Cache-Conscious Concurrent Data Structures

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

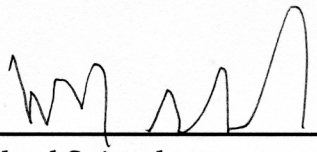
Michael Spiegel

May 2011

Approval Sheet

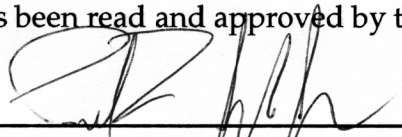
This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

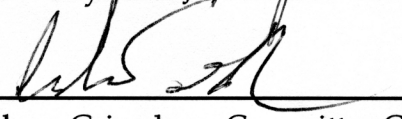


Michael Spiegel

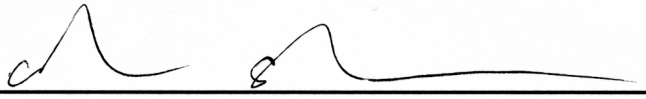
This dissertation has been read and approved by the Examining Committee:



Paul F. Reynolds Jr., Advisor

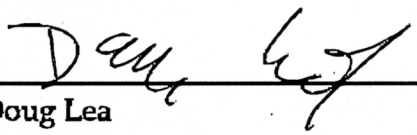


Andrew Grimshaw, Committee Chair



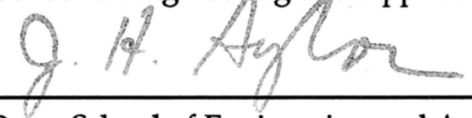
abhi shelat

Steven Boker



Doug Lea

Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

May 2011

In memory of Baruch Arazi (1921 – 2009) .

Acknowledgments

This dissertation began as a side project during a summer internship at Sun Laboratories. I would like to thank the members of the Programming Languages Research Group and the Scalable Synchronization Research Group at Sun Labs for their initial encouragement and guidance. Special thanks go to Victor Luchangco from Sun Labs and abhi shelat from the University of Virginia, for their time and insights as I came to them for feedback in various stages of the proofs for the skip tree structural properties. I would like to thank Doug Lea for his continual encouragement to work on Java concurrency issues and for making critical introductions with colleagues at Azul Systems that facilitated the successful supercomputer experiments of my dissertation.

I would like to thank my research advisor, Paul Reynolds, for lending his patience, guidance, and research experience to this dissertation. This is my second dissertation research proposal in my graduate career with Paul supervising both projects. Paul has always maintained an open mind throughout the process as this side project developed into something more substantial. My wife and I owe him a special debt of gratitude during this final year for accommodating my remote office in Greensboro, NC in the writing phase of the dissertation.

For the last three years I have been an active participant in the OpenMx project to develop an R package for advanced structural equation modeling. I have received funding from NIH grant 1R21DA024304–01 that provides funding for this

effort. This project has been lead by the Human Dynamics Lab at the University of Virginia in collaboration with Virginia Commonwealth University, Argonne National Labs, and other participating institutions. Thank you to Steven Boker and his team of graduate students at the Human Dynamics Lab for the opportunity to participate on the OpenMx project. It has been an ideal interdisciplinary collaboration, where each participant acknowledges, encourages, and respects the unique academic training and perspective that each discipline contributes to the success of the team.

I would like to thank the University of Virginia Alliance for Computational Science and Engineering for access to an Intel Xeon compute cluster, Jack Davidson for access to a Sun Fire T1000, the Pittsburgh Supercomputing Center (PSC) for access to Pople and Blacklight, an SGI Altix 4700 Itanium shared-memory system and an SGI Altix UV 1000 Intel Xeon shared-memory system, and Azul Systems for access to an Azul compute appliance. Andrew Grimshaw has been instrumental for his advice during the process of submitting a TeraGrid allocation request and for forwarding introductions to the staff at PSC. Thanks to PSC for granting us beta-tester status on Blacklight as it was preparing for production.

I want to thank my friends and family for their support in this endeavor. My peers in the department have kept a positive and exciting environment for work and study. I also wish to thank members of the eight block for deciding to study at the University of Virginia. I would like to thank my parents Sarah, Shel, Yitzhak, and Miriam for encouraging my academic endeavors. My siblings, my extended family, and my family-in-laws also deserve credit for their encouragement. Lastly, and most importantly, I would never have completed my studies without the support of my wife. Rachel has given meaning and purpose to this endeavor.

Abstract

The power wall, the instruction-level parallelism wall, and the memory wall are driving a shift in microprocessor design from implicitly parallel architectures towards explicitly parallel architectures. A necessary condition for peak scalability and performance on modern hardware is application execution that is aware of the memory hierarchy. The thesis of this dissertation is that cache-conscious concurrent data structures for many-core systems will show significant performance improvements over the state of the art in concurrent data structure designs for those applications that must contend with the deleterious effects of the memory wall. Lock-free cache-conscious data structures that maintain the abstraction of a linearizable set have been studied previously in the context of unordered data structures. We explore novel alternatives, namely lock-free cache-conscious data structures that maintain the abstraction of a linearizable ordered set. The two primary design contributions of this dissertation are the lock-free skip tree and lock-free burst trie algorithms. In both algorithms, read-only operations are wait-free and modification operations are lock-free. The lock-free skip tree has relaxed structural properties that allow atomic operations to modify the tree without invalidating the consistency of the data structure. We define the dense skip tree as a variation of the skip tree data structure, and prove cache-conscious properties of the dense skip tree. The proof techniques represent a significant departure from the methods outlined in the original skip tree paper.

We show that cache-conscious, linearizable concurrent data structures have advantageous performance that can be measured across multiple architecture platforms. The improved performance arises from better treatment of the memory wall phenomenon that is ubiquitous to current multi-core systems and almost certainly will continue to affect future many-core systems. Using a series of synthetic benchmarks we have shown that our lock-free skip tree and burst trie implementations perform up to x2.3 and x3.5 faster in read-dominated workloads on SPARC and x86 architectures, respectively, compared to the state of the art lock-free skip list. The minimum performance of the skip tree across all workloads and architectures is x0.87 relative to the skip list performance. An analysis of heap utilization of the data structures in the synthetic benchmark reveals the lock-free skip tree to use 59% of the heap utilization of the skip list and the lock-free burst trie to use 140% of the skip list heap utilization. In a series of four parallel branch-and-bound applications, two of the applications are x2.3 and x3.1 faster when using the lock-free skip tree as a concurrent priority queue as compared to the lock-free skip list. In a shared-memory supercomputer architecture the two branch-and-bound applications are x1.6 and x2.1 faster with the skip tree versus the skip list running at 80 hardware threads.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A necessary condition for peak scalability and performance on modern hardware is application execution that is aware of the memory hierarchy. The power wall, the instruction-level parallelism (ILP) wall, and the memory wall are driving a shift in microprocessor design from implicitly parallel architectures towards explicitly parallel architectures [1, 2]. The power wall represents the set of constraints imposed by power consumption and power dissipation issues due to decreases in transistor size and increases in clock frequency. On-chip power dissipation now exhausts the maximum capability of conventional cooling technologies; any further increases will require expensive and challenging solutions (e.g., liquid cooling), which would significantly increase overall system cost [3].

The ILP wall represents the limits of the exploitation of instruction-level parallelism. Pipelining of individual instruction execution into a sequence of stages has allowed designers to increase clock rates as instructions have been sliced into larger numbers of increasingly small steps, which are designed to reduce the amount of logic that needs to switch during every clock cycle. Superscalar processors were developed to execute multiple instructions from a single, conventional instruction stream on each cycle. These function by dynamically examining sets of instruc-

tions from the instruction stream to find ones capable of parallel execution on each cycle, and then executing them, often out of order with respect to the original program [1]. Instruction level parallelism is an implicit parallel execution technique. Instructions are executed concurrently and out of order, while maintaining the illusion that all instructions are being executed sequentially and in order. There are limits to the amount of usable parallelism in typical instruction streams [4] and these limits have been reached in modern hardware.

The memory wall represents a persistent disparity between memory latency improvements and memory bandwidth improvements [5, 6]. One report on the landscape of parallel computing found that out of thirteen broad classes of computational problems, the memory wall was the major obstacle to good performance for almost half of these classes [2]. A separate report on the effects of memory latency and bandwidth on supercomputer application performance found that doubling the memory latency results in an average drop in performance of 11% for floating point benchmarks and 32% for integer benchmarks [7]. Although the term 'memory wall' was initially applied to differences in both bandwidth and latency, the flat performance of memory latency has come to dominate the memory wall. Over the past two decades, while processor performance has doubled every 18 months, memory latency has improved only by about 7% a year [8]. Furthermore, in the time that it takes for memory bandwidth to double, latency improves by no more than a factor of 1.2 to 1.4 [9].

David Patterson says about the various walls: "The power wall + the memory wall + the ILP wall = a brick wall for serial performance" [2, 10]. The major microchip manufacturers such as IBM, Sun, and Intel have all moved away from relying only on instruction-level parallelism and are moving towards utilizing thread-level parallelism and data-level parallelism as well. While the implications of the power wall and the ILP wall lie primarily in the hardware domain, the implications

of the memory wall can be seen in the design of efficient algorithms for concurrent applications.

The thesis of this dissertation is that cache-conscious concurrent data structures for many-core systems will show significant performance improvements over the state of the art in concurrent data structure designs for those applications that must contend with the deleterious effects of the memory wall. Cache-conscious, linearizable concurrent data structures have advantageous performance that can be measured across multiple architecture platforms. The improved performance arises from better treatment of the memory wall phenomenon that is ubiquitous to current multi-core systems and almost certainly will continue to affect future many-core systems.

Lock-free cache-conscious data structures that maintain the abstraction of a linearizable unordered set have been described in the literature. The additional constraint of maintaining sorted order poses novel research questions in the construction of efficient lock-free cache-conscious data structures. A cache-conscious design favors the organization of data in contiguous regions in order to increase spatial locality. The lock-free approach eschews global balancing requirements in favor of atomic update operations. An important contribution of this dissertation is that it fills the gap in cache-conscious concurrent data structures by providing concurrent algorithms that implement an ordered set abstract data type. A cache-conscious unordered set stores elements with equal hash codes near each other but many possible orderings are valid. Maintaining sorted order imposes the constraint of exactly one valid logical ordering of elements in the set.

The two primary design contributions of this dissertation are the novel lock-free skip tree and lock-free burst trie algorithms. Both address the goal of developing cache-conscious concurrent data structures for those applications whose working set size exceeds the cache size. The lock-free skip tree uses comparison-based

sorting to provide expected cost $O(\log n)$ add, remove, and contains operations in the absence of concurrent operations. The lock-free burst trie uses radix-based sorting to provide $O(|e|)$ operations, where $|e|$ is the length of the input element. In both algorithms, read-only operations are wait-free operations and modification operations are lock-free operations.

The measures of evaluation are composed of a series of synthetic and application benchmarks. These benchmarks are used to assess the behavioral characteristics of the data structures. Neither synthetic nor application benchmark is adequate by itself; the two types of benchmarks serve complimentary roles. A synthetic benchmark measures the performance of the implementation without confounding factors such as application execution. An application benchmark serves as a single data point on the overall performance of complex tasks that rely on the data structure. Synthetic benchmarks allow various data structure implementations to be conveniently compared against one another. Application benchmarks give a measure of expected performance when used in a productive context, but the measure must be considered within the context of the specific application.

The measures of success are defined as reaching significant improvements for three metrics in the measures of evaluation. The three metrics are throughput, scalability, and memory footprint. Throughput is a measure of system-wide completion for a fixed task and a fixed set of resources. Scalability can be defined as a derived measure of throughput where the number of processors can vary independently. The memory footprint can be estimated using the maximum memory consumption or average memory consumption of a fixed task.

The lock-free skip tree and lock-free burst trie behave as a best in class design on the measures of success and outperform the best known implementations of related data structures. A series of synthetic benchmarks show that our lock-free skip tree and burst trie implementations perform up to x2.3 and x3.5 faster in read-

dominated workloads compared to the state of the art lock-free skip list with only a 13% maximum penalty across all workloads. An analysis of heap utilization of the data structures in the synthetic benchmark reveals that the lock-free skip tree uses 59% of the heap utilization of the skip list and the lock-free burst trie uses 140% of the skip list heap utilization. In a series of four parallel branch-and-bound applications, two of the applications are x2.3 and x3.1 faster when using the skip tree as a concurrent priority queue as compared to the lock-free skip list priority queue. The relative performance improvements for both data structures and decreased heap utilization of the lock-free skip tree is reported on multicore SPARC and x86 architectures. In a shared-memory supercomputer architecture the two branch-and-bound applications are x1.6 and x2.1 faster with the skip tree versus the skip list running at 80 hardware threads.

We have made the lock-free skip tree implementation available online [11]. The source code for the lock-free skip tree has been released to the public domain, as described at `http://creativecommons.org/licenses/publicdomain`. The Java Specification Request (JSR) 166 group on concurrency utilities releases all of its implementations into the public domain. We have implemented the ConcurrentSkipTreeMap and ConcurrentSkipTreeSet as drop-in replacements for the ConcurrentSkipListMap and ConcurrentSkipListSet data structures of the java.util.concurrent library. The intent is not for the skip tree implementations to replace the skip list implementations of the concurrency library. Rather the lock-free skip tree is intended to complement the lock-free skip list implementation in order to offer performance advantages under workloads characterized in this dissertation.

1.1 Background & Motivation

Concurrent data structure design in shared memory architecture is driven largely by the need for scalability. Any sequential segment of a concurrent algorithm quickly becomes a sequential bottleneck as the number of processors increases. If b is the fraction of the program that is subject to a sequential bottleneck, then Amdahl's law states that the maximum speedup that can be achieved on a machine using P processors is $1/(b + (1 - b)/P)$. For example, if 10% of a concurrent application is subject to a sequential bottleneck, the best possible speedup we can achieve on an 8-way machine is about 4.7, on a 16-way machine is about 6.4, and on a 64-way machine is only about 8.8. In order to meet increased scalability demands, the primary synchronization model for concurrent data structures has migrated from coarse-grained locking protocols to fine-grained locking protocols and finally to lock-free synchronization. Lock-free data structures implement concurrent objects without the use of mutual exclusion. Mutual exclusion resource protection schemes can suffer from performance concerns such as priority inversion, lock convoying, and lock contention and liveness concerns such as starvation and deadlock. Lock-free synchronization is achieved using a set of simple atomic synchronization primitives. There are multiple atomic primitives that are available on modern hardware instruction sets, and some atomic primitives are known to be more powerful than others [12]. However, all the synchronization primitives share the common property that they operate on one unit of memory at a time. Atomic instructions that operate on more than one memory unit have been proposed for future instruction sets, such as double compare-and-swap operations [13] or Isotach networks [14], but this area is currently a topic of research and the merits of these instructions are still being actively debated [15, 16].

Latency penalties for accessing shared memory in a multi-core processor are so effective at reducing program performance that they can be used as a basis for a

new class of denial of service attacks [17]. Memory access schedulers are designed to maximize the bandwidth obtained from the DRAM memory. Requests from a thread with a particular access pattern can get prioritized by the memory access scheduler over requests from other threads, thereby causing the other threads to experience very long delays. The processor to memory performance gap, which is already approaching a thousand cycles, is expected to grow by 50% per year according to some estimates. At the same time, the number of cores on a single chip is expected to continue to double every 18 months, and since limitations on space will keep the cache resources from growing as quickly, cache per core ratio will continue to go down [18].

In the era of vector and specialized shared memory supercomputers, it was quickly realized that a major limitation to the full utilization of the machine was the inability to get data to the processing units fast enough to keep the units busy [19, 20]. Program transformation techniques were applied to regular computational patterns (loops) in order to reduce the number of cache misses. Among these program transformations are loop fusion, loop distribution, loop interchange, loop unrolling, loop skewing, strip mining, etc. [21, 22]. Motivated by the improvements in cache behavior of loop structured programming, researchers began constructing techniques to improve the locality of irregular pointer-based applications. Such pointer-based applications rely on recursive data structures such as linked lists, trees, and graphs, where individual nodes are dynamically allocated and nodes are linked together through pointers to form the overall structure. Software prefetching techniques have been adapted to tree structures by prefetching children nodes when a parent node is visited [23]. Nodes likely to be accessed contemporaneously can be clustered into a cache block. Recursive data structures possess locational transparency; elements in a structure can be placed at different memory locations without changing a program's semantics [24].

Chapter 2 reviews consistency models, synchronization models, and related concurrent data structures. Consistency models that are reviewed in this chapter are strict consistency, sequential consistency, quiescent consistency, linearizable consistency, serializability, strict serializability, and transactional memory semantics. The limitations of mutual exclusion as a synchronization model are reviewed, and a history of lock-free algorithm design is outlined. All modern lock-free data structures use lock-free linked lists and arrays as building blocks in the construction of more complex algorithms. We trace the evolution of an efficient lock-free node deletion algorithm. The relative difficulty in constructing an efficient lock-free deletion algorithm as compared to an insertion algorithm is a theme that will be encountered several times in this document. We review the state of the art in lock-free skip list and lock-free concurrent hash table design. Many data structures have been designed with concurrent, lock-free, cache-conscious, or randomized properties. Each of the related data structures shares one, two, or three, but not all four properties of the lock-free skip tree design.

1.2 Lock-free Skip Tree

The lock-free skip tree algorithm is a transformation of Messeguer's skip tree [25]. It meets the dual criteria of effective use of the memory hierarchy and employment of atomic compare-and-swap operations. The majority of nodes in the original skip tree contain no elements. The purpose of these empty nodes is to maintain a path length invariant. Our first objective in designing a cache-conscious isomorphism of the skip list was to eliminate the empty nodes from the skip tree, thus the *dense* skip tree. We show that the number of keys per node of a dense skip tree is distributed according to a geometric distribution, and that the expected height of the tree of n keys is bounded by $\log n$. We are then able to show that the expected time for

sequential search, insert, and delete operations is $O(\log n)$. The proofs are novel to the dense skip tree design, and serve as a completion of the skip tree proofs that are incomplete in the literature. Our proof techniques represent a novel departure from the methods outlined in the original skip tree paper.

Prior to the construction of a lock-free concurrent algorithm, an initial optimistic concurrent dense skip tree algorithm is constructed. This algorithm uses an optimistic concurrency control technique where the necessary components of the data structure are locked on the assumption that components maintain a consistent state during the time period in which the locks are acquired. The optimistic concurrency technique provides a clear concurrency model, yields a relatively straightforward correctness proof, and performs well under workloads that are dominated by read-only operations. The optimistic skip tree algorithm fulfilled two objectives necessary for the construction of a lock-free skip tree algorithm. First, it provided insight into transforming a sequential skip tree algorithm into a concurrent skip tree algorithm. Second, a series of synthetic benchmarks showed the potential of the skip tree design as a concurrent cache-conscious data structure. When the working set size exceeds the cache size, the peak throughput of the optimistic skip tree is 144% and 155% relative to the lock-free skip list in a read-dominated synthetic workload on SPARC and x86 platforms. Of equal importance, the peak throughput of the optimistic skip tree is 73% and 85% relative to the lock-free skip list in a read-dominated synthetic workload for a small working set size. The results of the synthetic benchmarks suggest that it is possible to construct an efficient lock-free cache-conscious data structure that maintains the abstraction of a linearizable ordered set.

The lock-free skip tree algorithm is defined in Chapter 4. The lock-free skip tree has relaxed structural properties that allow atomic operations to modify the tree without invalidating the consistency of the data structure. The lock-free skip tree

definition has two primary differences from the dense skip tree definition. First, we introduce link references to allow nodes to split independently of their parent nodes. Second, we relax the requirement that non-leaf nodes behave as partitions on the tree. The data structure maintains consistency by defining a *reachability* relation from the root of the tree to any potential element stored in the tree for all possible states of the tree. As highlighted in the related work on lock-free data structures in Chapter 2, the bulk of the complexity in the lock-free skip tree algorithm is found in the deletion algorithm as compared to the insertion or search algorithms. Optimal paths through the tree are temporarily violated by deletion operations and eventually restored using online node compaction.

The lock-free skip tree implementation outperforms established algorithms, namely a highly-tuned lock-free skip list, a relaxed balance AVL tree, and a B^{link}-tree on synthetic benchmarks across different thread counts, operation mixes, and machine architectures when the working set size cannot be contained in cache. The peak throughput of the lock-free skip tree is 229% and 198% relative to the lock-free skip list on the read-dominated workloads with 5,000,000 elements on SPARC and x86 platforms. The lowest peak throughput of the lock-free skip tree relative to the skip list across all synthetic benchmarks is 87%. The synthetic benchmarks have shown that a lock-free cache-conscious data structure can perform up to x2.3 faster in some workloads compared to the state of the art with only a 13% maximum penalty across all workloads.

1.3 Application Benchmarks

Benchmarks that perform synthetic operations should be interpreted as measurements of performance that lack semantic context or purpose. Their results can be interpreted by individual developers to estimate the utility of the algorithm for

their needs. The disadvantage of synthetic operations is their lack of semantic context. As such, some caution must be exercised when projecting the results from the synthetic benchmarks onto a specific application domain.

In order to ensure balance in our analysis of the lock-free skip tree we identified a class of NP-hard problems that can be used to characterize the relative merits of the lock-free skip tree as compared to the lock-free skip list. These applications rely heavily on a linearizable data structure that preserves a sorted set or sorted map abstraction.

We identified four NP-hard problems that can be solved using a parallel branch-and-bound solver with a centralized concurrent priority queue. These problems are the N puzzle, the graph coloring problem, the asymmetric traveling salesman problem (ATSP), and 0-1 knapsack. In addition to the four NP-hard problems, we created a synthetic branch-and-bound application in order to test three hypotheses on the effects of specific properties of branch-and-bound applications on the relative performance of the lock-free skip tree versus the lock-free skip list. The three hypotheses are: (1) the distribution of lower bounds of the candidates in the search space affects the performance of the skip tree; (2) the computation time of the lower bound affects the performance of the skip tree; and (3) the branching factor of the application affects the performance of the skip tree. Based on the outcomes from the four application benchmarks and the synthetic benchmark, we provide a set of guidelines for selecting the lock-free skip tree to use as a centralized priority queue in a parallel branch-and-bound application versus the lock-free skip list.

The branch-and-bound applications were tested on a sixteen core Sun Niagara workstation, a quad core Intel Xeon workstation, and two shared memory supercomputers with different hardware instruction sets, network interconnections, and software runtime architectures. The lock-free skip tree shows an improvement in runtime of up to x2.4 and x3.1 on the N puzzle and graph coloring applications

relative to the lock-free skip list. Over 99% of the total runtime of the asymmetric traveling salesman problem solver is spent in the computational phase that calculates the lower bound estimate for each partial solution. The dominance of the computational phase precludes any significant benefit from selecting the lock-free skip tree using the Held-Karp [26, 27] lower bound estimation for solving ATSP. We use a primal-dual algorithm [28] for solving the knapsack problem. The primal-dual algorithm uses an inconsistent heuristic function. Consistent heuristic functions are general strategies for traversing through a state space that approach the solution state without taking any backward steps [29]. The knapsack is allowed to overfill so that successor states may remove elements from the knapsack. When applied to a breadth-first parallel branch and bound solver, the inconsistent heuristic function results in contention for the head of the priority queue.

We show that it is possible to use shared-memory supercomputers efficiently to solve parallel branch-and-bound problems. The Azul compute appliance is a custom shared memory supercomputer designed for the Java runtime environment. The processing unit of the compute appliance is a Vega 3 processor, a 54 core 64-bit RISC processor. Up to 16 Vega processors can be installed on a compute appliance, for a total of 864 hardware threads. In addition to the standard RISC instruction set, the Vega processor has a few specialized instructions to aid the Java virtual machine in object allocation and garbage collection. The SGI Altix UV 1000 consists of 256 blade servers connected by a NUMAlink® 5 Interconnect. Each blade holds 2 Intel Xeon X7560 processors with 8 hardware threads per processor, for a total of 4096 hardware threads on the machine.

For those parallel branch-and-bound applications that exhibit the set of properties we have characterized for selecting the lock-free skip tree as a centralized priority queue, on an Azul compute appliance the skip tree shows up to a x2.1 improvement in runtime as compared to the lock-free skip list when running on

88 hardware threads. In order for an application to scale across a shared-memory interconnection communication layer, it is necessary for all the runtime layers underneath the application to scale as well. The Azul compute appliance runs on top of a minimalist operating system and has its own Java virtual machine implementation that is based on the OpenJDK project. The SGI Altix UV 1000 runs a modified 2.6.32.12 Linux kernel and a patched version of the OpenJDK 7 release. Given the commercial existence of a Java compute appliance that can scale to hundreds of threads using specialized hardware, a specialized operating system, and a specialized Java virtual machine, it is most likely that significant modifications would be necessary to achieve the same scalability on a conventional hardware and software stack.

1.4 Lock-free Burst Trie

We study whether it is possible to improve upon the $O(\log n)$ expected cost in designing a lock-free cache-conscious data structure that implements an ordered set. The lower bounds for comparison-based sorted data structures are $\Omega(\log n)$. In order to improve upon these lower bounds, we must use a radix sorting scheme as the basis of the data structure. We define the lock-free burst trie algorithm which is the first of its kind. The lock-free trie offers a space/time tradeoff as compared to the lock-free skip tree. Interior nodes of the lock-free trie store unused children references in order to allow for a search through the tree that does not rely on element comparison operations.

A series of synthetic benchmarks was studied to measure the throughput of the concurrent trie as compared to concurrent skip list, skip tree, and B^{link}-tree implementations. The burst trie exhibits the highest peak throughput across all scenarios and architectures. The mean peak throughput of the burst trie is x3.5

higher than the data structure with the second highest peak throughput across all scenarios on the Sun Fire T1000 and x2.8 higher across all scenarios on the Intel Xeon L5430. The heap utilization of all four data structures was measured for the write scenario with five million elements on the Sun Fire T1000. The skip tree and B^{link}-tree use an average of 35 bytes per element, the skip list uses an average of 60 bytes per element and the burst trie uses an average of 84 bytes per element. The lock-free burst trie inherits both desirable and undesirable characteristics from its parental data structure designs, the search tree and the hash table. The trie maintains an ordered set abstraction and exhibits a relatively higher throughput than the lock-free skip tree on the synthetic benchmarks. In order to benefit from the advantages of the lock-free trie it is necessary to define a prefix function on the input domain in order to use the data structure. The tuning of three configuration parameters plays an important role in the performance of the lock-free burst trie.

A primary objective of this research is to reconcile the algorithmic impacts of two trends in microprocessor design that each encourages thinking about a problem in opposing views. The first trend is the shift from implicitly parallel processors towards explicitly parallel processors. To achieve scalability on explicitly parallel processors, blocking synchronization must be reduced or eliminated. Each critical section constructs a sequential roadblock to parallel performance. The second trend in microprocessor design is the deepening of the memory hierarchy. As more cores are packed onto a processor, more transistor real estate is spent on cache memory to keep the cores busy with sufficient data. Lock-free algorithms concentrate on atomic primitives that operate on either one or a handful of words at a time. Cache-conscious algorithms take into consideration cache size, line size, associativity, and degree of sharing among cores. Nonblocking synchronization algorithms use a model of the memory hierarchy in the small. Algorithms that mitigate the effects of the memory wall use a model of the memory hierarchy in

the large. We focus on reconciling the views of the memory hierarchy in the small and in the large for the purpose of constructing nonblocking cache-conscious concurrent algorithms.

Chapter 2

Related Work

This chapter reviews relevant background material on memory consistency models, synchronization models, and related concurrent data structures. A memory consistency model dictates how memory behaves with respect to read and write operations from multiple concurrent processes [30–32]. In a uniprocess system, a read should return the value of the "last" write to the same memory location, where "last" is precisely defined by program order, i.e., the order in which memory operations appear in the program. In a multiprocessor system, the memory consistency model must address the following questions regarding the passage of time: (1) Do all processes see the same order of events? (2) Do all processes see the correct order of events? We will review strict consistency, sequential consistency, quiescent consistency, linearizable consistency, serializability, strict serializability, and transactional memory semantics. In the next section, we will motivate why linearizable consistency was selected as the consistency model for our research.

A synchronization model is defined as a set of constraints on memory accesses that specify how and when synchronization needs to be done [33]. The traditional model for synchronizing access to shared resources has been to use mutual exclusion. We will discuss some of the disadvantages of mutual exclusion and review

the most popular non-blocking synchronization properties: lock-freedom, wait-freedom, and obstruction-freedom. Non-blocking synchronization allows synchronized code to be executed in an interrupt or (asynchronous) signal handler without danger of deadlock. Non-blocking synchronization minimizes interference between process scheduling and synchronization. A high priority process can access a synchronized data structure without being delayed or blocked by a lower priority process. Non-blocking synchronization aids fault-tolerance. It provides greater insulation from failures such as fail-stop processes failing or aborting and leaving inconsistent data structures. Finally, non-blocking synchronization can reduce interrupt latencies. Non-blocking synchronization first drew attention in operating systems research, where its properties were used to construct robust, scalable systems. Later, the synchronization model was adopted for concurrent application design. In Section 2.2, we will motivate why wait-freedom was chosen for read-only operations and lock-freedom for update operations.

There exist many data structures that have been designed for concurrent, cache-conscious, or randomized applications. The techniques used in this dissertation for the design of cache-conscious concurrent data structures expand upon the concepts used for the construction of the concurrent skip list and the concurrent B-tree. Section 2.3 reviews skip lists, skip trees, treaps, randomized search trees, cache-oblivious B-trees, and cache-conscious hash tables. The first lock-free concurrent data structures were linked lists. Lock-free deletion from a linked list was solved by the Maged-Harris algorithm [34, 35]. A node is first logically deleted from the linked list using an atomic operation and then subsequently it can be physically removed from the list. The lock-free skip list is constructed as a set of lock-free linked lists stacked on top of one another. From the concurrent B-tree, the use of link references to travel horizontally through a multiway search tree allows concurrent readers to find multiple paths to reach a target element in the pres-

ence of concurrent modifications to the tree. We review the literature of lock-free cache-conscious hashing algorithms in order to highlight the advances that have been accomplished in the design of cache-conscious concurrent data structures. In Chapter 6 we will show how to construct a cache-conscious lock-free hash function that preserves the natural ordering of the elements contained in the abstract data type.

2.1 Consistency Models

A memory consistency model dictates how memory behaves with respect to read and write operations from multiple concurrent processes. The consistency model of a shared-memory multiprocessor provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system [30–32]. Effectively, the consistency model places restrictions on the values that can be returned by a read in a shared-memory program execution. Each consistency model offers a unique approach to answering the following two questions concerning the passage of time: (1) Do all processes see the same order of events? (2) Do all processes see the correct order of events? In this section we will discuss the following consistency models: strict consistency, sequential consistency, quiescent consistency, linearizable consistency, serializability, strict serializability, and transactional memory semantics. Our research focuses on the design of concurrent data structures that assume the linearizability consistency model. This section serves to explain why linearizability has been chosen as the target consistency model.

The simplest consistency model is known as strict consistency, and it requires that a read operation on a location returns the result of the last write operation

which occurred at that location. In a programming model that contains exactly one process, the last write operation can be defined using the order of write operations in the program. In a multiprocessor system, strict consistency would require a global clock and would place significant performance burdens on the memory hierarchy. The strict consistency model provides the most straightforward answers to the two questions regarding the passage of time. But as strict consistency is an impractical consistency model for a multiprocessor architecture, the next simplest consistency model addresses only the first question on the passage of time and ignores the second question on the correct order of events.

The next simplest and the most familiar consistency model for shared memory multiprocessors is sequential consistency. A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [36]. Sequential consistency is an intuitive model for multiprocessor systems. Each process has its own local clock and the communication delays between processors are variable or otherwise unpredictable. This model enforces program order within each individual processor, and it allows all processors to assume they are observing the same order of events. The sequential consistency model is equivalent to the abstraction where memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Using this abstraction, sequential consistency provides both an ordering of events and a mechanism for specifying the atomicity of events.

The sequential consistency model does not address the second question regarding the passage of time: "Do all processes see the correct order of events?" In contrast, the quiescent consistency model is defined such that the operations of any processors separated by a period of quiescence should appear to take effect in

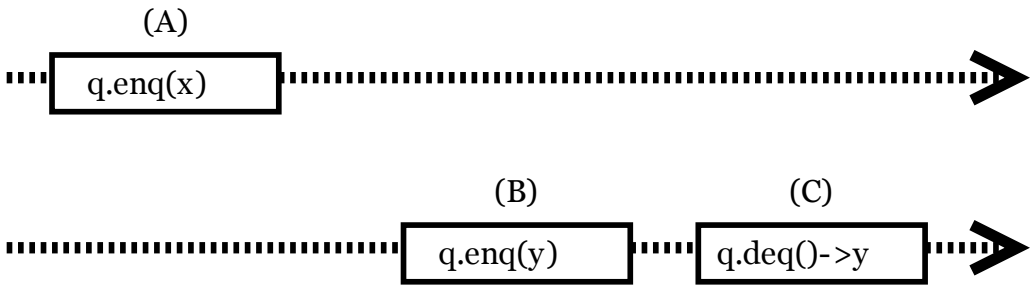


Figure 2.1: A concurrent queue data structure. (A) One process enqueues x. (B) A second process enqueues y. (C) The second process performs a dequeue operation which returns y. This history is permitted by sequential consistency, can be either permitted or forbidden by quiescent consistency, and is forbidden by linearizable consistency.

their real-time order [37, 38]. Quiescence guarantees are useful in shared memory systems where communication delays can be bounded in the absence of system failures. Quiescent consistency ignores the first question of consistency models regarding the passage of time and focuses on providing an answer to the second question. An example of a scenario that is allowed in the sequential consistency model and disallowed in the quiescent consistency model is shown in Figure 2.1. Two processes share a concurrent queue data structure. The first process enqueues x. At some non-overlapping subsequent interval, a second process enqueues y. Finally the second process performs a dequeue and receives y. This example is sequentially consistent but is not quiescently consistent, assuming that the time between enqueue operations falls outside the quiescence interval. Quiescent consistency is a compositional consistency model. A consistency model is compositional if and only if the specification of every object in a system satisfies the consistency model implies that the system as a whole satisfies the consistency model [39]. Sequential consistency is not a compositional consistency model. Quiescent consistency is useful for systems that assume a compositional consistency model, but quiescent consistency does not preserve program order.

Linearizable consistency is the weakest consistency model that preserves program order among individual processes and satisfies compositionality. A system

is linearizable if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program, and any non-overlapping operations appear in the same order in the sequential history as they appear in the execution [40]. An equivalent definition of linearizably consistency requires that all function calls have a linearization point at some instant between their invocation and their response and all functions appear to occur instantly at their linearization point, behaving as specified by a sequential definition. Two overlapping operations in a linearizable system can occur in either order in the corresponding sequential history, but two non-overlapping operations must appear in the same order in the concurrent and sequential histories.

A transaction is defined as a sequence of operations that are executed atomically. Linearizable consistency can be viewed as a special case of strict serializability where transactions are restricted to those consisting of a single operation applied to a single object [40]. Serializable consistency is an extension of the sequential consistency model, where the result of any execution is the same as if the transactions of all the processors were executed in some sequential order. Strict serializability adds the restriction that any non-overlapping transactions appear in the same order in the sequential history as they appear in the execution. Serializability is a stronger consistency model than linearizability. The serializable model allows programmers to reason about transactions as if they were sequential programs. Serializability and strict serializability are not composable consistency models, and both models are inherently blocking concurrency models. A property is inherently blocking if it can be enforced only by blocking a transaction's data access operations until certain events occur in other transactions [41].

Transactional memory refers to a family of concurrency control mechanisms that aims to support serializable consistency semantics in main memory. Support

for transactional memory was first proposed in hardware [42] and then later extended to software-only implementations [43]. Transactional memory semantics provide failure atomicity, consistency, and isolation. All the proposed transactional memory models start with the atomic block construct. An atomic block is required to either execute to completion or to appear not to have executed at all. The execution of a transaction cannot affect the result of other concurrently executing transactions. Transactional memory can offer either strong or weak atomicity semantics [44]. Weak atomicity guarantees transactional semantics only among transactions. Strong atomicity guarantees transactional semantics between transactions and non-transactional code. A program that is safe in one atomicity model is not necessarily safe in the other model. Transactional memory semantics are often augmented with transaction coordination primitives [45]. A transaction that executes a `retry` statement aborts and then re-executes. The re-execution can be delayed until one or more values read in the previous execution are changed. The `orElse` operation allows the execution of one transaction to be conditional on the execution of a second transaction. Hardware transactional memory systems impose strict limitations on program execution, while software transactional memory that supports strong atomicity offers weak execution performance. A comprehensive review of the state of the art in the design and implementation of transactional memory systems, as of early summer 2006, has been written by Larus and Rajwar [46]. The correct semantics for transactional memory models are an active area of research. For the scope of our research, we have limited ourselves to non-transactional memory semantics. Linearizability is appropriate for applications in which concurrency and composability are of primary interest.

2.2 Synchronization Models

The traditional model for synchronizing access to shared resources has been to use mutual exclusion. However there are several disadvantages associated with mutual exclusion. Mutual exclusion (also known as locking) causes blocking of threads or processes that must wait until a lock is released. Locks can cause deadlock, due to the early termination of a thread without releasing a lock, or due to subtle defects in the concurrent algorithm design. Undesirable effects such as priority inversion or convoying may occur [47]. Finally, lock contention limits scalability in highly-concurrent environments as lock-management becomes a sequential bottleneck, as an application of Amdahl's law. In order to sidestep these issues, a series of non-blocking properties has been developed for concurrent data structures. The most widely-used non-blocking properties are lock-freedom, wait-freedom, and obstruction-freedom.

The most popular non-blocking property is lock-freedom. A lock-free implementation of a concurrent data object is one that guarantees that some operation is always making progress towards completion regardless of any other concurrent operations in the system [48]. The lock-free property guarantees that some operation will complete after a finite number of steps. A wait-free implementation of a concurrent data object guarantees that all operations are always making progress towards completion [48]. The wait-free property guarantees that all operations must complete after taking a finite number of steps. The wait-free property yields guaranteed system-wide throughput and starvation-freedom, while the lock-free property allows individual threads to starve but guarantees system-wide throughput. A lock-free data structure will typically have a lower computational overhead than its wait-free counterpart, and will rely on well-known contention management techniques such as backoff [49] in order to achieve practical progress guarantees. Obstruction-freedom is a more recent non-blocking property that guarantees

that some operation will complete after a finite number of steps, provided that all other operations are suspended [50]. Obstruction-freedom is strong enough to avoid the problems associated with locks, but it is weaker than previous non-blocking properties. Obstruction-freedom allows for simple implementations of concurrent algorithms, with the hope that these implementations can be made as efficient as lock-free or wait-free implementations.

Numerous universal construction techniques have been published on automatically converting a sequential data structure into a lock-free or wait-free data structure. A comprehensive summary of these techniques can be found in Fraser's dissertation [51]. Yet these universal constructions exhibit poor performance in practice when compared to human designed non-blocking algorithms. The most efficient of these universal constructions require either nestable lock-linked/store-conditional primitives, or double compare and swap primitives, or scheduler activation support.

The scalability of non-blocking algorithms has made them desirable in several modern concurrency libraries. The Java 6 concurrency library includes a wait-free queue (based on the algorithm of Michael and Scott [52]), a hash table with an arbitrary number of concurrent readers and a tunable number of concurrent writers, and a lock-free skip list written by Lea [53] extended from a paper by Fraser and Harris [54]. Microsoft's .NET Framework currently supports atomic operations for shared variables, allowing operations such as read & write registers, swap, and compare and swap. The next version of .NET Framework concurrency library will contain implementations of lock-free stack and queue algorithms. The Intel Threading Building Blocks (TBB) library is a runtime-based parallel programming model for C++. The TBB library offers concurrent linearizable vector, queue, and hash table implementations [55]. Several studies indicate that non-blocking data structures can provide superior performance to traditional implementations using

mutual exclusion, such as the study by Michael and Scott [56] of concurrent data structures in microbenchmarks and real applications, or the study by Lumetta and Culler [57] of concurrent access to message queues.

2.3 Concurrent Data Structures

There exist many data structures that have been designed with concurrent, cache-conscious, or randomized properties. Many related data structures exhibit one or two, but not all three properties. The data structures reviewed in this section are summarized in Table 2.1 and their properties of interest are enumerated.

2.3.1 Skip Lists, Linked Lists, and Skip Trees

The sequential skip list [58] and the concurrent skip list [59] were invented by Bill Pugh. A skip list is a linked list data structure where each node contains some number of express lanes that skip ahead into the list. The height of an express lane determines how many nodes to skip ahead. An express lane of height h skips ahead 2^h nodes. When a node is inserted into the tree, it is assigned a height from a geometric probability distribution. A node of height h contains express lanes for heights $\{0, 1, 2, \ldots, h-1, h\}$. The skip list has an amortized cost of $O(\log n)$ for

Data Structure	Concurrent	Cache-conscious	Randomized	Sorted Order
Lock-free skip list	Y	N	Y	Y
Treap/Randomized search tree	N	N	Y	Y
Cache-oblivious B-tree	Y	N	Y[a]	Y
Lea ConcurrentHashMap	Y	closed address	N	N
Purcell and Harris Hash Table	Y	open address	N	N
Hopscotch Hash Table	Y	open address	N	N
HAT-trie	N	Y	N	Y

[a] randomized cache-oblivious B-tree algorithm does not support deletions

Table 2.1: Properties of Related Data Structures

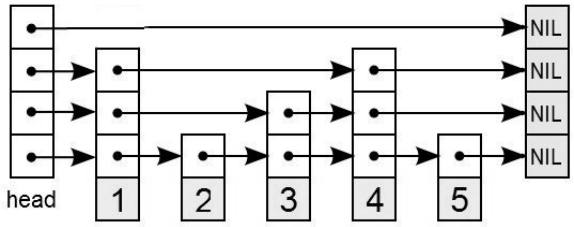


Figure 2.2: Skip list example

search, insert, and delete operations. An example skip list is shown in Figure 2.2. The local balancing of the skip list has made it an attractive data structure in a variety of distributed applications, such as constant-degree routing networks [60], peer-to-peer filesystems [61, 62], and concurrent lock-free ordered sets, ordered maps, and priority queues [51, 63–65].

The first lock-free concurrent data structures were linked lists. As many subsequent lock-free algorithms adapted techniques from the lock-free linked list, we provide a basic introduction to the lock-free linked list. Valois [66] defined a lock-free linked list algorithm using the compare-and-swap (CAS) synchronization primitive. Each node of the linked list stores one or more fields of data and a `next` field that stores a reference to the next node in the list. Insertion into the linked list is a straightforward process. To insert a new node at the successor of some node `x`, a new node `y` is created such that `y.next` is assigned the value of `x.next`, and then a CAS is performed on `x.next` to update its value to `y`.

Deletion of elements is a trickier process. An attempt to remove `y` by performing a CAS on `x.next` to update its value to `y.next` can lead to the accidental deletion of the successor of `y` that is inserted after the value of `y.next` is read. Valois solves the node deletion problem with auxiliary nodes that maintain consistency of the linked list and contain no actual data. Every regular node in the linked list must have an auxiliary node as its predecessor and successor. Harris [34] provides an alternate construction of a lock-free linked list that avoids the wasted space of auxiliary nodes. A node is first logically deleted from the linked list using an

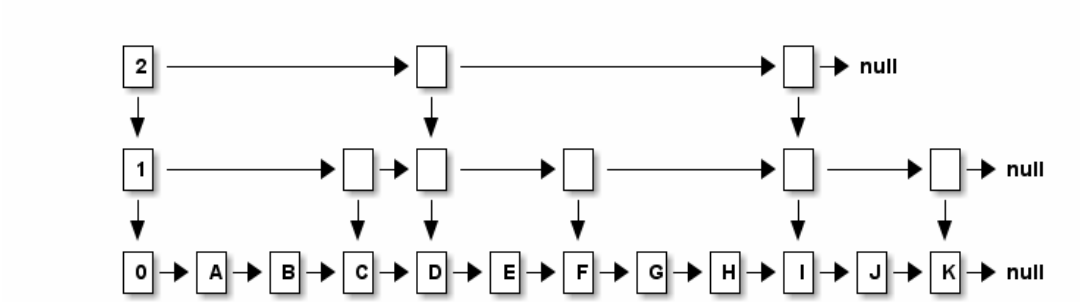


Figure 2.3: Doug Lea's lock-free skip list. Figure adapted from java.util.concurrent ConcurrentSkipListMap documentation [53].

atomic operation and then subsequently it can be physically removed from the list. The `next` field contains both the marked or unmarked state or a node and a reference to the next node in the list. Using this design, a marked node cannot have a new node inserted as its successor. Michael [35] presents a similar design for the lock-free linked list, with the addition of a tag field that is useful in runtime systems that require explicit memory management. This separation of concerns of the logical removal of an element from the abstract set and the physical removal of an element from the data structure is repeated in subsequent lock-free data structure designs.

The skip tree was introduced by Messeguer [25] as a generalization of the skip list for concurrent operations. A skip tree is a randomized multiway search tree such that each new element is assigned a height from a geometric distribution. Elements are stored in the leaves of the tree, and the interior nodes of the tree serve as partitions. An example skip tree is shown in Figure 2.4. The isomorphism between the skip list and the multiway search tree has been noted in other publications [67–70]. Messeguer defines the skip tree with the requirement that all paths from the root to the leaves are of identical length. To enforce this requirement, empty nodes are inserted into the tree that contain no data. These empty nodes are similar to the auxiliary nodes in Valois' lock-free linked list [66].

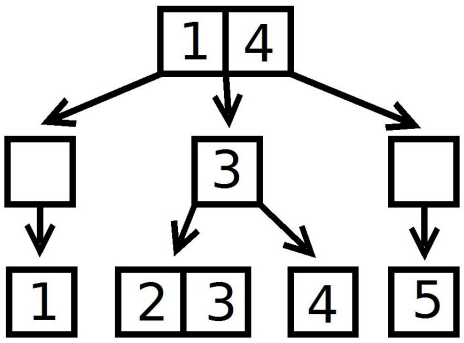


Figure 2.4: Skip tree example. Isomorphic to Figure 2.2.

The lock-free skip list implementation in the concurrency collections of the Java class library was written by Doug Lea with assistance from members of Java Community Process JSR-166 Expert Group and released into the public domain [53]. Unlike in a traditional skip list, the levels of a node are separated from each other. This separation allows for concurrent modifications to occur at different levels of the same node. An example of the segmented design is shown in Figure 2.3. Integer values on the left-hand side of the diagram indicate the level of the skip list, and letter values in the bottom level indicate elements of the container in sorted order. Element insertion begins with a traversal through the list to find the correct location at the leaf level. An element is a member of the abstract set if-and-only-if the element has been inserted at the leaf level of the list. Subsequent to the insertion at the leaf level, corresponding index nodes are added when the selected random height is greater than 0. A shifted geometric distribution is used such that approximately 75% of all heights are 0, and the remaining heights are generated by a geometric distribution with a probability of success at $1/2$.

Element deletion in the lock-free skip list uses a two-step scheme to distinguish between the tasks of removing an element from the abstract set and blocking additional elements from insertion onto a deleted node. These two tasks are typically combined into a single step. A boolean mark is attached to an atomic reference as described earlier in our review of Harris [34] and Michael [35]. An atomic mark-

able reference is implemented as boxed values at the language level, rather than the traditional method of bit stealing from address pointers. This leads to some space and time penalties associated with using boxed values. An element is logically deleted from the abstract set using a compare-and-swap on the value at a node from non-null to null. At the end of step one, additional nodes may be inserted at the tail of the deleted element. In the second step, the tail of the deleted element is replaced with a marker node. The marker node is used to signify that no more nodes can be inserted onto the tail of the deleted node. Eventually, the predecessor of the deleted node can replace its tail with the successor of the marker node.

2.3.2 Treaps, Randomized Search Trees, and Cache-Oblivious Trees

The treap and the randomized search tree are two related random binary tree data structures [69, 71]. In both cases, nodes are inserted in an in-order fashion by their keys, and are heap-ordered by their priorities which are randomly distributed. The treap selects random numbers from an unbounded uniform distribution, and the randomized search tree selects numbers from a uniform distribution bounded at the current number of keys. There are some similarities between the random binary trees and the dense skip tree, but the dense skip tree cannot be described simply as a treap or a randomized search tree implemented with a B-tree. The random binary tree uses uniform random priorities to create a binary tree that is probabilistically balanced. Likewise, the dense skip tree uses random heights to create a multi-way search tree that is probabilistically balanced. However, the skip tree uses a geometric distribution for the random heights in order for the number of keys per node to be a geometric distribution. A skip tree with a uniform distribution for the random heights would not be a cache-conscious data structure.

Cache oblivious algorithms are designed to perform asymptotically optimally

on a memory hierarchy of an unknown number of levels and line sizes [72]. Three concurrent cache-oblivious B-tree designs are proposed by Bender et al. [73]. One of the designs selects the height of a key from a random distribution. This design supports concurrent search and insert operations, but neither concurrent nor sequential delete operations. In addition, the probability distribution that a key is promoted from height $h-1$ to height h is $\frac{2^{-\alpha^h-1}}{\alpha-1}$, which is a function of h. Any cache-oblivious design has been shown to perform at best $(\log_2 e)(\log_B N)$ memory transfers per search operation under a 2-level memory model, where B is the cache-line size and N is the number of keys in the data structure [74]. The cache-oblivious B-tree is concurrent, randomized (with no deletions), and asymptotically optimal on an arbitrary memory hierarchy, but the lower bound of the search cost is approximately 44% higher than a cache-conscious design.

Numerous techniques have been developed for the cache-conscious access of data within the framework of a multilevel memory hierarchy. These techniques include loop transformation [75, 76], data coloring [24], garbage collection [77], dynamic profiling [78], cache-conscious structure layout [79–81], and cache-oblivious structure layout [82, 73]. Static methods for cache-conscious program transformation have traditionally performed well on dense matrix programs. Pointer-based structures are usually transformed into cache-conscious structures by enforcing spatial locality at the expense of memory fragmentation. One of the contributions of this dissertation is the introduction of a randomization technique that induces an expected cache-conscious access of data, but does not impose a size constraint on each component of the data structure and therefore does not suffer from internal memory fragmentation.

2.3.3 Concurrent Hash Tables

Three concurrent hash table algorithms will be reviewed in this section: Lea's ConcurrentHashMap, Purcell and Harris's non-blocking hash table with open addressing, and Herlihy et al.'s hopscotch hashing algorithm. These concurrent hash table algorithms are representative of the design choices available for scalable hash table implementations. Lea's ConcurrentHashMap is a part of the Java SE concurrency library [83]. The ConcurrentHashMap is a closed addressing algorithm and uses linked lists for the hash chains. Locks are striped across the bucket array, and the number of locks is specified at the construction of the data structure. The primary relevant feature of the ConcurrentHashMap is that only the values in a bucket chain are mutable. Modifying the keys of a bucket chain is only possible by modifying the reference to the head of the chain. The lock associated with the target hash bucket is always acquired by an insertion or deletion. An insertion operation creates a new list head if the key is missing from the chain. A deletion operation creates a new chain that is a clone of the old chain with the target element removed. The search operation traverses a chain without acquiring a lock. If the target key is found, then the value of the key is returned. If the key is not found, then the lock associated with the bucket is acquired and the search is repeated. The ConcurrentHashMap is designed such that successful search operations are wait-free operations. This feature along with the fine-grained locking protocol yields impressive performance for many concurrent applications.

Purcell and Harris published the first non-blocking hash table based on open addressing [84]. Each bucket stores an upper-bound on the number of collisions currently hashing to the same bucket. An insertion that increases the number of collisions increments the upper-bound, and a deletion that empties the last bucket in the collision sequence searches back for the previous collision and decreases the bound accordingly. A deletion that searches back for the previous collision enters

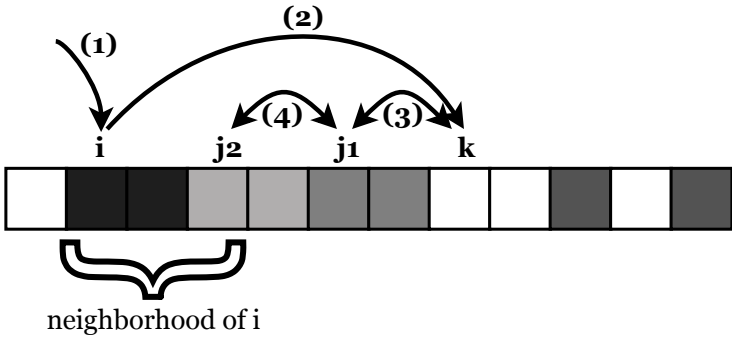


Figure 2.5: Hopscotch hash insertion. Items within the same neighborhood share the same shade of gray. (1) An item hashes to position i. (2) The first available location k is found. (3) Positions j_1 and k can be swapped because they lie in the same neighborhood. (4) Positions j_2 and j_1 can be swapped because they lie in the same neighborhood. The new item is inserted at j_2.

a so-called scanning phase to perform this operation. A scanning bit assigned to each bucket is used to ensure that at most one thread per bucket is in the scanning phase. A successful insert operation clears the scanning bit. A successful delete operation also clears the scanning bit. A delete operation that empties the last bucket in the collision sequence signals the beginning of scanning by setting the scanning bit. At the end of the scanning phase, if the scanning bit is true and the collision bound has its original value, then there have been no concurrent updates. Otherwise, the scanning phase is retried. As an open addressed lock-free hash table, this design has a low memory footprint (no link chains are created during runtime) and no sequential bottlenecks. The Purcell and Harris algorithm performs best at low table densities, in contrast to the next algorithm that continues to deliver good performance when the hash table is more than 90% full.

The hopscotch algorithm defines a class of concurrent cache-conscious resizable hash tables with $O(1)$ expected time for `contains`, `add`, and `remove` operations [85]. Hopscotch hashing uses an open addressed hashing scheme in which each bucket contains a neighborhood of buckets. The neighborhood of some bucket i consists of the next $H-1$ contiguous buckets, for some constant H. The cost of

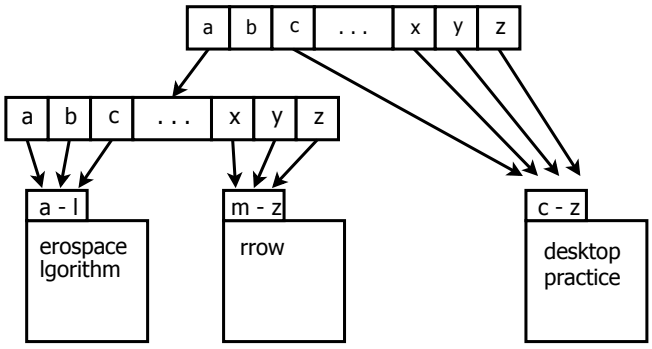


Figure 2.6: HAT-trie data structure. Traveling from one trie node to another consumes a single letter. The figure is adapted from Askitis and Sinha [86].

finding some element in a bucket is identical or very close to the cost of finding the element in that bucket's neighborhood. Note that nearby buckets have partially overlapping neighborhoods. This observation is critical to the design of the hopscotch hashing algorithm. Each bucket contains some meta-data about its neighborhood known as 'hop-information.' The hop-information keeps track of which entries in the neighborhood actually hashed to the original bucket (as neighborhoods partially overlap). When an item is to be inserted with hash value i, one of three possible outcomes will occur, as shown in Figure 2.5 : (i) if bucket i is empty, then the item is inserted there; (ii) otherwise, if the neighborhood of bucket i contains an available bucket, then the item is inserted into the neighborhood; (iii) otherwise, then use linear probing to locate the first available bucket k. Then select a bucket j such that $i < j < k$, and j and k are in a common neighborhood. The contents of j are moved into k, and this process is repeated until the neighborhood of i has an available bucket. In the concurrent hopscotch hashing algorithm, a set of locks is striped across the bucket array.

The burst trie is a cache-conscious prefix-tree data structure for storing strings while maintaining sort order [87]. The burst trie algorithm has $O(m)$ worst-case time complexity for `contains`, `add`, and `remove` operations, where m is the length of the string. A trie is a multi-way search tree in which each node of the tree cor-

responds to a common prefix of a set of words [88]. Each node of a trie contains an array of pointers whose length is equal to the size of the alphabet. Thus the trie is a very space-intensive data structure. Numerous techniques have been developed to reduce the memory footprint of a trie. These techniques are reviewed by Askitis and Sinha [86]. These techniques rely on either internal fragmentation using compaction or on trading space for time using compression. The HAT-trie is a cache-conscious burst trie where the leaves of the trie are collapsed into array hashes called buckets [86]. All of the strings inside a bucket share a common prefix. A bucket is a fixed-size contiguous region of memory. When a bucket cannot store any more strings, then the bucket is burst into a trie node with multiple buckets. The new smaller buckets form the leaves of the new trie node. Searching for a string is a two-step process, first with $O(m)$ time complexity for traversing through the trie nodes and then $O(1)$ time complexity for searching through a bucket (see Figure 2.6).

Lock-free cache-conscious data structures that maintain the abstraction of a linearizable set have been studied previously in the context of unordered data structures. We explore novel alternatives, namely lock-free cache-conscious data structures that maintain the abstraction of a linearizable ordered set. The Messeguer skip tree [25] definition will be modified to generate the dense skip tree. The dense skip tree is then modified to construct the lock-free skip tree. To construct a radix-based linearizable ordered set, the burst trie [87] definition will be modified to generate the lock-free burst trie. For each transformation of one data structure into a novel alternative, we will highlight the challenge of adding a new invariant onto the data structure while preserving the remaining original properties of the algorithm.

Chapter 3

Dense Skip Tree & Optimistic Concurrent Skip Tree

The skip tree was introduced by Messeguer [25] as a generalization of the skip list for concurrent operations. A skip tree is a randomized multiway search tree such that each new element is assigned a random height from a geometric distribution. Elements are stored in the leaves of the tree, and the interior nodes of the tree serve as partitions. The isomorphism between the skip list and the multiway search tree has been noted in several publications [67–70]. The skip tree uses a cooperative algorithm to support `contains`, `add`, and `remove` operations. Local rules are defined which operate on nodes in the tree until no more preconditions are met. The locality of the rules ensures that only a constant number of nodes is locked per operation at any point in time. The cooperative rules are inspired by the on-the-fly garbage collection algorithm of Dijkstra et al. [89]. The cooperative algorithm encompasses all three operations on the data structure. It is undesirable that the read-only `contains` operation must wait for all concurrent `add` and `remove` operations to terminate before it can return a result, allowing the possibility of starvation for `contains` operations.

The majority of nodes in the skip tree contain no elements. The purpose of these nodes is to maintain the path length invariant in the tree. Our first objective in designing a cache-conscious isomorphism of the skip list was to eliminate the empty nodes from the skip tree. In the next section, we introduce the dense skip tree as a variation on the skip tree that eliminates the path length invariant from the data structure. Using the dense skip tree definition, we prove that the sequential expected running time of `contains`, `add`, and `remove` operations is $O(\log n)$. Upon showing that the dense skip tree has the same asymptotic performance as the skip list, we construct an optimistic concurrent skip tree as an initial phase in the design of a lock-free skip tree algorithm. Optimistic synchronization is a technique in which the necessary components of the data structure are locked on the assumption that components maintain a consistent state as the locks are acquired. Once the locking phase is complete, then the optimistic algorithm performs a consistency check on the components to verify that the assumption is correct. If the components are consistent, then a modification to the data structure may proceed. If the components are not consistent, then the locks are released and the operation is attempted again.

In the optimistic skip tree implementation the `contains` operation is wait-free, i.e. an operation with an upper bound on the number of steps before the operation is completed. Under the read-dominated scenario and a working set size that exceeds cache size, the optimistic skip tree attains x1.4 lock-free skip list peak throughput on a Sun Fire T1000, and x1.6 of the lock-free skip list peak throughput on a quad-core Intel Xeon. Given this knowledge, we can evaluate the optimistic skip tree implementation using synthetic benchmarks and use the performance results for the read-dominated scenarios as indicators of the performance of a lock-free skip tree algorithm.

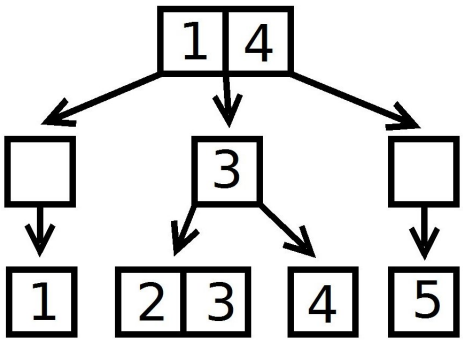


Figure 3.1: Skip tree example

3.1 Skip Tree

Definition 3.1. A *skip tree* [25] is defined to be a multiway search tree, such that the following properties hold for all nodes:

(D1) Each node contains k keys in sorted order, and $k+1$ possibly-null child references for $k \geq 0$.

(D2) Each node has a height h for $h \geq 0$.

(D3) Each key is assigned a random height at insertion.

(D4) A node with height h has children with height $h-1$. Nodes at zero height do not have children.

(D5) The left subtree of any key contains only keys of lesser or equal value. The right subtree of any key contains only keys of greater value.

An example skip tree is shown in Figure 3.1. A leaf node is a node with height 0. The root node is the node with the maximum height. In the skip tree, all paths from the root node to a leaf node have the same length. A skip tree node with 0 keys is defined as an empty node. The skip tree consists predominantly of empty nodes, in order to maintain the path length invariant.

The skip tree implements a sorted set abstract data type. Three operations are supported: `add`(v) adds v to the set and returns `true` iff v was not already in the set; `remove`(v) removes v from the set and returns `true` iff v was in the set; and `contains`(v) returns `true` iff v is in the set. A `contains` operation traverses the

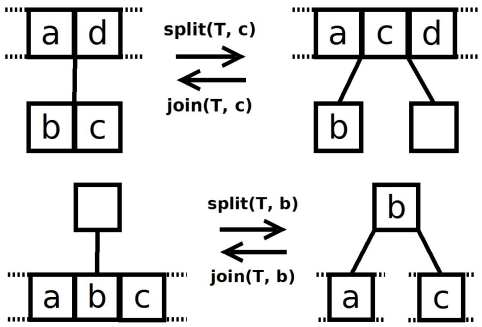
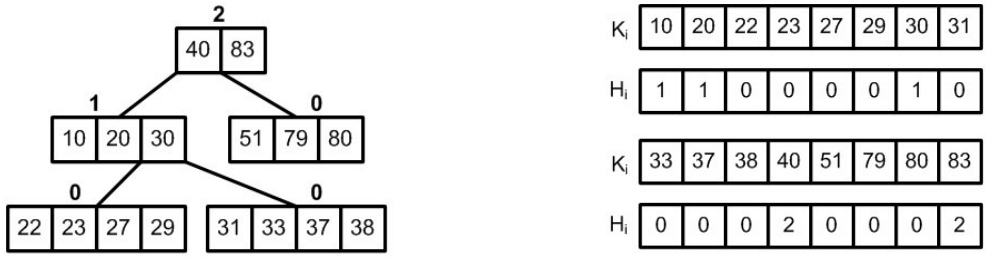


Figure 3.2: Split and join examples. Figure adapted from Messeguer [25].

search tree from the root node to a leaf node. An element is a member of the abstract set if-and-only-if the element is stored in a leaf node. An `add` operation begins in the same fashion as a `contains` operation, searching the tree from the root node to a leaf node. If the element is found at a leaf node then the operation returns `false`. If the element is not located then it is inserted into the leaf node. A random height is generated and a copy of the element will be inserted at the interior of the tree when the height is greater than zero. A `remove` operation seeks out the target element in the leaf nodes. If the element is not located at a leaf node then the operation returns `false`. If the element is found then it is deleted from the leaf node.

To insert a copy of an element in the interior of the tree, a series of splitting procedures is performed on the tree. A split procedure accepts a node and an element selected from the node, and transforms the node into two sibling nodes at the same height of the tree. The selected element in the split procedure, known as the splitting element, is pushed into the parent of the original node. When the splitting element is the leftmost or rightmost element of the node, then the newly created sibling node will be an empty node. To remove empty leaf nodes, the inverse of a split procedure is performed which is known as a join. Examples of split and join procedures are shown in Figure 3.2.



(a) dense skip tree

(b) K_i and H_i vectors

Figure 3.3: A dense skip tree with corresponding K_i and H_i vectors

3.2 Dense Skip Tree

The dense skip tree is defined as a compact variation on the skip tree data structure. The dense skip tree definition shares properties (D2), (D3), (D5), and (D6) from the original skip tree. (D4) is relaxed in the dense skip tree to permit any monotonic decreasing relationship between parent and children heights. With this relaxation, the existence of nodes with zero keys is no longer necessary.

Definition 3.2. A *dense skip tree* is defined to be a variant of the skip tree, such that the following properties are different:

(D1′) Each node contains k keys in sorted order, and $k+1$ possibly-null child references for $k \geq 1$.

(D4′) A node with height h has children with heights that are less than h. Nodes at zero height do not have children.

An example of a dense skip tree is shown in Figure 3.3(a). The number above each node indicates the height of that node.

In this section we will show that the number of keys per node of a dense skip tree is distributed according to a geometric distribution, and that the expected height of the tree of N keys is bounded by $\log_Q N$. Combining these two results will yield the claim that the expected time for search, insert, and delete operations is $O(\log_Q n)$. Property (D3) of the dense skip tree definition assigns random

heights to each key, which is clearly not a specification of the number of keys per node. The proofs are novel to the dense skip tree design, and serve as a completion of the skip tree proofs that are outlined by Messeguer [25]. The proof techniques used in this chapter represent a significant departure from the methods outlined in the original skip tree paper.

Let H indicate the random variable for the height of some key in the tree. Let H be distributed according to a geometric distribution:

$$\Pr(H = h) = q^h p \text{ where } p + q = 1 \text{ and } Q = 1/q$$

A Markov chain is a random process with the property that the next state depends only on the current state. Assume a set of states, $\Sigma = \{s_1, s_2, \ldots, s_r\}$. The Markov chain starts in one of these states and moves successively from one state to another. If the chain is currently in state s_i, then it moves to state s_j with a probability denoted by p_{ij}. The probabilities p_{ij} are known as transition probabilities. The transition probabilities do not depend on which states the chain was in before the current state. The transition probabilities can be represented in a square matrix, known as the transition matrix, where entry (i, j) represents the probability of transitioning from s_i to s_j [90].

Define the size of a skip tree node to be the number of elements that are contained within the node. Let the random variable S represent the node sizes of a dense skip tree.

Theorem 3.1. The mean value of S is $1/q$ and the variance of S is p/q^2.

Because the heights are assigned to keys independently of the natural ordering of the keys, we may sort the heights based on the natural ordering of the keys in the data structure. Then let H_i represent the height of key K_i in some sorted sequence of keys $\{K_0, K_1, K_2, \ldots\}$ as in Figure 3.3b. The size of a node is the number of

keys in the sequence observed with height $H_o = h$, before a key is found with height greater than h. This is because smaller heights do not divide the current node, while larger heights signal the start of a new node. The sequence of heights $\{H_0, H_1, H_2, \ldots\}$ can be characterized by a Markov chain with three possible states for each H_i random variable:

(INC) Increment state :
$(H_i = h)$ and $(H_0, H_1, H_2, \ldots, H_{i-1} \leq h)$

(NEU) Neutral state :
$(H_i < h)$ and $(H_0, H_1, H_2, \ldots, H_{i-1} \leq h)$

(TER) Terminating state :
At least one of $\{H_0, H_1, H_2, \ldots, H_i\}$ is greater than h. This is an absorbing state.

The transition matrix A for the sequence of heights is denoted by:

$$A = \begin{array}{c} \\ INC \\ NEU \\ TER \end{array} \begin{array}{c} \begin{array}{ccc} INC & NEU & TER \end{array} \\ \begin{pmatrix} p(H=h) & p(H<h) & p(H>h) \\ p(H=h) & p(H<h) & p(H>h) \\ 0 & 0 & 1 \end{pmatrix} \end{array} = \begin{bmatrix} q^h p & 1-q^h & q^{h+1} \\ q^h p & 1-q^h & q^{h+1} \\ 0 & 0 & 1 \end{bmatrix}$$

A state in a Markov chain is said to be absorbing if it is impossible to leave the state. A state that is not an absorbing state is known as a transient state. A Markov chain is an absorbing chain if-and-only-if it has at least one absorbing state, and from every state it is possible to eventually reach an absorbing state. The terminating state (TER) in the sequence of heights is an absorbing state. The increment and neutral states (INC and NEU) are transient states. The transition matrix for any absorbing Markov chains can be transformed into canonical form, such that the absorbing states form the bottommost rows and rightmost columns. Let Θ represent the transition matrix for the transient states in an absorbing Markov chain. For an absorbing Markov chain the matrix $F = (I - \Theta)^{-1}$ is called the fundamental

matrix of the Markov chain. The entry f_{ij} of F gives the expected number of times that the process is in the transient state s_j if it started in the transient state s_i [90].

$$\Theta = \begin{array}{c} \\ INC \\ NEU \end{array} \begin{array}{c} \begin{array}{cc} INC & NEU \end{array} \\ \begin{pmatrix} p(H=h) & p(H<h) \\ p(H=h) & p(H<h) \end{pmatrix} \end{array} = \begin{bmatrix} q^h p & 1-q^h \\ q^h p & 1-q^h \end{bmatrix}$$

$$F = (I-\Theta)^{-1} = \begin{bmatrix} 1-rp & 1-r \\ -rp & r \end{bmatrix}^{-1} = \begin{bmatrix} 1/q & 1/qr - 1/q \\ p/q & 1/qr - p/q \end{bmatrix} \quad \text{where } r = q^h$$

To determine the expected size of a node in the dense skip tree, we are searching for the number of times the sequence of heights is equal to the value h, prior to the first assignment $H_i > h$. The transient state of interest is INC which means that $s_f = INC$. As we have assumed that $H_o = h$, then the initial state $s_i = INC$. Combining these observations, the expected value for the size of a node in a dense skip tree is entry $f_{1,1}$ of the fundamental matrix F. The variance of the node size is entry $v_{1,1}$ of the variance matrix V that has been shown to equal $F(2F_{dg} - I) - F_{sq}$ [91]. F_{dg} is derived from F by setting each off-diagonal entry of F to zero. F_{sq} is computed by squaring each entry of F.

$$F_{dg} = \begin{bmatrix} 1/q & 0 \\ 0 & 1/qr - p/q \end{bmatrix}$$

$$F_{sq} = \begin{bmatrix} 1/q^2 & (1/qr - 1/q)^2 \\ p^2/q^2 & (1/qr - p/q)^2 \end{bmatrix}$$

Let $s = 1/q(1/r - 1)$ and $t = 1/q(1/r - p)$.

$$V = F(2F_{dg} - I) - F_{sq}$$

$$V = \begin{bmatrix} 2/q^2 - 1/q & s(2t-1) \\ 2p/q^2 - p/q & s(2t-1) \end{bmatrix} - \begin{bmatrix} 1/q^2 & s^2 \\ p^2/q^2 & t^2 \end{bmatrix} = \begin{bmatrix} p/q^2 & s(2t-1) - s^2 \\ p/q^2 & s(2t-1) - t^2 \end{bmatrix}$$

We have shown that the mean value of S is $f_{1,1} = 1/q$ and the variance of S is $v_{1,1} = p/q^2$. ■

Theorem 3.2. The probability mass function of S is $\Pr(S = s) = p^{s-1}q$ for $s \geq 1$.

To determine the probability mass function of S, reduce the problem to a game from probability analysis. The game consists of independent turns (τ) at which one of three mutually exclusive events can occur [92, 93]. The events have been labeled with their transition states from the Markov process. It is a single-player game that begins with a score of zero.

(INC) The game continues with the addition of one point to the score.
Let $a = \Pr(\tau = INC) = \Pr(H = h) = q^h p$.

(NEU) The game continues without addition to the score.
Let $b = \Pr(\tau = NEU) = \Pr(H < h) = 1 - q^h$.

(TER) The game terminates without addition to the score.
Let $c = \Pr(\tau = TER) = \Pr(H > h) = q^{h+1}$.

Let X represent the random variable for the number of points scored in a game. The probability mass function of X will be determined through its probability generating function. A probability generating function transforms a probability mass function into another space where a solution can be determined, and then the solution is transformed back into the space of probability mass functions.

The generating function of a discrete random variable is a power series representation of the probability mass function. Given that X is a discrete random variable taking on non-negative values, then the probability generating function of X is defined as

$$g_X(z) = E(z^X) = \sum_{k=0}^{\infty} \Pr(X = k) z^k.$$

The generating function of X is determined by application of the partition theorem for expectation. The partition theorem breaks down the first transition of the game into the three possible outcomes, related to INC, NEU, and TER. The variables a, b, and c are defined on the previous page.

$$\begin{aligned}
E(X) &= \Pr(\tau = INC) \cdot E(X|\tau = INC) + \Pr(\tau = NEU) \cdot E(X|\tau = NEU) \\
&\quad + \Pr(\tau = TER) \cdot E(X|\tau = TER) \\
E(X) &= a \cdot E(X|\tau = INC) + b \cdot E(X|\tau = NEU) + c \cdot E(X|\tau = TER) \\
g_X(z) &= E(z^X) = a \cdot E(z^X|\tau = INC) + b \cdot E(z^X|\tau = NEU) + c \cdot E(z^X|\tau = TER)
\end{aligned}$$

Each expectation is transformed so that the knowledge of the first transition is no longer necessary. If the first transition is to the INC state, then this transition can be ignored provided that we increment all the possible outcomes of X by 1. If the first transition is to the NEU state, then this transition can be ignored and the game continues to be played. If the first transition is to the TER state, then the game is terminated with a final score of 0. These transformations are all possible due to the memoryless property of the Markov chain.

$$
\begin{aligned}
E(z^X|\tau = INC) &= E(z^{X+1}) = zE(z^X) = zg_X(z) \\
E(z^X|\tau = NEU) &= E(z^X) = g_X(z) \\
E(z^X|\tau = TER) &= \sum_{k=0}^{\infty} z^k \Pr(X = k|\tau = TER) = \sum_{k=0}^{0} (z^k)(1) + \sum_{k=1}^{\infty} (z^k)(0) = 1 \\
g_X(z) &= a \cdot zg_X(z) + b \cdot g_X(z) + c \\
&= \frac{c}{1-b-za} = \frac{c}{1-b}\left(1 - \frac{za}{1-b}\right)^{-1} \\
&= \frac{c}{1-b}\sum_{k=0}^{\infty}\left(\frac{za}{1-b}\right)^k = \frac{c}{1-b}\sum_{k=0}^{\infty} z^k \left(\frac{a}{1-b}\right)^k
\end{aligned}
$$

The remaining task is to transform the probability generating function $g_X(z)$ back into the probability mass function $\Pr(X = x)$. The expression $(1-y)^{-1}$ is rewritten as the geometric distribution $\sum_{k=0}^{\infty}(-y)^k$ for $y = za(1-b)^{-1}$. In this form the definition of the probability generating function can be applied $E(z^X) = \sum_{k=0}^{\infty} z^k \Pr(X = k)$ to extract the value of $\Pr(X = k)$.

$$
\begin{aligned}
\Pr(X = k) &= \frac{c}{1-b}\left(\frac{a}{1-b}\right)^k = \frac{q^{h+1}}{1-(1-q^h)}\left(\frac{q^h p}{1-(1-q^h)}\right)^k = p^k q \\
\Pr(S = s) &= \Pr(X = s-1) = p^{s-1}q \quad \text{for } s \geq 1
\end{aligned}
$$

The reduction to the problem of computing node size is made by $S = X + 1$, as X fails to account for the initial key K_0 with height $H_0 = h$. ■

Corollary 3.1. The `contains`, `add`, and `remove` operations on a dense skip tree have expected cost $O(\log_Q n)$.

To derive estimates for the height of the tree, let M_n represent the maximum height observed on a sequence of n keys. M_n is an upper bound on the longest path from the root to the leaves of the tree. It has been shown that $E(M_n) = \log_Q n + \frac{\gamma}{L} + \frac{1}{2} - \delta(\log_Q n) + O\left(\frac{1}{n}\right)$, where $L = \log Q$, γ is Euler's constant, and

$\delta(x)$ is a periodic function of period 1 and mean 0 [94]. Thus the expected height of the tree is bounded by $\log_Q n$ for $n \gg 0$ as $\log_Q n$ is the dominating term. Theorem 3.1 has shown that the expected size of a node is a constant. If the expected height of the tree is bounded by $\log_Q n$, then the expected cost for search, insert, and delete operations is $O(\log_Q n)$.

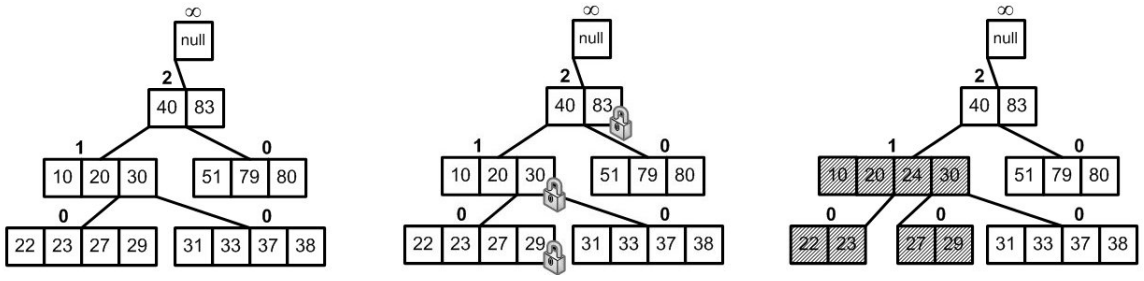
Theorem 3.3. The dense skip tree has fewer pointers per item than either a binary tree or a skip list.

The space efficiency of the dense skip tree can be estimated by the expected number of pointers per element of the tree. Keys cannot migrate up or down the tree once they have been inserted. Therefore leaf nodes do not need storage allocated for children pointers. p is the fraction of keys that reside at the leaves of the tree, while q is the fraction of keys that reside above the leaves. The expected number of pointers per node in the upper level of the tree is equal to the expected number of keys per node plus one. Therefore the expected number of pointers per key is: $q \cdot (E(S) + 1) + p \cdot (0) = q \cdot (\frac{1}{q} + 1) + 0 = 1 + q$. This is more space-efficient than the expected number of pointers per key in a binary tree (2 pointers per key) and the expected number of pointers per key in a skip list ($1/p$ pointers per key).

The basic operations on a dense skip tree share the same asymptotic costs for their expected values as the skip list and the skip tree. The asymptotic expected costs of `contains`, `add`, and `remove` operations on dense skip trees, skip trees, and skip lists equal the asymptotic worst-case costs of these operations on balanced binary trees.

3.3 Optimistic Concurrent Skip Tree

The concurrent dense skip tree algorithm implements a linearizable [40] sorted set data type. Three operations are supported: `add`(v) adds v to the set and returns



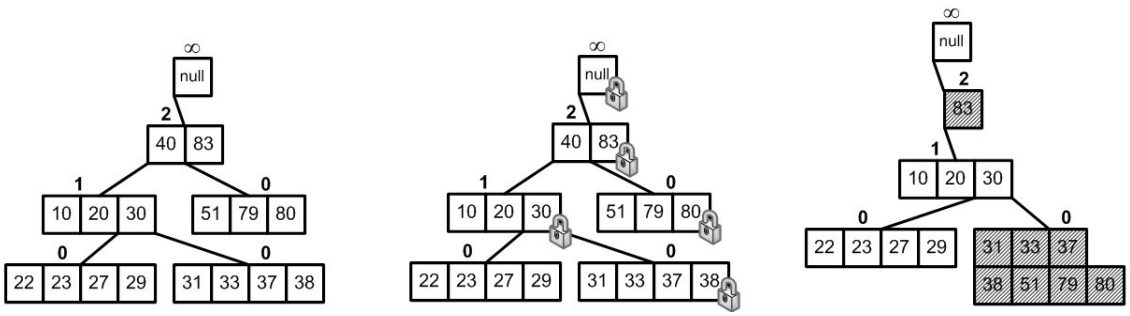
(a) before insertion (b) locking phase (c) after insertion

Figure 3.4: The insertion of key 24 at height 1. Modified nodes are shaded.

`true` iff v was not already in the set; `remove`(v) removes v from the set and returns `true` iff v was in the set; and `contains`(v) returns `true` iff v is in the set. An optimistic locking strategy has been adopted. An optimistic concurrent algorithm provides a clear concurrency model, yields a relatively straightforward correctness proof, and can perform as well as the best previously known lock-free algorithms under common search patterns [95].

The operations of an optimistic concurrent algorithm traverse a data structure without acquiring locks. Once all nodes to be modified have been located, the operation enters a locking phase. In the locking phase, first a set of nodes is locked and then the locked nodes are checked for consistency. If the set of locked nodes is not consistent, then the nodes are unlocked and the operation is retried. Otherwise the operation may execute to completion. Each node is augmented with a boolean `linked` field that is true iff the node is attached to a parent with a `linked` field of true. Additionally, the root node of a concurrent dense skip tree is defined as a sentinel node. A sentinel node has a height of ∞, stores zero keys and one child, and has a `linked` field that is always true.

The `contains`, `add`, and `remove` operations all traverse a path through the data structure. Starting from the sentinel node, the operation iteratively searches for the next node that would contain the target key, given rules (D5) and (D6) of the data structure. A `contains` operation terminates at the end of the traversal, and



(a) before deletion (b) locking phase (c) after deletion

Figure 3.5: The removal of key 40 from a dense skip tree. Modified nodes are shaded.

returns whether or not the target key was located along the path. An `add` operation terminates at the end of the traversal if the target key is located. If the target key is not located, then a subset of the nodes along the path is locked. Given a random height h, the nodes in the path with a height less than or equal to h are locked, and the parent of the tallest node from this set is also locked. An example of the locking phase for an `add` operation is shown in Figure 3.4b. A validation step checks that all locked nodes have a `linked` field of true, and that each node along the locked path contains a parent along the locked path. Finally the key can be inserted and a split operation is performed. A dense skip tree split operation is similar to a B-tree split operation. The difference between the two operations is that a B-tree splits a single node into two nodes of identical length, whereas a skip tree splits a node using the target key as a pivot. By selecting the target key as a pivot, properties (D5) and (D6) are preserved. The `add` and `remove` operations employ a copy-on-write modification strategy. The original nodes are replaced by their copies, and the `linked` fields of the original nodes are set to false (not shown in Figure 3.4c).

The `remove` operation begins with a traversal through the data structure. If the target key is not located, the operation terminates at the end of the traversal. If the target key is located, then three types of nodes are locked. The target node containing the key is locked, the parent of the target node is locked, and all nodes that

will be involved in the join operation are locked. The nodes involved in the join operation consist of the child to the left and the child to the right of the target node and the rightmost and leftmost descendants of these two children, respectively. In Figure 3.5b, the target node contains the key 40, the left child contains the key 10, the right child contains the key 51, and the rightmost descendants of the left child consist of the node containing the key 31. The join operation is the reverse of a split operation. Modified nodes are replaced by their copies, and the `linked` fields of the original nodes are set to false.

The optimistic concurrent algorithm design yields relatively straightforward correctness proofs. The algorithm is deadlock-free because locks are always acquired in a bottom-up fashion. Any overlapping non-`contains` operations consist of a set of common nodes that participate in both locking phases. The operation that acquires the lock on the common node with the minimum height will proceed (or the leftmost shared node with minimum height in the case of overlapping deletions). Once an operation acquires a lock on the common node with the minimum height, the remaining nodes in that operation's locking phase cannot be acquired by the overlapping operations, as locks are always acquired in a bottom-up fashion. The `contains` operation is wait-free as it does not acquire any locks and it never retries. The linearization point of the `add` operation occurs when a copy of the target node containing the inserted key is linked into the data structure. The linearization point of the `remove` operation occurs when a copy of the target node without the deleted key is linked into the data structure.

3.4 Synthetic Benchmarks

Performance analysis of the concurrent skip tree has been conducted using synthetic benchmarks under six workloads. The workloads vary in proportions of

`contains`, `add`, and `remove` operations and in the number of unique keys stored by the data structure. Half of the workloads use 90% `contains`, 9% `add`, and 1% `remove` operations, as read-intensive workloads are the most common steady-state configuration for many applications [95]. Half of the workloads use 33% `contains`, 33% `add`, and 33% `remove` operations, to represent steady-state configurations that are write-intensive. 5,000,000 operations are executed in each independent trial, while the total throughput of the data structure as measured by the number of concurrently executing threads varies from 1 to 2048. The number of unique keys is determined through selection of random values from a uniform distribution with a range of 500, or 200,000, or 2^{32} integers. The three ranges represent scenarios of (a) a very small number of keys, (b) a larger number of keys that still fit entirely within cache, and (c) a quantity of keys that cannot entirely be contained within the largest level of cache, respectively. Each independent trial is repeated 64 times. Keys that are designated for a `contains` or `remove` operation are pre-loaded into the data structure prior to the beginning of a trial.

Benchmarks were evaluated on a Sun Fire T1000 with eight cores at 1.0 GHz and 32 hardware threads, and an Intel Xeon L5430 with four cores at 2.66 GHz and 8 hardware threads. Each core of the UltraSPARC T1 processor has a 8 kB level-1 data cache and a 16 kB instruction cache. The cores share a 3 MB level-2 unified cache. Each core of the Xeon processor has a 32 kB level-1 data cache and a 32 kB level-1 instruction cache. Each pair of cores shares a 6 MB level-2 unified cache. The benchmarks were executed on the 32-bit server version of the HotSpot Java Virtual Machine version 1.6.0 update 14.

Four data structures are compared in the performance analysis. They are the dense skip tree, the B^{link}-tree, the optimistic skip list, and the lock-free skip list. The optimistic skip list implementation is that written by Herlihy et al. [95] and used in their benchmarks. It is an optimized implementation of the algorithm

discussed in their paper. The lock-free skip list implementation is from the Java SE 6 java.util.concurrent package developed by the JSR 166 concurrency utilities group [96]. The dense skip tree is implemented using the optimistic locking strategy described in the previous section. The B^{link}-tree implementation is an adaptation of the algorithm by Sagiv [97] for in-memory data structure access. When the B^{link}-tree algorithm is adapted to main memory access, a shared-reader/exclusive-writer lock is placed on each node [98–100]. The deletion algorithm used is that of Lehman and Yao [101], which allows for leaf nodes to 'underflow', meaning that there is no lower bound on the size of leaf nodes. The B^{link}-tree implementation uses an atomic reference to store the root of the tree in order to reduce lock-contention. This strategy cannot be replicated for non-root nodes, as the presence of the link reference yields up to two references per non-root node (a link reference cannot point to the root node).

The lock-free skip list implementation from the java.util.concurrent package is among the fastest implementations of a linearizable sorted set abstract data type. The B^{link}-tree has been found to perform the best among concurrent B-tree implementations over a wide range of resource conditions, B-tree structures, and workload parameters [100]. The optimistic skip list implementation is included in the performance analysis as a comparison to the lock-free skip list, in order to show the potential of a future lock-free dense skip tree implementation.

The results of the benchmarks on the Sun Fire T1000 and the Intel Xeon L5430 are shown in Figures 3.6 - 3.9. Tic marks denote the mean of the repeated experiments and error bars denote standard deviation. Parameter variations for the dense skip tree and the B^{link}-tree are shown in Figures 3.10 - 3.13. The parameter Q of the dense skip tree determines the mean node size, and the parameter M of the B^{link}-tree determines the minimum number of keys per node ignoring underflow caused by `delete` operations. In Figures 3.6 and 3.7, values of $Q = 16$ and

$M = 128$ are selected as they show the best overall performance across all scenarios. In Figures 3.8 and 3.9, values of $Q = 32$ and $M = 128$ are selected. The random height generator for the optimistic skip list implementation is identical to distribution of random values used in the java.util.concurrent lock-free skip list implementation, which returns 0 with probability $3/4$, i with probability $2^{-(i+2)}$ for $i \in [1, 30]$, and 31 with probability 2^{-32}.

None of the four data structures outperforms the other three across all six workloads. Under the read-dominated scenario and a range of 2^{32} keys, the skip tree and B^{link}-tree reach 144% and 166% of the lock-free skip list throughput at peak performance on the Sun Fire, and 155% and 177% of the lock-free skip list throughput at peak performance on the Intel Xeon. Under the same read-intensive scenario but a range of 500 keys, the skip tree reaches 73% of the lock-free skip list throughput on the Sun Fire, while the B^{link}-tree reaches only 6% of the lock-free skip list throughput. The B^{link}-tree performs the best relative to the other algorithms under the write-dominated scenario and a range of 2^{32} keys, yielding 168% of the lock-free skip list throughput on the Sun Fire while the skip tree reaches 118%. Both cache-conscious data structures perform poorly under the write-dominated scenario and a range of 500 keys, with 7% relative throughput for the skip tree and 2% for the B^{link}-tree. The poor performance of the B^{link}-tree is a consequence of lock contention when the data structure consists of a small set of nodes. There is no lock contention when the B^{link}-tree consists of one node, because the root node is accessed through an atomic reference. When the B^{link}-tree consists of two leaf nodes and a single root node, all concurrent threads must use one of the two reader/writer locks on the leaf nodes. In the read-dominated scenario, the 10% of write operations must wait for exclusive access to the reader/writer locks on the leaf nodes.

Several trends emerge from the synthetic benchmarks. The optimistic concur-

rent skip tree outperforms the lock-free skip list in read-dominated workloads when the working set size exceeds the cache size. The skip tree peak throughput is almost as high as the skip list peak throughput in read-dominated workloads for small working set sizes. The B^{link}-tree was designed for the disk/memory boundary where read and write operations operate at the granularity of disk block sizes. In adapting the B^{link}-tree as an in-memory data structure, the reliance on shared reader/writer locks incurs a penalty for the synthetic benchmarks with small working set sizes. The optimistic skip list shows a degradation in performance on the write-dominated workloads as compared to the lock-free skip list. In the next chapter we will introduce a lock-free skip tree algorithm. This algorithm will exhibit the same performance advantages as the optimistic skip tree in read-dominated workloads, and will not have a reduced throughput in write-dominated workloads.

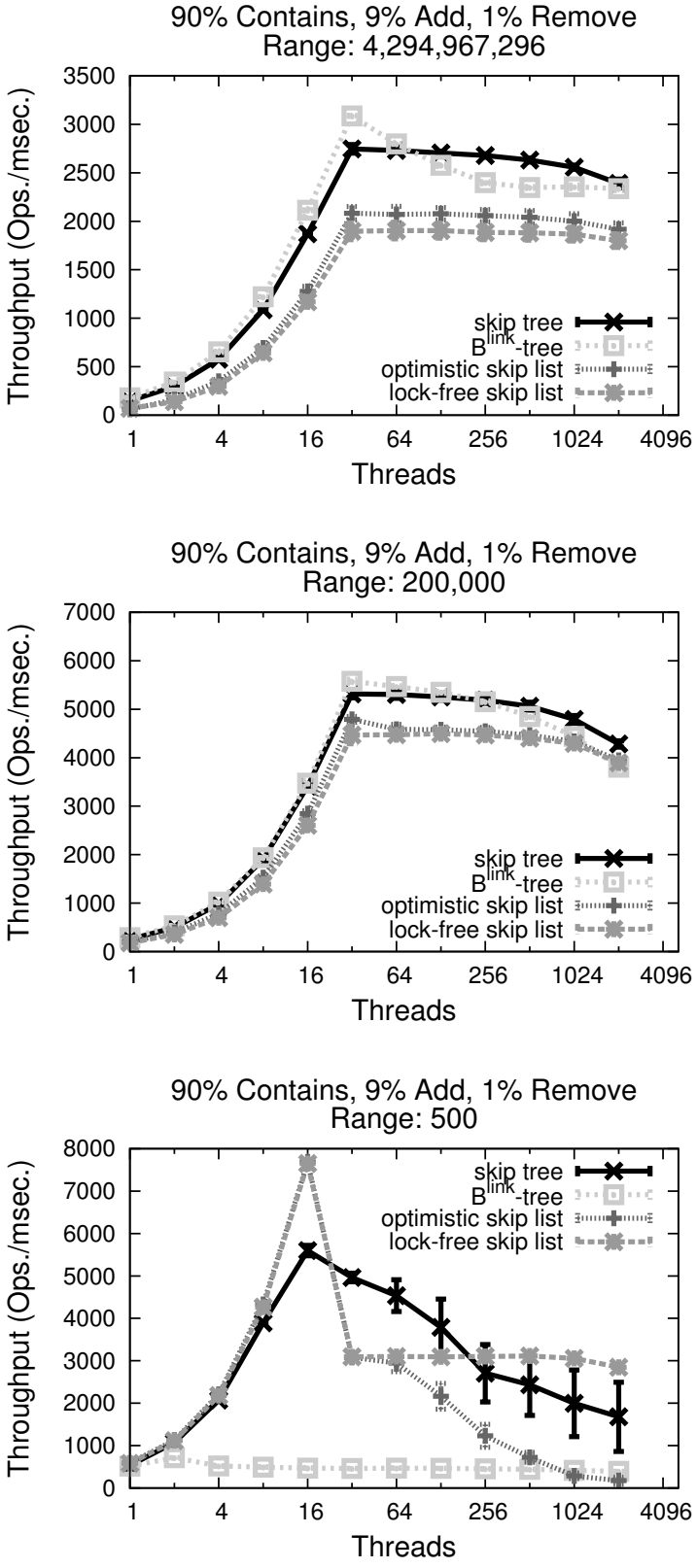


Figure 3.6: Read-dominated synthetic benchmarks on Sun Fire T1000

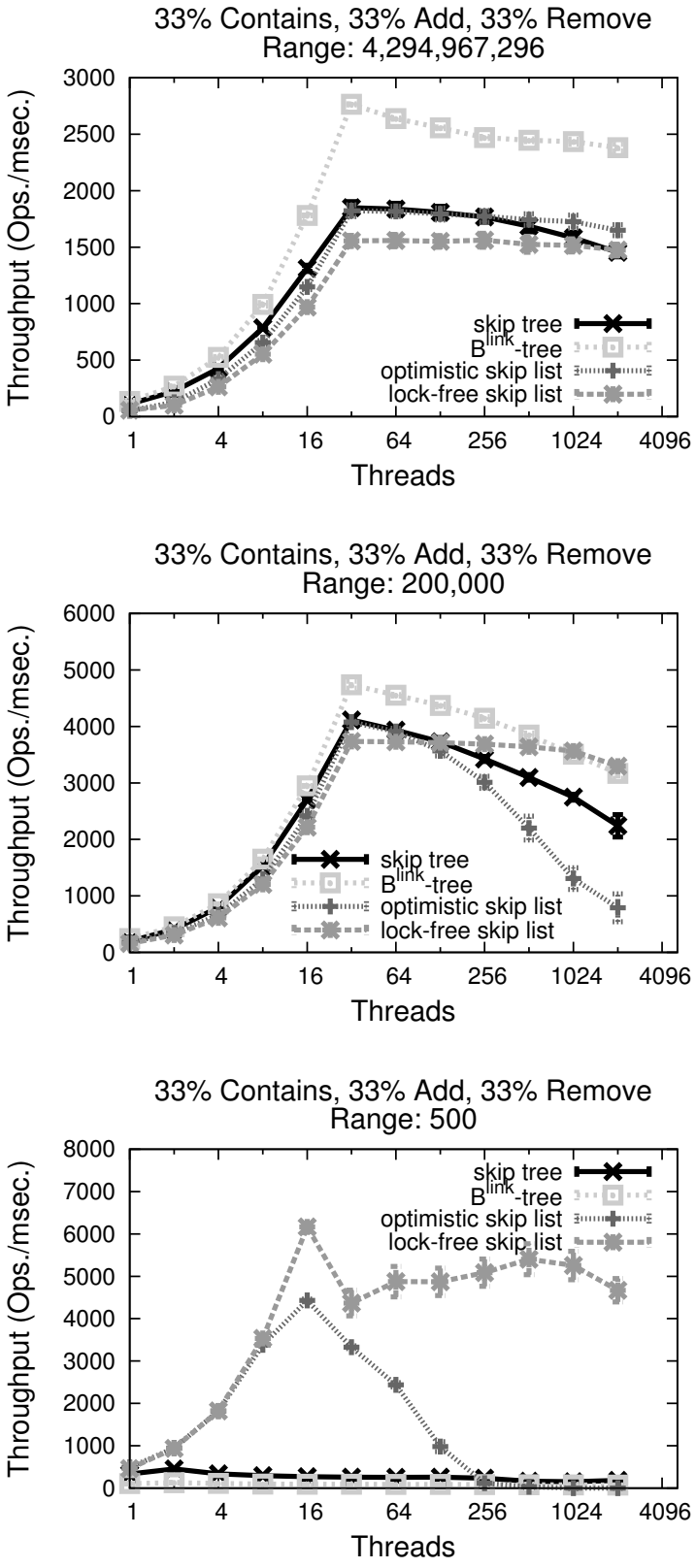


Figure 3.7: Write-dominated synthetic benchmarks on Sun Fire T1000

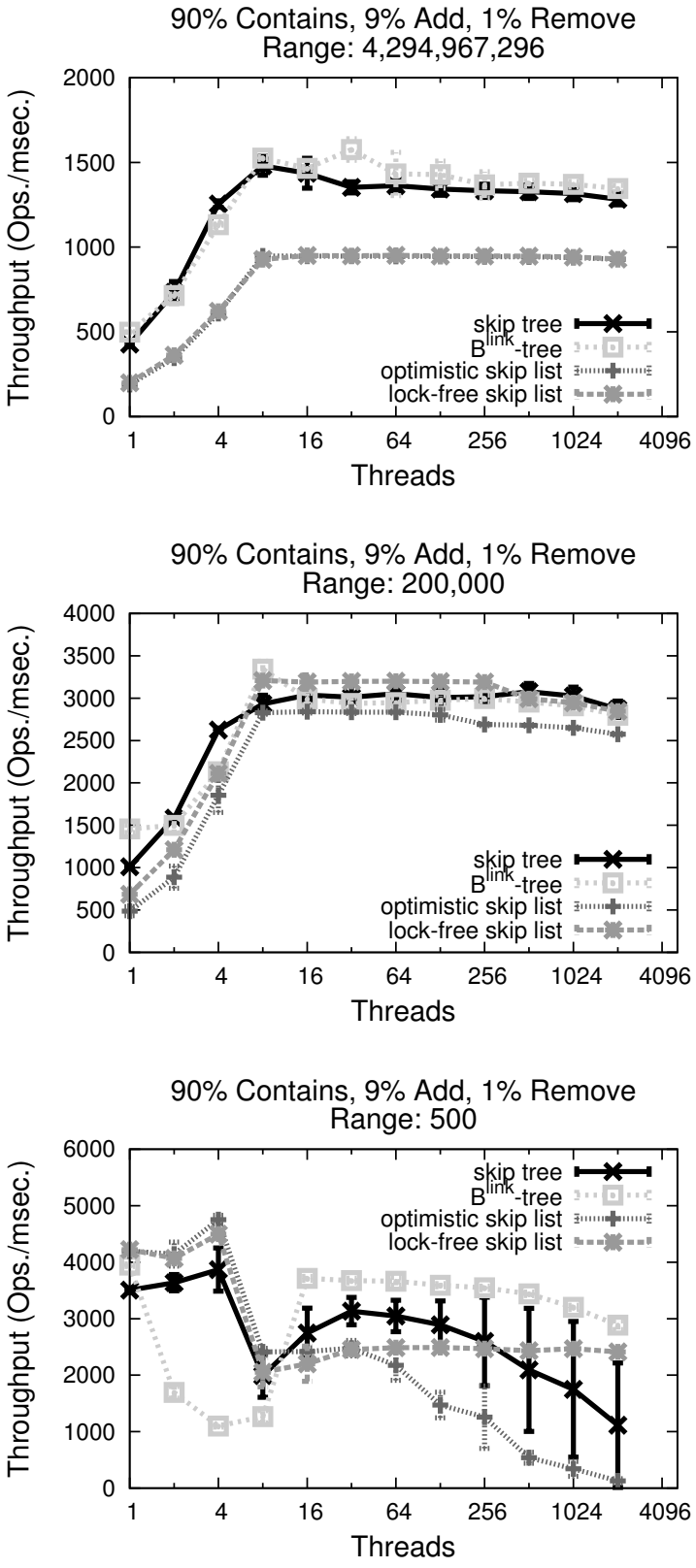


Figure 3.8: Read-dominated synthetic benchmarks on quad core Intel Xeon

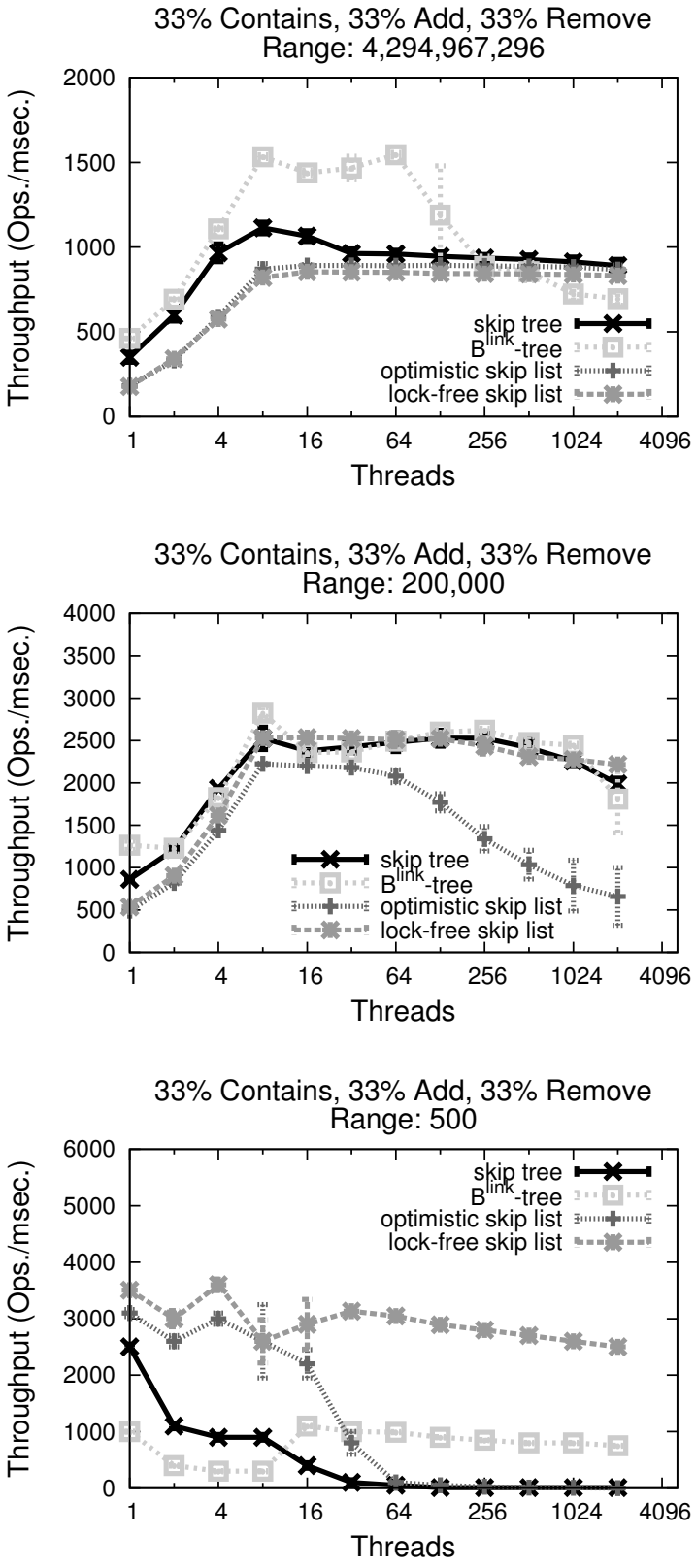


Figure 3.9: Write-dominated synthetic benchmarks on quad core Intel Xeon

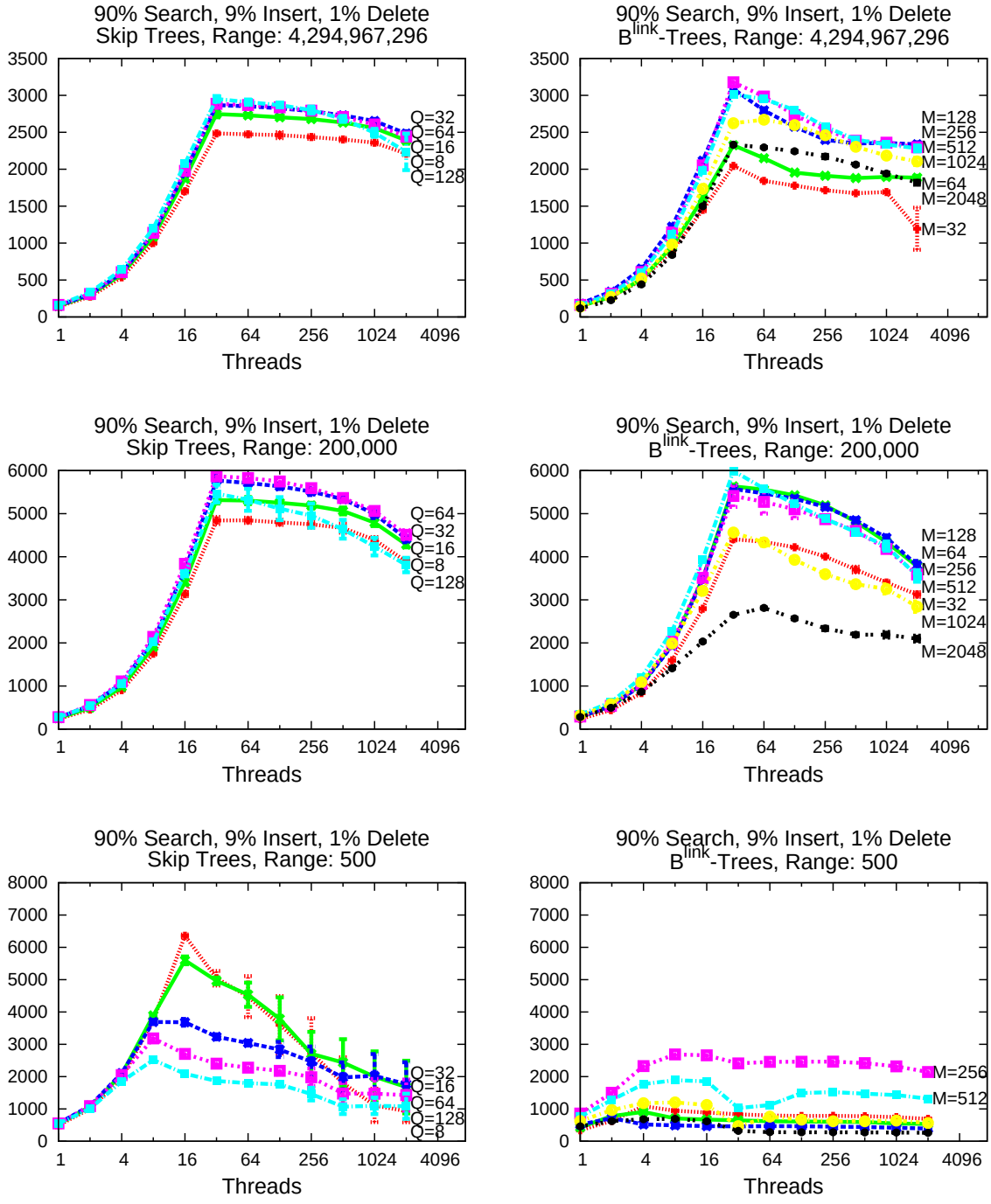


Figure 3.10: Parameter variations on the Sun Fire T1000 (read-dominated)

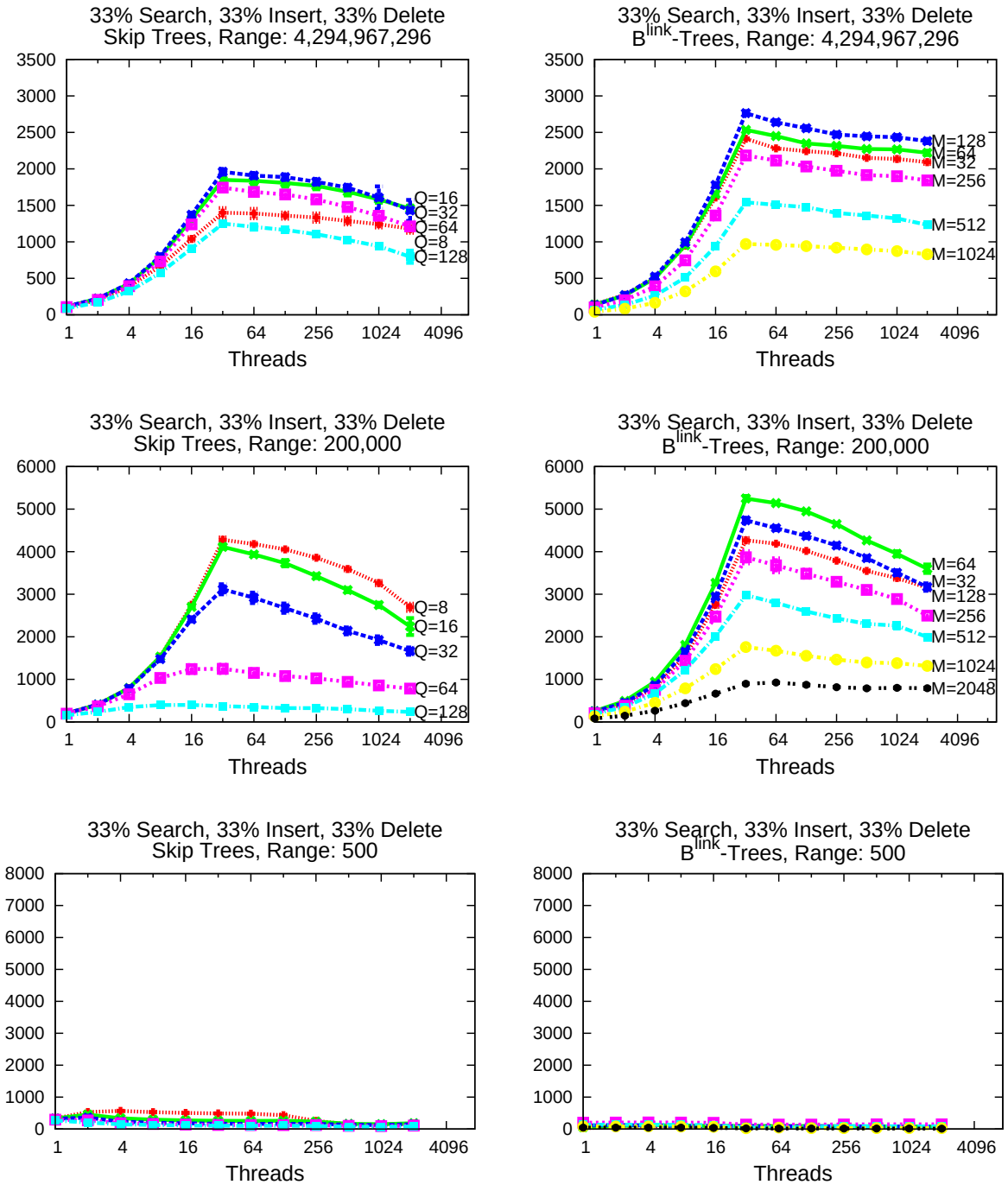


Figure 3.11: Parameter variations on the Sun Fire T1000 (write-dominated)

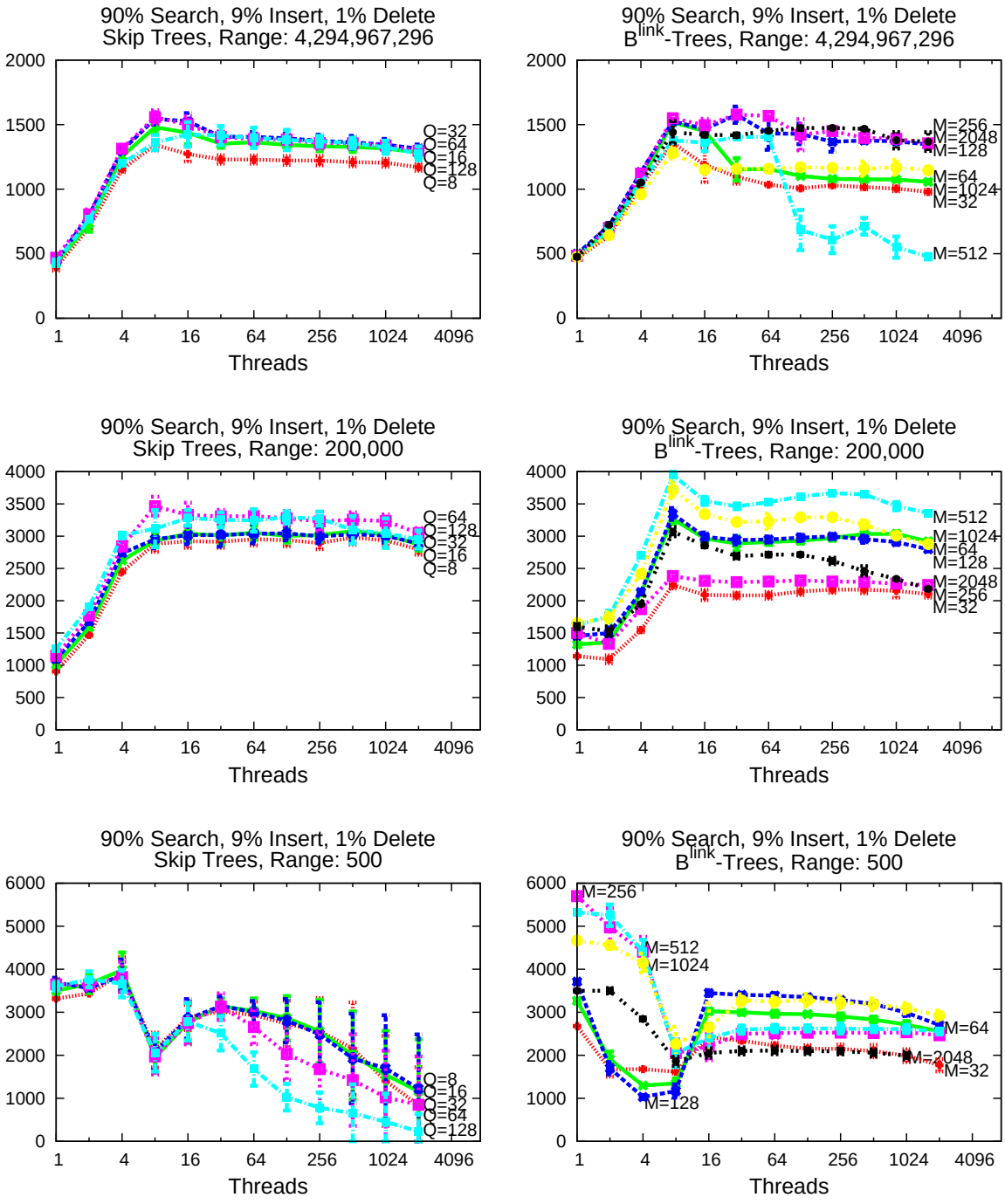


Figure 3.12: Parameter variations on the quad core Intel Xeon (read-dominated)

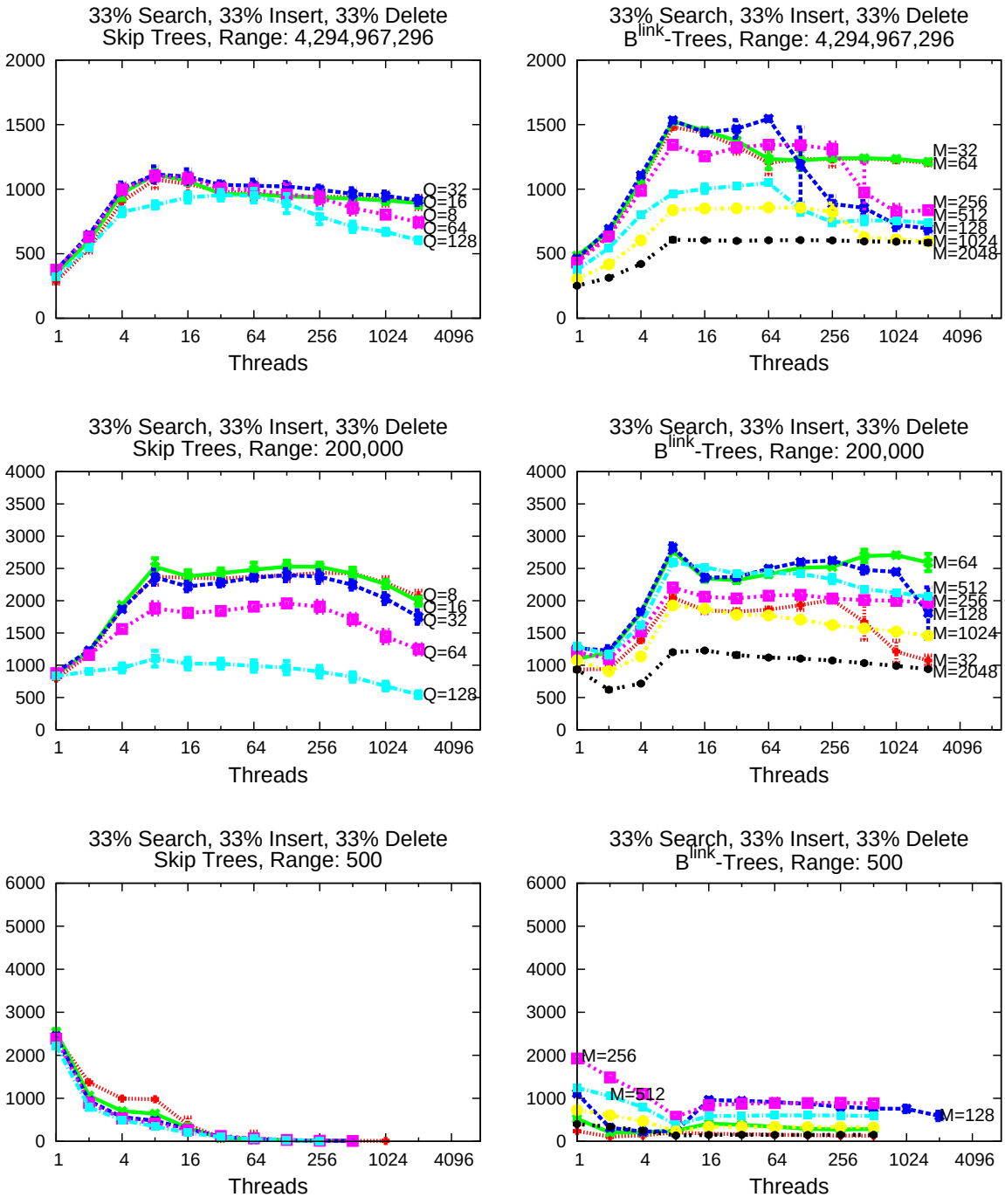


Figure 3.13: Parameter variations on the quad core Intel Xeon (write-dominated)

Chapter 4

Lock-Free Skip Tree

In this chapter we present a lock-free skip tree algorithm. The algorithm implements a lock-free randomized multiway search tree. In a series of synthetic benchmarks, the lock-free skip tree implementation performs up to x2.3 better in the time to completion of read-dominated workloads as compared to the state of the art lock-free skip list implementation. Across all synthetic benchmarks, the mean improvement of the skip tree is x1.4 compared to the skip list and the minimum relative performance is x0.87. Our lock-free skip tree implementation is available online [11].

Messeguer's skip tree [25] contains properties that are disadvantageous for its use as a cache-conscious data structure. Primarily, the most frequent node size in the Messeguer skip tree is zero. These empty nodes are necessary to preserve the length invariant of all paths from the root to the leaf nodes. In the previous chapter, we defined a dense skip tree that eliminated both the path length invariant and the existence of empty nodes. The optimistic skip tree was also defined in the previous chapter as a concurrent implementation of the dense skip tree. It uses an optimistic locking approach and yields performance improvements over the concurrent skip list in read-dominated synthetic workloads.

Several motivating factors encourage the design of a lock-free algorithm. Mutual exclusion causes blocking of threads or processes that must wait until a lock is released. Locks can cause deadlock, due to the early termination of a thread without releasing a lock, or due to subtle defects in the concurrent algorithm design. Undesirable effects such as priority inversion or convoying may occur [47]. Finally, lock contention limits scalability in highly-concurrent environments as lock-management becomes a sequential bottleneck. The lock-free skip tree algorithm has relaxed structural properties that allow atomic operations to modify the tree without invalidating the consistency of the data structure. Our skip-tree definition does not require that neighboring elements in the tree's interior serve as partitions on the tree. It maintains consistency by defining a *reachability* relation from the root of the tree to any potential element stored in the tree for all possible states of the tree (Section 4.3).

The following contributions are presented in this chapter:

- We describe a redesign of the skip tree in order to support lock-free operations. Our design has two primary differences from the Messeguer skip tree definition. First, we introduce link references to allow nodes to split independently of their parent nodes (Section 4.4). Second, we relax the requirement that routing nodes behave as partitions on the tree (Section 4.5).

- We describe a practical lock-free algorithm for the concurrent skip tree. The algorithm supports lock-free `add` and `remove` operations and wait-free `contains` operations. The algorithm is shown to be linearizable (Section 4.6).

- We show that our lock-free implementation outperforms a highly-tuned skip list, a relaxed balance AVL tree, and a B^{link}-tree across different thread counts, operation mixes, and machine architectures when the working set size cannot be contained in cache (Section 4.7).

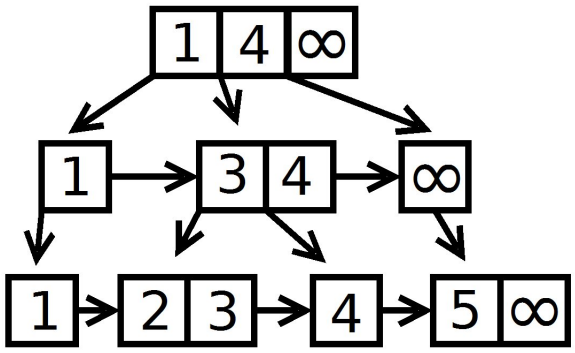


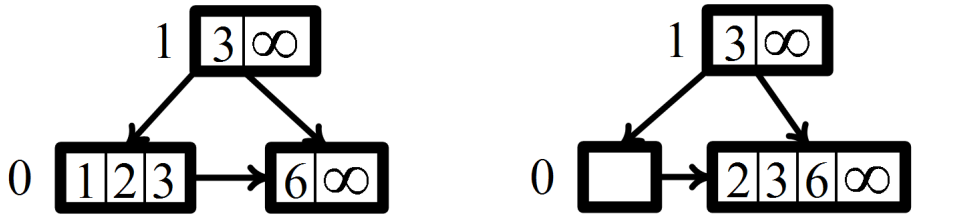
Figure 4.1: Lock-free skip tree example. Isomorphic to Figure 3.1.

4.1 Lock-free Skip Tree Definition

The lock-free skip tree implements a linearizable ordered set data structure over some domain T. Three operations are supported: `contains`, `add`, and `remove`. The lock-free skip tree consists of several linked lists stacked on top of each other. Each linked list is referred to as a level of the tree. The linked lists are composed of nodes. Each node contains some number of elements and a reference to the next node in the list. The number of elements per node varies over time and is independent of the number of elements in other nodes. In addition to storing elements and a reference to the next node in the list, each node stores references to nodes that are in the next lowest linked list level of the tree. A reference from one node to the next node within the same linked list level is called a link reference. A reference from one node to another node in a lower linked list level is called a child reference.

The lowest level of the tree consists of elements of height 0, and is defined to be level 0 or the leaf level of the tree. Non-leaf nodes are referred to as routing nodes. A routing node contains k child references to nodes that are one level below in the tree. Leaf nodes do not store any child references. A leaf node or a routing node with zero elements is referred to as an empty node. It is forbidden to insert new elements into an empty node.

An element is assigned a random height, h, upon insertion into the tree. The



(a) insertion into an empty tree

(b) elements $\{1,2,3\}$ are deleted then $\{2,3\}$ are reinserted.

Figure 4.2: A sequence of `add` and `remove` operations.

random height is a non-negative integer. An element is considered to be a member of the skip tree if-and-only-if the element is a member of a leaf node in the tree. If the random height h is greater than zero, then a copy of the element is inserted at each level from level 0 up to and inclusive of level h.

The root of the tree is defined to be the first node in the tree at the current highest level. An empty skip tree consists of one leaf node containing the value $+\infty$. Figure 4.2 shows examples of lock-free skip trees. To the left of the first node at each level is a number indicating the height of that level.

Any pair of adjacent elements share exactly one child reference. For example in Figure 4.2 the child reference of elements 3 and $+\infty$ at level 1 is the node containing $\{6, +\infty\}$ at level 0. If A is the last element of a node and B is the first element of the successor node at the same level, then the shared child reference of A and B is the first reference of the node containing element B. The first element of each level in the tree is assumed to be $-\infty$, although this value is not explicitly stored in the tree.

In order to explain the structural properties of a lock-free skip tree it will be useful to define the *tail set* of a node n at level i. The tail set of n is the set that contains n and all nodes subsequent to n at level i. The tail set of a node n is written as $tail(n)$.

Definition 4.1. A *lock-free skip tree* consists of a set of nodes. Each node contains a

sorted list of elements, a reference to the next node in the linked list, and possibly a list of child references. The elements stored within the nodes are members of some domain set T. The lock-free skip tree obeys the following properties:

(D1) Each level contains the element $+\infty$ at the last element of the last node. The element $+\infty$ appears exactly once for each level.

(D2) The leaf level may not contain duplicate elements.

(D3) For each level, given some $v \in \mathtt{T}$ there exists exactly one pair of adjacent elements A and B such that $A < v \leq B$.

(D4) Given levels i and $i-1$ and some $v \in \mathtt{T}$, there exist exactly two pairs of adjacent elements $\{A_i, B_i\}$ and $\{A_{i-1}, B_{i-1}\}$ that satisfy property (D3). If *source* is the child reference between elements A_i and B_i, and *target* is the node that contains element B_{i-1}, then $target \in tail(source)$.

(D5) For each node, given some $v \in \mathtt{T}$ such that v is greater than or equal to all the elements of the node, then v will always be greater than or equal to all elements of the node in all possible futures.

Theorem 4.1. Given a skip tree that obeys properties (D1) - (D5), each level of the tree is a sequence of elements in non-decreasing order.

Proof: Assume there exists a pair of consecutive elements A and B occurring in the sequence that is the first pair of elements occurring in decreasing order, i.e. $B < A$. Let C and D be the first pair of consecutive elements in increasing order that is subsequent to A and B. It is known that the pair C and D exists because property (D1) states that each level terminates with $+\infty$ and $+\infty$ appears exactly once for each level. Therefore each level terminates with a pair of adjacent elements in

increasing order. As C and D are defined as the first pair of consecutive elements in increasing order that are subsequent to A and B, then it follows that $C \leq B$. The two inequalities may be combined to show that $C < A$.

Select some $v \in \mathtt{T}$ such that $C < v \leq D$ and $v < A$. By combining the observation that $-\infty < v < A$ and the assumption of the proof that A and B are the first pair of consecutive elements in decreasing order, there must exist a pair of consecutive elements α and β in between elements $-\infty$ and A, inclusively, such that $\alpha < v \leq \beta$. However this violates property (D3) of the skip tree which requires exactly one pair of adjacent elements C and D such that $C < v \leq D$. Therefore the sequence cannot contain a pair of consecutive elements in decreasing order. ■

Corollary 4.1. The leaf level of the skip tree is a sequence of elements in increasing order.

Property (D2) of Definition 4.1 states the leaf level does not contain duplicate elements. Theorem 4.1 has shown that all levels contain a sequence of elements in non-decreasing order. Combining these two statements yields the conclusion that the leaf level is a sequence of elements in increasing order.

4.2 Class and Field Declarations

A node consists of a single atomic reference to a contents object. The contents object stores an array of items, an array of children, and a reference to the next node in the linked list. A node will not increase or decrease its height subsequent to node creation. The array of items is always a non-null value. The array of children is null if-and-only-if the enclosing node is a leaf node. A search object is used to keep track of the position of some $v \in \mathtt{T}$ in relation to a specific node. The search object contains a reference to some node, a reference to a contents object that represents a snapshot of the node, and the index of element v relative to the items stored in the

```
1  class Node<T> {
2    volatile Contents<T> contents;
3  }
4  class Contents<T> {
5    final Object[] items;
6    final Node<T>[] children;
7    final Node<T> link;
8  }
9  class Search<T> {
10   final Node<T> node;
11   final Contents<T> contents;
12   final int index;
13 }
14 class HeadNode<T> {
15   final Node<T> node;
16   final int height;
17 }
18 private volatile HeadNode<T> root;
```

Figure 4.3: Declarations for a tree with key type `T`

contents object. A head node stores a node reference and an integer representing the height of the node. The skip tree declaration contains a single field, which is an atomic reference to the root of the tree.

Figure 4.3 shows class and field declarations. All final fields contain immutable state information. An immutable reference cannot be used to modify the object to which it refers, including the transitive state of the object. The 'final' type modifier does not guarantee immutability [102]. In order to show linearizability (Section 4.6) it is assumed that the contents of a node are an immutable triplet of items, children, and a link reference. Immutability is not enforced by the Java type system so it is enforced by the implementation. The `AtomicReferenceFieldUpdater` interface is used to enable atomic updates to the contents field of a node or to the root field of a tree. For space reasons we have omitted the declaration of the singleton class that represents the value $+\infty$. The singleton class implements the `compareTo` method such that the method will always return a value of 1. It is assumed that the runtime system performs garbage collection. The garbage collector prevents instances of the ABA problem [103] from occurring.

4.3 Tree Traversal

Given some $v \in \texttt{T}$ it is possible to determine if v is an element of the skip tree by traversing through the tree starting at the root. The *reachable* relation is defined to determine whether a node is connected to the root of the tree with some arbitrary number of intermediate reachable nodes. The `contains` operation will determine whether a leaf node containing v is reachable from the root node of the tree.

Definition 4.2. Let nodes n_i and n_j be at levels i and j, respectively, such that $i \geq j$. Define the relation *reachable* such that n_j is reachable from n_i if-and-only-if either $n_j \in tail(n_i)$ or n_j is reachable from n_k where n_k is a child reference of some node in $tail(n_i)$. Given an element $v \in \texttt{T}$ and some node n_i, then v is reachable from n_i if-and-only-if there exists a node $leaf$ at level 0 that contains the element v and $leaf$ is reachable from n_i.

Let h be the current height of the tallest level, and let A_h and B_h be the neighboring elements at level h that satisfy property (D3), ie. $A < v \leq B$. Let n_h be the node that contains element B_h. Node n_h is reachable from the root, because all nodes at level h are in the tail set of the first node of level h. Let A_{h-1} and B_{h-1} be the neighboring elements at level $h-1$ that satisfy property (D3) and let n_{h-1} be the node that contains element B_{h-1}. Property (D4) requires that $n_{h-1} \in tail(n_h)$ which is sufficient to show that n_{h-1} is reachable from n_h. Combining the two reachability arguments it can be shown that n_{h-1} is reachable from the root node. For all i such that $i \leq h$, by induction n_i is reachable from the root node where n_i is the node at the i^{th} level of the tree containing the neighboring element B_i. n_0 is the leaf node containing the neighboring element B_0. Property (D2) states that the leaf level does not contain duplicate elements. Property (D3) states that $(A_0, B_0]$ is the only interval at the leaf level that contains the value v. Therefore v is a member of the set if-and-only-if v is reachable from the root of the tree.

```
19 boolean contains(T v) {
20   Node<T> node = root.node;
21   Contents<T> contents = node.contents;
22   int i = search(cts.items, v);
23   while(contents.children != null) {
24     if (-i - 1 == contents.items.length)
25       node = contents.link;
26     else if (i < 0)
27       node = contents.children[-i - 1];
28     else
29       node = contents.children[i];
30     contents = node.contents;
31     i = search(contents.items, v);
32   }     // end traverse routing nodes
33   while(true) {
34     if (-i - 1 == contents.items.length)
35       node = contents.link;
36     else if (i < 0) return false;
37     else return true;
38     contents = node.contents;
39     i = search(contents.items, v);
40   }     // end traverse leaf nodes
41 }
```

Figure 4.4: Determining whether `v` is in the set

The `contains` method implements the search algorithm described in the previous paragraph. Figure 4.4 shows the code for the method. Starting at the root, the node η_i is located such that η_i satisfies property (D3) for level i of the tree. The `search` method used on Lines 22 and 39 is a shorthand for calling the `Arrays.binarySearch` method. The `search` method returns the index of the search key, if the key is contained in the array. Otherwise the method returns $-(\text{insertion point}) - 1$, where the insertion point is the index of the search key if the key were contained in the array. The `contains` method travels through the routing nodes until it eventually reaches a leaf node. When the method can no longer travel to a successor node in the leaf level, then it terminates and returns a boolean value indicating whether v is a member of the leaf level.

4.4 Insertion

When an element is inserted into the dense skip tree it is assigned a random height. The heights of the elements in a skip tree are distributed according to a geometric distribution:

$$\begin{aligned} \Pr(H = h) &= q^h p \text{ where } p + q = 1 \\ &\quad (p \text{ and } q \text{ are constants}) \end{aligned}$$

An element is inserted into the skip tree through successive inserting and splitting of linked list levels of the tree. It is forbidden to insert new elements into an empty node. The node with zero elements acts as the 'marker' of the Michael-Harris algorithm [34, 35] that forbids concurrent updates and signals lazy elimination of the node.

To insert an element at height h, first the element must be inserted at level 0, then the linked list must be split at level 0, then the element must be inserted at level 1, then the linked list must be split at level 1, and this process continues until the element is inserted at level h.

Figure 4.5 shows code for the `add` operation. The add operation consists of an initial call to `traverseAndTrack` and continues with alternating calls to `insertList` and `splitList`. The method `traverseAndTrack` is a specialized version of the tree traversal operation described in the previous section. A path is traversed from the root to the leaves of the tree, while references to the nodes that are to be updated get stored in an array for later use. The `moveForward` method is called when retrying an `insertList` or `splitList` operation. The `moveForward` method accepts a node and an element, and returns a Search object containing the first node of the current linked list level with an element x such that $x \geq v$.

The `insertList` method accepts four arguments: an element to insert ($v \in$ `T`), a

```
42 boolean add(T v) {
43   int height = randomLevel();
44   Search[] searchs = new Search[height+1];
45   traverseAndTrack(v, height, searchs);
46   boolean success;
47   success = insertList(v,srchs,null,0);
48   if(!success) return false;
49   for(int i = 0; i < height; i++) {
50     Node<T> right = splitList(v,searchs[i]);
51     insertList(v, searchs, right, i + 1);
52   }
53   return true;
54 }
55
56
57 void traverseAndTrack(T v,
58       int h, Search[] searchs) {
59   HeadNode<T> root = this.root;
60   if (root.height < h)
61     root = increaseRootHeight(h);
62   int height = root.height;
63   Node<T> node = root.node;
64   Search<T> result = null;
65   while(true) {
66     Contents<T> contents = node.contents;
67     int i = search(contents.items, v);
68     if (-i - 1 == contents.items.length) {
69       node = contents.link;
70     } else {
71       result = new Search<T>(node, contents, i);
72       if (height <= h)
73         searchs[height] = result;
74       if (height == 0)
75         return;
76       if (i < 0) i = -i - 1;
77       node = contents.children[i];
78       height--;
79     }
80   }
81 }
```

Figure 4.5: Inserting v into the set

hint at the correct node for insertion, a target height at which to insert, and a new child reference to insert along with the new element. If the target height is non-zero and the child node argument is **null**, then the previous `splitList` operation was unsuccessful. If the previous splitList operation was unsuccessful or if element v is located at the current level, then the method terminates. If the element v is greater than the largest element in the node, then the method moves forward in the linked list level. Otherwise it constructs a new contents object that contains the new element and child reference. The compare-and-swap operation is attempted using the node, the expected contents object, and the constructed contents object. If the compare-and-swap is unsuccessful then the contents of the node are re-read and the operation is retried. The non-Java operator $\cup_i$ used on Lines 95-96 represents the operation of insertion at position i. Given an array and an element v, the operator $\cup_i$ returns a new array that contains all the elements of the original array in the same order with the addition of the element v inserted at position i. When both arguments to the $\cup_i$ operator are **null**, then the operator is defined to return **null** as a matter of convenience.

The `splitList` method accepts two arguments: a partition element ($v \in$ `T`) and a hint at the correct node to split. The split operation will transform a single node into two nodes: a left partition node and a right partition node. The left partition node consists of all elements less than or equal to the partition element, and the right partition node consists of all elements greater than the partition element. The original node is transformed into the left partition node using a compare-and-swap operation. The right partition node is the return value of the method.

Splitting a node preserves the reachability relations of the tree. Let node A be split into the nodes A' and B. The set of children references in node A is equal to the union of the children references of A' and B (Lines 119-120). The link reference of A and link reference of B refer to the same node. If an element $v \in$ `T` was reachable

from node A, then the element is reachable from either one of the child references of A or the link reference of A. The set of references is preserved across the split operation, therefore the element is reachable from either one of the child references of A', or one of the child references of B, or the link reference of B. Similarly, if an element $v \in \texttt{T}$ was not reachable from node A, then the element is not reachable from A' nor B.

The `insertList` method is written on the premise that inserting into a leaf node of the tree is nearly identical to inserting into a non-leaf node of the tree. In the case of the leaf node, there are no child references to modify. In the case of a non-leaf node, a new child reference is added to the node. The new child reference, *child*, is an argument to the `insertList` method. It will become the child reference to the right of the new element $v \in \texttt{T}$. The new child reference is created as a result of a split operation, therefore when the node was created it was true that $\forall x \in child, v > x$. By property (D5) of the lock-free skip tree, it will always remain true that the element v is greater than the elements of *child*. Let u and w be the members of level i that are the immediate neighbors of the new element v. Prior to the insertion at level i of the tree, there exists a node at level $i-1$ that is the child reference of elements u and w that satisfies property (D4) for values x such that $u < x \leq w$. A pre-condition of the `insertList` method at level i is the application of the `splitList` method at level $i-1$. The `insertList` method adds v to the level i of the tree and assigns a left child reference and right child reference to the two pairs of elements, (u, v) and (v, w), respectively. If the original child reference of (u, w) contained elements that were greater than v, these elements were removed from the child reference at the completion of the `splitList` method. The child reference of (u, w) prior to the `insertList` method is assigned as the left child reference of v upon the successful completion of the method. A post-condition of the split operation is that the new node that has been created contains the lowest

values that are strictly greater than v. The node created by the split operation will become the right child reference of v upon the successful completion of the `insertList` method.

```
82  boolean insertList(T v, Search[] searchs,
83        Node<T> child, int h) {
84    if (child == null && h > 0)
85      return false;
86    Search result = searchs[h];
87    while(true) {
88      Node<T> node = result.node;
89      Contents<T> contents = result.contents;
90      int i = result.index;
91      if (i >= 0)
92        return false;
93      else if (i > -contents.items.length - 1) {
94        i = -i - 1;
95        Object[] items = contents.items ∪_i v;
96        Node<T>[] children = contents.children ∪_{i+1} child;
97        Contents<T> update =
98          new Contents<T>(items, children, contents.link);
99        if (node.casContents(contents, update)) {
100         result = new Search<T>(node, update, i);
101         searchs[h] = result;
102         return(true);
103       } else result = moveForward(node, x);
104     } else result = moveForward(node, x);
105   }
106 }
107
108 Node<T> splitList(T v, Search<T> result) {
109   while(true) {
110     Node<T> node = result.node;
111     Contents<T> contents = result.contents;
112     int i = result.index;
113     int len = contents.items.length;
114     if (i < 0) return(null);
115     else if (len < 2 || i == (len - 1))
116       return(null);
117     Object[] lItems = contents.items[0:i];
118     Object[] rItems = contents.items[i+1:len-1];
119     Node<T>[] lChildren = contents.children[0:i];
120     Node<T>[] rChildren = contents.children[i+1:len-1];
121     Contents<T> rContents =
122       new Contents<T>(rItems, rChildren, contents.link)
123     Node<T> right = new Node<T>(rContents);
124     Contents<T> left =
125       new Contents<T>(lItems, lChildren, right);
126     if (node.casContents(contents, left))
127       return(right);
128     else result = moveForward(node, x);
129   }
130 }
```

Figure 4.6: Inserting and splitting a single level

4.5 Deletion and Node Compaction

Figure 4.7 shows the code for the `remove` operation. An element $v \in$ `T` is removed by searching for the presence of the element at the leaf level of the tree. If v is found, then construct a new array of elements minus v and update the node with a compare-and-swap. If the compare-and-swap is successful then return `true`. Otherwise retry the operation. If v is not found, then the `remove` operation returns `false`. The `moveForward` method is invoked when retrying a `remove` operation. The `moveForward` method accepts a node and an element, and returns a Search object containing the first node of the current linked list level with an element x such that $x \geq v$. The non-Java operator $\setminus$ used on Line 137 represents the set difference operation. Given an array (that does not hold duplicates) and an element in the array, the set difference operator returns a new array that does not contain the element.

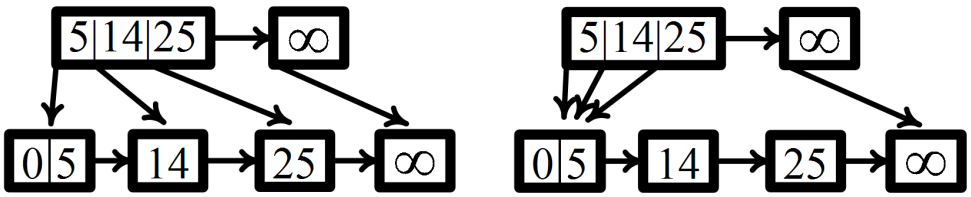
The `remove` operation can introduce empty nodes and suboptimal child references. Empty nodes are relatively straightforward to eliminate. As stated in the previous section, it is forbidden to insert new elements into an empty node. Therefore a link or child reference to an empty node can be replaced by a reference to the immediate successor of the empty node. The elimination of an empty node is shown in Figure 4.9a.

Definition 4.3. Let m contain a child reference to some node n. Let n' be the successor node of n. The child reference is an *optimal child reference* if-and-only-if replacing it with a reference to n' results in a violation of the skip tree properties. An *optimal node* is a node that contains only optimal child references.

Figure 4.8 illustrates two skip trees with identical nodes and highlights the differences between optimal and suboptimal child references. Optimal child references are necessary to preserve the expected $O(\log n)$ cost for sequential `contains`,

```
131 boolean remove(T v) {
132   Search search = traverseAndCleanup(v);
133   while(true) {
134     Node<T> node = search.node;
135     Contents<T> contents = search.contents;
136     if (search.index < 0) return(false);
137     Object[] items = contents.items \ v;
138     Contents<T> update =
139      new Contents<T>(items,null,contents.link);
140     if (node.casContents(contents, update))
141       return(true);
142     search = moveForward(node, v);
143   }
144 }
145
146 Search<T> traverseAndCleanup(T v) {
147   Node<T> node = root.node;
148   Contents<T> contents = node.contents;
149   Object[] items = contents.items;
150   int i = search(items, v);
151   T max = null;
152   while(contents.children != null) {
153     if (-i - 1 == items.length) {
154       if (items.length > 0)
155         max =(T) items[items.length-1];
156       node = cleanLink(node, contents);
157     } else {
158       if (i < 0) i = -i - 1;
159       cleanNode(node, contents, i, max);
160       node = contents.children[i];
161       max = null;
162     }
163     contents = node.contents;
164     items = contents.items;
165     i = search(items, v);
166   }     // end traverse routing nodes
167   while(true) {
168     if (i > -contents.items.length - 1)
169       return new Search<T>(node, contents, i);
170     node = cleanLink(node, contents);
171     contents = node.contents;
172     i = search(contents.items, v);
173   }     // end traverse leaf nodes
174 }
```

Figure 4.7: Deleting v from the set



(a) optimal references (b) suboptimal references

Figure 4.8: Optimal versus suboptimal child references.

`add`, and `remove` operations. Four transformations can be applied to the tree to eliminate empty nodes and suboptimal child references. These transformations are shown in Figure 4.9. Node compaction can be applied in a lazy manner and is performed alongside the `remove` operation. An optimal node can alternatively be defined as a node that does not benefit from any of the four node compaction operations.

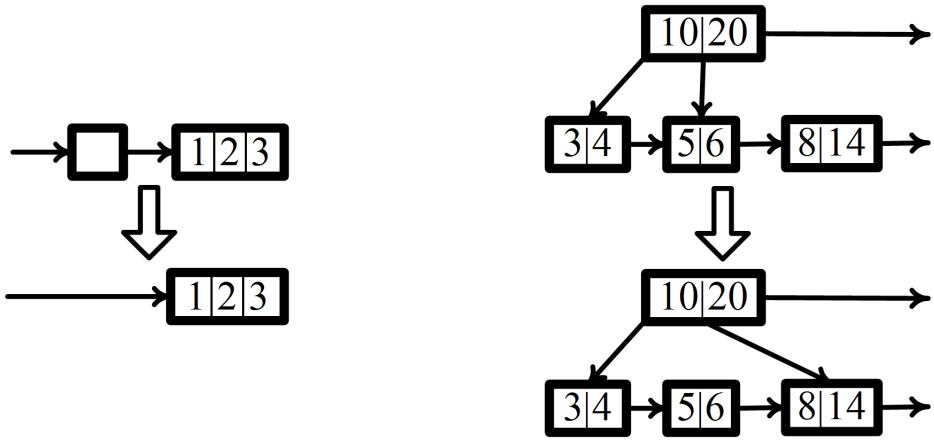
Given a pair of adjacent elements A and B and their shared child reference to node n, the child reference is suboptimal if $\max(n) < A$. A suboptimal child reference can be repaired as shown in Figure 4.9b. The reachability relationship is preserved during the transformation because of property (D5) of the skip tree definition. If $\max(n) < A$ at some point in time, then $\max(n) < A$ in all possible futures.

Any pair of adjacent elements in a linked list level share exactly one child reference. For example in Figure 4.2 the child reference of elements 3 and $+\infty$ at level 1 is the node containing $\{6, +\infty\}$ at level 0. As shown in Figure 4.9c, the node compaction operations can lead two adjacent child references to point to the same node. Given adjacent elements A, B, and C such that the child reference between A and B is equal to the child reference between B and C, then element B can be dropped from the node. In Figure 4.9c, β is the shared child reference of elements A, B, and C. If β is reachable by the intervals $(A, B]$ and $(B, C]$, then β must be reachable by the interval $(A, C]$. It is unsafe to apply this transformation to a node

that contains a single element. Assume element B from the previous example is the single element of some node n_B, and that element A is contained in node n_A and element C in node n_C. The intervals $(A, B]$ and $(B, C]$ satisfy property (D4). If the leftmost child reference in n_C is a suboptimal child reference, and n_C is repaired while concurrently node n_B is removed, then the interval $(A, C]$ may no longer satisfy property (D4).

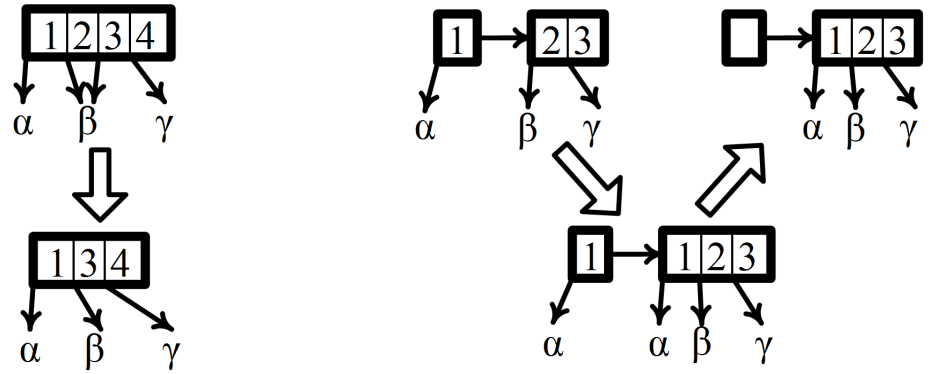
A copy and delete strategy is used to eliminate a node that contains a single element. This process is illustrated in Figure 4.9d. The inclusion of duplicate elements in routing nodes (Theorem 4.1) makes it possible to move the element to the successor node. Consecutive elements with identical values form an interval of the form $(X, X]$ that will never contain any values that satisfy property (D3). The purpose of introducing duplicate elements is to allow for suboptimal child reference elimination when the parent node of the suboptimal child contains a single element. Although element migration can be applied to any node that contains a single element, in practice it is applied only when child references α and β are pointing to the same node. The purpose of element migration is to eliminate a suboptimal node that contains a single element. There is no compelling argument for removing an optimal node that contains a single element.

The deletion of the singleton element from a routing node can interfere with shared child elimination. Let us reuse the earlier example with elements A, B, and C stored in nodes n_A, n_B, and n_C. Thread 1 is performing node migration and has copied element B to node n_C. Thread 2 detects the duplicate copy of element B on node n_C and begins shared child elimination. Thread 1 finishes the node migration operation by removing the original copy of element B from node n_B. A third concurrent thread of execution attempts suboptimal child reference elimination on node n_C. The thread has read element B from node n_B before the element was deleted. As both copies of B have been eliminated, the correct max element that is



(a) empty node

(b) suboptimal child reference



(c) shared child

(d) element migration

Figure 4.9: Four types of node compaction

a predecessor of n_C is less than B. B is too large to use as a max predecessor of node n_c and therefore the tree can reach an inconsistent state. This error is prevented by forbidding shared child elimination on a node that contains two elements. To remove a shared child on the first pair of elements in a node, element migration is applied to the rightmost element of the node.

There are three cases to consider regarding shared child elimination. The shared child appears either on a node with one element, or appears as the first child on a node with two or more elements, or appears on a node with two or more elements but not as the first child. If a node has exactly one element, then shared child elimination could be performed if the node and its immediate successor could be together updated atomically. As the double compare-and-swap operation is not supported in our runtime or by commonly available hardware [13], shared child elimination on a singleton node is performed using element migration. If the node has two or more elements and the shared child appears as the first child reference, it is possible that the child reference is a by-product of an ongoing element migration operation. The shared child elimination procedure maintains the reachability relationships of the skip tree based on the assumption that the immediate predecessor element preserves property (D5). With the exception of shared child elimination, all other node compaction operations do not remove the rightmost element from a node. As we have shown, deletion of the singleton element from a routing node can interfere with shared child elimination. Therefore, the deletion of the first child of a node is forbidden.

4.6 Correctness

Liveness: We will show that `add` and `remove` are lock-free operations, and `contains` is a wait-free operation. Lock-free operations guarantee that at least one thread

will complete in a finite number of steps. Wait-free operations guarantee that all threads will complete in a finite number of steps.

The `add` and `remove` operations each perform a finite number of traversals through the tree. The `add` operation performs up to two passes through the tree. A first pass inserts an element at the appropriate levels and a second pass splits each level after the insertion pass has completed. A `remove` operation consists of one pass through the tree that performs node compaction and possibly deletes an element from the leaf level.

In both `add` and `remove` operations a node is visited at most once per traversal through the tree. Each visit of a node performs at most one successful compare-and-swap operation. The failure of one compare-and-swap operation implies the success of a compare-and-swap operation from another concurrently executing operation. Therefore system-wide progress is guaranteed. The `contains` operation performs one pass through the tree. Each node of the tree is visited at most once, and the contents of each node are read at most once per visit. The operation performs no conditional atomic operations. Therefore per-thread progress is guaranteed.

Linearizability: To demonstrate the skip tree algorithm is linearizable [40] it is sufficient to define a linearization point for each operation and then show that operations produce equivalent results to a sequential execution in which the operations appear to occur instantly at the linearization point.

The `contains`, `add`, and `remove` operations are linearized with two possible actions. If the operation does not change the state of the abstract data type, then the operation has been linearized by a volatile read on the contents of a node. If the operation changes the state of the abstract data type, then the linearization point occurs at the success of a compare-and-swap operation.

The linearization point for `contains(`v`)` occurs on the last volatile read of the contents of a node, on Line 21, 30, or 38. At any snapshot in time there is at most one leaf node that contains v. The reachability relation from the root node to the node containing v is preserved by `add` and `remove` operations to ensure that v is reachable from the root of the node if-and-only-if there exists a leaf node that contains v.

The linearization point for `add(`v`)` occurs during the call to the `insertList` method at the leaf level (Line 47). The linearization point depends on whether the `add` operation is successful. A successful `add` operation is linearized on the compare-and-swap operation that inserts v into a leaf node. An unsuccessful `add` operation is linearized on the read of the contents of a leaf node that leads to the discovery of v as a element of the leaf node. A successful `remove` is linearized at Line 140 on the compare-and-swap that removes v from a leaf node. An unsuccessful `remove(`v`)` operation is linearized at Line 163 or 171 at the volatile read of a leaf node that does not contain v and contains some element greater than v.

The four node compaction operations preserve the structural consistency of the skip tree. Empty node elimination does not alter the ordering relationships among elements in a linked list level. Nor does empty node elimination affect any of the child references from one level to the next level below. Suboptimal child reference elimination can occur on a child reference n at level $i-1$ that is shared by neighboring elements A and B at level i if-and-only-if for all values v such that $A \leq v < B$, the corresponding node n' at level $i-1$ that satisfies property (D3) for value v is a successor node of n. Since n' must be in the $tail(n)$ by property (D4) and $n' \neq n$, then it is safe to move the child reference of A and B to the link reference of node n. If $n' \neq n$ holds at some point in time for all values v, then $n' \neq n$ holds in all possible futures by property (D5).

Shared child elimination preserves property (D4) using a similar argument that

was applied to suboptimal child reference elimination. Shared child elimination can occur on a child reference n at level $i-1$ that is shared by neighboring pairs (A, B) and (B, C) because property (D4) applies to the child reference n for neighboring pairs (A, B) if-and-only-if property (D4) applies to the child reference n for neighboring pairs (B, C). It is trickier to preserve property (D3) in the shared child elimination transformation. Shared child elimination is the only node compaction transformation that eliminates a non-duplicated element from a linked list level. Therefore it must be shown that, given some value v, there exists exactly one pair of adjacent elements α and β such that $\alpha < v \leq \beta$.

Let A, B, and C be three neighboring elements such that child reference n is shared by neighboring pairs (A, B) and (B, C). If A and B are on the same node, then these neighboring elements and child reference n satisfy properties (D3) and (D4) if-and-only-if B and C and child reference n satisfy properties (D3) and (D4). If A and B are not on the same node, then it is possible that A is identical to B, and that A was copied using element migration but has not been deleted. As described in the previous section, it is for this reason that shared child elimination cannot be applied to the first child reference of a node.

Element migration is a process that occurs in two atomic operations. The first operation introduces a copy of some element A and a child reference n into a linked list level. The copy of the element does not interfere with property (D3) as the interval $(A, A]$ does not contain any values. And the copy of child reference n cannot be traversed as the interval $(A, A]$ is empty so therefore there are no values that satisfy the predicate of property (D4). The second atomic operation removes the original element A and child reference n from the singleton node. Let A be the original element and A' be the copy of element A. If A is concurrently participating in shared child elimination then A it will not be eliminated by the concurrent operation as the rightmost element of a node is always preserved during shared

child elimination. It is forbidden for A' to participate in shared child elimination because A' is the first element in a node. Given this restriction, it is safe to eliminate A with the guarantee that A' will remain to preserve the interval property (D3).

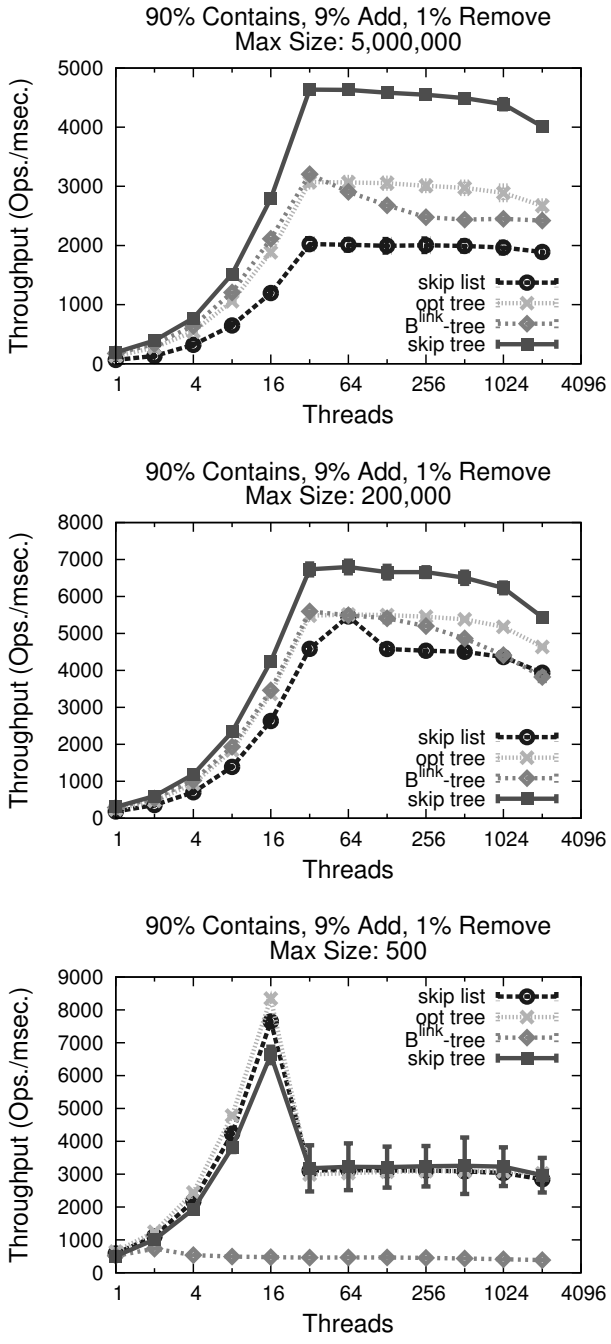


Figure 4.10: Read-dominated synthetic benchmarks on Sun Fire T1000

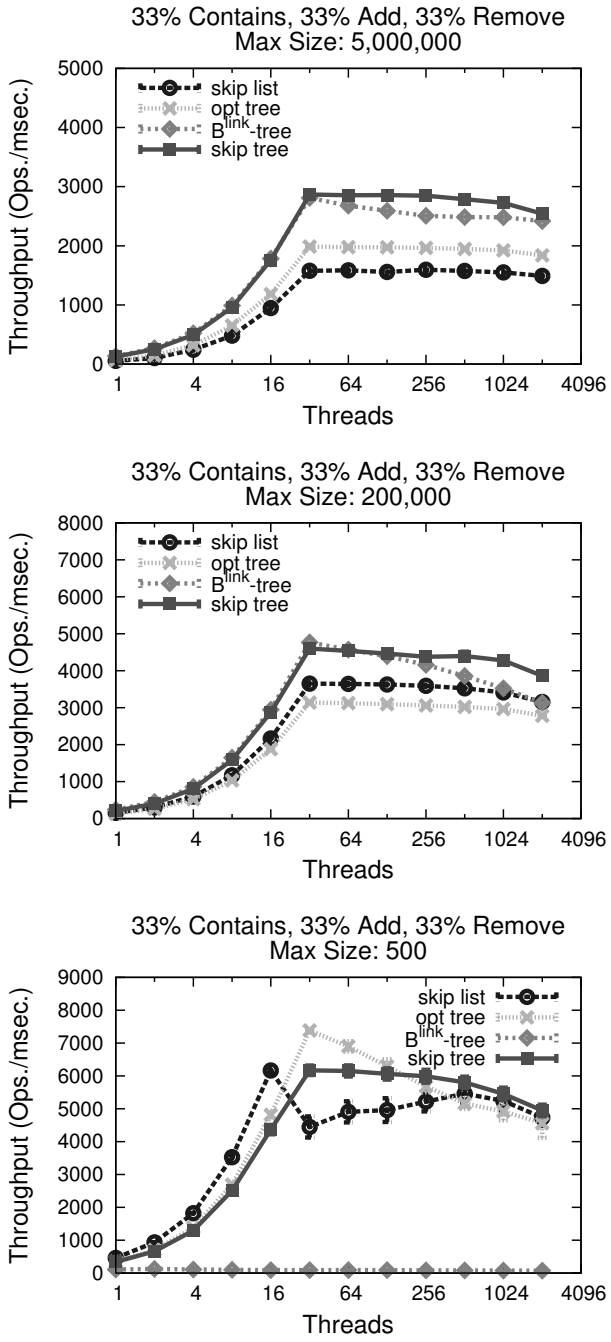


Figure 4.11: Write-dominated synthetic benchmarks on Sun Fire T1000

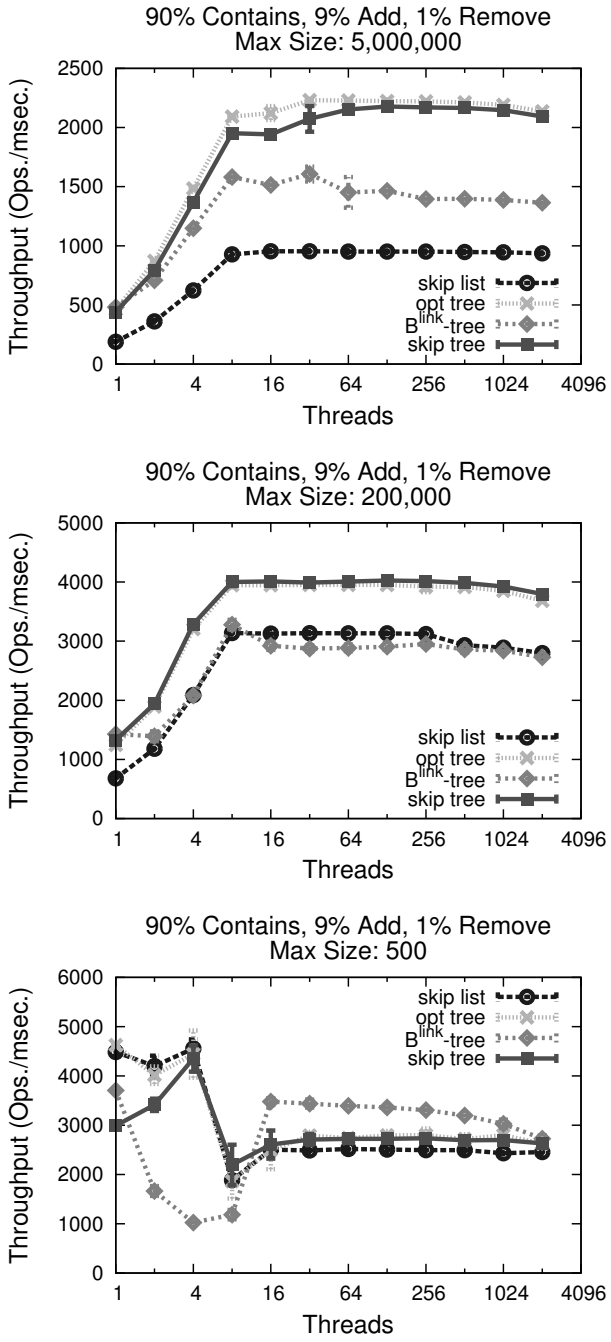


Figure 4.12: Read-dominated synthetic benchmarks on quad core Intel Xeon

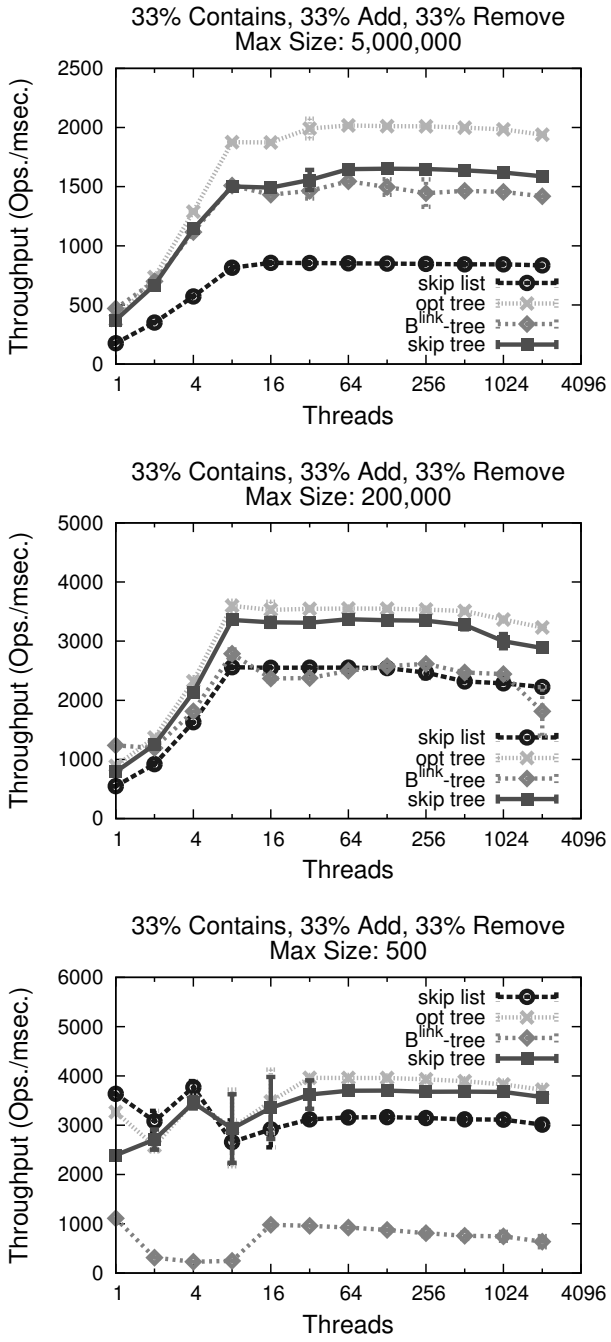


Figure 4.13: Write-dominated synthetic benchmarks on quad core Intel Xeon

4.7 Synthetic Benchmarks

Performance analysis has been conducted with an experimental design that is popular in the concurrent data structures literature [35, 104, 95]. Synthetic workloads are created that vary in proportions of `contains`, `add`, and `remove` operations and in the number of unique elements stored by the data structure. Half of the workloads use a $\frac{90}{100}$:$\frac{9}{100}$:$\frac{1}{100}$ ratio of operations. The other half use a $\frac{1}{3}$:$\frac{1}{3}$:$\frac{1}{3}$ ratio of operations. 5,000,000 operations are executed in each independent trial, while the total throughput of the data structure as measured by the number of concurrently executing threads varies from 1 to 2,048.

The maximum size of the tree is determined through selection of random elements from a uniform distribution with a range of 500, or 200,000, or 2^{32} integers. For each scenario, an upper bound on the size of the tree is the minimum of the range of input integers (500, 200,000 or 2^{32}) and the number of operations performed (5 million). Each independent trial is repeated 64 times. Integers that are designated for a `contains` or `remove` operation are pre-loaded into the tree prior to the beginning of a trial. The smallest working set size is selected to fit entirely within the L2 cache and the largest working set size is set to greatly exceed the L2 cache size.

Benchmarks were evaluated on a Sun Fire T1000 and an Intel Xeon L5430. The benchmarks were executed on the 32-bit server version of the HotSpot Java Virtual Machine version 1.6.0 update 16. Explicit parameters for the virtual machine are 2 GB heap size and 128 kB thread stack size. The Sun Fire has 8 UltraSPARC T1 cores at 1.0 GHz and 32 hardware threads. The cores share a 3 MB level-2 unified cache. The operating system version on the Sun Fire T1000 is Solaris 10. The Xeon L5430 has 4 cores at 2.66 GHz and 8 hardware threads. Each pair of cores share a 6 MB level-2 unified cache. The operating system distribution is CentOS release 5.3 with Linux kernel 2.6.29-2.

We compare four implementations of linearizable concurrent ordered sets:

- skip list - the `ConcurrentSkipListSet` in the `java.util.concurrent` library. Written by members of the JCP JSR-166 Expert Group.
- skip tree - our lock-free skip tree algorithm.
- opt tree - the optimistic relaxed balance AVL tree algorithm of Bronson et al. [104].
- B^{link}-tree - a concurrent B-tree algorithm developed by Lehman and Yao [101] and refined by Sagiv [97].

Figures 4.10 - 4.13 show the results of the synthetic benchmarks on the Sun Fire T1000 and the quad core Intel Xeon. Tic marks denote the mean of the repeated experiments and error bars denote the standard deviation. In all graphs, a higher value on the vertical axis denotes improved performance.

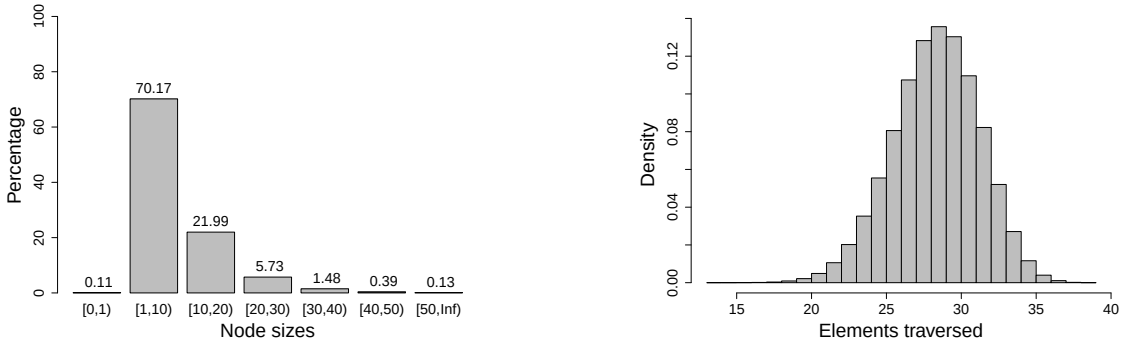
The skip tree structure is controlled by a single parameter, q, the failure rate of the underlying geometric distribution. The B^{link}-tree is also controlled by a single parameter, M, the minimum node size. Parameter variations for each of the six scenarios were conducted. The parameter value with the best average performance was selected ($q = 1/32$, $M = 128$).

On the Sun Fire T100, using a maximum set size of 5,000,000 elements, the peak throughput of the skip tree is 229% relative to the skip list on the read-dominated scenario and 181% on the write-dominated scenario. The peak throughput of the opt tree is 151% and 123% respectively relative to the skip list in the two scenarios, and the relative peak throughput of the B^{link}-tree is 158% and 184% respectively. The skip tree exhibits the greatest improvement to system throughput under read-dominated scenarios with a large working set size. Averaged over all scenarios, the skip tree peak throughput is 141% relative to the skip list and the opt tree relative throughput is 115%. Averaged over all scenarios excluding the maximum

range of 500 elements, the relative peak throughput of the B^{link}-tree is 143%. Averaged over all scenarios the relative peak throughput of the B^{link}-tree drops to 97%. The B^{link}-tree performs wait-free searches when implemented for the memory-disk boundary where a page can be accessed from disk atomically. When implemented in main memory, the tree uses shared reader-writer locks [98, 100]. The reader-writer lock strategy is a bottleneck when there are only a handful of nodes in the data structure.

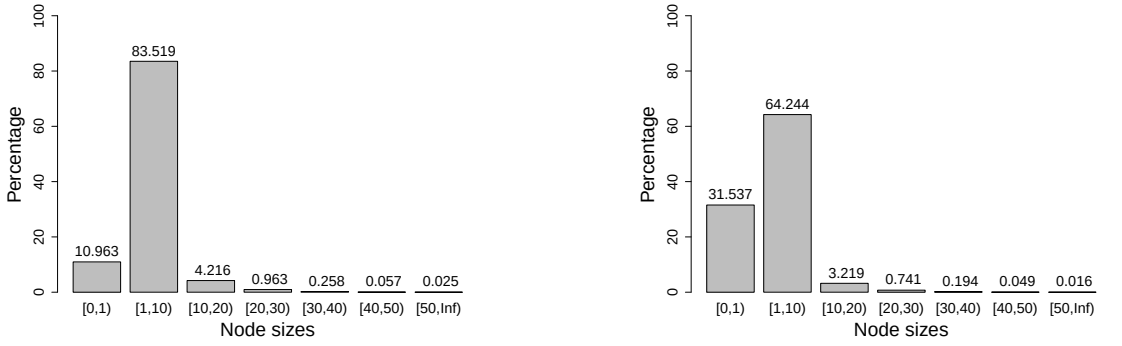
On the Xeon L5430, the peak throughput of the skip tree is 199% relative to the skip list on the read-dominated scenario and 125% on the write-dominated scenario, using a maximum set size of 5,000,000 elements. The peak throughput of the opt tree is 234% and 235% relative to the skip list in the two scenarios, and relative throughput of the the B^{link}-tree is 168% and 147%. The opt tree has the greatest improvement relative to the other data structures on the Intel processor. Averaged over all scenarios on the Xeon L5430, the peak throughput of the skip tree is 130% relative to the skip list and the peak throughput of the opt tree is 167% relative to the skip list. The skip tree and the opt tree yield almost equal performance on the Intel Xeon, and the skip tree outperforms the opt tree on the Sun Fire T1000.

The distribution of node sizes at the conclusion of the 90% `contains`, 9% `add`, and 1% `remove` synthetic workload with $q = 1/8$ is shown in Figure 4.14a. The observed mean node size is 7.92 with a standard deviation of 7.41. The expected mean node size is $1/q = 8$ and the expected standard deviation is $\sqrt{p}/q = 7.48$. The percentage of empty nodes is 0.11%, which can be attributed to the small fraction of `remove` operation in this workload. The number of nodes and the number of elements traversed per `contains` operation have been measured. 99.998% of all the 4.5 million `contains` operations in this workload encountered exactly 8 nodes during their traversal of the skip tree. This observation closely matches the expected



(a) node size distribution (b) search element distribution

Figure 4.14: Statistics for 90% `contains`, 9% `add`, 1% `remove` scenario



(a) with node compaction (b) without node compaction

Figure 4.15: Node sizes for 0% `contains`, 20% `add`, 80% `remove` scenario

number of encountered nodes which is $\lceil \log_8 5,000,000 \rceil = 8$. The distribution of the number of elements encountered per `contains` operation is shown in Figure 4.14b. The number of elements encountered follows a normal distribution with a mean value of 28.8.

To measure the effectiveness of the node compaction algorithm, the distribution of node sizes was measured after a 0% `contains`, 20% `add`, and 80% `remove` synthetic workload with $q = 1/8$. Node compaction was enabled in a first experiment and disabled in a second experiment. The node size distributions are shown in Figures 4.15a and 4.15b. A x2.9 reduction in the number of empty nodes is observed when node compaction is enabled.

The performance of sequential iteration through the set is examined while concurrent operations are performed. The 90% `contains`, 9% `add`, and 1% `remove` workload is selected with a maximum set size of 5,000,000 elements. A single thread continuously iterates over the elements of the set. Competing threads perform `contains`, `add`, and `remove` operations. The throughput of the iterator thread is shown in Figure 4.16. To measure the performance of sequential iteration, the opt tree is replaced by the snap tree, an extended algorithm that provides support for fast cloning and snapshots. Our experiments confirmed the original findings that the snap tree outperforms the opt tree for sequential iteration [104]. The skip tree shows a 18% improvement over the skip list with zero thread contention and a 97% improvement at the highest thread contention. The snap tree shows a 29% decrease in performance over the skip list at zero thread contention and a 25% improvement at the highest thread contention.

Comparing the synthetic benchmarks of this chapter and the previous chapter, it can be concluded that the lock-free skip tree exhibits the same performance advantages as the optimistic skip tree in read-dominated workloads, and does not have a reduced throughput as compared to the optimistic skip tree in write-

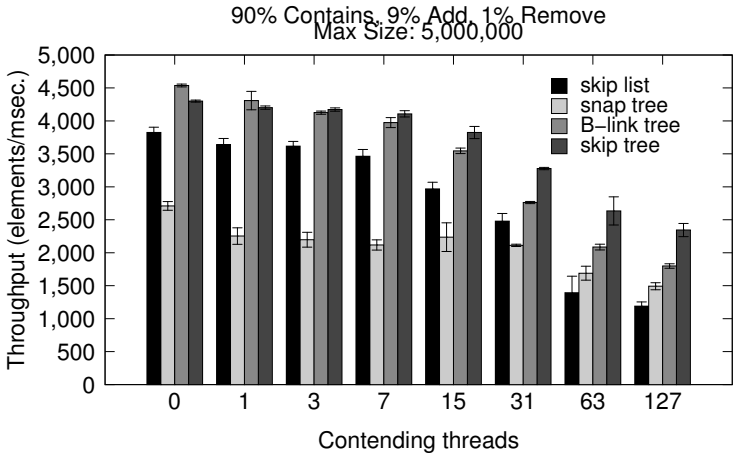


Figure 4.16: Iteration throughput of a single thread.

dominated workloads. On the Sun Fire T1000, the peak throughput of the lock-free skip tree is 229% relative to the lock-free skip list on the read-dominated workloads with 5,000,000 elements. The peak throughput of the optimistic skip tree is 144% relative to the skip list on the same workload. On the write-dominated workload with 512 elements, the lock-free skip tree exhibits the same throughput as the skip list to within 1%, but the optimistic skip tree exhibits only 5% of the performance of the skip list. The lowest peak throughput of the lock-free skip tree relative to the skip list across all synthetic benchmarks is 87% on the read-dominated workload with 512 elements. The synthetic benchmarks have shown that a lock-free cache-conscious data structure can perform up to x2.3 better in some workloads compared to the state of the art with only a 13% maximum penalty across all workloads.

The synthetic benchmarks have shown that the design of cache-conscious concurrent data structures for many-core systems exhibits significant performance improvements over the state of the art in lock-free data structure designs. Benchmarks that perform synthetic operations should be interpreted as measurements of performance that lack semantic context or purpose. The benchmark results can be interpreted across all application developers to estimate the utility of the algo-

rithm for the specific needs of the developer. The disadvantage of the synthetic operations is that they do not have any explicit meaning. As such, some caution must be exercised when projecting the synthetic benchmarks onto a specific application domain. In the next chapter, we study a class of NP-hard problems that can be solved using a linearizable concurrent priority queue. We study four examples from this class of problems, and compare the relative performance of the lock-free skip tree and skip list when used as concurrent priority queues.

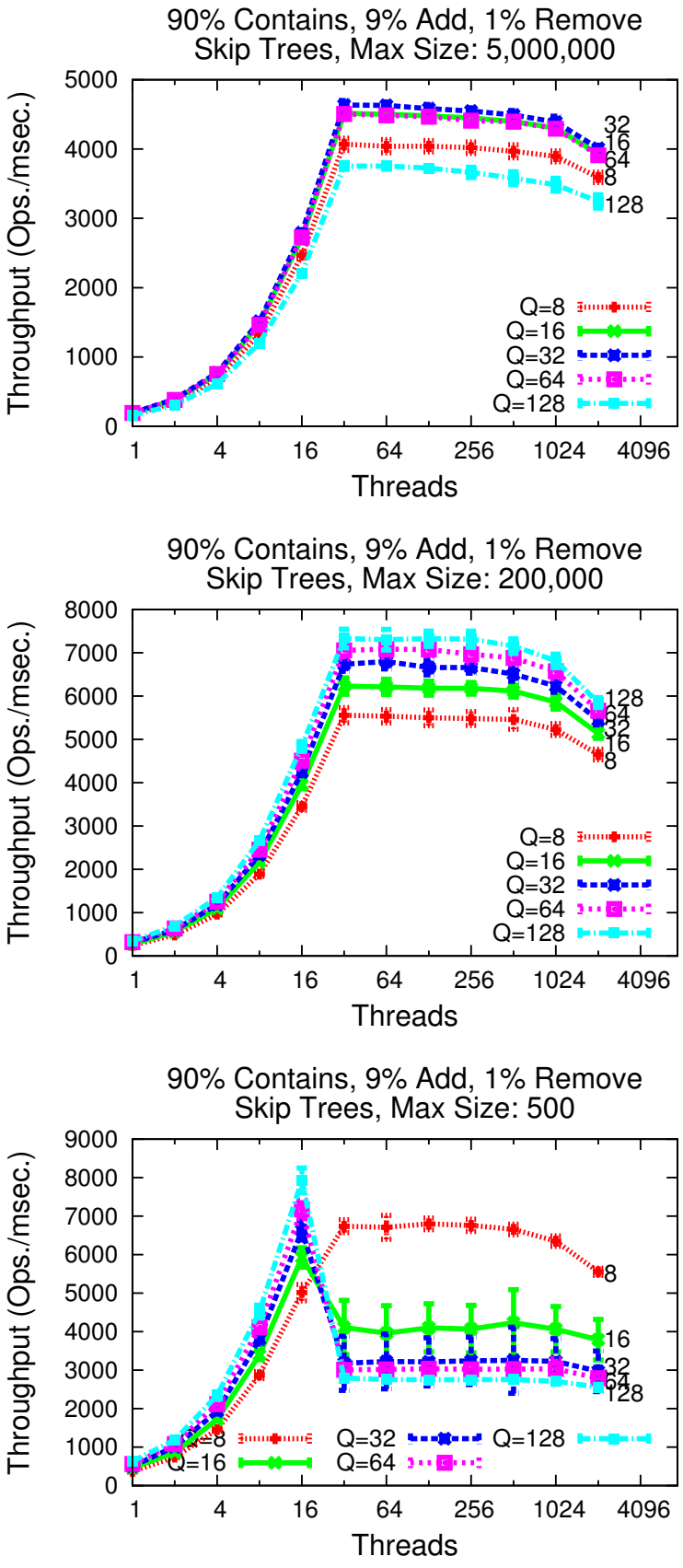


Figure 4.17: Parameter variations on the Sun Fire T1000 (read-dominated)

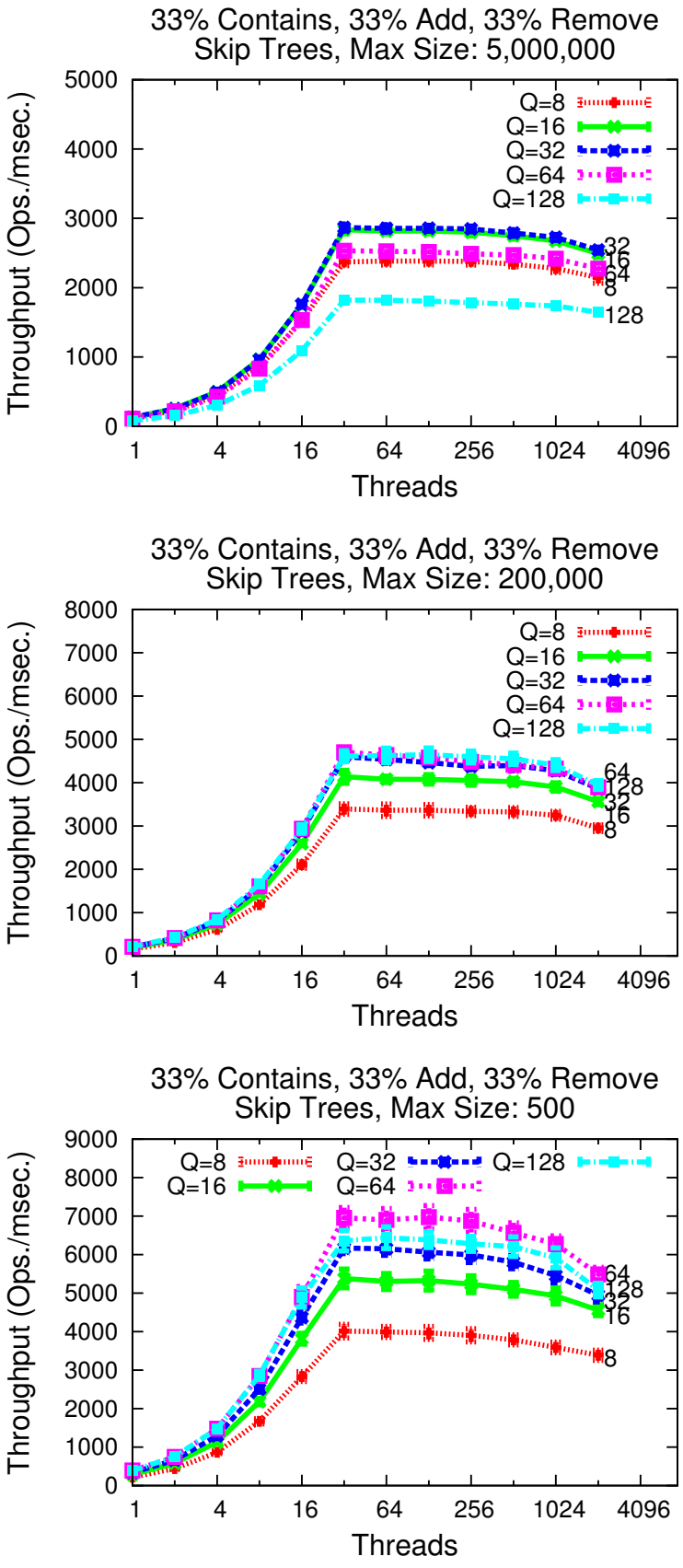


Figure 4.18: Parameter variations on the Sun Fire T1000 (write-dominated)

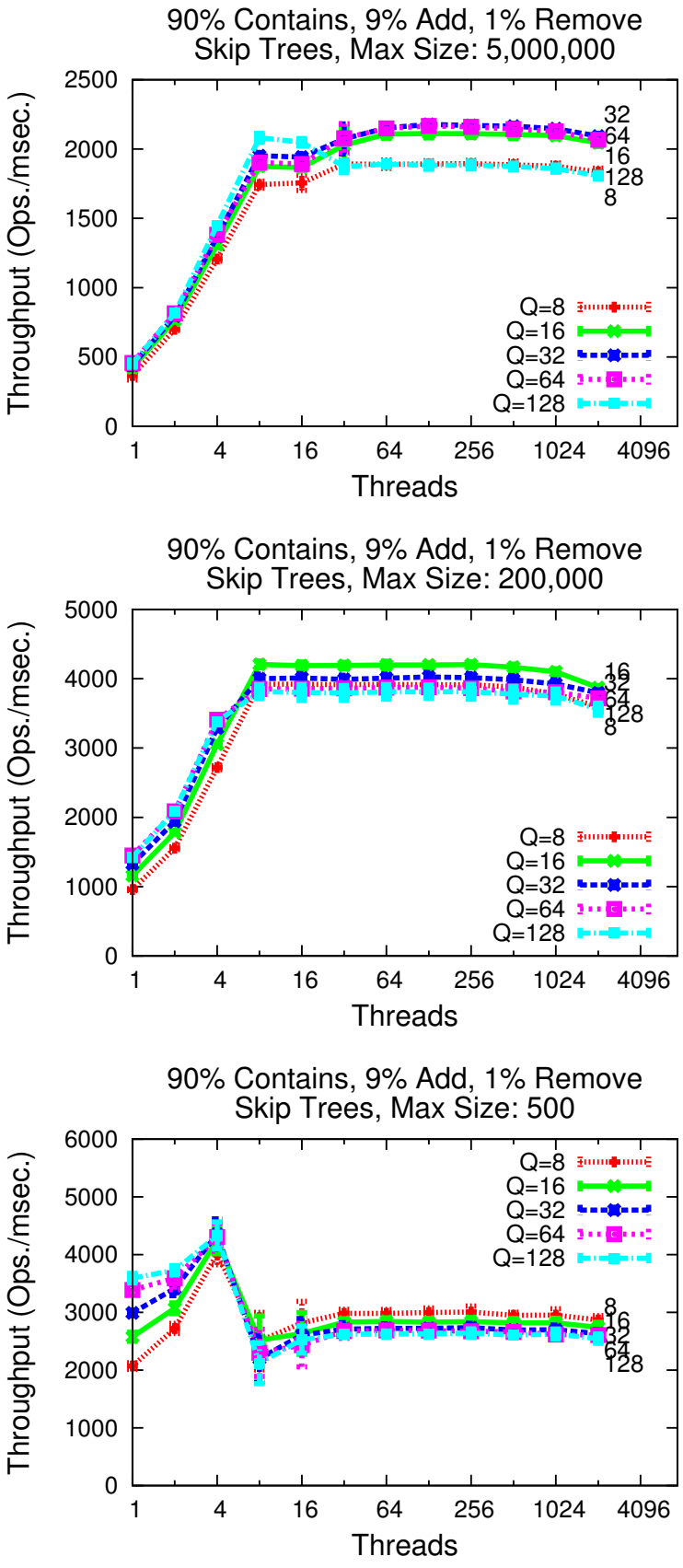


Figure 4.19: Parameter variations on the quad core Intel Xeon (read-dominated)

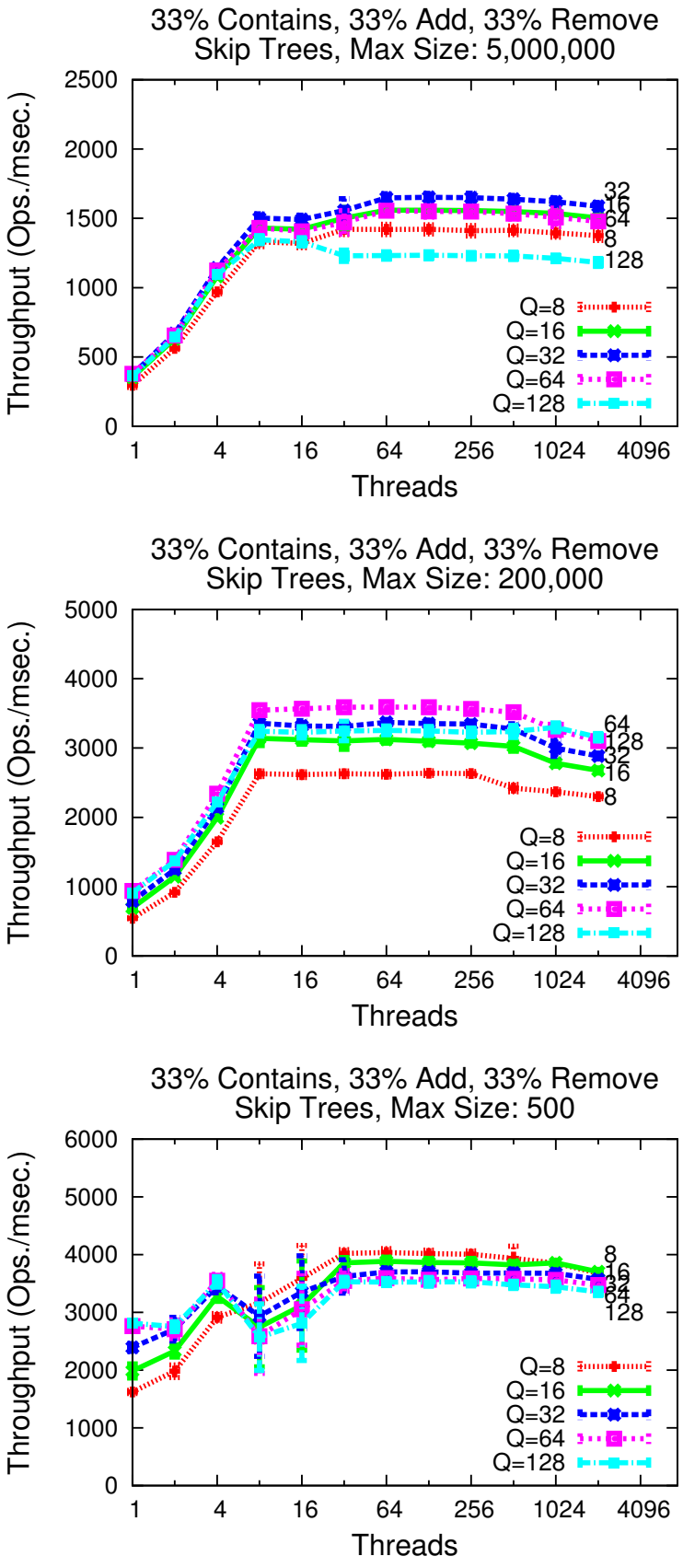


Figure 4.20: Parameter variations on the quad core Intel Xeon (write-dominated)

Chapter 5

Application Benchmarks

In the previous chapter, we showed using a series of synthetic benchmarks that the lock-free skip tree outperformed the lock-free skip list in large working set sizes as measured by operations per second as the quantity of concurrent threads is varied. The disadvantage of synthetic benchmarks is that they do not have any explicit meaning. As such, some caution must be exercised when projecting the synthetic benchmarks onto a specific application domain. In this chapter, we identify a class of problems that can be used to characterize the relative merits of the lock-free skip tree as compared to the lock-free skip list. We selected four NP-hard problems to solve using a parallel branch-and-bound technique: N puzzle, graph coloring, asymmetric traveling salesman, and 0-1 knapsack. In a series of four parallel branch-and-bound applications, two of the applications are x2.3 and x3.1 faster when using the skip tree as a concurrent priority queue as compared to the lock-free skip list. In a shared-memory supercomputer architecture the two branch-and-bound applications are x1.6 and x2.1 faster with the skip tree versus the skip list running at 80 hardware threads.

In this chapter we study the relative performance of a parallel branch-and-bound solver on four NP-hard problems using a lock-free skip list and lock-free

skip tree as concurrent priority queues. The four benchmarks are evaluated on a Sun Fire T1000 and an Intel Xeon L5430. The Sun Fire has 8 UltraSPARC T1 cores at 1.0 GHz and 32 hardware threads. The cores share a 3 MB level-2 unified cache. The operating system version on the Sun Fire T1000 is Solaris 10. The Xeon L5430 has 4 cores at 2.66 GHz and 8 hardware threads. Each pair of cores shares a 6 MB level-2 unified cache. The operating system distribution is CentOS release 5.3 with Linux kernel 2.6.29-2. The Sun Fire T1000 benchmarks are run using the 64-bit build of the HotSpot Java virtual machine version 1.6.0_18 with command-line arguments "-Xmx7G -XX:+UseParallelGC -XX:+UseParallelOldGC". The Intel Xeon benchmarks are run using the 64-bit build of the HotSpot Java virtual machine version 1.6.0_20 with command-line arguments "-Xmx43G -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:ParallelGCThreads=8."

To evaluate the relative performance of the lock-free skip tree as a concurrent priority queue, we have created a synthetic branch-and-bound application. The synthetic application is a simplification of a real branch-and-bound application. It has been designed to test three hypotheses of the branch-and-bound applications: (1) the distribution of lower bounds of the candidates in the search space affects the performance of the skip tree; (2) the computation time of the lower bound affects the performance of the skip tree; and (3) the branching factor of the application affects the performance of the skip tree. Based on the four application benchmarks and the synthetic benchmark, we provide a set of guidelines for selecting the lock-free skip tree to use as a centralized priority queue in a parallel branch-and-bound application versus the lock-free skip list. Finally, at the conclusion of this chapter we present results from analyzing the parallel branch-and-bound benchmarks on two shared-memory supercomputer architectures.

5.1 Candidate Application Benchmarks

In selecting the class of problems for this chapter, we were looking for a set of concurrent algorithms that relies heavily on a linearizable data structure that preserves the sorted set or sorted map abstraction. Our first choice was the Apache Hadoop implementation of Google's MapReduce framework. MapReduce is a programming model for processing and generating large datasets. Users specify the computation in terms of a map and a reduce function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks [105]. The MapReduce specification guarantees that within each of the reduce partitions, the intermediate key/value pairs are processed in sorted order. One possible implementation of this guarantee is to use a ConcurrentSkipListMap to collect key/value pairs as they are generated in the map phase. But we discovered that the Apache Hadoop implementation does not use a ConcurrentSkipListMap or ConcurrentSkipListSet.

Our second choice was the Apache Cassandra distributed database project [106]. Apache Cassandra uses a multidimensional key/value data model. A column represents a single key/value pair. Columns are grouped together into sets called column families. A set of column families can be grouped together into a super column family. Applications can specify the sort order of columns within a column family or a super column family. Column families and super column families are implemented using ConcurrentSkipListMaps. We replaced the ConcurrentSkipListMap instances in Cassandra 0.6.0 with ConcurrentSkipTreeMap instances. The performance of the two data structures was compared using the stress.py tool in the Cassandra source distribution. stress.py is intended for benchmarking and load testing a Cassandra cluster. At first we found no difference in performance using the skip list or the skip tree. Some investigation revealed

that column families are kept at relatively small sizes in memory before they are stored persistently on disk. After modifying the max column family size parameter setting, again no performance difference was exhibited between the skip list and the skip tree. A runtime profile of the Cassandra server during the execution of stress.py shows that the most time-consuming work phases are disk I/O on the server and network I/O between the client and server.

Our experiences with Apache Hadoop and Apache Cassandra led us to revise the prerequisites for the application benchmarks to focus on a class of problems that are solved with a single algorithmic technique that does not rely on tuning configuration parameters to maximize I/O bandwidth and minimize I/O latency. We settled on NP-hard problems that can be solved using a parallel branch and bound algorithm. Parallel branch and bound algorithms can be implemented using a concurrent priority queue. In the literature, a significant portion of papers that improve concurrent priority queue designs are published in the context of parallel branch-and-bound applications [107–112]. We selected four NP-hard problems to solve using a parallel branch-and-bound technique: N puzzle, graph coloring, asymmetric traveling salesman, and 0-1 knapsack.

For each of the NP-hard problems, we selected an algorithm for estimating the lower bound of each candidate solution that had been published as the best solution at some point in time, and is used in the current literature as a reference strategy against which novel approaches are measured. Many improvements to the lower bound estimate are made by improving the set of heuristics which can be applied to the candidate solution [29]. A sophisticated heuristic can reduce search space for a branch-and-bound application. For our purposes, the advanced heuristics modify the space of initial configurations with working set sizes that can fit in the memory available on the machine. In acknowledging the shift in configurations that have large working set sizes, the critical observation is that

a non-empty set of configurations exists that can benefit from a cache-conscious priority queue, regardless of the technique used to estimate the lower bounds.

One future application of lock-free skip trees is that in some parallel applications, a set of subtasks may be able to collect local ordered sets/maps, and then merge them in during a reduction step. It may be worthwhile to bulk-merge subtrees using skip tree versus just have all operations directly add to single result set. The indices of non-leaf nodes in the skip tree may be used as partitions to divide the merge operation into subsets. For example, imagine merging two subtrees of the same height. One subtree has a root node with elements $\{10, 30, 50\}$ and the other subtree has a root node with elements $\{20, 40, 60\}$. Each interval of the left (or right) subtree overlaps with two intervals from the right (or left) subtree, with the exception of intervals containing $\pm\infty$.

5.2 Parallel Branch-and-Bound Algorithms

Branch and bound is an algorithmic technique for finding optimal solutions of optimization problems. Given a finite set of candidate solutions X and an objective function $f(X) \rightarrow \mathbb{R}$, an optimal solution $x^\star \in X$ exists such that $f(x^\star) = \min\{f(x) | x \in X\}$. The goal of the branch and bound technique is to systematically traverse the space of candidate solutions in order to find an optimal solution. It is assumed that a splitting procedure exists that accepts an input set of candidates S and returns two or more smaller sets $S_1, S_2, \ldots, S_n$ whose union covers S. Each child of the input set is typically distinguished by the imposition of one or more constraints upon S to select the subset of candidates S_i. A second assumption is that there exists a method for estimating the lower and upper bounds of the objective function values in a set of candidate solutions. The branch and bound method was first proposed by Land and Doig [113] for solving integer linear programming

problems at British Petroleum [114].

The bounding procedure takes advantage of the fact that if the lower bound for some set A of candidate solutions is greater than the upper bound of another set B, then A may be discarded from the search space. In practice, the bounding procedure is implemented with global state that keeps track of the current best solution that has been identified so far. The branch and bound algorithms studied in this section perform a best-first search. Candidate solutions are searched in increasing order of their lower bound. Best-first selection is optimal with respect to the number of decomposed subproblems, with no ties occurring among lower bounds, and the branching and bounding operations do not depend on previous history [115]. We have selected the best-first search strategy to analyze the performance of the lock-free skip tree versus the lock-free skip list when used as a concurrent priority queue. The disadvantage of a best-first search is the memory requirement for storing all unexplored candidates in the search tree until an optimal solution is discovered. If an optimal solution can be discovered before running out of memory, then it is found relatively quickly as compared to a depth-first search.

Gendron and Crainic [107] identify three main approaches to designing parallel branch and bound algorithms. The first approach uses a sequential search strategy, but improves the performance of calculating the lower bound estimate with a concurrent implementation of this calculation. The second approach builds a single centralized search tree with multiple worker threads concurrently extracting candidate solutions from the tree and inserting children solutions into the tree. The third approach builds several search trees in parallel with some method of coordination among the search trees to balance the workload across all trees. The second approach has been the most popular approach to designing a parallel branch and bound application [108–112].

The java.util.concurrent ConcurrentSkipListSet is a lock-free implementation

of a quiescent priority queue backed by a skip list data structure [53]. With the addition of an operation to extract the minimum element, the lock-free skip tree implementation of the previous chapter implements a quiescent priority queue. Recall that quiescent consistency is defined such that the operations of any processors separated by a period of quiescence should appear to take effect in their real-time order. A linearizable priority queue would prohibit the following situation from occurring: Thread 1 inserts element x into the priority queue and then inserts element y into the priority queue, such that $y < x$. Concurrent with both insert operations, Thread 2 is performing an extract minimum operation. Thread 2 returns the value x. In order to enforce linearizability, one strategy is to redefine the extract minimum operation to select from elements that have completed insertion before the beginning of the extract min operation. This strategy can be implemented with timestamps per each node that note the logical time of insertion into the queue [116]. The quiescent priority queue is sufficient for branch-and-bound applications, where the only necessary requirement is that when $y < x$, then y is extracted from the queue before x, given a period of quiescence has expired. An alternative strategy to implementing a concurrent priority queue is to begin with a lock-free skip tree and restrict `remove` operations to the head of the data structure. It is possible that node compaction could be simplified under this strategy, as the preponderance of node compaction operations would be happening at the head of the queue.

5.3 N puzzle

The N puzzle is a game played on an $m \times m$ grid containing n square tiles, where $n = m^2 - 1$. Each tile is assigned a unique number from $1 \ldots n$. One tile in the grid is always empty, called the 'blank' tile, and depending on its position, the

blank has two, three, or four adjacent tiles. A move in the game swaps the positions of the blank tile and a single neighboring tile. The tiles are initially set to some random configuration. The goal is to rearrange the tiles into a specific goal configuration. Typically the goal configuration consists of the tiles in row-wise consecutive order, with the blank tile designated to appear either first or last in the sequence. Korf [117] provides more information concerning the history of the N puzzle problem. An excellent survey of NP-complete single-player games has been written by Kendall et al. [118].

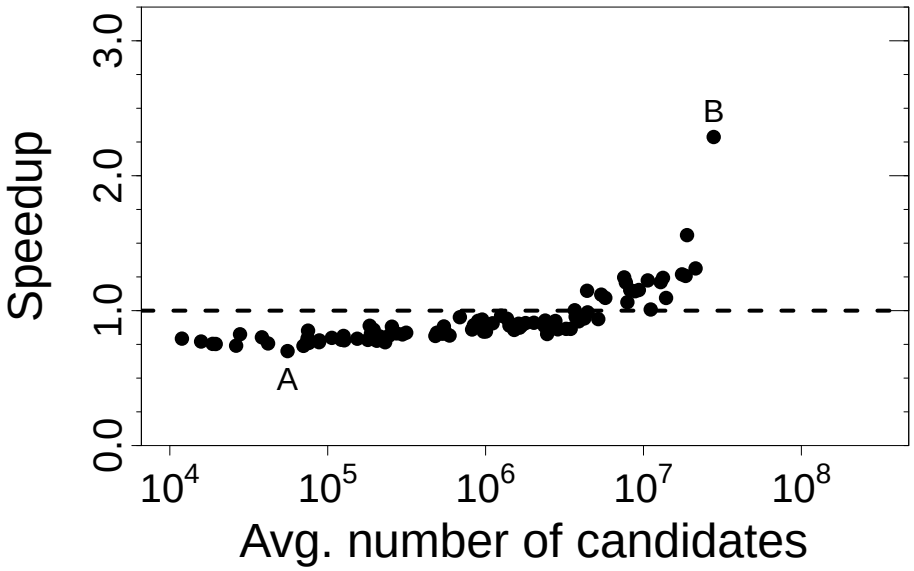
Half of the starting configurations of the N puzzle cannot reach the goal state [119]. A parity argument can be made using the parity of permutations plus the parity of the Manhattan distance moved by the blank tile. The Eight Puzzle contains $\frac{9!}{2}$ or $181,440$ possible configurations. The solutions for all legal starting configurations of the Eight Puzzle have been solved using brute force [120]. The average length of a shortest path between two states is about 22 moves, and the maximum distance between any pair of states is 31 moves. The Fifteen Puzzle contains $\frac{16!}{2}$ or over 10^{13} possible states. The Fifteen Puzzle is used as the NP-complete example in the publication that introduced the Iterative-Deepening A* (IDA*) algorithm [121]. One hundred randomly generated Fifteen Puzzle instances are used in the IDA* paper and their solutions are included in the publication. The average optimal solution length of these one hundred instances is about fifty-three moves. The Iterative-Deepening A* algorithm performs a depth-first search. We show that a parallel branch-and-bound solver can solve all one hundred instances presented in the IDA* paper using a breadth-first search.

The lower bound calculation used as a baseline for the parallel branch-and-bound solver is the sum of the Manhattan distances of each tile's current position relative to the tile's goal position. Two admissible heuristics are used to improve the lower bound: the linear-conflict heuristic and the last moves heuristic. Each

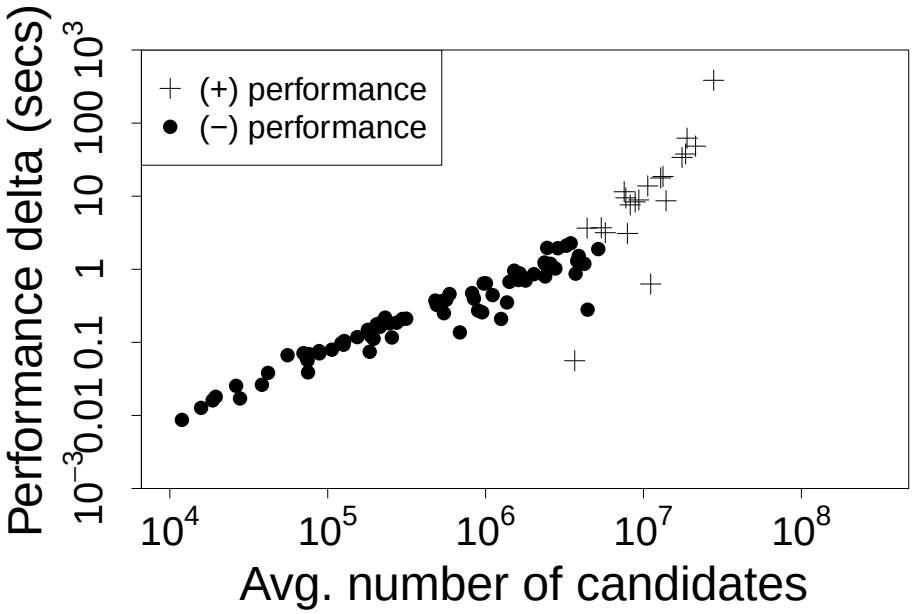
heuristic can improve upon the lower bound when a specific set of conditions mandates additional state transitions beyond those accounted for in the Manhattan distance lower bound. Additional care must be taken to ensure that multiple admissible heuristics do not interact unfavorably to produce a lower bound estimate that is not admissible (ie. never overestimates the lower bound). The best upper bound on the N puzzle problem is independent of the initial board state. The upper bound serves no useful purpose to limit the possible search space. Assuming that N is a fixed value for the N puzzle problem, calculation of the lower bound is $O(1)$.

The linear-conflict heuristic can increase the lower bound estimation when two tiles are in their goal row or column, but are reversed relative to their goal positions [122]. For example, if the top row of the puzzle contains the tiles $(2, 1)$ in that order, then one of the tiles must move down into the next row, in order to allow the other tile to move into the correct position. The last moves heuristic exploits the necessity of the blank tile to be the last tile that reaches the goal position, which is the upper-left corner in this case. The last move must either move the 1 tile to the right, or the 4 tile down. If the 1 tile is not in the left-most column, and the 4 tile is not in the top row, then two moves can be added to the Manhattan distance and still preserve admissibility [123]. If either the 1 tile or the 4 tile is in a linear-conflict state, then the last moves heuristic cannot be applied.

Figure 5.1a shows the relative speedup of the parallel branch-and-bound solver applied to one hundred instances of the Fifteen Puzzle executed on a Sun Fire T1000. The basis for comparison in these graphs is the runtime performance of the solver when the java.util.concurrent lock-free skip list is used as a priority queue. The vertical axis represents the relative speedup of the solver when the lock-free skip tree algorithm is used. The horizontal axis represents the average number of partial solutions generated in the search space until the goal state was reached.

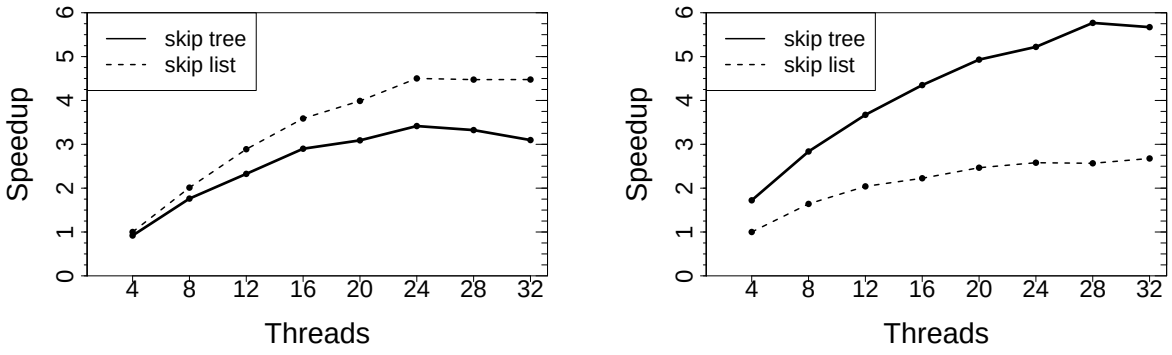


(a) relative speedup of lock-free skip tree. Vertical axis is $\frac{\Lambda}{T}$.



(b) absolute speedup of lock-free skip tree. Vertical axis is $|\Lambda - T|$.

Figure 5.1: N Puzzle instances on Sun Fire T1000. Λ is skip list execution time and T is skip tree execution time.



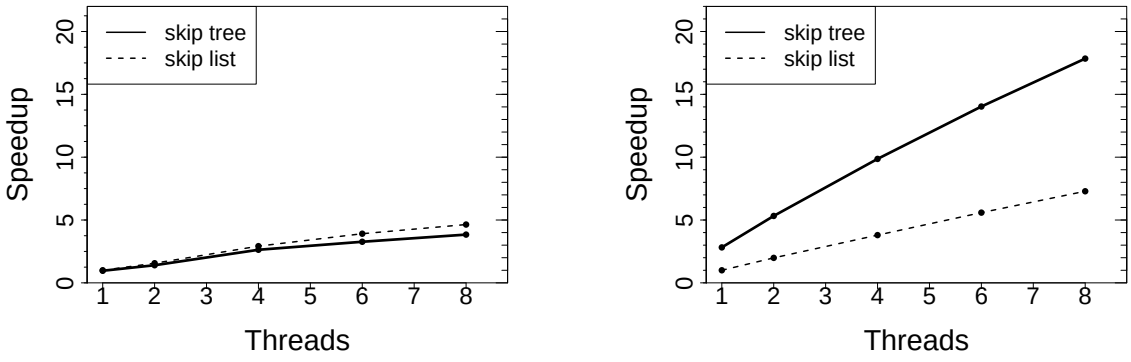
(a) problem 'A'

(b) problem 'B'

Figure 5.2: Worst-case and best-case N Puzzle instances on Sun Fire T1000. Speedup is the ratio of skip list to skip tree execution times.

Worker threads in the parallel branch-and-bound algorithm will not process partial solutions in the same order across repeated trials, and so therefore the average of the number of partial solutions is reported.

Instances of the Fifteen Puzzle with a relatively small working set appear on the left side of the graphs, and instances with a relatively large working set appear on the right side of the graphs. Figure 5.1a shows the relative speedup of the skip tree versus the skip list. For initial board configurations with a relatively small working set size, the skip list outperforms the skip tree. For initial board configurations with the largest working set sizes that could be contained in memory on this machine, the skip tree outperforms the skip list by as much as x2.3. Figure 5.1b shows the absolute difference in the runtime of the skip tree and the skip list. The vertical axis denotes the log of the execution time with the skip list subtracted from the execution time with the skip tree. Skip tree performance improvements are plotted with the '$+$' symbol and skip tree performance declines are plotted with the '$\bullet$' symbol. The absolute performance graphs illustrate the trend that an initial board configuration with a small working set size will execute in less time than an initial board configuration with a large working set size. As a corollary,



(a) problem 'E'

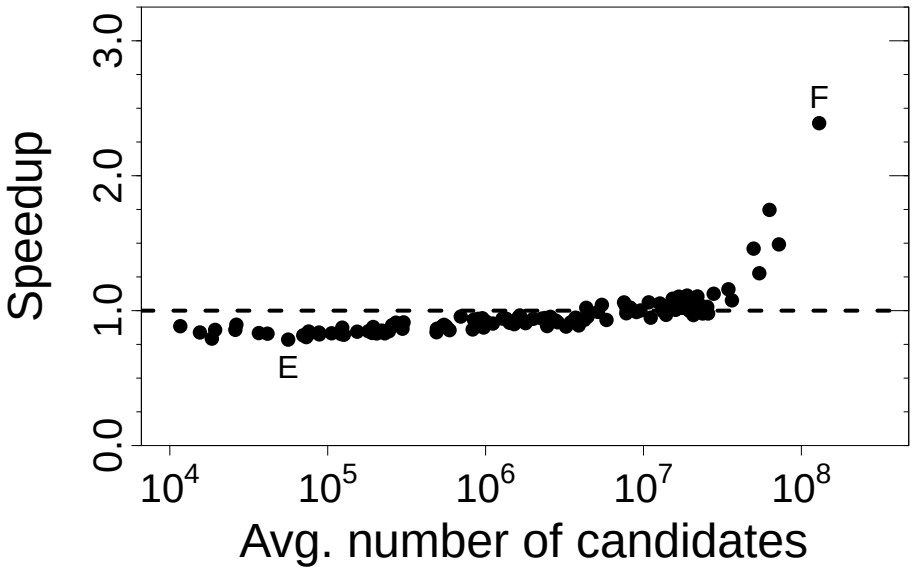
(b) problem 'F'

Figure 5.3: Worst-case and best-case N Puzzle instances on quad core Intel Xeon. Speedup is the ratio of skip list to skip tree execution times.

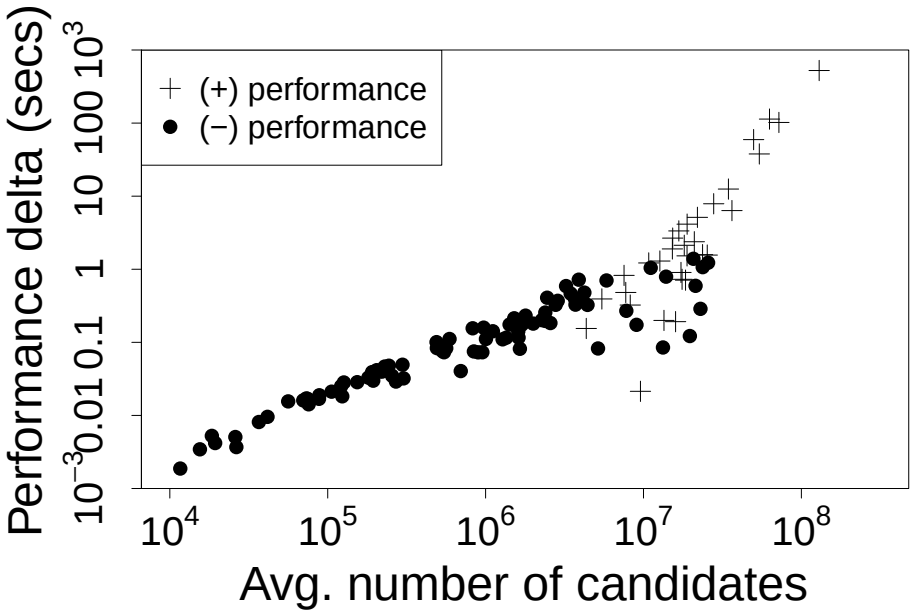
any relative performance deficiencies of the skip tree at small working set sizes correspond to a small performance penalty in absolute units.

The initial board configurations with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'A' and 'B' in Figure 5.1a. The speedup of configurations 'A' and 'B' relative to the number of available processors is shown in Figures 5.2a and 5.2b. Initial board 'A' is solved in 53 moves with an average of 5.8×10^4 partial solutions generated in the search space. Initial board 'B' is solved in 55 moves with an average of 2.8×10^7 partial solutions generated in the search space. There is no speedup of the skip tree relative to the skip list in board configuration 'A' when running on a single core (4 threads). As the thread count increases, the application speedup is smaller using the skip tree as compared to using the skip list. Initial board 'B' on a single core is solved using the skip tree with a speedup of x1.7 as compared to the skip list. Running on all eight cores, the speedup of the skip list relative to a single core is x2.7 and the speedup of the skip tree relative to the skip list on a single core is x5.7.

Performance results of the Fifteen Puzzle solver on a quad core Intel Xeon are shown in Figure 5.4. There are more initial board configurations in these results as



(a) relative speedup of lock-free skip tree. Vertical axis is $\frac{\Lambda}{T}$.



(b) absolute speedup of lock-free skip tree. Vertical axis is $|\Lambda - T|$.

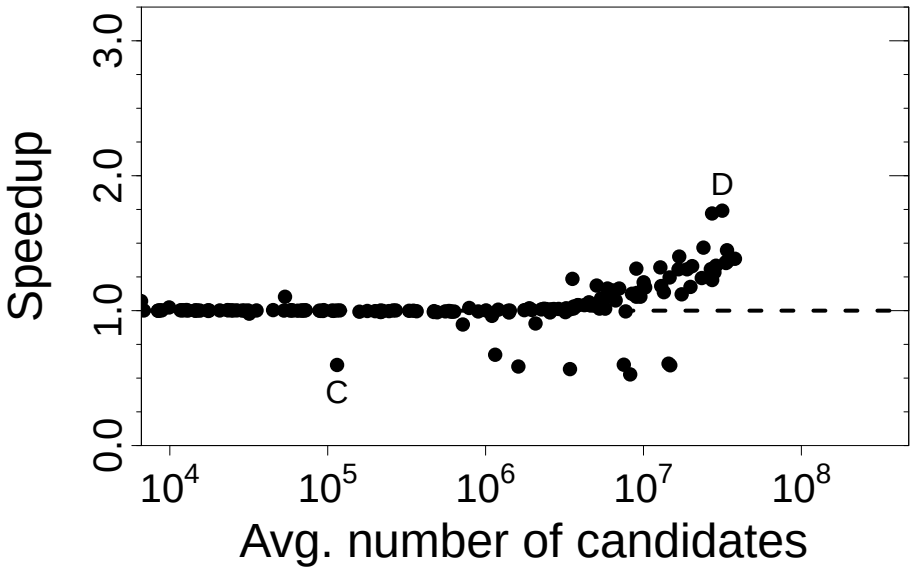
Figure 5.4: N Puzzle instances on quad core Intel Xeon. Λ is skip list execution time and T is skip tree execution time.

compared to the Sun Fire T1000, as this Intel Xeon had more memory available, implying that configurations with a larger working set size could be solved. The relative speedup of the skip tree to the skip list follows a similar trend on the Intel Xeon benchmarks as seen earlier on the Sun Fire benchmarks. The initial board configurations with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'E' and 'F'. There is no speedup of the skip tree relative to the skip list in board configuration 'E' when running on a single thread. Initial board 'F' on a single core is solved using the skip tree with a speedup of x1.7 as compared to the skip list. Running with eight threads, the speedup of the skip list relative to a single thread is x7.3 and the speedup of the skip tree relative to the skip list on a single thread is x17.8.

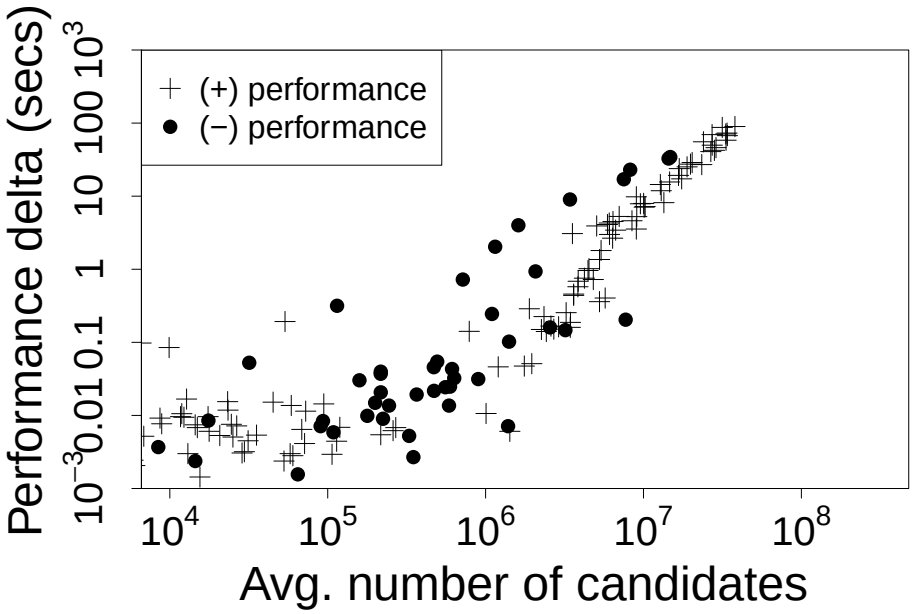
5.4 Graph coloring

A graph G is an ordered pair $G = (V, E)$, where V is a finite set of vertices, and E is a finite set of unordered pairs of vertices representing edges. A legal vertex-coloring of graph $G = (V, E)$ is a function $c : V \to \mathbb{N}$, in which any two incident vertices $u, v \in V$ are assigned different colors, meaning $\{u, v\} \in E \implies c(u) \neq c(v)$. The function c is called the coloring function. A graph G for which there exists a vertex-coloring which requires k colors is called k-colorable. The smallest number k for which there exists a k-coloring of graph G is called the chromatic number of graph G and is denoted by $\chi(G)$ [124]. The initial board configurations with the largest improvement of the lock-free skip tree as compared to the lock-free skip list exhibits a x1.5 runtime improvement on the Sun Fire and a x3.1 improvement on the Intel Xeon.

The saturation largest-first (SLF) algorithm, also known as DSATUR algorithm, solves the graph coloring problem by iteratively selecting the uncolored vertex v

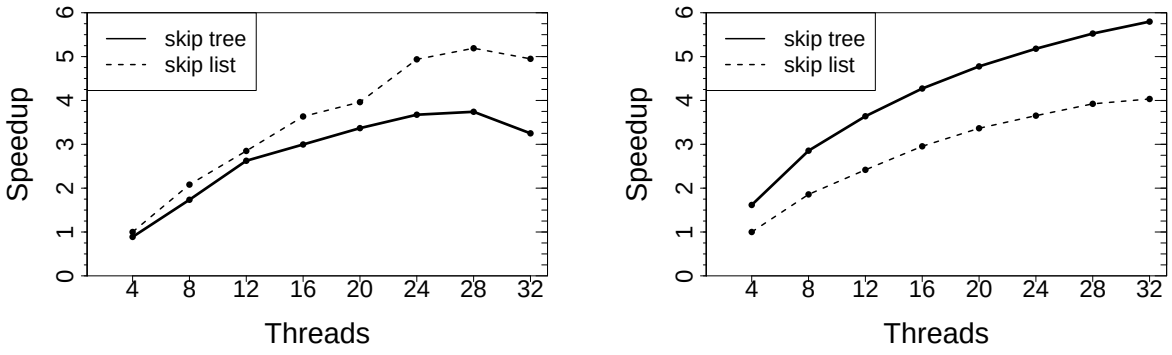


(a) relative speedup of lock-free skip tree. Vertical axis is $\frac{\Lambda}{T}$.



(b) absolute speedup of lock-free skip tree. Vertical axis is $|\Lambda - T|$.

Figure 5.5: Graph coloring instances on Sun Fire T1000. Λ is skip list execution time and T is skip tree execution time.

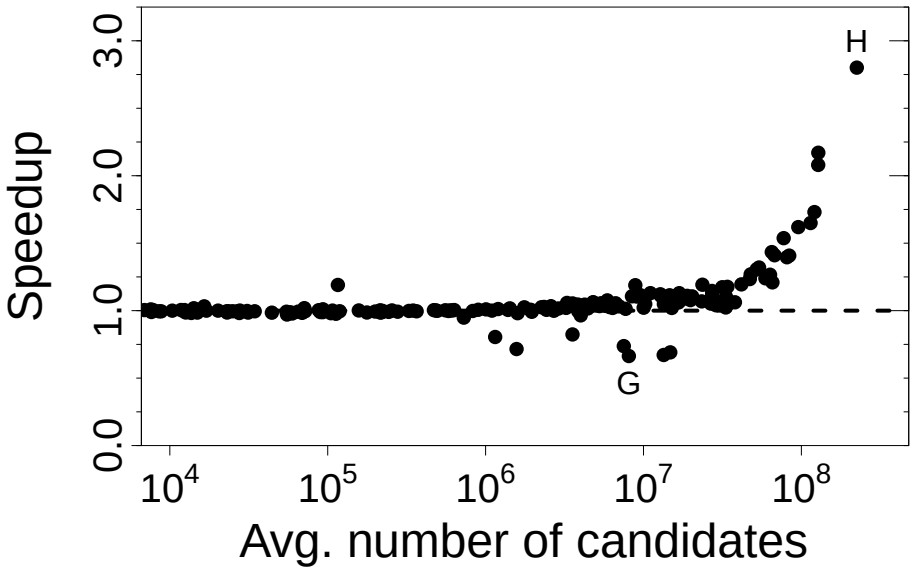


(a) problem 'C' (b) problem 'D'

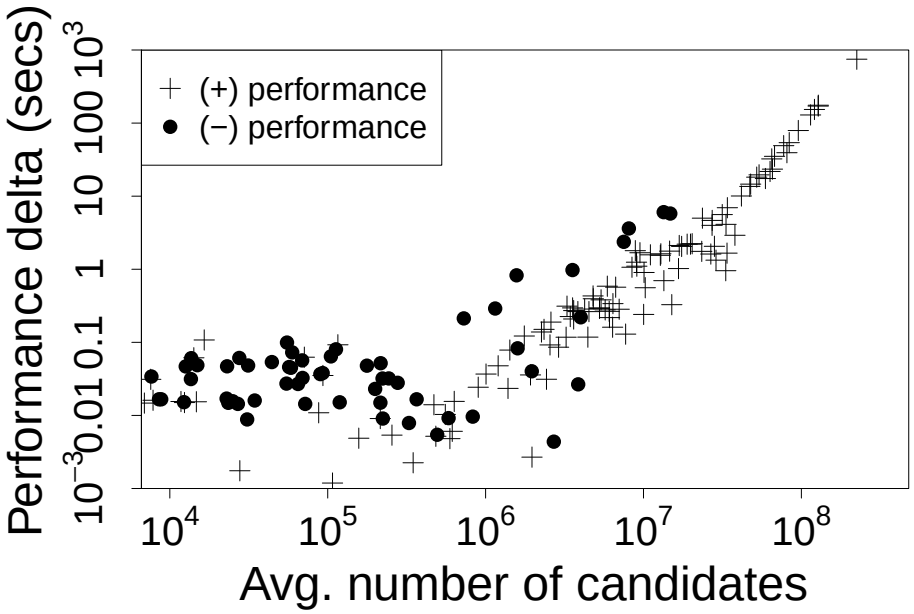
Figure 5.6: Worst-case and best-case graph coloring instances on Sun Fire T1000. Speedup is the ratio of skip list to skip tree execution times.

with the highest saturation degree in G and then branching on the legal colorings of vertex v [125]. The saturation degree of a vertex v in a partially colored graph G is defined as the number of distinctly colored vertices adjacent to v. The DSATUR algorithm has been considered a "*de facto* standard among exact graph coloring algorithms" [126]. An initial upper bound is estimated using a greedy iterative variation of the SLF algorithm, whereby the smallest possible color is assigned to uncolored vertex v with the highest saturation degree at each iteration. In our implementation, each vertex v builds a temporary bit vector that marks the distinct colors adjacent to v. The cost of building the bit vector entries requires traversing over all the edges of the uncolored vertices, which is $O(E)$. Selecting the uncolored vertex v with the highest saturation degree in G entails finding the max value of the bit vector entries which is $O(V)$. Combining these steps yields a worst-case running time of computing of the lower bound to be $O(E) + O(V)$.

The runtime performance of the graph coloring solver is tested with a set of randomly generated graphs. $G_{n,p}$ is a random graph with vertices $\{1, \ldots, n\}$ such that the probability of an edge between any two vertices is $0 < p < 1$, and the edge probability is independent of all other edges. These random graphs are commonly

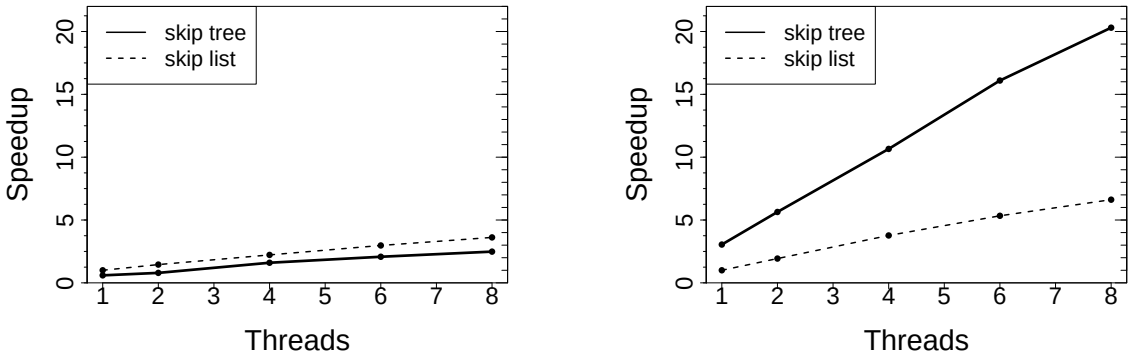


(a) relative speedup of lock-free skip tree. Vertical axis is $\frac{\Lambda}{T}$.



(b) absolute speedup of lock-free skip tree. Vertical axis is $|\Lambda - T|$.

Figure 5.7: Graph coloring instances on quad core Intel Xeon. Λ is skip list execution time and T is skip tree execution time.



(a) problem 'G'

(b) problem 'H'

Figure 5.8: Worst-case and best-case graph coloring instances on quad core Intel Xeon. Speedup is the ratio of skip list to skip tree execution times.

used as benchmarks in graph coloring publications [127–129]. In our experiments, n was selected from the set $\{10 \cdot i : i \in \mathbb{Z},\ 5 \leq i \leq 15\}$ and p was selected from the set $\{0.1, 0.3, 0.5, 0.7, 0.9\}$. Ten random graphs were generated for each possible combination of values for n and p, yielding 550 test cases. Performance results of the graph coloring solver on a Sun Fire T1000 are shown in Figure 5.5. Graph instances with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'C' and 'D' in Figure 5.5a. Graph instance 'C' was generated using $n = 70$ and $p = 0.3$. It has a chromatic number of 7 with average of 1.1×10^5 partial solutions generated in the search space. Graph instance 'D' was generated using $n = 90$ and $p = 0.3$. It has a chromatic number of 9 with an average of 3.2×10^7 partial solutions generated in the search space. There is no speedup of the skip tree relative to the skip list in graph instance 'C' when running on a single core (4 threads). As the thread count increases, the application speedup is smaller using the skip tree as compared to using the skip list. Graph instance 'D' on a single core is solved using the skip tree with a speedup of x1.6 as compared to the skip list. Running on all eight cores, the speedup of the skip list relative to a single core is x4.0 and the speedup of the skip tree relative to

the skip list on a single core is x5.8.

Performance results of the graph coloring solver on a quad core Intel Xeon are shown in Figure 5.7. There are more graph instances in these results as compared to the Sun Fire T1000, as this Intel Xeon had more memory available, allowing more configurations with a larger working set size to be solved. The relative speedup of the skip tree to the skip list follows a similar trend on the Intel Xeon benchmarks as seen earlier on the Sun Fire benchmarks. The initial board configurations with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'G' and 'H'. There is no speedup of the skip tree relative to the skip list in board configuration 'G' when running on a single thread. Initial board 'H' on a single core is solved using the skip tree with a speedup of x3.0 as compared to the skip list. Running with eight threads, the speedup of the skip list relative to a single thread is x6.6 and the speedup of the skip tree relative to the skip list on a single thread is x20.3.

5.5 Asymmetric Traveling Salesman Problem

The asymmetric Traveling Salesman Problem (ATSP) begins with a directed graph G of vertices V and edges E. Each edge is assigned a weight, w_{ij}, such that the weight from v_i to v_j is not necessarily equal to the weight from v_j to v_i. The objective of ATSP is to find a shortest Hamiltonian path, defined as a path that visits each vertex exactly once and has the smallest sum of all traversed edge weights [130, 131]. Let the cost of the shortest Hamiltonian path for graph G be designated as $ATSP(G)$. In all instances, there was no runtime improvement of the solver when using the skip tree versus the skip list.

When a new branch-and-bound algorithm for solving ATSP is published, the Held-Karp lower bound is the most common lower bound that is used as a basis

of comparison [132]. The Held-Karp lower bound on optimal tour length is constructed from a relaxation of the constraints imposed by ATSP [133, 26]. The ATSP imposes a constraint on the optimal tour such that each vertex must have an indegree of one, and the constraint that each vertex must have an outdegree of one. If the outdegree constraint is eliminated and some arbitrary node is designated as a root node, then an optimal minimum spanning tree (MST) can be generated. The optimum minimum spanning tree for graph G is designated as $MST(G)$. The directed MST problem can be solved by the Chu-Liu/Edmonds algorithm [134–136] which is a polynomial time algorithm. The directed MST is generated by a relaxation of the constraints for ATSP. Therefore the cost of the directed MST is less than or equal to the cost of the optimal Hamiltonian path:

$$MST(G) \leq ATSP(G) \tag{5.1}$$

If each vertex in the directed minimum spanning tree has an outdegree of one, then the shortest Hamiltonian path has been discovered. Otherwise, an iterative procedure is used to transform the weights of the graph in order to encourage the directed MST to become a Hamiltonian path. Let $\pi(v)$ be any function that maps a vertex v to some real number. Let the transformed weights be defined as $w_{ij}^{\pi} = w_{ij} - \pi(i)$. In the Hamiltonian path each vertex must have an outdegree of 1. This implies that the Hamiltonian path is affected by the weight transformation exactly once per vertex:

$$ATSP(G^{\pi}) = ATSP(G) - \sum(\pi(v) \ : \ v \in V) \tag{5.2}$$

Combining equations 5.1 and 5.2 yields the observation that the transformed directed MST is always a lower bound on the optimal Hamiltonian path:

$$max_{\pi}\left(MST(G^{\pi}) + \sum(\pi(v) \ : \ v \in V)\right) \leq ATSP(G) \tag{5.3}$$

In order to derive a good lower bound on the optimal Hamiltonian path, $\pi(v)$ is assigned some negative value when the outdegree of v is greater than one and a positive value when the outdegree is less than one. If $deg^{+}(v)$ is the outdegree of v, then $\pi(v) = k\,(1 - deg^{+}(v))$ where k is an arbitrary constant. In practice, the Held-Karp process is not run long enough to converge on the optimal Hamiltonian path. However, after only a few iterations of the Held-Karp process, a sufficiently accurate lower bound estimate can be calculated.

The TSPLIB library was used for input graphs to test the performance of the ATSP solver. TSPLIB is a collection of graphs along with their solutions for the Traveling Salesman Problem and related problems [137]. The collection of graphs for the asymmetric traveling salesman problem come from a variety of real-world scheduling and resource management applications [132]. The instances of the TSPLIB that could be tested by our branch-and-bound solver contained between 17 and 171 cities. In all instances, there was no runtime improvement of the solver when using the skip tree versus the skip list. In Section 5.7 we will show that the ATSP solver is a compute-bound problem, which is to say that over 99% of the total runtime is spent computing the lower bounds of the partial solutions.

5.6 0-1 Knapsack

The 0-1 knapsack problem [138], hereafter referred to as the knapsack problem, is defined using a set of n items, such that each item j is assigned a profit p_j and a weight w_j. The knapsack has a capacity c. A set of binary decision variables $\{x_1, \ldots, x_n\}$ are defined such that item j is placed into the knapsack if $x_j = 1$, and item j is not placed in the knapsack if $x_j = 0$. The objective of the knapsack prob-

lem is to maximize $\sum_{j=1}^{n} p_j x_j$ subject to the constraint $\sum_{j=1}^{n} w_j x_j \leq c$. Let the efficiency of item j be defined as the ratio of profit to weight. Assume the items are sorted by their efficiency in decreasing order such that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}$. A greedy solution to the knapsack problem is to select the largest consecutive sequence of most-efficient items that will fit in the knapsack. The first item s that cannot fit into the knapsack is referred to as the split item. The solution vector $\hat{x}$ with $\hat{x}_j = 1$ for $j = 1, \ldots, s-1$ and $\hat{x}_j = 0$ for $j = s, \ldots, n$ is known as the split solution.

A branch-and-bound algorithm commonly used as a basis for comparison on the knapsack problem is the primal-dual algorithm [28]. This algorithm is based on the observation that for many instances of the knapsack problem only a relatively small subset of items is crucial for the determination of the optimal solution. Items with very high efficiencies will almost certainly be included in the optimal solution. Items with very low efficiencies will almost certainly be excluded from the optimal solution. What remains is the set of items of "medium efficiency", referred to as the core set of items. The core set begins with the split item and expands outwards in both directions. That is, the core set of items expands to the "left" to include more efficient items, and expands to the "right" to include less efficient items. In the primal-dual algorithm, the binary decision variables $x_{a+1}, \ldots, x_{b-1}$ are fixed at some value and it is assumed that $x_j = 1$ for $j \leq a$ and $x_k = 0$ for $k \geq b$. The profit and weight sums of the current configuration are denoted $\bar{p}$ and $\bar{w}$, respectively. If $\bar{w} \leq c$ an item is inserted into the knapsack and otherwise an item is removed from the knapsack.

The primal-dual algorithm has a branching factor of 2. Given a configuration of binary decision variables, assume that $\bar{w} \leq c$, then one child configuration is generated by assigning $x_b = 0$ and another configuration is generated by assigning $x_b = 1$. In either case, the right boundary b is incremented, $b' = b + 1$. In a similar fashion, if $\bar{w} > c$ then one configuration is generated by assigning $x_a = 0$ and

another is generated by assigning $x_a = 1$. The left boundary of these children is decremented, $a' = a - 1$. Suboptimal solutions are eliminated using an upper bound estimate that is based on either the most efficient non-fixed item in the case of an underfilled knapsack: $\frac{(c - \bar{w})\, p_b}{w_b}$, or the least efficient item in the knapsack in the case of an overfilled knapsack: $\frac{(c - \bar{w})\, p_y}{w_y}$ such that y is the max where $x_y = 1$.

There are three types of random distributions that are commonly used to generate test instances of the knapsack problem [28]. Each distribution assumes a data range R for the profit and weight values and a problem size n for the total number of items. The uncorrelated distribution assumes that p_j and w_j are randomly distributed in $[1, R]$. The weakly correlated distribution assumes that w_j is randomly distributed in $[1, R]$ and p_j is randomly distributed in $\left[w_j - \frac{1}{10}R, w_j + \frac{1}{10}R\right]$. The term "weakly correlated" is somewhat misleading, as weakly correlated instances have a very high correlation between the profit and weight of an item. The strongly correlated distribution assumes that w_j is randomly distributed in $[1, R]$ and $p_j = w_j + 10$. In all instances, there was no runtime improvement of the solver when using the skip tree versus the skip list. In the next section we show that the knapsack solver is a communication-bound problem. The cost of extracting the minimum element and inserting new elements into the priority queue dominates over the cost of computing the next child configuration.

5.7 Performance Analysis

The purpose of this section is to explore why the lock-free skip tree shows an improvement of up to x2.4 and x3.1 on the 15 puzzle and graph coloring applications, but did not yield any improvements on the asymmetric traveling salesman problem and the 0-1 knapsack problem. Determining the reasons behind the performance differences will guide a construction of a series of properties that are

necessary for a branch-and-bound application to prefer a cache-conscious priority queue (Section 5.9). The first step is to identify the computational phases that are common to all four applications and then study the application differences in behavior across these common phases. The parallel branch-and-bound algorithm consists of three primary phases. These are: (1) extract the minimum element from the shared priority queue, (2) generate several children candidate solutions based on the current candidate solution, and (3) if the bounding criterion has not been reached, then insert each new child candidate solution into the shared priority queue. Extracting the minimum element is an $O(1)$ operation in the absence of thread contention. Inserting a new candidate solution into the shared priority queue is an $O(\log S)$ operation in the absence of thread contention, where S is the number of candidate solutions currently stored in the priority queue. In general, it is difficult to estimate a tight upper bound on S for a specific input instance to a parallel branch-and-bound solver.

To generate a child candidate solution, the full state of the parent solution must be regenerated based on the partial state information stored in the parent solution (explained in Section 5.2), and then the lower and upper bounds for the child candidate solution must be computed. For the 15 puzzle, generating the upper bound estimate is $O(1)$. For the graph coloring problem, computing the lower bound of the chromatic number is $O(E) + O(V)$, where E is the number of edges and V is the number of vertices in the graph. For the asymmetric traveling salesman problem, our directed minimum spanning tree solver uses the Bock adjacency matrix representation [136] rather than the Chu–Liu/Edmonds graph representation [134, 135] to avoid recreation of the graph at every partial solution. Calculating the upper bound for the ATSP using the Bock representation has a worst case running time of $O(EV)$. The 0-1 knapsack problem calculates the upper bound of the profit on a candidate solution in $O(1)$ time for underfilled knapsacks and $O(N)$ time for

overfilled knapsacks, where N is the number of items from which selection is performed.

Tables 5.1 and 5.2 show the runtime percentage of each phase of computation for each of the four applications on the Niagara and quad core Xeon processors. Each row shows a percentage that is relative to the total runtime of that problem instance using the skip list data structure. The proportions in each skip list row sum to one hundred, and the results in each skip tree row are some fraction relative to the appropriate skip list row. The fourth column in the table records the total number of candidate solutions explored in the search space. This column is used as a surrogate for the maximum value of S, the number of candidate solutions currently stored in the priority queue. The total number of candidates is used as a proxy measure for the size of the queue. Estimating the number of elements stored in a lock-free data structure at any point in time is an impractical exercise. Input instances 'A' and 'B' for the 15 puzzle problem are specified in Figure 5.1a, and input instances 'C' and 'D' for the graph coloring problem are specified in Figure 5.5a.

Based on the results shown in Tables 5.1 and 5.2, the computational phase that is responsible for the majority of the performance differences between the skip list and the skip tree is the insertion of candidate solutions into the shared priority queue. For input problems with a large total number of candidate solutions explored in the search space, the proportion of time spent on inserting into the queue is approximately halved. On the Niagara architecture, this proportion for instance 'B' of the 15 puzzle problem moves from 89% for the skip list to 38% for the skip tree and from 62% to 28% for instance 'D' of the graph coloring problem. For the instances of the 15 puzzle and graph coloring problems that showed a relative loss in performance using the skip tree versus the skip list, the insertion of candidate solutions is the culprit for this performance penalty. A verification that

Application (input)	Extract min	Insert queue	Computation	Number of candidates
	— skip list —			
	— skip tree —			
15 puzzle (A)	4.8 ± 0.34	$44. \pm 2.4$	$51. \pm 1.5$	$5.9 \cdot 10^4 \pm 1.7 \cdot 10^3$
	4.3 ± 0.27	$76. \pm 6.2$	$52. \pm 3.6$	$5.7 \cdot 10^4 \pm 4.3 \cdot 10^3$
15 puzzle (B)	1.1 ± 0.23	$89. \pm 4.8$	9.7 ± 0.36	$2.8 \cdot 10^7 \pm 2.2 \cdot 10^3$
	0.49 ± 0.075	$38. \pm 8.3$	7.2 ± 0.18	$2.8 \cdot 10^7 \pm 1.8 \cdot 10^3$
Graph color (C)	2.7 ± 0.65	8.5 ± 1.6	$89. \pm 11.$	$1.2 \cdot 10^5 \pm 2.2 \cdot 10^4$
	5.8 ± 3.3	$75. \pm 21.$	$83. \pm 9.4$	$1.1 \cdot 10^5 \pm 2.4 \cdot 10^4$
Graph color (D)	0.95 ± 0.10	$62. \pm 2.4$	$37. \pm 0.37$	$3.2 \cdot 10^7 \pm 6.4 \cdot 10^2$
	0.54 ± 0.15	$28. \pm 2.4$	$42. \pm 0.085$	$3.2 \cdot 10^7 \pm 5.6 \cdot 10^2$
Asymmetric TSP	$0.0087 \pm 8.4 \cdot 10^{-5}$	0.11 ± 0.002	$99. \pm 0.11$	$1.0 \cdot 10^6 \pm 0$
	$0.0081 \pm 2.9 \cdot 10^{-4}$	0.16 ± 0.004	$99. \pm 0.06$	$1.0 \cdot 10^6 \pm 0$
0-1 Knapsack	$17. \pm 2.4$	$65. \pm 9.5$	$17. \pm 3.7$	$2.3 \cdot 10^7 \pm 0$
	$26. \pm 3.7$	$49. \pm 4.6$	$20. \pm 1.9$	$2.3 \cdot 10^7 \pm 0$

Table 5.1: Relative performance on Sun Fire T1000 (as percentage)

Application (input)	Extract min	Insert queue	Computation	Number of candidates
	— skip list —			
	— skip tree —			
15 puzzle (E)	$12. \pm 0.95$	$45. \pm 3.7$	$42. \pm 3.3$	$5.6 \cdot 10^4 \pm 4.1 \cdot 10^3$
	$16. \pm 2.1$	$55. \pm 5.2$	$45. \pm 3.7$	$5.4 \cdot 10^4 \pm 4.5 \cdot 10^3$
15 puzzle (F)	1.9 ± 0.23	$88. \pm 3.3$	9.7 ± 0.39	$1.3 \cdot 10^8 \pm 1.5 \cdot 10^4$
	2.1 ± 0.11	$36. \pm 2.4$	9.3 ± 0.18	$1.3 \cdot 10^8 \pm 1.6 \cdot 10^4$
Graph color (G)	$15. \pm 3.5$	$20. \pm 4.4$	$65. \pm 12.$	$7.8 \cdot 10^6 \pm 1.6 \cdot 10^6$
	$15. \pm 2.6$	$55. \pm 12.$	$63. \pm 11.$	$8.0 \cdot 10^6 \pm 1.6 \cdot 10^6$
Graph color (H)	0.92 ± 0.25	$79. \pm 2.7$	$20. \pm 1.1$	$2.2 \cdot 10^8 \pm 8.2 \cdot 10^3$
	0.85 ± 0.065	$18. \pm 1.2$	$20. \pm 1.1$	$2.2 \cdot 10^8 \pm 1.3 \cdot 10^4$
Asymmetric TSP	$0.018 \pm 4.5 \cdot 10^{-4}$	0.13 ± 0.002	$99. \pm 0.10$	$1.0 \cdot 10^6 \pm 0$
	$0.014 \pm 1.3 \cdot 10^{-4}$	0.17 ± 0.003	$95. \pm 0.10$	$1.0 \cdot 10^6 \pm 0$
0-1 Knapsack	$24. \pm 5.0$	$55. \pm 14.$	$21. \pm 9.2$	$2.3 \cdot 10^7 \pm 0$
	$19. \pm 3.3$	$43. \pm 6.2$	$19. \pm 3.9$	$2.3 \cdot 10^7 \pm 0$

Table 5.2: Relative performance on quad core Intel Xeon (as percentage)

the insertion of candidate solutions is the bottleneck for small working set sizes is shown below. The distribution of retries for insertion and extraction operations on the queue is measured. For the small problem instances, insertion operations on the skip tree require more attempts in order to succeed as compared to extraction operations.

Why is there no performance improvement when using the skip tree over the skip list on the asymmetric traveling salesman problem and the 0-1 knapsack problem? The evidence suggests that the working set size of these algorithms exceeds the cache size for the specific problem instances that are evaluated in Table 5.1. The total numbers of candidate solutions explored in the search space for the ATSP and 0-1 knapsack problems are $1.0 \cdot 10^6$ and $2.3 \cdot 10^7$, respectively. The total numbers of candidate solutions explored in the search space for the large instances of the 15 puzzle and graph coloring problems are $2.8 \cdot 10^7$ and $3.2 \cdot 10^7$. The equal runtime of the asymmetric traveling salesman problem solver is explained by the computational phase that calculates the lower bound estimate for each partial solution. Over 99% of the total runtime is spent in this phase for the ATSP solver. This leads to the first lesson in our characterization of parallel branch-and-bound applications in order to show benefit from a cache-conscious data structure: compute bound applications will not show speedup as a consequence of Amdahl's Law.

Table 5.3 shows the elapsed time per operation on the Sun Fire T1000. In general, the extract minimum operation is the fastest operation. The elapsed time of element insertion is proportional to the total number of candidate solutions explored in the search space. Calculating the upper bound for instances of the traveling salesman problem is two orders of magnitude longer than the bound calculations for the other three applications. The 0-1 knapsack problem exhibits a relatively long elapsed time for the extract minimum operation as compared to the other three applications. For both the skip list and the skip tree, the average time

Application (input)	Extract min	Insert queue	Computation
	— skip list —		
	— skip tree —		
15 puzzle (A)	4.44 ± 0.37	18.8 ± 0.96	22.4 ± 0.53
	3.29 ± 0.09	37.3 ± 0.91	23.4 ± 0.57
15 puzzle (B)	8.52 ± 1.16	$483. \pm 48.2$	42.3 ± 2.99
	6.44 ± 2.91	$308. \pm 68.9$	35.5 ± 8.24
Graph color (C)	7.65 ± 16.9	10.3 ± 1.08	51.7 ± 7.78
	3.28 ± 0.83	79.4 ± 5.84	54.8 ± 9.32
Graph color (D)	8.08 ± 1.03	$124. \pm 9.61$	51.5 ± 5.21
	4.10 ± 0.28	51.4 ± 0.91	52.1 ± 0.39
Asymmetric TSP	0.92 ± 0.03	8.56 ± 0.39	$9.76 \times 10^4 \pm 102.$
	0.83 ± 0.02	11.7 ± 0.22	$9.76 \times 10^4 \pm 70.2$
0-1 Knapsack	21.6 ± 8.37	82.8 ± 32.4	4.16 ± 2.27
	31.6 ± 5.63	61.5 ± 18.5	3.97 ± 1.40

Table 5.3: Elapsed time per operation on Sun Fire T1000 (in microseconds)

to completion is 21.6 and 31.6 microseconds. The geometric means for the skip list and the skip tree of the other five problem instances are 4.64 and 2.98 microseconds. Why should the extract minimum operation take longer on the 0-1 knapsack problem as compared to the other five problem instances? This is especially puzzling as we have established that the extract minimum operation has $O(1)$ cost in the absence of thread contention. We have established that these delays are due to thread contention. To demonstrate this hypothesis, we conducted a series of tests on the number of retries for the concurrent priority queue in all problem instances, except for the asymmetric traveling salesman problem as we have explained the behavior of that system. The next series of experiments measured the distributions of retries for the extract minimum and insert element operations on the priority queue.

In both the skip list and the skip tree, the failure of a compare-and-swap on an

	0	1-3	4-6	7-9	10+
15 puzzle (A)	62.5	31.7	4.90	0.75	0.15
15 puzzle (B)	83.2	15.9	0.85	0.06	0.01
Graph color (C)	64.5	30.3	4.53	0.58	0.01
Graph color (D)	91.9	7.92	0.17	0.00	0.00
0-1 Knapsack	29.6	40.0	14.7	6.27	9.40

(a) skip list extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	83.6	15.3	1.00	0.00	0.00
15 puzzle (B)	91.4	8.6	0.00	0.06	0.00
Graph color (C)	65.0	32.5	2.28	0.14	0.01
Graph color (D)	98.4	1.58	0.00	0.00	0.00
0-1 Knapsack	99.9	0.01	0.00	0.00	0.00

(b) skip list insertion (CAS)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	73.5	19.0	5.09	1.59	0.07
15 puzzle (B)	87.0	12.5	0.46	0.05	0.01
Graph color (C)	42.3	41.6	11.0	3.44	1.67
Graph color (D)	97.1	2.92	0.01	0.00	0.00
0-1 Knapsack	100.0	0.01	0.00	0.00	0.00

(c) skip list insertion (tree traversal)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	40.0	61.1	2.70	0.19	0.07
15 puzzle (B)	57.2	41.5	1.21	0.07	0.02
Graph color (C)	45.6	52.1	1.90	0.26	0.12
Graph color (D)	77.9	21.9	0.16	0.01	0.01
0-1 Knapsack	10.3	25.5	20.1	13.8	30.2

(d) skip tree extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	59.1	11.3	7.31	4.96	17.3
15 puzzle (B)	72.6	18.5	5.11	1.84	1.95
Graph color (C)	2.41	19.3	15.4	11.2	51.7
Graph color (D)	89.7	9.36	0.89	0.11	0.04
0-1 Knapsack	99.9	0.09	0.01	0.00	0.01

(e) skip tree insertion

Table 5.4: Number of retries for queue operations on Sun Fire T1000 (as percentage)

Application (input)	Extract min	Insert queue	Computation
	— skip list —		
	— skip tree —		
15 puzzle (E)	1.16 ± 0.04	2.11 ± 0.05	2.10 ± 0.09
	1.53 ± 0.04	2.84 ± 0.07	2.14 ± 0.08
15 puzzle (F)	1.05 ± 0.03	27.3 ± 1.08	3.07 ± 0.15
	1.24 ± 0.08	11.6 ± 0.74	2.88 ± 0.05
Graph color (G)	1.47 ± 0.04	1.82 ± 0.05	2.91 ± 0.15
	1.38 ± 0.10	5.24 ± 0.18	2.72 ± 0.12
Graph color (H)	1.32 ± 0.10	31.0 ± 1.45	5.26 ± 0.19
	1.45 ± 0.35	7.70 ± 0.69	5.82 ± 0.44
Asymmetric TSP	1.31 ± 0.04	1.35 ± 0.02	$3.59 \times 10^3 \pm 9.19$
	1.06 ± 0.06	1.67 ± 0.02	$3.57 \times 10^3 \pm 8.29$
0-1 Knapsack	2.23 ± 0.65	5.11 ± 1.34	0.33 ± 0.12
	1.87 ± 0.41	4.08 ± 0.71	0.28 ± 0.07

Table 5.5: Elapsed time per operation on quad core Intel Xeon (in microseconds)

extract minimum operation mandates restarting the extract min operation from the head of the data structure. To measure the number of retries on an insert element operation, we measured the attempts only for the leaf levels of the skip list and the skip tree. When a compare-and-swap is unsuccessful on element insertion in the skip tree, a localized search can proceed to find the appropriate node for insertion. The skip list performs several consistency checks before an element is inserted at the leaf level of the list. If one of the consistency checks is unsuccessful, then the data structure is re-traversed from the root node to the leaves. In addition, if a compare-and-swap operation is unsuccessful then another compare-and-swap attempt will be made. We have separated the measurements for the number of re-traversals and the number of compare-and-swap attempts on a skip list insert operation.

In the measurements for the distribution of operation retries, the insert ele-

ment operation is not a performance bottleneck in the 0-1 knapsack application (see Table 5.4). Almost 100% of the element insertion operations succeed on the first attempt using either the skip list or the skip tree. The knapsack problem suffers contention from the extraction of elements from the head of the queue. With the skip tree, 14.8% of extract min operations on the knapsack problem require five or more attempts. With the skip list, 13.6% of the extract min operations on the knapsack problem require five or more attempts. The contention at the head of the priority queue raises the total proportion of execution time that is spent on the extract minimum operations. This behavior has been confirmed in Tables 5.1 and 5.2. When a greater proportion of time is spent on the extract min operations, as a consequence a smaller proportion of time must be spent on insert element operations. The performance improvements of the skip tree on the insert element operations are observed, but their effects are diluted by the greater time on extract minimum operations.

In the next section, our goal is to generalize the observations made here to an arbitrary optimization problem that is solved using the parallel branch-and-bound solver, based on the information collected from the four applications. We have created a synthetic branch and bound application to study the effects of application characteristics on the relative performance of the skip tree priority queue. The synthetic application is a simplified model of a real application. As a model it captures the characteristics of the actual application problems that are of interest, and creates a set of assumptions that simplifies application characteristics that are not of interest. It is our hypothesis that too much elapsed time on the lower bound estimate leads to no improvement in performance, as observed in the asymmetric traveling salesman problem. And too little elapsed time on the lower bound estimate also leads to no improvement in performance, as observed in the 0-1 knapsack problem. Of equal interest are the effects of the branching factor and the

	0	1-3	4-6	7-9	10+
15 puzzle (A)	67.4	28.6	3.56	0.39	0.04
15 puzzle (B)	93.8	6.08	0.15	0.00	0.00
Graph color (C)	62.2	32.7	4.54	0.51	0.05
Graph color (D)	97.4	2.61	0.01	0.00	0.00
0-1 Knapsack	45.3	41.2	9.78	2.63	1.17

(a) skip list extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	91.8	8.13	0.09	0.00	0.00
15 puzzle (B)	98.9	1.13	0.00	0.00	0.00
Graph color (C)	75.5	24.1	0.38	0.01	0.00
Graph color (D)	99.9	0.23	0.00	0.00	0.00
0-1 Knapsack	100.0	0.00	0.00	0.00	0.00

(b) skip list insertion (CAS)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	81.3	15.2	2.75	0.59	0.15
15 puzzle (B)	96.8	3.15	0.07	0.00	0.00
Graph color (C)	48.5	37.9	9.67	2.77	1.11
Graph color (D)	99.3	0.73	0.00	0.00	0.00
0-1 Knapsack	100.0	0.00	0.00	0.00	0.00

(c) skip list insertion (tree traversal)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	48.4	43.4	6.01	1.35	0.83
15 puzzle (B)	79.9	19.4	0.59	0.04	0.00
Graph color (C)	56.4	39.0	3.72	0.65	0.22
Graph color (D)	87.6	12.3	0.03	0.00	0.00
0-1 Knapsack	37.2	47.9	10.6	2.8	1.43

(d) skip tree extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	69.4	21.6	6.11	1.88	0.98
15 puzzle (B)	87.7	11.6	0.67	0.05	0.00
Graph color (C)	30.7	47.8	14.8	4.52	2.08
Graph color (D)	96.5	3.52	0.02	0.00	0.00
0-1 Knapsack	100.0	0.03	0.00	0.00	0.00

(e) skip tree insertion

Table 5.6: Number of retries for queue operations on quad core Intel Xeon (as percentage)

distribution of lower bound estimates on the performance of the skip tree priority queue.

5.8 Synthetic Application

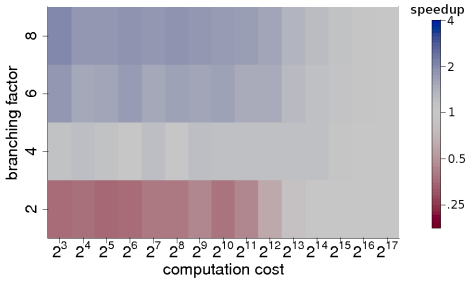
We created a synthetic branch-and-bound application in order to study the characteristics of an application that will yield a speedup when the skip tree is used as the centralized data structure. This synthetic application is a simplification of a real branch-and-bound application. It has been designed to test three hypotheses of the branch-and-bound applications: (1) the distribution of lower bounds of the candidates in the search space affects the performance of the skip tree; (2) the computation time of the lower bound affects the performance of the skip tree; and (3) the branching factor of the application affects the performance of the skip tree. In the synthetic application we assume the objective function is a minimization problem. We also assume that there is no upper bound. All generated children instances are valid instances for the priority queue. In a real application, a tight upper bound serves to limit the effective branching factor of the application, and would also add additional computational cost of the upper bound that is not taken into consideration by the synthetic application.

The synthetic application has a unique termination condition in that it does not return a solution that has any semantic meaning. Instead it generates problem instances until a predetermined limit is reached. When this limit has been reached then the algorithm terminates. In our test cases this limit has been fixed at $2.5 \cdot 10^7$ instances on the Niagara server and $1.0 \cdot 10^8$ instances on the Intel Xeon server, based on the number of generated candidates from the application benchmarks as shown in Tables 5.1 and 5.2. The Xeon processor has larger cache sizes and our test machine is configured with more memory as compared to the Niagara

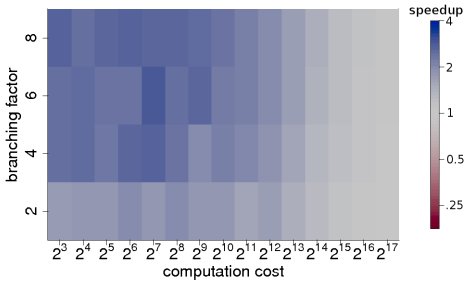
server. Each thread generates candidate solutions until a predetermined number of solutions has been generated per worker thread. The state of a problem instance in the synthetic application consists of a single 64-bit integer value that stores the lower bound of the candidate solution. The lower bound of a child instance is derived from the lower bound of the parent instance using one of three random distributions.

The first distribution is uniform: it does not use the parent's lower bound and selects a new lower bound from a uniform probability distribution across the range of 64-bit values. The second distribution is monotonically increasing: take the parent's lower bound and add to this lower bound a positive integer from a uniform distribution. The monotonically increasing distribution places the least amount of contention at the head of the priority queue, as elements tend to be inserted towards the tail of the queue. The 15 puzzle and graph coloring problems are both instances where the lower bound of a child is never less than or equal to the lower bound of the parent. In the 15 puzzle problem, each child adds a new move to the sequence of moves that have already been taken. In the graph coloring problem, each child adds a new coloring to the set of vertices that have already been colored.

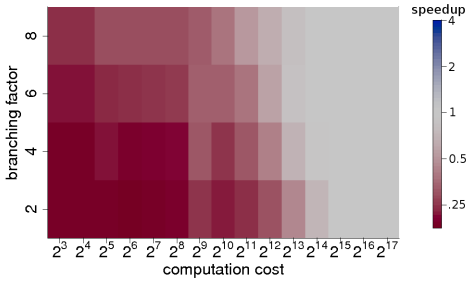
The third distribution is a restricted distribution: take the parent's lower bound and with a probability of 99.98% return the same lower bound, or with a probability of 0.01% increment the lower bound by one, or with a probability of 0.01% decrement the lower bound by one. The primal-dual algorithm we used to solve the 0-1 knapsack problem generates optimal profit estimates that fluctuate within a narrow band of possibilities near the split solution. This characteristic is not specific to the primal-dual method of solving the 0-1 knapsack problem. A restricted distribution of candidate solutions will arise because the optimal solution of the 0-1 knapsack problem tends to be very similar in structure to the greedy solution of the knapsack problem. Fast solvers of the 0-1 knapsack problem that take ad-



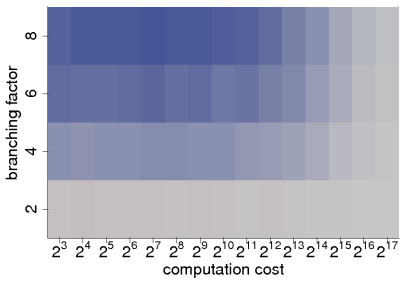
(a) Sun Fire uniform distribution



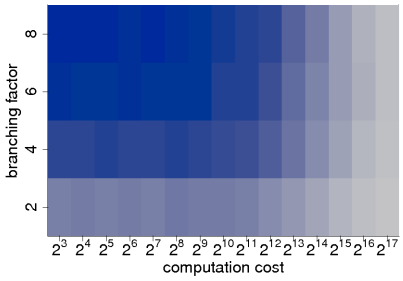
(b) Sun Fire monotonic distribution



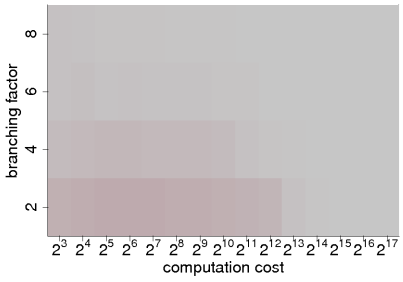
(c) Sun Fire restricted distribution



(d) Intel Xeon uniform distribution



(e) Intel Xeon monotonic distribution



(f) Intel Xeon restricted distribution

Figure 5.9: Relative speedup on synthetic benchmark. Speedup is the ratio of skip list to skip tree execution times.

vantage of the expanding core set of items (see Section 5.6) will have a restricted distribution of solutions.

A proxy calculation is used to represent the elapsed time of computing the lower bound. The proxy calculation is the sum of the integers from 1 to N, where N can be varied to adjust the duration of computation time. This implementation is tunable, reproducible, and independent of any memory hierarchy effects. In our initial experimental setup, the elapsed time of computing the lower bound was to be determined using an independent variable that would vary the amount of time a worker thread would sleep. The sleep interval would simulate the compute time necessary for estimation of a lower bound. However, sub-millisecond precision for sleep intervals is not implemented on the HotSpot Java Virtual Machine for either x86 or SPARC architectures. Based on the limitations of the virtual machine, the proxy calculation of the computation interval was developed to forgo precision of the estimated computation time in exchange for accuracy.

The relative performance on the synthetic benchmark of the skip tree versus the skip list on the Sun Fire and Intel Xeon is shown in Figure 5.9. The value N for the computation cost is shown on the horizontal axis. The branching factor is shown on the vertical axis. A heat-map of the relative speedup is plotted on a logarithmic scale. The logarithmic scale yields the same distance between 200% to 400% relative speedup as with 25% to 50% relative speedup. Positive speedup is blue, negative speedup is red, and no change in performance is white. Each probability distribution of lower bound estimates has a direct impact on the relative performance. A monotonic distribution incurs the least contention among concurrent threads, while a restricted distribution incurs the most contention. The branching factor affects the number of candidate solutions that are computed in between extract minimum operations. Therefore the branching factor dictates the rate of the extract minimum operations. In general, a higher branching factor increases the

performance of the skip tree over the skip list. A lower branching factor is usually more desirable to minimize the overall size of the search space. The benefit of a low branching factor is that the increase in search space size due to calculating two generations of candidate solutions is relatively small when the branching factor is low. The effect of a longer computation time is to dilute the effects of either the performance advantage or disadvantage of the skip tree.

The distribution of lower bound estimates has a dominant impact on the performance of the skip tree priority queue. On the Niagara architecture with a computation cost between the values of 2^3 and 2^{13}, the mean speedup of the uniform distribution is x1.25, the mean speedup of the monotonic distribution is x2.21, and the mean speedup of the restricted distribution is x0.33. On the Intel Xeon architecture with the same computation costs, the mean speedups are x1.79 for the uniform distribution, x2.67 for the monotonic distribution, and x0.85 for the restricted distribution.

5.9 Branch-and-Bound Guidelines

In this section, we provide guidelines for when to select the lock-free skip tree to use as a centralized priority queue in a parallel branch-and-bound application versus the lock-free skip list. These guidelines are based on the results of the application benchmarks on the N Puzzle, graph coloring, asymmetric traveling salesman problem, and 0-1 knapsack problem and the results of the synthetic benchmarking experiments just described.

Rule #1: Avoid contention at the head of the queue

A natural bottleneck in a concurrent priority queue is the head of the queue. When new elements are inserted into the middle of the queue or the end of the queue,

then fewer operations are placed on the head of the queue. In the synthetic benchmark, the monotonic distribution inserts elements into the end of the queue and the uniform distribution inserts elements into any portion of the queue. The restricted distribution inserts candidates within a narrow range of lower bound estimates. The lock-free skip tree shows mean speedups of x2.21 and x2.67 on SPARC and x86 platforms under the monotonic distribution and mean speedups of x1.25 and x1.79 under the uniform distribution. The skip tree exhibits worse performance than the skip list under the restricted distribution with mean speedups of x0.33 and x0.85.

In our performance analysis of the knapsack solver, 30% of extract minimum operations using the skip list required four or more retries and 64% of extract minimum operations using the skip tree required four or more retries. The next highest percentage of four or more retries in extract minimum operations is 5% on the graph coloring problem using the skip list and 3.7% on the 15 puzzle problem using the skip tree. Using the performance analysis for the knapsack solver and the synthetic benchmark results, we can show that a relatively small computational cost for the lower bound estimate correlates with higher contention at the head of the queue. Tables 5.3 and 5.5 show the elapsed time per operation in microseconds for the Sun Fire and Intel Xeon platforms. The elapsed time per computation for the knapsack solver is 4.06 microseconds on the Sun Fire and 0.31 microseconds on the Intel Xeon. The average elapsed time per computation on the other application benchmarks (excluding the asymmetric traveling salesman solver) is 41.7 microseconds on the Sun Fire and 3.36 microseconds on the Intel Xeon. Figure 5.9 (c) and (f) show the relative performance of the skip tree on the synthetic application with the restricted distribution to generate lower bound estimates. Smaller computation costs are correlated with weaker performance from the lock-free skip tree.

Rule #2: Seek monotonic heuristic functions

Consistent (or monotone) heuristic functions are general strategies for traversing through a state space that converge to the solution state without taking any backward steps. Given a node n from the search space and a successor n' of n, the estimated cost of reaching the goal from n is no greater than the cost of getting from n to n' plus the estimated cost of reaching the goal from n': $h(n) \leq c(n, n') + h(n')$. The fifteen puzzle, graph coloring, and traveling salesman problem solvers use monotonic estimates for the lower bound of partial solutions. In the fifteen puzzle, each successor state in the search space represents the addition of a move towards the target configuration of the board. In the graph coloring problem, each successor state assigns a color to an uncolored node until eventually all of the nodes have been assigned colors.The ATSP solver either selects a path along the tour of the graph or eliminates a path from the selection process.

The primal-dual algorithm for solving the knapsack problem uses an inconsistent heuristic function. The knapsack is allowed to overfill so that successor states may remove elements from the knapsack. There are cases when an inconsistent heuristic function is preferable to a consistent heuristic function [139, 140]. In the knapsack problem, the primal-dual algorithm constrains the state space to improve the overall performance of the algorithm. These techniques are most commonly applied to the iterative deepening A* algorithm, which performs iterative rounds of depth-first search. When applied to a breadth-first parallel branch and bound solver, these techniques can add to contention to the head of the priority queue.

7	15	8	2
13	6	3	12
11		4	10
9	5	1	14

(a)

7	6	8	1
11	5	14	10
3	4	9	13
15	2		12

(b)

Figure 5.10: Two fifteen puzzle instances from Korf [121].

Rule #3: Seek large working set sizes

For those applications that satisfy Rules #1 and #2, the greatest improvements in performance of the skip tree versus the skip list are observed for those input problems that have the largest working set sizes. In Figures 5.1-5.7, all scenarios have some threshold number of candidates, T, such that all initial states that have T or more candidate solutions exhibit speedup using the skip tree versus the skip list. In general, it is difficult to estimate the number of partial solutions that will be generated for a specific initial state. For example, the N puzzle problem has no known upper bound that is a function of the initial board state. Figure 5.10 shows two initial states from the iterative deepening A* (IDA) algorithm used to solve one hundred instances of the 15 puzzle in Korf [121]. Both instances have an initial lower bound estimate of 41. The instance on the left generates 24,492,852 partial solutions using the IDA solver. The instance on the right generates 1,369,596,778 partial solutions. We have shown that initial configurations with small working set sizes can perform relatively worse using the skip tree versus the skip list. A smaller working set size implies a shorter total execution time. The absolute time that is lost on these initial configurations using the skip tree is small as compared to the amount of time that is gained in those configurations with large working set sizes.

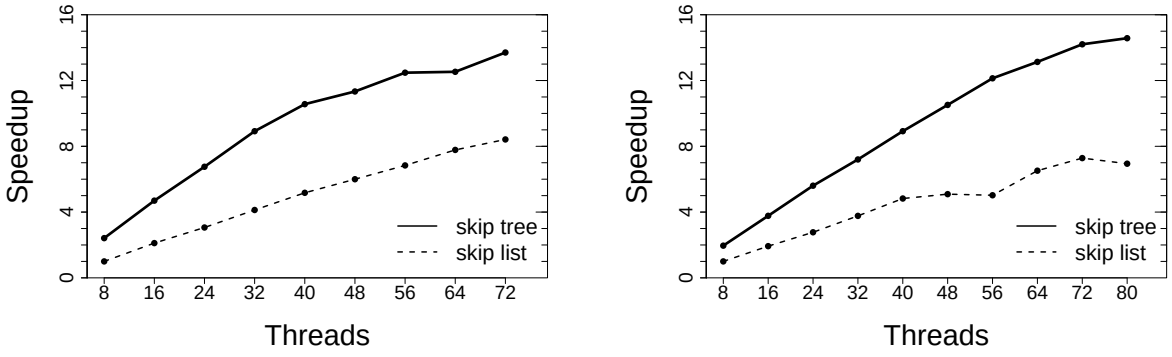
5.10 Shared Memory Supercomputers

5.10.1 Azul Appliance

The Azul compute appliance is a custom shared memory supercomputer designed for the Java runtime environment. The processing unit of the compute appliance is a Vega 3 processor, a 54 core 64-bit RISC processor. In addition to the standard RISC instruction set, the Vega processor has a few specialized instructions to aid the Java virtual machine in object allocation and garbage collection. Up to 16 Vega processors can be installed on a compute appliance, for a total of 864 hardware threads. On the compute appliance, each processor has three available banks of four memory modules. Each core has a 16 kB L1 instruction cache and 16 kB L1 data cache. Each processor has six 2 MB unified L2 caches. Groups of 9 cores share one L2 cache [141]. The Azul compute appliance runs on top of a minimalist operating system and has its own Java virtual machine implementation that is based on the OpenJDK project.

The Azul Java virtual machine uses a pauseless garbage collection algorithm [142]. At no point in the pauseless garbage collection algorithm is there a Stop-The-World pause, a pause in which all application threads must be simultaneously stopped. Pauseless garbage collection provides soft real-time guarantees with regard to memory management. A comparison of the SpecJBB benchmark with the Azul Java virtual machine versus the HotSpot and IBM Java virtual machine reported that worst-case transaction execution times were over 45 times better and average execution times were comparable on the Azul JVM.

The tradeoff of the pauseless garbage collection is that low latency transactions come at a penalty of increased heap size needed for the copying phase of the garbage collection algorithm. We tested fifteen puzzle problem ‘F’ instance and graph coloring problem ‘H’ instance on the Azul compute appliance. These



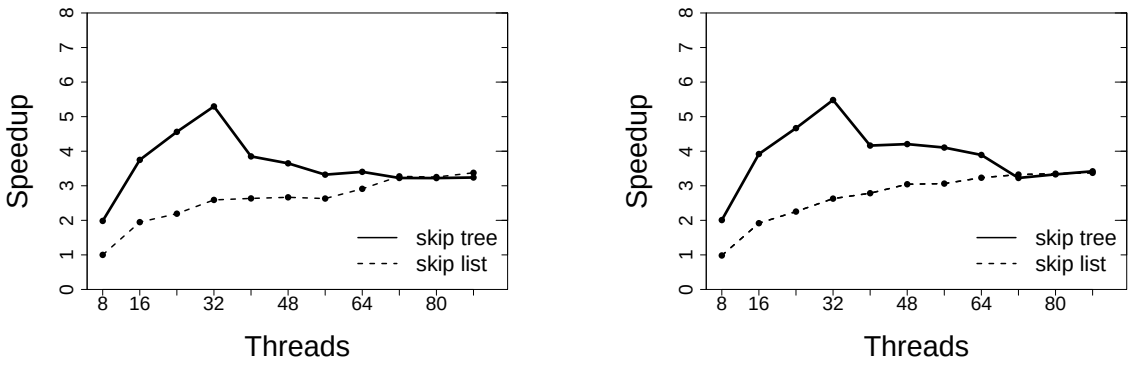
(a) fifteen puzzle problem 'F' instance

(b) graph coloring problem 'H' instance

Figure 5.11: Azul compute appliance. Speedup is relative to skip list execution time using 8 threads.

were selected as the problem instances with the largest working set sizes from the two applications that exhibited a performance improvement using the skip tree versus the skip list. In the branch-and-bound applications, the average number of processed candidates is independent of the number of worker threads that are running. The total working set size and the ratio of garbage objects to live objects are constant factors relative to the number of worker threads. Therefore an increase in worker threads yields an increase in the rate of non-live objects produced per unit of time. Using Azul's Real-Time Profiling and Monitoring Tool, the fifteen puzzle and graph coloring applications running at 100 cores were shown to spend 85% of the total runtime on the garbage collection algorithm under the available heap size limitations.

Our resource allocation on an Azul compute appliance comprised 207 hardware threads and 55 GB of heap space. The results of our tests on the Azul compute appliance are shown in Figure 5.11. The baseline for the two applications we ran is the execution runtime for the skip list using 8 hardware threads. The speedup of the skip tree on the fifteen puzzle is x13.7 using 9 times as many threads when compared to the baseline, and x14.5 on the graph coloring problem when using 10



(a) without NUMA-aware GC (b) with NUMA-aware GC

Figure 5.12: Graph coloring on Altix UV 1000. Speedup is relative to skip list execution time using 8 threads.

times as many threads as the baseline. The speedup of the skip list on the fifteen puzzle is x8.4 using 9 times as many threads as the baseline, and x6.9 on the graph coloring problem when using 10 times as many threads.

5.10.2 SGI Altix UV 1000

The second shared-memory supercomputer that was used to test the application benchmarks was Blacklight, a SGI Altix UV 1000. Blacklight consists of 256 blades connected by a NUMAlink® 5 Interconnect. Each blade holds 2 Intel Xeon X7560 processors with 8 hardware threads per processor, for a total of 4096 hardware threads on the machine. The 16 hardware threads of each blade share 128 GB of memory; the total capacity of the machine is 32 TB. The processors are connected in an 8 x 8 paired node 2D torus topology. Blacklight is running SUSE Linux Enterprise Server 11 with a modified 2.6.32.12 Linux kernel.

The HotSpot Java virtual machine can run the fifteen puzzle and graph coloring benchmarks on Blacklight up until about 64 hardware threads before no further speedup is observed by the addition of more processors. The same behavior is observed for the lock-free skip list and the lock-free skip tree. The default garbage

collection implementations in HotSpot are NUMA-unaware. The parallel collector, also known as the throughput collector, can be optionally enabled with NUMA-aware support. Enabling NUMA support on HotSpot when running on Blacklight crashes the virtual machine during its initialization phase. This behavior has been observed with Java SE 1.6.0_22 and OpenJDK 1.7.0 binary release 114. This crash is caused by an interaction of two features of the Linux kernel that are used to allocate hardware resources on shared-memory NUMA architectures.

libnuma is a user space shared library that provides an API for implementing NUMA policies in applications [143]. It allows an application to determine the underlying memory topology of the hardware at runtime by querying the operating system. libnuma can also be used to bind a thread to a specific processor. NUMA support has been available in the Linux kernel in one form or another for all of the 2.6 release series.

Another mechanism for assigning a set of processors and memory nodes to a set of tasks in the Linux kernel is the cpusets interface. A cpuset is composed of a set of processor nodes and a set of memory nodes. The root cpuset contains all of the processor nodes and memory nodes. Given a cpuset, a child cpuset can be defined that contains a subset of the parent resources. Cpusets may be marked exclusive, which ensures that no other cpuset except direct ancestors and descendants may contain overlapping processor or memory resources. Cpusets appeared in version 2.6.7 of the Linux kernel. The Portable Batch System (PBS) is a job scheduler for the allocation of batch jobs in a shared computational environment. PBS can be configured to use cpusets to isolate tasks to a specific set of processors.

The NUMA-aware garbage collector in the HotSpot Java virtual machine is designed to use libnuma for memory allocation, but does not query the cpuset interface. The current implementation queries the machine for the total number of online processors using the sysconf POSIX interface [144]. The libnuma API sets

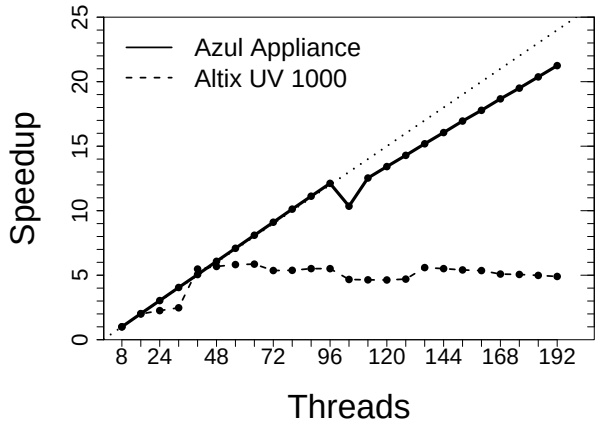


Figure 5.13: 16 queens on shared-memory supercomputers. Speedup is relative to skip list execution time using 8 threads.

a hard limit on the number of processors that can participate in a NUMA memory set. The total number of online processors on the Blacklight supercomputer is much larger than the processor limit set by the libnuma API. When the Java virtual machine attempts to create a NUMA memory set as large as the number of total processors on the system, the result is a crash in the initialization of the virtual machine. We have written a patch for the OpenJDK that detects whether the cpuset interface is present, and if so, then uses the cpuset interface to determine the number of memory processors available.

The libnuma library exports two versions of its API to the user: libnuma 1.1 and libnuma 1.2. The libnuma 1.1 API is not aware of cpuset constraints on a processor, and the libnuma 1.2 API is aware of cpuset constraints. The OpenJDK currently uses the libnuma 1.1 API. We have written a second patch to update the OpenJDK to use the libnuma 1.2 interface. While writing the patch to the OpenJDK for the libnuma 1.2 interface, we encountered a bug in the libnuma library. The documentation for the function numa_get_mems_allowed() states the function returns a bit vector of all the available memory nodes in the cpuset of a process. The function returned an empty bit vector when a cpuset was applied to a process. We submit-

ted a patch to correct the behavior of numa_get_mems_allowed() in the libnuma library. The patch was adopted in the release of libnuma 2.0.6-rc4.

Figures 5.12a and 5.12b show the results of the graph coloring problem on the Altix UV 1000. Figure 5.12a uses the non-NUMA concurrent garbage collection algorithm, and Figure 5.12b uses the NUMA-aware concurrent garbage collector. Speedup is normalized to the runtime of 8 threads using the skip list priority queue. Each blade of the supercomputer can support 32 hardware threads: 16 cores with hyperthreading enabled. In all cases, the relative speedup does not improve using more than one blade of the machine. The speedup using 88 threads and the skip list priority queue is x1.3 the speedup using 32 threads. The speedup using 88 threads and the skip tree priority queue is x0.6 the speedup using 32 threads. The NUMA-aware concurrent garbage collector yields a small improvement on the performance of the skip tree priority queue on the graph coloring solver.

To study the behavior of the OpenJDK on the Altix UV 1000 using a standard Java benchmark that does not rely on our lock-free skip tree, we selected the N queens puzzle benchmark from the Java 7 fork/join testing framework. The fork/join testing framework has been written by the members of the JCP JSR-166 Expert Group to measure the performance of the classes in the java.util.concurrent library. The runtime of the N queens problem was measured for the 16 x 16 chess board with 16 queens on the Altix UV 1000 and the Azul Systems compute appliance. The relative speedup on this problem is shown in Figure 5.13. Speedup is normalized to a runtime of 8 threads on the UV 1000 for measurements on that supercomputer, and a runtime of 8 threads on the Azul Systems compute appliance for measurements on that supercomputer. Speedup is almost linear on the Azul compute appliance. The N queens benchmark exhibits marginal speedup using more than one blade on the Altix UV 1000.

In this section, we have shown that it is possible to use shared-memory supercomputers efficiently to solve parallel branch-and-bound problems. For those parallel branch-and-bound applications that exhibit the properties enumerated in Section 5.9, the lock-free skip tree shows up to a x2.1 improvement in runtime as compared to the lock-free skip list when running on 88 hardware threads.

In order for an application to scale across a shared-memory interconnection communication layer, it is necessary for all the runtime layers underneath the application to scale as well. The Azul compute appliance runs on top of a minimalist operating system and has its own Java virtual machine implementation that is based on the OpenJDK project. The SGI Altix is running a modified 2.6.32.12 Linux kernel and a patched version of the OpenJDK 7 release. The OpenJDK runtime was unable to scale across multiple blades when measured by the Java 7 fork/join testing framework. There are several possible sources of resource contention on the SGI Altix that are absent on the Azul compute appliance. The virtual memory management of the operating system may be negatively impacting performance of the OpenJDK garbage collection algorithm [145]. Virtual memory management on the Azul appliance is cooperatively managed by the operating system and the Java virtual machine. An Azul compute appliance has a relaxed consistency model that shared characteristics of the Itanium memory ordering specification. The Azul JVM is responsible to inserting memory fences wherever necessary to maintain the Java memory model. Given the commercial success of a Java compute appliance that can scale to hundreds of threads using specialized hardware, a specialized operating system, and a specialized Java virtual machine, it is most likely that significant modifications would be necessary to achieve the same scalability on a conventional hardware and software stack.

Chapter 6

Lock-Free Burst Trie

In this chapter we introduce the lock-free burst trie as an alternative to the lock-free skip tree for those domains that can support a hash function that facilitates radix sorting. To the best of our knowledge the lock-free burst trie algorithm is the first of its kind. The lock-free trie offers a space/time tradeoff as compared to the lock-free skip tree. Interior nodes of the lock-free trie store unused children references in order to allow for a search through the tree that does not rely on element comparison operations. After introducing the lock-free trie algorithm, we present a series of synthetic benchmarks that measure the throughput of the concurrent trie as compared to concurrent skip list, skip tree, and B^{link}-tree implementations. The burst trie exhibits the highest peak throughput across all four scenarios and two architectures. The mean peak throughput of the burst trie is x3.5 higher than the data structure with the second highest peak throughput across all scenarios on the Sun Fire T1000 and x2.8 higher across all scenarios on the Intel Xeon L5430.

In Chapter 4 we constructed a lock-free cache-conscious data structure that implements an abstract ordered set with $O(\log n)$ expected cost for sequential `contains`, `add`, and `remove` operations. Next, we study whether it is possible to improve upon the $O(\log n)$ expected cost in designing a lock-free cache-conscious

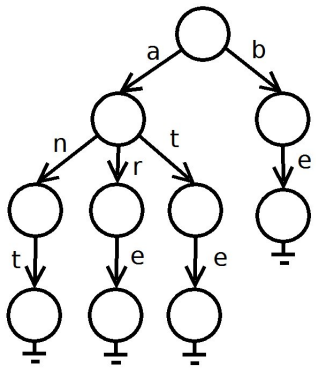


Figure 6.1: Trie example. Contains the strings {"ant", "are", "ate", "be"}.

data structure that implements an ordered set. The lower bounds for comparison-based sorted data structures is $\Omega(\log n)$. In order to improve upon these lower bounds, we must use a radix sorting scheme as the basis of the data structure. The question of how to improve upon the $O(\log n)$ expected cost of the skip tree is motivated by the existence of the hopscotch hashing algorithm. The hopscotch algorithm defines a class of concurrent cache-conscious resizable hash tables with $O(1)$ expected time for `contains`, `add`, and `remove` operations. There exists a lock-free cache-conscious hopscotch hash table algorithm [85], yet the hash table implements an abstract unordered set abstraction. One ordered data structure that combines tree-based data structures and hash-based data structures is a trie, or a prefix tree. In this chapter we introduce a novel lock-free trie algorithm and compare the performance of the lock-free trie implementation to our lock-free skip tree implementation.

A trie, or a prefix tree, is an ordered tree data structure. A trie node separates an input element into a prefix component and a remainder component, and then uses the prefix to locate the next node in the data structure. A child node will accept the remainder component of a parent node and recursively search for the target element [146, 88, 147–149, 87]. The leaf nodes contain the remainder components of one or more elements. One example of a trie is shown in Figure 6.1.

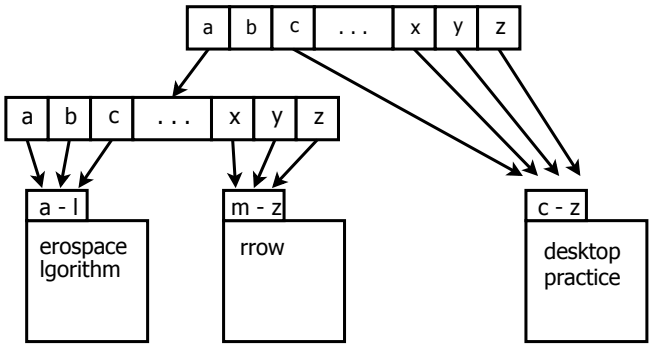


Figure 6.2: HAT-trie data structure. The figure is adapted from Askitis and Sinha [86].

The set of domains T that are permissible to be stored in a trie data structure are those domains that have defined a total ordering and a prefix operation $pre(e) : T \rightarrow T$ for all elements of length greater than 0. The prefix operation must shorten the length of the element, $|pre(e)| < |e|$, and it must preserve the total ordering of the elements: $pre(e_1) < pre(e_2) \rightarrow e_1 < e_2$. A trie provides $O(|e|)$ `contains`, `add`, and `remove` operations while maintaining the ordered property of the elements. The interior nodes may contain either a strict subset of all possible elements returned by the prefix operation (as in Figure 6.1), or a set equal to all possible elements returned by the prefix operation (as in Figure 6.2). If the interior nodes contain a strict subset of all possible elements, then an interior node with i prefixes contains $2i + 1$ possibly-null references to children nodes. One variation of tries that contains exactly one prefix per interior node is known as a ternary search tree [150, 151], with each interior node containing three child references. If the interior node contains all possible elements returned by the prefix operation, then an interior node with i prefixes contains i possibly-null references to children nodes.

A burst trie is a specialized trie such that each leaf node of the trie is a container for some small, constant number of elements. As elements are inserted into the burst trie, the leaf nodes increase in size until a container reaches its maximum

size [87]. When a container reaches its maximum size, the container is burst into an internal trie node along with a family of new empty container nodes. The child container nodes are then populated by the elements that were stored in the original container. Each interior node of the burst trie contains all possible elements returned by the prefix operation. In the original burst trie paper, the container is implemented using a balanced binary search tree. Askitis and Sinha [86] define the HAT-trie as a variation of the burst trie whereby the containers are implemented as fixed size arrays. An example of a HAT-trie is shown in Figure 6.2.

6.1 Lock-Free Algorithm

The lock-free burst trie implements a linearizable ordered set data structure over some domain T. On the domain T a prefix function is defined that satisfies the following properties. The prefix operation must shorten the length of the element, $|pre(e)| < |e|$, and it must preserve the total ordering of the elements: $pre(e_1) < pre(e_2) \rightarrow e_1 < e_2$. Three operations are supported by the lock-free burst trie: `contains`, `add`, and `remove`. The lock-free trie contains two types of nodes: an *array* node and an *index* node. An array node is implemented as an immutable array of elements $e \in T$. The array nodes form the leaf nodes of the burst trie. An array node may contain at most n elements for some constant $n \in \mathbb{N}$. An index node contains an array of atomic references of length i, where i is the total number

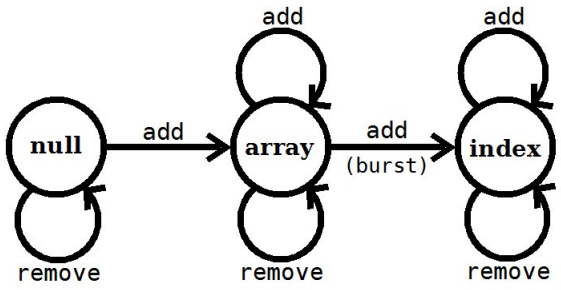


Figure 6.3: Burst trie state diagram.

of prefixes that could be returned by the prefix operation. An array of atomic references supports the atomic compare-and-swap operation for each cell of the array. Each child reference in a container node can point to one of the following three types of objects: a null reference, an array node, or an index node.

A new burst trie is represented with a single index node with all children as null references. The state transition diagram for the lock-free burst trie is shown in Figure 6.3. Each child reference moves from the null state to the array state to the index state during its lifecycle. A child reference is permitted only to move forward in the state transition diagram. A child reference in the array state can become as small as an array of length zero. The subtree of a node in the index state can only become as small as recursively allowed by the child references of the index node. Our experience with designing a lock-free skip tree has shown that adding elements into a lock-free data structure is relatively easy. Removing elements from a lock-free data structure and eliminating nodes of the data structure is more complicated. The trie design requires a space/time tradeoff to improve the upper bound of `contains`, `add`, and `remove` operations in the absence of thread contention from $O(\log N)$ to $O(|e|)$. Our lock-free implementation further exploits the reliance on available space. Empty array nodes and index nodes are not eliminated from the data structure. As we will show, retaining empty nodes simplifies the process of bursting array nodes into index nodes in the presence of concurrent updates to the data structure.

A `contains` operation begins at the root of the trie. The prefix of the target element is used to determine the index of the correct child reference in which to continue traversing the tree. The tree traversal recursively searches the subtrie whose root is the child reference. Eventually, either a null child node or an array child node will be encountered. If a null child reference is encountered, then the target element is not a member of the abstract set. If an array node is encountered,

then the target element is a member of the abstract set if-and-only-if it is a member of the array node. Recall that array nodes are immutable objects. Therefore the first encounter with an array node object is a linearization point of the `contains` operation. The other linearization point is the first encounter with a null child reference.

On the traversal from a parent index node to a child index node, the prefix of the target element can be discarded for the remainder of the search. Target elements that are shorter than the path length through the trie are resolved with the addition of a special terminal token to the set of possible values in the prefix operation. The terminal token is reserved for no more input, we use the '\0' character for ASCII strings. When the terminal token is encountered in an index node, the child reference is either a null reference, an array node that contains zero elements, or an array node that contains a single element consisting of the special token. Tokens are not consumed when traversing from a parent index node to a child array node.

An `add` operation begins as a `contains` operation with a trie traversal to find the point of insertion. At the end of the traversal a child reference is reached that is either a null reference or an array node reference. If an array node reference is reached and the array node contains the target element, then the `add` operation terminates. Otherwise a compare-and-swap is performed on the designated index in the atomic reference array of the index node. The linearization point of all update operations on the lock-free burst trie is the parent index node of the array node to be updated. Array nodes are immutable and linearizable semantics are preserved through atomic operations on index nodes. If the child reference is either a null reference or an array node reference that contains less than n elements, then a new array node is created that contains all the existing elements and the new element. A compare-and-swap replaces the existing child reference with the new array node. If the array node contains n elements, then the array node must be burst into an

index node before the new element can be added.

The burst of an array node creates an index node whose children contain the same set of elements as in the original array node. A compare-and-swap replaces the full array node with the new index node. Concurrent `add` operations that are attempting to insert into a full array node will perform redundant work as each thread creates an independent index node as a result of a burst. Only one of the concurrent add operations will successfully compare-and-swap the full array node with the new index node. After a node has been burst, the insertion of the target element by the `add` operation may continue.

A `remove` operation is successful through the transformation of an array reference into a new array reference that contains all of the elements in the original array node with the exception of the element to be removed. Empty array nodes are those that contain zero elements. This technique has been borrowed from the lock-free skip tree implementation, where zero element arrays are used in shrinking the data structure. The lock-free skip tree uses on-demand elimination of zero element arrays. New elements are not inserted into zero element arrays to ensure that the arrays can be removed from the data structure. The lock-free burst trie does not shrink when elements are deleted. The burst trie implementation allows new elements to be inserted into a zero element array using the compare-and-swap technique.

6.2 Synthetic Benchmarks

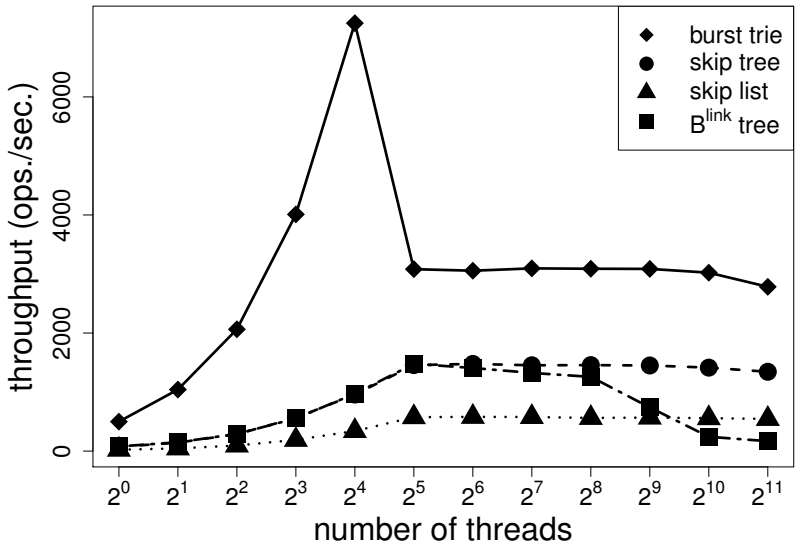
Performance analysis is conducted with the experimental design that was used in chapters 3 and 4 [35, 104, 95]. Synthetic workloads are created that vary in proportions of `contains`, `add`, and `remove` operations and in the number of unique elements stored by the data structure. Half of the workloads use a $\frac{90}{100}$:$\frac{9}{100}$:$\frac{1}{100}$ ra-

tio of operations. The other half use a $\frac{1}{3}$:$\frac{1}{3}$:$\frac{1}{3}$ ratio of operations. Five million operations are executed in each independent trial, while the total throughput of the data structure as measured by the number of concurrently executing threads varies from 2^0 to 2^{11}. The benchmarks in Chapters 3 and 4 used 32-bit integers as the domain from which five million elements were randomly selected. In the throughput benchmarks of sequential burst trie implementations, randomly generated strings are the most common domain that is tested in the literature [87, 86]. Strings are generated with a random length from a uniform distribution between fifty and one hundred characters. In our implementation, the set of ASCII characters is used as the set of acceptable input characters to be stored in the data structures. By limiting the input to the set of ASCII characters, each index node of the lock-free burst trie contains two hundred fifty six child references. If the set of input characters had been extended to the set of UTF-16 Unicode characters supported by the Java language specification, some sort of multilevel index node would have been necessary to accommodate all 2^{16} possible input tokens.

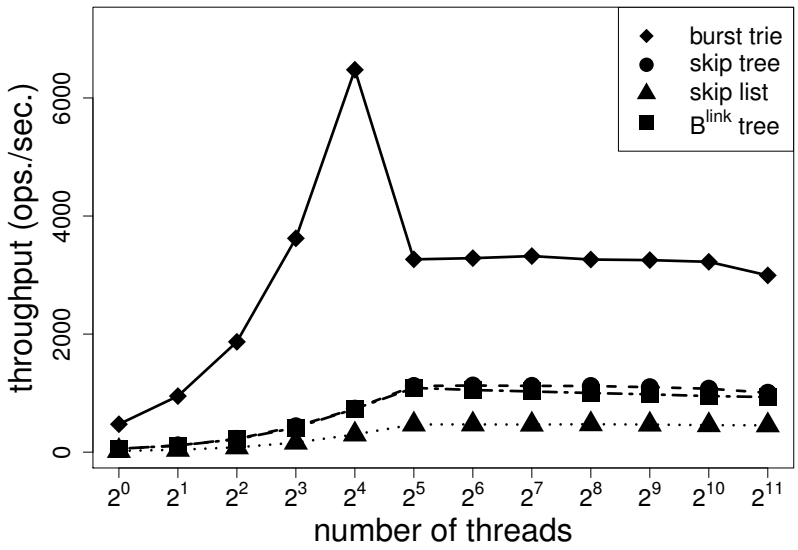
Benchmarks were evaluated on a Sun Fire T1000 and an Intel Xeon L5430. The benchmarks were executed on the 32-bit server version of the HotSpot Java Virtual Machine version 1.6.0 update 16. Explicit parameters for the virtual machine are 2 GB heap size and 128 kB thread stack size. The Sun Fire has eight UltraSPARC T1 cores at 1.0 GHz and thirty two hardware threads. The cores share a 3 MB level-2 unified cache. The operating system version on the Sun Fire T1000 is Solaris 10. The Xeon L5430 has four cores at 2.66 GHz and eight hardware threads. Each pair of cores shares a 6 MB level-2 unified cache. The operating system distribution is CentOS release 5.3 with Linux kernel 2.6.29-2.

We compare four implementations of linearizable concurrent ordered sets:

- skip list - the `ConcurrentSkipListSet` in the `java.util.concurrent` library. Written by members of the JCP JSR-166 Expert Group.

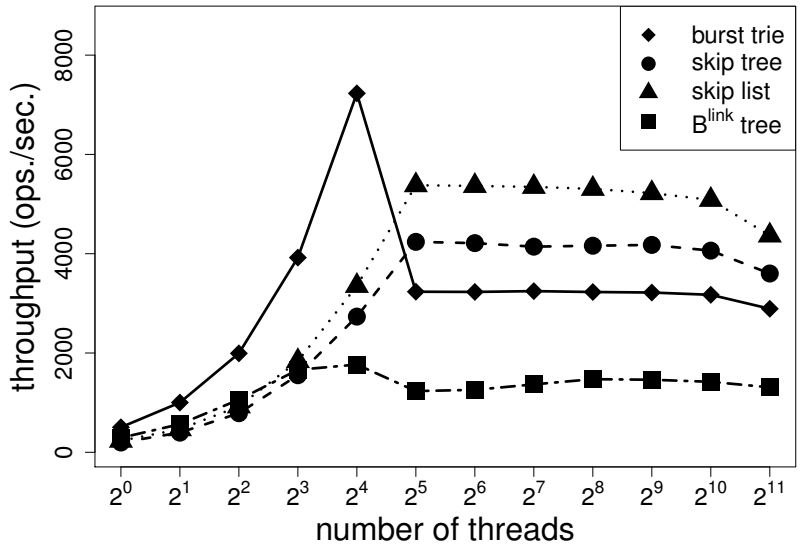


(a) read scenario with 5,000,000 elements

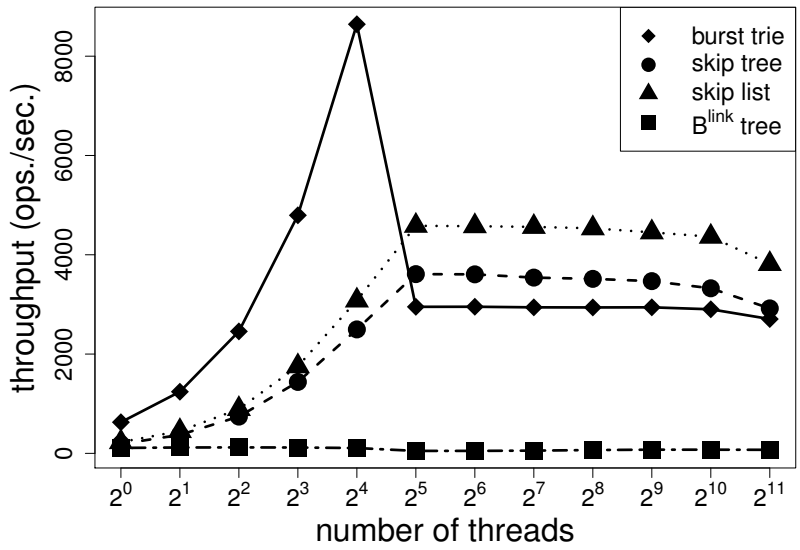


(b) write scenario with 5,000,000 elements

Figure 6.4: Large synthetic string benchmark on Sun Fire T1000

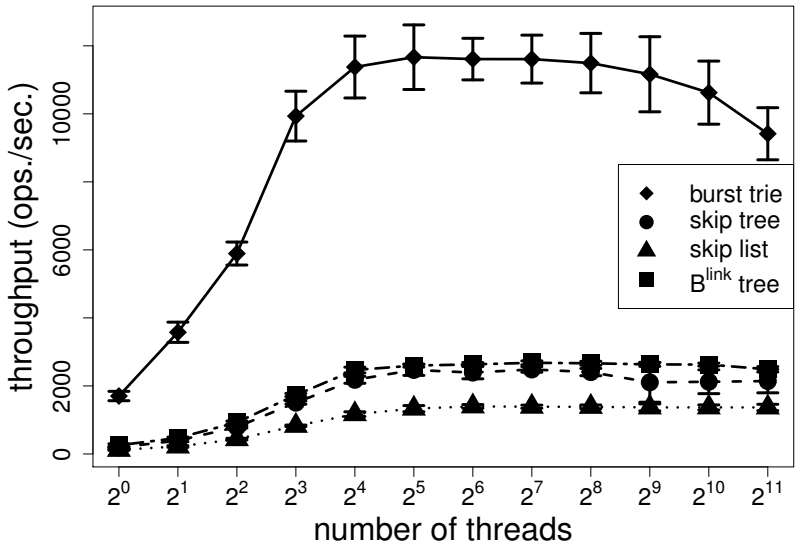


(a) read scenario with 512 elements

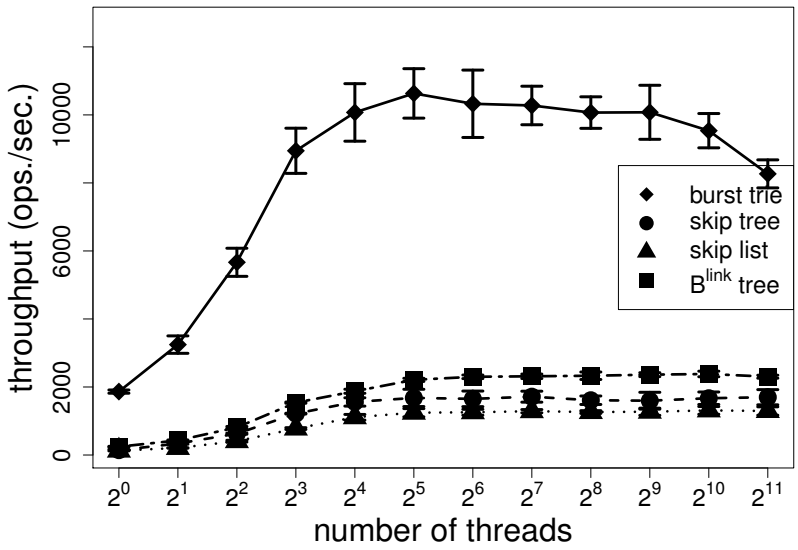


(b) write scenario with 512 elements

Figure 6.5: Small synthetic string benchmark on Sun Fire T1000

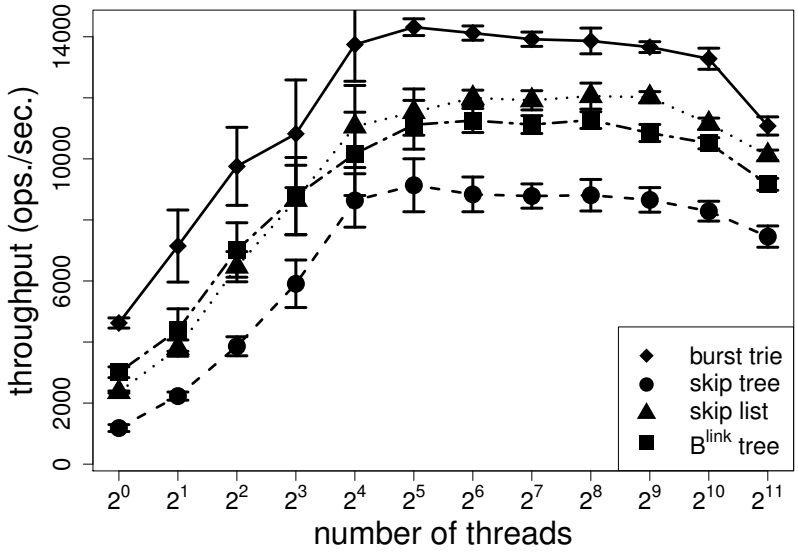


(a) read scenario with 5,000,000 elements

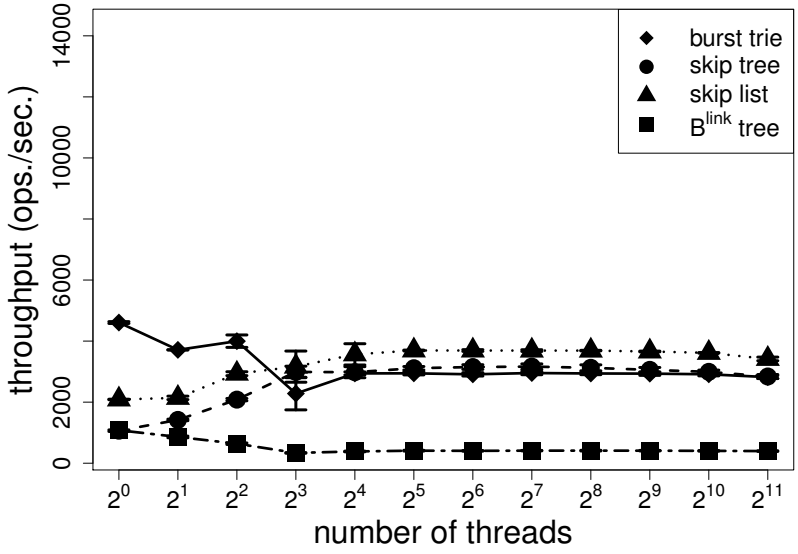


(b) write scenario with 5,000,000 elements

Figure 6.6: Large synthetic string benchmark on quad core Intel Xeon



(a) read scenario with 512 elements



(b) write scenario with 512 elements

Figure 6.7: Small synthetic string benchmark on quad core Intel Xeon

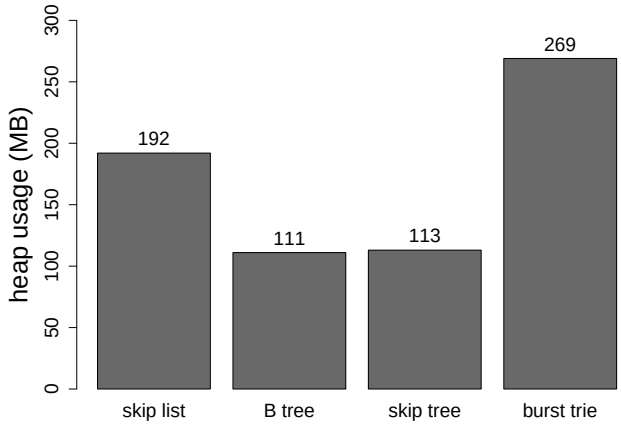


Figure 6.8: Heap usage of data structures in the beginning of the write scenario with 5,000,000 elements on Sun Fire T1000.

- skip tree - the lock-free skip tree algorithm defined in chapter 4.
- burst trie - the lock-free burst trie algorithm defined in this chapter.
- B^{link}-tree - a concurrent B-tree algorithm developed by Lehman and Yao [101] and refined by Sagiv [97].

Configuration parameters for the skip tree and B^{link}-tree were selected from the optimal values in the parameter sweeps conducted in Section 4.7. The probability of failure q for the geometric distribution of heights in the skip tree was assigned a value of $1/32$ for the Sun Fire T1000 and $1/8$ on the Intel Xeon L5430, which are the optimal values for the synthetic benchmarks as determined in Chapter 4. The minimum node length selected for the B^{link}-tree was 256 for both architecture configurations. The maximum size of array nodes in the burst trie was selected to be 32 elements.

The results of the synthetic benchmarks are shown in Figures 6.4 - 6.5 for the Sun Fire T1000 and Figures 6.6 - 6.7 for the Intel Xeon L5430. The burst trie exhibits the highest peak throughput across all four scenarios and both architectures. The

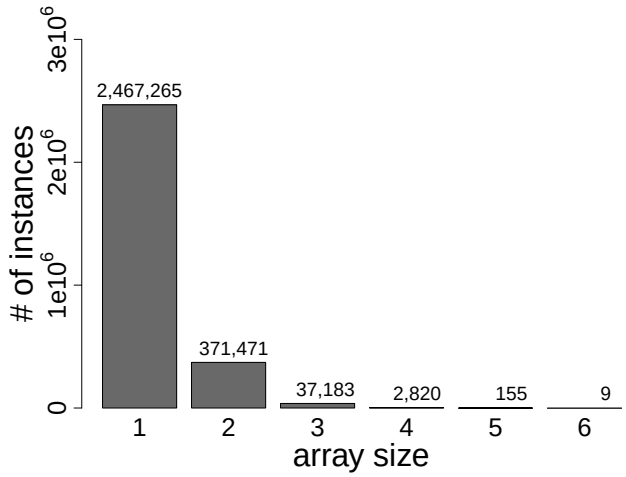


Figure 6.9: Distribution of trie array node sizes in the beginning of the write scenario with 5,000,000 elements on Sun Fire T1000.

mean peak throughput of the burst trie is x3.5 higher than the data structure with the second highest peak throughput across all scenarios on the Sun Fire T1000 and x2.8 higher across all scenarios on the Intel Xeon L5430. The skip list, skip tree, and B^{link}-tree have the same relative performance characteristics as measured in the synthetic benchmarks from Section 4.7. The skip tree dominates the skip list when the working set size exceeds the cache size. The B^{link}-tree exhibits low throughput for workloads that contain a small number of elements due to lock contention. The peak throughput of the burst trie is observed at 16 threads across all four workloads on the Sun Fire, unlike the skip list, skip tree, and B^{link}-tree which all have peak throughput at 32 threads. Each UltraSPARC T1 processor is made up of 8 cores, and each core supports four threads, known as a "thread group." Each thread in the group has a unique set of registers and an instruction buffer, but the thread group shares L1 cache, instruction and data TLB entries, execution units and other pipeline resources [152]. The maximum throughput is observed when each thread group is only half full across all processors, presumably due to contention for one or more thread group shared resources.

The heap utilization of all four data structures was measured for the write scenario with five million elements on the Sun Fire T1000. A heap dump was generated subsequent to the loading of strings that are designated for `contains` and `remove` operations, and prior to the actual multithreaded testing of throughput. Unreachable objects are eliminated from the heap dump, and all instances of classes within the data structure are tallied. The space allocated for the strings themselves is not included in the analysis. The results of the heap analysis are shown in Figure 6.8. At the time of the heap dump, the data structure consists of the elements for `contains` and `remove` operations or approximately $3,333,333$ elements. The skip tree and B^{link}-tree use an average of 35 bytes per element, the skip list uses an average of 60 bytes per element and the burst trie uses an average of 84 bytes per element. The synthetic benchmarks on the Niagara and Intel architectures were executed using a 32-bit Java virtual machine. The distribution of array node sizes for the burst trie is shown in Figure 6.9. Why is an exponential distribution observed for the array sizes? The array nodes can have a size between 0 and n elements, where n is some maximum allowable size ($n = 32$ in the synthetic benchmarks). Given a uniform distribution of input elements to the data structure, at first impression we would expect a uniform distribution of array sizes. But what happens when an array node is burst? In circumstances when the input elements can be modeled as a uniform distribution, we would expect the creation of $\min(m, n)$ new array nodes, where m is the number of indices per index node ($m = 256$ in the synthetic benchmark). The average size of each new array node is $\lceil \frac{n}{m} \rceil$. Therefore, when $m \gg n$ there is a bias towards small array nodes. The abundance of small array nodes yields an inefficient use of heap space as singleton elements become boxed inside array nodes of length 1.

6.3 Lock-free Trie and Tree Comparisons

The burst trie exhibits the highest peak throughput across all four scenarios and both architectures in the synthetic benchmarks. With an input domain of randomly generated ASCII strings of fifty to one hundred characters in length, the heap usage of the burst trie is x2.4 larger as compared to the skip tree. We have shown that the increase in heap usage is due to the large number of singleton elements that become boxed inside array nodes of length 1. Askitis and Sinha [86] have shown that array nodes from several different prefix sets can be combined into a single node in order to avoid an overwhelming number of small array nodes. The illustration of a HAT trie in Figure 6.2 contains three combined nodes: 'a' - 'l', 'm' - 'z', and 'c' - 'z'. There are several issues to be worked out in order to use a combined array node in the lock-free burst trie. The new concurrent algorithm would contain a new fourth state, "combined array", to be placed in between the states "array" and "index" in Figure 6.3. Askitis and Sinha use several heuristics to determine the boundaries for defining new combined array nodes upon the bursting of an index node. It would be prudent to prove the generality of these heuristics beyond the specific application of randomly distributed ASCII strings. Another strategy for reducing the memory footprint of a trie is to cache several child entries inside each index of an array node. This strategy is employed by Bagwell's hash array mapped tries [153].

The primary limitation of the lock-free burst trie is that it places an additional constraint upon the domain set of elements stored in the data structure. Our implementation of the burst trie accepts a proxy object when it is constructed. The type of the proxy object implements an interface upon the domain set T that contains one method with the signature $prefix\,(input : T,\, depth : \mathbb{N}) \rightarrow \mathbb{N}$. Given a member of the domain set, $input$, and a depth at which to extract a prefix of $input$, the method returns an integer between 0 and $m - 1$, where m is the number of in-

dices per index node. The prefix operation on the domain set must preserve the total ordering of the elements: $\forall d \in \mathbb{N},\ prefix(input_1, d) < prefix(input_2, d) \rightarrow input_1 < input_2$. The parameter m has an impact upon the performance and heap usage of the trie. A small value of m will increase the depth of the trie, and a large value of m will increase the size of each index node. For very large value of m, such as $m = 2^{16}$ in the case when the input domain is the set of all UTF-16 Unicode characters supported by the Java primitive type '`char`', a multi-level indexing structure would be required for the index nodes. For UTF-16 Unicode characters, a burst trie implemented using a one level index would take up 2^{21} bytes or 2 MB of heap space per index node, assuming 32-bit pointers per child reference.

An optimal lock-free burst trie requires the tuning of three parameters: m, the number of indices per node, n, the maximum number of elements per array node, and l, the number of levels in the indexing structure. An optimal set of configuration values for a burst trie parametrized on the domain set T is dependent on the selected implementation of the $prefix$ function for that specific domain. In contrast, the lock-free skip tree requires the tuning of exactly one parameter, the average number of elements stored per node. As we have shown in the application benchmarks in sections 5.3 and 5.4, the average number of elements in a skip tree node can be tuned to a specific architecture and deliver optimal results independently of the domain set. The lock-free burst trie has promise for improved performance in a concurrent application, but the complexity of its configuration limits its usefulness as a general-purpose data structure.

The question raised at the onset of this chapter was whether it is possible to improve upon the $O(\log n)$ expected cost in designing a lock-free cache-conscious data structure that implements an ordered set. A solution has been found that provides $O(|e|)$ sequential `contains`, `add`, and `remove` operations. The lock-free burst trie inherits both desirable and undesirable characteristics from its parental data

structure designs, the search tree and the hash table. The trie maintains an ordered set abstraction and exhibits a relatively higher throughput than the lock-free skip tree on the synthetic benchmarks. In order to benefit from the advantages of the lock-free trie it is necessary to define a prefix function on the input domain in order to use the data structure. The tuning of the three parameters m, n, and l play an important role in the performance of the data structure. The skip tree has exhibited good performance characteristics across a range of input domains: synthetic benchmarks using 32-bit integer input in Section 4.7, synthetic benchmarks using ASCII character input in the previous section, and application benchmarks solving the 15 Puzzle and Graph Coloring problems in Sections 5.3 and 5.4. The hopscotch algorithm has also showed good performance characteristics in several application domains [154, 155]. The skip tree leverages the strengths of comparison-based indexing, and the hopscotch algorithm leverages the strengths of hash-based indexing. It is not surprising that the combination of these two algorithmic designs does have performance advantages but is constrained to work successfully only for those input domains that can leverage both the comparison-based and hash-based indexing schemes.

Chapter 7

Conclusions & Future Work

We have shown that the design of cache-conscious, linearizable concurrent data structures has advantageous performance that can be measured across multiple architecture platforms. The improved performance arises from the memory wall phenomenon that is ubiquitous to current multi-core systems and almost certainly will continue to affect future many-core systems. The two primary design contributions of this dissertation are the novel lock-free skip tree and lock-free burst trie algorithms. In both algorithms, read-only operations are wait-free and modification operations are lock-free.

A series of synthetic benchmarks have shown that our lock-free skip tree and burst trie implementations perform up to x2.3 and x3.5 faster in read-dominated workloads on SPARC and x86 architectures, respectively, compared to the state of the art lock-free skip list. The minimum performance of the skip tree across all workloads and architectures is x0.87 relative to the skip list performance. An analysis of heap utilization of the data structures in the synthetic benchmark reveals the lock-free skip tree to use 59% of the heap utilization of the skip list and the lock-free burst trie to use 140% of the skip list heap utilization.

We selected four NP-hard problems to solve using a parallel branch-and-bound

technique: N puzzle, graph coloring, asymmetric traveling salesman, and 0-1 knapsack. Two of the applications are x2.3 and x3.1 faster when using the skip tree as a concurrent priority queue as compared to the lock-free skip list priority queue. On a shared-memory supercomputer, the speedup of the skip tree on the fifteen puzzle is x13.7 using 9 times as many threads when compared to the baseline execution, and x14.5 on the graph coloring problem when using 10 times as many threads as the baseline execution. The speedup of the skip list on the fifteen puzzle is x8.4 using 9 times as many threads as the baseline, and x6.9 on the graph coloring problem when using 10 times as many threads as the baseline. The two branch-and-bound applications are x1.6 and x2.1 faster using the skip tree versus the skip list as a centralized priority queue running on either 80 or 88 hardware threads. Based on the four application benchmarks and the synthetic benchmark, guidelines were determined for selecting the lock-free skip tree to use as a centralized priority queue in a parallel branch-and-bound application.

We have made the lock-free skip tree implementation available online [11]. The source code for the lock-free skip tree has been released to the public domain, as described at `http://creativecommons.org/licenses/publicdomain`. It is our intent to submit this implementation to the Java Concurrency Expert Group as a proposal for inclusion to jsr166z and eventual inclusion into the Java 8 concurrency library. We anticipate that the configuration parameters of the lock-free skip tree implementation will not contribute to a significant variability in application performance. The lock-free skip tree relies on a single parameter, q, to determine the average number of elements per node of the data structure. To improve the efficiency of the pseudo-random number generator, q is restricted in our implementation to $1/2^k$, $k \in \mathbb{N}$. The optimal value of q was determined to be $1/8$ on the quad core Intel Xeon and $1/32$ on the Sun Fire T1000. Our study of parameter variations on q for both architectures (Figures 4.17 - 4.20) have shown that values

of k that are near each other yield similar maximum throughput on the synthetic benchmarks.

We anticipate to find more applications of the lock-free skip tree and lock-free burst trie data structures. The lock-free skip tree implementation is a drop-in replacement for the popular java.util.concurrent skip list implementation. The lock-free skip tree will be adopted by applications that experience a performance bottleneck in skip list operations. For example, the Apache Cassandra distributed database relies on the skip list to store column families in sorted order (see Section 5.1). Our tests indicate that the disk access latency and network access latency comprise the primary bottlenecks in database throughput in the Cassandra benchmark suites. If configured as an in-memory database in a network with low-latency interconnects, it is possible that our skip tree implementation would improve performance on the benchmarks.

Our lock-free burst trie implementation contains both opportunities and challenges for its adoption in real-world applications. In the synthetic benchmarks, the mean peak throughput of the burst trie is x3.5 higher than the data structure with the second highest peak throughput across all scenarios on the Sun Fire T1000 and x2.8 higher across all scenarios on the quad core Intel Xeon. Good performance of the burst trie requires tuning three parameters: m, the number of indices per node, n, the maximum number of elements per array node, and l, the number of levels in the indexing structure. The target domain of elements to be stored in a burst trie must also define an efficient prefix function. One target domain would be bioinformatics applications that typically have small alphabet sizes: 4 letters for nucleotide base pairs, 20 letters for amino acids, etc. The success of the burst trie in the domain of bioinformatics depends on the determination of an application that requires input data to be both categorized and sorted. If only categorization is needed, to identify uniqueness for example, then a concurrent hash table would

suffice.

A necessary condition for peak scalability and performance on modern hardware is application execution that is aware of the memory hierarchy. There are at least three strategies for designing applications that are aware of the memory hierarchy. The first strategy involves transforming the implementation of the application to increase locality of reference. The second strategy moves the coordination of data access into an application library that can be used by multiple applications. This is the strategy employed by the lock-free skip tree and burst trie implementations. The third strategy moves the coordination of data access into the runtime system. If the application is written in a language that specifies *what* to compute without specifying *how* to compute, there are opportunities for the runtime system to determine the most efficient use of available resources.

Our near-term goals rest on identifying developer libraries that stand to benefit from cache-conscious randomization algorithms. A great deal of previous work has focused on cache-conscious developer libraries for regular computational patterns (loops). PLASMA is a linear algebra library for multicore architectures. MAGMA is a linear algebra library for hybrid x86-based multicores accelerated with GPUs [156]. Less work is available on irregular pointer-based algorithms. Such pointer-based applications rely on recursive data structures such as linked lists, trees, and graphs, where individual nodes are dynamically allocated and nodes are linked together through pointers to form the overall structure.

Our long-term goals seek to transform runtime systems so that the runtime can manipulate input programs to respond to heterogeneous parallel architectures. A new generation of parallel languages provide programming constructs for specifying what a program should produce without explicit instructions on how the computation is performed. However, runtime frameworks are largely designed to accept intermediate-level program representations that have thrown away much

of the high-level information. As the number of cores per processor increases, the relative penalty of devoting an entire core to the runtime optimization framework decreases.

7.1 Future Work: Developer Libraries

The general strategy for increasing spatial locality in pointer-based computational patterns is the enforcement of some type of contiguous layout of data onto an irregular data structure such as a linked list, a tree, or a graph. This strategy can be subverted by the presence of concurrent modifications. Concurrent reads to neighboring locations of memory improve data access by populating the memory hierarchy. However, concurrent writes to neighboring locations increase communication among processors. We have employed a randomization technique to impose an overall structure on the data structure and yet divide the data structure into smaller regions so that concurrent writes are split across the regions. By using a randomized algorithm, concurrent threads do not need to communicate to maintain some type of internal balancing requirement.

One promising area of developer libraries that could benefit from cache-conscious randomized algorithms are graph processing libraries. A number of research projects have been undertaken to perform data processing over acyclic nested data structures [157–159] or cyclic graph-based data structures [160]. Graph algorithms often exhibit poor locality of memory access, very little work per vertex, and a changing degree of parallelism over the course of execution. The distribution of work in these application libraries could benefit from a cache-conscious randomized technique. For example, the Pregel graph library requires a user to specify a custom assignment function from graph vertices to compute machines in order to exploit locality [160]. The graph library could benefit from a dynamic re-partitioning mech-

anism, possibility similar in structure to the hopscotch hashing technique [85]. Spatial locality is preserved in hopscotch hashing by using overlapping neighborhoods of buckets in which an element of one neighborhood can be swapped with an element of another neighborhood in order to create an empty element in a neighborhood that is otherwise full (Figure 2.5).

There is a strong interest in maximizing the efficiency of parallel algorithms that have irregular communication patterns. The Berkeley report on the Landscape of Parallel Computing identifies thirteen broad classes of computational algorithms which the authors believe will be important for science and engineering for at least the next decade [2]. Five of the thirteen classes have irregular communication patterns: Unstructured Grids, Graph Traversal, Backtrack and Branch & Bound, Constructing Graphical Models, and Finite State Machines. The Lonestar benchmarks is a suite of parallel irregular programs developed at the University of Texas at Austin. This benchmark suite consists of Agglomerative Clustering, Barnes-Hut N-Body Simulation, Delaunay Mesh Refinement, Delaunay Triangulation, Focused Communities, and Survey Propagation [161]. Our near-term goals rest on identifying scientific applications with irregular communication patterns that stand to benefit from cache-conscious randomization algorithms.

7.2 Future Work: Application Runtime

Microprocessor architecture appears to be trending towards more complex memory hierarchies and more concurrent computational units. A research challenge for the immediate future is to develop portable, scalable, and efficient techniques for writing applications to run on these architectures. The threads model of programming requires explicit synchronization among communicating threads. A threadless concurrency model provides a simpler programming model than a thread

model and has a potential for portability across different parallel architectures. Examples of programming languages that provide thread-less concurrency models are Cilk [162], X10 [163], Chapel [164], and Fortress [165]. However, these next generation programming languages are implemented on current generation runtime systems. The intermediate representation that is provided to the runtime has eliminated the abstraction of *what* to compute without specifying *how* to compute. In this section, we describe three example microprocessor architectures and then we provide suggestions for developing next generation runtime systems.

The Roadrunner supercomputer at Los Alamos National Laboratory is a good example of an architecture that highlights both trends of more complex memory hierarchies and more concurrent computational units. The supercomputer consists of 12,240 IBM PowerXCell 8i processors and 12,240 AMD Opteron cores [166]. The Opteron cores have a split-level L1 cache and a unified L2 cache. The PowerXCell 8i processors contain one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The PPE has a traditional memory hierarchy of a split-level L1 cache and a unified L2 cache. The SPE can directly access only 256 KB of memory called the local store. Main memory can be accessed by the SPE only through direct memory access transfers to or from the local store. The supercomputer memory hierarchy is a combination of three distinct subsystems: the SPE local store and access to main memory, the PPE access to main memory, and the Opteron conventional memory hierarchy.

The NVIDIA Fermi GPU architecture has an unusual two level cache hierarchy. The architectural model contains streaming multiprocessor units where each streaming multiprocessor is a set of computational processors [167]. Each streaming multiprocessor has 64 KB of configurable shared memory. The shared memory may be configured as 48 KB of scratchpad memory [168] and 16 KB of L1 cache or 16 KB of scratchpad memory and 48 KB of L1 cache. All streaming multiprocessors

share 768 KB of L2 cache. The total size of the registers is larger than the total size of the L1 caches, and the total size of the L1 caches equals the L2 cache size.

The Tilera TILE64 is a system-on-chip multicore architecture [169]. The multiprocessor consists of 64 tile processors that are arranged in an 8 x 8 array. The tiles are connected through a 2D mesh network with I/O interfaces on the periphery. Each processor has a split-level 16KB L1 cache backed by a unified 64 KB L2 cache. The L2 caches can be shared across tiles to provide a virtual 4 MB L3 cache with non-uniform access times. The tiles are connected through five independent networks that support five distinct functions. One network transports data requests from the tiles to the memory controllers. Another network communicates data requests among the L2 caches of each tile. Two of the networks are reserved for explicit user-level communication. The final network handles I/O requests.

The Roadrunner supercomputer, the Fermi GPU, and the TILE64 processor are three examples of the diversity in microprocessor design. All three designs have a large number of concurrent processing units and a heterogeneous memory hierarchy. Several approaches to implementing parallel languages that are programmed for the memory hierarchy rely on compile-time information in order to tune the program to the target architecture [170, 171]. Other approaches can use runtime information to respond to the target architecture. Two notable examples of these runtime systems are the Merge platform [172] and the Ocelot framework [173]. The former uses virtual function calls while the latter uses just-in-time compilation to respond to changes at runtime.

The next generation programming languages are implemented on current generation runtime systems. As the number of cores per processor increases, the relative penalty of devoting an entire core to the runtime optimization framework decreases. We argue that the high-level representation of the program should be available to the runtime system. Explicit parallelism is now ubiquitous in com-

modity hardware. Traditionally, an emphasis has been placed on extracting maximum efficiency from parallel and distributed computing. Modern supercomputer centers are configured with some combination of three distinct computational regimes: isolated memory spaces (message passing, MPI), shared memory spaces (multicore, OpenMP), and hardware accelerators (GPUs, PPUs). In our experience interacting with application scientists, they are most interested in a breadth of parallelism that can be applied towards their diverse research projects. The future will contain a set of rich concurrent runtime systems that take advantage of cheap computation to dynamically restructure the communication patterns of a program. The end result will be more implicit parallelism in the runtime, when previously we had relied on implicit parallelism in the microprocessor, in order to improve application productivity.

Bibliography

[1] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, 2005.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, University of California at Berkeley, 2006.

[3] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing. In *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[4] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, 1991.

[5] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23:20–24, March 1995.

[6] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, April 2004.

[7] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *IEEE 10th International Symposium on Workload Characterization*, 2007.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition edition, 2007.

[9] David A. Patterson. Latency lags bandwidth. *Communications of the ACM*, 47:71–75, 2004.

[10] John L. Manferdell. The many-core inflection point for mass market computer systems. *CTWatch Quarterly*, 3:11–17, 2007.

[11] Michael Spiegel. ConcurrentSkipTreeMap and ConcurrentSkipTreeSet implementations. `https://github.com/mspiegel/lockfreeskiptree`.

[12] Maurice Herlihy. Wait-free synchronization. *ACM TransactIons on Programming Languages and Systems*, 11:124–149, 1991.

[13] Michael Barry Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

[14] Paul F. Reynolds, Jr., Craig Williams, and Wagner R. Raymond, Jr. Isotach networks. *IEEE Transactions on Parallel and Distributed Systems*, 8:337–348, 1997.

[15] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, 2004.

[16] Hagit Attiya and Eshcar Hillel. The power of DCAS: highly-concurrent software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007.

[17] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: denial of service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[18] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In *Proceedings of the 8th international conference on Applied parallel computing*, 2006.

[19] James T. Kuehn and Burton J. Smith. The horizon supercomputing system: Architecture and software. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing (SC 88)*, 1988.

[20] Allan Kennedy Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, 1989.

[21] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[22] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2001.

[23] Chi-Keung Luk and Todd C. Mowry. Compiled-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996.

[24] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, 1999.

[25] Xavier Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Theoretical Informatics and Applications*, 31(3):251–269, 1997.

[26] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[27] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, 1972.

[28] David Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem. *European Journal of Operational Research*, 87:175–187, 1995.

[29] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1985.

[30] Sarita Vikram Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin at Madison, 1993.

[31] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[32] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 9512, Rice University, 1995.

[33] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *Proceedings of the 17th international symposium on Computer Architecture (ISCA '90)*, 1990.

[34] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, 2001.

[35] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th annual ACM symposium on Parallel algorithms and architectures*, 2002.

[36] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[37] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41:1020–1048, 1994.

[38] Nir Shavit and Asaph Zemach. Diffracing trees. *ACM Transactions on Computer Systems*, 14:385–428, 1996.

[39] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transaction on Programming Languages and Systems*, 11(2):249–282, 1989.

[40] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. In *ACM Transactions on Programming Languages and Systems*, volume 12, pages 463–492, July 1990.

[41] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*, 1992.

[42] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, 1993.

[43] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997.

[44] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[45] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2005.

[46] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2007.

[47] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[48] Maurice Herlihy. A methodology for implementing highly concurrent data objects. In *ACM Transactions on Programming Languages and Systems*, 1993.

[49] A. Agarawl and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th annual international symposium on Computer architecture*, 1989.

[50] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[51] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2003.

[52] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of ACM Symposium on Principles of Distributed Computing*, 1996.

[53] Doug Lea. Concurrent skip list map. Approved in JSR 166 for java.util.concurrent in J2SE5.0, 2004.

[54] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25:5, 2007.

[55] James Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[56] Maged M. Michael and Michael L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[57] Steven S. Lumetta and David E. Culler. Managing concurrent access for shared memory active messages. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998.

[58] William Pugh. Skip lists: A probablistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[59] William Pugh. Concurrent maintenance of skip lists. Technical Report UMIACS-TR-90-80, University of Maryland, College Park, 1990.

[60] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.

[61] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: a scalable overlay network with practical locality properties. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, volume 4, 2003.

[62] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):384–393, 2007.

[63] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.

[64] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1438–1445, 2004.

[65] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Proceedings of the 10th International Conference On Principles of Distributed Systems*, 2006.

[66] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, 1995.

[67] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete Algorithms*, 1992.

[68] Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.

[69] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[70] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th annual ACM southeast regional conference*, 2007.

[71] Conrado Martinez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45:288–323, 1998.

[72] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–397, 1999.

[73] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35:341–358, 2005.

[74] Michael A. Bender, Gerth S. Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro Lopez-Ortiz. The cost of cache-oblivious searching. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science*, pages 271–280, 2003.

[75] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991.

[76] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *ACM SIGOPS Operating Systems Review*, 28 (28):252 – 262, December 1994.

[77] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st International Symposium on Memory Management*, 1998.

[78] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architecture Support for Programming Languages and Operating Systems*, 1998.

[79] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

[80] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 SIGMOD Conference*, 2000.

[81] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B+ trees: Optimizing both cache and disk performance. In *Proceedings of the ACM 2002 SIGMOD Conference*, 2002.

[82] Lars Arge, Gerth Stolting Brødal, and Rolf Fagerberg. *Handbook on Data Structures and Applications*, chapter 38 Cache-Oblivious Data Structures. CRC Press, 2004.

[83] Doug Lea. Concurrent hash map. Approved in JSR 166 for java.util.concurrent in J2SE5.0, 2004.

[84] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In *Proceedings of the International Symposium on Distributed Computing*, 2005.

[85] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the International Symposium on Distributed Computing*, 2008.

[86] Nikolas Askitis and Ranjan Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the Thirtieth Australasian Computer Science Conference*, 2007.

[87] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)*, 20:192–223, 2002.

[88] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1960.

[89] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An excercise in cooperation. *Communications of the ACM*, 21:966–975, 1978.

[90] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability: Second Revised Edition*. American Mathematical Society, 1997. Available online under the GNU Free Documentation License.

[91] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Springer-Verlag, 1976.

[92] D.V. Lindley. *Introduction to Probability and Statistics From a Bayesian Viewpoint*, volume 1. Cambridge University Press, 1965.

[93] A. Thavaneswaran. Course notes for 5.305: Probability models. Available online at `http://home.cc.umanitoba.ca/~thavane`., January 2005.

[94] Wojciech Szpankowski and Vernon Rego. Yet another application of a binomial recurrence. Order statistics. *Computing*, 43:401–410, 1990.

[95] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In Giuseppe Prencipe and Shmuel Zaks, editors, *Structural Information and Communication Complexity, 14th International Colloquium*, volume 4474 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2007.

[96] Doug Lea, Joseph Bowbeer, Brian Goetz, David Holmes, and Tim Peierls. Java specification request (JSR) 166: Concurrency utilities. http://jcp.org/en/jsr/detail?id=166, September 2004.

[97] Yehoshua Sagiv. Concurrent operations on B-trees with overtaking. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1985.

[98] Theodore Johnson and Dennis Sasha. The performance of current B-tree algorithms. *ACM Transactions on Database Systems*, 18:51–101, 1993.

[99] Theodore Johnson and Dennis Shasha. A framework for the performance analysis of concurrent B-tree algorithms. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1990.

[100] V. Srinivasan and Michael J. Carey. Performance of B+ tree concurrency control algorithms. *VLDB Journal*, 2:361–406, 1993.

[101] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[102] Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, 2005.

[103] IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. publication no. SA22-7085.

[104] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, 2010.

[105] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[106] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, 2010.

[107] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[108] V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37:1657–1665, 1988.

[109] Vipin Kumar, V. Nageshwara Rao, and K. Ramesh. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conf. on Artificial Intelligence (AAAI-88)*, 1988.

[110] Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the cm-5. *SIAM Journal on Optimization*, 4:794–814, 1994.

[111] Jonathan Eckstein, Cynthia A. Phillips, and William E. Hart. Pico: An object-oriented framework for parallel branch and bound. In *Studies in Computational Mathematics, Volume 8: Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*. Elsevier, 2001.

[112] Kurt Anstreicher, Nathan Brixius, Jean-Pierre Goux, and Jeff Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91:563–588, 2002.

[113] Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[114] Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, 2010.

[115] B. L. Fox, J. K. Lenstra, A. H. G. Rinnooy Kan, and L. E. Schrage. Branching from the largest upper bound: Folklore and facts. *European Journal of Operational Research*, 2:191–194, 1978.

[116] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. Technical report, Chalmers University of Technology and Göteborg University, 2003.

[117] Richard E. Korf. Sliding-tile puzzles and rubik's cube in AI research. *IEEE Intelligent Systems*, 14:8–11, 1999.

[118] G. Kendall, A. Parkes, and K. Spoerer. A survey of np-complete puzzles. *International Computer Games Association Journal*, 31:13–34, 2008.

[119] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2:397–404, 1879.

[120] P.D.A. Schofield. Complete solution of the eight puzzle. *Machine Intelligence*, 3:125–133, 1967.

[121] Richard E. Korf. Depth-first iterative-deepening: An optimal admissable tree search. *Artificial Intelligence*, 27:97–109, 1985.

[122] Othar Hansson, Andrew Mayer, and Moti Young. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63:207–227, 1992.

[123] Richard E. Korf and Larry A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[124] Marek Kubale, editor. *Graph Colorings*. American Mathematical Society, 2004.

[125] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[126] Thomas J. Sager and Shi-Jen Lin. A pruning procedure for exact graph coloring. *ORSA Journal on Computing*, 3:226–230, 1991.

[127] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39:378–406, 1991.

[128] E. C. Sewell. An improved algorithm for exact graph coloring. In *Cliques, coloring and satisfiability: second DIMACS implementation challenge*, pages 359–372, 1996.

[129] D. Costa and A. Hertz. Ants can colour graphs. *The Journal of the Operational Research Society*, 48:295–305, 1997.

[130] Eugène L. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and David B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley-Interscience Publications, 1985.

[131] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook, editors. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[132] Jill Cirasella, David S. Johnson, Lyle A. McGeoch, and Weixiong Zhang. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. In *Proceedings of the Third International Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2001.

[133] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[134] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[135] J. Edmonds. Optimum branchings. *Journal Research of the National Bureau of Standards*, 71B:233–240, 1967.

[136] Frederick Bock. An algorithm to construct a minimum spanning tree in a directed network. In *Developments in Operations Research: Proceedings of the Third Annual Israel Conference on Operations Research*, 1969.

[137] Gerhard Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

[138] Hans Kellerer, Ulrich Pferschy, and David Pisinger, editors. *Knapsack Problems*. Springer, 2004.

[139] Uzi Zahavi, Ariel Felner, Robert Holte, and Jonathan Schaeffer. Dual search in permutation state spaces. In *Proceedings of the 21st national conference on Artificial intelligence*, 2006.

[140] Uzi Zahavi, Ariel Felner, Jonathan Schaeffer, and Nathan Sturtevant. Inconsistent heuristics. In *Proceedings of the 22nd national conference on Artificial intelligence*, 2007.

[141] Cliff Click. Azul's experiences with hardware/software co-design. Keynote presentation at the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), 2009.

[142] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, 2005.

[143] Andreas Kleen. A NUMA API for LINUX. Whitepaper, April 2005. Prepared by SUSE LINUX Products GmbH, a Novell Business.

[144] *The Open Group Base Specifications. IEEE Standard 1003.1*. Institute of Electrical and Electronics Engineers (IEEE) and the Open Group, 2004.

[145] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[146] Rene De La Briandais. File searching using variable length keys. In *AFIPS Joint Computer Conferences*, 1959.

[147] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, second edition edition, 1973.

[148] Arne Andersson and Stefan Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46:295–300, 1993.

[149] Stefan Nilsson and Matti Tikkanen. Implementing a dynamic compressed trie. In *Proceedings of the Second Workshop on Algorithm Engineering (WAE)*, 1998.

[150] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 1997.

[151] J. Clément, P. Flajolet, and B. Vallée. Dynamical sources in information theory: A general analysis of trie structures. *Algorithmica*, 29:307–369, 2000.

[152] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

[153] Phil Bagwell. Ideal hash trees. Technical report, École polytechnique fédérale de Lausanne, 2001.

[154] Alfons Laarman, Jaco van de Pol, and Michael Weber. Boosting multi-core reachability performance with shared hash table. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design*, 2010.

[155] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing*, 70:839–848, 2010.

[156] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009.

[157] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical report, Carnegie Mellon University, 1992.

[158] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[159] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data*, 2009.

[160] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, 2010.

[161] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pages 65 –76, 2009.

[162] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1995.

[163] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005.

[164] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-level Parallel Programming Models and Supportive Environments*, 2004.

[165] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. Fortress language specification 1.0, March 2008.

[166] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*, 2008.

[167] David Patterson. The top 10 innovations in the new NVIDIA fermi architecture, and the top 3 next challenges. White paper, September 2009.

[168] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th interntional symposium on hardware/software codesign*, 2002.

[169] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zoo. TILE64 processor: A 64-core SoC with mesh interconnect. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 08)*, 2008.

[170] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006. ISBN 0-7695-2700-0.

[171] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, 2010.

[172] Michael David Linderman. *A Programming Model and Processor Architecture for Heterogeneous Multicore Computers*. PhD thesis, Stanford University, 2009.

[173] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010.