Buffering Accesses to Memory-Resident Index Structures

Jingren Zhou Kenneth A. Ross*

Columbia University
{jrzhou, kar}@cs.columbia.edu

Abstract

Recent studies have shown that cache-conscious indexes outperform conventional main memory indexes. Cache-conscious indexes focus on better utilization of each cache line for improving search performance of a single lookup. None has exploited cache spatial and temporal locality between consecutive lookups. We show that conventional indexes, even "cache-conscious" ones, suffer from significant cache thrashing between accesses. Such thrashing can impact the performance of applications such as stream processing and query operations such as index-nested-loops join.

We propose techniques to buffer accesses to memory-resident tree-structured indexes to avoid cache thrashing. We study several alternative designs of the buffering technique, including whether to use fixed-size or variable-sized buffers, whether to buffer at each tree level or only at some of the levels, how to support bulk access while there are concurrent updates happening to the index, and how to preserve the order of the incoming lookups in the output results. Our methods improve cache performance for both cache-conscious and conventional index structures. Our experiments show that buffering techniques enable a probe throughput that is two to three times higher than traditional methods.

*This research was supported by NSF grants IIS-01-20939, and EIA-00-91533.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

1 Introduction

Recent advances in the speed of commodity CPUs have far outpaced advances in memory latency. Main memory access is therefore increasingly a performance bottleneck for many computer applications, including database systems [4, 5]. As random access memory gets cheaper, it becomes affordable to build computers with large main memories. More and more query processing work can be done in main memory. Recent database research has demonstrated that memory access is becoming a significant — if not the major — cost component of database operations [4, 5].

We focus on memory-resident tree-structured indexes. There are many applications of such indexes, since they speed up access to data. The applications we address involve access patterns in which a large number of accesses to an index structure are made in a small amount of time. An example would be processing stream data from a large number of mobile sensors [1, 2]. As each sensed point arrives, a spatial index is consulted to determine objects in the vicinity of the point. A second example would be an index-nested-loops join in a database system with a memory-resident index on the inner table. The index is probed once for each record of the outer table.

Our high-level goal is to make a bulk lookup substantially faster than a sequence of single lookups. Our proposed solutions place minimal requirements on the index structure. We assume the index is a tree of some kind, and we require access to some statistics such as the average branching factor. We do not place conditions on the node size, or on architectural parameters such as the cache size. We do not assume that each lookup requires an exact match; for example, the technique would apply for "overlap" queries in an R-Tree, or "nearest value" in a one-dimensional ordered tree.

The basic idea is to create buffers corresponding to non-root nodes in the tree. As a lookup proceeds down the tree, a record describing the lookup (containing a probe identifier and the search key) are copied into a buffer of such records. These buffers are periodically emptied, and divided[1] among buffers for the

[1]For some kinds of lookup, an access record may traverse multiple branches of the tree.

child nodes. While there is extra copying of data into buffers, we expect to benefit from improved spatial and temporal locality, and thus to incur a smaller number of cache misses. Buffering is not used for single-lookup access, and so the index performance for such lookups is unchanged. The optimizer decides whether lookups need to be buffered.

We experimentally study buffering with several kinds of index structures, including B$^+$-Trees [9], R-Trees [10], and CSB$^+$-Trees [16]. Of these, the CSB$^+$-Tree was designed with main-memory performance of single lookups in mind. We repeat our experiments on three different architectures. The design parameters include whether to use fixed-size or variable-size buffers, whether to buffer at each tree level or at only some of the levels, how to support bulk access while there are concurrent updates happening to the index and how to preserve the order of the incoming lookups in results. We provide architecture-sensitive guidelines for choosing buffering parameters.

Our results show that conventional index structures have poor cache miss rates for bulk access. More surprisingly, even cache-sensitive algorithms such as CSB$^+$-Trees also have moderately high cache miss rates for bulk access, despite their cache-conscious design. Our new search algorithms yield speedups by a factor of three over conventional B$^+$-tree index lookups, and by about a factor of two over CSB$^+$-tree and R-tree index lookups. Our algorithms can gracefully handle a moderate number of updates mixed with a large number of searches. Our algorithms can achieve more than a factor of two improvement in throughput, while retaining a response time guarantee of less than one second for each probe.

Our methods improve cache performance for *both* cache-conscious and conventional index structures. Interestingly, with buffering, conventional index structures can achieve cache performance that is comparable to cache-conscious index structures for batch lookup. Our algorithms can be implemented in current commercial database systems without significant changes to existing code.

The rest of this paper is organized as follows. We discuss related work in Section 1.1. We briefly discuss hierarchical memory systems in Section 2. We survey cache-conscious index structures and demonstrate why they are suboptimal for batch lookups in Section 3. In Section 4, we present different buffering techniques and detailed data structures. We derive guidelines to choose where to place buffers in Section 5. In Section 6, we present detailed experiments and validate our algorithms. We conclude in Section 7.

1.1 Related Work

The Y-tree [11] adds a special lookup structure (heap) in an index tree node to avoid random disk seeks; it allows high volume fast insertions, but is slow for lookups. A buffering idea similar to ours was proposed for I/O optimization of non-equijoins in [18]. While there is a high-level similarity between various levels of the memory hierarchy, there are important aspects of CPU architectures, such as TLB-misses and cacheline prefetching, that make the details of main-memory techniques more involved. Also, the algorithm of [18] requires processing all accesses through each level in turn, in a top-down breadth-first fashion. Such traversal behavior is clearly inapplicable for processing continuous data streams, because one never has a "complete" probe input. Finally, our techniques allow index updates to be interleaved with probes, which is not considered by [18].

XJoin is an operator for producing the join of two remote input streams [17]. Our techniques are different from XJoin in that we're joining one remote stream with a local relation that has a local index. [5] considers the impact of cache misses and TLB misses on a radix-clustered hash join. Their algorithms do not involving buffering, and apply only to equijoins.

We focus on applications that are probe-intensive, with a relatively small number of updates. In situations with large numbers of updates per second, bulk update operations [8, 14] may be appropriate.

2 Memory Hierarchy

Modern computer architectures have a hierarchical memory system, where access by the CPU to main memory is accelerated by various levels of cache memories. Cache memories are designed based on the principles of *spatial* and *temporal* locality. A *cache hit* happens when the requested data is found in the cache. Otherwise, it loads data from a lower-level cache or memory, and incurs a *cache miss*. There are typically two or three cache levels. The first level (L1) cache and the second level (L2) cache are often integrated on the CPU's die. Caches are characterized by three major parameters: capacity, cacheline size, and associativity. Latency is the time span that passes after issuing a data access until the requested data is available in the CPU. In hierarchical memory systems, the latency increases with the distance from the CPU.

Logical virtual memory addresses used by application code have to be translated into physical page addresses. The memory management unit (MMU) has a translation lookaside buffer (TLB), a kind of cache that holds the translation for the most recently used pages. If a logical address is not found in the TLB, a *TLB miss* occurs. Depending on the implementation and the hardware architecture, TLB misses can have a noticeable impact on memory access performance.

Memory latency can be hidden by correctly prefetching data into the cache. On some architectures, such as the Pentium 4, common access patterns like sequential access are recognized by the hardware, and hardware prefetch instructions are automatically

	Pentium 4	Pentium 3	UltraSparc
OS	Linux 2.4.17	Linux 2.4.17	Solaris 8
CPU speed	1.8 GHz	1 GHz	296 MHz
Main-memory size	1 GB	512 KB	1 GB
Memory type	RDRAM	DDR	DRAM
L1 cache size	16 KB	16KB	16KB
L1 cacheline size	64 bytes	32 bytes	32 bytes
L2 cache size	256 KB	256 KB	1 MB
L2 cacheline size	128 bytes	32 bytes	64 bytes
TLB entries	64	64	64
L1 miss latency	18 cycles	7 cycles	10 cycles
L2 miss latency	276 cycles	101 cycles	74 cycles
TLB miss latency	46 cycles	5 cycles	51 cycles
Hardware prefetch	Yes	No	No

Table 1: System Specifications

executed ahead of the current data references, without any explicit instructions required from the software.

Table 1 lists the specifications of our experimental systems.[2] The TLB miss latency of the Pentiums is smaller than that of the UltraSparc. In our experiments, L1 miss latency is so low that the impact of L1 misses on the total performance is insignificant. For the rest of the paper, we will focus on the impact of TLB misses and L2 misses. We use Sun's `CC` compiler on the UltraSparc and GNU's `gcc` on the Pentiums.

3 Index Structures

Recent studies have shown that cache-conscious indexes such as CSS-Trees [15] and CSB$^+$-Trees [16] outperform conventional main memory indexes such as B$^+$-Tress. The key idea is to eliminate all (or most) of the child pointers from an index node to increase the fanout of the tree, and to choose a node size of the order of the cache line size. Multidimensional cache-conscious indexes such as CR-Trees [13] use compression techniques to pack more entries in a node. These techniques effectively reduce the tree height and improve the cache behavior of the index. Other indexes such as pB$^+$-Trees [6] and fpB$^+$-Trees [7] use prefetching to create wider nodes and reduce the height of the B$^+$-tree. Different index structures propose different optimal node sizes.

All these techniques focus on better utilization of each cache line and optimize the search performance of a single lookup. None has exploited cache spatial and temporal locality between consecutive lookups. Cache memories are typically too small to hold the whole index structure. If probes to the index are randomly distributed, consecutive lookups end with different leaf nodes, and almost every lookup will find the corresponding leaf node is not in the cache. By loading the leaf node into cache, capacity cache misses occur to evict other leaf nodes out of cache. The upper level nodes are more frequently accessed, and thus less likely to be the victims. However, the evicted leaf nodes are required again by subsequent lookups, and have to be loaded again in the future. This *cache thrashing* can have a severe impact on the overall performance.

[2]We use LMbench [3] to measure the L1, L2 and TLB miss latency on different systems.

	B$^+$-tree	CSB$^+$-tree	R-tree
Key type	32-bit floating point		
Node Size	4 KB	128 B (cacheline)	4 KB
Index (Leaf) Node Capacity	510 key-pointer (510 key-RID)	30 keys (14 key-RID)	204 MBR-pointer (204 MBR-RID)
Indexed Items (Inner)	5 million keys		1 million unit rectangles
Query Entries (Outer)	5 million random searches		1 million random 2x2 rectangles
Query Type	exact match		overlap
Bulkloaded	75%		75% using [12]

Table 2: Index Setup

We illustrate index node cache thrashing by implementing an index nested-loop join using a B$^+$-tree, a CSB$^+$-tree, and an R-tree, as shown in Table 2. The result of all three joins are pairs of outer RID and inner RID of matching tuples. Subsequent tuple reconstruction is equal for all algorithms, and we do not include that in our comparison. We measure the elapsed time to compute the join on a Pentium 4 system.[3]

We explicitly measure the number of L2 cache misses in each experiment. Modern machines have hardware performance counters to measure such statistics without any loss in performance.[4] In the graphs we show the "cache miss penalty" as the total time taken if each cache miss takes exactly the measured cache miss latency. This is an approximation, because sometimes multiple cache misses can be processed concurrently by the memory subsystem. Note also that some computation may be happening during the cache miss wait time, and so the "cache miss penalty" bar on the graph may also hide some concurrent computation. Despite these caveats, the graph will give us a sense of the contribution of cache miss latency to the overall response time, and thus the opportunity for improvement by processing accesses in batches.

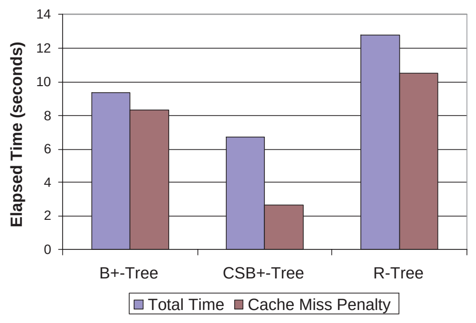


Figure 1: Cache Thrashing Impact

Figure 1 shows the results for the three index structures. The CSB$^+$-tree does improve cache performance over the conventional B$^+$-tree. However, both suffer significantly from index node thrashing. For the CSB$^+$-tree, more than 30% of the time can be attributed to index node cache misses. The B$^+$-tree and

[3]The other two systems have similar behavior.

[4]We use the Intel VTune Performance Tool on Pentium PCs and use Solaris native hardware performance counters on the UltraSparc machine.

R-tree suffer even more. There is thus an opportunity for improving the cache behavior.

4 Buffering Techniques

Given an index tree, we first organize nodes into *virtual nodes*. Virtual nodes are the units of buffering. A virtual node may contain a single tree node, or a node together with several levels of its descendants. Having larger virtual nodes corresponds to doing less buffering. For now, we assume that the mapping from tree nodes to virtual nodes is given, and defer the discussion of how to choose virtual nodes until Section 5. Unless we state otherwise, a "node" is a "virtual node" unless we explicitly call it a "tree node".

Each node, excluding the root node, is associated with a corresponding buffer. This buffer contains batched queries (records describing the probes) that have been passed down from higher levels, and have yet to be divided according to the keys in the current node. Thus, the buffer contains the "input" for a node. Batch index lookups begin with the root node. The query stream is passed through the root node, and distributed among the buffers of its child nodes, according to the usual traversal protocol of the index structure. We continue the process until query entries reach the leaf nodes and produce the join result.

What is the point of buffering? All it seems to do is slow down probes as they progress through the tree! The point of buffering is that it *increases temporal and spatial locality*. When an index node is used for traversal, instead of being used for just one probe, it is used for many. As a result, the memory overhead for bringing index nodes into the cache is smaller, amortized over many accesses. Since this memory latency is a large component of the total cost, buffering can improve the *throughput*, i.e., the rate at which the probe stream is processed.

At any point in time, a node can be *flushed*, at which time its buffer is emptied and the records are distributed to the node's children. We'll describe choices for *when* a buffer should be flushed below. We may also flush the entire tree, which corresponds to flushing all nodes in turn. When the entire tree is flushed, parents must be flushed before their children. A preorder traversal is used to facilitate cache reuse between a parent and its children. We allow the probe stream to contain explicit *flush* instructions, which force a flush of the entire tree. A finite stream always ends with such a flush instruction.[5]

4.1 Fixed-Size Buffer

In this method, each node explicitly has a corresponding fixed size buffer. The buffers are created before any probes happen and stay until all the probes are finished. Figure 2(a) shows a fixed-size buffered tree with the buffer size of 3.

We divide up accesses until some child node buffer becomes full. When a child node buffer is full, we flush its buffer by dividing its accesses recursively among its children, and then return to continue processing the parent. We may have to interrupt the child node's flush temporarily if one of *its* children's buffers becomes full, and so on. When accesses reach the leaves, output records are produced. A partially-full buffer may be flushed when the whole tree is being flushed. In Figure 2(a), each node has a pointer pointing to its buffer. The root node A continues processing the query accesses until the fourth query access when the buffer of node B is full. We flush the buffer by distributing its accesses between the children C and D, and return to process the root node A.

The choice of buffer size can be important. As the buffer size increases, search performance improves, due to avoiding cache misses. On the other hand, larger buffers require more memory, and all buffers must exist for the duration of the probe stream. Considering both performance[6] and memory requirements, for the remainder of this paper we choose a buffer size equal to the (virtual) node size. Thus, the total size of the buffers is roughly equal to the size of the index.

The fixed-size buffering technique eliminates a large number of cache misses, but not all of them. Intermediate index nodes may still be loaded into the cache multiple times. Nevertheless, we have some degree of control over the memory usage by adjusting the size of the buffers — most of the benefit of buffering is derived from the initial portion of the allocated memory.

4.2 Variable-Size Buffer

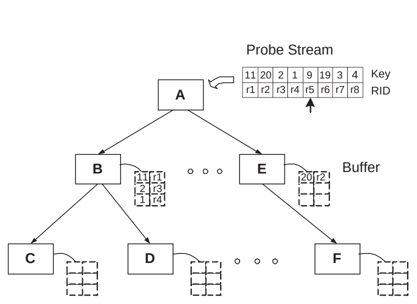
In this method, buffers do not have a fixed capacity, but can grow arbitrarily large. A node is flushed only in response to an explicit tree-flush operation. We completely eliminate index node cache thrashing by loading each node *exactly once*.

Figure 2(b) shows a variable-size buffered tree. Buffers are dynamically created. There is no pointer necessary from a tree node to its buffer. The root node A continues processing all the query accesses in the outer relation. Buffers for nodes B and E are created as required. Then the buffer of node B gets flushed, buffers for nodes C and D are created and query accesses are distributed between the buffers. When this step finishes (not shown in the figure), buffers for nodes C and D get flushed and generate the join results.

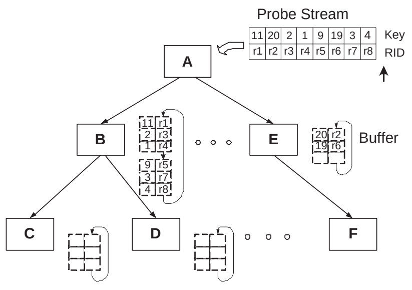
A buffer is made up of a linked list of fixed-size segments. To flush a node, a short array of size equal to the number of its children is created. Each entry contains one pointer which points to the current not-yet-full segment in the buffer segment list and a slot number which indicates the next available slot in that

[5]An infinite stream should be flushed regularly; otherwise some access could sit in a rarely used buffer forever.

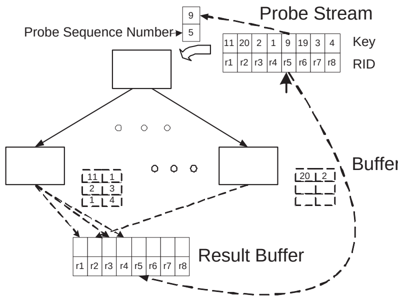
[6]Experiment omitted due to lack of space.



(a) Fixed-Size Buffered Tree



(b) Variable-Size Buffered Tree



(c) Order-Preserving Buffering

Figure 2: Buffered Tree

buffer. This design makes sure that storing one query item involves only two cache lines. Since the array is frequently accessed, there is a high probability that it resides in the cache. Thus, storing one query item incurs roughly one cache miss.

Buffers get smaller by roughly the branching factor as one descends a level in the tree. Also, bias in the probe distribution may make some sibling buffers larger than others. Without any prior knowledge, we estimate that query entries are randomly distributed among different branches. Therefore, we choose a buffer segment size that is slightly larger than the total input size divided by the number of nodes at the given level. (We could derive a more accurate number by just dividing the parent's total buffer size by the number of children, but that would prevent the segment re-use optimization described below.) In the event of a biased probe distribution, we may use more memory than necessary, but overall time performance is not sensitive to the segment size.

After processing a node, we go through each buffer segment list, load the corresponding node and process the query items from the *last* buffer segment, because the last buffer segment has a higher probability of residing in the cache. Because the buffer list is cyclic, we are able to traverse all segments in the list.

When implementing variable-sized buffering, we can take advantage of the fact that no pair of sibling nodes will have active child buffers at the same time. The memory allocated to one node's buffer can be re-used later for flushing its siblings. In Figure 2(b), buffers for nodes C and D can be re-used for node F, for example. Not only does this save overhead for allocation and deallocation of memory, but it also reduces cache thrashing for buffers. For buffers smaller than the L2 cache, the buffer is likely to remain in the cache as we move from a node to its sibling.

For variable-size buffering, the memory requirement is approximately $\frac{B}{B-1}I$, where B is the branching factor of the index, and I is the size of the input.

4.3 Order-Perserving Buffering

Simple buffering methods do not guarantee that the output order matches the probe order. For query execution plans that care about the result order, we propose order-perserving buffering as shown in Figure 2(c). A result buffer of outer RIDs and inner RID-lists has size equal to the number of probes between flushes. The RID-lists are initialized to be empty. Before a probe traverses the root, we assign the probe a sequence number and copy the RID into the corresponding result buffer entry. In the buffered tree, pairs of a searching key and a probe sequence number describe probes and are units of buffering. When a match is found, the matching RIDs are appended to the result buffer entry corresponding to the probe sequence number. In the event that at most one match per probe is possible (e.g., an equality probe stream on a key attribute), we can optimize the implementation by using an RID rather than a RID-list for matches. As we shall see in the experiments, the overhead to maintain the probe order is small.

4.4 Updates

We now describe variant algorithms that allow concurrent updates to the index. The main insight is to allow updates to be buffered along with the probes. Probes and updates that happen after an update operation U are guaranteed to see an index in which U has been applied to the index. We will make changes to the index update methods, but the changes will be small.

Update operations are buffered just as probes are buffered, whether we use the fixed-size buffer technique or the variable-size buffer technique. When an update is received at the root node, an immediate flush operation happens at the root. Whenever an update is passed down to a child buffer, that buffer is also flushed. We always distribute the probes ahead of the updates. When a leaf node is processed, all of the probes generate output results, followed by the update operation that may modify the node. If the modification does not propagate beyond the leaf then we are done, and we return to processing the input stream.

Depending on the kind of data structure used, there may be different kinds of node modification that can result from an update. For fixed-size buffering the changes we require to the update methods for the index are simple, namely:

- Before a node is modified, call the flush operation on that node.
- When a node is created, create a new buffer too.
- When a node is destroyed, deallocate its buffer.

Modifications that occur only along the root-to-leaf path that the update followed do not need to be explicitly flushed again, because flushes were invoked on the way down. Access requests may remain in the buffers for unmodified nodes that are not on the update's path. Figure 3 shows an example traversal. If the update path is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, then the buffers associated with nodes 2, 3, and 4 are flushed on the way down. If a modification results in a change to node 5, then that node's buffer is also flushed before the modification.

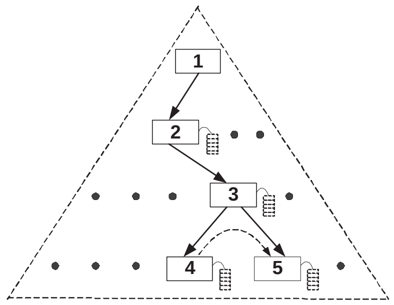


Figure 3: Buffering with Updates

For variable-size buffering, we dynamically use buffers for different nodes. At any given level of the tree, there is only one active set of buffers. As a result, in order to flush an update without requiring additional buffers to be allocated, we need to flush the entire subtree below the appropriate child of the root node. Also, when the root node is changed, we need to recreate a buffer segment list and make entries point to the right buffers. We experimentally measure the performance of each buffering technique with concurrent updates in Section 6.

4.5 Throughput versus Response Time

Two key parameters of applications are the *response time* and the *throughput*. Buffering achieves high throughput at the cost of relatively slow response time. For query plans in which response time is also important, we flush the buffered tree more frequently to improve the response time. Section 6.7 demonstrates the trade-off.

5 Virtual Nodes

We now address the question of where to place buffers. We group tree nodes into "virtual nodes," as illustrated in Figure 4. Each virtual node has an associated buffer, with the exception of the root virtual node. By grouping multiple levels of a subtree into a virtual node, we reduce the amount of buffering that is done.

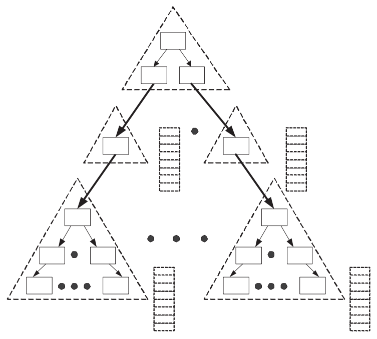


Figure 4: Virtual Nodes

Every buffer level incurs the cost of copying the data to a buffer and rereading it to process the next level. If this copying cost is significant, then there is a potential for "too much" buffering. On the other hand, if a virtual node is larger than the L2 cache, then dividing the accesses from one buffering level to the next may incur a large number of cache misses. Thus there is also a potential for "too little" buffering. (The unmodified index methods can be seen as examples of "too little" buffering.)

Ideally, one would like to come up with a cost-based way of estimating the expense of different buffering configurations. We illustrate such an approach based on the L2 cache size in Section 5.1. Nevertheless, architectural issues such as the TLB miss behavior, and whether the underlying architecture applies hardware prefetching, have a noticeable impact on the performance of any given configuration. We discuss these issues in Section 5.2.

5.1 Virtual Nodes Based on the Cache Size

Let N be a virtual node of height d, i.e., the full d-level subtree of a node in the index. Let L denote the cache line size, and B the average branching factor of the index. The branching factor of node N is B^d.

An active component of an index is one that is potentially referenced during a lookup operation. For example, if a B-tree node was 75% full, then three quarters of the node would be considered active, and one quarter of the node would be inactive. Since only active components cause cache misses, we can discount inactive components from our memory requirement calculations. If I is an index node, then we say that the *effective size* of I, written $|I|_e$, is the total size of the active components of I.

Let $|N|_e$ denote the total effective size of N, in bytes. If e is the average effective size of (tree) nodes in the index, then $|N|_e = \frac{B^d-1}{B-1}e$. For most indexes, e is a small multiple m of B; for a B-tree, m corresponds to the size of a pointer plus the size of a key.

If N does not include any leaves of the index, then the amount of L2 cache needed to avoid all (capacity) cache misses is $|N|_e + (B^d + 1)L$.

We need to store the entire node N and the tail cache lines of B^d buffers in the cache, as well as one cache line for the tail of the input buffer. (Of course, even with this much cache memory, we will encounter conflict misses and compulsory misses.) If N is a leaf node of the virtual-node tree, i.e., the lowest index nodes in N are all leaves, then the amount of L2 cache needed to avoid all (capacity) cache misses is $|N|_e + 2L$.

Leaf nodes are different from internal nodes with respect to buffering, because their "output" is not buffered. We need just one cache line to store the input buffer tail, and one cache line to store the tail of the output result. (For index structures that can be unbalanced, an intermediate formula between these two extremes may apply.)

In either case, the memory needed is roughly a multiple of B^d. We choose the largest d such that the memory needed is less than the size of the L2 cache.[7] Further, since a cache miss is generally more expensive than a copy operation on cache-resident data (and is likely to become even more so in future architectures), the more important criterion is minimizing cache misses.

Example 5.1 *A B^+-tree with a 4KB node size, a 75% occupancy factor, a 4 byte key, and a 4 byte pointer has $B = 375$. For a two-level internal virtual node, the total memory requirement is over 10MB on a machine with 64 byte cache lines. Since it is unlikely that the L2 cache would be that large on a current commodity machine, one concludes that a single level for each virtual node would be optimal. On future machines with a sufficient large L3 cache, the conclusion may be different. (When the root node may have significantly fewer keys than other nodes, one should also consider special cases for combining the root node with its children when the root branching factor is small.)*

Example 5.2 *For a CSB$^+$-tree with a 64 byte node size, a 75% occupancy factor, a 4 byte key, and a 4 byte pointer, $B = 11$. For a four-level virtual node, the total memory requirement for internal nodes is over 1MB on a machine with 64 byte cache lines, while for three levels it is approximately 90KB. For leaf-level virtual nodes the requirements are approximately 90KB and 9KB for four-level and three-level virtual nodes, respectively. The appropriate choices depend on the L2 cache size.*

There may still be many configurations that conform to the cache-based guidelines for choosing the size of virtual nodes. If an index has three levels, and virtual nodes containing two levels are recommended by the cache analysis above, then there are two different ways of partitioning the levels into virtual nodes. We argue that it is unlikely to make a significant difference, in terms of cache misses, which of the two alternatives is chosen. In both cases there is one level of buffering, and the overhead is essentially the same. Nevertheless, there may be a difference due to TLB misses, which we discuss in Section 5.2.

Our configurations are based on the average branching factor of the index. Our experience indicates that the results are not sensitive to local variation in the node branching factor.

[7]If there is a moderate amount of L2 cache space left over, it is conceivable that one could benefit by adding *part* of the next level of the index to the virtual node. We implemented this idea, but the increase in the complexity of the code to traverse the virtual node offset any potential performance gains.

5.2 TLB Misses and Prefetching

We now analyze the TLB behavior of variable-size buffering.[8] The TLB will contain one entry per page of data. To calculate how many TLB entries we need, we need to use the full size $|I|$ for an index node rather than the effective size, since we cannot assume that all inactive parts of a node are both contiguous and larger than a page. The number of TLB entries required for the index is approximately $\frac{B^d-1}{B-1}\lceil |I|/P \rceil$, where P is the virtual memory page size. (This is approximate, because alignment issues may result in additional TLB entries.)

For leaf nodes just 2 additional TLB entries are required. The real problem for the TLB is the buffers for internal nodes. Given sufficiently large buffers, each buffer requires its own TLB entry. For internal nodes, that means B^d TLB entries. In almost all practical cases, we expect to encounter TLB thrashing. The exceptions will be cases when $d = 1$ and B is no larger than the TLB size (typically 64 entries on modern machines). Even if the average branching factor is high, the root node may have a small key count, and thus may also be an exception in this sense.

In these exceptional cases, we may expect significantly fewer TLB misses when we choose to buffer at each level. If the cost of a TLB miss is larger than the cost of writing to and reading from a buffer, then we might expect to get better performance from buffering more often than the cache analysis would suggest. For example, in a CSB$^+$-tree with a branching factor of 11 (Example 5.2), it might be preferable to have three initial buffered levels, rather than one buffered virtual node of depth 3.

Some machines, such as the Pentium 4, employ hardware prefetching. The machine automatically recognizes common access patterns, such as sequential access, and prefetches ahead of the current references. In our context, such prefetching reduces the cache-miss cost of buffering, since buffering tends to use predictable access patterns. This is good news in terms of measuring overall speedups. The reduction in cache miss cost amplifies the impact of TLB misses on the overall cost.

[8]The TLB behavior of fixed-size buffering is similar, but slightly more involved, and is omitted due to space considerations.

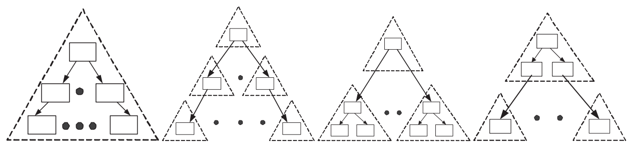


Figure 5: Possible Placement of A 3-Level Node

5.3 Overall Algorithm

We initially run a calibration experiment to determine whether, for the given index, on the given architecture, it is worth partitioning a multiple-level virtual node into smaller ones. For example, Figure 5 shows all possible ways to break up a 3-level node. Partitioning will be beneficial when different ways have similar L2 cache behavior and the TLB miss cost is not negligible. Since the depth of a virtual node is likely to be small, it is feasible to consider all possible ways of breaking up a virtual node, and to determine the split with best overall performance.

We calibrate by simulating each possible buffer placement by probing with a relatively small uniformly distributed input stream, and measuring the performance. We only simulate the part of the index under consideration. Thus, if we are considering whether to split a three-level virtual node that includes the root of the tree, we traverse only the first three levels of the index during the simulation.

In the case of virtual nodes that do not include the root, we choose a sample of such virtual nodes from the index. Each of them is simulated with a number of probes that is large enough to touch almost all tree nodes within the virtual node. The overall performance is calculated as the average over all samples.

As shown in Section 6.2, calibration can typically be performed in less than a second. The results of such calibration experiments can be stored with other database statistics, to avoid redundant work.

An important benefit of calibration, when compared to an analytic derivation of virtual node size, is that it is sensitive to the implementation characteristics of the index. For example, when bulk-loading an index, it is typical to allocate all nodes of a given level contiguously. (If many nodes fit within a page, then many fewer TLB entries would be required to access the nodes.) On the other hand, a tree that has evolved over time by insertions and deletions is much less likely to have siblings close to each other in physical memory. Calibration can measure the actual TLB behavior of the index, which will reflect the degree to which sibling nodes are contiguous. It would be very difficult for an analytic method to quantify this effect, since it depends on low-level memory characteristics. A similar effect occurs when trying to quantify the impact of hardware prefetching on the overall cost.

As we shall see in the experiments, cache misses have much larger effect than TLB misses. Therefore, our TLB calibration is based on the configuration suggested by the cache analysis. Our algorithm for choosing where to buffer proceeds as follows:

1. Perform the cache analysis of Section 5.1, and choose a hierarchy of virtual nodes consistent with having few capacity cache misses.
2. Use the calibration results to determine whether it is beneficial to split a multi-level virtual node. If so, partition the virtual node according to the calibration experiment's recommendation.
3. Recalibrate if the cache analysis changes or significant tree structure changes happen.

We shall validate the choices made by this algorithm in Section 6.2.

6 Experimental Validation

6.1 Methodology

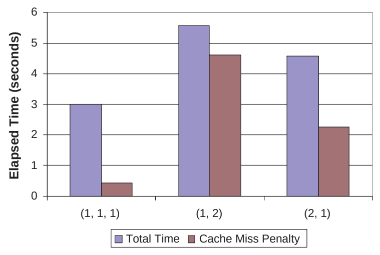
We first evaluate the various index methods with uniformly distributed probes over uniformly distributed data. This is a worst-case workload for all methods, since there is no intrinsic locality in the probe stream. Our choice of uniform data for evaluation is motivated by the fact that designs based on worst-case behavior are not likely to be subject to "surprises" when the data crosses a locality threshold such as the cache size. We compare buffering performance with non-uniform probes in Section 6.5.

6.2 Buffer Placement

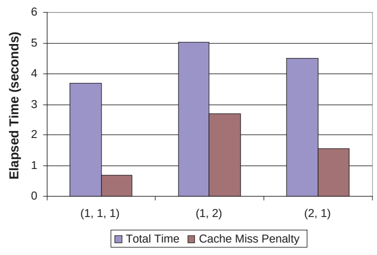
We now assess how well our buffer placement algorithm works in practice in our three candidate architectures. In the first experiment, we validate the accuracy and the importance of the cache analysis of Section 5.1. In the second experiment, we demonstrate the use of calibration to choose an optimal buffer placement. For brevity, we consider just variable-size buffering in this section.

We first implement a 3-level B$^+$-tree with the characteristics of Table 2. The L2 cache is too small to hold either a two-level internal virtual node or a two-level leaf virtual node. Therefore, cache analysis recommends buffering every level. We denote this method as $(1, 1, 1)$. We also implement the two other possible ways to implement buffering: a two-level root virtual node labeled $(2, 1)$, and a two-level leaf virtual node labeled $(1, 2)$. Figure 6 shows the elapsed time and cache miss penalty of performing 5 million randomly generated query accesses on our three architectures. The method $(1, 1, 1)$ has the smallest cache miss penalty in all architectures. The method $(1, 1, 1)$ is confirmed to be the best buffer placement.

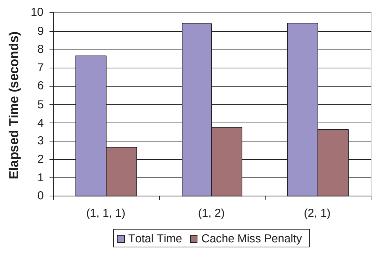
In the second experiment, we implement a 7-level CSB$^+$-tree with a node size of 64 bytes. The Pentiums have a cache line size different from 64 bytes, but we choose to use the same index structure to make a clear comparison. The index also has 5 million keys and is



(a) Pentium 4



(b) Pentium 3



(c) Sun UltraSparc II

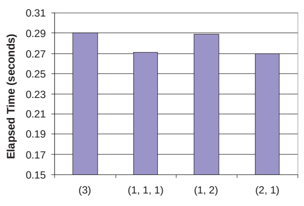
Figure 6: Search Performance of 4 KB Node B+-Tree

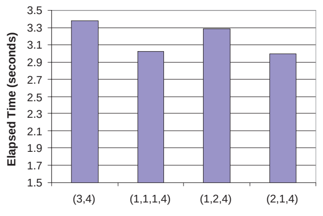
bulkloaded with a 75% occupancy factor. A leaf node contains 5 (key,RID) pairs and an index node contains 11 keys. The root node contains 4 keys.

The cache analysis suggests a 3-level root virtual node and 4-level leaf virtual nodes represented as $(3, 4)$. The branching factor of the 3-level root virtual node is larger than the TLB size. It may thus be beneficial to partition the root virtual node. We can partition the 3-level root virtual node in four ways, as shown in Figure 5. We calibrate the four placements by running 1 million randomly generated query accesses only over the *first three levels* of the index tree. (One could get accurate measurements with fewer than 1 million probes in this case, since there are only about 550 leaves in this 3-level subtree, but even 1 million probes can be performed quickly.)

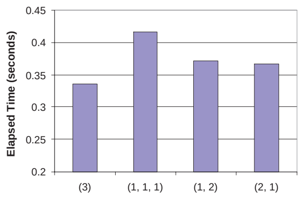
Calibration of 3-Level Nodes

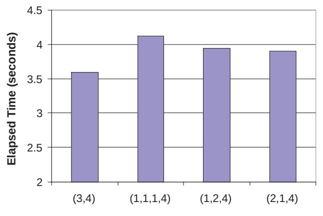
Search Performance of CSB+-Tree



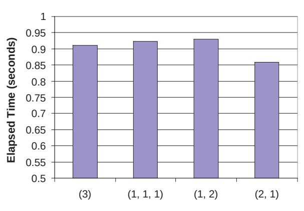


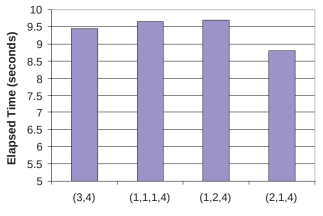
Pentium 4





Pentium 3





Sun UltraSparc II

Figure 7: CSB+-Tree

The left columns in Figure 7 show the results of calibration on three architectures. We measure the elapsed time; all calibrations complete in less than a second. On the Pentium 3, the TLB penalty is small (see Table 1) and method (3) is the fastest because all three levels can fit in the L2 cache. On the UltraSparc, the TLB penalty is high. On the Pentium 4, the TLB cost is also relatively high, and amplified by the use of hardware prefetching to reduce the net cache miss penalty. In either case, the TLB cost is sufficiently large that method (3) is not the fastest. Interestingly, since the root node contains only 4 keys, the total branch factor of the first two levels combined is still smaller than the TLB size. As a result, the method $(2, 1)$ is faster than either $(1, 1, 1)$ or $(1, 2)$.

Is there a need to partition the 4-level virtual node that includes the leaf level? According to our algorithm, we simulate various partitions of the 4-level nodes, and find that the 4-level node performs best. It turns out that all four levels can be processed without TLB thrashing. This is due to the combination of two memory-related effects. First, all siblings in a CSB$^+$-tree are allocated contiguously, meaning that they will usually be on the same page. Second, because the tree is generated by bulk-loading, all leaves (not just groups of siblings) are contiguous. This analysis demonstrates the benefits of calibration discussed in Section 5.3; calibration can "detect" that the tree has been bulkloaded.

The right columns in Figure 7 show the overall performance for 5 million probes on the three architectures. The results match the calibration estimates. The structure $(3, 4)$ is fastest on the Pentium 3 in which the TLB miss latency is low. On both the UltraSparc and Pentium 4, the $(2, 1, 4)$ is the fastest.

Of the three architectures, the Pentium 4 is the only one that supports a TLB miss performance counter. We measure both the cache and TLB miss penalty on the Pentium 4 for all four placements, as shown in Figure 8. The L2 cache cost is dominant for the unbuffered index. All buffered placements have similar L2 cache behavior, while differing in the impact of TLB misses. These confirm that the performance difference among the four placements is due to TLB misses.

The main conclusions from these experiments are that when considering buffer placement, (a) the cache

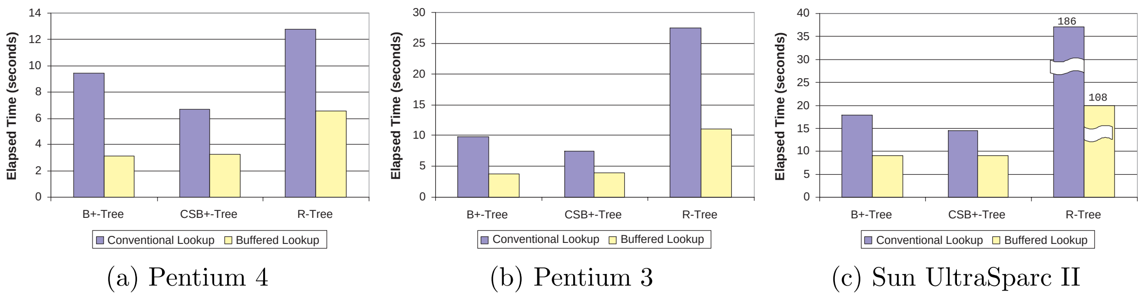


Figure 9: Overall Performance

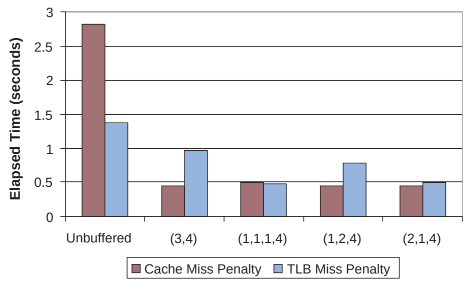


Figure 8: Cache and TLB Impact on Pentium 4

miss penalty is the main source of latency, (b) the TLB cost, while smaller, is still measurable (especially with good cache behavior), (c) different architectures have different optimal virtual node structures, and (d) calibration is a quick and effective tool for determining buffer placement.

6.3 Overall Search Performance

We repeat the experiments of Figure 1 on the three architectures and compare the original algorithm with variable-size buffering. We follow our guidelines for the choice of virtual node size. For the B$^+$-tree and the R-tree, each node is a virtual node.

Figure 9 shows the results. Buffered lookup is uniformly better, with speedups by a factor of two to three. Interestingly, the performance of B$^+$-trees and CSB$^+$-trees differ little once buffering techniques are applied. In both cases, we have a three-level virtual tree and we perform buffering twice. The buffering overhead is the same; the size of nodes is relatively unimportant for bulk access.

In the rest of this section, we show results on the Pentium 4; the other systems have similar behavior. Due to space limitations, we show results only for buffered B$^+$-trees.

6.4 Fixed-size versus Variable-size

We compare the search performance of different buffering techniques for a sequence of 5 million probes. We implement both fixed-size and variable-size buffering algorithms and compare with B$^+$-tree and CSB$^+$-tree. For fixed-size buffering, we choose the buffer size to be the same as the size of an index node, 4 Kbytes. We begin our experiments with a clear cache.

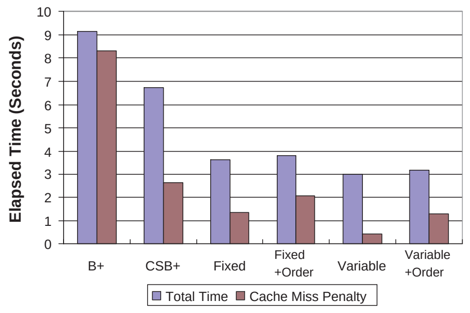


Figure 10: Search Performance

Figure 10 shows the elapsed time and the cache miss penalty for different algorithms. The total elapsed time for variable-size buffering includes the time to dynamically create buffers. We preallocate a large memory pool and space is allocated from the pool. Both buffering techniques significantly decrease the impact of cache misses. Fixed-size buffering does not eliminate all cache thrashing. With the lowest cache miss rate, variable-size buffering is fastest for pure search performance, and is three times faster than traditional lookup. Order-preserving buffering may incur additional cache misses for writes to the result buffer. In this experiment, the result buffer is equal in size to the number of probes, and we use the RID optimization described in Section 4.3. The results demonstrate that the overhead for maintaining probe order is small; the additional cache miss latency can be overlapped with other useful work.

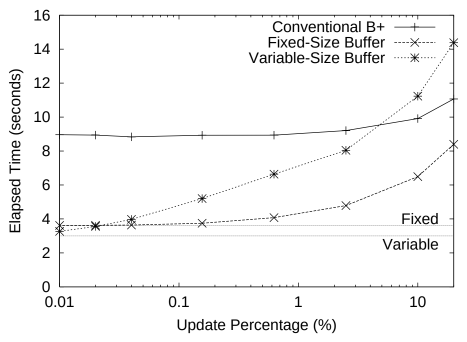


Figure 11: Buffering with Updates

We now examine the performance of both buffering techniques on a workload that contains both probes and updates (insertions).

Figure 11 compares a conventional B$^+$-tree index, with unbuffered probes, with both fixed-size and variable-size buffering. We randomly mix insertion and search operations, controlling the percentage of updates. We measure the elapsed time for a total of 5 million operations. The two horizontal lines respectively represent the search performance for fixed-size and variable-size buffering when there are no updates.

When the update percentage is small, both buffering techniques show small overhead. This is likely to be the most common scenario in our target applications: there are likely to be orders of magnitude more probes than updates in a given time window. As insertions increase in frequency, buffers are flushed more frequently and the advantage of buffering diminishes. Fixed-size buffering shows better performance than variable-size buffering, because the impact of flushing is smaller; variable-size buffering flushes a relatively large subtree for each update.

6.5 Non-Random Probes

We now examine how buffering techniques adjust to different properties of probes. We consider 5 million probes:

- Gaussian distributed probes such that 95% of the probes touch 20% of the leaf nodes, and 5% of the probes touch 80% of the leaf nodes
- Uniformly distributed probes sorted by the probe keys

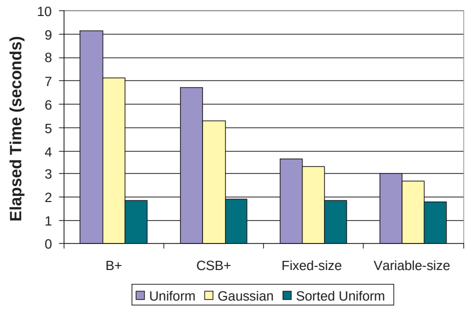


Figure 12: Three Datasets

When probes are skewed, both buffered and non-buffered structures show better performance and the speedup due to buffering, while still significant, is smaller (Figure 12). If probes are sorted by the search keys, all methods show good spatial locality and share similar performance. The performance for sorted access shows that all methods have roughly the same number of L2 cache misses and the buffering overhead is minimal; the cost of buffering is offset by a significant decrease in the number of L1 cache misses. Sorted access is unlikely to be a common access pattern, and explicitly sorting probes would be expensive.

6.6 Small Number of Probes

Buffering is helpful for a large number of probes. For a small number of probes, the overhead of buffering

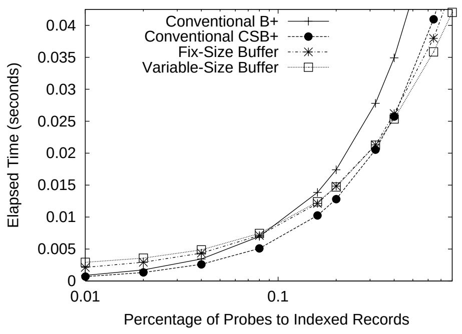


Figure 13: Small Number of Probes

dominates the overall cost. Figure 13 compares the performance when the probe cardinality ranges from 0.01% to 0.8% of the (5 million) indexed keys. Both buffering methods get faster than the B$^+$-tree when the outer relation size is larger than 0.08%, and become faster than the CSB$^+$-tree when the number exceeds 0.4%. The optimizer needs to consider the probe cardinality to decide when buffering can be beneficial.

6.7 Response Time

Basic buffering methods do not guarantee fast response time. We implement fixed-size buffering and a timer thread that, after some fixed time interval t, forces the whole tree to be flushed. That way, response time is at most t; we call t the *response time guarantee*.

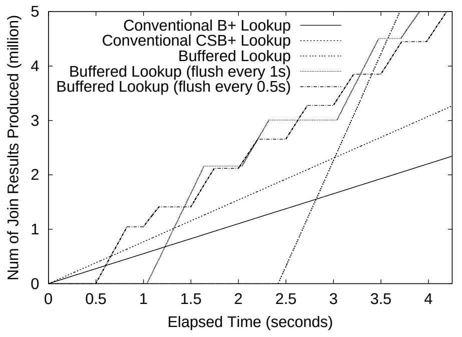


Figure 14: Response Time

In a first experiment, we model an index-nested-loops join that is a component of a pipelined query processing plan. We probe a B$^+$-tree with the parameters of Table 2. Figure 14 shows response time for different flush intervals. The experiment shows that after a short initial period, buffered access produces more output results (for processing by subsequent operations) than unbuffered access.

In a second experiment, we model a stream-processing application in which the data stream is used to probe an index. We perform 10^7 probes, and measure the throughput as $10^7/T$, where T is the total time needed to complete the experiment.

Figure 15 shows the throughput as a function of the response time guarantee. The horizontal dashed lines respectively represent the throughputs for con-

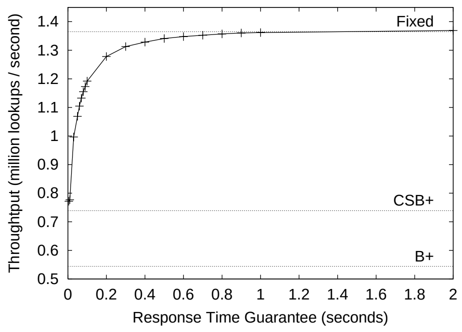


Figure 15: Throughput

ventional B$^+$-trees, CSB$^+$-trees, and the same fixed-size buffered tree but flushing the tree only at the end.

When the response time guarantee is small, there are more flush operations. The benefit of buffering is small and the throughput is comparable with conventional techniques. However, even when the response time guarantee is 0.1 seconds, the throughput of the buffered tree is noticeably better than those for the B$^+$-tree and the CSB$^+$-tree. The throughput improves as the response time guarantee increases. With a response time guarantee of one second, the system can achieve almost the same throughput as a fixed-size buffer tree that only flushes at the end.

The importance of this experiment is that it demonstrates that buffered indexes can achieve a throughput that is simply not achievable with conventional indexes. In a scenario in which the application needed to handle 1 million lookups per second, and could tolerate small response time delays, *buffered trees would work on the given tree and architecture, while conventional indexes would not.*

7 Conclusion

Both conventional and cache-conscious index structures suffer from cache thrashing between query accesses. We propose techniques to buffer accesses to memory-resident tree-structured indexes to exploit cache spatial and temporal locality. The buffer structures are separate from the index structures. For single lookups, one may use the index in a conventional fashion. For bulk lookups, our algorithms improve the rate at which probes can be processed typically by a factor of two to three. Our algorithms can also handle updates mixed with probes, while providing a response time guarantee for probes.

These results are important for applications such as stream processing, where probe throughput can be a critical parameter. They are also relevant for speeding up more conventional query processing operations, such as index-nested-loops joins, in main memory.

While we have described buffering in the context of a tree-based index, it also applies to structures such as hash indexes. If the hash table is too large to fit in the L2 cache, it may be beneficial to partition and buffer the probes into smaller groups using a prefix of the hashed key, so that the hash table fragment required by each group can fit in the cache.

References

[1] Berkeley Telegraph Project. http://telegraph.cs.berkeley.edu.

[2] Cornell Cougar Project. http://cougar.cs.cornell.edu.

[3] LMbench - Tools for Performance Analysis. http://www.bitmover.com/lmbench/.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of VLDB conference*, 1999.

[5] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of VLDB Conference*, 1999.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of ACM SIGMOD Conference*, 2001.

[7] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *Proceedings of ACM SIGMOD Conference*, 2002.

[8] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: A generalized r-tree bulk-insertion strategy. In *Symposium on Large Spatial Databases*, 1999.

[9] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of ACM SIGMOD Conference*, 1984.

[11] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertions. In *Proceedings of VLDB conference*, 1999.

[12] I. Kamel and C. Faloutsos. On packing r-trees. In *International Conference on Information and Knowledge Management*, 1993.

[13] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of ACM SIGMOD Conference*, 2001.

[14] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylnen. Concurrency control in b-trees with batch updates. In *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[15] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of VLDB Conference*, 1999.

[16] J. Rao and K. A. Ross. Making B+ trees cache conscious in main memory. In *Proceedings of ACM SIGMOD Conference*, 2000.

[17] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), 2000.

[18] J. van den Bercken, B. Seeger, and P. Widmayer. The bulk index join: A generic approach to processing non-equijoins. In *Proceedings of IEEE Conference on Data Eng.*, 1999.