SANDIA REPORT
SAND2012-2093
Unlimited Release
Printed March 2012

Effective Software Design and Development for the New Graph Architecture HPC Machines

Damian Dechev

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND 2012-2093
March 2012
Unlimited Release

Effective Software Design and Development for the New Graph Architecture HPC Machines

Damian Dechev
Scalable and Secure Software R&D Department,
Sandia National Laboratories
P.O. Box 969, MS-9158
Livermore, CA 94551-0969
ddechev@sandia.gov

March 17, 2012

Abstract

Software applications need to change and adapt as modern architectures evolve. Nowadays advancement in chip design translates to increased parallelism. Exploiting such parallelism is a major challenge in modern software engineering. Multicore processors are about to introduce a significant change in the way we design and use fundamental data structures. In this work we describe the design and programming principles of a software library of *highly concurrent scalable and nonblocking* data containers. In this project we have created algorithms and data structures for handling fundamental computations in massively multithreaded contexts, and we have incorporated these into a usable library with familiar look and feel.

In this work we demonstrate the first design and implementation of a wait-free hash table. Our multiprocessor data structure design allows a large number of threads to concurrently insert, remove, and retrieve information. Non-blocking designs alleviate the problems traditionally associated with the use of mutual exclusion, such as bottlenecks and thread-safety. Lock-freedom provides the ability to share data without some of the drawbacks associated with locks, however, these designs remain susceptible to starvation. Furthermore, wait-freedom provides all of the benefits of lock-free synchronization with the added assurance that every thread makes progress in a finite number of steps. This implies deadlock-freedom, livelock-freedom, starvation-freedom, freedom from priority inversion, and thread-safety.

The challenges of providing the desirable progress and correctness guarantees of wait-free objects makes their design and implementation difficult. There are few wait-free data structures described in the literature. Using only standard atomic operations provided by the hardware, our design is portable; therefore, it is applicable to a variety of data-intensive applications including the domains of embedded systems and supercomputers.Our experimental evaluation shows that our hash table design outperforms the most advanced locking solution, provided by Intel's TBB library, by 22%. When compared to more traditional locking designs we show a performance improvement by a factor of 7.92. When compared to alternative non-blocking designs, our hash table demonstrates solid performance gains in a large majority of cases, typically by a factor of 3.44.

Acknowledgements

The author thanks Curtis Janssen, Keith Vanderveen, Jon Berry, Gilbert Hendry, and Joe Kenney for their helpful feedback and support for this work.

Contents

List of Figures

List of Tables

List of Publications

Below we provide a brief list of publications and poster presentations that are a part of this work.

Publications in Conference Proceedings:

- Andres Vidal, Damian Dechev, Alain Kassab, *Evaluation of two acceleration techniques in a multithreaded 2D Poisson equation solver*, In Proceedings of the International Conference on Computational Science (ICCS 2012), Omaha, Nebraska, June 4-6, 2012 (To Appear). SAND Number: 2012-1797C

- Steven Braeger, Nicholas Arnold, Damian Dechev, *Scalable N-Body Event Prediction*, Central European Journal of Computer Science, Springer-Verlag, March, 2012. SAND Number: 2012-1460 J

- Andres Vidal, Alain Kassab, Daniel Mota, Damian Dechev, *A Multithreaded Solver for the 2D Poisson Equation*, In Proceedings of the 20th ACM/SIGSIM High Performance Computing Symposium (HPC 2011), Orlando, FL, March 26-29, 2012 (To Appear). SAND Number: 2012-1461 C

- Shiyuan Jin, Damian Dechev, Zhihua Qu, *Parallel Particle Swarm Optimization (PPSO) on the Coverage Problem in Pursuit-Evasion Games*, In Proceedings of the 20th ACM/SIGSIM High Performance Computing Symposium (HPC 2011), Orlando, FL, March 26-29, 2012 (To Appear). SAND Number: 2012-1459 C

- Damian 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), Orlando, FL, October 15-18, 2011. *(Presenter)*. SAND Number: 2012-1525 C

- Tae-Hyuk Ahn, Damian Dechev, Heshan Lin, Helgi Adalsteinsson, Curtis Janssen, *Evaluating Performance Optimizations of Large-Scale Genomic Sequence Search Applications Using SST/macro*, In Proceedings of the 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications, SIMULTECH 2011, Noordwijkerhout, The Netherlands, July 29 - 31, 2011. Acceptance Rate: 25/141=17.7%. *(Presenter)*. SAND Number: 2011-3682C

- Peter Pirkelbauer, Damian Dechev, Bjarne Stroustrup, *Support for the Evolution of C++ Generic Functions*, In Proceedings of 3rd International Conference on Software Language Engineering (SLE 2010), Eindhoven, The Netherlands, October 2010. Acceptance Rate: 17/62=27.4%. SAND Number: 2011-0081C

- Damian Dechev, Peter Pirkelbauer, Bjarne Stroustrup, *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*, In Proceedings of 13th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (IEEE ISORC 2010), Carmona, Spain, May 2010. *(Presenter)*. SAND Number: 2010-0971C

Workshop and Poster Participation:

- Steven Feldman, Emile Kanhai, Damian Dechev, *Lock-Free Concurrent Hash Tables*, In Science Magazine, Special Feature: International Science & Engineering Visualization Challenge 2011, February, 2012.

- Amruth Dakshinamurthy, Curtis Janssen, Damian Dechev, *A Compiler-based Framework for Automatic Extraction of Software Models for Exascale Hardware/Software Co-Design*, In Proceedings of the Fifteenth Annual High Performance Embedded Computing Workshop (HPEC 2011), MIT Lincoln Laboratory, Lexington, MA, September 21 - 22, 2011. *(Presenter)*. SAND Number: 2012-1523 C

- Steven Feldman, Pierre LaBorde, Damian Dechev, *A Lock-Free Concurrent Hash Table Design for Effective Information Storage and Retrieval on Large Data Sets*, In Proceedings of the Fifteenth Annual High Performance Embedded Computing Workshop (HPEC 2011), MIT Lincoln Laboratory, Lexington, MA, September 21 - 22, 2011. SAND Number: 2012-1524 C

- Steven Feldman, Pierre LaBorde, Damian Dechev, *Facilitating Efficient Parallelization of Information Storage and Retrieval on Large Data Sets*, In Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011), Tucson, AZ, May 31- June 4, 2011. SAND Number: 2011-3222C

- Amruth Dakshinamurthy, Damian Dechev, *Automatic Extraction of SST/macro Skeleton Models*, In Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011), Tucson, AZ, May 31 - June 4, 2011. *(ACM Student Research Competition Finalist)*. SAND Number: 2011-3224C

1 Introduction

Advances in chip design have brought multi-threaded parallel programming to mainstream computing and have made exploiting such parallelism a major challenge in modern software engineering. In [24], Gartner identifies “Parallel Programming” and “Increase Programmer Productivity 100-fold” as two of the seven “Grand Challenges Facing IT” by examining the fundamental issues and technologies whose resolutions “will have a broad and extremely beneficial economic, scientific or societal effects on all aspects of our lives”. The ever expanding heterogeneity of modern architectures are demanding developers to effectively manage a growing variety of available resources such as high degree of parallelism, single-chip multiprocessors, and the deep hierarchy of shared/distributed cache and memories, Nowadays parallelism is offered in a variety of ways, including PC clusters, distributed multiprocessors, SMP, multi-core, GPU, vector units, among others. Each machine typically features a deep hierarchy of shared/distributed caches and memories, and good data locality is required to achieve computation efficiency. The ever expanding heterogeneity of modern architectures demands applications to quickly migrate to a variety of architectures while maintaining scalability to an increasingly larger number of nodes. Both small-scale multicores and future exascale supercomputers have to employ new methodologies for effective access to shared memory and the routing of data. The article suggests [24] that “the main challenge with parallel computing is to create applications that fully exploit a multicore architecture” and also that such an enormous increase in productivity can be achieved by “minimizing the time required to find the perfect software module and avoiding the need to modify reusable software.” The increasing variety of parallel architectures has led to an even greater degree of highly specialized codes. The development of such codes requires deep expertise in multithreading and parallel computing and a broad knowledge of the hardware architecture. A critical element of our effective use of such new and future multi-processor architectures is the design and application of highly concurrent multi-processor data structures [47].

Programming shared-memory multiprocessors (multicores) is notoriously hard because of the unpredictable asynchronous nature of modern computer systems. Pingali et al. [42] and Shavit [47] argued that many modern mainstream applications exhibit complex, irregular, and rapidly changing communication patterns and interactions. Such parallelism is substantially different than the traditional parallel applications known in scientific computing with slow changing and regular synchronization patterns. In spite of the critical role in emerging applications of such irregular parallel algorithms, currently we have few insights of how to effectively and safely parallelize such irregular algorithms demanding irregular and rapid data exchanges and sharing of large volumes of data among multiple threads. The irregular computation and synchronization patterns of the emerging parallel applications and the increasing variety of the modern hardware architectures pose an enormous challenge in constructing highly reusable and portable multi-processor software libraries. For instance, large graph problems that arise in complex network analysis, data mining, computational biology, enabling predictive discrete event simulation, and other critical applications have sparked the development of new types of high-performance architectures and codes [2, 29]. These codes involve very large numbers of threads and much greater sharing of information among processors than traditional scientific computing approaches, posing much greater concurrency challenges. It is difficult to effectively and safely parallelize such irregular algorithms which demand irregular and rapid data exchanges and sharing of large volumes of data among multiple threads.

To accommodate the paradigm shift towards concurrency and heterogeneity, a plethora of new programming models are being developed and tested to support the development of concurrent multi-threaded applications on multi-core architectures. However, most of the existing concurrent programming models are based on mutual exclusion, which is the most common synchronization technique for shared data but can lead to significant overhead, high complexity, and reduced parallelism and key safety hazards including race conditions, deadlock, livelock, and ordering violations [22, 12]. Such errors are hard to reproduce, often lead to unpredictable real-time behavior and are difficult to debug. According to a study at NASA

Ames [34, 43], the current development and validation techniques are prohibitive for problems of such complexity. Programming models and concurrent data structures that are easier to use and capable of scaling to arbitrary numbers of threads are in critical needs to ameliorate the problems.

Applications and algorithms need to change and adapt as modern architectures evolve. Such adaptations have become increasingly difficult for developers as they are required to effectively manage an ever-growing variety of resources such as a high degree of parallelism, single-chip multi-processors, and the deep hierarchy of shared and/or distributed caches and memories. Developers writing concurrent code face challenges not known in sequential programming: notably, to correctly manipulate shared data. A critical element for effective use of such new and future multi-processor architectures is the design and application of highly-concurrent, multi-processor data structures [48]. ISO C++ [50] is widely used for parallel and multi-threaded software. The recently released C++ standard, C++11 [28], simplifies concurrent programming by including a new memory model and standard library support for threading and atomicity. However, C++11 and the majority of the commonly used programming languages still do not provide standard multi-processor data structures and algorithms.

1.1 Motivation

Currently, the most common synchronization technique is the use of mutual exclusion locks. This can seriously affect the performance of the system by diminishing its parallelism [22]. The behavior of mutual exclusion locks can sometimes be optimized by using a fine-grained locking scheme [27], [40] or context-switching. However, the interdependence of processes implied by the use of locks, even efficient locks, introduces the dangers of deadlock, livelock, starvation, and priority inversion. Wait-freedom implies freedom from such hazards [22]. However, all of these appealing properties of wait-freedom are difficult to implement in practice as they require that no thread can block another, in such a way that the other thread is unable to make progress in an amount of time that is guaranteed to be finite. Guaranteeing this property is especially difficult as it requires reasoning about all of the possible ways that all of the possible operations may interleave, in a concurrent environment. In many cases, the problem with locks is one of difficulty of providing correctness more than one of performance [13]. As a result, there are not many wait-free algorithms in the literature. In fact, there are no other wait-free hash tables at all. Though there is one hash table that comes close, it is wait-free until it must resize [18], and there are several lock-free hash tables implemented as well [22], [37], [46].

1.2 Our Approach

Our work delivers practical design, implementation, and support tools for highly concurrent and reusable multi-processor data structures. In particular, we have developed a collection of highly concurrent scalable data containers together with an associated compiler to support *nonblocking synchronization of shared memory operations* [22, 47, 12], a recent synchronization mechanism known to provide better safety and performance than traditional blocking synchronization techniques by eliminating hazards such as deadlock, livelock, and priority inversion and by being highly scalable in supporting large numbers of threads. The entire system integrates an extensive set of architecture-specific optimizations, e.g. reduction of synchronization overhead and enhancement of data locality, and parameterizes their configurations so that they can be automatically tuned when porting together with our runtime system to a variety of multicore platforms. In addition, our research integrates a mature platform for developers to study and analyze their application performance at scale by integrating the Structured Simulation Toolkit's macroscale simulation components (SST/macro) [2, 44]. Through our collaborations with the developers of SST/macro, our team employs a highly functional architecture simulator with which we are deeply familiar, and our framework uses the internally integrated SST/macro to interactively assist developers in attaining our objectives. The goal is to create a portable generic software platform and building blocks to greatly assist domain scientists and engineers who are not experts in concurrency theory and formal methods to produce highly efficient and correct code.

When building large-scale software applications developers apply the core problem-solving approach as when constructing other large-scale artifacts – namely the processes of *decomposition* and *abstraction* [31]. Library-centric software development has become increasingly important in modern software design [49, 28]. Through this work we deliver high-quality and reliable building blocks for multi-processor programming and an easy and effective composition process. Our research creates algorithms and data structures for handling fundamental computations in multithreaded contexts and incorporates these into a practical and usable library with a familiar look and feel. The main challenge is in the design and implementation of the librarys data structures and our support environment. That way the burden of multicore algorithms design and optimization is placed on the library developers (the PI of this work and our team) and not on the users of our collections of data containers. Thus, our nonblocking library hides the complexity of parallel programming to the majority of its users. Many modern parallel programming codes contain a large number of architecture-sensitive programming techniques and optimizations that are tailored to specific problem domains or hardware platforms. Often, to achieve a high degree of efficiency, developers of such concurrent codes need to re-engineer the very fundamental data structures and algorithms known in computer science such as queues, stacks, hash tables, linked-lists, and vectors [47, 41, 42]. Our approach helps deliver easy, effective, and scalable multi-processor programming in C/C++ and allows for a dramatic increase in the reuse and portability of complex multithreaded software components.

2 Designing the Scalable Template Library

Processes in shared memory multiprocessor systems communicate with each other by executing atomic reads and writes to various shared data. The result of the concurrent execution of a number of operations is dependent on the processes interleaving. Thus, various synchronization mechanisms are necessary to ensure that the desired interleaving of operations is enforced at all times and the program semantics are preserved. The most common technique for synchronizing concurrent processes is the use of a mutual exclusion locks [22], which guarantees thread-safety of a concurrent objects by blocking all contending threads except the one holding the lock. Even for conventional multicore systems of modest size, the use of blocking reduces parallelism and can lead to convoying effects that can seriously detriment application performance [17]. Further, incorrect use of locks is hard to detect using traditional testing procedures, whereas a program can be deployed and used for a long period of time before the flaws become evident and cause anomalous behavior [10, 12]. A key component of our research is to develop a library of composable and highly scalable generic concurrent data structures to support *nonblocking synchronization* [22, 47, 17, 12], which enhances thread safety and enables fast execution by eliminating hazards associated with mutual exclusion locks.

Our multi-processor scalable template library provides concurrent data structures with composable, generic and linearizable C++ interfaces similar to the C++ Standard Template Library [28]. Nonblocking synchronization is a recent alternative to mutual exclusion that allows the implementation of highly scalable data objects for multithreaded systems and at the same time eliminates entire classes of safety hazards associated with the application of locks. A concurrent object is *lock-free* [22] if it guarantees that *some* process will always make progress in a *finite* number of steps. It is additionally *wait-free* if *all* processes are guaranteed progress. Lock-free and wait-free algorithms do not use mutual exclusion locks and can efficiently utilize resources on shared memory multiprocessors with low latency and high bandwidth of interprocessor communication [22]. Our existing work shows that it can eliminate whole classes of concurrency hazards while delivering significant performance improvements by a large factor for many scenarios [12, 10, 8, 15, 7, 11].

A study by Tsigas et al. [55] demonstrates that large scale scientific applications using nonblocking synchronization generate fewer cache misses, exhibit better load balancing, and show significantly better scalability and performance when compared to their counterparts using locks.

ISO C++ [50] is widely used for parallel and multi-threaded software. The recently released C++ standard, C++11 [28], includes a new memory model and standard library support for threading and atomicity. As stated by Stroustrup [51], the designer and inventor of C++, C++11 includes "lock-free programming facilities for people who want to provide something radically different". However, C++11 and the majority of the commonly used programming languages still do not provide standard multi-processor data structures and algorithms. We will design and implement a collection of nonblocking data containers for C/C++ with interfaces that are easy to use and composable, with enhanced safety by eliminating the hazards of deadlock, livelock, and priority inversion, and with significant performance gains for a large number of multicore applications. Our library delivers fundamental concurrent data structures such as stacks, queues, dequeues, linked-lists, dictionaries, dynamic arrays, and hash tables, starting with two critical containers, a hash table and a priority queue, for modern data-intensive applications. Each container includes components needed to easily compose them as nonblocking building blocks, such as iterators, lock-free memory management, and memory allocation schemes. We extend the generic object-oriented ISO C++ STL interface and evaluate our approach by using our library within a number of real-world concurrent codes and a number of benchmark suites including the Mentovo suite [45] for scientific codes.

While there is an increasing demand for nonblocking algorithms, currently we lack a collection of portable and generic C/C++ nonblocking objects that can easily be applied to existing code. In this work we develop the theory, techniques, and building blocks that can be used by domain scientists and engineers who are not experts in parallel programming and nonblocking synchronization to compose efficient, safe, and scalable codes. Composition is a natural way to construct and reason about large and complex systems. We develop compositional strategies for building scalable nonblocking applications and for optimizing such application and their use of system resources. To overcome the drawbacks associated with the application of blocking synchronization and deliver high scalability and performance, we focus on the application of nonblocking synchronization techniques [22, 47, 12, 7].

3 Implementation Challenges

Two key challenges exist in designing scalable concurrent data structures: 1) the asynchronous nature of multi-threaded computation, where the correctness of concurrency is hard to guarantee as threads execution can be interleaved in an arbitrary fashion, and their progresses and completion are sensitive to variations of operating system scheduling decisions, interrupts, and page faults; and 2) complexity of modern hardware platforms, where the interconnect among memory and processors, the layout of data in memory, and the dynamic communication patterns [22, 2] among threads make the performance of synchronous operations hard to predict. The correctness and performance issues are directly related [47], as the more optimizations applied to improve a parallel algorithm's performance, the harder it becomes to reason about this algorithm's correctness. The difficulties in guaranteeing correctness, which is established by reasoning about *safety and liveness*, while attaining high performance also result in several other challenges in developing a library of composable and reusable nonblocking data containers, discussed in the following.

Safety guarantee: which establishes that no possible threads interleaving can corrupt the semantics of a concurrent object, i.e. "nothing bad will happen" [22]. Guaranteeing safety in a concurrent algorithm is challenging because of the many possible scenarios in which threads can interleave, which require rules for mapping concurrent executions to sequential ones. One such set of rules has been described by Lamport [32] in his sequential consistency model, which requires that the results of a concurrent execution are equivalent to those yielded by *some* sequential execution. A more strict version of sequential consistency is linearizability [22], where a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time, called *linearization point*, between the time of operation invocation and the time of its completion. In effect, linearizability additionally requires that the total ordering in the derived sequential history respects the *real-time ordering* of operations of the concurrent execution. According to real-time ordering, an operation o_1 is said to precede an operation o_2 if o_2's invocation occurs after o_1's response. Operations that do not have real-time ordering are defined as concurrent. To allow for greater reusability of our multi-processor library components, **we aim to support linearizability as our safety guarantee.**

Liveness guarantee: which establishes that eventually progress will be made in the system, i.e. "something good will happen" [22]. Defining a *liveness guarantee* means specifying its progress condition such as lock-freedom, wait-freedom, deadlock-freedom, and starvation-freedom [22]. For instance, a *wait-free* operation is independent and can complete without requiring support from the scheduler, while a *deadlock-free* operation guarantees concurrent operations will not enter a state of deadlock but requires strong support from the scheduler [47]. **We aim to support wait-freedom and lock-freedom as our liveness properties.**

Composability While well-established, the sequential consistency model implies that sequentially consistent objects are not composable. This means that a data object implemented using sequentially consistent components may not be sequentially consistent. Such lack of composability is a significant disadvantage when building concurrent libraries and seriously inhibits the reusability of parallel software. By enforcing the more strict linearizability consistency model, our library will guarantee composability as well.

Scalability By Amdahl's Law [22], the speedup of a parallel algorithm is the ratio of a program's execution time on a single processor to its execution time on *P* processors. A concurrent data structure is *scalable* if its observed speedup increases with *P*. In practice, as concurrency grows it becomes increasingly hard to obtain scalable behavior. This is illustrated by a simple application of Amdahl's Law: using 10 processors we achieve only a twofold speedup with 60% concurrent execution time and five fold speedup with 90% concurrent execution time. If we increase the processors we use up to 20, we will get a sevenfold speedup. Therefore, regardless of the processing power of our multicore machine and the increasing number of cores we use, our speedup will be significantly limited by only a 10% sequential execution of our program. For a majority of multi-threaded applications, the portions of code that are hard to parallelize are those implementing multi-threaded synchronizations and communications [47, 29, 6, 2, 44]. On a multicore platform with shared memory, these synchronization mechanism coordinate the access to the shared data structures. Often applying common synchronization mechanisms that work well with low concurrency can severely limit the possibility of scalable behavior. In particular, the use of mutual exclusion is known to introduce *sequential bottlenecks and inhibit scaling*. Further, the completion of concurrent algorithms can be significantly impacted by variations of the scheduler's decisions, making it hard to provide guarantees of progress, e.g., if the scheduler delays the thread holding the lock, all other threads waiting for this lock will be blocked. Another potential performance bottleneck is *memory contention* caused by all contending threads attempting to access a single shared memory location [22], which causes prolonged waiting time and heavy memory traffic on platforms employing typical cache coherence protocols. Our research will address all the above synchronization problems through efficient implementations of wait-free and lock-free synchronous operations.

4 Implementation Approach

Our library is centered around a collection of practical highly concurrent *nonblocking* data containers with interfaces that are composable and easy to use, with enhanced safety by eliminating the hazards of deadlock, livelock, and priority inversion, and with portable performance gains for a large number of multicore applications. The research is based on the following technical approaches.

Safety and progress guarantees: We focus on providing data structures that are **linearizable** [22], where concurrent operations appear to execute instantaneously in any given point of time, and **wait-free** [22], where a concurrent object guarantees that *all* processes will make progress in a *finite* number of steps. For algorithms that do not allow for an effective wait-free implementation, we will guarantee they are *lock-free*, where at least one process will always make some progress in a *finite* number of steps.

Linearization is a more restrictive form of the sequential consistency model [32] and will allow concurrent operations supported by our library to be easily composed while maintaining their sequential consistency. Wait-freedom is most desirable as it guarantees *system-wide throughput and starvation-freedom* in a finite number of steps. This property significantly reduces the performance problems of parallel algorithms by avoiding sequential bottlenecks and memory contentions associated with the traditional blocking (lock-based) synchronization mechanisms and some naive lock-free approaches. Theoretically, it has been shown that all algorithms can be implemented in a wait-free manner [22]. The authors in [22] demonstrate a *universal construction* that allows for the wait-free implementation of concurrent objects. However, the use of a universal construction leads to the memory cost growing linearly with the number of threads. In practice, effective implementations of wait-free algorithms require the use of programming techniques similar to their lock-free counterparts combined with a methodology for bounding the steps each operation can take.

Implementation of wait-free and lock-free algorithms: for the implementation of our nonblocking data structures we plan to apply the following commonly used programming techniques.

- Completion of operations through fine-grained optimistic speculation: lock-free and wait-free algorithms do not use locks but instead rely on a set of read-modify-write atomic primitives such as the word-size CAS instruction [26]. Lock-free algorithms most commonly utilize the compare-and-set instruction to implement a speculative manipulation of a shared object. Each contending process applies a set of writes on a local copy of the shared data and attempts to exchange a shared object with the updated copy by executing a CAS operation. This speculative execution guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps.

- Assisting aborted threads: lock-free algorithms often employ a methodology for assisting obstructed threads such as the use of descriptor objects [12, 17]. A significant challenge in designing lock-free algorithms is implementing a correct assistance methodology. Assisting other operations is the most error-prone and costly part of a lock-free algorithm [12].

- Type and state value marking: wait-free and lock-free implementations often use a pointer bit marking technique to atomically exchange a pointer and its state using only a single-word CAS operation. The pointer bit marking technique exploits the last two bits of a pointer value, which are unused in a pointer representation, to store up to three additional value states.

- Duplicate memory copy: certain lock-free algorithms use a descriptive log storing a record of all pending reads and writes and a duplicate memory copy used to perform speculative updates that are invisible to all other threads until the linearization point of the entire transaction.

Given the above programming techniques, a typical lock-free algorithm executes in four phases [22]: 1) completion of its own operation; 2) assisting an obstructed operation; 3) aborting an obstructed operation; and 4) waiting. In a naive lock-free implementation, the repeated attempts of contending threads executing a CAS on the same memory location would result in memory contention. Advanced lock-free algorithms use a contention management algorithm to decide when to assist, abort, or wait.

5 Design and Implementation of a Wait-free Hash Table

In this section we describe the design and implementation of the *first, bounded, wait-free*, extendible, concurrent hash table C++ design. The main goals of our design are to deliver *scalability*, *safety*, and *high performance* for multi-processor applications. A hash table [5] is a data container that uses a hash function to map a set of identifying values, known as keys, to their associated values. The standard interface of a hash table consists of insert, delete, and search, each with an average time-complexity of $O(1)$. The new C++ Standard Template Library [28] provides the `unordered_map` class as a standard sequential hash table implementation. The known lock-based hash table designs inherit the disadvantages of blocking synchronization as discussed earlier [13]. The negative impact of blocking synchronization is increased during the process of the redistribution of the elements in a hash table, also known as a global resize, that occurs when adding new buckets. Thus, the problem of engineering practical and efficient non-blocking hash table designs has been of both practical and theoretical interest [37], [46], [22]. Our wait-free implementation avoids global resizes through new array allocation which is bounded by the number of indirections (pointers to chase). The presented design includes dynamic hashing, which means that each element has a unique final, as well as current, position. The code representing the implementation we discuss here is open source and we intend to make it freely available.

Our multi-processor hash table design offers the following contributions:

(a) *Wait-freedom*: we describe the first bounded wait-free hash table design. Our data structure guarantees that all threads complete their operations in a finite number of steps. This implies that we have a non-blocking execution.

(b) *Non-blocking*: we ensure that no thread can prevent another from accessing information. By providing the non-blocking progress guarantee for our operations, the presented design avoids the typical safety hazards associated with blocking synchronization.

(c) *Perfect hashing*: each element has a unique final position in the table, and this position allows the algorithm to insert, find, and delete elements concurrently, in constant time.

(d) *Extendible hashing*: we treat the hash as a bit string and rehash incrementally [14].

(e) *Portability*: because we only rely on atomic operations available on most modern architectures (such as atomic read, atomic write, and Compare-And-Swap or simply CAS [22]), we ensure that our implementation can be used on a wide range of multi-processor architectures.

(f) *Thread-death safety*: if a thread suddenly dies, regardless of the point during its execution, no data is lost; except, at most, the thread's own operation.

(g) *High performance*: our wait-free hash table design outperforms, by *22%*, the state of the art blocking designs such as the concurrent hash table implementation in Intel's Threading Building Blocks library. Furthermore, we outperform traditional blocking designs by a factor of *7.92*. When compared to alternative non-blocking solutions, our hash table provides consistent performance gains (by a factor of *3.44*).

(h) *High scalability*: our implementation demonstrates *scalable behavior*, and our performance gains are even larger compared to the alternative approaches in scenarios with more contention — greater numbers of threads competing for shared resources.

(i) *Safety*: our design goals help us achieve a high degree of safety, and our design avoids the hazards of lock-based designs: deadlock, livelock, priority inversion, and starvation.

The rest of this work is organized as follows: Section 5.1 briefly introduces the fundamental concepts of non-blocking synchronization, in Section 5.2 we discuss related work, Section 5.3 presents the algorithms of our wait-free hash table design, Section 5.8 offers a discussion of our performance evaluation, Section 5.9 provides an overview of the practical impact of our work, and in Section 6 we conclude and discuss our future work.

5.1 Background

As defined by Herlihy et al. [22], a concurrent object is *lock-free* if it guarantees that *some* process in the system makes progress in a *finite* number of steps. An object that guarantees that *each* process makes progress in a *finite* number of steps is defined as *wait-free* [22]. Non-blocking (lock-free and wait-free) algorithms do not apply mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. By definition, a lock-free[1], concurrent data structure guarantees that when multiple threads operate simultaneously on it, *some* thread completes its task in a *finite* number of steps despite failures and waits experienced by other threads. By applying atomic primitives such as CAS, non-blocking algorithms implement a number of techniques such as optimistic speculation and thread collaboration to provide for their strict progress guarantees.

The practical implementation of lock-free containers is known to be difficult: in addition to addressing the hazards of race conditions, the developer must also use non-blocking memory management and memory allocation schemes [21].

Recent research into the design of lock-free data structures includes: linked-lists [19], [37]; queues [36], [53], [39]; stacks [20], [39]; hash tables [37], [39], [18], [46]; binary search trees [16], and vectors [9].

The problems encountered include low parallelism, excessive copying, inefficiency, and high overhead. The use of hash tables is widespread in real-world applications, however, the problem of the design and implementation of a wait-free hash table is difficult — there have been no previous implementations. Designing lock-free containers is hard and achieving wait-freedom is an enormous challenge.

5.2 Related Work

A possible solution, that delivers scalability and high performance for modern multi-processor programming, is a lock-free data structure; and the only design that could be better is one that is also wait-free, such as the one presented in this paper. There are no pre-existing wait-free hash tables in the literature, as such, the related work that we discuss consists entirely of lock-free designs.

In order to achieve our design goals, we are developing an innovative approach to the structure of the hash table that uses an extendible hashing scheme that is similar to the work of Shalev et. al. in the paper titled “Split-Ordered Lists: Lock-Free Extensible Hash Tables” [46]. The key differences between their design and our work are that Shalev et. al. do not implement a hash table structure, and their design is not wait-free. Another approach for implementing a non-blocking hash table is presented by Gao et. al [18]. They implement a solution that is wait-free in the common case, but degrades in performance to lock-free when a table resize is in progress. Michael [37] describes a lock-free hash table design that uses linked lists for collision-resolution; whereas, we use arrays — to avoid the overhead of next pointers. There is only one claim of a wait-free hash table implementation that the authors are aware of [4]. This hash table design has been described in the author’s blog discussions and has not been published in the academic literature. Click’s design is presented as a Java-based implementation and allows stacked, global resizes which are undesirable as they are known to have a negative performance impact. Furthermore, the intricacies involving the C/C++ implementation of a wait-free hash table are avoided in [4].

[1] Lock-freedom is a progress guarantee (as defined in [22])

5.3 Algorithms

In this section we define a semantic model of the hash table's operations, outline a correctness proof based on that model, address concerns related to memory management, and provide a description of the design and the applied implementation techniques. The presented algorithms have been implemented, in both ISO C and ISO C++, and designed for execution on an ordinary, multi-threaded, shared-memory system — supporting only single- word read, write, and CAS instructions.

5.3.1 Implementation Overview

The hash function that we use is a one-to-one hash, where each key produces a unique hash value. Our hash function reorders the bits in the key in order to promote a more even distribution of elements. The user may specify his own hash function as long as the number of bits in the key is equal to the number of bits in the hash. The length of the memory array used in the table is defined as a power of two. By taking the first N bits of the hashed key, where N is equal to the binary logarithm of the length of the memory array, we determine the location, on the main array, to place the key-value pair. If that position is another memory array, then we shift another N bits to determine the position on that next array. The hash table structure is composed of nested arrays, where a position in one points to another array. However, the total number of arrays is bounded by the key length and the size of each array. When we refer to *indirections* or *depth* we mean the number of entries one might need to check before a key is found. Note, this places no limit on the total number of elements that can be stored in the hash table, the table can expand to hold all unique keys. However, the number of keys is limited by the bits in the key.

The maximum number of indirections — the `maxDepth` — is equal to the number of bits in the key divided by X, where X is the number of bits taken from the key, as described earlier. For example, a 32-bit key with an array of length 64 would have at most five indirections. If users have a large data set, they should choose a much larger length for the memory array, which further decreases the number of indirections caused by going from memory array to memory array. The size of the main memory array should be equal to the expected number of data elements that need to be stored — the data structure handles any further expansion as necessary. If the position that a key hashes to is occupied, then a new memory array must be created. When the new memory array is added to the table, then the current element is moved into it. By allowing concurrent table expansion this structure is *free from the overhead of an explicit resize* that involves copying the whole table, thus facilitating concurrent operations. Moreover, some related algorithms, such as the implementations of Click [4] and that of Gao et. al [18], allow stacked resizes, which can lead to starvation of one or more threads. By allowing concurrent table expansion our data structure is free from global and/or stacked resizes. These resizes lead to performance loss due to contention as well as inefficient use of memory — due to multiple sparsely-used, old tables.

5.3.2 Operations

This section provides a detailed description of the implementation of our hash table's operations.

Definitions:

We begin by presenting a number of definitions that help us explain the key algorithms in further details.

(a) *Key*: a unique identifying value for a piece of data. In this paper we assume a 32-bit integer key — we have also tested with multiword keys, such as the 20 bytes needed for SHA1. The hashed key is expressed as a continuous list of 6-bit sequences e.g. $A-B-C-D$, where A is the first 6-bit sequence, B is the next 6-bit sequence, and so on — these represent positions at various depths. These bits sequences are isolated using logical shifts.

(b) *Value*: the information associated with a key.

(c) dataNode: holds the hashed value of a key and the value that is associated with that key.

(d) *Memory array length*: the constant length of the memory arrays.

(e) *MaxDepth*: is the maximum number of *spineNodes* that a thread must traverse before it reaches a position where no collisions can occur. The `maxDepth` is equal to the ceiling of the length of the key in bits divided by spine pow.

(f) spineNode: a memory array of fixed length, where the length is a power of two. The binary logarithm of the memory array length is used to determine how many bits are necessary to determine the location to place a *dataNode* in the *spineNode*. For example, in a memory array of length $64 = 2^6$, we would take 6 bits for each successive *spineNode*. A *spineNode* is recognized by a bit mark on the pointer to it.

(g) *Memory array depth*: the number of levels of spines to traverse where *A* is at level one, *B* is at level two, *C* is at level three, and *D* is at level four; where *A*, *B*, *C*, and *D* refer to the same concept defined above.

(h) *Independent operations*: operations performed on different memory arrays are considered independent, because no operation on one memory array affects an operation on any other memory array.

(i) *Watching/Unwatching*: we use a `watchList` that allows us to determine if it is safe to reuse *dataNodes* that were removed from the hash table. The act of watching a hashed key is assigning to a threads *watchValue* the hashed key that thread is operating on. The act of unwatching a hashed key is assigning *null* to a threads *watchValue*. The importance of watching and unwatching hashed keys is described in Section 5.3.9 where we discuss our memory management approach in more details.

(j) *key match*: a key is a match to another key when both keys contain the same bits in the same order. By extension a *key match* also when the hashed value of a key is equal to the hashed value of another key.

(k) *watchList*: an array, where each thread has its own position that is used to store the hashed value of the key that thread is operating on.

(l) watchValue: refers to a position in the `watchList` that belongs to a specific thread.

(m) *failCount*: a thread-local counter that is incremented whenever a CAS fails and the thread must retry its attempt to update the table.

(n) *maxFailCount*: a user defined constant used to bound the maximum number of times that a thread continues an operation after a CAS operation fails.

(o) *local*: the *spineNode* that an operation is currently working on.

(p) *pos*: the position on `local` that an operation is currently working on.

(q) markedDataNode: a pointer to a *dataNode* that has been bitmarked at the least significant bit (LSB) of the pointer to the node.

(r) `reuseSpineStack`: an array of pointers, where each thread has its own position that is used to hold pointers to *spineNodes* that the thread had allocated but failed to add to the table.

(s) *reuseDataNodeArray*: a two-dimensional array with T rows — the number of threads. Each row of this array is known as a thread's reuseArray. This array has ten columns — an experimentally derived constant — that is used to store nodes for reuse.

(t) *reuseDataNodeStack*: an array of pointers, where each thread has its own position that is used to store a linked list of nodes for reuse

(u) *Valid Insert Location*: a memory location whose contents are *null* or a *key match*.

User Defined Constants:

(a) *spineSize*: the length of the list in the spine data structure

(b) *spinePow*: the binary logarithm of the `spineSize`

(c) `keySize`: the length in bits of the key

Structure and Traversal of the Hash Table:
The hash table is composed of memory arrays of fixed length where each position points to either a *dataNode*, a *spineNode*, or *null*. Traversing the table is done by taking the first set of bits from the hashed key and examining the pointer at that position on the current memory array. If the pointer stores the address of a *spineNode*, then the next set of bits are taken, and that position on the new memory array is examined. Figure 1 illustrates this scenario.

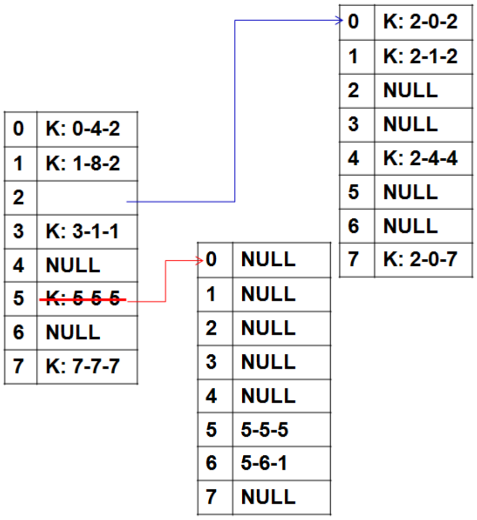

Figure 1: An example of the structure of the hash table.

Figure 1 illustrates the two operations detailed below. Note that the spines have a length of eight; this means that exactly three bits are needed to determine the position to store our *dataNode* on any spine.

If we would like to find the key 0-4-2, then we first need to hash the key. We assume that this operation yields 2-1-2. We shift the hash to isolate the first three bits which equal the decimal value two. So, we examine position two of the main memory array. We see that this position is a spine so we must determine the position on this spine to examine. This is done in the same manner by shifting the original hash again to isolate the next three bits of interest which have the decimal value of one. We examine position one of the spine that is at position two of the main spine. This position contains a *dataNode*, so we compare its hash to the hash of the key that we are searching for. The comparison reveals that the hashes are both equal to 2-1-2, so we return the value associated with this node.

If we would like to insert the key 1-2-3, then — in a similar manner to find — we must first hash the key. We assume that this operation yields 5-6-1, and we examine position five of the main memory array.

This position already holds a *dataNode*, so we must expand the table by adding a spine here. A reference to the *dataNode* that was already there must be present in our new spine before we swap that *dataNode* for the spine. After a successful CAS we then insert our *dataNode* into the new spine.

5.3.3 Main Functions

In this section we provide a brief overview of the main operations implemented by our wait-free hash table. Unless otherwise noted all line numbers refer to the current algorithm being discussed. Note that a *spineNode* or a bit mark cannot exist at `maxDepth`, because no hash collisions can occur there. Unless otherwise stated, all algorithms are bounded by a constant number of steps that is equal to the number of lines in that algorithm.

5.3.4 Algorithm 1 - `put (key, value, threadID)`

The `put` function is used to insert a key-value pair into the hash table if the key is not already in the table, and update the key's value if the key is in the table. In order to promote a more even distribution of keys a hash value is generated by reordering the bits in the key (line 1). The method for generating the hash value is application-dependent and is left to the end-user of the data structure. The only restriction placed on generating a hash value is that perfect hashing must be used, e.g. if the hash of A is equal to the hash of B then A must equal B. A node is then allocated to hold the hash of the key and the key's value (line 2). Next, the thread assigns the hash value, that it is operating on, to its position in the `watchList` (line 4), see Section 5.3.9 for details on our strategy for memory use and reclamation. Finally, the `put` function calls the `putSub` function (defined in Section 5.3.5 that places the key-value pair into the table (line 5). Before returning, the `put` function assigns zero to its position in the `watchList` (Lines 6-7).

Algorithm 1 put *key*, *value*, *threadID*

```
1: hash=hashKey(key);
2: insertThis=allocateNode(value,hash,threadID); \\ see Algorithm 13 for allocateNode
3: local=head;
4: threadWatch[threadID]=hash;
5: putSub(head,insertThis,hash, threadID);
6: threadWatch[threadID]=0;
7: return ;
```

5.3.5 Algorithm 2 - `putSub` `(local, insertThis, hash, threadID)`

The `putSub` function places the node to be inserted, `insertThis`, into the hash table in a bounded number of steps. The operation examines a finite number of memory locations a finite number of times before returning. The number of memory locations that we examine is limited by the `maxDepth` (see Section 5.3.2) and the number of times each memory location is examined is limited by the `maxFailCount`, which is discussed in detail later in this section. For a `put` operation to complete it must reach a memory location that is a *dataNode* or *null*. When it does, it performs a CAS. There are two cases that cause the operation to be considered completed: either the CAS succeeds; or it fails, but the thread can infer that another thread's operation immediately overwrote its `put` operation, this case is described in detail later. Otherwise, the thread re-examines the current contents of the memory location, incrementing its `failCount`. If the the `maxFailCount` is reached it forces an expansion on that position. The expansion provides a new spine that the competing threads can use to insert into, with less contention. When the `maxDepth` is reached, regardless of the result of CAS, the operation is completed. The reason that we still perform a CAS, in this case, is to reuse memory. Whichever thread's CAS succeeds reuses the *dataNode* that was replaced with a *spineNode* during the expansion.

The `putSub` function first determines the position, p, where `insertThis` belongs, on the main spine (line 2) – the same method, for determining the position of interest, is used in the `get` and `remove` functions. The value p is determined by isolating the `spinePow` leftmost bits of the hashed value. For subsequent positions, the hashed key is shifted to the right by `spinePow` bits. The thread then checks the contents of *local[pos]* by performing an atomic read (line 8). Any modifications to the table that occurred before the atomic read (line 8) are considered to have occurred before this operation.

If the value at p contains a pointer to a *spineNode*, signified by a bit mark on the second LSB of the pointer, then the thread examines that *spineNode* (line 10).

If it contains a *markedDataNode* (line 12), then the thread calls `expandTable` (defined in Algorithm 12), on that position. When the `expandTable` operation returns, the contents of that position must be a *spineNode*, which the thread then examines (line 13).

If the position that we called `putSub` on contains a *null* value, then the thread attempts to replace the *null* value with a pointer to `insertThis`, using the atomic CAS operation. If the thread succeeds, then the thread returns, after adjusting the element count (Lines 16-17). When a CAS operation fails, as a result of a thread replacing *null* with a *spineNode* or placing a bit mark on *null* (which becomes a *spineNode* after `expandTable` operation), the thread examines the new spine and tries again, to insert its node. If the CAS fails as a result of another thread inserting a *key match*, then the thread returns because its operation linearizes (see Sections 5.4, 5.5) such that its value was inserted, but was immediately replaced by the other thread's operation (line 29). If the CAS fails as a result of anything else, then the thread increments its `failCount`, and re-examines the position (line 31).

If it contains a *dataNode* that is a *key match*, then the thread attempts to replace the current node with the thread's new node using the atomic CAS operation. If the thread succeeds, then the thread returns, after freeing the node that was removed (line 35). If the thread fails, as a result of another thread bit marking the node, then the thread calls `expandTable` on the position (line 46) and re-examines the new position on the *spineNode* that has been read. If the current node is a spine, then the thread also re-examines the new position on the *spineNode* that has been read (line 43). In all other cases, it can be reasoned that the outcome of this thread's operation occurred, and was immediately overwritten by a subsequent operation.

If the position that we called `putSub` on contains a *dataNode* with a different key, then the executing thread calls `expandTable` (line 52). If the expansion is successful, then the thread re-examines the new position on the *spineNode* that has been read and attempts to insert its node on the new spine. If the expansion failed, then the thread increments its `failCount` and re-examines the contents at that position.

5.3.6 Algorithm 3 - `get (key, threadID)`

The `get` operation determines whether or not a key exists in the table. If it does, then the function returns the key's associated value. The `get` operation examines, at most, the same number of memory locations that the `put` operation does, `maxDepth`; this constant bounds the possible number of steps that this operation could attempt before successful completion.

To search for a key, a hash value is generated from the key and the thread assigns the hash to its *watchValue* (line 2). The position is extracted from the hash, the same process used in `put`, and the thread reads the value held at the position (Lines 5-6). If the value read is a pointer to a *spineNode*, then the thread examines a new position (line 9). If the the value read is a node whose hash value matches (line 11), then the node's value is returned; otherwise, *null* is returned. Before returning, the thread sets its *watchValue* to *null*.

Algorithm 2 putSub $local, insertThis, hash, threadID$

```
for int r=0; r<keySize;r+=spinePow do
    pos=hash&(spineSize-1);
    hash=$hash >> spinePow$;
    failCount=0;
    while true do
        if failCount >maxFailCount then
            markedDataNode(local,pos);
        node=getNodeRaw(local,pos);
        if isSpine(node) then
            local=node;
            break;
        else if isMarked(node) then
            local=expandTable(threadID,local,pos,node,r);
            break;
        else if node==null then
            if CAS(local[pos],null, insertThis) then
                atomicAdd(&currentSize, 1);
                return ;
            else
                node=getNodeRaw(local,pos);
                if isSpine(node) then
                    local=node;
                    break;
                else if isMarked(node) then
                    local=expandTable(threadID,local,pos,node,r);
                    break;
                else if node->hash == insertThis->hash then
                    freeNodeStack(insertThis, threadID);
                    return ;
                else
                    failCount++;
        else
            if node->hash == insertThis->hash then
                if CAS(local[pos],node,newNode) then
                    freeNode(node, threadID);
                    return ;
                else
                    node2=getNodeRaw(local,pos);
                    if node2==null then
                        freeNodeStack(insertThis, threadID);
                        return ;
                    else if isSpine(node2) then
                        local=node2;
                        break;
                    else if isMarked(node2) && unmark(node2)==node then
                        local=expandTable(threadID,local,pos,node,r);
                        break;
                    else
                        freeNodeStack(insertThis, threadID);
                        return ;
            else
                local=expandTable(threadID,local,pos,node,r);
                if !isSpine(local) then
                    failCount++;
                else
                    break;
```

Algorithm 3 get $key, threadID$

```
hash=hashKey(key);
threadWatch[threadID]=hash;
local=head;
for int right=0; right<keySize;right+=spinePow do
    pos=hash&(spineSize-1);
    hash=$hash >> spinePow$;
    node= getNode(local,pos)
    if isSpine(node) then
        local=node;
    else
        if node->hash == hash then
            value=node->value;
        break;
threadWatch[threadID]=0;
return value;
```

5.3.7 Algorithm 4 - `remove (key, threadID)`

The `remove` operation determines if a key exists in the table and, if so, it removes the key from the table. Note that the `remove` operation does not call `get` because `get` returns a value, not a position and a spine. The `remove` operation examines at most the same constant number of memory locations that the `put` operation does, `maxDepth`; this constant bounds the possible number of steps that this operation could attempt before successful completion.

To search for a key, a hash value is generated from the key and the thread assigns the hash to its *watchValue* (line 2). The position is extracted from the hash, the same process used in `put` and `get`, and the thread reads the value held at the position (Lines 5-6).

If the value read is a pointer to a *spineNode*, then the thread examines a new position (line 27). If it is *null*, or a *dataNode* whose hash does not match (Lines 9, 25), the thread exits the `for loop`, clears its *watchValue*, and returns `false`. If a marked node is at the current location, then the thread calls `expandTable` and re-examines the new position on the *spineNode* that has been read (line 11).

If the value read was a *dataNode* whose hash matches, then the thread calls CAS on the position, replacing the value that was read with *null*. If the operation succeeds, then the thread frees the node, decreases the element count, clears its *watchValue*, and returns `true` (Lines 15-17, 28-29). If it fails, and the next read reveals a bit-marked version of the same node or a *spineNode*, then the thread re-examines the new position on the *spineNode* that has been read. Otherwise, the thread returns (line 23), and we reason that the outcome of this thread's operation occurred, but was immediately overwritten by a subsequent operation. The subsequent operation could be a `put` or `remove` with the same key. If the subsequent operation was a `put`, then we reason that we removed the key, but immediately after, another operation `put` the original key. Similarly, if the subsequent operation was `remove`, then that `remove` occurred first and our operation occurred immediately after.

Algorithm 4 remove *key, threadID*

```
 1: hash=hashKey(key);
 2: threadWatch[threadID]=hash;
 3: local=head;
 4: for int r=0; r<keySize;r+=spinePow do
 5:     pos=hash&(spineSize-1);
 6:     hash=$hash >> spinePow$;
 7:     node= getNodeRaw(local,pos)
 8:     if node == null then
 9:         break;
10:     else if isMarked(node) then
11:         local=expandTable(threadID, local,pos,node,r);
12:     else if !isSpine(node) then
13:         if node->hash == hash then
14:             if res=CAS(local[pos], node, null) then
15:                 freeNode(node,threadID);
16:                 decrementSize();
17:                 break;
18:             else
19:                 node2=getNodeRaw(local,pos);
20:                 if isMarked(node2)&&unmark(node2)==node then
21:                     local=expandTable(threadID,local,pos,node,r);
22:                 else
23:                     break;
24:         else
25:             break;
26:     else
27:         local=node;
28: threadWatch[threadID]=0;
29: return res;
```

5.3.8 Supporting Functions

This section briefly describes the supporting functions referenced in the pseudocode of the preceding algorithms. A more in-depth discussion of these algorithms, along with corresponding pseudocode, can be found in Appendix A.

(a) `getNode`: returns the pointer value, without bit marks, held at *pos* on *local*.

(b) `getNodeRaw`: returns the pointer value held at *pos* on *local*.

(c) `inUse`: returns `true` if the hash is present in another thread's *watchValue*, otherwise, it returns `false`.

(d) `markDataNode`: atomically places a bit mark on the value held at *pos* on *local*.

(e) `unmark`: removes the bit marks from a pointer value

(f) `isMarked`: returns `true` if the pointer has a bit mark at its second LSB

(g) `isSpine`: returns `true` if the pointer has a bit mark at its LSB

(h) `expandTable`: adds a new spine when there is a hash collision or a high amount of contention on a single memory location.

(i) `allocateNode`: this function returns a pointer to a node that can be used to store a key-value pair. If there is a valid node in the thread's `reuseDataNodeStack`, then the thread pops the node off the stack and returns that a pointer to that node. If the stack is empty, then the thread checks its `reuseArray` for nodes that were referenced but now are not. If one is found, then the thread uses that node; otherwise, it calls the memory allocator.

(j) `freeNode`: this function determines if there is a possibility that the node is referenced. If there is the potential, then it places the node into the `reuseArray`. If another node is removed from the vector, then that node is moved to the `reuseDataNodeStack`, and the freed node takes its place. If there are no empty positions in the `reuseArray`, then its size is increased by one — as opposed to the standard practice of doubling in size, because the `reuseArray` rarely, if ever, increases in size in practice.

(k) `freeNodeStack`: takes a node that no thread can have a reference to, and pushes it onto the `reuseDataNodeStack`.

5.3.9 Memory Management

This section discusses the allocation and reuse of memory. When designing concurrent applications choosing an appropriate memory management scheme is important, and the one chosen must be thread-safe. As the standard memory allocator is blocking, special provisions must be made for lock-free and wait-free programs. Because of this we have made the design decision to allow the user to pick which memory allocation scheme they use with our hash table — in our case, it must be wait-free. This adds the benefit of allowing our hash table to use memory allocation schemes designed for specific hardware. In order for the hash table to behave in a wait-free manner, the user must choose a memory allocator that can allocate memory in a wait-free manner [52], [33]. We choose the Lockless Memory Allocator [33] because it uses a wait-free queue for synchronization. We have taken care of freeing memory, and have done so in a way that reuses memory in a thread-conscious manner that allows us to reduce overhead.

If a key is removed from the hash table by a successful CAS, then the *dataNode* holding the key is sent to a recycling procedure. We do this because free and allocate are expensive operations, and we can reduce the amount of times that our algorithm calls these functions by reusing memory. In addition to decreasing the number of costly operations executed, recycling *dataNodes* allows us to control which memory locations are used by the hash table, preventing the memory allocator from allocating a memory location that was freed by one thread but is still referenced by another thread; which, would lead to inconsistency in the data structure and the ABA[2] [25] problem.

Our memory reclamation scheme relies on a `watchList`; which, is a global array of length T, the number of threads. Each executing thread stores the hashed value of the key that the thread is operating on to its *watchValue*, the position on the `watchList` that corresponds to the thread's id. By restricting the reuse of a memory address to the thread that recycled it, our reclamation scheme is able to operate without the need for additional compare and swap operations. Since it has been shown that hash tables only need to increase in size [23], once a memory array has been added to the table there is no need to watch it as it is not `remove`d.

Each thread has a `reuseDataNodeStack` and a `reuseDataNodeArray`, nodes located on the stack are not referenced by any other thread and can be reused without additional checks. To place a node on the stack, the node must not be in use — this means that either the node is not in the table or no thread is operating on the hashed value of the node's key. Nodes that are in use are placed in a valid slot in the thread's `reuseDataNodeArray`. Nodes that are no longer in use can be moved from the `reuseDataNodeArray` to the `reuseDataNodeStack`.

5.3.10 ABA Problem

The ABA problem is fundamental to all CAS-based systems [38]. To eliminate the ABA problem we have developed a memory management scheme described in Section 5.3.9.

5.4 Semantics

The brief description of the hash table's semantics, that is presented in the following sections, requires a history of invocations and responses and defines a real-time order. An operation o_1 is said to precede an operation o_2 in real-time order, if o_2's invocation occurs after o_1's response. Operations that do not have real-time ordering are defined as concurrent [22].

5.5 Correctness

The main correctness requirement of the semantics of the hash table's operations is linearizability [22]. A concurrent operation is linearizable if it appears to execute instantaneously in some moment of time between the time point t_{inv} of its invocation and the time point t_{resp} of its response. Firstly, this definition implies that each concurrent history yields responses that are equivalent to the responses of some legal sequential history for the same requests. Secondly, the order of the operations within the sequential history must be consistent with the real-time order. Let us assume that there is an operation $o_m \in S_{hash}$, where S_{hash} is the set of all the hash table's operations. We assume that o_m can be executed concurrently with N other operations o_1,o_2...,o_N $\in S_{hash}$. We outline a proof, that operation o_m is linearizable, in the following sections.

5.6 Wait-Freedom

We prove the non-blocking property of our implementation by showing that out of *n* threads, all makes progress. Since the progress of `get` operations are independent of other concurrent operations, `get` operations are wait-free. If multiple threads are performing conflicting CAS operations, then the single thread that succeeds in completing the CAS operation makes progress by virtue of a successful insert. The failing threads also make progress in that their failure alerts them to one of three possible scenarios. The first scenario occurs when it can be reasoned that the thread's operation is no longer needed. This operation is considered to have been completed, and then immediately overwritten. Following this line of reasoning it

[2] ABA is not an acronym

becomes clear that this operation does not actually need to be performed. The second possibility is that this operation causes an expansion to occur. The final alternative outcome is that the thread simply retries its operation. If this is the case then the thread only retries its operation a limitied number of times (equal to `maxFailCount`) before it is forced to perform an expansion.

By virtue of one-to-one hashing, there are a bounded number of expansions that can occur, $\sum_{i=0}^{\mathtt{maxDepth}-1} memoryArrayLength^i$, because only one unique key can be inserted at a position of this depth. If a thread fails a CAS operation at this stage, then it returns, its operation can be considered to have occurred, then immediately overwritten.

5.6.1 Linearization Points

Linearization points are defined as the requirement that each method have at least a single point where the method appears to take effect. In the presented design, the linearization points are the points when the thread performs an atomic read or a CAS. The ordering of these atomic operations determines the order in which these overlapping method calls occur in the derived sequential history. An atomic read does not modify the state of the data structure, and the read must occur either before or after a write operation, not while a write operation is occurring. This means that the value that is read must have at some point in time existed at the specified memory address. Similarly, for a CAS operation to return `true`, it must have written a value to a memory address. Once the value is written, then it is visible to all threads. If the CAS failed and a *key match* now resides at the location, then this implies that the value was inserted and then immediately replaced. This is intuitive if thought of as a linear program with two sequential insert operations that use the same key. If a *key match* does not reside at the location then the thread expands the table at that point using CAS.

Threads Interleaving

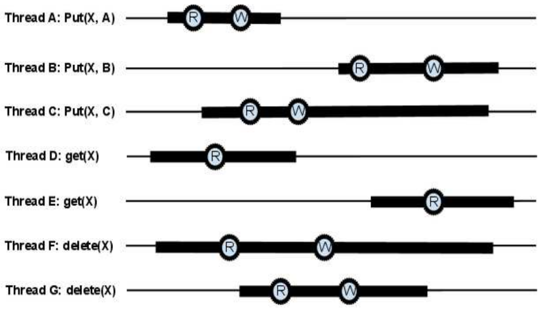

Figure 2: An example of possible threads interleaving.

Figure 2 illustrates various scenarios of the interleaving of threads' method calls and responses on the hash table. Each line represents a single operation that a single thread is executing. The thread executes the circles on its line from left to right. The circle with "R" in the middle refers to the thread reading the pointer value at an arbitrary position on a memory array, and the circle with "W" refers to a CAS operation that uses the value that was previously read to confirm that the value has not changed, and that it is therefore valid to replace it with a new value.

The notation "Thread:Operation" is used. For example, the first circle would be described as A:R, and the one to the left of it would be A:W. The thick black lines represent a method call by a thread and each circle represents a serializable point of execution, and overlapping black lines represent method calls that happen concurrently.

The gaps between circles and end points reflect arbitrary passages of time that are not based on the lengths of the lines between them. Furthermore, circles on different lines that have overlapping black lines, can be executed in any order, but each must still execute from left to right on its own line. For example, if only threads A and C were considered one potential valid order of execution would be A:R then C:R then C:W then finally A:W. However, A:W then A:R then B:R, then finally B:W, would not be a valid execution since A:W must occur after A:R.

We assume that all threads are executing on the same memory array and that the initial state of the position on the memory array is *null*. We also propose that no thread has priority and each CAS failure implies that the thread inserted its element, and then that element was instantaneously replaced with a newer element.

Serializability of Multiple `put` **Operations:**
Considering threads A through E from Figure 2 in this example, we propose an ordering, diagrammed below:

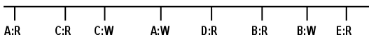

Figure 3: A potential ordering of multiple `put` operations.

Consider the following scenario wherein threads return values that are prefixed with "val_." Thread C would pass its CAS operation, unlike A, which fails its CAS because of C's write operation. Thread D's `get` operation would return "val_C," then thread B would replace "val_C" with "val_B," and finally thread E would return "val_B." A sequential history is shown in the timeline that follows in Figure 4. Regardless of the number of threads or the order of operations, a valid sequential history can be derived by using the linearization points.

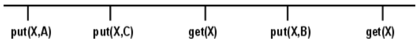

Figure 4: A sequential history of the proposed ordering.

5.7 Linearizability

The following section discusses how we can derive a valid sequential history from a set of concurrent operations of our hash table. The serialization of concurrent modifications to distinct memory locations is considered trivial. The case where multiple `get` operations are called on the same memory address is also considered trivial.

For `put` and `remove` operations that use atomic load and atomic CAS operations we define the initial contents of the memory location as the value read by the respective atomic load of `put` or `remove`.

When considering several competing concurrent operations, one operation succeeds in the CAS. Each operation can be ordered relative to the operation that succeeds. The operations that fail, all of those except the succeeding operation, are considered to have taken place but had their changes immediately replaced by that of the succeeding operation. Therefore, these failed operations are ordered before the succeeding operation. All possible orderings of the failed operations with each other and the succeeding operation, lead to the same result — the result of the succeeding operation.

A `get` operation that executes concurrently with a `put` or `remove` operation, on the same memory address, returns the value placed there by the most recent CAS operation that occurred before the `get`'s atomic read.

In order to make the illustration of the linearization of concurrent function calls as clear as possible, it has been broken up into three tables — Table 1, Table 2, and Table 3 — detailing the possible values that may be present at an arbitrary position in the hash table. Read operations performed on spine nodes are omitted from the tables, because at no point does a position change from a *spineNode* to anything else.

The top left corner of each table contains the initial value held at the memory address that the threads are operating on. The leftmost column contains one of the concurrent operations that is performed, this is denoted by 'o_1'. The top row holds the action that the other operation performs, this is denoted by 'o_2'. The cell at the intersection of a row and a column contains the point in o_1's code and the point in o_2's code where o_1's actions and o_2's actions can be linearized.

The notation used in the table includes: 'CAS(A,V_o,V_n)' which denotes a compare- and-swap on a memory location A, comparing the current value with the old value, V_o, replacing the current value with the new value, Vn, if V_o equals the current value; 'S' which refers to adding a *spineNode* as opposed to a *dataNode*; 'Read(A),' which is an atomic read on memory address A; and 'Mark(A),' which refers to performing the atomic, bit-wise 'and' operation. Furthermore, the phrase "new phase" is used to describe the case wherein an operation fails its CAS. In this scenario, the thread determines — as described in Section 5.6 — if it still needs to perform its operation; if so, it increases its `failCount` and retries.

Below we provide a detailed explanation of the linearization points shown in Table 1, Table 2, and Table 3.

Table 1: Linearization of concurrent operations on a position that is initially *null*.

Initial Value: V_o=*null*	o_2: `get`(k)	o_2: `put`(k,V_{n_2})	o_2: `remove`(k)	o_2: `put`(j,V_{n_2})	o_2:mark(A)
o_1:`put`(k,V_{n_1})	o_1: CAS(A,V_o,V_{n_1}) o_2: Read(A) **1**	o_1: CAS(A,V_o,V_{n_1}) o_2: CAS(A,V_o,V_{n_2}) **2**	o_1: CAS (A,V_o,V_{n_1}) o_2: Read(A) **3**	o_1: CAS(A,V_o,V_{n_1}) o_2: CAS(A,V_o,V_{n_2}) **4**	o_1: CAS(A,V_o,V_{n_1}) o_2: Mark(A) **5**

(1) If o_1 completes its CAS before o_2 completes its atomic read, then o_1 occurred before o_2. The result of this is that o_2 returns V_{n_1} as opposed to the initial value. If o_2 completes its atomic read before o_1 completes its CAS, then o_2 occurred before o_1. The result of this is that o_2 returns the initial value, then o_1 updates the key.

(2) If o_1 completes its CAS before o_2 completes its CAS, then o_2 fails its CAS operation. However, it is considered that o_2 occurs before o_1, and that o_1 immediately overwrote it. If o_2 completes its CAS before o_1 completes its CAS, then o_1 fails its CAS operation. However, it is considered that o_1 occurs before o_2, and that o_2 immediately overwrote it.

(3) If o_1 completes its CAS before o_2 completes its atomic read, then o_1 occurred before o_2. The result of this is that o_1 inserted its value into the hash table, then o_2 read the value and proceeds to `remove` it by invoking a CAS operation. If o_2 completes its atomic read before o_1 completes its CAS, then o_2 occurred before o_1. The result of this is that o_2 attempted to `remove` a key that wasn't in the table and returned `false`, then o_1 inserted that key.

(4) If o_1 completes its CAS before o_2 completes its CAS, then o_1 occurred before o_2. The result of this is that o_1 inserted its value into the hash table, then o_2, upon failing the CAS, performs an atomic read, and enters a new phase. If o_2 completes its CAS before o_1 completes its CAS, then o_2 occurred before o_1. The result of this is that o_2 inserted its value into the hash table, then o_1, upon failing the CAS, performs an atomic read, and enters a new phase. See Table 2 for more details.

(5) If o_1 completes its CAS before o_2 completes its atomic mark, then o_1 occurred then o_2 occursred. The result of this is that o_1 was inserted then it was marked. If o_2 completes its atomic mark before o_1 completes its CAS, then o_2 occurred before o_1. The result of this is that o_2 marked the node, causing o_1 to fail the CAS. Both operations upon reading a marked node calls `expandTable`.

Table 2: Linearization of concurrent operations on a position that is initially a *dataNode*.

Initial Value: *dataNode* with Key k, Value Z	o_2: `get`(k)	o_2: `put`(k,V_{n_2})	o_2: `remove`(k)	o_2: `put`(j,V_{n_2})	o_2:mark(A)
o_1:`put`(k,V_{n_1})	o_1: CAS(A,V_o,V_{n_1}) o_2: Read(A)	o_1: CAS(A,V_o,V_{n_1}) o_2: CAS(A,V_o,V_{n_2})	o_1: CAS (A,V_o,V_{n_1}) o_2: CAS(A,V_o,*null*)	o_1: CAS(A,V_o,V_{n_1}) o_2: CAS(A,V_o,S_{n_2})	o_1: CAS(A,V_o,V_{n_1}) o_2: Mark (A)
	1	**2**	**6**	**7**	**5**
o_1: `remove`(k)	o_1: CAS(A,V_o,*null*) o_2: Read(A)	o_1: CAS(A,V_o,*null*) o_2: CAS(A,V_o,V_{n_2})	o_1: CAS (A,V_o,*null*) o_2: CAS(A,V_o,*null*)	o_1: CAS(A,V_o,*null*) o_2: CAS(A,V_o,S_{n_2})	o_1: CAS(A,V_o,*null*) o_2: Mark (A)
	8	**9**	**10**	**11**	**12**

(6) If o_1 completes its CAS before o_2, then o_1 occurred before o_2. The result of this is that o_1's value was inserted, then o_2, upon failing the CAS, performs an atomic read then `remove` the key. If o_2 completes its CAS before o_1, then o_2 occurred before o_1. The result of this is that o_2 removed the node, then o_1, upon failing the CAS, attempts to insert the node again.

(7) If o_1 completes its CAS before o_2, then o_1 occurred before o_2. The result of this is that o_1's value was inserted, then o_2, upon failing the CAS, performs an atomic read then call `expandTable` again. If o_2 completes its CAS before o_1, then o_2 occurred before o_1. The result of this is that o_2 expanded the table, then o_1, upon failing the CAS, attempts to replace the node again.

(8) If o_1 completes its CAS before o_2 completes its atomic read, then o_1 occurred before o_2. The result of this is that o_1 has removed the node, then o_2 has return *null* because the key was no longer in the table. If o_2 completes its atomic read before o_1 completes its CAS, then o_2 occurred before o_1. The result of this is that o_2 has returned the current value associated with key k, then o_1 has removed the node containing key k.

(9) If o_1 completes its CAS before o_2 completes its CAS, then o_2 occurred before o_1. The result of this is that o_2 replaced the current node with a new node, then o_1 immediately removed that node from the table. If o_2 completes its CAS before o_1 completes its CAS, then o_1 occurred before o_2. The result of this is that o_1 removed the node containing the key k, but o_2 immediately inserted a node containing key k.

(10) If o_1 completes its CAS before o_2 completes its CAS, then o_1 occurred before o_2. The result of this is that o_1 removed the node containing key k from the table then o_2 failed to `remove` the node, causing o_2 to return. If o_2 completes its CAS before o_1 completes its CAS, then o_2 has occurred before o_1. The result of this is that o_1 removed the node containing key k from the table then o_1 failed to `remove` the node, causing o_1 to return.

(11) If o_1 completes its CAS before o_2 completes its CAS, then o_1 occurred before o_2. The result of this is that o_1 removed the node containing key k, and o_2 failed to expand the table because it failed CAS a spine in. o_2 enters a new phase where it reads a *null* value and acts accordingly. If o_2 completes its CAS before $o_1$completes its CAS, then o_2 has occurred before o_1. The result of this is that o_2 expanded the table causing o_2 fail its CAS. Both o_1 and o_2 enter a new phase where they read the current value at memory address A and act accordingly.

(12) If o_1 completes its CAS before o_2 completes its marking of address A, then o_1 has occured before o_2. The result of this is that o_1 removed the node at A setting the value of that position to *null*, then o_2 marked the *null* value (*null* is 0x0, and a mark would produce 0x1). o_2 then enters a new phase where it calls `expandTable` at memory address A. If o_2 completes its marking of address A, before o_1 completes its CAS, then o_2 has caused o_1 to fail its CAS. o_1 then performs an atomic read on A, which returns

either a marked node or a *spineNode*, and then enter a new phase. o_2 also enters a new phase where it calls `expandTable` at memory address A.

Table 3: Linearization of concurrent operations on a position that is initially a *markedDataNode*.

Initial Value: *markedDataNode* with Key k, Value Z	o_2: `get`(k)	o_2: `put`(k,V_{n_2}) o_2: `remove`(k) o_2: `put`(j,V_{n_2})	o_2:mark(A)
o_1: `put`(k,V_{n_1}) o_1: `remove`(k)	o_1: CAS(A,V_o,S_{n_1}) o_2: Read(A)	o_1: CAS(A,V_o,S_{n_1}) o_2: CAS(A,V_o,S_{n_2})	o_1: CAS(A,V_o,S_{n_1}) o_2: Mark(A)
	13	14	15

(13) If o_1 completes its CAS before o_2 completes its atomic read, then o_1 has completed its `expandTable` call before o_2, as a result o_2 read a *spineNode* and enter a new phase and o_1 also enters a new phase, where it attempts to complete its operation the *spineNode* it just inserted. If o_2 completes its atomic read before o_1 completes its CAS, then o_2 occurred before o_1 completes its `expandTable` operation. As a result o_1 returns the current value, and o_1 enters a new phase, where it attempts to complete its operation the spine node it just inserted.

(14) If o_1 or o_2 completes its CAS operation then the other fails their CAS operation, the failing thread performs an atomic read which would return a *spineNode* pointer. Both operations enters a new phase based on this pointer.

(15) If o_1 completes its CAS before o_2 completes its atomic mark operation, then o_2 has completed its expansion operation, the the result of this is that o_2 marks the *spineNode*, then both operations enters a new phase based on this *spineNode*. A mark on a *spineNode* point has no affect because of the way the algorithm interprets *spineNode* pointers. If o_2 completes its mark of address A before o_1 completes its CAS then o_1 fails its CAS and enter a new phase where it reads the current value. Any operation reading of a marked node calls `expandTable`, then enter a new phase.

5.8 Performance Evaluation

We tested several algorithms against our wait-free implementation (Wait-Free). As there are no other wait-free algorithms we chose several lock-free algorithms as well as some standard locking algorithms. The lock-free algorithms that we compare against are Split-Ordered Lists (Split Lists) [46] and Michael's lock-free hash table (Michael) [37]. The standard locking solutions include two data structures with global locks, these are the standard template library hash table (STL) [28] and the Boost unordered map (Boost) [1]. We also compare against the industry standard — the best available locking solution — Intel's Threading Building Blocks concurrent hash map (TBB) [27].

All tests were run on an HP Z600 workstation with an Intel X5670 hex-core processor running at 2.93 GHz and six gigabytes of RAM that is running 64-bit Ubuntu Linux version 11.04. The testing variables for the graph presented below include creating a hash table that has an initial capacity of 2^{18} = 262,144 elements. This number of elements is chosen because it is the closest power of two to five percent of the expected number of inserted elements. This table was filled to one percent of its capacity and then 50,000,000 operations were performed. The Boost random number generator was used to avoid the locking version in the standard C++ implementation. Each thread then used this random number generator to select what type of operation it should perform, where each operation was assigned an arbitrary yet consistent opcode. We tested a variety of scenarios and our hash table consistently outperforms the rest with a variety of distributions of the executed operations. Here we show just one scenario from our evaluation tests. We selected a distribution of operations that contained 88% `get`, 10% `put`, and 2% `remove` operations. This distribution was selected because it was reported to be typical for use of this data structure [46]. We have executed a large variety of scenarios and observed results that indicate that our algorithm scales well with the number of threads.

A graph, Figure 5, was produced to illustrate the performance of these algorithms under these conditions. When considering the graph note the logarithmic scale and the fact that a higher number of clock ticks signifies lower performance. Also, the STL and Boost tables scale similarly and the lines that depict their performance overlap almost exactly. Our implementation is represented by the line at the bottom of the graph, using the fewest clock ticks, and showing the best performance of all algorithms considered, in the aforementioned environment. Similar results can be observed for up to 128 threads in Figure 6. We present another performance graph that was tested in the same environment — same machine and same variables — as the previously presented results with only the number of threads changed.

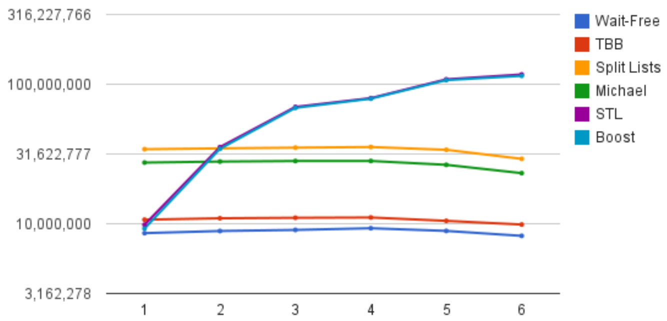

Figure 5: The average number of clock ticks, per thread, used by each algorithm when running the specified total number of threads.

5.9 Relevance

We believe that a lock- and wait-free hash table allows significant performance increases across any parallel architecture. Our design of a wait-free hash table is critical for a large number of applications, such as: computational biology[35]; simulation [42], [57]; discrete event simulation [30]; and search-indexing [59]. The application and availability of a wait-free design for a hash table data structure helps improve the performance and scalability of such existing and future applications. Specifically, our data structure could be used in biological research where both search and computation can involve retrieving and processing vast libraries of information [54]. The same data is also often accessed by different users with varied goals and it would be ideal if their work did not interfere with that of another.

6 Conclusions and Future Work

We presented the first wait-free hash table implementation. Our implementation provides the progress guarantee of wait-freedom with a performance improvement over the best available locking solution and all

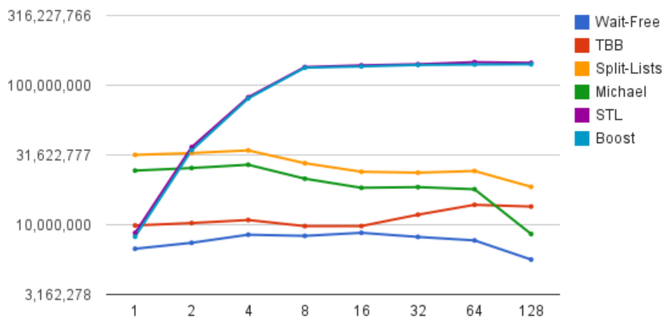

Figure 6: The average number of clock ticks, per thread, used by each algorithm when running the specified total number of threads.

tested lock-free solutions. We discussed the relevance of this work and its applicability in the real-world.

We are currently developing a project that applies advanced program analysis provided by POET [58] to automaticallly replace TBB's concurrent hash map with our wait-free hash table in real-world applications and a number of benchmarks such as PARSEC [3]. Furthermore, we plan to perform a formal validation using a model checker, such as SPIN to exhaustively test all possible thread interleavings [56].

References

[1] Boost c++ libraries. `http://www.boost.org/`, January 2012.

[2] T.-H. Ahn, D. Dechev, H. Lin, H. Adalsteinsson, and C. Janssen. Evaluating Performance Optimizations of Large-Scale Genomic Sequence Search Applications Using SST/macro. In *Proceedings of the 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications, SIMULTECH 2011*, 2011.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] Cliff Click. A lock-free hash table (`http://www.azulsystems.com/events/javaone\_2007/2007\_LockFreeHash.pdf`). Retrieved 12/12/2011.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] A. Dakshinamurthy and D. Dechev. Automatic Extraction of SST/macro Skeleton Models. In *Proceedings of the 25th ACM International Conference on Supercomputing*, 2011.

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

[8] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In A. A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2006.

[9] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In M. Shvartsman, editor, *Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer Berlin / Heidelberg, 2006.

[10] D. Dechev, N. Rouquette, P. Pirkelbauer, and B. Stroustrup. *Programming and Validation Techniques for Reliable Goal-driven Autonomic Software, Book Chapter in Autonomic Communication*. Springer, Reading, Massachusetts, 2009.

[11] D. Dechev and B. Stroustrup. Reliable and Efficient Concurrent Synchronization for Embedded Real-Time Software. In *Proceedings of 3rd IEEE International Conference on Space Mission Challenges for Information Technology (IEEE SMC-IT 2009)*, 2009.

[12] 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.

[13] D. Dechev and B. Stroustrup. Scalable nonblocking concurrent objects for mission critical code. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 597–612, New York, NY, USA, 2009. ACM.

[14] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4:315–344, September 1979.

[15] S. Feldman, P. LaBorde, and D. Dechev. Facilitating Efficient Parallelization of Information Storage and Retrieval on Large Data Sets. In *Proceedings of the 25th ACM International Conference on Supercomputing*, 2011.

[16] K. Fraser. Practical lock-freedom. In *Computer Laboratory, Cambridge Univ*, 2004.

[17] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[18] H. Gao, J. F. Groote, and W. H. Hesselink. Almost wait-free resizable hashtable. In *IPDPS*, 2004.

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

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

[21] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23:146–196, May 2005.

[22] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[23] M. Hsu and W.-P. Yang. Concurrent operations in extendible hashing. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 241–247, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[24] I. Gartner. Grand Challenges for IT (http://www.gartner.com/it/page.jsp?id=643117). Retrieved 12/11/2011.

[25] IBM Corporation. *System/370 Principles of Operation*. 1983.

[26] Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, Santa Clara, CA, 2004.

[27] Intel Corporation. Reference for Intel Threading Building Blocks (http://threadingbuildingblocks.org/). Retrieved 12/12/2011.

[28] ISO/IEC 14882 Standard for Programming Language C++. *Programming languages C++*. American National Standards Institute, September 2011.

[29] C. L. Janssen, H. Adalsteinsson, S. Cranford, D. Dechev, J. P. Kenny, N. Lemaster, J. Mayo, A. Pinar, and D. A. Evensky. Exascale Co-design with Sandia's Structural Simulation Toolkit (SST) Coarse-grained Components. In *In Proceedings of 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 2010), Supercomputing (SC 2010)*, 2010.

[30] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny. Using simulation to design extremescale applications and architectures: programming model exploration. *SIGMETRICS Perform. Eval. Rev.*, 38:4–8, March 2011.

[31] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM.

[32] L. Lamport. How to make a multiprocessor computer that correctly executes programs, *IEEE Trans. Comput.*, September 1979.

[33] Lockless Inc. Technical specifications for the lockless inc. memory allocator. `http://locklessinc.com/technical_allocator.shtml`, December 2011.

[34] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[35] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[36] 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*, pages 651–660, 2003.

[37] 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*, pages 73–82, New York, NY, USA, 2002. ACM Press.

[38] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004.

[39] Microsoft. System.collections.concurrent namespace. `http://msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx`, 2011. .NET Framework 4.

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

[41] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[42] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[43] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[44] Sandia Nationional Labs. SST/macro (http://sst.sandia.gov/using_sstmacro.html). Retrieved 10/22/2011.

[45] Sandia Nationional Labs. Mantevo (https://software.sandia.gov/mantevo). Retrieved 10/22/2011.

[46] 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*, pages 102–111, New York, NY, USA, 2003. ACM Press.

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

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

[49] A. Stepanov and P. McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

[50] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[51] B. Stroustrup. About C++ and few more things. In *The Software Developer's Journal*, 2011.

[52] H. Sundell. Wait-free reference counting and memory management. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 24b, april 2005.

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

[54] O. Trelles, P. Prins, M. Snir, and R. C. Jansen. Big data, but are we ready? *Nature Reviews Genetics*, 12(3):224–224, March 2011.

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

[56] M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In C. Pasareanu, editor, *Model Checking Software*, volume 5578 of *Lecture Notes in Computer Science*, pages 261–278. Springer Berlin / Heidelberg, 2009.

[57] J. R. Williams, D. Holmes, and P. Tilke. Parallel computation particle methods for multi-phase fluid flow with application oil reservoir characterization. In E. Oate and R. Owen, editors, *Particle-Based Methods*, volume 25 of *Computational Methods in Applied Sciences*, pages 113–134. Springer Netherlands, 2011.

[58] Q. Yi. Poet: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience*, 2011.

[59] Y. Zhao, H. Tang, and Y. Ye. Rapsearch2: A fast and memory-efficient protein similarity search tool for next generation sequencing data. *Bioinformatics*, 2011.

A Supporting Functions

In this section we briefly introduce several supporting functions that are important for the effective wait-free implementation of the hash table's main operations.

A.1 Algorithm 5 - `getNode (local, pos)`

This is a wrapper function for atomic read operation and it returns the memory address of the node stored at *pos* on *local*'s memory array. The function also clears any bit mark that another thread may have placed (line 1). The `get` function is the only function that calls `getNode`, and since the `get` function does not perform any table modifications, it ignores bit marks; which are used by threads that modify the table.

Algorithm 5 getNode $local, pos$

```
1: return (local[pos]) AND NOT 2;
```

A.2 Algorithm 6 - `getNodeRaw (local, pos)`

This functions is similar to `getNode`. It is a wrapper function for atomic read operation and it returns the memory address at *pos* on *local*'s memory array. The difference is that this function does not ignore bit marks.

Algorithm 6 getNodeRaw $local, pos$

```
return (local[pos]);
```

A.3 Algorithm 7 - `inUse (hash, threadID)`

This function takes a hash value and searches the `watchList` for a match. If a match is found, then that implies that a thread is performing an operation using that hash value. If a match is found then the function returns `true`, otherwise, it returns `false`. Please see the section titled "Memory Management" for more details on how this function is used.

Algorithm 7 inUse $hash, threadID$

```
for int i=0; i<numThreads;i++ do
    if hash==threadWatch[i] AND $threadID ! = i$ then
        return true;
return false;
```

A.4 Algorithm 8 - `markDataNode (local, pos)`

This function is a wrapper function for atomic `or` operation, it is used to atomically bit-mark the value stored at *pos* in the array *local*.

Algorithm 8 markedDataNode *local*, *pos*

```
AtomicOR(local[pos],2);
```

A.5 Algorithm 9 - `unmark (node)`

This function takes a pointer value, and returns the `unmark`ed pointer value.

Algorithm 9 unmark *node*

```
return ((unsigned long)node AND NOT 2);
```

A.6 Algorithm 10 - `isMarked (node)`

This function returns `true` or `false` depending on whether or not it is a *markedDataNode*.

Algorithm 10 isMarked *node*

```
return ((unsigned long)node AND 2);
```

A.7 Algorithm 11 - `isSpine (node)`

This function returns `true` or `false` depending on whether or not it is a *spineNode*.

A.8 Algorithm 12 - `expandTable (threadID, local, pos, node, right)`

This function is used to expand the table when there is a hash collision or a high amount of contention on a single memory location. If the current value at *pos* in *local* is marked, it is guaranteed that when the function returns, the contents at *pos* in *local* is a *spineNode*. This function simply returns the state after a single CAS operation; therefore, it is bounded by the fifteen steps enumerated in the algorithm as there are no loops. First, the function calculates the position that the current node belongs in on the new spine (line 1). Then, it checks to make sure that the node still resides at the position, returning the current value if it does not (Lines 2-3). There is no reason to attempt a CAS if the thread knows that it fails, because the current value no longer equals the expected value. Then, the thread checks to see if it has a *spineNode* in its `reuseSpineStack`, if not, it allocates a new *spineNode* (Lines 4-8); otherwise, it uses the *spineNode* from its stack. After inserting the node, into the *spineNode*'s memory array, the thread attempts to replace the current node with the *spineNode* (line 10). If it succeeds, it returns the spine. Otherwise, it returns the current node, after placing the spine in its `reuseSpineStack`.

A.9 Algorithm 13 - `allocateNode (value, hash, threadID)`

This function returns a pointer to a node that can be used to store a key-value pair. First, the thread checks its `reuseDataNodeStack` to see if it has any nodes that it can reuse, if it does, it pops the top element (Lines

Algorithm 11 isSpine *node*

```
return ((unsigned long)p & 1);
```

Algorithm 12 expandTable $threadID, local, pos, node, right$

```
ipos=(node->$hash >> (right + spinePow)$)&(spineSize-1);
if $local[pos]! = node$ then
    return local[pos];
if reuseSpineStack[threadID]==null then
    spine=malloc(sizeof(Spine));
else
    spine=reuseSpineStack[threadID];
    reuseSpineStack[threadID]=null;
spine[ipos]=node;
if CAS(local[pos]), node, spine) then
    return spine;
else
    spine[ipos]=null;
    reuseSpineStack[threadID]=spine;
    return getNodeRaw(local,pos);
```

1-3). Otherwise, it checks its `reuseDataNodeArray` for any nodes that were in use, but are no longer in use (Lines 5-9). If no node is available for reuse, then the thread allocates a new node (line 11). Then, the hash and the value are assigned to the node and a pointer to the node is returned (Lines 12-14). The bound of this algorithm is determined by the memory allocator which is outside of the scope of this algorithm.

Algorithm 13 allocateNode $value, hash, T/*threadID*/$

```
1:  node=ThreadPoolStack[T];
2:  if node!=null then
3:      ThreadPoolStack[T]=node->next;
4:  else
5:      for int i=1; $i < ThreadPoolVector[T][0]$; i++ do
6:          node=ThreadPoolVector[T][i];
7:          if $node! = null$ && !inUse(node->hash,T) then
8:              ThreadPoolVector[T][i]=null;
9:              break;
10:     if node==null then
11:         node=malloc(sizeof(DataNode));
12: node->value=value;
13: node->hash=hash;
14: return node;
```

A.10 Algorithm 14 - `freeNode (value, hash, threadID)`

This function is used to recycle a node that is removed from the table, so that it can be reused later. First, it checks to see if any thread is performing an operation with the same hash as the node that was `remove`d, if not, then the node can be placed on the *dataNode* stack for immediate reuse (Lines 1-4). Otherwise, the thread searches its own `reuseDataNodeArray` for a slot to place the node, if it comes across an empty position, it places the node there (Lines 8-9). If it comes across a position with a node that is not in use, then it places that node into its `reuseDataNodeStack` and place the original node in that position (Lines 10-13). In the rare case that no valid slot can be found, the thread reallocates the array and place the node in the last position (Lines 17-19). This function is bounded by T + S where T is the number of threads and S is the size of the `reuseDataNodeArray`; this constant bounds the possible number of steps that this operation could attempt before successful completion.

Algorithm 14 freeNode $node, T/*threadID*/$

```
if !inUse(node,T) then
    node->next=ThreadPoolStack[T];
    ThreadPoolStack[T]=node;
    return ;
else
    size=ThreadPoolVector[T][0][
    for int i=1; $i < size$; i++ do
        D=ThreadPoolVector[T][i];
        if D==null then
            ThreadPoolVector[T][i]=node;
            return ;
        else if !inUse(D,threadID) then
            D->next=ThreadPoolStack[T];
            ThreadPoolStack[T]=D;
            ThreadPoolVector[T][i]=node;
            return ;
    ThreadPoolVector[T]=realloc(ThreadPoolVector[T],size+1);
    ThreadPoolVector[T][0]=size+1;
    ThreadPoolVector[T][size]=node;
    return ;
```

A.11 Algorithm 15 - `freeNodeStack (node, threadID)`

This function is used to recycle a node that was never in the table, so that it can be reused later. Since this node was never in the table, it can be placed in the `reuseDataNodeStack` without concern, because no other thread could possibly hold a reference to it. This function simply returns the state after a single CAS operation; therefore, it is bounded by the two steps enumerated in the algorithm as there are no loops.

Algorithm 15 freeNodeStack $node, threadID$

```
node->next=ThreadPoolStack[threadID];
ThreadPoolStack[threadID]=node;
return ;
```

DISTRIBUTION

1	MS0899	Technical Library	9536 (electronic copy)

Sandia National Laboratories