Creating a Concurrent In-Memory B-Tree Optimized for NUMA Systems

by

Marlon McKenzie

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The size of main memory is becoming larger. With the number of Central Processing Unit (CPU) cores ever increasing in modern systems, with each of them being able to access memory, the organization of memory becomes more important. In multicore systems, there are two main architectures for memory organization with respect to the cores - Symmetric Multi-Processor (SMP) and Non-Uniform Memory Architecture (NUMA).

Prior work has focused on the improvement of the performance of B-Trees in highly concurrent and distributed environments, as well as in memory, for shared-memory multiprocessors. However, little focus has been given to the performance of main memory B-Trees for NUMA systems. This work focuses on improving the performance of B-Trees contained in main memory of NUMA systems by introducing modifications that consider its storage in the physically distributed main memory of the NUMA system. The work in this thesis makes the following contributions to the development of a distributed B-Tree, specifically in a NUMA environment, modified from a B-Tree originally designed for high concurrency:

- It introduces replication of internal nodes of the tree and shows how this can improve its overall performance in a NUMA environment.
- It introduces NUMA-aware locking procedures with the aim of managing contention and exploiting locality of lock requests with reference to previous client operation request locations.
- It introduces changes in the granularity of locking, starting from the original locking of every node to the locking of certain levels of nodes, showing the tradeoff between the granularity of locking and the performance of the tree based on the workload.
- It considers the combination of the different techniques, with the aim of finding the combination which performs well overall for varying read-heavy workloads and number of client threads.

Keywords: B-Tree, in-memory, NUMA, distributed computing, concurrency control, locking, lock-free, replication

Acknowledgements

I would like to thank my supervisors, Professor Wojciech Golab and Professor Mahesh Tripunitara for their guidance in selection of a suitable topic and how to proceed with experiments to support the topic.

Dedication

This is dedicated to my parents, Roland and Marilyn McKenzie, who have always supported me in all my endeavours, and my brother Ronald McKenzie, who, although mostly reserved verbally, always managed to give some form of encouragement through good and bad times.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

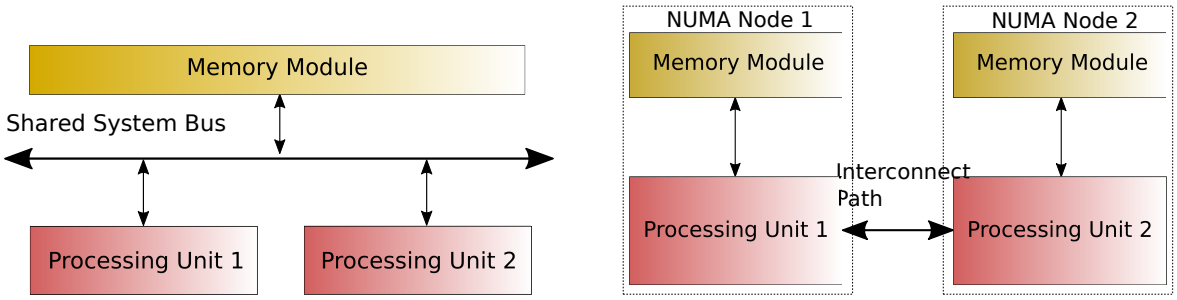
Modern computer systems now incorporate multiple processing cores, allowing for increasing parallelism of operations. The size of main memory in modern systems is becoming larger and less expensive, with servers coming with hundreds of Gigabytes of memory. The larger size of main memory in these systems allows for storage and modification of data sets entirely in main memory, instead of entirely on disk as was previously the only way of storing the data set. Previously, the storage of data sets on slow magnetic disks resulted in the need for data structures that took this slow disk access time into account in its design. One of the focuses of data structures in relation to the shift in its location to main memory is to optimize it for use in main memory. One approach is to design entirely new data structures to organize the use of the data efficiently in main memory, but this is a difficult task. The other approach is to modify existing data structures for main memory use. The B-Tree has been used in many data stores for on-disk storage of data, and this thesis focuses on optimizing it for use in the main memory of a system with varying memory architectures.

A B-tree is a data structure that organizes data in a way that allows for search and update (insertions and deletions) operations in worse-case logarithmic time. It was originally created for single core systems with a memory hierarchy that includes a magnetic hard disk. The B-Tree tries to minimize the number of hard disk accesses, since such accesses are much slower than accesses to main memory. In a B-Tree, records are organized into collections of key/value pairs, called nodes, that are normally the size of a disk block. Each node can have pointers to an associated value, or to child nodes, at a new, lower level. A node that has child nodes is called an internal node. A node at the lowest level that has no child nodes is called a leaf node, and all search operations that arrive at a leaf node also terminate at that node. The root node is an internal node that has no parent, and is the

first node accessed when an operation is started. Within each node, key/value pairs are arranged in ascending order of keys. For internal nodes, keys also serve as separators for child nodes containing ranges of key/value pairs which are not contained within the node. The child node associated with a separator key contains key/value pairs within the range of the previous adjacent key separator key and the associated key. All operations begin with a search operation for a key. A search operation starts from the root node, with a search of the stored key list for the key specified. If the key is found and its associated value is not a child node pointer, its value is returned, otherwise, the search then follows a pointer to a child node that is associated with the first key that is greater than or equal to the current key being searched for. The same procedure done for the root is now done for the child node, and the process continues until the key is found or the key has not been found and the search has reached a leaf node. Inserts and updates are performed similarly to the search operation, except that for inserts, the key might not be found, and is inserted into the correct place of the node that should have it. To maintain a balanced height of nodes, a minimum and maximum number of keys per node is enforced, with merges and splits of nodes done when this key number range is violated. Variations of the B-Tree such as the B^+-Tree [17] and the B^*-Tree [6] add more restrictions for efficiency, such as restricting values to be found only in the leaves, adding extra pointers to certain nodes to aid in the search for keys, and restricting the allowed key space vacancy to be smaller. For more information on the basics of B-Tree structure and operations, the reader is referred to [17] and [18].

When the B-Tree data structure was created, single core, single memory architectures were the systems of the time. Modern systems now incorporate Central Processing Units (CPUs) which have multiple cores and/or processors within one system, and multiple memory modules. With this came the decision of how to interconnect the CPUs to handle cache coherency between the processing units, and how to organize the memory modules with respect to the CPUs. The aim of the organization is to allow for increase in performance through parallelism of execution, while minimizing delays due to communication between CPUs and memory modules. Two main architectures followed from the organization decisions, Symmetric Multi-Processor (SMP) and Non-Uniform Memory Architecture (NUMA).

Fig. 1.1a shows an example of the layout of CPUs and main memory for a SMP system. SMP systems normally consist of a group of identical CPUs, each connected to each other via a common system bus and shared main memory. With this kind of architecture, true concurrent processing can be achieved for as many CPUs that the system has, and memory access times are about the same for each processor. However, since the main memory is a single, shared resource, it can become a bottleneck when being accessed frequently by



(a) Example SMP Architecture

(b) Example NUMA Architecture

Figure 1.1: Examples of the layout of CPUs and main memory modules for SMP and NUMA systems

many different CPUs. The work described in this thesis does not focus on this type of architecture.

Fig. 1.1b shows an example of the layout of the CPUs and main memory for a NUMA system. NUMA systems also consist of a group of CPUs connected to each other and to main memory. However, the connection between processors may not necessarily resemble a single, common bus line. Also, memory modules may be organized such that they may be closer, or local, to one or more CPUs, but need to be accessed across the interconnection network (remotely) by other CPUs. Note that this is different from a Shared Nothing (SN) architecture system, where there is no sharing of storage (for example, memory modules and disk storage) between nodes. In the NUMA architecture, where the data is placed, relative to a CPU that the requesting thread is run on, becomes important. Local memory accesses are memory accesses which can be satisfied entirely by any cache or main memory module directly connected to the CPU on which the executing thread is run. Remote memory accesses require accesses from a main memory module associated with another CPU. Local memory accesses would be satisfied more quickly than remote memory accesses, since remote accesses would require transferring the required main memory data from the remote memory module across the interconnect, and into the cache of the local CPU for the executing thread. Synchronization mechanisms also become an issue, since the state to be kept for a lock can be in any of caches, but need to be used by all cores at some point in execution of operations. This would require this state to be moved to other memory modules and caches, resulting in largers access delays. Local lock accesses would be faster than remote lock accesses, for the same underlying reason of local versus remote memory accesses.

The Operating System creates an abstraction of the physical memory modules as one large virtual memory, with a virtual address space, which is what is used by processes. Each memory allocation request has the potential to be satisfied by one or more of the physical memory modules, based on which CPU the thread is executing on at the time and the amount of main memory space available in each memory module. Allocation of memory and memory binding in Linux by default follows a "first touch" policy – page faults which occur for code running on a CPU are satisfied by the memory module attached to that CPU, so the first CPU to access that page will have that page allocated in its local memory module. Also, each socket's CPU may have many cores and many levels of cache, used to store most frequently used data and instructions. Since there can be many copies of the same page in different caches, a coherence protocol has to be established between all caches, which can introduce interconnect traffic due to coherence messages crossing the interconnect. Interconnect traffic also occurs when operations initiated on one CPU have to access main memory data contained in memory modules associated with other CPUs, since this data has to be pulled into the cache hierarchy of the local processor from the memory of the containing processing unit. Given a required remote memory reference, the higher the interconnect traffic, the less bandwidth there is available to satisfy the request. Less bandwidth for a remote request causes the remote reference access to be slower. Algorithms which consider data locality in memory and more efficient cache usage are an intuitive step to improving the performance of the B-Tree.

Many works consider the use and optimization of B-Trees for use in-memory [59, 16, 42, 10, 43, 53], by considering the cache line size and temporal/spatial locality of memory data in SMP systems. These techniques, while applicable to a NUMA system, do not consider physical distance of memory modules with respect to the CPU executing the operations, and may incur heavy interconnect traffic when the number of clients per socket is large.

Other works have looked at replication in a multicomputer environment [19, 65, 39, 44, 4], where separate processors communicate by message passing across a network. The focus was mainly on smart replication with minimal extra memory usage, that is, knowing just how many nodes to replicate and where to replicate them, while still retaining good performance. In the NUMA system environment, these techniques still apply. However, it is slightly different since all of the memory is still in the same memory address space and the CPUs are all in the same system. Many of the memory abstractions used in the implementations for the multicomputer environment are not required, and the cost, in terms of time, of replica creation and retrieval is much less. If the total memory of the system is very large, there is less need for sophisticated replica placement algorithms, since all of the nodes can be replicated without worry of running out of memory.

Other efforts focused on producing NUMA-aware locking techniques for use in con-

current data structures [57, 50, 21, 14], and NUMA-aware organization of data among worker threads [48] for concurrent data structures. Works which focused on the former techniques assume that many different threads would compete for the lock. The works which focused on NUMA-aware data organization, considered reorganization of operations or the data structure itself for quicker access by requesting threads. The techniques from both categories do not consider how their integration affects the performance of specific data structure operations, such as B-Tree operations).

This thesis considers the replication concepts that were introduced in previous work, and combines it with previous work done on locking mechanisms for B-Trees and in NUMA environments, and techniques for improving throughput of B-Trees in memory. The aim was to find the combination of techniques that delivers the best improvement in throughput of the B-Tree, contained entirely in memory in a NUMA environment, for varying types of read-heavy workloads and number of clients per socket. The contributions of this thesis are:

- It introduces basic replication of internal nodes of the tree and shows how this can improve operation throughput of nodes in a NUMA system.
- It introduces NUMA-aware locking procedures with the aim of managing contention and exploiting locality of lock requests with reference to previous client operation request locations.
- It explores changes in the granularity of locking, starting from the original locking of every node to the locking of certain levels of nodes, investigating the tradeoff between the granularity of locking and the performance of the tree based on the workload.
- It introduces Lock-Free Read Operations and shows how they can improve overall throughput even in the presence of a relatively large number of write operations.
- It considers a combination of the different techniques, with the aim of finding the combination which performs well overall for varying read-heavy workloads and number of clients.

In the body of the thesis, the challenges of introducing replication and modification of the locking used in B-Trees are addressed, and the results of these modifications are discussed.

- Chapter 2 discusses previous work done on efficient node replication, B-Tree locking mechanisms and NUMA-aware locking mechanisms.

- Chapter 3 first describes the functionality of the original B-Tree code used. It then describes the modifications done for this work — node replication, use of NUMA-aware locks and use of locking granularity in a NUMA system.
- Chapter 4 describes the results of experiments run with a various combinations of the modifications made.
- Chapter 5 states the conclusions from the the experiments and indicate areas for further work and improvement.

Chapter 2

Literature Review

Due to the trade-offs between storage speed, cost and size, computers have used storage hierarchies to reduce the average access time of data. Faster memories of smaller sizes were used higher up in the hierarchy, while slower, more expensive storage was used for larger, more long term storage lower down. In this model of storage, complete data sets were previously only able to be stored on hard disks. Hard disks store data on circular platters with a magnetic coating. Data is read or written in blocks by a read/write head attached to a mechanical arm. A location to be read or written is found by determining the proper position on a platter by rotating it and moving the read/write head over the location required, and the location is determined based on instructions provided to the storage device. This physical movement required to read or write data to the hard drive made data access from it very slow when compared to main memory, often the bottleneck for operations on the dataset.

The B-Tree data structure, first discussed by Bayer and McCreight in [5] was originally created for use in single processor, single memory systems, where the aim was to store data such that inserts and updates would require minimal disk accesses. The data structure became the basis of many storage systems, such as relational databases. B-Trees are still being actively used in systems today. However, the underlying architecture has seen changes in the form of the number of processing units and memory modules they provide. Althought these changes increased the overall performance of the systems, it was quickly realized that such systems would require a different approach to algorithm design in order to fully harness the power. Many data structures, if modified appropriately for these new architectures, should see better overall performance than allowed by the original implementation.

The aim of this work is to improve the performance of the B-Tree data structure in such NUMA systems. The improvement is an increase in throughput of operations (where "Throughput" means the number of search and update operations which can be carried out per second). The improvement is achieved by reducing remote memory references by improving locality of the data, and by increasing concurrency of operations within the b-tree by considering Lock-Free and architecture aware techniques.

To improve performance of the B-Tree in a NUMA environment, two areas of the data structure will be considered — (1) location of key/value data relative to the running thread and (2) the locking mechanisms used to both protect the structure of the B-Tree and the data it contains. In this section we shall look at work which has been done on B-Trees to accommodate concurrent and distributed operations. There are two main categories of work which shall be considered - locking techniques for concurrent operations and replication/smart node placement techniques for distributed environments.

2.1 Concurrent Operations in B-Trees

Concurrent operations help obtain maximum throughput of operations for all clients of the B-Tree. Since uncontrolled concurrent update operations by different clients can cause the internal state of the B-Tree or its contents to enter different, and sometimes incorrect states (as seen by retrieval operations), care has to be taken by the data structure to ensure that the correct outcome is achieved at the end of all such update operations. To ensure correctness, update operations are serialized at some point for each operation to allow it to complete or rollback its modifications. This serialization has historically been achieved through locks, but more recently lock-free techniques have also been considered.

2.1.1 Locking Techniques

A lock is a synchronization structure that is used to limit the access of clients to a common resource. It may have rules for limiting such access based on the type of access of the client (reading versus updating) and the number of access types allowed (mutual-exclusion versus some number of concurrent clients). As stated by Graefe in [30, 65], in the case of a B-Tree, maintenance of consistency through lock-based protection can be split into two main categories, protection of the datastructure itself and protection of the data stored by the datastructure. Data structure protection is normally required for only a short time, and is provided by latches. This "lightweight" protection requires mutual exclusion with minimal

overhead, and is normally implemented by the simplest of locks, such as spin locks. Data structure content protection protects the key/value pairs from concurrent client transaction operations, for which concurrent reads are allowed, and are provided by more complex locks, such as Reader/Writer locks. Various studies have analyzed the relative performance of the different types of locks. One of the most comprehensive studies is by David *et al.* in [20]. The results of the study suggested the use of simplest type locks where they are suitable for protection, and in NUMA systems, keeping the lock variable information local for as long as possible. Locks applied on the B-Tree alone do not solve all concurrency problems. Concurrency of operations can cause observable inconsistencies when different nodes are being modified by different operations, and one node (from a retrieval operation) requires another node (modified by an update operation) in its traversal. Concurrent operations can also cause issues with following child pointers, if these pointers no longer point to valid information. To ensure that the transition to the child node is successful, Latch Coupling is used. Latch Coupling involves the holding of the latch on the parent node until a valid pointer on the desired child node is obtained.

Another issue is with splits/merges of nodes when and update operation violates the capacity rule for the number of keys allowed in a node. A split may necessitate one or more new nodes to be created and introduced into the tree and a merge may require one or more nodes to be removed from the tree. Any operation which arrives at nodes which are being split may have originally found that the key being searched for was in the node but is no longer there when searching through the node. Lehman and Yao in [47] suggested their modification to a B-Tree, called a B^{link} Tree, which uses the concept of high fence keys and a new right node pointer, to be used with new nodes that are created when the current node is being split. All new nodes that are created are always created to the right of the older node. In this way, if an operation has found that it has reached a node that it original thought had the desired key but now doesn't, it should be able to follow one or more right node pointers (in the case of multiple, adjacent node splits) to find the desired node with the key.

Key Range and Granularity of Locking

Apart from locking a single key-value pair, locks in the B-Tree can also be used to lock a set, or range, of key-value pairs in the tree. Such key range or value range locking is a form of logical locking, as stated by Graefe in [30], which attempts to lock a range of keys and/or values, based on some common ground, such as being in the same subtree of the associated separator key. It is a form of predicate locking, as described by [24, 41], which describes logical locking of a group of objects that satisfy some predicate, the predicate

being the common ground, such as being of the same subtree. When applied to the logical contents of the B-Tree, a lock can protect from as little as one key up to a broad range of key and value pairs, including the gaps across nodes. Key range locking including gaps can come in the form of locking the range of key-value pairs in the child node from the key after the previous key in the node being searched up to and including the current key in the node being searched, or the current key in the node being searched up to the key in the child node before the next key in the current node. This type of locking can improve the performance of the tree, depending on the workload. In the case of read only workloads, it would require fewer locks to be acquired before obtaining the required value. Workloads with conflicting operations which require the same locks would see worse performance, since there is a higher probability that the lock being acquired may be already acquired for some other operation on a key or value of the same group.

2.1.2 Lock-Free Techniques

One technique that has been suggested to improve concurrency is to use Lock-Free techniques to reduce or remove the use of locking in datastructures used for retrieval. As described by Fraser in [26], a Lock-Free system is one which is guaranteed to have system-wide progress (that is, progress in at least one thread of execution). In addition, the term Lock-Free is also used to describe algorithms which do not use mutually exclusive, blocking algorithms to achieve deadlock freedom. Since locks are not used, other techniques must be used to achieve synchronization of the different threads operating on the datastructure.

One method is the use of the Compare-And-Swap (CAS) primitive to make atomic changes to datastructures. The idea is to make sure that a change to or read from a pointer is done atomically, to ensure that a thread executing that operation sees a consistent view of the change — either its value before modification or after. Any inconsistencies found from the use of this operation would cause the operation to retry or fail, depending on the requirements of the datastructure. Some works also provide an extension of the original CAS primitive for their purposes [26, 27]. Another technique involves Software Transactional Memory (STM) which has various implementations, including those of Shavit and Tuoitou [61], Fraser [26], Fraser and Harris [27], and Herlihy and Moss [35, 36]. STM creates an abstraction which groups related operations, giving each thread of operation the impression of having its own private memory space. Since the thread is given the memory abstraction of its own non-shared memory, it does not have to perform explicit locking. For any activity that requires updates, a commit of all the operations of the transaction is validated and done at the end of the transaction by the STM implementation. In the case of an operation A being blocked due to a conflicting operation B, the technique of

Recursive Helping is used to allow operation A to progress. In Recursive Helping [26], operation A recursively calls operation B to ensure that the task of B is done, since one call will succeed while the other will detect that the operation is already being completed and return without duplicating the modification. Some works use flag variables [8] to represent the current state of the node, simulating a lock but without the overhead of blocking the executing thread and switching to another, if the state represents a conflict with the operation being executed by the thread.

With these techniques, various Lock-Free datastructures have been created [11, 64, 26, 62, 64, 23, 13, 12, 25, 31, 34, 60]. Some works focus on providing a datastructures using Lock-Free primitives. Bender et. al. in [8] supplies extra information with the operation status so that operations which encounter any indication of the operation can help the operation to complete and eventually progress to its own completion. Other works [33] use the technique of reading, copying, updating and replacing [55] to make modifications on a copy of the data before swapping it with the oudated version.

Lehman and Yao in [47] describe their B-link Tree, which was a modification of the B-Tree to improve concurrent operation performance through changes of the locking procedure. Searches were changed to be done without locks, while Updates (Insertions) were changed to only perform exclusive locking of the node that is to be modified when the update is performed. To find a key which was moved from node to an adjacent node when a concurrent insert on the same node was being performed, the node was modified to have two features, a fence key and a right node pointer. The right pointer is set to point to the newly created node from a split of a node. On each level, every node from left to right has a right pointer to its adjacent node, created from a split, and so the level itself forms a linked list of nodes. The fence key represented the highest key value contained in the node, and was checked and updated if necessary by every insert operation done on the node. Search operations consult the upper fence key as soon as the node is obtained. A search operation originally chooses a child node based on the separator key as in the original B-Tree, but if a concurrent split caused the higher half of values to be placed in the adjacent node, the fence key of the old node would be updated to be the equal to the highest value in the lower half of keys. The search operation would follow one or more right pointers until it found a node with a fence key greter than or equal to the key being searched for, and would search that node for the key. Splits of nodes would create two new fence keys, and the new fence key of the old node is inserted as the new separator key into the parent node, recursively up to the root as necessary for each split that is necessary. They did not explicilty consider deletes or merges in their work.

Fraser in [26] gives an implementation of a Binary Search Tree (BST) using an extension of CAS for replacement operations and an implementation of Red-Black Tree using

an implementation of STM called FSTM. His extension, Multi-Word Compare-And-Swap (MCAS) provides the same functionality as CAS, but for multiple memory locations, which is important for relocation of objects within a datastructure, which require a removal from one pointer and an insertion into another, all done atomically to ensure consistent views among concurrent threads. To ensure that searches for contained keys always return correct results in the presences of node relocations (due to deletion of other nodes), Fraser based his design of the tree on the Threaded B-Tree, as proposed by Perlis and Thornton in [56]. The Threaded B-Tree uses thread links in the place of empty subtrees, where thread links are pointers which go to the immediate parent and/or successor of the node. If the node is a leaf node, the successor is considered to be the root. Fraser also gives an implementation of a Red-Black tree using FSTM, which creates a transaction for each operation, restarting the operation if the commit done at the end of the operation fails.

Ellen *et al.* in [23] also describe an implementation of a non-blocking, linearizable Binary Search Tree that uses only the basic CAS mechanism, along with reads and writes. To allow for the single word CAS to be used, their tree is leaf oriented, requiring that only one pointer operation to make the structural change to the new tree. An additional flag variable per node handles peculiarities with concurrent conflicting operations. When a node is to be deleted, its state is changed to *mark* so none of its pointers can be changed by other operations. The node for which the pointer is to be changed is marked with a flag representative of the operation (*iflag* for insertion, *dflag* for deletion) to prevent a concurrent conflicting operation from changing that same node's pointers concurrently and creating an inconsistent state. When the pointer is being changed, its state is changed to *ichild* or *dchild* respectively. When the operation completes, the node is marked with uiflag, udflag, and then clean to allow any operation after. Also, in addition to the state, the same node variable holds a pointer to an info record, which contains information about the operation being done, so that other concurrent, conflicting operations can help that operation to complete to ensure eventual progress.

Sewal *et al.* in [60] consider making a B^+-Tree "Latch-Free" by considering a group of queries at a time, and redistributing the work accordingly, so that conflicting operations are handled by the reorganization and execution of the operations, without the need for explicit locking. Reads, which do not conflict with each other, are organized to be done first and in parallel. Updates are organized are partitioned among threads so that each thread works on a specific set of tree nodes, and to ensure that serializability is still kept for the group of operations. The modifications are done at each level from the leaves up to the root, which is modified by only one thread. Locking for modifications is implicitly done by the fact that one thread is allowed to modify only its share of nodes, giving it "ownership over those nodes".

Bronson *et al.* in [12] describe an implementation of a Binary Search Tree using optimistic concurrency control techniques, based on an implementation of an External AVL Tree. To track changes occuring in a node, each node has a version number, which consists of 64 bits, of which one bit indicates that an update of the node is occuring and other bits indicate growth and shrinking of a subtree under a node due to it moving positions and whether or not the node is still contained in the tree. Traversals for all operations use hand-over-hand version reads, where the current node and child's version number are read in, the child version's number is validated as unchanged, and the parent's version number is validated as unchanged afterwards, to view the traversal of the link from the parent to the child as valid. Updates lock the parent of the node to be modified, adding a child if no node for the key is present, or updating the value for the key if it is present. Deletes use the routing node feature of external B-Trees only when the node to be deleted has two children, so that traversals can still continue through to children of the removed node. Nodes with one or no children are immediately unlinked.

Braginsky and Petrank in [11] implement a Lock-Free, balanced B^+-Tree using the basic CAS operation and a method for ensuring a contiguous run of memory for a node, called chunking. The chunk consists of a linked-list of array entries, for which they impose a minimum and maximum number of entries, as well as additional pointers for use with splitting and merging when the minimum or maximum size constraint is violated. Since key and value changes must be atomic, only CAS operations are used. The chunk, and hence the node size is chosen to be a factor of the cache size, to take advantage of cache prefetching. Operations take place in the same way for all operations, except that the choice of a node to merge with is given by the structure of the tree, and is a call on the tree itself, and operations can help others to complete (Recursive helping), if the operation being helped is preventing the current operation from completing.

2.2 NUMA-Aware Techniques

2.2.1 Locking

A basic lock is represented by a value in a memory location. Locks assume that all clients which use the lock have access to the location. In SMP, there is a single memory module, and memory distance/access time is symmetric from all processing cores across a shared bus. Most works initially were concerned about keeping good performance in the cases of both low and high contention. Various categories of locks have been created, from those based simply on spinning [3], to those based on queues [54, 51], and those based on reducing

request frequencies by some appropriate amount [2]. However, in NUMA systems, each socket has its own local memory module, and an interconnect exists between the cores to allow for access of any remote memory module from any core. The operating system creates an abstraction of these memory modules as a single, contiguous memory pool, but actual access time is dependent on the executing code's location and the location of the required memory address. If the memory address is in the memory of the local NUMA node, access is quick, since it requires no messaging across the interconnect. However, if the memory address is in a remote memory module, then messages must be sent across the interconnect to pull the information into the local memory module. Updates cause invalidation messages to be sent to all remote locations which contain the information in the cache. As stated by David *et al.* in [20], in such an environment, local memory accesses minimize memory access time. Locks however, may be used by code on any of the processing cores, so remote memory access cannot be avoided, only kept to a minimum.

Radovic and Hagersten in [57] describe a NUMA-aware Hierarchical Backoff lock. The lock aimed to reduce remote accesses by "handing over" locks to local threads waiting to acquire them before remote threads, and to reduce contention at lock handover time. The lock itself is represented as an integer value in a shared memory location, initialized to a value, `FREE`, indicating that the lock is not held. Acquisition of the lock has the thread write its NUMA `node id` to the shared lock location. Threads which have been denied the lock are given a backoff time to wait until retrying its request. To account for node locality, threads which read the same `node id` as their own are given a smaller backoff time in comparison to those which do not, to give the local threads a higher chance of obtaining the lock. To account for threads which may not have the same `node id` but may be on the same chip (and still considered neighbours), the address of the lock is copied to a local node specific storage area and neighbours instead spin on that before attempting to access the globally shared lock variable. Starvation is also taken into account by having a threshold number of lock denials and reducing backoff time and writing values to the local node specific storage of other nodes to increase their backoffs.

Luchango *et al.* in [50] describe a Hierarchical version of the CLH lock [51], which extends the original algorithm to be NUMA-aware. To facilitate NUMA-awareness, they extended the concept of the thread waiting queue to be of two parts, one global queue overall, and local queues for each cluster (NUMA node). Thread nodes contained in the global queue wait to acquire the lock directly. The head thread of the global queue, when dequeued, is allowed to enter the critical section. The head thread of the local queue becomes *cluster master*, and is responsible for splicing the members of its local queue into the global queue. To support the distribution of queues, the node (*qNode*) word member variable was modified to contain two other components in addition to the *successor_must_wait* boolean

flag – a *cluster_id* and *tail_when_spliced* flag. When spinning on the predecessor's member status a thread can tell whether it is in the local or global queue based on what flag it is spinning on. It first waits to become cluster master (at which point its *tail_when_spliced* flag becomes true), then it waits for acquisition of the lock (at which point the original algorithm's variable, *successor_must_wait*, is false). A thread releases the lock by setting its *successor_must_wait* flag value to false. The algorithm gives preference to handing off the lock to local threads as much as possible, since a local cluster master attempts to splice in its entire local queue onto the global queue. Starvation was not explicitly considered, but was implicitly assumed not to be a problem since any local queue for a cluster will eventually be spliced into the global queue.

Dave Dice *et al.* in [21] improve upon the idea provided by [50] by the use of the Flat Combining technique in [32] in combination with the MCS queue lock of [54]. Their technique mostly agress with the HCLH algorithm in terms of using a queue based lock as the basis and having local (*FCQueue*) and global (*GlobalQueue*) queues, which uses the notion of clustering nodes and having a cluster id. However, they identified two problems in the HCLH algorithm – relatively small local queues and more potential memory access traffic generated from accessing information from predecessors. To deal with the first problem, they included in the acquiring algorithm the notion of a combiner thread, which has the job of creating a local MCS queue out of potential nodes of threads which want to access the lock (kept in a publication list) and splicing this list into the global MCS queue. To deal with the second problem, they used an MCS queue instead of the CLH queue for the basis of their work, since threads spin on their local flag values instead of their predecessor's flag values. In acquiring the lock, the thread first tries to become a combiner by trying to acquire the local *FCLock*. If the thread does succeed in becoming a combiner, it performs the previously described logic of creation of the local MCS queue, reactivates itself in the publication list, releases the *FCLock* and restarts the lock acquire operation. If it fails (another thread is a combiner), it first spins on *requestReady*. If *requestReady* is true, it continues spinning, adding itself to the publication list if it was removed. When *requestReady* becomes false, the thread goes on to spinning on *isOwner* for the actual lock acquisition, waiting for it to be set by its predecessor thread. *isOwner* being set as true for a thread indicates that the thread has acquired the lock. A thread releases the lock by setting the *isOwner* flag of the next node in the global list (for the successor thread) to false. A check for if the node could potentially be the tail is done to prevent the extra potential remote access from setting the global tail to null in the original MCS algorithm.

Dice *et al.* in [22] describe Lock Cohorting, their technique of building a NUMA-aware lock from more basic locks. The idea is to have one Global, NUMA-oblivious lock and a set of NUMA-aware locks, one for each NUMA node. The lock has three states, Locked, Global

Release, Local Release. To obtain the lock, a thread first checks the state of the local lock variable. If it is Locked, then it will wait to obtain the local lock using the technique of the local lock type. If it is Local Release, then it knows that a local thread had the lock before, the global lock is still Locked, and it only acquires the local lock variable. If it is Global Release it must try to acquire the local then the global lock, to continue. When releasing the lock, a thread checks to see if there are other threads blocked on the local lock. If there are, it does a local release, else, it does a global release. Starvation is addressed using a threshold count and forcing a thread to do a global release for other nodes to have a chance, when the number of local threads have already acquired the lock, is equal to the threshold count.

Calciu *et al.* in [14] build upon the Cohort lock by making a reader/writer variant. They aim to expand Cohort Locks to conform to the multiple reader/single writer access model of a regular reader/writer lock, but also increasing throughput for NUMA architectures using the cohorting mechanism and increasing the average response time by clustering writer requests from the same node whenever possible, which would in turn allow reader requests to be batched as well, happening in preference to writers. Readers update local read indicator count variables to access the lock while Writers check these variables on all nodes to determine if the lock can be accessed exclusively. Writers preferentially hand off the exclusive lock to other writers on the same node using the same technique of Lock Cohorting. Neutral Preference, Reader Preference and Writer Preference versions of the lock are described. The Neutral Preference performs similarly to the PThread Reader/Writer lock, by allowing multiple simultaneous readers but having exclusive writers. The other preferences build on this to provide faster lock acquisition for readers or writers. The Reader Preference allows for readers to more quickly obtain the lock by avoiding locking the shared global Cohort lock but being able to check it locally. The writers always try to acquire the global Cohort lock directly. If there are readers present, the writer unlocks the global cohort lock and waits for the readers to finish, then tries to reacquire it. Starvation of Writer threads is avoided by only allowing a maximum (threshold) number of Reader thread acquisitions before allowing Writers to acquire the lock. The Writer Preference allows for writers to have preference over the readers by making readers increment the reader counter before checking the cohort lock status, and immediately decrementing the reader counter if the global cohort lock is locked and waiting for writers to finish. Starvation is also prevented using a similar barrier strategy to the reader preference case.

2.2.2 Data Access

Besides the lock structure, other aspects of the data structure can be modified to improve the performance of the data structure in a NUMA system.

Li *et al.* in [48] improve the performance of datastructures in a NUMA system by organizing the accesses to data among threads to reduce interconnect bottlenecks due to overuse of some of the connections by shuffling the data among threads which produce and consume common data. For N threads distributed evenly among cores on S sockets, the data for each thread is broken into N partitions. By considering each thread's thread number (t_i) and socket number (s_i), the accesses to data by each thread are organized so that each thread obtains all of the partitions it needs to complete its operation, while the overall set of operations utilize each interconnect connection path almost uniformly.

2.3 Distributed B-Trees

The need for very large amounts of storage and processing power has become too much for single machine processing to handle. Distributed and parallel computing has become a very important paradigm in satisfying such needs. The distribution can be on the level of multiple cores in different machines or multiple cores within the same machine. The distribution of resources for storage requires the data being stored to be distributed as well. However, this has resulted in new issues as it relates to the relationship between the logical and physical representation of data, and the placement of data to support retrieval and storage requirements of clients. With distribution there can also be a choice of where operation requests will be directed to and a protocol for communicating with all machines which are participating in the storage of the data. When the fundamental components of communication protocol and storage are being considered, performance becomes important. Due to distribution, the physical components of the storage can be varying distances apart, being as close as separate memory modules in the same machine as in the NUMA architectures, and as far apart as separate storage devices (disks and/or memory) in separate machines, geographically far from other. One of the normal requirements of a client of the datastore is that the data requested or being stored be done so as quickly as possible.

2.3.1 Node Replication and Placement

One of the ways to obtain better operation performance is to keep the data being operated on as close to the client as possible. Since clients can be located anywhere in relation to

the data this would require a copy of the data to be stored locally at every possible access point for the datastore. Unreliability of the network can cause the communication between storage nodes to fail, resulting in unavailability of the required data. Replication of the data can help to combat this, since having multiple copies of the data can maximize the chances of the data being available to a client upon request from any of the access point to the datastore. However, the replication of data comes with other problems. Once such problem is that such replication of data may be contrary to original resource management requirements – one wants to add more resources to improve performance, but in an efficient manner, and if efficient use of storage space (a resource) is a requirement, then this duplication of the information stored actually works against the requirement. There must therefore be a tradeoff between this improvement in operation performance and increase in storage space usage. Another problem is that with duplication of the data comes the requirement to maintain consistency of the data stored among the many copies that exist when updates occur. The level of consistency required is dictated also by client requirements and can be very strong (all updates must be reflected in all copies before the operation returns to the client), or varying levels of weak (clients can read varying data depending on where it was retrieved from). Other classifications of consistency consider the time it takes for all copies to receive updates from an updated copy (Eventual Consistency) and the sequence in which consecutive conflicting operations occur (Sequential Consistency, Serializability and Linearizability [37]). Many methods of measuresurement and benchmarks have also been developed to try to quantify the various levels of consistency [9, 58, 29]. Also, replication used to combat network partition has a problem – the data accessible due to the partition may not always be the most updated version. How up-to-date this information being retrieved from the partition is depends on the consistency model used, and there are principles such as Brewer's *Consistency-Availiability-Partition (CAP)* principle [28] and Abadi's modification of the principle, called *Partition - Availiability/Consistency, Else - Latency/Consistency (PACELC)* [1] which elaborate on the tradeoffs of performance in the presence/absence of this network unreliability. Good replication algorithms therefore take all of these tradeoffs into account, as well as the client requirements in order to provide an acceptable level of performance from the perspective of the client, while still making efficient use of resources.

Wang in [65] describes how replication can improve the performance of the implementations of the memory models of coherent shared memory and multi-version memory. The concept of the replication factor is introduced and used to describe the number of copies of a node to be made on a processor. Number of copies of each node are chosen based on the height of the node in the tree, with the guiding reasoning being the higher the node is in the tree, the more chance it has to be accessed (as a path to lower nodes in the tree) and

the higher its copy count should be, with the minimum being one copy, and the maximum being the number of processors that can access the tree. By this reasoning, the number of copies of the root was set to be equal to the total number of accessing processors, and the replication factor of all leaves was set to be one. The number of copies of each internal node was determined to be the minimum of the total number of copies of the root and the number of copies of the nodes in the level above the node multiplied by a constant factor, called the *Replication Factor*.

Johnson and Colbrook in [39, 40] describe the dB-Tree, which uses a B^{link} Tree as a base and replication to improve its performance. The focus of the replication is on the inner nodes in relation to the leaf nodes that are contained in the memory associated with the processor, so that for every leaf node in that processor's memory, there is a complete path from the root node to that leaf node in that processor's memory. This would ensure that a read operation to a key contained on a specific processor would not have to leave that processor once it has arrived at it. The distribution of leaves are tracked by *extents*, which are the group of leaves contained by a processor. The group of leaves for an extent is selected so that the leaves are neighbours in the logical structure of the tree. Using the collection of processor extents, the number of leaves are balanced as evenly as possible among the processors. If one processor has too many leaves in its extent, excess leaves from that processor are sent to another processor with an extent of neighbouring leaves, which can accept more leaves for its extent.

Krishna and Johnson in [44] also describe the use of replication and load balancing to manage the performance of a B^{link}-Tree in a Distributed environment. Nodes are identified with a logical pointer that is unique among all processors. The internal nodes are replicated based on a strategy called "Path to Root". For every leaf node contained on a processor, all of the nodes along the path to that node have a local copy on that processor. One copy of a node serves as the primary copy, and it initiates all update synchronization messages to the other copies when an update operation occurs. Leaf nodes are not replicated, but are allowed to migrate to more lightly used processors to reduce the load. Nodes are grouped on each processor with their siblings as much as possible to reduce communication among nodes. Nodes also contain pointers to all their siblings, parents and children, which are used for regular traversal on one node, or to redirect the traversal to another node, if the leaf node being searched for is not contained on the current node.

Cosway in [19] also suggested improvements to a distributed B^{link} Tree using static and dynamic techniques for replication. The observations for static replication were that the frequency of access of nodes was dependent on the level of the node in the tree, withe root node always being accessed, and all other nodes in increasingly lower levels being accessed less frequently due to the route taken by the operation being focused on

a specific node of the level. From this two rules were developed, one for basic replication of frequently accessed nodes (with access above a certain threshold), and another for load balancing capacity, with the capacity being probabilistically defined based on the number of processing cores, the number of copies required per node and the number of nodes on the level. The latter rule aimed to find the number of copies required for an acceptable capacity, based on space requirements. A hybrid of the two rules was used to determine how many node replicas to create. Each replica was placed uniformly at random on one of the processors that did not already contain a copy of the node. The dynamic version built on top of the static replication by giving the nodes the ability to keep track of their access frequencies and to request replication if necessary. Each node was made to keep track of access counts and previous access time. On every access, the time difference between the current access time and the last access was found. If the time difference was greater than a threshold time, the access count was incremented, else it was decremented. After a certain observed time, if the frequency count was above a threshold, the node would send a replication request and number (based on the hybrid technique of the static replication technique) to the processor holding the master copy. Each processor has a fixed size cache for replicas, and uses cache replacement algorithms for replicas if it is full, sending the evicted replica back to the processor holding the original copy.

Asaduzzaman and Bochmann in [4] describe the construction of a replicated B-Tree with a focus on high throughput and elimination of bottlenecks, when operations on the tree can be initiated from any of the participating storage processors. Similar to Krishna and Johnson in [44], they realized the need for replication of nodes to increase operation throughput. A similar decision to replicate internal nodes was made, by distributing leaf nodes among participating processors, and replicating only the internal nodes on the path to each leaf node on the processor. The tracking of the replication was done differently. Each node had an associated Routing Table and a Backward Pointer table. The Routing table maintained information for each processor known to contain nodes for each child node range in the tree. The Backward Pointer table maintained a pointer back to all processors which are known to contain a local version of the same node. It was noted that the replication of internal became a bottleneck for strongly consistent updates as the number of participating processors increased. To address this, the consistency requirements were relaxed, and synchronizing updates were treated as separate, atomic updates, which were performed after the update on the local processor was completed. Optimistic concurrency control in the form of a version number, checked before and after operations and incremented after update operations, maximized concurrency.

2.3.2 Multiversioning

When replication is used in a distributed datastore, multiple physical copies of the same logical data may exist and the issue of consistency arises. Different clients may have different consistency requirements, and may be tolerant to having slightly stale versions of the data, but have strict constraints on operation response time. With these types of constraints the technique of management and serving of multiple versions of the data becomes acceptable.

Lomet and Salzberg in [49] describe their use of multiversioning in a B-Tree, which they called the Time-Split B-Tree. The idea was to split the content stored by the tree into a current version, which was stored on multiple-write secondary storage (such as a regular hard disk), and to push historical versions of the data onto another, possibly write-once, secondary storage device (such as an optical disk). The versioning was done on the key level, with each key having a timestamp indicating its insertion time, and with updates to the same key represented as inserting a new version of the key into the node with a more current timestamp. The splitting of the node was always done on the mutable storage holding the current version of the data before pushing the historical content to the archiving device. This was done to allow for the current versions of nodes to accept current changes (by keeping it on the multiple-write storage) and to make the most efficient use of the archival storage device (which could possibly be write-once), by writing a full node of past key value content to it. A choice between splitting by time or by key could be done for all nodes, with the choice being made by minimizing a cost function which includes total space and version space requirements. Split by a key value works as in the regular B-Tree case, with keys lower than the split key being put into the left node and all keys larger than or equal to the split key being put into the new right node. Split based on a time would put all versions of a key with a timestamp less than the split value into the historical node (for archiving) and the most current versions of keys, as well as versions greater than the split value into the current node (and kept on the multiple-write disk).

Weihl and Wang in [66] discussed an implementation of multiversion memory for B-Trees (their change being based on the B^{link} Tree), as well as a general transformation of current dictionary based algorithms to support the multiversion memory, in a distributed environment. Each node of storage was assigned a counter, the version number, which starts at an initial version and is incremeted for each update operation performed. The tree itself has a copy of the most current data for each node, and a fixed-size cache of previous versions for various different nodes. Readers perform the following steps in order – *read_pin* and *read_unpin*. *read_pin* searches the cache for a copy of the requested node, creating a locally cached version if necessary. The reader then sets this cached copy to

pinned to prevent it from being removed from the local cache, and reads the information. *read_unpinned* sets the node as *unpinned*. Readers can (and must) request any cached version, including the current version. Writers perform the following operations which correspond to the relevant reader operations – *write_pin* and *write_unpin*. *write_pin* performs the same operation as *read_pin*, except that it changes the local copy to be the current one and attempts to acquire the lock on the node's mutex. *write_unpin*, in addition to what is done for *read_pin*, writes the updated copy back to the base copy (with all necessary communication for notification of updates), and unlocks the mutex.

Becker *et al.* in [7] describe a general algorithm to convert a single-version B-Tree to a multiversion B-Tree for which the resulting regular operations (insert, delete and search) are asymptotically comparable to the single-version operations in terms of time and space. The tree itself is partitioned into differerent version ranges, which are each their own tree, with their own root. Search operations are directed by both a key and a required version, while update operations are restricted to only the most current version of the tree. Versioning is done at the key/value entry level, where a record would be represented by the key, insertion version number, deletion version number and the value (data for a leaf node and a pointer to a child node if an internal node). To handle structural changes caused by updates (splits and merges), two satisifactory conditions were defined, both based on the number of entries a Node block could hold, after the space occupied by its accounting structure is considered (defined as the block capacity b) – *Weak versioning* and *Strong versioning*. These two conditions guaranteed a minimum/maximum number of entries contained in a node and a minimum number of operations which would occur until the next structural change. Node overflow is always initially handled by a current version split. If this initial split results in a violation of any of the two conditions, another restructuring operation restores the conditions. In the case of an underflow, merging with another sibling is done, and in the case of an overflow, a key split is done.

Twigg *et al.* in [63] describe their Stratified B-Tree as a replacement of the Copy-on-Write (CoW) versioned tree. The aim of the Stratified B-Tree with the aim of reducing the space usage of the entire structure by reducing query performance. The tree is represented as a collection of arrays of (key-version-value) tuples, arranged into levels. Arrays hold keys with an assigned range of version numbers, and arrays at one level are roughly twice as large as those at the level below it. All arrays at a certain level contain disjoint sets of versions, and merges are used to correct overlaps. Any array that is too large for its level may have parts promoted to higher levels. Array density is the condition which manages storage within the array. Every version contained in an array must have a subset of all keys contained in that array. Manipulation of the fraction allows for adjustment of space and performance characteristics of the tree. Inserts are buffered in memory and then flushed

to an array of the lowest level. Deletes are handled like updates, except that tombstone elements are used, which are removed when merges occur. Lookups query a Bloom Filter on each array with the required version and then look into those arrays which satisfy the filter.

2.4 In-Memory Indexes

The size of main memory contained in machines have been increasing steadily over the years, and there are now machines with total main memory size comparable to the size of secondary storage drives. Because of this, more of the data that is needed to be accessed can be kept in main memory, with less need to access long-term storage drives for current data and instead using them for periodic checkpointing. As a result, various datastructures that have existed before the modern shift have been revised and updated and new datastructures have been proposed for use with the data now being accessed and maintained mainly in memory.

Rao and Ross in [59] describe their efforts to make the B^+-Tree more efficient in memory and cache, in their CSB^+-Tree. They realized from previous work that B-Trees performed very well in memory when the size of the node is equal to the size of the cache line, but a lot of the space of a node was still used for keeping track of the structure of the tree with child pointers, rather that keeping track of data. To reduce this overhead, adjacent sibling nodes on each level were grouped and stored in a contiguous segment of memory, and the starting address of the group was stored as a child pointer in the parent node. Traversal operations proceeded as before, with the required node address being calculated as an offset from the starting address of the group known to contain the child node. Storing only one pointer to the start of the group allows nodes to have more space for data and enables the use of a cache line for data, instead of structure overhead. Updates tried to use adjacent, non-used nodes for splits/merges. Splits causing the number of nodes in a group to be greater than allowed in the leaf level caused reallocation of a new group of size larger than the previous by one node, the copying of all of the information of the previous group into the new one, and updating of the parent pointer if necessary. Similar splits in internal nodes cause allocation of another group, redistribution of contained nodes between the groups and updating of the corresponding child pointers.

Chen *et al.* in [16] describe their Fractal-Prefetching B^+-Tree, which aimed to optimize the performance for the tree for both disk I/O and cache operations, by building upon the their prefetching techniques for nodes and pages[15], used to improve search and range-scan operation performance. The prefetching technique uses a node size that is a multiple of

the cache line size, with all cache lines associated with a node being prefetched before the node is accessed. Range scans are performed by finding the node for the starting key using the regular search operation and the other nodes, prefetched sufficiently before needed, are found by following sibling pointers maintained between nodes which are parents of leaf nodes. To further include optimizations for disk operations, the tree is considered in two parts. The tree is organized into a "tree of trees", organized according to disk optimization first or cache optimization first. *Disk-First* organization arranges the tree first according to the regular disk page size, then within each page, the keys are organized into an internal, cache-optimized tree. *Cache-First* organization optimizes the structure of the tree for the cache, then groups an integral number of nodes into a disk page for storage on the disk. Nodes are selected for a group to enhance locality, by grouping neighbouring leaf and parent nodes as much as possible.

Kim *et al.* in [42] considers the optimization of Binary Trees in memory by taking into account processors which include Single Instruction Multiple Data (SIMD) instructions, as well as the other memory characteristics (page size, cache line size and cache sharing). *Hierarchical blocking* is used to choose node size of nodes as well as their locations, relative to the size of the SIMD data register and the size of cache lines, so that groups of nodes can be pulled into the cache and executed on by SIMD when needed. Nodes are clustered so that a group would contain siblings and parent nodes for multiple levels to minimize cache misses. *Key compression* is used to handle variable length keys, by compressing longer keys to be of a fixed size. This translated to bounds on the minimum number of nodes (and levels of nodes) which would be contained in a cache line, and a reduction in cache misses due to longer keys and variable numbers of keys contained in each node.

Boehm *et al.* in [10] consider the use of prefix trees for in-memory storage and retrieval of data. They presented a generalized prefix tree, used for storage of arbitrary data of fixed and variable length keys. The keys are used in terms of byte representation. To ensure that the tree was balanced, the maximum length of a key used was fixed, and the prefix length used was a factor of the the maximum prefix length, ensuring that all of the leaves of the tree were at the same level. Variable length keys with a length less that the maximum were padded with zeroes. They identify two problems with the regular prefix tree that pose problems for in-memory usage – large tree height based on the prefix length, and potential for large memory consumption due to full tree expansion when more keys than the key space currently provides are inserted. To deal with these problems, a few optimizations were introduced. The *bypass jumper array* technique was introduced to deal with potentially large tree height by skipping leading zero prefixes in the key (and bypassing the associated level search) by counting how many leading zeroes were present. All the childnodes with zero prefixes were preallocted and associated with the array. Keys

smaller than the maximum key length were padded with zeros at the beginning to make the maximum length. The *dynamic trie expansion* technique was introduced to reduce memory usage by trees that are sparsely populated as well as the time taken to retrieve the data, by allowing nodes of inner levels to point to actual data, provided that there is only one data node associated with the prefix at the time. Padding for keys smaller than the maximum is done at the end for this technique to preserve order. Optimization of memory usage was also considered with techniques such as preallocation of node arrays, reducing pointers to offsets and using a node size aligned to the size of a cache line.

Kissinger *et al.* in [43] describe their KISS-Tree, which was an adaptation of the generalized prefix-tree for in-memory indexing, with the aim of minimizing memory accesses for operations and memory consumption by the Tree structure and data. The tree consists of three levels, and operations use a key size of 32 bits, with each level of the tree indexed using a different fragment of the key with a different length. The first level, called the *Virtual Level*, uses the first 16 bits of the key to directly access thee nodes of the second level, for which the nodes are stored in order of their content. The second level uses node sizes which are equal to the memory page size, and uses the virtual memory assignment by the operating system to only allow consumption of memory when a page is accessed for writing. Since this level uses the next 10 bits of the key, each node contains 2^{10} entries. With an expected page size of 4 KB, this allows for 4 Byte pointers to the third level. This pointer is broken into a bucket offset (6 bits) and the block number (26 bits) from the selected node for the third level. Each node in the third level uses the remaining 6 bits to access a bitmap of used buckets in the node, allowing for consumption of only the memory of occupied buckets. Updates are done using the Read-Copy-Update [55] pattern when the required bucket at the third level is found.

Mao *et al.* in [53] address in-memory indexing with cache awareness by combining the strengths of the Trie and B^+-Tree datastructures into one hybrid tree, which they called MassTree. They noted that long keys with shared prefixes were better served with a Trie, while the B^+-Tree was better for short keys and concurrency control. The result was the outer tree being a Trie, which used a prefix size of 8 bytes, with each node being a B^+-Tree. Inside of the node, keys are stored as close to the root of the B^+-Tree as possible, with all nodes at a level m (having length above the last 8 bytes of the previous Trie node = n) having a shared prefix of length 8h + n, where h is the height of the outer Trie. Locking in MassTree is done at the level of the node by using a version number. The version number contains bits used for indicating the node is locked, splitting, inserting, whether the node is the root or a border node, and separate version counters for splits and inserts. Reads do not lock the node, but instead read the version number before and after reading the required key from the node. If the version is different or the split/insert

bit is set, the read operation is retried. Updates first atomically set the locked bit to stop other writers from concurrently updating, then required bit (insert or delete) to indicate what type of operation is being performed and to inform readers of the need to restart. If a split is required during the update operation, that bit is set and unset accordingly for the duration of the split. When the update operation is finished, the insert or delete bit is cleared and the corresponding counter is incremented.

2.5 Contributions

This work considers the use of the B-Tree datastructure completely in main memory of a NUMA system, and suggests three modifications to the B-Tree for this environment – node replication, changing of the granularity of locking (the level up to which normal locking is allowed), and optimistic concurrency control techniques for Read operations (Lock-Free Reads). The modifications for node replication are adapted from the work done on distributed B-Trees [65, 39, 44, 4] for the NUMA environment, with physically distinct NUMA nodes being treated as the distinct servers in a distributed system. However, in the case of a NUMA system, all NUMA nodes have access to the same address space and so can access each other's data directly. The modifications for NUMA-aware locking replaces the default locking of the tree (PThread Reader/Writer locks) with an implementation of the NUMA-aware Reader/Writer lock as described in [14]. It also considers the use of replicated latch managers (in the case of node replication) for each NUMA node, and compares them with the original and NUMA-aware Reader/Writer lock implementations. The modifications for varying lock granularity consider changing the minimum node height for which the corresponding lock will be used (henceforth called Lock Granularity). The work focuses on the combination of this technique with replication to reduce lock contention, and to benefit from NUMA node workload distribution decisions. The modifications for Lock-Free reads use a version number locking technique similar to that used in [53, 23, 12]. However, instead of having many different operation bits or version numbers, it uses a simpler odd/even version number counter technique to determine when it is safe to read and when a modification to the node is occuring. "Modification" groups all operations which results in physical changes to nodes, and are represented odd version numbers. A node that is safe to read has an even version number. Also, since the tree uses logical node pointers in the form of page numbers, any inconsistencies found by an operation resulted in only that node being re-read, instead of restarting the entire operation from the root node.

Chapter 3

The Problem and Solution Implementation

3.1 The Problem

As explained in Chapter 2, much work has been done on top of the basic B-Tree data structure to improve its operations during period of increased concurrency and in distributed environments using the techniques of replication and multiversioning. However, NUMA-based architectures have not received as much attention, and although the techniques described before are applicable, many of them have steps which have too much overhead or are unnecessary, since the entire storage environment is physically contained in one address space. Also, with the size of main memory rapidly increasing, many of the datastores can now be kept in main memory, and only flushed to disk for checkpointing and fault tolerance. The work described by this thesis looks at improving the overall operation throughput of a B-Tree stored completely in the memory of such NUMA environments, by considering various combinations and variations of the locking, distribution and versioning techniques described before. The B-Tree was chosen for analysis and modification due to its usage in various different datastores for physical storage, but the techniques described here can also be extended to include any dictionary type datastore, where there is separate storage of various segments of the range of data in dictionary and for which locking techniques are used.

3.2 The Original Code

The code base upon which modifications were based is developed by Karl Malbrain, accessible from `https://github.com/malbrain/Btree-source-code` and described in the arXiv paper [52]. The tree is an implementation of a B^{link} Tree, in C, that combines the original B-link tree description described by Lehman and Yao in [47], but instead uses the techniques based upon the methods described in [38] to maintain the logarithmic time of operations by keeping the tree balanced and the deletion methods described in [46], since the original description of the B-link tree does not describe the process of deletion explicitly. This section describes the techniques used in the original code to provide the highly concurrent foundation upon whichour performance improvements build.

3.2.1 The Node and Segment

Fig. 3.1 shows the layout of the node used for key storage in the tree. The node contains a header for page accounting information. The rest of the page is used for the storage of keys (`BtKey`) and slots (`BtSlot`), which keys are stored in ascending order, by key. The node allows for storage of variable-length keys from 1 byte up to 255 bytes. The Tree is a user-specified subset of nodes stored in a contiguous area of memory, called a segment. The size of a node is specified as the size of a page, and the size of a segment of nodes is specified by the number of nodes contained in a segment. Both the page size and the size of the segment is specified by the user when the tree is created. All segments are represented as a memory-mapped locations in a file created to represent the tree when the tree is created.

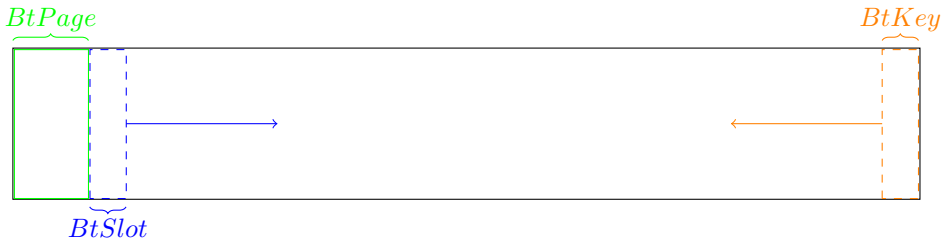


Figure 3.1: Layout of a Node

The accounting information (`BtPage`) stored at the beginning of the node includes the key count (`cnt`), the number of active (not deleted) keys (`act`), the level of the node in

the Tree (`lvl`), whether or not it contains deleted keys (`dirty`), and a right link to a potentially new node created during a split of the node (`right`). A slot (`BtSlot`) may hold logical child page number pointer if the node is an internal, or an unsigned integer if the node is a leaf node (the number can represent any useful data). The node stores slots (`BtSlot`) from the memory right after the header information towards the end of the page. Actual key information (`BtKey`) is stored from the end of the node, going towards the start of the page. The `BtKey` stores the fields `len` and `key`, defined as follows:

- `len` stores the length of the key
- `key` stores the supplied key as an array of characters of size `len`

The `BtSlot` stores the fields `off`, `dead`, `tod` and `id`, which are used as follows:

- `off` is used in a macro calculation to get the required `BtKey` information to get the length and data of the key.
- `dead` is a flag to indicate that this key has been deleted
- `tod` is a timestamp
- `id` is the logical child page number pointer associated with the separator key for an internal node, or the value being stored for the associated key for a leaf node

A node does not have an explicit fence key stored in the header. Instead, the fence key is deduced from the associated key for the largest slot value contained in the node, and whether or not the node contains a right pointer (indicating that it was split). When the area used for storing `BtSlot` structures grows to the point that it overlaps with the area used for storing `BtKey` structures, the node is declared full and is split in half, with the lower half of the keys remaining in the node currently being split and the upper half of the keys moving to the new node that is created to the right of the current node. The fence key of the old node is changed to reflect the maximum of keys within that node, and a new fence key is posted in the new node and parent node (replacing the old one) as well. Root splitting is a special case of the splitting of a node where a new root with a few keys is created and the height of the tree increases. A node is merged with its right sibling if there are no more active keys in the node after a delete operation. The contents of the right sibling node are moved into the node, the right node's deleted bit is set, and the right node's right pointer is set to point to the node to the left of it. After all of the pointers to this node have been removed, the node is returned to the free list of nodes previously allocated, which is stored in the Latch Manager (`BtLatchMgr`).

3.2.2 The Latch Mechanism

The locks used in the original implementation (`BtLatch`) are a wrapper for the implementation of Reader/Writer locks contained in the pthread library `pthread_rw_lock_t`. The locking granularity used within the tree for reads and updates is at the level of one lock used per page. The method of locking used is latch-coupling, similar to the S (Shared)-U (Update)-X (Exclusive) latching mechanism specified by Jaluta *et al.* in [38]. Malbrain identified the following issues with the locking methods specified in [38]:

- Tree traversal during Updates use hand-over-hand locking of update locks, which are incompatible with all other update lock attempts. When descending to the required node, the parent and the child node are both held under these update locks, which prevents other concurrent operations from happening on the parent as well as the child.
- Exclusive locks (X) are used on the child node when it is being split, which also is not compatible with any other lock types. This prevents any concurrent access to that node by readers, even if the key/value data required by the reader is unaffected by the split operation.

To avoid these issues, Malbrain [52] instead defined the following other combinations of lock types:

- The **`AccessIntent/NodeDelete`** lock is used for draining current and preventing new readers/writers from accessing a node that has now become empty.
- The **`Readlock/WriteLock`** lock is used just as in the case of the regular reader and writer locks.
- The **`ParentModification`** lock is used to lock the parent node to place its fence key value into the parent as the new key delimiter which directs operation traversal to that child node.

The pairs of locks were represented as one reader/writer lock with different acquiring modes (for example, the lock for AccessIntent/NodeDelete would have AccessIntent as acquiring the lock in Read mode and NodeDelete as acquiring the lock in Write mode). The resulting three locks were grouped into one logical unit, a `BtLatchSet`, which operations used for sychronization when operating on a node.

Table 3.1: Latch Compatibility Matrix for the Original Code, from [52]

		Latch Requested				
		AI	ND	RL	WL	PM
Latch Held	AccessIntent (AI)	Y	N	Y	Y	Y
	NodeDelete (ND)	N	N	Y	Y	Y
	ReadLock (RL)	Y	Y	Y	N	Y
	WriteLock (WL)	Y	Y	N	N	Y
	ParentModification (PM)	Y	Y	Y	Y	N

The compatibility of each lock when another is being requested is shown in Table 3.1.

Note that a `NodeDelete` lock is only granted if there is no `AccessIntent` locks on the node, and when there are no pointers to that node in the tree. When performing a dictionary operation, when descending to each child node, a `ReadLock` on the parent node is coupled with an `AccessIntent` lock on the child. The `ReadLock` on the parent prevents it from being deleted while a concurrent insert may be happening (which may require a split and hence, a fence key update in the parent node). The `AccessIntent` lock also prevents deletion of the child node, and does not block provided there is not a `NodeDelete` lock on the node. Once the `AccessIntent` lock is obtained on the child node, it is then latch-coupled with a `ReadLock` on the child node, and the descent continues. For update operations, when the operation has arrived at the child node which is to be modified, the lock is changed to a `WriteLock` and the modification of the node is started.

The `ParentModification` lock is used to allow at most one insert operation at a time to perform updating of parent separator keys associated with the current child node being split. When an update causes the split of a node, a `ParentModification` lock is set to indicate that a fence key update for the node's parent must be done. This lock does not block any of the other lock groups from being acquired on the same node, so it still allows other concurrent operations to continue while updating the parent. An update to the parent is treated as an insert of the fence key to that parent node's level, so the required locking mechanism to have a safe insert into the parent is handled by that new insert operation. The newly created node also has a `WriteLock` and `ParentLock` acquired, to allow for modification of a potentially new parent for the new sibling, and the `WriteLock` of both nodes are kept until the keys are split evenly between the two child nodes and then released. The `ParentModification` locks are released after the insertion of the fence keys into the parent nodes.

The distribution of the `BtLatchSets` is done from a central point, the Latch Manager (`BtLatchMgr`), which occupies one page. A number of pages (the number is calculated using

the size of the page and the size of a `BtLatchSet` is used by the Latch Manager as memory to be used for allocating `BtLatchSets` to nodes. The Latch Manager contains fields for latch distribution accounting – total latch pages available for distribution (`nlatchpage`), latch pages deployed (`latchdeployed`), a hash table for caching currently used latches which are pinned (`table`), a list consisting of the next available page, and pages which were freed after consolidation and are available for reuse (`alloc`), and a small spinlatch for guarding modifications to this node page allocation list (`lock`). Each entry of the hash table has an associated spinlock for reads and modifications and an associated slot entry used to index the actual latch set page to retrieve the latch set used by the node page. When a latch is requested for a page, the modulus of the page number and the number is used as an index into the hash table of latches, with the retrieved entry being locked. If the entry was assigned to the page before, or is a new assignment, it is returned. If there are no such entries to return, the code waits for one of the latch sets to become unpinned, stores an entry in the Latch Manager's hash table for it and returns it. When operation is done with the page, it unpins the page, allowing use by other node pages. Both the `BtLatchMgr` and the pages used for allocating `BtLatchSets` are memory-mapped to the B-Tree file.

3.2.3 The Buffer Pool Manager

Management of the Nodes in the B-Tree is handled similarly to [59], in that the entire collection of B-Tree nodes is further collected into segments of a fixed number of pages (`BtPool`), for which a hash table of the segments is managed by the manager of the B-Tree (`BtMgr`). The contiguous segment pages are mapped directly to the associated B-Tree file, and has a user defined size, which by default is the size of a disk page, but can be assigned to any size seen fit by the user of the tree. Pinning and eviction logic similar to what is used for the latch sets are used with the segments. A page within a segment is obtained by using an offset, calculated from the page number, added to the starting address of the mapping. The number of segments that are managed is specified upon B-Tree creation, with a maximum of 65,536, the total number of simultaneous file mappings allowed by the Windows and Linux operating systems.

3.3 Solution Implementation

The improvements to the B-Tree code focus on two aspects of the tree structure – location of the nodes and protection of the data and the structure. The modifications to the B-Tree

focus on Reads and Inserts only, since the focus of this work is on read-heavy workloads, but can be extended to Deletes as well, and such modifications will be discussed in Chapter 5. I added 7815 lines of code to the original code to implement all of the modifications discussed below.

3.3.1 Latch Manager and Buffer Pool Manager Spinlock Removal

The experiments involving varying number of client threads initially showed a drastic drop in throughput performance, even with just a relatively small increase in the number of threads (up to about 8 client threads). This indicated that there was some contention elsewhere in the code besides the group of latches allocated to a node, and the contention was most likely due to locking. The Latch and Buffer Pool Managers were the only other place in the code that used locks, specifically the hash table for referenced segments in the Buffer Pool Manager, and the hash table for the allocated latch sets in the Latch Manager. Both hash tables used spinlocks to protect each hash slot and hash chains for items which mapped to the same hash slot, if the hash table size was smaller than the total number of unique requests. The spinlock was used in write (exclusive) mode when updating a hash chain for a previously unhandled request, otherwise it was used in read mode. There was also a limit on the total number of concurrent available segment references and latch sets which the respective managers were responsible for at any point in time. The managers used a caching scheme to maintain (or "pin" in the code) the most recently referenced segments and latch sets in the table. Mapped memory for evicted segments was written back to the Tree file, and evicted latch sets were re-initialized for use with the new node. Contention for the lock on a slot would occur if at least one of the requests for the same segment or latch set caused a hash chain update. To overcome this bottleneck, the following modifications were introduced:

1. The size of hash tables that were used to hold the references to the Segments and Latches was set to be equal to the total number of Segments and Latches which were created, so that every segment and latch set would have its own location in the respective table. This reduced the size of the hash chains for each slot to be at most one element.

2. All of the Segments and Latches that would be needed were pinned upon creation, before any request operation. This allowed future requests to be served by the the hash table without the need to update hash chains.

3. Locks for all of the hash table slot were removed.

This improved the performance of the Tree and was used for all experimental results obtained.

3.3.2 NUMA-aware Locking

Each Node in the original implementation is granted a grouping of latches (`BtLatchSet`) from the latch manager (`BtLatchMgr`), used for sychronizing reads and updates. Each latch used in the grouping uses the PThread library implementation of the Reader/Writer Lock (`pthread_rwlock_t`) as the underlying implementation of the latch. Although Reader/Writer locks allow for increased concurrency of readers, the implementation is oblivious to the physical main memory layout. Since the environment being considered is one with a NUMA architecture, an implementation of locks that takes advantage of the physical main memory layout of the architecture could provide improvements in overall operation throughput. As stated in Section 2.2.1, more recent techniques focus on local acquisitions of the lock and hand-off to other threads of the same node. Since the original implementation used was a Reader/Writer Lock, a NUMA-aware version of the Reader/Writer lock was used as a replacement [14]. The implementation of the lock used was adapted from Azu-Labs (`https://github.com/azu-labs/rw-numa-locks.git`), with modifications to work in the environment used and to remove node-identification system calls, which are time consuming.

3.3.3 Node Replication

The physical main memory and processor layout of a NUMA system can result in operations accessing data contained in a memory module of a remote processor's NUMA node. These remote memory accesses require the requested memory data to be transported across interconnect links into the cache hierarchy of the requesting processor. This architecture is similar in layout to a distributed system consisting of a set of physically separate servers, so techniques which are used to improve the performance in a distributed system are also applicable to a NUMA system. One technique used to improve turnaround time for operations is keeping a copy of the required data near to the location of the processor executing the operation. The replication of information in a distributed systems can be adapted for use within the NUMA system, but without the need for global identifiers, since all system memory is available from a single address space provided by the Operating System. Since

the node is the smallest unit used to store and access data in the Tree, it was selected as the unit to be replicated, as done in the distributed B-Tree works described previously [65, 39, 44, 4]. This implementation created a copy of the entire B-Tree within the main memory module of all NUMA nodes. When an executing thread calls a B-Tree operation, the B-Tree reads each node from the copy of the Tree that is contained within the main memory module of the NUMA node that the operation is being executed on.

Custom Segment and Page Memory Allocator

To replicate B-Tree nodes and segments for each NUMA node in the system, all B-Tree node/segment creation requests from a NUMA node were satisfied by allocating memory from the node where the request originated using the Linux numa library (`numa.h`'s `numa_alloc_onnode()`). Almost immediately, this resulted in the inability to satisfy the large number of requests being made for memory. This is because internally, every call to `numa_alloc_onnode()` creates a new anonymous memory map, bound to the NUMA node specified. In Linux, there is a cap on the number of such mappings which can be held by a process at one time, stored in `/proc/sys/vm/max_map_count`. To get around this, a custom memory allocator for segments and pages was created to handle those allocation requests. Internally, it created one large memory map for the total number of expected segments and another one for the total number of expected pages. The allocation requests for each were tracked internally using allocation lists for each. This lowered the number of mapping requests by the B-Tree from an unbounded number to just two.

Key Lookup and Insertion

Section 2.3 describes how replication of B-Tree nodes on each physical server in the Distributed System decreased the time required to execute read operations and update operations. This work adapts the B-Tree node replication technique, but instead, treating each NUMA node as the physical server. Newly created B-Tree nodes and segments(from split operations) were replicated on every NUMA node. The file mapping from the original code was not kept, since the main consideration of this thesis was the B-Tree performance in main memory, and not persistence or fault tolerance. The Latch Manager was also replicated for each NUMA node, so that the latch sets allocated for nodes of a local B-Tree were also obtained from the local memory module containing the B-Tree nodes. Replication of nodes, segments and latch sets allowed for operations on a local node to have all its memory reference requests be satisfied from the local memory module. This eliminated remote memory accesses for nodes, segments and latch sets. The algorithms for the Read

and Update Operations were modified in three places, the code to load the correct page for the key, the code that splits a page when it is full, and the code that updates/reads the key from the page as necessary. The pseudocode for traversing the B-Tree is shown in Algorithm 1. The find key pseudocode is shown in Algorithm 2. The pseudocode for Inserts is similar to the pseudocode for finding the key, except that all copies of the B-Tree are handled during Tree traversal, key insertion and node splitting.

Algorithm 1 Pseudocode for `replicatedLoadPageRead`

Input: *key* (Byte Array), *value* (Integer), *page_set_pointer* (Structure), *node_lvl* (Integer), *lock_mode* (Structure)
Output: slot for *key* returned (Integer), *page_set_pointer* set appropriately

```
 1: current_page_no := ROOT_PAGE_NO
 2: previous_page_no := current_page := current_segment := current_page_lock := keyslot := ∅
 3: lvl := ∞
 4: while true do
 5:     current_segment := segmentFromLocalPool(current_page_no)
 6:     current_page := pageFromSegment(current_segment, current_page_no)
 7:     if lvl = ∞ then
 8:         lvl := current_page.lvl
 9:     end if
10:     lock(current_page_lock,ACCESS_INTENT)
11:     if previous_page_no ≠ ∅ then
12:         unlock(previous_page_lock, READ_LOCK)
13:     end if
14:     if lvl = node_lvl then
15:         lock(previous_page_lock, lock_mode)
16:     else
17:         lock(previous_page_lock, READ_LOCK)
18:     end if
19:     lock(current_page_lock,READ_LOCK)
20:     unlock(current_page_lock,ACCESS_INTENT)
21:     keyslot := scanForSlot(current_page, key)
22:     previous_page_no := current_page_no
23:     previous_page_lock := current_page_lock
24:     if lvl = node_lvl then
25:         if keyslot ≠ ∞ then
26:             page_set_pointer.page := current_page
27:             page_set_pointer.segment := current_segment
28:             page_set_pointer.lock := current_page_lock
29:             return keyslot
30:         else[Page was being split while keyscan was done]
31:             current_page_no := getRightPageNumber(current_page)
32:         end if
33:     else
34:         current_page_no := getChildPageNumberFromSlot(current_page, keyslot)
35:         lvl := lvl − 1
36:     end if
37: end while
```

Algorithm 2 Pseudocode for `replicatedFindKey`

Input: *key* (Byte Array), *node_lvl* (Integer)
Output: *value* associated with *key* (Integer) or ∅ if key not found

```
1: keyslot := page := segment := ∅
2: page_set_pointer := memory allocated for structure
3: keyslot:= replicatedLoadPageRead(key, page_set_pointer, node_lvl, READ_LOCK)
4: if keyslot ≠ ∅ then
5:     bt_slot := getBtSlotFromPage(keyslot, page_set_pointer.page)
6:     return bt_slot.value
7: else
8:     return ∅
9: end if
```

The functions that are not in bold existed in the original code, and are used for obtaining node and key information from the tree.

- `pageFromSegment` takes as paramaters a page number (logical page pointer) and a pointer to the segment containing the page, and returns a pointer to a page.
- `lockForPage` takes the page number as a parameter, and returns the latchset structure to be used with the page
- `scanForSlot` takes a pointer to a page and a key as parameters and returns slot number for the `BtSlot`/`BtKey` pair associated with the key
- `getChildPageNumberFromSlot` takes as a page pointer and slot as parameters and returns the logical child node pointer (associated with the key used to obtain the slot) from the `BtSlot` at at the specified slot

The original algorithm of `findKey` was modified with the functions in bold. The function in bold, contained the modifications to support the replication of nodes. The rest of the code was similar to what is done in the non-replicated case. The two modified functions were implemented as follows:

- `replicatedLoadPageRead` contains similar logic to the original version of `loadPage`, except that it loads the local version of the page that contains the key that is being searched for, or the local verson of the page that would have contained the key if it existed. It loads the version of the page contained in the local version of the pool, loaded by `segmentFromLocalPool`, using the `node_id` supplied by the caller.

- `segmentFromLocalPool` contains the same logic from the original code for mapping the page number of the node to the segment that contains it. However, the manager of the tree keeps local, identical copies of the segment pool on each node, at a known address. The `node_id` is supplied to the call to indicate which pool of segments to check for the the required segment. Since only local segments from the local memory module are read in this function, the operation execution remains local to the node where the operation was started.

The code for `replicatedLoadPageUpdate` is similar to `replicatedLoadPageRead`, except that all segments with the required page (`allSegments`) and all copies of the required page (`replicatedLoadPageUpdate`) are obtained during traversal. `replicatedInsertKey` updates all pages with the inserted key and required splits before the operation returns. This results in strong consistency for key inserts and updates. This consistency model was chosen since the original code implicitly had the same consistency model (updates are seen immediately by subsequent operations). Different consistency models can be considered based on client needs. This was not considered in this thesis, but could be Future Work.

Handling Inserts

Node replication introduces a new problem with the contents of the tree when updates are included — consistency of changed nodes. Many previous works talk about how consistency can be addressed in a distributed environment, and many of these techniques can be adapted to fit the NUMA environment. This thesis considers two versions of the update operation — obtaining of all required nodes and locks (with the correct mode of Read/Write) as the tree is traversed, and insertion of keys only into the local copy of the node. The former version guarantees strong consistency among replicas after the operation is completed, but its execution time depends on the number of replicas and is very slow. The latter provides very fast updates but produces inconsistent versions of B-Tree nodes in the tree, which need to be handled separately from the operations being performed on the B-Tree by clients. Finding a technique that handles eventual consistency in the NUMA environment was not the main focus of this work, and is instead discussed in the Future Work section of Chapter 5.

3.3.4 Granular Locking

The original implementation of the Tree assigns a latch set to each Node page to protect operations performed on it. Allocating a latch set for each page allows for maximum

concurrency when there are conflicting operations which need to read the same path of ancestor nodes to get to the required node. However, the locking of a node can be considered overhead for the operation, since it does not contribute towards the result of the operation. The more locks required during the operation, the more overhead experienced (and time wasted) during the operation. Read-heavy and read-only workloads mostly read from the required nodes, and can potentially reduce locking overhead by using a subset of the locks during each operation. The use of a subset of the total number of locks on the path to the node required is what is referred to as "more Granular Locking" in this work. The approach to granular locking used here is to associate latches to Node pages up to a certain level in the tree, measured from the root. For all Node pages below this level, the latch that was granted to the last level is reused. The pseudocode for the loading the page for the required operation using this mechanism is shown in Algorithm 3 and Algorithm 4.

The minimum latch grant level is stored within the manager of the tree. When searching for the correct page and slot in `loadPage`, the code checks to see if the level of the page that was retrieved is at a level where a new latch grant request will be allowed. If the level is above the level that is allowed, then the regular locking procedure is done. If the level goes below, then `loadPageInternal` is called. `loadPageInternal` differs from the original `loadPage` code, in that it performs the hand-over-hand locking once outside of the loop, and does no further locking when pages at the current level and lower are accessed. The update versions of the two code calls are identical, except for the fact that in `loadPageInternal`, the final lock mode used is a write lock, since at some point, the required page will require that lock. The aim of the `loadPageInternal` call is to minimize the number of lock acquisitions for the reused lock (to 1 acquisition).

3.3.5 Lock-Free Reads

Another technique that was considered for the B-Tree was lock-free traversal. Lock-free techniques could be implemented for all operations, or a subset of operations. Depending on the operations being considered, the performance could be better or worse, and the code would become much more complex. The choice of B-Tree operations to modify was made based on the workload type that was expected, the operations that were being considered, and how conflicting operations would be handled. The main workload type being considered was read-heavy (at most 25% updates). Since reads were expected to be most common, they were modified to use the Lock-Free techniques. Also, since Lock-Free techniques would result in operation restarts when inconsistencies were discovered upon descent and access of nodes, a read-heavy workload would have fewer restarts than an update-heavy one, since there are fewer updates happening in the workload. For update

Algorithm 3 Pseudocode for `loadPage` with the Granular Locking Mechanism

Input: *key* (Byte Array), *page_set_pointer* (Structure), *node_lvl* (Integer), *lock_mode* (Integer)
Output: slot for *key* returned (Integer), *page_set_pointer* set appropriately

```
 1: current_page_no := ROOT_PAGE_NO
 2: current_page := current_page_lock := previous_page := current_segment := keyslot := ∅
 3: lvl := ∞
 4: while true do
 5:     current_segment := segmentFromPool(current_page_no, node_id)
 6:     current_page := pageFromSegment(current_segment, current_page_no)
 7:     if lvl = ∞ then
 8:         lvl := current_page.lvl
 9:     end if
10:     if current_page.lvl ≤ LOCK_GRANT_LEVEL then
11:         return loadPageInternal(key, current_page, previous_page, current_segment,
    current_page_lock)
12:     end if
13:     current_page_lock := lockForPage(current_page_no)
14:     lock(current_page_lock, ACCESS_INTENT)
15:     if previous_page exists then
16:         unlock(previous_page_lock, READ_LOCK)
17:     end if
18:     if lvl = node_lvl then
19:         lock(current_page_lock, lock_mode)
20:     else
21:         lock(current_page_lock, READ_LOCK)
22:     end if
23:     unlock(current_page_lock, ACCESS_INTENT)
24:     keyslot := scanForSlot(current_page, key)
25:     previous_page_no := current_page_no
26:     previous_page_lock := current_page_lock
27:     if lvl = node_lvl then
28:         if keyslot ≠ ∞ then
29:             page_set_pointer.page := current_page
30:             page_set_pointer.segment := current_segment
31:             page_set_pointer.lock := current_page_lock
32:             return keyslot
33:         else[page was being split while keyscan was done]
34:             current_page_no := getRightPageNumber(current_page)
35:         end if
36:     else
37:         current_page_no := getChildPageNumberFromSlot(current_page, keyslot)
38:         lvl := lvl − 1
39:     end if
40: end while
```

Algorithm 4 Pseudocode for `loadPageInternal` with the Granular Locking Mechanism

Input: *key* (Integer), *page_set_pointer* (Pointer), *node_lvl* (Integer), *lock_mode* (Integer), *current*$_p$*age* (Pointer), *current_segment* (Pointer), *current_page_lock* (Pointer), *previous_page* (Pointer), *previous_segment* (Pointer),*previous_page_lock* (Pointer)
Output: slot for *key* returned (Integer), *page_set_pointer* set appropriately

```
1: keyslot := ∅
2: lvl := LOCK_GRANT_LEVEL
3: lock(current_page_lock, ACCESS_INTENT)
4: if previous_page exists then
5:     unlock(previous_page_lock, READ_LOCK)
6: end if
7: lock(current_page_lock, lock_mode)
8: unlock(current_page_lock, ACCESS_INTENT)
9: while true do
10:    current_segment := segmentFromPool(current_page_no, node_id)
11:    current_page := pageFromSegment(current_segment, current_page_no)
12:    if lvl = ∞ then
13:        lvl := current_page.lvl
14:    end if
15:    keyslot := scanForSlot(current_page, key)
16:    previous_page_no := current_page_no
17:    previous_page_lock := current_page_lock
18:    if lvl = node_lvl then
19:        if keyslot ≠ ∞ then
20:            page_set_pointer.page := current_page
21:            page_set_pointer.segment := current_segment
22:            page_set_pointer.lock := current_page_lock
23:            return keyslot
24:        else[page was being split while keyscan was done]
25:            current_page_no := getRightPageNumber(current_page)
26:        end if
27:    else
28:        current_page_no := getChildPageNumberFromSlot(current_page, keyslot)
29:        lvl := lvl − 1
30:    end if
31: end while
```

operations, only inserts and updates were considered. This simplified the complexity of descent through the tree using Lock-Free techniques, because in this case there would not be a chance of trying to read from the wrong node or a node concurrently being modified. The second consideration was how to deal with conflicting operation types while using Lock-Free techniques. Read/Write conflicts were handled by making a version number change visible to the read operation just before and after some data contained in the Node is modified. A reader would not block a writer if they happened at the same time, but could detect conflicts based on the state change initiated by the writer and restart. A write/write conflict would also need a state change to mark the node for modifications, that would be detectable by future writers upon access. This marked state would make the Node inaccessible for updates until the write operation that marked it has finish. This, however, is locking, with blocked writers restarting in a loop instead of being suspended and awakened. When these factors were considered, it was determined that the use of Lock-Free techniques in the read operation case only was preferred. The pseudocode for Read operations is shown in Algorithm 5 and Algorithm 6. All version numbers were were implemented using the `atomic_uint_fast64_t` type of the `<stdatomic.h>` header. The access member functions to the integer type included memory barriers to ensure proper in-order execution of accesses to the version number. Version number access was guaranted to always be atomic on the test system through the internal use of memory barriers for memory access, and due to its size (in bits) allowing its modification in a single memory access. Version number access was also guarantted to be lock free on the test system by verifying that the library specific macro `ATOMIC_LONG_LOCK_FREE` was set to 2 (indicating lock-free access to the variable's memory always) before Tree creation.

Algorithm 5 Pseudocode for `loadPageLockFreeRead`

Input: *key* (Byte Array), *value* (Integer), *page_set_pointer* (Structure), *node_lvl* (Integer), *lock_mode* (Integer), *previous_version_ptr* (Atomic Integer Pointer)
Output: slot for *key* returned (Integer), *page_set_pointer* set appropriately

```
 1: page_set_pointer.page_no := ROOT_PAGE_NO
 2: keyslot := ∅
 3: lvl := ∞
 4: while true do
 5:     page_set_pointer.lock := lockForPage(page_set_pointer.page_no)
 6:     page_set_pointer.segment := segmentFromPool(page_set_pointer.page_no)
 7:     page_set_pointer.page := pageFromSegment(page_set_pointer.segment,
    page_set_pointer.page_no)
 8:     previous_version := page_set_pointer.page.version
 9:     if previous_version is odd then
10:         continue
11:     end if
12:     keyslot := scanForSlot(page_set_pointer.page, key)
13:     if lvl = node_lvl then
14:         if keyslot ≠ ∞ then
15:             if (page_set_pointer.page.version is even) AND
    (previous_version = page_set_pointer.page.version) then
16:                 return keyslot
17:             else
18:                 previous_version := page_set_pointer.page.version
19:                 continue
20:             end if
21:         else
22:             page_no := page_set_pointer.page_no
23:             page_set_pointer.page_no := getRightPageNumber(page_set_pointer.page)
24:             if (page_set_pointer.page.version is odd) OR
    (previous_version ≠ page_set_pointer.page.version) then
25:                 page_set_pointer.page_no := page_no
26:                 continue
27:             end if
28:         end if
```

Algorithm 5 Pseudocode for `loadPageLockFreeRead` (continued)

```
29:     else
30:         page_no := page_set_pointer.page_no
31:         page_set_pointer.page_no := getChildPageNumberFromSlot(page_set_pointer.page,
    keyslot)
32:         if (page_set_pointer.page.version is odd) OR
    (previous_version ≠ page_set_pointer.page.version) then
33:             page_set_pointer.page_no := page_no
34:             continue
35:         end if
36:         lvl := lvl − 1
37:     end if
38: end while
```

Note that the variable `previous_version` is a local thread integer variable, of which the address is passed to the `loadPage` function for updating within the function. The structure representing the accounting information for pages (`BtPage`) was updated to hold an integer (`version`) which is read and updated atomically. Updates still use the locking functionality of the original code. Update operations load pages using the regular hand-over-hand locking technique, and always start with an even number for the version number of the node. The version number is updated once just before the modification, indicating to readers that an update operation is taking place. The version number is then incremented after the update operation is complete, to indicate to readers that the Node is in a consistent state for reading. If the page needs to be split before the insert (`splitPage()`, not shown here), then before the original page is modified to contain the lower keys, its version is incremented (now odd). A copy of that page is made having the keys greater than the middle value with a new version number of zero. Reads can only access this through the right pointer and inserts would not be able to access it until its fence key and logical pointer is inserted into the parent. Fence key updates to the parent(s) are done using the same insert mechanism as regular keys, but instead at the level of the parent. Splitting of the root is done similarly to splitting of a page, and is performed as a check within `splitPage`. Reading of pages is contained in a loop which is only broken if the version number for the node has not changed and is not odd. Since the tree uses logical pointers (page number) instead of actual pointers, and memory that is allocated for nodes is never deleted in an update, there is no case of accessing an invalid memory location for a node. Version number coupling, similar to the latch coupling discussed in Section 2.1.1, was therefore not implemented. Deletes were not considered in this implementation (they would require version number coupling). Handling Deletes is discussed in the Future Work section of Chapter 5.

Algorithm 6 Pseudocode for `findKey` for the Lock-Free Read Mechanism

Input: *key* (Byte Array), *node_lvl* (Integer)
Output: *value* associated with *key* (Integer) or ∅ if key not found

```
 1: keyslot := page := segment := ∅
 2: page_set_pointer := memory allocated for structure
 3: keyslot := page := segment := ∅
 4: previous_version_ptr :=∅
 5: while true do
 6:     keyslot := loadPageLockFreeRead(key, page_set_pointer, node_lvl,
    previous_version_ptr)
 7:     if keyslot ≠ ∅ then
 8:         if (previous_version is even) AND
    (previous_version = page_set_pointer.version) then
 9:             bt_slot := getBtSlotFromPage(keyslot, page_set_pointer.page)
10:             return bt_slot.value
11:         else
12:             continue
13:         end if
14:     else
15:         return ∅
16:     end if
17: end while
```

Algorithm 7 Pseudocode for `insertKey` for the Lock-Free Read Mechanism

Input: *key* (Byte Array), *value* (Integer), *node_lvl* (Integer)
Output: **true** if *key* inserted, else **false**

```
 1: keyslot := page := segment := ∅
 2: page_set_pointer := memory allocated for structure
 3: keyslot := page := segment := ∅
 4: previous_version_ptr :=∅
 5: status := false
 6: (keyslot, page, segment) := loadPage(key, previous_version)
 7: keyslot := loadPage(key, page_set_pointer, node_lvl,)
 8: if keyslot ≠ ∅ then
 9:     if page_set_pointer.page is full then
10:         splitPage(page_set_pointer)
11:     end if
12:     Atomically increment page_set_pointer.version
13:     bt_slot := getBtSlotFromPage(keyslot, page_set_pointer.page)
14:     bt_slot.value := value
15:     status := true
16:     Atomically increment page_set_pointer.version
17: end if
18: return status
```

3.3.6 Combination of Techniques

In addition to the techniques described before, this work also considers combination of a subset of the techniques. Replication is orthogonal to all of the locking techniques, so it can be applied in combination with all of the locking techniques discussed. Some of the locking techniques are also applicable in combination with each other. The locking combinations considered are as follows:

- NUMA-aware reader/writer locks and the Granular Locking Technique
- Lock-Free Read Operations and Write operations with PThread Reader/Writer locks(Granular Locking and/or
- Lock-Free Read operations and Write operations with NUMA-aware Reader/Writer locks)
- Replication with each of the three combinations described aboe

The aim of trying these combinations of techniques was to find the combination of techniques which performed the best for various type of read-heavy workloads (at most 25% updates in the workload), varying number of clients, and varying tree sizes. When the number of clients was varied, the clients were distributed among the NUMA nodes as evenly as possible.

Chapter 4

Experimental Results

4.1 The Test Environment

A machine with a NUMA architecture layout as shown in Fig. 4.1 was used to test the optimizations described in Chapter 3. Each NUMA node contained an Intel Xeon E5-4620 CPU with turbo boost enabled (base frequency of 2.2 GHz and maximum turbo frequency of 2.6 GHz) and a 64 GB memory module. Each Xeon CPU had:

- 8 cores, with 2 threads per core
- Cache line size of 64 B
- Private L1 Instruction and Data Caches for each core, each of size 32 kB
- Private L2 Cache for each core of size 256 kB
- Shared L3 Cache for all the cores of size 16 MB

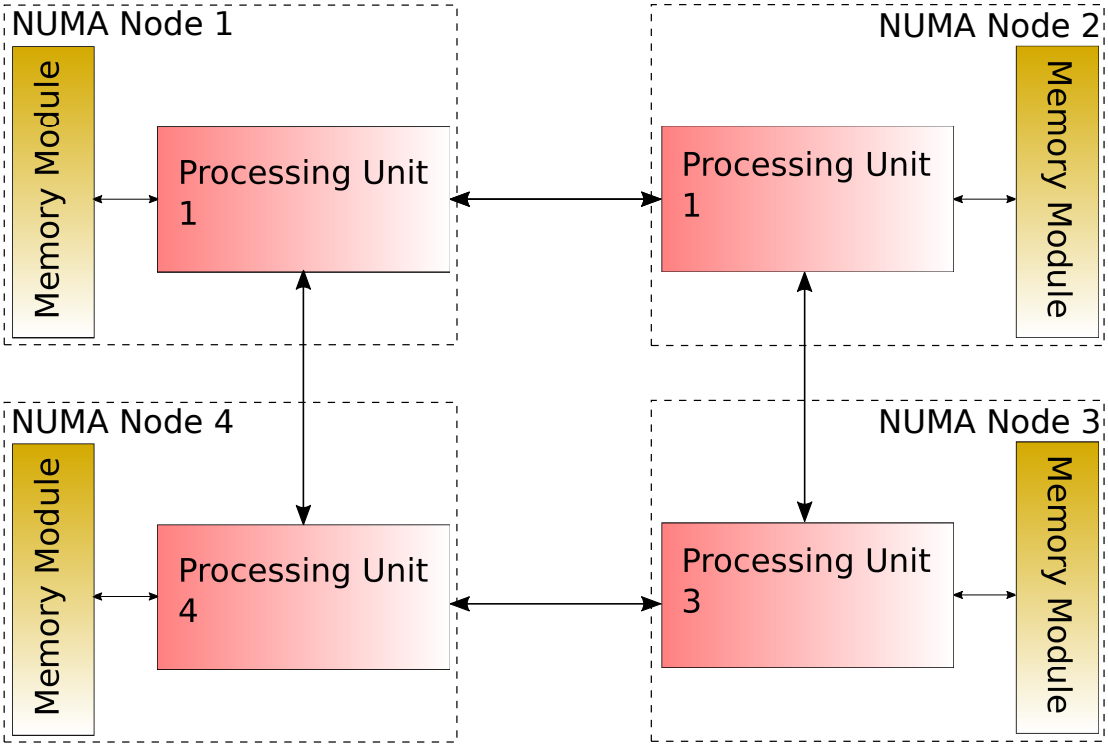


Figure 4.1: Diagram of the NUMA Node Topology of the Test System

4.2 Comparing Performance: Throughput

To evaluate the performance of the Tree with the modifications specified in Chapter 3 a metric had to be considered. Since the aim of introducing the techniques was to improve the overall performance of concurrent operations on the tree for read-heavy workloads, the throughput of operations performed on the tree for such a workload was chosen as the performance metric. Each required modification was added to the object code for the B-Tree through the use of `#DEFINE` symbols. A driver program was built to perform Insert and Update operations from each of the NUMA nodes in the system. The number of clients, number of operations per client and the workload ratio (read/write ratio) were all options provided to the driver. When the driver is started, the driver tries to place all of the client threads as evenly as possible among the NUMA nodes, provides each with a list of read and update operations to execute, and lets them run. Each client thread measures the time it takes to execute all of the operations and then calculates and outputs its operation throughput to a file on the disk. The driver was run three times and the average of all of the runs for each client thread file was taken as the throughput of that client for the particular run options. The average throughput of all threads was then added together to get the overall throughput of all of the client threads for the run.

4.3 Choosing the Page and Segment Size

When creating the tree, the following parameters can be set:

- **Page Size (in bits)** - This controls the size of a Node, the space used for the Latch Manager, and the space used for Latch Sets that are granted for a `loadPage()` request. It can be specified as at least 12 bits (4,096 B) and at most 20 bits (1 MB)
- **Number of Pages per Segment (in bits)** - This controls how much of a contiguous run of the file mapping (and memory) used for Node pages are returned as a group for a Segment.
- **Number of Segments** - This controls the number of Segments and indirectly, the number of pages, that can be mapped in-memory at any given time. This has an upper limit of 65,530 (the maximum number of memory mappings on the system being used).
- **Hash Size** - This controls the size of the hash table used to monitor all allocations of Segments.

Since the throughput of operations on the tree was the main concern, all parts of the code which could serve as a contention bottleneck were considered, which brought three things to the immediate attention – the size of the hash table used to manage the Segments, the size of the hash table used to manage the Latch Manager hash entries for granted latches, and the number of Latch Sets which were being managed. Each request to find a Segment or a Latch Set uses a Spinlatch granted from a hash kept by the Buffer Pool Manager in the case of Segments, and the Latch Manager in the case of Latch Sets. The more entries there are, the less contention per Spinlatch. The size of the Buffer Pool manager hash table is set to be the **Hash Size** stated above, and to minimize contention, value of this was set to be at least the total number of segments required. To calculate the number of Segments required, the following equation was used, knowing that 4 byte integers were used as the keys, and an 8 byte integer was stored as the value associated with each key (either a child page number if page is internal, or an associated value in the page if a leaf):

$$\#\text{ of Pages} = \left\lceil \frac{\text{Total \# of keys}}{\left\lfloor \frac{\text{Page Size}-\text{sizeof(BtPage)}}{\text{(sizeof(key)+sizeof(value))}} \right\rfloor} \right\rceil$$

$$\#\text{ of Segments} = \left\lceil \frac{\#\text{ of pages}}{\#\text{ of pages per segment}} \right\rceil$$

The size of the Latch Manager hash table, and therefore, the number of available Spinlatches for use when updating Latch Set related information in the hash table, is controlled by the size of the page selected. The available spinlatches are obtained from the remaining space in a page, after the Latch Manager Structure space usage has been accounted for:

$$\#\text{ of Spinlatches} = \#\text{ of Pages}$$

The number of Latch Sets available for association with Node pages is determined by the size of a page as well:

$$\#\text{ of Latch Pages} = \left\lfloor \frac{\text{BT\_latchtable}}{\left\lfloor \frac{\text{Page Size}}{\text{sizeof(Latch Set)}} \right\rfloor} \right\rfloor + 1$$

$$\#\text{ of Latch Sets} = \#\text{ of Latch Pages} \times \text{Page Size}$$

In the original code `BT_latchtable` is a constant, defined as 128, and was the number of slots desired in the hash table for the Latches.

When considering the number of Spinlatches used for the Latch Manager and the number of Latch Sets available for use by Node pages, to decrease contention, the Page Size should be as large as possible.

Considering Socket Layout

The way that data is laid out in and accessed from memory also affects the Tree's performance. As seen in Fig. 4.1, each node has 64 GB of main memory, and a CPU with 8 cores, each with private L1 and L2 caches, and all cores sharing an L3 cache. The best performance would use the caches and main memory most efficiently. Since each core has its own private L1 and L2 cache, any access and modifications that can be contained within the TLB or cache without a miss or the need for invalidation of other caches would speed up operations. Each page access also requires first finding the segment it belongs to, then its position in the segment. With this in mind, page sizes from the minimum up to at most the size of the L2 cache were considered. The segment size for these varying page sizes was set to the size of the L1 data cache and the L2 cache respectively. With the page size and segment size set and using the original code, a tree of 1,000,000 keys was created using 4 different threads, each pinned to a unique NUMA node, to insert a subset of the total number keys. After the tree was created, 4 other client threads were used to perform a set number of read operations concurrently on the Tree, each pinned to a unique NUMA node and calculating the overall throughput of operations. The results for a Segment size of 32 kB and 256 kB are shown in Fig. 4.2.

A page size of 128 kB and a segment size of 256 kB gave the highest throughput results. This setting for page size and segment size was used for the remaining experiments.

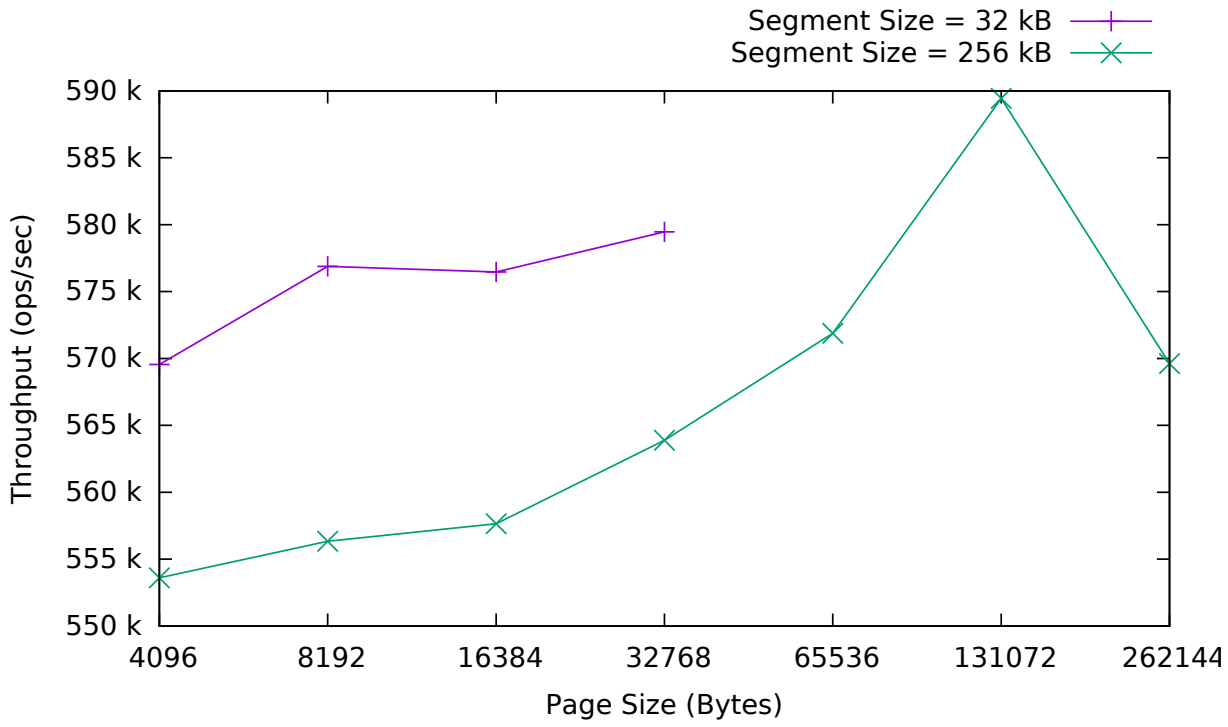


Figure 4.2: Throughput vs Page Size for Segment Sizes of 32 kB, 256 kB, Tree Size of 1,000,000 keys, 4 Client Threads, 100 % Reads

4.4 Operation Throughput for Separate Techniques

The first performance comparison was done for each of the techniques separately, relative to the performance of the original B-Tree. Fig. 4.3 and Fig. 4.4 show the results for varying number of client threads (4 - 32) and read-heavy workload ratios (100% Reads/0% Updates, 75% Reads/25% Updates). For a Read-Only workload, Lock-Free Reads, NUMA-aware

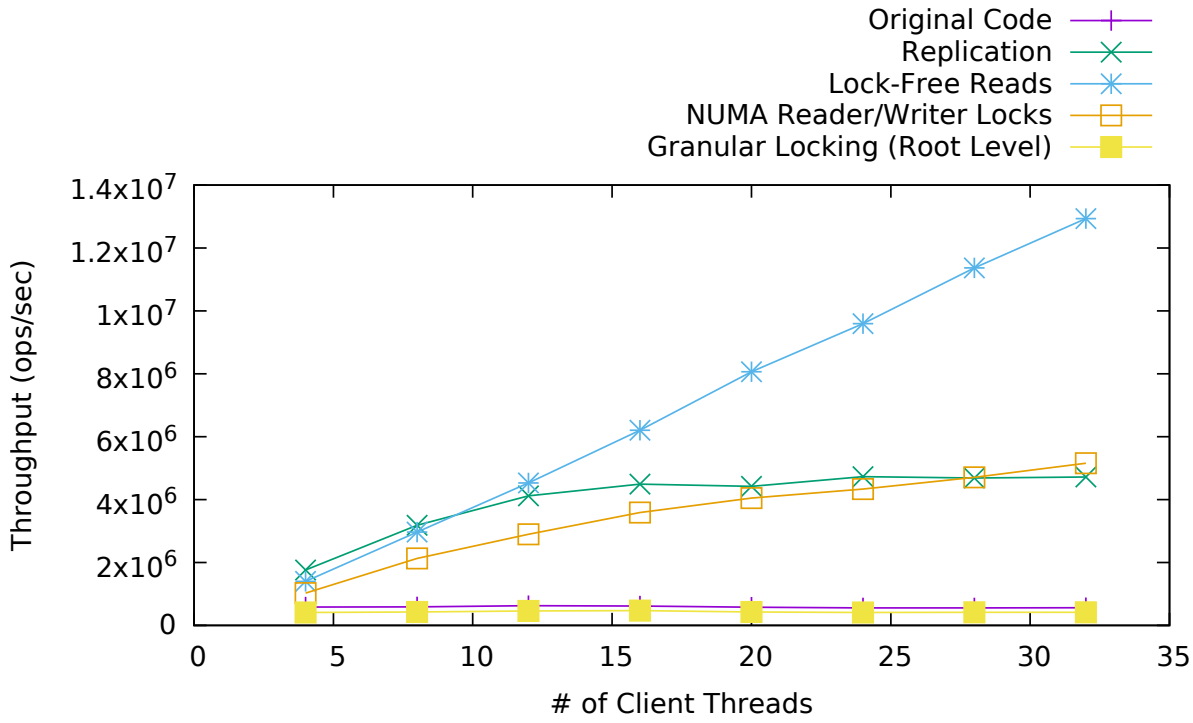


Figure 4.3: Operation Throughput vs Number of Clients, Initial Number of Keys = 1,000,000 , 100% Reads, 0% Updates

Reader/Writer Locks and Replication all perform better than the original code. All of the mentioned techniques have throughput increases relative to the original code when the number of client threads is increased (23×, 9×, 8× for Lock-Free Reads, NUMA-aware Reader/Writer Locks and Replication with 32 client threads respectively). Lock-Free Reads give the best performance, with an almost linear throughput increase with the number of client threads. readers only read the integer representation of the version number, without updating any state. Once this version number is within a cache line, there is no need for coherence messages to be sent to other local and remote caches since it is never written to.

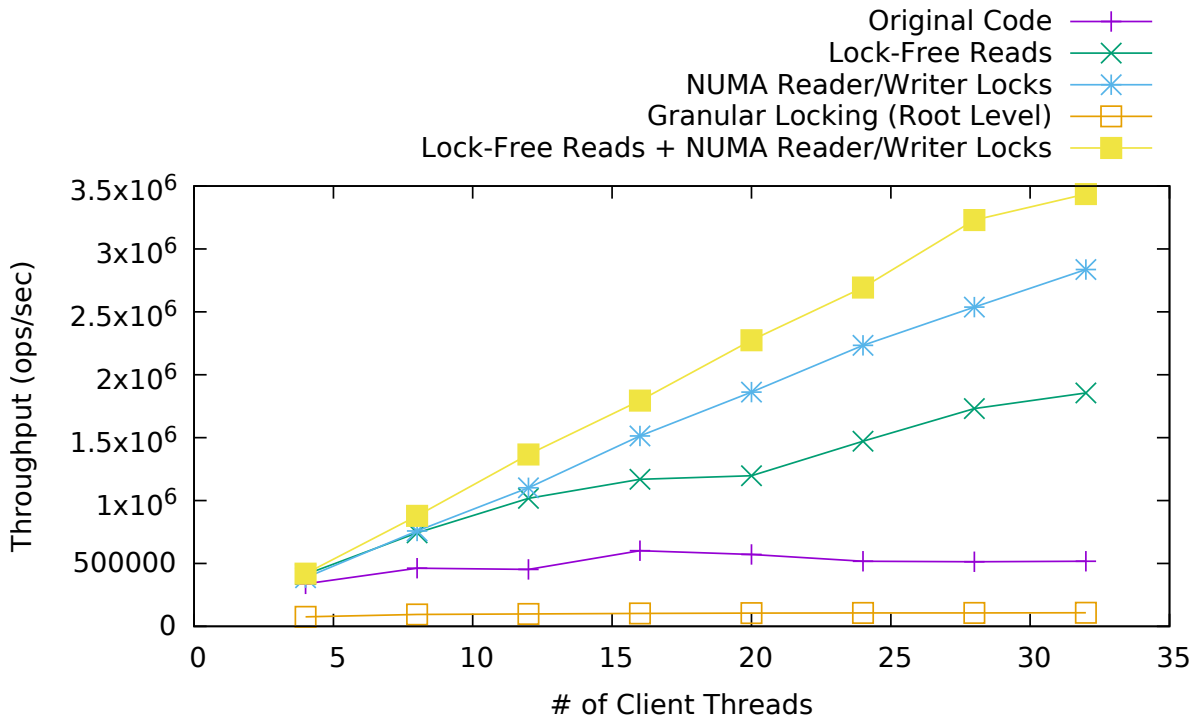


Figure 4.4: Operation Throughput vs Number of Clients, Initial Number of Keys = 1,000,000 , 75% Reads, 25 % Updates

NUMA-aware Reader/Writer locks tries to keep lock acquiring among the local threads, and as long as the local lock is being acquired among the local threads, the state of that lock, and the global lock remains unchanged as well, and causes no coherence messaging to other caches. Replication also only requests Latches from its local latch manager, and so only makes modifications to local data, however, there still can be invalidation messages between private caches of the cores on the same NUMA node. However, everytime the number of client threads requested for driver creation increases by 4, the number of client threads created per NUMA node only increases by 1, due to the driver's even distribution of the client threads among the NUMA nodes. Contention for the locks from the local Latch Manager in the Replicated Tree is much less than the contention for locks from the shared Latch Manager in the Tree from the Original Code. Both Replication and NUMA-aware Reader/Writer Locks have throughput increases which become less pronounced with increasing number of client threads. This is due to the locks having internal state updates (writes) to represent the locked state, even though the lock is being acquired in Reader mode. Granular Locking performs very poorly, because in addition to the larger amount of contention of the original code, contention is further being increased due to a decrease in the number of latches being used for locking by the nodes. Only the latch on the root is used for the locking purposes, and this greatly increases contention, since there are $\sum_{i=1}^{\text{tree height}} \text{fanout}^i$ (fanout = Number of Child Pointers) nodes, in addition to the root, using the same latch, instead of one latch per node in the Original Code.

The results for the workload of 75% Reads, 25% Updates are similar to the Read-only case in terms of relative performance of the modifications when compared to the original code case, except that the improvement in throughput is lower (3× for Lock-Free Reads and 5× for NUMA-aware Reader/Writer Locks). Updates with Replication will be considered separately, since the consistency between the multiple copies of updated nodes has to be addressed as well. For Lock-Free Reads, although read operations use the Lock-Free version number technique, update operations still use the regular PThread Reader/Writer locks. Since there are more updaters in this workload, there are more stalls for other concurrent updaters which access Nodes that are being modified. The readers access the version integer to determine the state of the node, which is only changed by updaters. Read operations will only re-read a node if the node's version number is odd or not equal to what was read initially. readers are not blocked, and do not have to wait to be unblocked when the lock is released. They instead just wait until the node is safe to read by repeatedly checking that its version number is even and unchanged. NUMA-aware Reader/Writer locks perform better than the Lock-Free Read case. The NUMA-aware Reader/Writer lock shares the lock with other threads local to the same NUMA node as the current lock owner causing fewer remote reads for the lock state. Version numbers are stored in the nodes for the

Lock-Free Read operations. There may be many remote references for a version number of a node, if the requesting thread is executing on an NUMA node that is different from the one which contains the node. Granular Locking performs badly once again for the same reasons as for the Read case, except that there is more stalling of operations due to exclusive Write locking of the PThread Reader/Writer lock for updates to Nodes. The combination of Lock-Free Reads and NUMA-aware Reader/Writer locks for updaters gave the best performance (7× more operations than the Original Code), since it combines the benefits of Lock Freedom in the readers and the NUMA-aware locking for updaters.

4.5 Retries for Lock-Free Reads with Increasing Update Proportions in the Workoad

One issue with Lock-Free Reads is the necessity of re-reading a Node or restarting the traversal operation if the version number checks on the node before and after do not match, or if the version number of the node is odd (indicating an Update is in progress). As the ratio of updates in the workload increases, it is expected that Reads will have to re-read Nodes or restart Tree traversal more frequently. Fig. 4.5 shows the results for a tree with initial number of keys = 1,000,000 and 4 client threads.

The number of retries increases almost linearly with the Update proportion in the workload ratio, and an increase in the number of threads causes a large increase in the number of retries for the same increase in workload ratio. However, considering the results are for an intial tree size of 1,000,000 keys with a read-heavy workload of 75 % Reads, 25 % Updates and 10,000 operations (7,500 Reads, 2,500 Updates), the number of retries per read operation is less than 5 (1% of the 7,500 read operations).

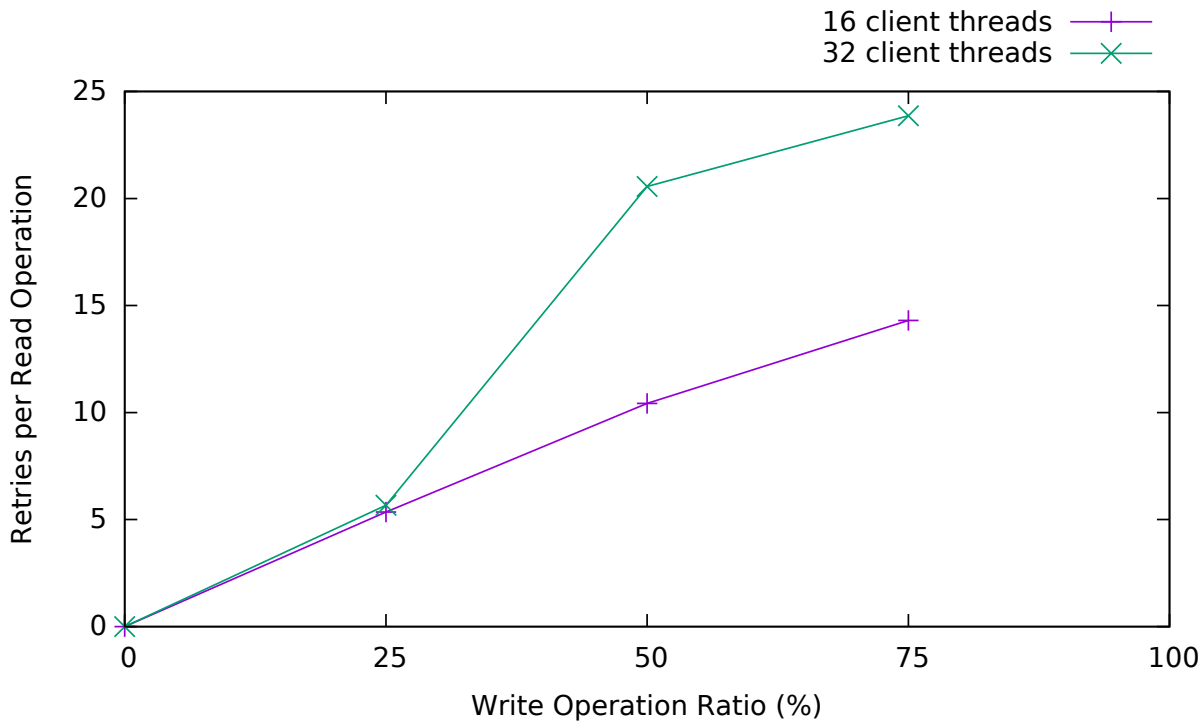


Figure 4.5: Average Number of Retries per Read Operation for Varying Workload Ratios

4.6 Operation Throughput for Techniques in Combination with Replication

The modification techniques described and assessed before can also be combined to provide further improvments in throughput. Replication is independent of the other techniques, so it can be combination with all of the other techniques. Lock-Free Reads can be combined with NUMA-aware Reader/Writer locks, and Granularity changes can be combined with all lock-based modifications. The results for a workloads of 100% Reads and 0% Updates are shown in Fig. 4.6.

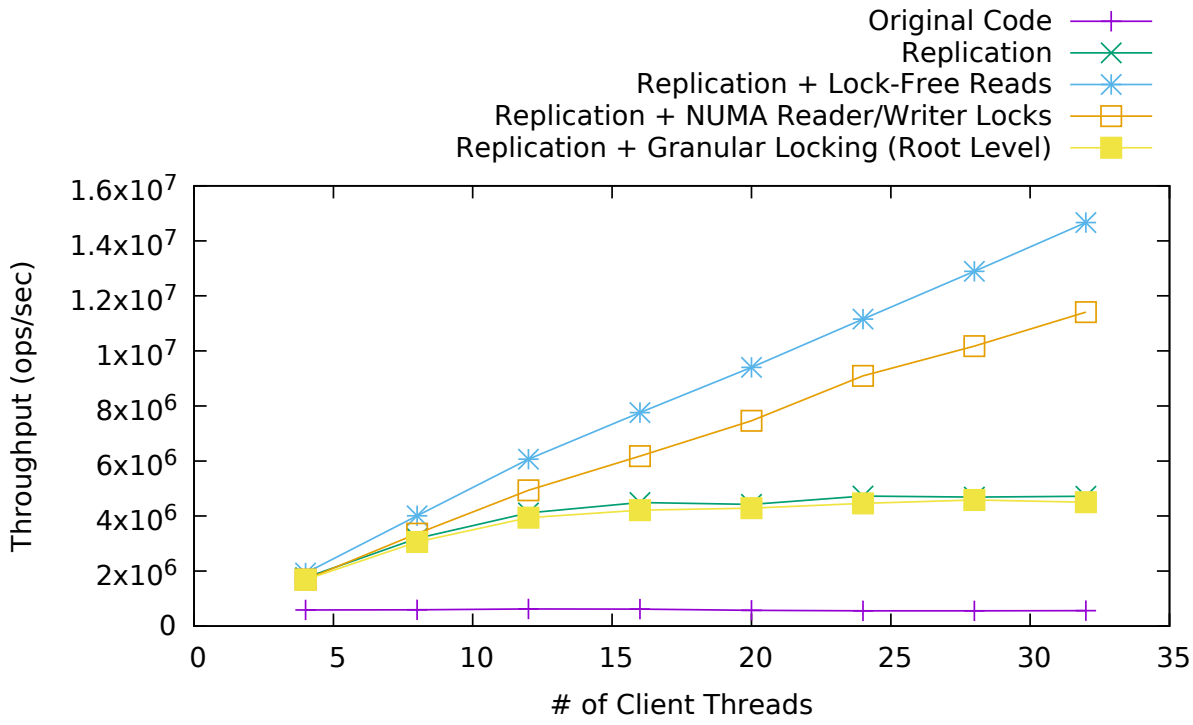


Figure 4.6: Operation Throughput vs Number of Clients, Other Techniques with Replication, Initial Number of Keys = 1,000,000 , 100% Reads, 0% Updates

The combinations of the various other techniques with replication provide improvement over just replication alone (similar to the results of Fig. 4.3), and improvement over the techniques without replication (about 200,000 more operations per second). Contention for locks is now only among the clients associated with the replica of a Node. For NUMA-

aware Reader/Writer Locks, the need for a separate global lock now becomes unnecessary, since the locks will only be requested from the same NUMA node. However, when there are many readers waiting to acquire the lock, the state of the lock is not updated. The lock is instead passed to a subsequent Updater, without a change of state, which would require a memory update. This avoidance of a memory update improves the performance of the NUMA-aware Reader/Writer Lock over the PThread Reader/Writer Lock, used by the Replication technique alone. NUMA-aware Reader/Writer Locks give less improvment than Lock-Free Reads due to the threshold reader count number forcing a local and global lock release after a certain number of readers have acquired the lock on the NUMA node. Lock-Free Reads still provide the benefit of version reads without memory updates for readers, but adding Replication eliminates remote memory reads, which occured in the non-replicated version. Lock-Free Reads therefore perform the best for this workload. Adjustment of the locking granularity to be at the root level now becomes comparable to the other schemes, because it benefits from the local repository of locks, and due to the driver evenly distributing the client threads among NUMA nodes, the contention in each local tree stays very low (up to 8 client threads per node when the total number of clients = 32 threads). However, when the thread count increases per node, this technique starts to perform slightly worse than the others, as seen from the number of client threads greater than 24.

Updates become a lot more complicated with replicated Tree nodes, since an Updater has to make its changes to all versions of the Tree node on each NUMA node. There can be various forms of the update operation, based on the number of copies updated before returning, and the state of the node that other operations see before, during and after the update has completed. The more copies which need to be updated before the operation returns, the longer the operation would take. When considering the number of copies to update, there are two extremes, all or one. Fig. 4.7 shows the throughput results for only local Tree node updates and for updating all Tree node copies for a workload of 75% Reads and 25% Updates.

There is a large difference between the two versions of the update operation, with the local insert performing 83× more operations per second than strongly consistent inserts with 32 client threads. However, local inserts gives no consistency guarantees, for operations originating from any NUMA node. The alternative gives the strongest consistency but is also the slowest, with the Original Code even providing an order of magnitude more of throughput in operations. Between these two extremes, there is a level of eventual consistency that can give better throughput results but result in some stale reads. Finding such an update model was not considered in this work and is discussed in the Future Work section of Section 5.1. For all further experiments involving replication and updates, the

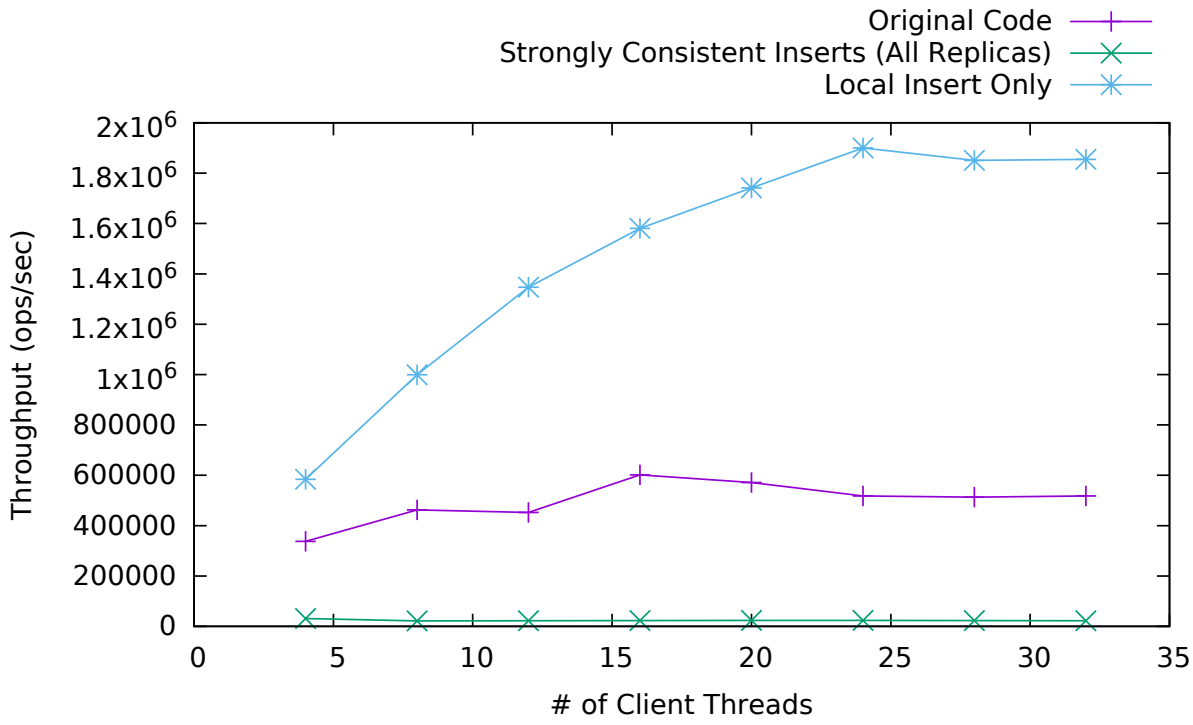


Figure 4.7: Comparison of Throughput, Workload = 75% Reads, 25% Updates, 4 - 32 Client Threads, Local Insert Only vs Updating of All Replicas Immediately

local update model is used. Deriving similar results for the alternatives can be obtained by altering the results by the percentage difference between the techniques.

The results for a workload of 75% Reads, 25% Updates for the local insert model and the various combination of techniques with replication is shown in Fig. 4.8.

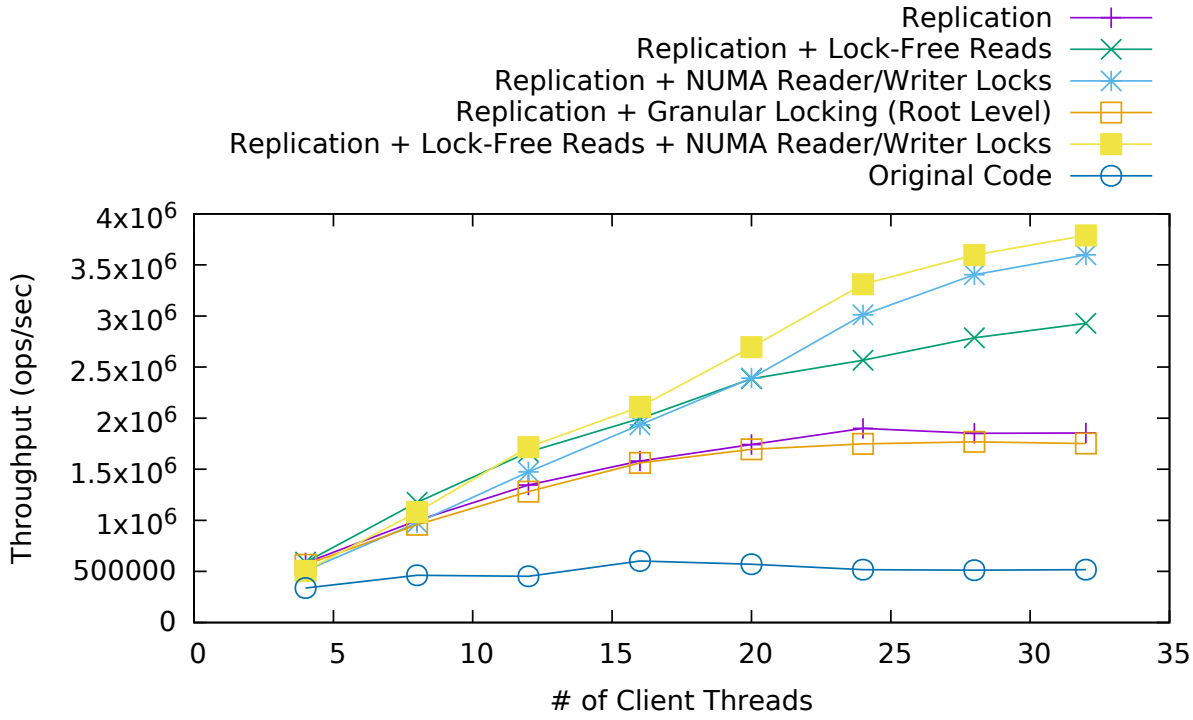


Figure 4.8: Operation Throughput vs Number of Clients, Other Techniques with Replication, Initial Number of Keys = 1,000,000 , 75% Reads, 25% Updates

Chapter 5

Conclusions & Future Work

This thesis explores the problem of creating an in-memory B-Tree optimized for a NUMA System, by considering four techniques — Lock-Free Reads, NUMA-aware Reader/Writer Locks, Granularity changes in locking (at each level) and Node/Lock Replication. NUMA-aware Reader/Writer/ Locks have been introduced previously by Calciu *et al.* in [14]. Node/Lock Replication for B-Trees has been considered by various works for distributed systems [39, 40, 44, 19, 4], but this thesis introduces its use in a NUMA system. Varying the granularity of locking also has been considered by previous works [30, 24, 41], but thesis considers it by itself and in combination of replication for B-Trees used in NUMA systems. Lock-Free techniques have been considered for various datastructures [11, 64, 26, 62, 64, 23, 13, 12, 25, 31, 34, 60], but this thesis introduces a version of the technique that uses odd/even versioning of B-Tree nodes to detect concurrent updates. The aim of the work presented was to find the combination of techniques that provides the best overall operation throughput improvement for read-only and read-heavy workload types over a NUMA oblivious B-Tree implementation.

In order to measure the throughput of operations in a NUMA system, a driver program was developed to evenly distribute a specified number of client threads among NUMA sockets, and provide an operation workload for each client. Different versions of the B-Tree were created and used by the driver program by specifying the symbol(s) of the modification(s) required for the testing.

When the separate techniques were compared with the original code, all provided improvement in throughput for Read-only workloads (23×, 9×, 8× for Lock-Free Reads, NUMA-aware Reader/Writer Locks and Replication with 32 client threads respectively). The type of Granular locking introduced in this work performed poorly, due to the in-

crease in contention for a smaller pool of locks. For Read-heavy workloads (75% Reads, 25% Updates) with Replicated nodes, performance was very poor with respect to the original code if strong consistency of the data contained in the tree was enforced, but could be much better if the consistency constraints were relaxed, with the best being Updates being performed locally only.

Combinations of the techniques with Replication provide similar improvement to the techniques applied separately for Read-Only workloads and an improvment over the corresponding technique without replication in all cases (about 200,000 more operations per second). However, the performance with Updates involved is dependent on the consistency model that is required, and if strong consistency is required, the Replicated technique combinations implemented in this work perform much more poorly than the non-replicated counterparts. Granular locking performed well when replication was introduced, since the increase in threads per each socket B-Tree replica is much less when the clients are distributed evenly among the sockets.

A user of the B-Tree would select modifications based on the type of workloads that are expected, and the freshness of data that is required from operations. For Read-heavy workloads that have a fairly large number of Updates in a small period of time, Replication may be avoided, and the use of Lock-Freedom in Reads combined with NUMA-aware Reader/Writer locks for Updates would give good throughput performance. Replication can be used to provide improvements in throughput performance in the Read-only case, but would not be recommended due to implementation complexity and poor performance of updates.

5.1 Future Work

In this thesis, improvement of the throughput of operations of the B-Tree datastructure contained in-memory in a NUMA system for read-heavy workloads was considered, by the use of three techniques — Replication, use of NUMA-aware Reader/Writer Locks and the use of Optimistic Concurrency Control in Reads, through atomic access of a version number per page of the B-Tree. While it was shown that combinations of these techniques improve the performance for various read-heavy type workloads, most techniques could be improved or extended for even better operation throughput performance.

The replication introduced into the B-Tree made copies of each Segment Pool, Page, and Latch Set on each node. There was no focus on space constraints, because the assumption was the system would have a large amount of memory, to hold the replicas. If this is not

the case, then there is room for improvment by considering replication of the components to improve throughput performance, while keeping to space constraints [65, 39, 44, 19, 4]. Deletes also need to be considered for the replicated environment.

Also, replication introduces issues with consistency among the copies of the Pages and the Segments of the tree. Strongly consistent updates were shown to perform poorly compared to updates focused only on the local node. Work can be done to improve updates as it relates to consistency requirements of the clients as well as keeping the overhead of sending the updates to other replicas low for the original client operation.

The Lock-Free technique used for Readers involved the atomic reading of a Node version number before and after accessing information from the Node. This technique can be extended for use with Writers as well [11, 64, 26, 62, 64, 23, 13, 12, 25]. Also, deletes in the original version of the tree do not physically delete the pages, but instead reuse them for new split requests. To support deletes, the version number can be reset when the node has been selected for deletion.

References

[1] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design. *Computer-IEEE Computer Magazine*, 45(2):37, 2012.

[2] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 396–406, New York, NY, USA, 1989. ACM.

[3] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[4] Shah Asaduzzaman and G. V. Bochmann. Distributed B-tree with weak consistency. *unpublished report, see http://www. site. uottawa. ca/∼ bochmann/Curriculum/Pub/2010*, 2010.

[5] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[6] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, March 1977.

[7] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal*, 5(4):264–275, December 1996.

[8] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent Cache-oblivious B-trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 228–237, New York, NY, USA, 2005. ACM.

[9] Philip A. Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the 2013 international conference on Management of data*, pages 923–928. ACM, 2013.

[10] Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*, pages 227–246, 2011.

[11] Anastasia Braginsky and Erez Petrank. A lock-free B+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 58–67. ACM, 2012.

[12] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010.

[13] Trevor Brown, Faith Ellen, and Eric Ruppert. A General Technique for Non-blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, Orlando, Florida, USA, 2014. ACM.

[14] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *ACM SIGPLAN Notices*, volume 48, pages 157–166. ACM, 2013.

[15] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving Index Performance Through Prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 235–246, New York, NY, USA, 2001. ACM.

[16] 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 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 157–168, New York, NY, USA, 2002. ACM.

[17] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[18] Thomas H. Cormen, editor. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 3rd ed edition, 2009.

[19] Paul Richard Cosway. *Replication control in distributed B-trees*. PhD thesis, Citeseer, 1995.

[20] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. pages 33–48. ACM Press, 2013.

[21] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74. ACM, 2011.

[22] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In *ACM SIGPLAN Notices*, volume 47, pages 247–256. ACM, 2012.

[23] Faith Ellen, Fatourou Panagiota, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, Zurich, Switzerland, 2010. ACM.

[24] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, November 1976.

[25] Mikhail Fomitchev and Eric Ruppert. Lock-free Linked Lists and Skip Lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, St. John's, Newfoundland, Canada, 2004. ACM.

[26] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

[27] Keir Fraser and Tim Harris. Concurrent Programming Without Locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[28] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[29] Wojciech Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. Client-centric Benchmarking of Eventual Consistency for Cloud Storage Systems. 2014.

[30] Goetz Graefe. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)*, 35(3):16, 2010.

[31] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[32] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[33] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *SIGOPS Oper. Syst. Rev.*, 26(2):12–, April 1992.

[34] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[35] Maurice P. Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93 Proceedings of the 20th annual international symposium on computer architecture*. ACM, 1993.

[36] Maurice P. Herlihy, Victor Luchangco, Mark Moir, III Scherer, and N. William. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, Boston, Massachusetts, 2003. ACM.

[37] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[38] Ibrahim Jaluta, Seppo Sippu, and Eljas Soisalon-Soininen. Concurrency Control and Recovery for Balanced B-link Trees. *The VLDB Journal*, 14(2):257–277, April 2005.

[39] Theodore Johnson and Adrian Colbrook. A distributed data-balanced dictionary based on the B-link tree. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pages 319–324, Beverly Hills, CA, March 1992. IEEE.

[40] Theodore Johnson and Adrian Colbrook. A distributed, replicated, data-balanced search structure. *International Journal of High Speed Computing*, 6(4):475–500, 1994.

[41] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision Locks. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 143–147, New York, NY, USA, 1981. ACM.

[42] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[43] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. KISS-Tree: Smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 16–23. ACM, 2012.

[44] Padmashree Krishna and Theodore Johnson. Implementing Distributed Search Structures. Technical report, 1994.

[45] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[46] Vladimir Lanin and Dennis Shasha. A Symmetric Concurrent B-tree Algorithm. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM '86, pages 380–389, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[47] Philip L. Lehman and s. Bing Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981.

[48] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[49] David Lomet and Betty Salzberg. Access Methods for Multiversion Data. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 315–324, New York, NY, USA, 1989. ACM.

[50] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical CLH queue lock, 2006.

[51] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171, April 1994.

[52] Karl Malbrain. A Blink Tree method and latch protocol for synchronous node deletion in a high concurrency environment. Technical report, 2010.

[53] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[54] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[55] McKenney Paul and Sarma Dipankar. Read Copy Update. In *Ottawa Linux Symposium*, pages 338–369, 2002.

[56] A. J. Perlis and Charles Thornton. Symbol Manipulation by Threaded Lists. *Commun. ACM*, 3(4):195–204, April 1960.

[57] Zoran Radovic and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *HPCA '03 Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[58] Muntasir Raihan Rahman, Wojciech Golab, Alvin AuYoung, Kimberly Keeton, and Jay J. Wylie. Toward a principled framework for benchmarking consistency. In *Proceedings of the Eighth USENIX conference on Hot Topics in System Dependability*, pages 8–8. USENIX Association, 2012.

[59] Jun Rao and Kenneth A. Ross. Making B+- Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM.

[60] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

[61] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[62] Afroza Sultana, Helen A. Cameron, and Peter CJ Graham. Concurrent B-trees with Lock-free Techniques. 2011.

[63] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified B-trees and versioning dictionaries. *arXiv:1103.4282 [cs]*, March 2011. arXiv: 1103.4282.

[64] John D. Valois. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM.

[65] Paul Wang. An in-depth analysis of concurrent B-tree algorithms. Technical report, DTIC Document, 1991.

[66] W.W. Weihl and P. Wang. Multi-version memory: software cache management for concurrent B-trees. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, 1990*, pages 650–655, December 1990.