Improving Index Performance through Prefetching

Shimin Chen
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
chensm@cs.cmu.edu

Phillip B. Gibbons
Information Sciences Research Center
Bell Laboratories
Murray Hill, NJ 07974
gibbons@research.bell-labs.com

Todd C. Mowry
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
tcm@cs.cmu.edu

ABSTRACT

This paper proposes and evaluates *Prefetching B^+-Trees* (pB^+-Trees), which use prefetching to accelerate two important operations on B^+-Tree indices: searches and range scans. To accelerate searches, pB^+-Trees use prefetching to effectively create wider nodes than the natural data transfer size: e.g., eight vs. one cache lines or disk pages. These wider nodes reduce the height of the B^+-Tree, thereby decreasing the number of expensive misses when going from parent to child without significantly increasing the cost of fetching a given node. Our results show that this technique speeds up search and update times by a factor of 1.2–1.5 for main-memory B^+-Trees. In addition, it outperforms and is complementary to "Cache-Sensitive B^+-Trees." To accelerate range scans, pB^+-Trees provide arrays of pointers to their leaf nodes. These allow the pB^+-Tree to prefetch arbitrarily far ahead, even for nonclustered indices, thereby hiding the normally expensive cache misses associated with traversing the leaves within the range. Our results show that this technique yields over a *sixfold* speedup on range scans of 1000+ keys. Although our experimental evaluation focuses on main memory databases, the techniques that we propose are also applicable to hiding disk latency.

1. INTRODUCTION

As the gap between processor speed and both DRAM and disk speeds continues to grow exponentially, it is becoming increasingly important to make effective use of caches to achieve high performance on database management systems. Caching exists at multiple levels within modern memory hierarchies: typically two or more levels of SRAM serves as caches for the contents of main memory in DRAM, which in turn is a cache for the contents on disk. While database researchers have historically focused on the importance of this latter form of caching (also known as the "buffer pool"), recent studies have demonstrated that even on traditional disk-oriented databases, roughly 50% or more of execution time is often wasted due to SRAM cache misses [1, 2, 10, 18]. For main-memory databases, it is even clearer that SRAM cache performance is crucial [19]. Hence several recent studies have revisited core database algorithms in an effort to make them more cache friendly [5, 17, 19, 20, 21].

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

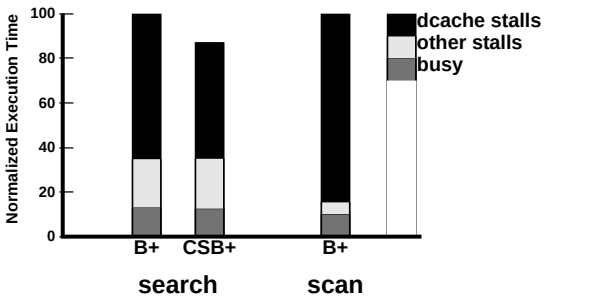


Figure 1: Execution time breakdown for index operations (B+ = B^+-Trees, CSB+ = CSB^+-Trees).

1.1 Cache Performance of B^+-Tree Indices

Index structures are used extensively throughout database systems, and they are often implemented as B^+-Trees. While database management systems perform several different operations that involve B^+-Tree indices (e.g., selections, joins, etc.), these higher-level operations can be decomposed into two key lower-level access patterns: (i) *searching* for a particular key, which involves descending from the root to a leaf node using binary search within a given node to determine which child pointer to follow; and (ii) *scanning* some portion of the index, which involves traversing the leaves through a linked-list structure for a non-clustered index. (For clustered indices, one can directly scan the database table after searching for the starting key.) While search time is the key factor in single value selections and nested loop index joins, scan time is the dominant effect in range selections.

To illustrate the need for improving the cache performance of both search and scan on B^+-Tree indices, Figure 1 shows a breakdown of their simulated performance on a state-of-the-art machine. For the sake of concreteness, we pattern the memory subsystem after the Compaq ES40 [8]—details are provided later in Section 4. The "search" experiment in Figure 1 looks up 100,000 random keys in a main-memory B^+-Tree index after it has been bulkloaded with 10 million keys. The "scan" experiment performs 100 range scan operations starting at random keys, each of which scans through 1 million $\langle$`key`, `tupleID`$\rangle$ pairs retrieving the `tupleID` values. (The results for shorter range scans—e.g., 1000 tuple scans—are similar). The B^+-Tree node size is equal to the cache line size, which is 64 bytes. Each bar in Figure 1 is broken down into three categories: busy time, data cache stalls, and other stalls. Both search and scan accesses on B^+-Tree indices (the bars labeled "**B+**"—we will explain the "**CSB+**" bar later) spend a significant fraction of their time—65% and 84%, respectively—stalled on data cache misses. Hence there is considerable room for improvement.

1.2 Previous Work on Improving the Cache Performance of Indices

In an effort to improve the cache performance of index *searches* for main-memory databases, Rao and Ross proposed two new types of index structures: "Cache-Sensitive Search Trees" (CSS-Trees) [19] and "Cache-Sensitive B^+-Trees" (CSB^+-Trees) [20]. The premise of their studies is the conventional wisdom that the optimal tree node size is equal to the *natural data transfer size*, which corresponds to the *disk page size* for disk-resident databases and the *cache line size* for main-memory databases. Because cache lines are roughly two orders of magnitude smaller than disk pages (e.g., 64 bytes vs. 4 Kbytes), the resulting index trees for main-memory databases are considerably deeper. Since the number of expensive cache misses is roughly proportional to the height of the tree, it would be desirable to somehow increase the effective fanout (also called the *branching factor*) of the tree, without paying the cost of additional cache misses that this would normally imply.

To accomplish this, Rao and Ross [19, 20] exploit the following insight: by restricting the data layout such that the location of each child node can be directly computed from the parent node's address (or a single pointer), we can eliminate all (or nearly all) of the child pointers. Assuming that keys and pointers are the same size, this effectively doubles the fanout of cache-line-sized tree nodes, thus reducing the height of the tree and the number of cache misses. CSS-Trees [19] eliminate all child pointers, but do not support incremental updates and therefore are only suitable for read-only environments. CSB^+-Trees [20] do support updates by retaining a single pointer per non-leaf node that points to a contiguous block of its children. Although CSB^+-Trees outperform B^+-Trees on searches, they still perform significantly worse on updates [20] due to the overheads of keeping all children for a given node in sequential order within contiguous memory, especially during node splits.

Returning to Figure 1, the bar labeled "**CSB+**" shows the execution time of CSB^+-Trees (normalized to that of B^+-Trees) for the same index search experiment. As we see in Figure 1, CSB^+-Trees eliminate 20% of the data cache stall time, thus resulting in an overall speedup[1] of 1.15 for searches. While this is a significant improvement, over half of the remaining execution time is still being lost to data cache misses; hence there is significant room for further improvement. In addition, these search-oriented optimizations provide no benefit to *scan* accesses, which are suffering even more from data cache misses.

1.3 Our Approach: Prefetching B^+-Trees

Modern microprocessors provide the following mechanisms for coping with large cache miss latencies. First, they allow multiple outstanding cache misses to be in flight simultaneously for the sake of exploiting parallelism within the memory hierarchy. For example, the Compaq ES40 [8] supports 32 in-flight loads, 32 in-flight stores, and eight outstanding off-chip cache misses per processor, and its crossbar memory system supports 24 outstanding cache misses. Second, to help applications take full advantage of this parallelism, they also provide *prefetch* instructions which enable software to move data into the cache before it is needed. Previous studies (which did not target databases specifically) have demonstrated that for both array-based and pointer-based program codes, prefetching can successfully *hide* much of the performance impact of cache misses by overlapping them with computation and other misses [13, 16]. Hence for modern machines, it is not the *number* of cache misses that dictates performance, but rather the amount of *exposed miss latency* that cannot be successfully hidden through techniques such as prefetching.

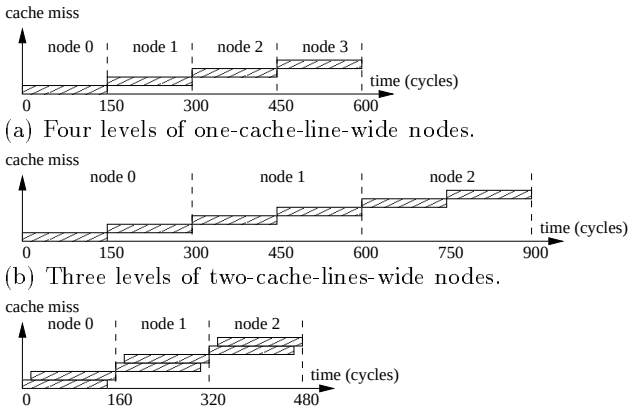
In this paper, we propose and study *Prefetching B^+-Trees* (pB^+-Trees), which use prefetching to limit the exposed miss latency. Tree-based indices such as B^+-Trees pose a major challenge for prefetching search and scan accesses since both access patterns suffer from the *pointer-chasing problem* [13]: The data dependencies through pointers make it difficult to prefetch sufficiently far ahead to limit the exposed miss latency. For index *searches*, pB^+-Trees reduce this problem by having wider nodes than the natural data transfer size, e.g., eight vs. one cache lines (or disk pages). These wider nodes reduce the height of the tree, thereby decreasing the number of expensive misses when going from parent to child. The key observation is that by using prefetching, the wider nodes come almost for free: all of the cache lines in a wider node can be fetched almost as quickly as the single cache line of a traditional node. To accelerate index *scans*, we introduce arrays of pointers to the B^+-Tree leaf nodes which allow us to prefetch arbitrarily far ahead, thereby hiding the normally expensive cache misses associated with traversing the leaves within the range. Of course, indices may be frequently updated. Perhaps surprisingly, we demonstrate that insertion and deletion times actually *decrease* with our techniques, despite any overheads associated with maintaining the wider nodes and the arrays of pointers.

1.4 Contributions of This Paper

This paper makes the following contributions. First, to our knowledge, this is the first study to explore how prefetching can be used to accelerate search and scan operations on B^+-Tree indices. We propose and study the *Prefetching B^+-Tree* (pB^+-Tree). Second, we demonstrate that contrary to conventional wisdom, the optimal B^+-Tree node size on a modern machine is often *wider* than the natural data transfer size, since we can use prefetching to fetch each piece of the node simultaneously. Our approach offers the following advantages relative to CSB^+-Trees: (i) we achieve better search performance because we can increase the fanout by more than the factor of two that CSB^+-Trees provide (e.g., by a factor of eight); (ii) we achieve better (rather than worse) performance on updates relative to B^+-Trees, because our improved search speed more than offsets any increase in node split cost due to wider nodes; and (iii) we do not require fundamental changes to the original B^+-Tree data structures or algorithms. In addition, we find that our approach is *complementary* to CSB^+-Trees. Third, we demonstrate how the pB^+-Tree can effectively hide over 90% of the cache miss latency suffered by (non-clustered) index scans, thus resulting in *a factor of 6.5–8.7 speedup* over a range of scan lengths. While our experimental evaluation is performed within the context of main memory databases, we believe that our techniques are also applicable to hiding disk latency, in which case the prefetches will move data from disk into main memory.

The remainder of this paper is organized as follows. Sections 2 and 3 discuss how pB^+-Trees use prefetching to accelerate index searches and scans, respectively. To quantify the benefits of these techniques, we present experimental results in Section 4. Finally, we discuss further issues and conclude in Sections 5 and 6, respectively.

[1] Throughout this paper, we report performance gains as *speedup*: i.e. the original time divided by the improved time.



(a) Four levels of one-cache-line-wide nodes.

(b) Three levels of two-cache-lines-wide nodes.

(c) Part (b) with prefetching.

Figure 2: Performance of various B$^+$-Tree searches where a cache miss to memory takes 150 cycles, and a subsequent access can begin 10 cycles later [8].

2. PREFETCHING INDEX SEARCHES

Recall that during a B$^+$-Tree search, we start from the root, performing a binary search in each non-leaf node to determine which child to visit next. Upon reaching a leaf node, a final binary search returns the key position. Regarding the cache behavior, we expect at least one expensive cache miss to occur each time we move down a level in the tree. Hence the number of cache misses is roughly proportional to the height of the tree (minus any nodes that might remain in the cache if the index is reused). Thus, having wider tree nodes for the sake of reducing the height of the tree might seem like a good idea. Unfortunately, in the absence of prefetching (i.e. when all cache misses are equally expensive and cannot be overlapped), making the tree nodes wider than the *natural data transfer size*—i.e. a cache line for main-memory databases (and a disk page for disk-resident databases)—actually *hurts* performance rather than helps it, as has been shown in previous studies [19, 20]. The reason for this is that the number of additional cache misses at each node more than offsets the benefits of reducing the number of levels in the tree.

As a small example, consider a main-memory B$^+$-Tree holding 1000 keys where the cache line size is 64 bytes and the `keys`, child pointers, and `tupleIDs` are all four bytes. If we limit the node size to one cache line, then the B$^+$-Tree will contain at least four levels. Figure 2(a) illustrates the resulting cache behavior, where the four cache misses cost a total of 600 cycles on our Compaq ES40-based machine model [8]. If we double the node width to *two* cache lines, the height of the B$^+$-Tree can be reduced to *three* levels. However, as we see in Figure 2(b), this would result in *six* cache misses, thus increasing execution time by 50%.

With prefetching, however, it becomes possible to *hide* the latency of any miss whose address can be predicted sufficiently early. Returning to our example, if we prefetch the second half of each two-cache-line-wide tree node so that it is fetched in parallel with the first half—as illustrated in Figure 2(c)—we can achieve significantly *better* (rather than worse) performance compared with the one-cache-line-wide nodes in Figure 2(a). The extent to which the misses can be overlapped depends upon the implementation details of the memory hierarchy, but the trend is toward supporting greater parallelism. In fact, with multiple cache and memory banks and crossbar interconnects, it is possible to completely overlap multiple cache misses. Figure 2(c) illustrates the timing on our Compaq ES40-based machine model, where back-to-back misses to memory can be serviced once every 10 cycles, which is a small fraction of the overall 150 cycle miss latency. Therefore even without perfect overlap of the misses, we can still potentially achieve large performance gains (a speedup of 1.25 in this example) by creating wider than normal B$^+$-Tree nodes.

Hence the first aspect of our pB$^+$-Tree design is to use prefetching to "create" nodes that are *wider* than the natural data transfer size, but where the entire miss penalty for each extra-wide node is comparable to that of an original B$^+$-Tree node.

Table 1: Terminology used throughout this paper.

Variable	Definition
w	# of cache lines in an index node
m	# of child pointers in a one-line-wide node
N	# of $\langle$`key`, `tupleID`$\rangle$ pairs in an index
d	# of child pointers in non-leaf node ($= w \times m$)
T_1	full latency of a cache miss
T_{next}	latency of an additional pipelined cache miss
B	normalized memory bandwidth $\left(B = \frac{T_1}{T_{\text{next}}}\right)$
k	# of nodes to prefetch ahead
c	# of cache lines in jump-pointer array chunk
p^{w}B$^+$-Tree	plain pB$^+$-Tree with w-line-wide nodes
p^w_eB$^+$-Tree	p^{w}B$^+$-Tree with *external* jump-pointer arrays
p^w_iB$^+$-Tree	p^{w}B$^+$-Tree with *internal* jump-pointer arrays

2.1 Modifications to the B$^+$-Tree Algorithm

We consider a standard B$^+$-Tree node structure: Each *non-leaf* node is comprised of some number, $d \gg 1$, of `childptr` fields, $d-1$ `key` fields,[2] and one `keynum` field that records the number of `keys` stored in the node (at most $d-1$). (All notation is summarized in Table 1.) Each *leaf* node is comprised of $d-1$ `key` fields, $d-1$ associated `tupleID` fields, one `keynum` field, and one `next-leaf` field that points to the next leaf node in key order. Our first modification is to store the `keynum` and all of the `keys` prior to any of the pointers or `tupleIDs` in a node. This simple layout optimization allows the binary search to proceed without waiting to fetch all the pointers. Our search algorithm is a straightforward extension of the standard B$^+$-Tree algorithm, and we now describe only the parts that change.

Search: Before starting a binary search, we prefetch all of the cache lines that comprise the node.

Insertion: Since an index search is first performed to locate the position for insertion, all of the nodes on the path from the root to the leaf are already in the cache before the real insertion phase. The only additional cache misses are caused by newly allocated nodes, which we prefetch in their entirety before redistributing the keys.

Deletion: We perform *lazy deletion* as in Rao and Ross [20]. If more than one key is in the node, we simply delete the key. It is only when the last key in a node is deleted that we try to redistribute keys or delete the node. Since index search is also performed prior to deletion, the entire root-to-leaf path is in the cache. Key redistribution is the only potential cause of additional misses; hence when all keys in a node are deleted, we prefetch the sibling node from which keys will be redistributed.

[2] Throughout this paper, we consider for simplicity *fixed-size* `keys`, `tupleIDs`, and pointers. We also assume that `tupleIDs` and pointers are the same size.

Prefetching can also be used to accelerate the *bulkload* of a B$^+$-Tree. However, because this is expected to occur infrequently, we focus instead on the more frequent operations of search, insertion and deletion.

2.2 Qualitative Analysis

As discussed earlier in this section, we expect search times to improve through our scheme because it reduces the number of levels in the B$^+$-Tree without significantly increasing the cost of accessing each level. What about the performance impact on updates? Updates always begin with a search phase, which will be sped up. The expensive operations only occur either when the node becomes too full upon an insertion and must be split, or when a node becomes empty upon a deletion and keys must be redistributed. Although node splits and key redistributions are more costly with larger nodes, the relative frequency of these expensive events should decrease. Therefore we expect update performance to be comparable to, or perhaps even better than, B$^+$-Trees with single-line nodes.

The space overhead of the index is strictly reduced with wider nodes. This is primarily due to the reduction in the number of non-leaf nodes. For a full tree, each leaf node contains $d-1$ $\langle$`key`, `tupleID`$\rangle$ pairs. The number of non-leaf nodes is dominated by the number of nodes in the level immediately above the leaf nodes, and hence is approximately $\frac{N}{d(d-1)}$. As the fanout d increases with wider nodes, the node size grows linearly but the number of non-leaf nodes decreases quadratically, resulting in a near linear decrease in the non-leaf space overhead.

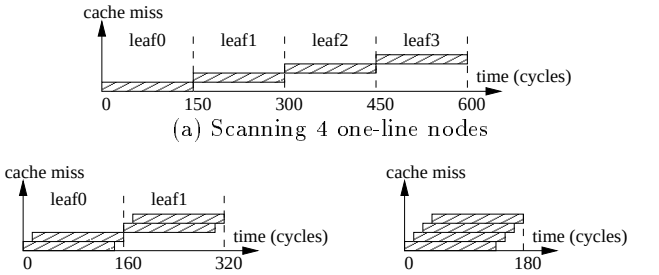
Finally, an interesting consideration is to determine the optimal node size, given prefetching. Should nodes simply be as wide as possible? There are two system parameters that affect this answer. The first is the extent to which the memory subsystem can overlap multiple cache misses. We quantify this as the latency of a full cache miss (T_1) divided by the additional time until another pipelined cache miss would also complete (T_{next}). We call this ratio (i.e. $\frac{T_1}{T_{\text{next}}}$) the *normalized bandwidth* (B). For example, in our Compaq ES40-based machine model, $T_1 = 150$ cycles, $T_{\text{next}} = 10$ cycles, and hence $B = 15$. The larger the value of B, the greater the system's ability to overlap parallel accesses, and hence the greater likelihood of benefiting from wider index nodes. In general, we do not expect the optimal number of cache lines per node (w_{optimal}) to exceed B, since beyond that point we could have completed a binary search and moved down to the next level in the tree. The second system parameter that potentially limits the optimal node size is the size of the cache, although in practice this does not appear to be a limitation given realistic values of B.

Let us now consider a more quantitative analysis of the optimal node width w_{optimal}. A pB$^+$-Tree with N $\langle$`key`, `tupleID`$\rangle$ pairs contains at least $\left\lceil \log_d \left(\frac{N}{d-1} \right) + 1 \right\rceil$ levels. With our data layout optimization of putting keys before child pointers, $\frac{3}{4}$ of the node is read on average. Hence the average memory stall time for a search in a full tree is

$$\begin{aligned} &\left\lceil \log_d \tfrac{N}{d-1} + 1 \right\rceil \times \left(T_1 + \left(\left\lceil \tfrac{3w}{4} \right\rceil - 1 \right) \times T_{\text{next}} \right) \\ &= T_{\text{next}} \times \left\lceil \log_{wm} \tfrac{N}{wm-1} + 1 \right\rceil \times \left(B + \left\lceil \tfrac{3w}{4} \right\rceil - 1 \right) \end{aligned} \quad (1)$$

By computing the value of w that minimizes this cost, we can find w_{optimal}. For example, in our simulations where $m = 8$ and $B = 15$, by averaging over tree sizes $N = 10^3, \ldots, 10^9$, we can compute from equation (1) that $w_{\text{optimal}} = 8$. If the memory subsystem bandwidth increases such that $B = 50$, then w_{optimal} increases to 22.

In summary, comparing our pB$^+$-Trees with conventional B$^+$-Trees, we expect better search performance, comparable or somewhat better update performance, and lower space overhead. Having addressed index search performance, we now turn our attention to index range scans.



(a) Scanning 4 one-line nodes

(b) Scanning 2 two-line nodes (c) Scanning with prefetching

Figure 3: Cache behaviors of index range scans.

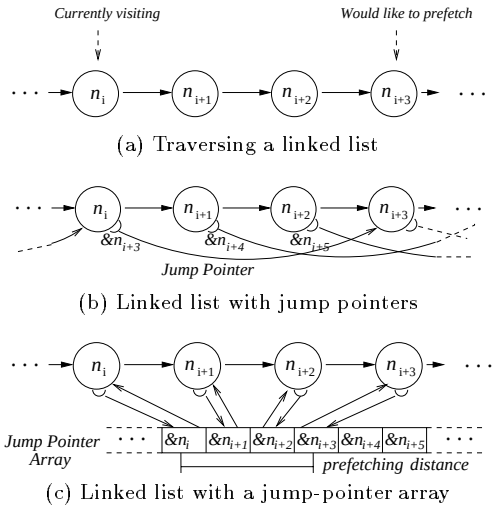
3. PREFETCHING INDEX SCANS

Given starting and ending keys as arguments, an index range scan returns a list of either the `tupleID`s or the tuples themselves with keys that fall within this range. First the starting key is *searched* in the B$^+$-Tree to locate the starting leaf node. Then the scan follows the `next-leaf` pointers, visiting the leaf nodes in order. As the scan proceeds, the `tupleID`s (or tuples) are copied into a return buffer. This process stops when either the ending key is found or the return buffer fills up. In the latter case, the scan procedure pauses and returns the buffer to the caller (often a join node in a query execution plan), which then consumes the data and resumes the scan where it left off. Hence a range selection involves one key search and often multiple leaf node scan calls. Throughout this section, we will focus on range selections that return `tupleID`s, although returning the tuples themselves (or other variations) is a straightforward extension of our algorithm, as discussed in the full paper [7].

As we saw already in Figure 1, the cache performance of range scans is abysmal: 84% of execution time is being lost to data cache misses in that experiment. Figure 3(a) illustrates the problem: a full cache miss latency is suffered for each leaf node. A partial solution is to use the technique described in Section 2: If we make the leaf nodes multiple cache lines wide and prefetch each component of a leaf node in parallel, we can reduce the frequency of expensive cache misses, as illustrated in Figure 3(b). While this is helpful, our goal is to *fully* hide the miss latencies to the extent permitted by the memory system, as illustrated in Figure 3(c). In order to do that, we must first overcome the *pointer-chasing problem*.

3.1 Solving the Pointer-Chasing Problem

Figure 4(a) illustrates the *pointer-chasing problem*, which was observed by Luk and Mowry [13, 14] in the context of prefetching pointer-linked data structures (i.e. linked-lists, trees, etc.) in general-purpose applications. Assuming that three nodes worth of computation are needed to hide the miss latency, then when node n_i in Figure 4(a) is visited, we would like to be launching a prefetch of node n_{i+3}. To compute the address of node n_{i+3}, we would normally follow the pointer chain through nodes n_{i+1} and n_{i+2}. However, this would incur the full miss latency to fetch n_{i+1} and then to fetch n_{i+2}, before the prefetch of n_{i+3} could be launched,

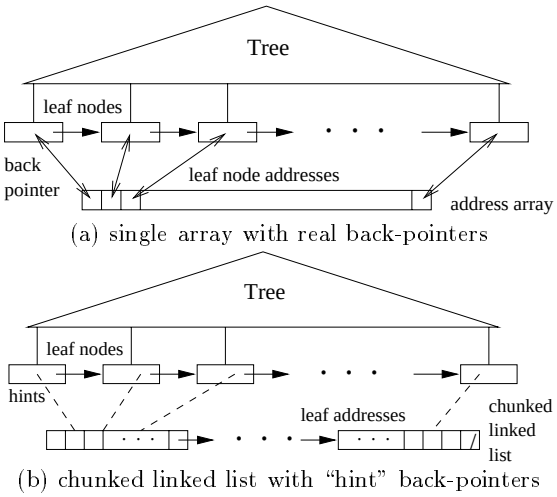


(a) Traversing a linked list

(b) Linked list with jump pointers

(c) Linked list with a jump-pointer array

Figure 4: Addressing the pointer-chasing problem.



(a) single array with real back-pointers

(b) chunked linked list with "hint" back-pointers

Figure 5: External jump-pointer arrays.

thereby defeating our goal of hiding the miss latency of n_{i+3}.

Luk and Mowry proposed two solutions to the pointer-chasing problem that are applicable to linked lists [13, 14]. The first scheme (*data-linearization prefetching*) involves arranging the nodes in memory such that their addresses can be trivially calculated without dereferencing any pointers. For example, if the leaf nodes of the B$^+$-Tree are arranged sequentially in contiguous memory, they would be trivial to prefetch. However, this will only work in read-only situations, and we would like to support frequent updates. The second scheme (*history-pointer prefetching*) involves creating new pointers—called *jump pointers*—which point from a node to the node that it should prefetch. For example, Figure 4(b) shows how node n_i could directly prefetch node n_{i+3} using three-ahead jump pointers.

In our study, we will build upon the concept of jump pointers, but customize them to the specific needs of B$^+$-Tree indices. Rather than storing jump pointers directly in the leaf nodes, we instead pull them out into a separate array, which we call the *jump-pointer array*, as illustrated in Figure 4(c). To initiate prefetching, a back-pointer in the starting leaf node is used to locate the leaf's position within the jump pointer array; then based on the desired prefetching distance, an array offset is adjusted to find the address of the appropriate leaf node to prefetch. As the scan proceeds, the prefetching task simply continues to walk ahead in the jump-pointer array (which itself is also prefetched) without having to dereference the actual leaf nodes again.

Jump-pointer arrays are more flexible than jump pointers stored directly in the leaf nodes. We can adjust the prefetching distance by simply changing the offset used within the array. This allows dynamic adaptation to changing performance conditions on a given machine, or if the code migrates to different machines. In addition, the same jump-pointer array can be reused to target different latencies in the memory hierarchy (e.g., disk latency vs. memory latency).

From an abstract perspective, one might think of the jump-pointer array as a single large, contiguous array, as illustrated in Figure 5(a). This would be efficient in read-only situations, but in such cases we could simply arrange the leaf nodes themselves contiguously and use *data-linearization prefetching* [13, 14]. Therefore a key issue in implementing jump-pointer arrays is to handle updates gracefully.

3.2 Implementing Jump-Pointer Arrays to Support Efficient Updates

Let us briefly consider the problems created by updates if we attempted to maintain the jump-pointer array as a single contiguous array as shown in Figure 5(a). When a leaf is deleted, we can simply leave an empty slot in the array. However, insertion can be very expensive. When a new leaf is *inserted*, an empty slot needs to be created in the appropriate position for the new jump pointer. If no nearby empty slots could be located, this could potentially involve copying a very large amount of data within the array in order to create the empty slot. In addition, for each jump-pointer that is moved within the array, the corresponding back-pointer from the leaf node into the array also needs to be updated, which could be very costly too. Clearly we would not want to pay such a high cost upon insertions.

We improve upon the naive contiguous array implementation in the following three ways. First, we break the contiguous array into a chunked linked list—as illustrated in Figure 5(b)—which allows us to limit the impact of an insertion to its corresponding chunk. (We will discuss the chunk size selection later in Section 3.3).

Second, we actively attempt to interleave empty slots within the jump-pointer array so that insertions can proceed quickly. During bulkload or when a chunk splits, the jump pointers are stored such that empty slots are evenly distributed to maximize the chance of finding a nearby empty slot for insertion. When a jump-pointer is deleted, we simply leave an empty slot in the chunk.

Finally, we alter the meaning of the back-pointer in a leaf node to its position in the jump-pointer array such that it is merely a hint. The pointer should point to the correct chunk, but the position within that chunk may be imprecise. Therefore when moving jump pointers in a chunk for inserting a new leaf address, we do not need to update the hints for the moved jump pointers. We only update a `hint` field when: (i) the precise position in the jump-pointer array is looked up during range scan or insertion, in which case the leaf node should be already in cache and updating the `hint` is almost free; and (ii) when a chunk splits and addresses are redistributed, in which case we are forced to update the `hints` to point to the new chunk. The cost of using hints, of course, is that we need to search for the correct location within the chunk in some cases. In practice, however, the

hints appear to be good approximations of the true positions, and searching for the precise location is not a costly operation (e.g., it should not incur any cache misses).

In summary, the net effect of these three enhancements is that nothing moves during deletions, typically only a small number of jump pointers (between the insertion position and the nearest empty slot) move during insertions, and in neither case do we normally update the `hints` within the leaf nodes. Thus we expect jump-pointer arrays to perform well during updates.

3.3 Algorithm and Qualitative Analysis

Having described the data structure to facilitate prefetching, we now describe our prefetching algorithm. Recall that the basic range scan algorithm consists of a loop that visits a leaf on each iteration by following a `next-leaf` pointer. To support prefetching, we add prefetches both prior to this loop (for the *startup* phase), and inside the loop (for the *steady-state* phase). Let k be the desired *prefetching distance*, in units of leaf nodes (we discuss below how to select k). During the startup phase, we issue prefetches for the first k leaf nodes.[3] These prefetches proceed in parallel, exploiting the available memory hierarchy bandwidth. During each loop iteration (i.e. in the steady-state phase), prior to visiting the current leaf node in the range scan, we prefetch the leaf node that is k nodes after the current leaf node. The goal is to ensure that by the time the basic range scan loop is ready to visit a leaf node, that node is already prefetched into the cache. With this framework in mind, we now describe further details of our algorithm.

First, in the startup phase, we must locate the jump pointer of the starting leaf within the jump-pointer array. To do this, we follow the `hint` pointer from the starting leaf to see whether it is precise—i.e. whether the `hint` points to a pointer back to the starting leaf. If not, then we start searching within the chunk in both directions relative to the hint position until the matching position is found. As discussed earlier, the distance between the hint and the actual position appears to be small in practice.

Second, we need to prefetch the jump-pointer chunks as well as the leaf nodes, and handle empty slots in the chunks. During the startup phase, both the current chunk and the next chunk are prefetched. When looking for a jump pointer, we test for and skip all empty slots. If the end of the current chunk is reached, we will go to the next chunk to get the first non-empty jump-pointer (there is at least one non-empty jump pointer or the chunk should have been deleted). We then prefetch the next chunk ahead in the jump-pointer array. Because we always prefetch the next chunk before prefetching any leaf nodes pointed to by the current chunk, we expect the next chunk to be in the cache by the time we access it.

Third, although the actual number of `tupleIDs` in the leaf node is unknown when we do range prefetching, we will assume that the leaf is full and prefetch the return buffer area accordingly. Thus the return buffer will always be prefetched sufficiently early.

We now discuss how to select the prefetching distance and the chunk size.

Selecting the prefetching distance k. The *prefetching distance* (k, in units of nodes to prefetch ahead) is selected as follows. Normally this quantity is derived by dividing the expected worst-case miss latency by the computation time spent on one leaf node (similar to what has been done in other contexts [16]). However, because the computation time associated with visiting a leaf node during a range scan is quite small relative to the miss latency, we will assume that the limiting factor is the memory bandwidth. Roughly speaking, we can estimate this bandwidth-limited prefetching distance as

$$k = \left\lceil \frac{B}{w} \right\rceil , \tag{2}$$

where B is the normalized memory bandwidth and w is the number of cache lines per leaf node, as defined in Table 1. In practice, there is no problem with increasing k a bit to create some extra slack, because any prefetches that cannot proceed are simply buffered within the memory system.[4]

Selecting the chunk size c. Chunks must be sufficiently large to ensure that we only need to prefetch one chunk ahead to hide the miss latency of accessing the chunks themselves. Recall that during the steady-state phase of a range scan, when we get to a new chunk, we immediately prefetch the next chunk ahead so that we can overlap its fetch time with the time it takes to prefetch the leaf nodes associated with the current chunk. Since the memory hierarchy only has enough bandwidth to initiate B cache misses during the time it takes one cache miss to complete, the chunks would clearly be large enough to hide the latency of fetching the next chunk if they contained at least B leaf pointers (there is at least one cache line access for every leaf visit). For a full tree with no empty leaf slots and no empty chunk slots, each cache line can hold $2m$ leaf pointers (since there are only pointers and no keys), in which case we can estimate the minimum chunk size in units of cache lines as

$$c = \left\lceil \frac{B}{2m} \right\rceil . \tag{3}$$

To account for empty chunk slots, we can multiply the denominator in equation (3) by the occupancy of chunk slots (a value similar to the *bulkload factor*), which would increase c somewhat. Another factor that could (in theory) dictate the minimum chunk size is that each chunk should contain at least k leaf pointers so that our prefetching algorithm can get sufficiently far ahead. However, since $k \leq B$ from equation (2), the chunk size in equation (3) should be sufficient. Increasing c beyond this minimum value to create some extra slack for empty leaf nodes and empty chunk slots does not hurt performance in practice.[4]

Remarks. Given sufficient memory system bandwidth, our prefetching scheme hides the full memory latency experienced at every leaf node visited during range scan operations. With the data structure improvements in Section 3.2, we also expect good performance on updates.

However, there is a space overhead associated with the jump-pointer array. Since the jump pointer array only contains one pointer per leaf node, the space overhead is relatively small. Since a next-leaf pointer and a back-pointer are stored in every leaf, there are at most $d-2$ $\langle$`key`, `tupleID`$\rangle$ pairs in every leaf nodes (d is defined in Table 1). So the jump pointer for a full leaf node only takes $\frac{1}{2(d-2)}$ as much space as the leaf node. Given our technique described earlier in Section 2 for creating wider B$^+$-Tree nodes, the resulting increase in the fanout d will help reduce this overhead. How-

[3] Note that the buffer area to hold the resulting `tupleIDs` needs also to be prefetched; to simplify presentation, when we refer to "prefetching a leaf node" in the range scan algorithm, we mean prefetching the cache lines for both the leaf node and the buffer area where the `tupleIDs` are to be stored.

[4] Details are in the full paper [7].

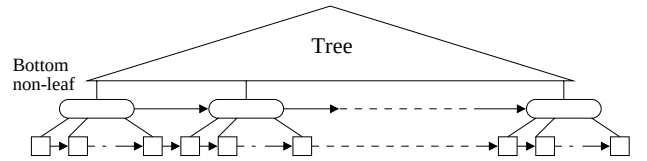


Figure 6: Internal jump-pointer arrays.

ever, if this space overhead is still a concern, we now describe how it can be reduced further.

3.4 Internal Jump-Pointer Arrays

So far we have described how a jump-pointer array can be implemented by creating a new *external* structure to store the jump pointers (as illustrated earlier in Figure 5). However, there is an existing structure *within* a B$^+$-Tree that already contains pointers to the leaf nodes, namely, the *parents* of the leaf nodes. We will refer to these parent nodes as the *bottom non-leaf nodes*. The child pointers within a bottom non-leaf node correspond to the jump-pointers within a chunk of the external jump-pointer array described in Section 3.2. A key difference, however, is that there is no easy way to traverse these bottom non-leaf nodes quickly enough to perform prefetching. A potential solution is to connect these bottom non-leaf nodes together in leaf key order using linked-list pointers. (Note that this is sometimes done already for concurrency control purposes [22].)

Figure 6 illustrates the *internal* jump-pointer array. Note that leaf nodes do not contain back-pointers to their positions within their parents. It turns out that such pointers are not necessary for this internal implementation, because the position will be determined during the search for the starting key. If we simply retain the result of the bottom non-leaf node's binary search, we will have the position to initiate the prefetching appropriately.

This approach is attractive with respect to space overhead, since the only overhead is one additional pointer per bottom non-leaf node. The overhead of updating this pointer should be insignificant, because it only needs to be changed in the rare event that a bottom non-leaf node splits or is deleted. One potential limitation of this approach, however, is that the length of a "chunk" in this jump-pointer array is dictated by the B$^+$-Tree structure, and may not be easily adjusted to satisfy large prefetch distance requirements (e.g., for hiding disk latencies).

In the remainder of this paper, we will use the notations "p$_e$B$^+$-Tree" and "p$_i$B$^+$-Tree" to refer to pB$^+$-Trees with *external* and *internal* jump-pointer arrays, respectively. Further details on the algorithms using external and internal jump pointer arrays can be found in the full paper [7].

4. EXPERIMENTAL RESULTS

To facilitate comparisons with CSB$^+$-Trees, we present our experimental results in a *main-memory* database environment. We begin by describing the framework for the experiments, including our performance simulator and the implementation details of the index structures that we compare. The three subsections that follow present our experimental results for index search, index scan, and updates. Finally, we present a detailed cache performance study for a few of our earlier experiments.

4.1 Experimental Framework

Machine Model. We evaluate the performance impact of *Prefetching B$^+$-Trees* through detailed simulations of fully-functional executables running on a state-of-the-art machine. Since the gap between processor and memory speeds is continuing to increase dramatically with each new generation of machines, it is important to focus on the performance characteristics of machines in the near future rather than in the past. Hence we base our memory hierarchy on the Compaq ES40 [8] (one of the most advanced computer systems announced to date), but we update it slightly to include a dynamically-scheduled, superscalar processor similar to the MIPS R10000 [23] running at a clock rate of 1 GHz. The simulator performs a cycle-by-cycle simulation, modeling the rich details of the processor including the pipeline, register renaming, the reorder buffer, branch prediction, branching penalties, the memory hierarchy (including all forms of contention), etc. Table 2 shows the key parameters of the simulator.

Table 2: Simulation parameters.

Pipeline Parameters	
Clock Rate	1 GHz
Issue Width	4 insts/cycle
Functional Units	2 Integer, 2 FP, 2 Memory, 1 Branch
Reorder Buffer Size	64 insts
Integer Multiply/Divide	12/76 cycles
All Other Integer	1 cycle
FP Divide/Square Root	15/20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	gshare [15]

Memory Parameters	
Line Size	64 bytes
Primary Data Cache	64 KB, 2-way set-assoc.
Primary Instruction Cache	64 KB, 2-way set-assoc.
Miss Handlers	32 for data, 2 for inst.
Unified Secondary Cache	2 MB, direct-mapped
Primary-to-Secondary Miss Latency	15 cycles (plus any delays due to contention)
Primary-to-Memory Miss Latency	150 cycles (plus any delays due to contention)
Main Memory Bandwidth	1 access per 10 cycles

Given the parameters in Table 2, one can see that the normalized memory bandwidth (B)—i.e. the number of cache misses to memory that can be serviced simultaneously—is:

$$B = \frac{T_1}{T_{next}} = \frac{150}{10} = 15. \quad (4)$$

This is slightly pessimistic compared with the actual Compaq ES40 [8], where $B = 16.25$, and is intended to reflect other recent memory system designs [3]. As shown in the full paper [7], small variations in B do not substantively alter the results of our studies.

We compiled our C source code into MIPS executables using version 2.95.2 of the gcc compiler with optimization flags enabled. We added prefetch instructions to the source code by hand, using the gcc `ASM` macro to translate these directly into valid MIPS prefetch instructions.

B$^+$-Trees Studied and Implementation Details. Our experimental study compares pB$^+$-Trees of various node widths w with B$^+$-Trees and CSB$^+$-Trees. We consider both p$_e^w$B$^+$-Trees and p$_i^w$B$^+$-Trees (described earlier in Sections 3.2–3.3 and Section 3.4, respectively). We also consider the combination of both pB$^+$-Tree and CSB$^+$-Tree techniques, which we denote as a pCSB$^+$-Tree.

We implemented bulkload, search, insertion, deletion, and range scan operations for: (i) standard B$^+$-Trees; (ii) p^{w}B$^+$-Trees for node widths $w = 2, 4, 8$, and 16; (iii) p$_e^8$B$^+$-Trees; and (iv) p$_i^8$B$^+$-Trees. For these latter two cases, the node

width $w = 8$ was selected because our experiments showed that this choice resulted in the best search performance (consistent with the analytical computation in Section 2). We also implemented bulkload and search for CSB$^+$-Trees and pCSB$^+$-Trees. Although we did not implement insertion or deletion for CSB$^+$-Trees, we conduct the same experiments as in Rao and Ross [20] (albeit in a different memory hierarchy) to facilitate a comparison of the results. Although Rao and Ross present techniques to improve CSB$^+$-Tree search performance *within* a node [20], we only implemented standard binary search for all the trees studied because our focus is on memory performance (which is the primary bottleneck, as shown earlier in Figure 1).

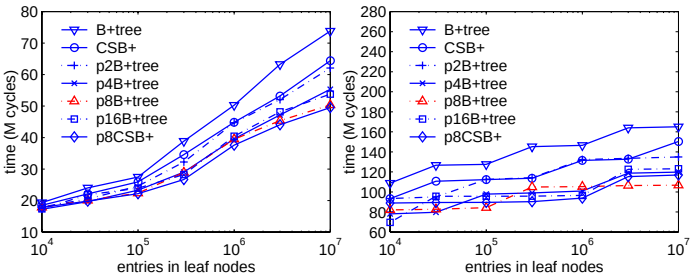
Our pB$^+$-Tree techniques improve performance over a range of `key`, pointer, and `tupleID` sizes. For concreteness, we report experimental results where the `keys`, pointers, and `tupleIDs` are 4 bytes each, as was done in previous studies [19, 20]. As discussed in Section 2, we use a standard B$^+$-Tree node structure, consistent with previous studies. For the B$^+$-Tree, each node is one cache line wide (i.e. 64 bytes). Each non-leaf node contains a `keynum` field, 7 `key` fields and 8 `childptr` fields, while each leaf node contains a `keynum` field, 7 `key` fields, 7 associated `tupleID` fields, and a `next-leaf` pointer. The nodes of the pB$^+$-Trees are the same as the B$^+$-Trees, except that they are wider. So for eight-cache-line-wide nodes, each non-leaf node is 512 bytes and contains a `keynum` field, 63 `key` fields, and 64 `childptr` fields, while each leaf node contains a `keynum` field, 63 `key` fields, 63 associated `tupleID` fields, and a `next-leaf` pointer. For the p_e^8B$^+$-Tree, non-leaf nodes have the same structure as for the pB$^+$-Tree, while each leaf node has a `hint` field and one fewer `key` and `tupleID` fields. The only difference with a p_i^8B$^+$-Tree compared to a pB$^+$-Tree is that each bottom non-leaf node has a `next-sibling` pointer, and one fewer `key` and `childptr` fields. For the CSB$^+$-Tree and the pCSB$^+$-Tree, each non-leaf node has only one `childptr` field. For example, a CSB$^+$-Tree non-leaf node has a `keynum` field, 14 `key` fields, and a `childptr` field. All tree nodes are aligned on a 64 byte boundary when allocated.

For the p_e^8B$^+$-Tree and p_i^8B$^+$-Tree experiments, we need to select the prefetch distance (for both) and the chunk size (for the former). According to equations (2) and (4), we should select $k = \lceil \frac{B}{w} \rceil = \lceil \frac{15}{8} \rceil = 2$. However, as discussed in Section 3, it is often advantageous to slightly increase k in order to create some extra slack. We set $k = 3$, to create extra slack for the prefetching of chunks and non-leaf nodes. (Our sensitivity analysis in [7] showed that selecting $k = 2$, 3, or 4 results in similar scan performance.) As for the chunk size, according to equation (3) and the discussion that follows, we should select c to be at least $\lceil \frac{B}{2m} \rceil = \lceil \frac{15}{16} \rceil = 1$. We conservatively select $c = 8$—i.e. each chunk is 8 cache lines wide—so that each chunk contains 126 leaf pointer fields. (Our sensitivity analysis [7] showed that selecting $c = 1$ through 32 results in similar scan performance.)

4.2 Search Performance

We first evaluate index search performance for B$^+$-Trees, CSB$^+$-Trees, p^{w}B$^+$-Trees (where $w = 2$, 4, 8, and 16), and p^{8}CSB$^+$-Trees (which combine our prefetching approach with CSB$^+$-Trees).

Varying the number of leaf nodes. Figure 7 shows the execution time of 100K random searches after bulkloading 10K, 30K, 100K, 300K, 1M, 3M, and 10M keys into the trees (nodes are 100% full except the root).[5] In the experiments shown in Figure 7(a), search operations are performed one immediately after another (the "warm cache" case); whereas in the experiments shown in Figure 7(b), the cache is cleared between each search (the "cold cache" case). Depending on the operations performed between the searches, the real-world performance of an index search would lie in between the two extremes: closer to the warm cache case for index joins, while often closer to the cold cache case for single value selections. From these experiments, we see that: (i) the B$^+$-Tree has the worst performance; (ii) the trees with wider nodes and prefetching support (pB$^+$-Trees, pCSB$^+$-Tree) all perform better than their non-prefetching counterparts (B$^+$-Tree, CSB$^+$-Tree); and (iii) the p^{8}B$^+$-Tree is comparable to or better than all other pB$^+$-Trees over the entire range of tree sizes. For warm caches, the speedup of the p^{8}B$^+$-Tree over the B$^+$-Tree is between a factor of 1.27 to 1.47. The warm cache speedup of the p^{8}B$^+$-Tree over the CSB$^+$-Tree is between a factor of 1.14 to 1.28 once the tree no longer fits in the L2 cache. Likewise, the cold cache speedups are 1.32 to 1.55 and 1.14 to 1.34, respectively.



(a) warm cache (b) cold cache

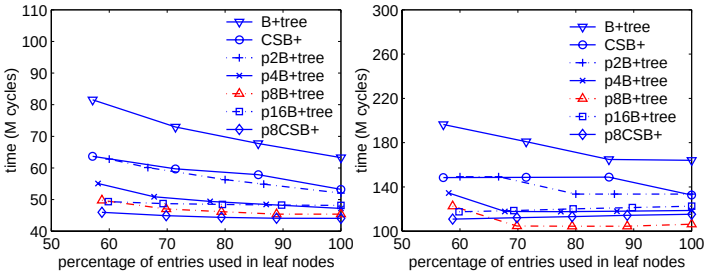
Figure 7: 100K searches after bulkloading 10K-10M keys.

Table 3: The number of levels in trees for Figure 7.

Tree Type	Number of Keys						
	10K	30K	100K	300K	1M	3M	10M
B$^+$-Tree	5	6	6	7	7	8	8
CSB$^+$-Tree	4	5	5	5	6	6	7
p^{2}B$^+$-Tree	4	4	5	5	6	6	6
p^{4}B$^+$-Tree	3	3	4	4	4	5	5
p^{8}B$^+$-Tree	3	3	3	4	4	4	4
p^{16}B$^+$-Tree	2	3	3	3	3	4	4
pCSB$^+$-Tree	3	3	3	3	3	4	4

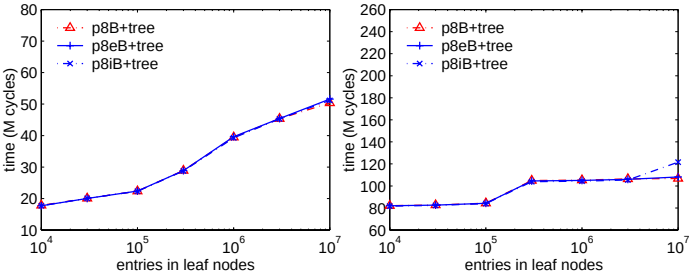
The cold cache curves provide insight into the index search performance. The trend of every single curve is clearly shown in the cold cache experiment: the curves all increase in discrete large steps, and within the same step they increase only slightly. The large steps for a curve occur when the number of levels in the tree increases. This can be verified by examining Table 3, which depicts the number of levels in the tree for each data point plotted in Figure 7. Within a step, additional leaf nodes result in more keys in the root node (the other nodes in the tree remain full), which in turn increases the cost to search the root. The step-up trend is blurred in the warm cache curves because the top levels of the tree may remain in the cache. For different curves, we can see that generally the higher the tree structure, the larger the search cost; when trees are of the same height, the smaller node size yields better performance. We

[5] Note that throughout this paper, "K" and "M" correspond to 1000 and 1,000,000, respectively, except for when we refer to the size of a memory structure (e.g., a cache), in which case they correspond to 1024 and 1,048,576, respectively.



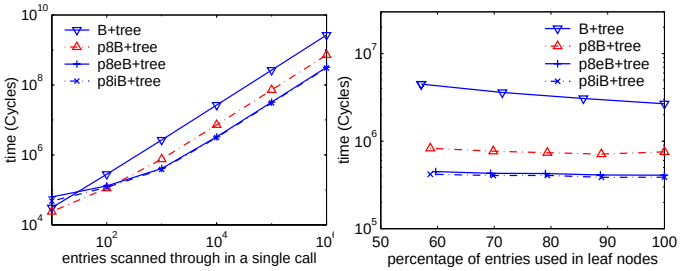
(a) warm cache (b) cold cache

Figure 8: 100K searches after bulkloading 3M keys with various bulkload factors.



(a) warm cache (b) cold cache

Figure 9: 100K searches after bulkloading 10K to 10M keys into p^8B^+-Trees with and without jump-pointer arrays.



(a) varying number of `tupleIDs` scanned for each request (b) varying bulkload factor

Figure 10: Range scan performance.

conclude that the performance gains for wider nodes stem mainly from the resulting decrease in tree height.

Another observation worth mentioning is that when the number of levels are the same, the p^2B^+-Tree and the CSB^+-Tree have very similar performance. This is because the second cache line in a p^2B^+-Tree node stores pointers, and the cost of retrieving these second lines is partly hidden by the key comparisons. By eliminating all but one pointer, the CSB^+-Tree has almost the same number of keys as the p^2B^+-Tree, resulting in similar key comparison costs.

Varying the bulkload factor. Figure 8 shows the effect on search performance of varying the bulkload factor. All the trees are bulkloaded with 3M ⟨`key`, `tupleID`⟩ pairs, with bulkload factors of 60%, 70%, 80%, 90%, and 100%. Because the actual number of used entries in leaf nodes in an experiment is the product of the bulkload factor and the maximum number of slots (rounded to the nearest integer), we computed and used the true percentage of used entries when plotting the data—hence they may not be aligned with the target bulkload factors. As in the previous experiments, Figure 8 shows that: (i) the B^+-Tree has the worst performance; (ii) the trees with wider nodes and prefetching support (pB^+-Trees, $pCSB^+$-Tree) all perform better than their non-prefetching counterparts (B^+-Tree, CSB^+-Tree); and (iii) the p^8B^+-Tree is the best of all the pB^+-Trees.

In the cold cache experiment, we see a step-down pattern in the curves: the steps correspond to the number of levels in the trees, since the tree height decreases (in a step-wise fashion) as the bulkload factor increases. Within a step, however, the curves increase slightly. This is because in our bulkload algorithms, the bulkload factor also determines the number of keys in non-leaf nodes. So the larger the bulkload factor, the larger the number of keys in each non-root node, and hence the larger the key comparison cost.

In the full paper [7], we also present experimental results for *mature* trees [20], created by bulkloading 10% of the ⟨`key`, `tupleID`⟩ pairs and then inserting the remaining 90%. We find similar performance for mature trees as for trees immediately after bulkloads.

Searches on trees with jump-pointer arrays. Our next experiment determines whether the different structures for speeding up range scans have an impact on search performance. We use node width $w = 8$ for these experiments, because the p^8B^+-Tree resulted in the best search performance among the pB^+-Trees. Figure 9 compares the search performance of the p^8B^+-Tree, the $p^8_eB^+$-Tree, and the $p^8_iB^+$-Tree. The same experiments as in Figure 7 were performed. Recall that both the $p^8_eB^+$-Tree and the $p^8_iB^+$-Tree consume space in the tree structures relative to the p^8B^+-Tree: the maximum number of keys in leaf nodes is one fewer for the $p^8_eB^+$-Tree, and the maximum number of keys in bottom non-leaf nodes is one fewer for the $p^8_iB^+$-Tree. Figures 9(a) and 9(b) show that these differences have a negligible impact on search performance. In one cold cache case, when 10M keys are in the tree, the $p^8_eB^+$-Tree suffers from having one more level than the other two trees, but otherwise both the warm and cold cache performances are basically the same for all three trees, over the entire range of 10K to 10M keys.

4.3 Range Scan Performance

In our next set of experiments, we evaluate the effectiveness of our techniques for improving range scan performance. We compare B^+-Trees, p^8B^+-Trees, $p^8_eB^+$-Trees, and $p^8_iB^+$-Trees. As indicated above, we restrict our attention to node width $w = 8$ because this is the best width for searches, which are presumed to occur more frequently than range scans. As discussed in Section 4.1, we set the prefetching distance to 3 nodes and the chunk size to 8 cache lines.

Varying the range size and the bulkload factor. Figure 10 shows the execution time of range scans while varying (a) the number of `tupleIDs` to scan per request (i.e. the size of the range), or (b) the bulkload factor. Because of the large performance gains for pB^+-Trees, the execution time is shown on a logarithmic scale. In Figure 10(a), the trees are bulkloaded with 3M ⟨`key`, `tupleID`⟩ pairs, using a 100% bulkload factor. Then 100 random starting `keys` are selected, and a range scan is requested for m `tupleIDs` starting at that starting key value, for m = 10, 100, 1K, 10K, 100K, and 1M. The execution time plotted for each m is the total for the 100 starting `keys`. In Figure 10(b), the trees are bulkloaded with 3M ⟨`key`, `tupleID`⟩ pairs, with bulkload factors of 60%, 70%, 80%, 90%, and 100%. 100

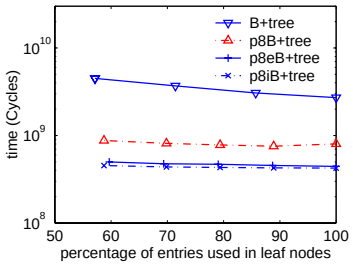
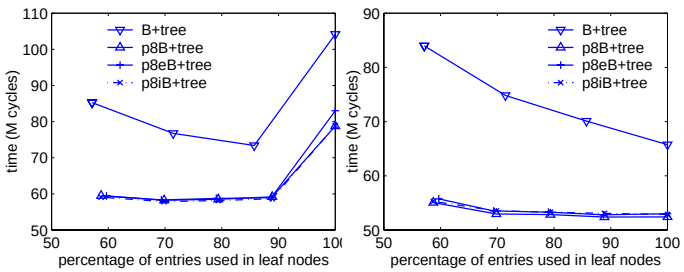
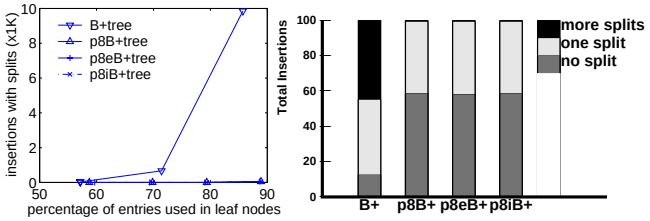


Figure 11: Large (segmented) range scans with various bulkload factors.



(a) insertion (b) deletion

Figure 12: 100K update operations with various bulkload factors.



(a) trees with empty slots (b) trees that are 100% full

Figure 13: Analysis of insertion behavior.

random starting `keys` are then selected, and a range scan is requested for 1000 tuple IDs starting at that value. Between the range scan requests, the caches are cleared to more accurately reflect scenarios in which range scan requests are interleaved with other database operations or application programs (which would tend to evict any cache-resident nodes).

As we see in Figure 10, $p^8_e B^+$-Trees and $p^8_i B^+$-Trees achieve a *factor of 6.5 to 8.7 speedup* over standard B^+-Trees for range scans of 1K to 1M `tupleIDs`. As the bulkload factor decreases, the number of leaf nodes to be scanned increases (since we must skip an increasing number of empty slots), and hence our prefetching schemes achieve even larger speedups. Figure 10 also shows the contribution of both aspects of our pB^+-Tree design to overall performance. First, extending the node size and simultaneously prefetching all cache lines within a node while scanning (and performing the initial search)—similar to what was illustrated earlier in Figure 3(b)—results in a speedup of 3.5 to 3.7, as shown by the difference between $p^8 B^+$-Trees and B^+-Trees in Figure 10. Second, by also using jump-pointer arrays to prefetch ahead across the (extra-wide) leaf nodes, we see an additional speedup of roughly 2 in this case, as shown by the improvement of both $p^8_e B^+$-Trees and $p^8_i B^+$-Trees over $p^8 B^+$-Trees in Figure 10. Since both $p^8_e B^+$-Trees and $p^8_i B^+$-Trees achieve nearly identical performance, there does not appear to be a compelling need to build an external (rather than an internal) jump-pointer array, at least for these system parameters. (Note that this conclusion depends upon the ratio of w and k; in other scenarios with different ratios—e.g., when prefetching to hide *disk* as well as memory latencies—the flexibility of an external jump-pointer array may be needed.)

When scanning far fewer than 1K `tupleIDs`, however, the startup cost of our prefetching schemes becomes noticeable. For example, when scanning only 100 `tupleIDs`, pB^+-Trees are only twice as fast as standard B^+-Trees. When scanning only 10 `tupleIDs`, $p^8 B^+$-Trees are only slightly faster than B^+-Trees, and $p^8_e B^+$-Trees and $p^8_i B^+$-Trees are actually slower. This suggests a scheme where jump-pointer arrays are only exploited for prefetching if the expected number of `tupleIDs` within the range is significant (e.g., 100 or more). This estimate of the range size could be computed either by using standard query optimization techniques such as histograms, or else by simultaneously searching for both the starting and ending keys to see how far apart they are.

Large segmented range scans. We now consider the behavior of large range scans. In practice, these large scans are often broken up into smaller segments either to permit other operations and queries to proceed, or else to avoid overflowing the return buffer. For example, an indexed scan providing sorted input to a sort-merge join operator will have its return buffer consumed at a rate dependent on the data profile of the other input to the join. Figure 11 shows the execution time for performing segmented range scans: each scan consists of a search for the starting key followed by 1000 range scan requests, each of which scans (and places into the return buffer) the next segment of 1000 ⟨`key`, `tupleID`⟩ pairs, resulting in a total of 1M pairs. The trees are bulkloaded with 3M ⟨`key`, `tupleID`⟩ pairs, with bulkload factors ranging from 60% to 100%. The reported execution times are the total for 100 segmented range scans, starting from 100 randomly selected starting `keys`. As we see in Figure 11, the performance gains for segmented range scans are similar to what we saw earlier in Figure 10 for non-segmented range scans.

4.4 Update Performance

In addition to improving search and scan performance, another one of our goals is to achieve good performance on *updates*, especially since this had been a problem with earlier cache-sensitive index structures [19, 20]. To quantify the impact of pB^+-Trees on update performance, Figure 12 shows the execution time for 100K random insertions or deletions on a tree bulkloaded with 3M ⟨`key`, `tupleID`⟩ pairs, with bulkload factors ranging from 60% to 100%, and with warm caches. (The cold cache results exhibit the same trends [7].) As we see in Figure 12, all three pB^+-Tree schemes (i.e. $p^8 B^+$-Trees, $p^8_e B^+$-Trees, and $p^8_i B^+$-Trees) perform roughly the same, and all are significantly faster than the B^+-Tree. For example, when the bulkload factor is 100%, the pB^+-Trees achieve at least a 1.24 speedup over the B^+-Tree for both insertions and deletions. This result may appear somewhat surprising, given the additional overheads of maintaining the external jump-pointer arrays for $p^8_e B^+$-Trees.

There are two primary factors contributing to the faster update times for pB^+-Trees compared with the B^+-Tree. First, search is an integral part of both insertion and deletion, and our pB^+-Trees enjoy faster search times due to their wider nodes (as we saw earlier in Section 4.2). Second, node splits occur less frequently for wider nodes. Let us start by considering trees that are not full—i.e. with bulk-

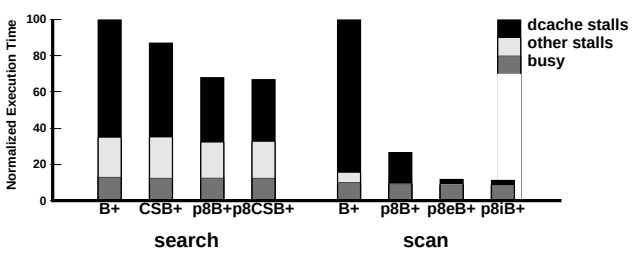


Figure 14: Impact of various pB$^+$-Trees on the cache performance of index search and range scan.

load factors ranging from 60% to 90%. Figure 13(a) shows that when the bulkload factor is within this range, the number of the 100K insertions that cause node splits is extremely small for each of the pB$^+$-Trees, and is even less than 10% for B$^+$-Trees. Given that Figures 12(a) and (b) show similar trends to Figure 8 within this range of bulkload factors, we conclude that the dominant effect is improved search time for these less-than-full trees.

In contrast, when the trees are full, many insertions will cause node splits, as shown in Figure 13(b). Due to their smaller nodes, B$^+$-Trees suffer far more node splits than pB$^+$-Trees, and over 40% of the insertions result in multiple splits due to splitting *non-leaf nodes*. Hence although the cost of each node split in a pB$^+$-Trees is greater, this cost is more than offset by the reduced frequency of node splits and the improved search times.

Turning our attention to deletion performance in Figure 12(b), since both pB$^+$-Trees and B$^+$-Trees use lazy deletion, very few deletions result in a deleted node or a key redistribution. Hence the performance gains for pB$^+$-Tree deletions are due solely to faster search times.

4.5 Cache Performance

Finally, our last set of experiments present a more detailed cache performance study, using two representative experiments: one for index search and one for index range scan. A central claim of this paper is that the demonstrated speedups for pB$^+$-Trees are obtained by effectively limiting the exposed miss latency of previous approaches. In these experiments, we confirm that claim.

Our starting point is the experiments presented earlier in Figure 1 which illustrated the poor cache performance of existing B$^+$-Trees on index search and scan. We reproduce those results now in Figure 14, along with several variations of our pB$^+$-Trees. The bars on the left (labeled "search") correspond to the experiment shown earlier in Figure 7(a) with 10M ⟨`key`, `tupleID`⟩ pairs bulk-loaded, and the bars on the right (labeled "scan") correspond to the experiment shown earlier in Figure 10(a) with 1M `tupleID`s scanned.

Each bar in Figure 14 represents execution time normalized to a B$^+$-Tree, and is broken down into the following three categories that explain what happened during all potential *graduation slots*.[6] The bottom section (*busy*) is the number of slots where instructions actually graduate. The other two sections are the number of slots where there is no graduating instruction, broken down into data cache stalls and other stalls. Specifically, the top section (*dcache stalls*) is the number of such slots that are immediately caused by the oldest instruction suffering a data cache miss, and the middle section (*other stalls*) is all other slots where instructions do not graduate. Note that the *dcache stalls* section is only a first-order approximation of the performance loss due to data cache misses: these delays also exacerbate subsequent data dependence stalls, thereby increasing the number of *other stalls*.

As we see in Figure 14, pB$^+$-Trees significantly reduce the amount of exposed miss latency (i.e. the *dcache stalls* component of each bar). For the index search experiments, we see that while CSB$^+$-Trees eliminated 20% of the data cache stall time that existed with B$^+$-Trees, p^{8}B$^+$-Trees eliminate *45%* of this stall time, thus resulting in an overall speedup of *1.47* (compared with 1.15 for CSB$^+$-Trees). A significant amount of data cache stall time still remains for index searches, since we still experience the full miss latency each time we move down a level in the tree (unless the node is already in the cache due to previous operations). Eliminating this remaining latency appears to be difficult, as we will discuss in the next section. In contrast, we achieve nearly ideal performance for the index range scan experiments shown in Figure 14, where both p$_e^8$B$^+$-Trees and p$_i^8$B$^+$-Trees eliminate *97%* of the original data cache stall time, resulting in an impressive *eightfold overall speedup*. These results demonstrate that the pB$^+$-Tree speedups are indeed primarily due to a significant reduction in the exposed miss latency.

5. DISCUSSION

We now discuss several possible improvements to pB$^+$-Trees and related issues. While our approach of using prefetching to create wider nodes improves search performance by a factor of 1.2–1.5, we still suffer a full cache miss latency at each level of the tree. Unfortunately, this is a very difficult problem to solve given: (i) the data dependence through the child pointer; (ii) the relatively large fanout of the tree nodes; and (iii) the fact that it is equally likely that any child will be visited (assuming uniformly distributed random search keys). While one might consider prefetching the children or even the grandchildren of a node in parallel with accessing the node, there is a duality between this and simply creating wider nodes. Compared with our approach, prefetching children or grandchildren suffers from: (i) additional storage overhead for the children and grandchildren pointers, and (ii) the restriction that the "size" of a node (i.e. the number of cache lines prefetched) can only grow by multiples of the tree fanout.

Extending the idea of adding pointers to the bottom non-leaf nodes, it is possible to use *no additional pointers at all*. Potentially, we could retain all the pointers from the root to the leaf during the search, and then keep moving this set of pointers, sweeping through the entire range prefetching the leaf nodes. Note that with wider nodes, trees are shallower and this scheme may be feasible.

Lehman and Carey, in an early paper on index structures for main memory databases, proposed and studied the T-Tree [11, 12]. At the time of their study (the mid-80's), the T-Tree outperformed the B$^+$-Tree, and was considered the index structure of choice for main memory databases for over a decade. However, more recent studies have shown that the B$^+$-Tree outperforms the T-Tree on modern processors [19], due in large part to the exponential growth these past 15 years in cache miss latency relative to processor speed.

Previous work has also considered key compression schemes (e.g., [4, 6, 9]), in order to pack more keys into an index node. As with CSB$^+$-Trees, these techniques can be used in conjunction with our approach, as desired.

Although our discussions and experiments have focused

[6] The number of graduation slots is the issue width (4 in our simulated architecture) multiplied by the number of cycles. We focus on graduation slots rather than issue slots to avoid counting speculative operations that are squashed.

on main memory databases, pB$^+$-Trees can also be used to improve both the I/O performance and the memory performance of disk-resident databases. Because the index node size for a disk-resident database is typically a disk page of 4KB, the fanout is much larger than with main memory indices. This may effect the benefits of using even wider nodes for searches. However, our range scan prefetching techniques applied to pages would likely continue to have a significant benefit. Furthermore, main memory performance is important even for disk-resident databases, so it would be interesting to apply our methods for both cache lines and pages, and quantify the overall performance gains.

6. CONCLUSIONS

While eliminating child pointers through data layout techniques has been shown to significantly improve main memory B$^+$-Tree search performance, a large fraction of the execution time for a search is still spent in data cache stalls, and index insertion performance is hurt by these techniques. Moreover, the cache performance of index scan (another important B$^+$-Tree operation) has not been studied. In this paper, we explored how prefetching could be used to improve the cache performance of index search, update, and scan operations. We proposed the *Prefetching B$^+$-Tree* (pB$^+$-Tree) and evaluated its effectiveness in modern memory systems.

We showed that the optimal B$^+$-Tree node size is often wider than a cache line on a modern machine, when prefetching is used to retrieve the pieces of a node, effectively overlapping multiple cache misses. Our results can be summarized as follows:

- For index search, this prefetching technique achieves a speedup of 1.27 to 1.55 over the B$^+$-Tree, by decreasing the height of the tree.
- For index updates (insertions and deletions), the technique achieves a speedup of 1.24 to 1.52 over the B$^+$-Tree, due to the faster search and the less frequent node splits with wider nodes.
- For index scan, the technique achieves a speedup of 3.5 to 3.7 over the B$^+$-Tree, again due to the faster search and wider nodes. Moreover, we proposed *jump-pointer arrays*, which enable effective range scan prefetching across node boundaries. Overall, the pB$^+$-Tree achieves a speedup of **6.5 to 8.7** over the B$^+$-Tree for range scans. We proposed two alternative implementations of jump-pointer arrays, with comparable performance.

From our results, we conclude that the cache performance of B$^+$-Tree indices can be greatly improved by exploiting the prefetching capabilities of state-of-the-art computer systems. We believe that this work makes an important contribution towards applying prefetching techniques to advantage throughout a DBMS.

Acknowledgements

We thank Berni Schiefer at IBM for his many helpful comments regarding this work. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship and by a Faculty Development Award from IBM.

7. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th VLDB*, pages 266–277, Sept. 1999.

[2] L. A. Barroso, K. Gharachorloo, and E. D. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th ISCA*, pages 3–14, June 1998.

[3] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *Proceedings of the 6th HPCA*, pages 3–14, Jan. 2000.

[4] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Trans. on Database Systems*, 2(1):11–26, March 1977.

[5] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. In *Proceedings of the 41st IEEE FOCS*, pages 399–409, Nov. 2000.

[6] P. Bohannon, P. McIlroy, and R. Rastogi. Improving Main-Memory Index Performance with Partial Key Information. In *Proceedings of the 2001 SIGMOD Conference*, May 2001.

[7] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance through Prefetching. Technical Report CMU-CS-00-177, School of Computer Science, Carnegie Mellon University, Dec. 2000.

[8] Z. Cvetanovic and R. E. Kessler. Performance Analysis of the Alpha 21264-Based Compaq ES40 System. In *Proceedings of the 27th ISCA*, pages 192–202, June 2000.

[9] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of the 14th ICDE*, pages 370–379, Feb. 1998.

[10] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 25th ISCA*, pages 15–26, June 1998.

[11] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th VLDB*, pages 294–303, Aug. 1986.

[12] T. J. Lehman and M. J. Carey. Query Processing in Main Memory Database Management Systems. In *Proceedings of the 1986 SIGMOD Conference*, pages 239–250, May 1986.

[13] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th ASPLOS*, pages 222–233, Oct. 1996.

[14] C.-K. Luk and T. C. Mowry. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Transactions on Computers*, 48(2):134–141, Feb. 1999.

[15] S. McFarling. Combining Branch Predictors. Technical Report WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th ASPLOS*, pages 62–73, Oct. 1992.

[17] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the 1994 SIGMOD Conference*, pages 233–242, May 1994.

[18] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the 8th ASPLOS*, pages 307–318, Oct. 1998.

[19] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th VLDB*, pages 78–89, Sept. 1999.

[20] J. Rao and K. A. Ross. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proceedings of the SIGMOD 2000 Conference*, pages 475–486, May 2000.

[21] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th VLDB*, pages 510–521, Sept. 1994.

[22] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, New York, New York, 3rd edition, 1997.

[23] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.