased in Figs. 10b, 10d, and 10f, we can see a clear performance degradation for all hash map and queue implementations when the number of threads per pipeline stage exceeds MTPQ, where a large number of threads are competing to access a single shared queue. In spite of the fact that the shared queue operations only comprise a negligible fraction of the overall *Dedup* execution time, they could become the overall performance bottleneck as each thread spends a significant amount of time waiting to operate on the queues. As a result, the performance benefit gained from the higher degree of concurrency by the wait-free hash map can be lost, resulting in identical performance from using all hash map implementations. While the number of items per enqueue/dequeue operation can ameliorate the performance degradation to a degree, this effect is limited when contention is severe.

9. Core Utilization

In addition to measuring the execution time of the varying implementations of *Dedup*, we tracked their utilization of each CPU core using mpstat [18]. Fig. 11 presents the core utilization patterns of three representative implementations that use the *Dedup*'s original hash table and queue (Fig. 11a), TBB's hash map and queue (Fig. 11b), and the Wait-Free hash map and the Lock-Free queue (Fig. 11c) respectively. Each graph shows the percentage of utilization of each core from the third to the fifth second of execution, where the majority of the utilization occurs. All three graphs use *simnative* as the input file, with the *max thread per queue* set to 4, *items per en/dequeue* set to 20, and with 64 threads executing per pipeline stage. All three graphs have similar patterns of core



(a) MTPQ: 1; Items per enqueue/dequeue: 1

(b) MTPQ: 64; Items per enqueue/dequeue: 1

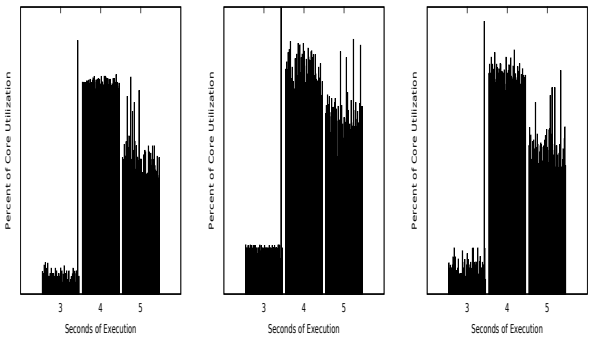
(c) MTPQ: 16; Items per enqueue/dequeue: 40

(d) MTPQ: 32; Items per enqueue/dequeue: 40

(e) MTPQ: 16; Items per enqueue/dequeue: 60

(f) MTPQ: 32; Items per enqueue/dequeue: 60

Figure 10: Scalability study of `Dedup`



(a) Original Data Structures

(b) TBB Data Structures

(c) Non blocking Data Structures

Figure 11: Core Utilization `Dedup`

utilization, with the TBB configuration having the highest average utilization, followed by the non-blocking configuration, and then by original configuration. This pattern reveals that in addition to completing computation faster, the non-blocking configuration also uses less processing power to do so.

10. Related Work

In [19], Tsigas et al. present a performance study of integrating non-blocking containers into several key applications from the benchmark suites SPLASH-2 [20] and Spark98 [21]. The authors show that the use of non-blocking containers in large-scale high performance computing applications can lead to a performance increase by a factor of 24 when compared to the standard implementations. Their findings indicate that the use of non-blocking containers does not lead to performance loss in any use scenario.

Further analysis by Tsigas et al. [19] reveals that large-scale scientific applications using non-blocking synchronization generate fewer cache misses, exhibit better load balancing, and show significantly better scalability and performance when compared to their blocking counterparts. In contrast, we focus on the in-depth exploration of a single application and the process of effective and practical concurrent data structure integration and optimization in multiprocessor software. A number of prior studies explore the practical application of non-blocking containers in the context of micro-benchmarks, such as [22] and [23].

Existing research in the design of non-blocking data structures includes linked-lists [24, 25], queues [6, 26, 27], stacks [27, 28], hash maps [25, 27, 29], hash tables [30], binary search trees [31], and vectors [32]. The designs of these new concurrent data structures typically focus on reasoning about their efficiency and practicality through synthetic tests. In contrast, our work focuses on studying the practical impact and performance tradeoffs of using these concurrent data containers within real-world applications.

11. Conclusion

This paper presents a study on integrating two general-purpose concurrent data structures, hash maps and shared queues, within a multi-threaded data deduplication application from the PARSEC benchmark suite [8]. The goal is to investigate the performance trade-offs between using conventional lock-based synchronization mechanisms vs. using lock-free and wait-free synchronizations to implement these data structures and to seek a better understanding of two important issues: 1) the transformations required to modify an existing multi-threaded application from using lock-based synchronizations to using non-blocking synchronizations; and 2) the use case scenarios where non-blocking data structures provide significant performance benefits.

Our results indicate that the application often needs specialized operations, e.g., enqueueing and dequeue elements in batches, that require the standard APIs of a general-purpose concurrent data structure to be adapted for efficiency. In the case of the concurrent queue, the degree of concurrency is minimized since all of its operations are performed on the head and tail pointers. *Dedup* resolves the concurrency issues of its local queues by supporting batch enqueue/dequeue operations and allowing multiple queues to be used per pipeline stage. As a result, the lock-free queues we integrated within *Dedup* do not perform as well as the original *Dedup* queue implementation, because they cannot be easily adapted to support

special batch enqueue and dequeue operations. Similar strategies need to be adopted within the non-blocking queue implementations to enhance the internal concurrency and thereby scalability of their applications.

The performance gains of using non-blocking data structures are often dictated by the distribution of operations on the data, the context within which the operations are invoked, and the degree of concurrency allowed within these operations. Within *Dedup*, the use of non-blocking hash maps produced significant performance gains, as the hash map allows a greater degree of concurrency by allowing operations on different hash keys to execute in parallel.

We did not encounter many problems while replacing traditional locking data structures with ones that provide concurrent APIs. The main diffculty was understanding the subtle implications of removing the locks, which often contain additional work that is not part of the data structure. For developers that seek to make legacy code non-blocking, these subtlies are understood, and the process of incorporating new data structures is straightforward.

Acknowledgment

This research is funded by the National Science Foundation under Grant Numbers CCF-1218100 and CCF-1218179.

References

[1] D. Dechev and B. Stroustrup, "Scalable Nonblocking Concurrent Objects for Mission Critical Code," in *OOPSLA '09: Proceedings of the ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2009.

[2] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008. [Online]. Available: www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0123705916

[3] D. Dechev, *A Concurrency and Time Centered Framework for Autonomous Space Systems*. LAP LAMBERT Academic Publishing, August 2010.

[4] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Lock-Free Dynamically Resizable Arrays," in *OPODIS*, ser. Lecture Notes in Computer Science, A. A. Shvartsman, Ed., vol. 4305. Springer, 2006, pp. 142–156.

[5] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, p. 5, 2007. [Online]. Available: www.ddj.com/dept/cpp/184401890

[6] M. Michael, "CAS-Based Lock-Free Algorithm for Shared Deques," in *Euro-Par 2003: The Ninth Euro-Par Conference on Parallel Processing, LNCS volume 2790*, 2003, pp. 651–660.

[7] P. Tsigas and Y. Zhang, "The non-blocking programming paradigm in large scale scientific computations," in *PPAM*, 2003, pp. 1114–1124.

[8] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[9] Intel, "Intel Threading Building Blocks," threadingbuildingblocks.org/, November 2011. [Online]. Available: threadingbuildingblocks.org/

[10] M. Moir and N. Shavit, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC Press, 2007, ch. Concurrent Data Structures, pp. 47–1–47–30.

[11] M. Herlihy, "Wait-Free Synchronization," in *Trans. on Programming Languages and Systems*. ACM, 1991, pp. 124–149.

[12] D. Dechev, "The ABA Problem in Multicore Data Structures with Collaborating Operations," in *Proceedings of the 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2011)*, 2011.

[13] S. Feldman, P. LaBorde, and D. Dechev, UCF Technical Report (cse.eecs.ucf.edu/private/UCF-TechReport-HashTable.pdf). Retrieved 04/05/2013.

[14] S. Feldman, P. LaBorde, and D. Dechev, "A Lock-Free Concurrent Hash Table Design for Effective Information Storage and Retrieval on Large Data Sets," in *Proceedings of the 15th Annual High Performance Computing Workshop (HPEC 2011)*, 2011.

[15] D. Lea, "Concurrenthashmap," gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/ConcurrentHashMap.html, May 2013.

[16] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 267–275. [Online]. Available: doi.acm.org/10.1145/248052.248106

[17] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004. [Online]. Available: www.research.ibm.com/people/m/michael/ieeetpds-2004.pdf

[18] Linux User's Manual, "mpstat," linuxcommand.org/man_pages/mpstat1.html, May 2013.

[19] P. Tsigas and Y. Zhang, "Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies," in *Proceedings of the 3rd international workshop on Software and performance*, ser. WOSP '02. New York, NY, USA: ACM, 2002, pp. 55–67. [Online]. Available: doi.acm.org/10.1145/584369.584378

[20] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *22nd International Symposium on Computer Architectures*, June 1995, pp. 24–36.

[21] D. R. O'Hallaron, "Spark98: Sparse matrix kernels for shared memory and message passing systems," Tech. Rep. CMU-CS-97-178, October 1997.

[22] B. Lim and A. Agarwal, "Reactive synchronization algorithms for multiprocessors," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994, pp. 25–35.

[23] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," in *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, 1998, pp. 1–26.

[24] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. London, UK: Springer- Verlag, 2001, pp. 300–314.

[25] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM Press, 2002, pp. 73–82.

[26] H. Sundell and P. Tsigas, "Lock-free deques and doubly linked lists," *J. Parallel Distrib. Comput.*, vol. 68, pp. 1008–1020, July 2008.

[27] Microsoft, "System.collections.concurrent namespace," Microsoft, 2011, .NET Framework 4. [Online]. Available: msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx

[28] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," *J. Parallel Distrib. Comput.*, vol. 70, pp. 1–12, January 2010.

[29] H. Gao, J. Groote, and W. Hesselink, "Almost wait-free resizable hashtables," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004, p. 50.

[30] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2003, pp. 102–111.

[31] K. Fraser, "Practical lock-freedom," in *Computer Laboratory, Cambridge Univ*, 2004.

[32] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Lock-free dynamically resizable arrays," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, M. Shvartsman, Ed. Springer Berlin / Heidelberg, 2006, vol. 4305, pp. 142–156. [Online]. Available: dx.doi.org/10.1007/11945529_11