REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Practical Performance Considerations for the Prefetching B$^+$-tree

Árni Már Jónsson
Master of Science
June 2006

Supervisor:
Björn Þór Jónsson
Associate Professor

Reykjavík University - Department of Computer Science

M.Sc. Thesis

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

PRACTICAL PERFORMANCE CONSIDERATIONS FOR THE PREFETCHING B$^+$-TREE

by

Árni Már Jónsson

Thesis submitted to the Department of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of
Master of Science

June 2006

Abstract

In recent years the relative speed difference between CPUs and main-memory has become so great that many applications, including database management systems, spend much of their time waiting for data to be delivered from main-memory. In particular, B^+-trees have been shown to utilize cache memory poorly, triggering the development of many cache-conscious indices. While early studies of cache-conscious indices used simulation models, the trend has recently swung towards performance measurements on actual computer architectures. This thesis is part of this trend towards the deployment of cache-conscious structures "in the field". We study the performance of the pB^+-tree on the Itanium 2 processor, focusing on various implementation choices and their effect on performance.

Útdráttur

Afkastamunur milli örgjörva og aðalminnis hefur aukist síðustu ár. Þessi munur er orðinn svo mikill að mörg hugbúnaðarkerfi, þ.á.m. gagnasafnskerfi, eyða miklum tíma í að bíða eftir að gögn séu sótt í aðalminni. Sér í lagi hefur verið sýnt að leitarvísirinn B$^+$-tré notar skyndiminni á mjög óhagkvæman hátt, sem hefur hrundið af stað þróun á minnisvænum leitarvísum. Fyrri rannsóknir á minnisvænum leitarvísum hafa einkum notast við hermilíkön til afkastamælinga, en í nýlegum rannsóknum hafa afköst verið mæld á raunverulegum tölvukerfum. Þessi ritgerð er hluti af þessari þróun sem hefur það að markmiði að koma minnisvænum leitarvísum í almenna notkun. Við mælum afköst á leitar vísinum pB$^+$-tré, sérstaklega með tilliti til útfærsluatriða og áhrifa þeirra á afköst.

Acknowledgements

This work was partially supported by Rannís Research Fund Grant 04004031 and a Graduate Research Fund FS Grant.

We would like to thank Michael Samuel, Anders Uhl Pedersen and Philippe Bonnet of DIKU, for access to their implementation of the pCSB$^+$-tree, and for the use of their Itanium 2 server, without which much of this work would not have been possible.

Publications

Parts of this work have been accepted for publication in the Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems [6].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For the last 20 years CPU speeds have been increasing at a much faster rate than memory speeds. As a consequence, many modern applications spend much of their time waiting for data to be fetched from memory. Database Management Systems (DBMSs) are no exception, and recent studies [2, 5] show that less than half the CPU time used by commercial DBMSs is spent on useful computations. One of the main components of DBMSs are indices, which historically have been optimized to minimize disk I/O. DBMSs do quite well in hiding the disk I/O latency, but with the growing gap between CPU and memory speeds, index performance is becoming increasingly bound by memory speeds. This has triggered the development of many memory based or “cache-conscious” indices, which use memory more efficiently. The main contribution of this thesis is an implementation of one such index, the “Prefetching B^+-tree” (pB^+-tree), and measurements of its performance on a modern server CPU.

1.1 The Main-Memory Bottleneck

Modern computers are based on the von Neumann architecture [29], which separates the processing unit (CPU) and storage (main-memory). The trend in the last two decades has been that CPU speed is increasing at a rate of about 60% per year, whereas main-memory speed has only been increasing by about 10% per year [13]. The relative speed difference has become so great, in fact, that in many applications CPUs spend much of their time waiting for data to be delivered from main-memory [22].

The common method of dealing with this main-memory bottleneck is the use of cache memory (or cache), which is a limited size, high-speed memory stored on the same chip as the CPU [28].

When referenced data is found in the cache, it can be read very quickly by the CPU. Otherwise, its “cache-line” must be transferred from main-memory into the cache (termed a “cache miss”). Cache misses take much longer than reading data (e.g., on Pentium 4 a cache miss can cost as many as 276 CPU cycles [33]). Typical modern architectures have two or three layers of cache memory, successively smaller and faster as they are closer to the CPU.

Two key techniques have been used to reduce the effects of cache misses. First, by improving the locality of memory accesses, fewer misses are seen as data brought into the cache is utilized well. Second, many architectures allow the compiler (or application programmer) to prefetch data into the cache, reducing the performance degradation due to cache misses.

1.1.1 Cache-Conscious Indices

Database Management Systems (DBMSs) are not exempt from the main memory bottleneck. In their seminal paper, Ailamaki et al. [2] showed that less than half of the CPU time for commercial DBMSs is spent on computations. A recent follow-up study showed that this situation is not improving [5], indicating that commercial DBMSs are not being engineered to cope with the main-memory bottleneck.

Indices—predominantly B$^+$-trees—are a key performance component of DBMSs. Unfortunately, however, B$^+$-trees have been shown to utilize cache memory poorly [9], triggering the development of many cache-conscious indices. The CSS-tree [26] and CSB$^+$-tree [25] improve cache performance by not storing pointers to all the children of a node, effectively compacting the index structure and improving locality. The pB$^+$-tree [9] has nodes that are several cache-lines in width and uses prefetching to avoid perceived cache latency during index traversal. In [27] it was shown that these two techniques are complementary: the CSB$^+$-tree can be implemented using wide nodes and node prefetching.

While early studies of cache-conscious indices primarily used simulation models, the trend has recently swung towards performance measurements on actual computer architectures. In [27], the CSB$^+$-tree was implemented and measured using the Itanium 2 processor. In [8] the pB$^+$-tree was also measured using the Itanium 2 processor. Ultimately, of course, the goal of this work is the integration of cache-conscious indices into DBMSs.

1.2 Contributions

This thesis presents the following contributions:

- A survey of the main-memory bottleneck and methods to reduce its effect
- A survey of cache-conscious indices and an analytical model of their cache-performance
- An experimental framework used to automate performance measurements
- An implementation of the pB$^+$-tree, optimized for the Itanium 2 processor
- Experimental results showing performance benefits of implementation and architecture specific optimizations

- Experimental results showing that prefetching has considerable performance benefits

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 surveys the growing gap between main-memory and CPU speeds, along with techniques used to reduce its effect. Chapter 3 reviews common tree indices, both disk-based and main-memory based, as well as an analysis of their cache-performance. Chapter 4 presents the experimental setup and the experimental framework used. Chapter 5 describes the implementation of the pB$^+$-tree and experimental results showing the performance effects of various implementation and architecture specific optimizations. Chapter 6 describes the results of experiments showing the performance effects of prefetching. Chapter 7 concludes.

Chapter 2

The Memory Bottleneck

As mentioned in the introduction, modern computers are based on the von Neumann architecture [29], which separates the CPU and main-memory. This separation has created the so-called *von Neumann bottleneck*, because (for many workloads) CPUs are now capable of processing data much faster than it can be delivered from main-memory. The term was coined by John Backup in his 1977 ACM Turing award lecture [3]:

> Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.

There are many techniques used which reduce the effect of this *memory-bottleneck*. In this chapter we will cover the fundamentals of *cache-memory* [13], the most important of these. We will also present the main results of a short paper by Wulf and McGee from 1995 [31], which argues that (even given very optimistic assumptions) in a few years many programs will be limited by memory performance— they will have hit a *memory-wall*. Lastly, we will describe some software techniques which reduce the effect of the memory-bottleneck.

2.1 Cache-Memory

Cache-memory (or simply cache) is a small, high speed memory. Unlike the main-memory, which is situated across the *memory-bus*, it is stored on the same chip as the CPU. The cache has much lower latency then the main-memory, since it is closer to the CPU, and runs on the same clock frequency. The cache copies the most frequently used data from main-memory, so that it does not need to be transferred over the memory-bus as often.

The disk, main-memory and cache form a *memory-hierarchy*, where each level has less capacity than the level below, but lower latency. In addition, most modern CPUs have up to 3 levels of cache, where the topmost level is situated closer to the CPU core than the others.

Each new generation of computers has both increased the capacity and decreased the latency of main-memory. In the 1970s the main computer performance bottleneck of computers was moving data between the disk and main-memory. In those days CPUs did not have any on-chip cache, since the speed difference between CPUs and main-memory was very small. The trend in recent years has been to increase the speed of CPUs and the capacity of main-memory, with main-memory speeds growing at a lower rate. This is what has created the memory-bottleneck, and necessitated the development of the on-chip cache.

The smallest amount of data moved between main-memory and cache is the *cache-line*. It is a continuous piece of data, usually ranging from 8 to 512 bytes, depending on the computer architecture. The cache-memory has much lower capacity than main-memory, and a *replacement policy* is used to decide which cache-line is thrown out to make room for a new one. The replacement policy must predict which cache-line in the cache is least likely to be used in the future. As with most other memory and cache structures, the most popular replacement is *least-recently-used*, which replaces the cache-line which was least recently used. Other replacement policies include *least-frequently-used*, which replaces the least referenced cache-line, and *random*, which selects cache-lines to replace at random.

When a line in the cache is modified, the changed data data must eventually be reflected at the level below. The *write policy* of a cache controls when the lower level is updated. A *write through* policy updates the lower level after each write. A *write back* policy updates the lower level only when the cache-line is replaced, and only if it has been modified. Under that policy the cache-line has a *dirty bit* which is set when it is modified.

When a memory location is accessed, the CPU must check whether it is in the cache. The number of locations it needs to check in order to detect this is called the *associativity* of the cache. A *fully associative* cache needs to check every location, and a *direct associative* cache needs to check only one. An *n-way set associative* cache needs to check n locations. A cache with higher associativity has higher latency than cache with lower associativity, since more locations need to be checked. On the other hand, higher associativity implies that there is more choice of cache-lines to replace, thus increasing the probability of contemporaneously used data being in the cache. However, increasing associativity beyond a certain point has decreasing performance improvements. Figure 2.1 shows how increased associativity decreases the miss rate, and that a fully associative cache is only marginally better than an 8-way associative cache. The data used to plot Figure 2.1 was obtained by Cantin and Hill [7], by measuring the miss-rate of various SPEC CPU2000 benchmark programs, and averaging them.

Hill and Smith [14] classify cache-misses into 3 categories:

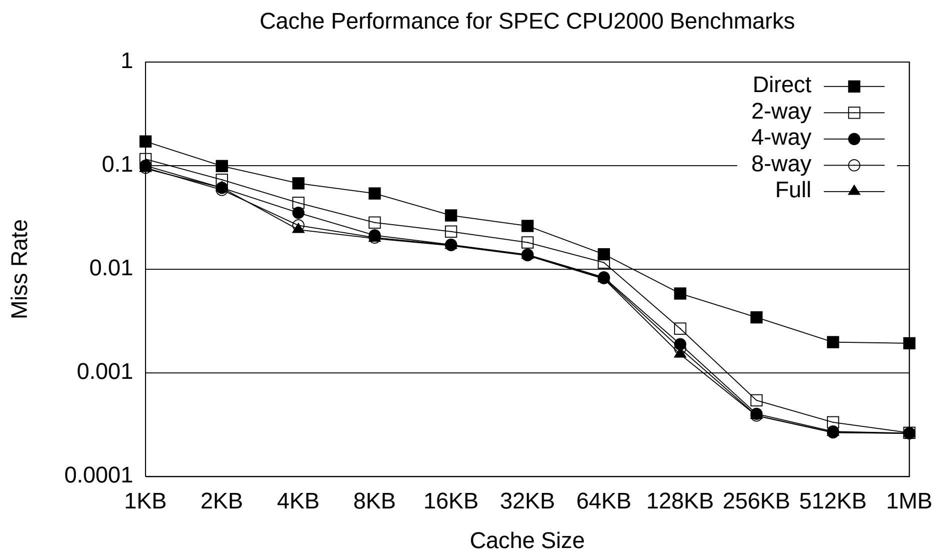


Figure 2.1: The average of miss rates for various SPEC CPU2000 benchmarks.

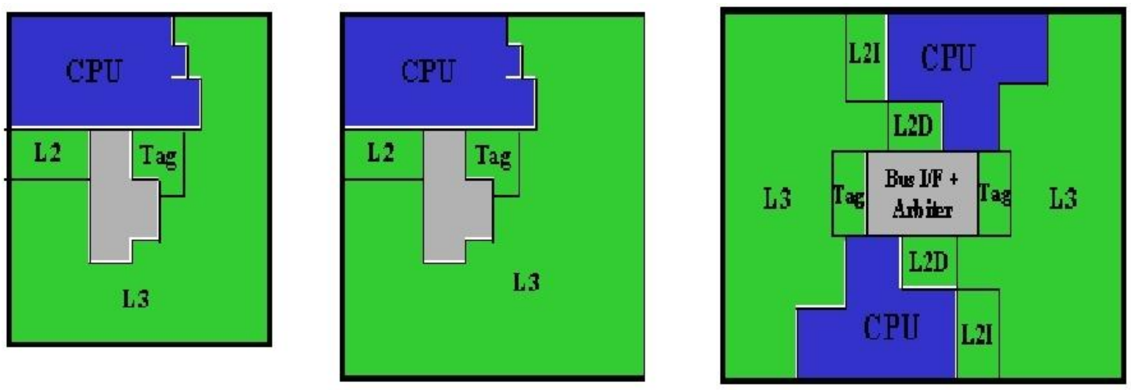
- *Compulsory misses* occur when the cache-line is first referenced.
- *Capacity misses* occur when the cache-line being accessed has been replaced because the cache was full.
- *Conflict misses* occur when the cache-line accessed has been replaced because of limited associativity, but would be found in a fully associative cache.

Out of these 3 types of misses, only compulsory misses are unavoidable. The number of capacity misses can be reduced by increasing the size of the cache, and the number of conflict misses can be reduced by increasing the associativity of the memory; a fully associative memory has none.

Modern CPUs have up to 3 levels of cache, each positioned closer the CPU core. Figure 2.2 shows the layout of 3 versions of the Itanium 2 processor. The Itanium 2 processor has 3 levels of cache, and the latest generation has 2 CPU cores. It is noteworthy that 2/3 of the die area is occupied by cache, which shows what an important role it plays in modern computing.

2.2 The Memory-Wall

Wulf and McKee published a short paper in 1995 [31] which reflected on the trend of diverging CPU and memory speeds and its effect on program performance. The previous decade had showed that CPU and memory speeds were diverging at an alarming rate. The question they tried to answer was when this would become a serious performance issue. The authors made very optimistic assumptions about future developments, and yet



a) b) c)

Figure 2.2: The layout of 3 versions of the Itanium 2 processor.
a) Madison with 6M L3 b) Madison with 9M b) Montecito (dual core) with 24MB L3.

came to the conclusion that in the near future, programs would generally become memory bound.

The paper first presents a simple equation for the average memory access time, assuming there is only one level of cache. If t_c is the access time for the cache-memory, t_m is the access time for main-memory, and p is the cache-memory hit-rate, then the average memory access time t_{avg} is given by:

$$t_{avg} = p \times t_c + (1 - p) \times t_m$$

Then, some optimistic assumptions are made. First, that there are no conflict or capacity misses; in other words, the cache is perfect. Second, that t_c is constant in terms of CPU cycles. Lastly, that 20% of all instructions access memory.

As t_c and t_m diverge, t_{avg} will grow, and when $t_{avg} \geq \frac{100\%}{20\%} = 5$ programs will hit a "Memory-Wall" . At that time their performance will be entirely dependent on main-memory speed, since the CPU will spend more time on the memory instructions than the remaining 80%.

To put this into perspective, the authors took then current estimates of CPU and memory speed trends. Given these trends, the appropriate values for the variables were $t_c \times 4 = t_m$, $t_{avg} = 1$ and $p = 0.99$. CPU speeds were assumed to increase at a rate of 80% per year and main-memory speeds at 7% per year. Given these, p_{avg} will become 1.53 in 5 years, 8.25 in 10 years and 98.8 in 15 years. These numbers can of course be adjusted, as the authors do, but they still come to the conclusion that the "Memory-Wall" was only 10-20 years away in 1995.

The main assumption that there is a fixed hit-rate is questionable, since there are programs which access very little memory. In a follow-up article by McKee [22], it is indeed noted that some applications, such as multimedia applications, are not affected much by memory speeds. But others, such at transaction processing applications and high-performance computing (HPC), see up to 65% and 95% CPU waiting time, respectively.

2.3 Techniques

Many techniques have been proposed, which help in reducing the effect of the memory bottleneck. In this section we will discuss a few software techniques which make applications more cache-conscious, and one hybrid software/hardware technique. For a more detailed survey of both software and hardware based techniques, we refer to a survey by Machanick [20].

2.3.1 Data Placement

Since the smallest amount of data moved between main and cache-memory is a cache-line, it makes sense to use as much of the data as possible before it is replaced. Data should also be accessed such that frequently used data does not map to the same cache-lines, given a cache with small associativity.

Chilimbi et al. [11] discuss two techniques to achieve this. *Clustering* tries to place contemporaneously accessed data in the same cache-line, and *coloring* tries to avoid conflict misses by placing contemporaneously accessed data in cache-lines which do not map to the same set.

In [11] two tools are presented which programs can use for better data placement. The *ccmorph* tool is a data re-organizer which uses the topology of a linked-data-structure (LDS) to place contemporaneously accessed nodes into the same cache-lines and places those cache-lines into non-conflicting regions of the memory. *ccmorph* needs to be run periodically, and is useful for infrequently updated data-structures. It does, however, place some restrictions on the usage of the data-structure, since all pointers within it can change. Another tool, *ccmalloc*, is a memory allocator which can cluster contemporaneously accessed data. To use it, the programmer, when allocating memory, supplies a pointer to data which is likely to be accessed contemporaneously. *ccmorph* then tries to place the new data in the same cache-line as the existing data. Using these techniques, the authors achieved up to 42% speedup for the lighting simulation software RADIANCE [30].

2.3.2 Compression

Sometimes it is possible to store more data per cache-line by compressing it. For example, it is possible to store only one address in a doubly-linked list by storing the bitwise XOR of the address of the previous and next elements. The pointer to either neighbor can be computed by XOR-ing the XOR field with the address of the current node. With this scheme it is only possible to traverse the list by having the address of at least two consecutive elements.

It is also possible to compress pointers in pointer-intensive programs by allocating elements from a pool, and restricting that pool to hold a limited number of items. The pointers are then simply indices to slots in the memory pool, thus needing fewer bits.

In tree data structures, it might be possible to eliminate child pointers by restricting their placement in memory. The CSB$^+$-tree e.g., eliminates most pointers by forcing children of the same parent to be stored sequentially in memory. Each parent then only needs to store a pointer to the first child, instead of storing one pointer to each one.

2.3.3 Prefetching

Prefetching is a technique which may help reducing the time the CPU sits idle waiting for data to be fetched from main-memory. If the memory subsystem is able to process many memory requests at once, a program can request data to be moved to the cache before it is needed. Then, when the data is needed, it is already in the cache or on its way there. This reduces the perceived memory latency. Prefetching is most effective when the program is able to overlap useful computations and prefetching. To achieve this, the program needs to know in advance which data is going to be needed, and be able to perform other computations while the data is being prefetched. Prefetching can easily be applied to regular access patterns such as array iterations, but is harder to apply to less regular access, such as linked list traversal. The position of elements in a linked list is only known when the element linking to it has been fetched, making it hard to prefetch more than one element at a time.

Prefetching is usually performed explicitly by issuing so-called prefetch instructions. The CPU sends a request to the memory subsystem to move a certain cache-line to cache. Some CPUs, like the Pentium-4 [15] issue prefetch instructions when they detect that a program reads memory with regular strides, in which case they prefetch data they predict the program will soon read.

A hybrid software/hardware technique has been proposed [32], where a special purpose logic-controller is placed at each cache and memory level. It is designed to prefetch linked-lists. The logic-controller is placed in each level of the memory hierarchy. When an application needs to prefetch a linked-list it gives the micro-controller a pointer to the first item in the list, along with instructions on how to iterate over it and how many items to send up the memory hierarchy. The data dependency of linked-lists is a good example of the *pointer-chasing problem*, where it is impossible to prefetch the next node before the node pointing to it is in the cache. By placing the iteration logic in the memory controller, the addresses of the nodes do not need to be seen by the CPU before the next node is fetched.

Chapter 3

Index Structures

Indices are a very important component of database systems. They allow data to be searched efficiently and are at the core of most database operations. Historically, index structures have been designed with disk-based storage in mind, but with the growing gap between memory and CPU speeds other more cache-efficient index-structures have been proposed.

In this chapter we will describe the B$^+$-tree, which is the most popular tree-based index structure; all but one of remaining trees described in this chapter are based on the B$^+$-tree. Those are the main-memory based CSS-tree [26] and the CSB$^+$-tree [25], the main-memory based pB$^+$-tree [9] and the related disk based but cache-sensitive fpB$^+$-tree [10].

The discussion of each tree structure is followed by an analytical cost model describing its cache-performance. We open the chapter with cost model preliminaries and conclude it with a comparison based on these cost models, using parameters characteristic of a modern CPU.

3.1 Cost Model Preliminaries

We define the cache-performance of each tree as the number of CPU cycles spent waiting for data to transferred from main-memory to cache during a single query, in terms of a single cache-miss latency[1]. All trees are assumed to be completely in main-memory but not in the cache and nodes are assumed to be cache-line aligned.

There are many different parameters used in the cache-performance cost models, both tree dependent and architecture dependent. This includes parameters such as the number of records in the tree and how many keys fit in a single cache-line. We also use two equations which help simplify the analysis, as discussed below. A description of the parameters can be found in Table 3.1.

[1] Note that we do not necessarily count how many cache-misses occur, since some of the trees can overlap cache-misses resulting in reduced waiting time.

Parameter	**Definition**
N	Number of items in the index
T_1	Cache miss latency in CPU cycles
T_{next}	Latency of an additional pipelined cache miss in CPU cycles
p	Maximum number of key/pointer pairs in a cache-line
t	Maximum number of keys in a cache-line
b	Number of cache-lines per disk page
$H(b,L)$	Height of tree with branch factor b and L leafs
$D(h)$	Average node height of a full binary tree of height h

Table 3.1: Parameters and equations used in the cache-performance analysis.

For each tree, we need to know its height, based on its branch factor and number of leafs. The tree height H for a tree with branch factor b and L leafs is:

$$H(b,L) = \lceil \log_b(L) \rceil + 1 \tag{3.1}$$

We assume that binary search is used by all the trees during queries. Binary search reads memory in an unpredictable way. All reads of a binary search cause a cache-miss (assuming data is not already in cache), except for the very last ones, since they may be to the same cache-line. To compute the cache-performance of binary search we need to know how many cache-lines it accesses on average. If items are the size of a cache-line, the number of cache-lines accessed is simply the average node height of the corresponding recursion tree. Equation 3.2 gives the average node height $D(h)$ of a full binary tree, where h is the height of the tree and $2^h - 1$ is the number of nodes in the tree. If items are smaller than a cache-line, several items will be fetched from the same cache-line at the bottom of the recursion tree. By using the number of cache lines in Equation 3.2, rather than the number of items, we can obtain the try number of cache misses. We do not handle the case where items are larger than a cache-line.

$$D(h) = \frac{\sum_{i=1}^{h} 2^{i-1} \times i}{2^h - 1} = \frac{(h-1) \times 2^h + 1}{2^h - 1} \tag{3.2}$$

3.2 The B$^+$-tree

General Description

Since the main computing bottleneck used to be disk-access times, early index structures were designed with disk-access in mind. The B$^+$-tree [4, 12] has become one of the most popular indexing structure used. It is optimized for disk-access, and guarantees at least 50% space utilization. Its development was motivated by the drawbacks of an earlier tree-based index structure, the *Indexed Sequential Access Method* (ISAM), which could not adapt well to changing data distribution. The ISAM tree structure is fixed at creation

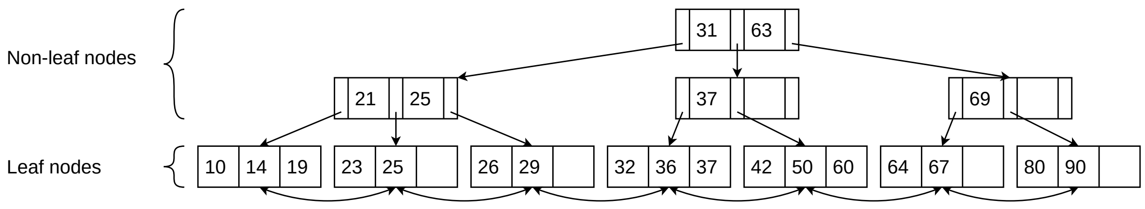


Figure 3.1: An example B$^+$-tree.

time, optimized for the initial tree data. Additional data is inserted into a linked-list of pages, connected to the appropriate leaf. Under certain conditions, the query performance degrades to the performance of a sequential search. The structure of the B$^+$-tree, however, changes with changing data, and there is a guaranteed upper bound (logarithmic) on tree height, and thus query performance.

Leafs and nodes are the size of a disk page, which allows them to be fetched from disk in a single I/O. Leafs and nodes, except for the root, are guaranteed to be at least half-full. Each node contains $n \in [t, 2t - 1]$ sorted keys, and $n + 1$ pointers. Each key is associated with two pointers, pointing at the left and right subtree of that key. All keys in the left subtree are less than or equal to the key, and all keys in the right subtree are larger. This information is used to guide queries down to the correct leaf. Leafs store a sorted array of key and tuple pairs. Figure 3.1 shows an example of a B$^+$-tree.

Tree Operations

Queries start at the root. A binary search is performed on each node to select which node to visit next. This is done until a leaf is found, where one more binary search is performed to find the tuple corresponding to the query key.

The B$^+$-tree supports efficient range scans, in both ascending and descending order, by linking each leaf with its immediate neighbors. Since all tuples are stored in leafs, all keys and their tuples can be enumerated (in sorted order) by simply reading the leafs left-to-right (or right-to-left), following the neighbor link to the next leaf when needed. Range scans are thus performed by finding the smallest (or largest) key in the tree which falls inside the query range, and reading the leafs left-to-right (or right-to-left), following the neighbor links when appropriate until a key outside the query range is encountered.

Insertion is performed by first finding the leaf where the insertion key would be, if it were a part of the tree. The key and tuple are then inserted into the leaf, possibly needing to shift data around to make space for the new data, since the key/tuple array needs to be sorted. If, however, the leaf cannot hold more keys a new leaf is allocated. Half of the keys and tuples in the first leaf are moved over to the new one. The insertion key is then inserted into one of the two leafs. In addition, a pointer to the new leaf is inserted into the leaf parent, along with the appropriate *split key*. If the parent is also full a similar split operation is performed on the parent, including allocating a new node, moving keys and pointers over to that and updating the next parent above. This is done until a non-full parent is encountered. If the root needs to be split the tree height increases by one.

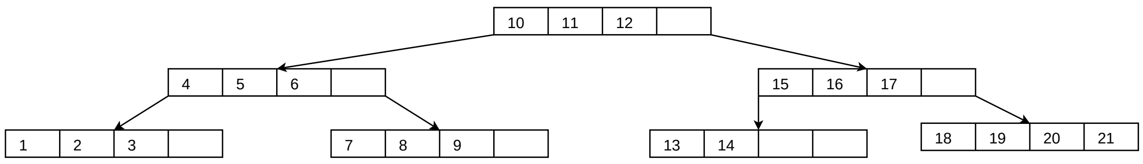


Figure 3.2: A T-tree which can hold up to 4 keys per node.

Deletion is analogous to insertion, but trickier. When a leaf underflows as the result of a deletion, one of two things needs to be one. Either items need to be moved to this leaf to another on the same level, or two leafs must be merged. The parents need to be updated accordingly, and if they underflow as a result, a similar data distribution needs to be performed there. The algorithm is rather involved but is described in [24, 17, 21].

Cache Performance

The number of cache-misses M_B^+ for a single B$^+$-tree query can be computed my multiplying the height of tree (Eq. 3.1) by the number of cache-misses per node (Eq. 3.2) as follows:

$$M_B^+ = H(bp, \lceil N/t \rceil) \times D(\lceil \log_2(b) \rceil) \tag{3.3}$$

3.3 The T-tree

General Description

The T-tree [19] was originally proposed as a better alternative to main-memory indexing than the AVL-tree [1]. Like the AVL-tree, it is a balanced binary tree, but has more than one element per node. This reduces the number of rotations during updates, since nodes are created and deleted less frequently. Keys in nodes are adjacent and stored in order. The left subtree of each node contains keys less than or equal to the smallest key in the node, and the right subtree contains keys larger than the largest key in the node. An example of a T-tree with nodes which can hold up to 4 keys can be found in Figure 3.2. The search algorithms always uses 2 keys (or 1 key when there is only 1 key per node) per node to decide whether to search either subtree or the current node.

The branch factor of the tree is only 2, which makes it much deeper than an equivalent B$^+$-tree. Only two keys per node are used to guide the search, so much data is not used, resulting in poor utilization of cache-lines. The tree was proposed when memory speed was similar to CPU speed, so reducing the number of CPU instructions was more important than optimizing memory access.

Cache Performance

When computing the number of cache-misses during a T-tree query, we assume that each node is one cache-line wide, and full. We also assume that the tree is full, i.e. all nodes have 2 children. When looking at a full T-tree, it looks very much like the recursion tree for binary search. T-tree does in fact behave very much like binary search, in the respect that the search does not always move down to leafs, and the branch factor is 2. So in order to find the number of cache-misses we simply compute the average node depth for the corresponding binary-search recursion tree using Equation 3.2. The T-tree and recursion tree structures are the same. The number of cache-misses incurred during a T-tree query is given by Equation 3.4.

$$M_{T-tree} = D(H(2, \lceil N/t \rceil)) \tag{3.4}$$

3.4 The CSS-tree

General Description

The "Cache-Sensitive Search Tree" (CSS-tree) [26] and the more recently proposed "Cache-Sensitive B$^+$-tree" (CSB$^+$-tree) [25] are main-memory indices. They are similar to the B$^+$-tree, but with nodes and leafs the size of a cache-line. The main idea behind them is that all or most pointers can be eliminated, thus making better use of each cache-line, although this comes at the expense of more expensive tree updates.

The CSS-tree has no pointers. It is stored consecutively in memory, in a breadth-first order, except for leafs which are stored after the nodes. The memory layout of the tree can be seen in Figure 3.3. Since the size of each level is known, it is possible to compute the address of any leaf or node without pointers. There are two variants, the Full CSS-tree and the Level CSS-tree.

The Full CSS-tree is an $m + 1$-ary search tree, where m is chosen such that a single node is the size of a cache-line. Each node stores m keys, but no pointers. The Level CSS-tree is similar, but instead of storing m keys per node, only $m - 1$ keys are stored. It uses slightly more memory, since the branch factor is less, resulting in deeper trees. This results in fewer key comparisons than for an equivalent Full CSS-tree, so it might be more efficient when comparisons are more expensive compared to a cache-access.

Tree Operations

Search proceeds in the same manner as in B$^+$-trees, except that child pointers need to be computed. Because of the strict memory layout, the CSS-tree cannot handle incremental updates. Updates can be batched and performed periodically, but the tree has to be rebuilt each time.

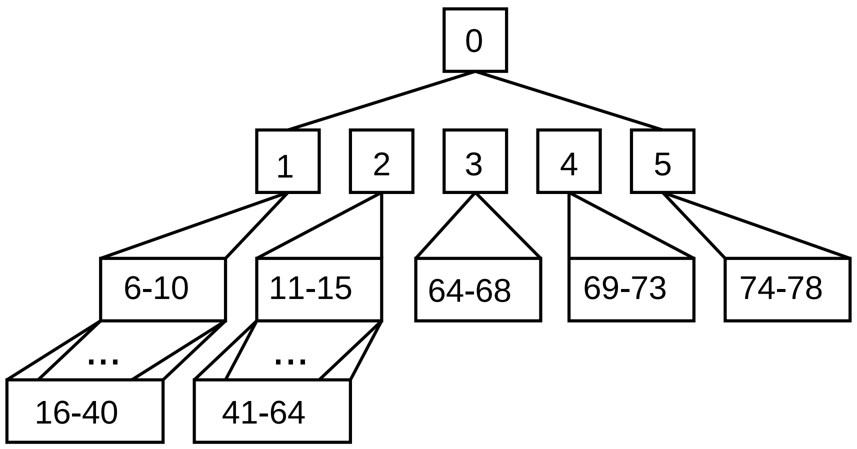


Figure 3.3: Memory layout of a Full CSS-tree.
Note that the numbers are not keys, but memory addresses. Adapted from [26].

Since the node size is known at compile-time, and the number of keys is fixed, it is possible to write a specialized binary-search which only handles that number of keys. The implementation described in [26] implements binary search by using nested if-than-else statements, thus eliminating a one add and division operation in each iteration of the binary search loop.

Cache Performance

The number of cache-misses M_{css} for a single query on a CSS-tree is equal to the tree height. Note that the branch factor of the tree is $2p$, since the the size of pointers and keys are assumed to be the same.

$$M_{css} = H(2p, \lceil N/t \rceil) \tag{3.5}$$

3.5 The CSB$^+$-tree

General Description

The CSB$^+$-tree [25] is a variant of the B$^+$-tree. The node format is the same as the CSS-tree, except that it stores one or more pointers in each node. There are two variants, the Full CSB$^+$-tree and the Segmented CSB$^+$-tree.

The Full CSB$^+$-tree stores one pointer per node. Sibling nodes and leafs (called a *node-group*) are stored contiguously in memory. Having a pointer to the node-group is enough to find the address of all leafs or nodes within it. The memory allocated for a node group is large enough to hold the maximum number of siblings. Since all siblings are stored together, half of them (on average) need to shifted during splits.

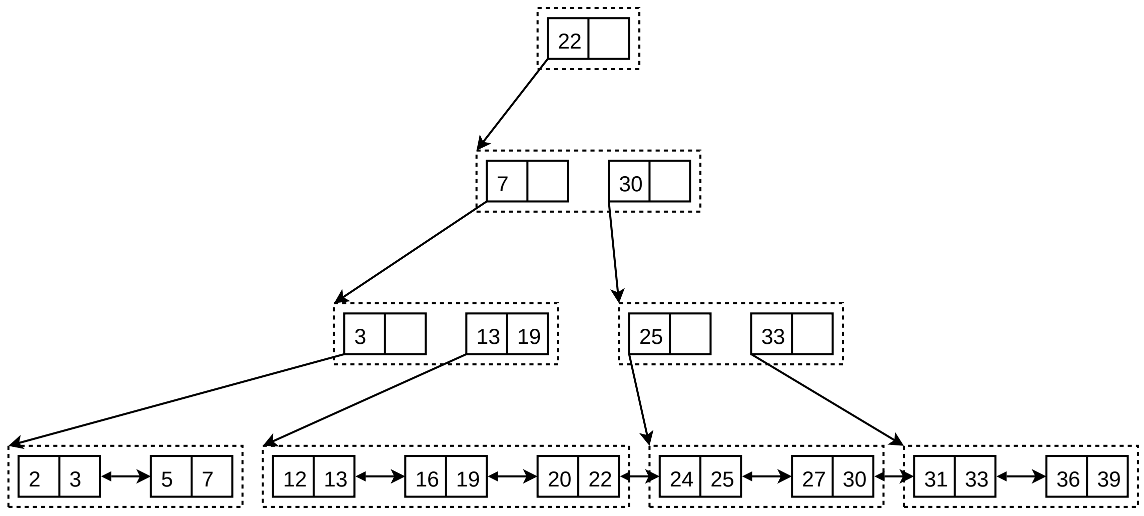


Figure 3.4: A Full CSB$^+$-tree.
Nodes within a dotted square are stored contiguously in memory. Adapted from [25].

The n Segmented CSB$^+$-tree stores $n \geq 1$ pointers per node. Node groups are split into n groups, and the pointers point to the start of each. The memory allocated for a group is large enough to hold just that group, so when a leaf is split, new memory needs to be allocated to hold the extra leaf or node, and the entire group has to be copied to the new memory. Less data has to be moved around during splits, compared to the Full CSB$^+$-tree, but there is extra overhead involved in allocating and deallocating memory. Figure 3.4 has an example of a Full CSB$^+$-tree, which is in fact equivalent to a 1-Segmented CSB$^+$-tree.

Tree Operations

Searching the tree is the same as for the B$^+$-trees. As do insert and delete operations, except that node splitting needs extra work, because of the grouping of sibling nodes.

An interesting code expansion technique for implementing the node search function of the tree search is proposed in [25]. The CSS-tree node search function only has to handle a single number of keys, whereas the CSB$^+$-tree search function has to handle multiple number of keys. A separate piece of search code is written to handle each possible number of keys. A label is placed at the start of each piece, and pointers to each label are placed in an array, which is indexed according to the number of keys that piece of code handles. This is a feature supported by the GCC compiler called *Labels-As-Values*. Implementing the search this way has the benefit of avoiding the overhead of a function call. A code snippet demonstrating this technique is shown here:

Cache Performance

The number of cache-misses in the CSB$^+$-tree is very similar to the CSS-tree, since the node format is nearly identical. The only difference is that the branch factor is slightly

```
static void* jump_table[] = { &&key_10, &&key_11, &&key_12 };

void search(void* node, size_t n_keys)
{
        goto *jump_table[n_keys];
key_10:
        // search node, assuming it has 10 keys
        goto search_end;
key_11:
        // search node, assuming it has 11 keys
        goto search_end;
key_12:
        // search node, assuming it has 12 keys

search_end:
}
```

Figure 3.5: An Example Binary Search Implementation in the CSB+-tree.

smaller. Therefore, Equation 3.5 is a good approximation of the number of cache-misses when querying a CSB$^+$-tree.

3.6 The Prefetching B$^+$-tree

General Description

The Prefetching B$^+$-tree (pB$^+$-tree) [9] is another main-memory indexing method, which is almost identical to the B$^+$-tree. It uses prefetching (see Section 2.3.3), which allows it to have nodes wider than a cache-line, without having to wait an entire cache-miss latency for each cache-line accessed. Wider nodes result in shallower trees, and the benefits of having a shallow tree usually outweigh the prefetching overhead.

Many modern CPUs have parallel memory systems. They can fetch multiple cache-lines from main-memory at the same time. Programs can instruct the CPU to fetch a given cache-line by issuing so-called prefetch instructions. Multiple prefetch instructions can be executed at the same time. For example, the Alpha 21264 CPU has a 150 cycle cache-miss latency. It can, however, fetch 15 cache-lines at the same time, meaning that a cache-line can be delivered to cache every 10 cycles. If a program knows what cache-line it will need 150 cycles later, it can prefetch it and 150 cycles later it is available in cache. This way the perceived cache-miss latency is 0 cycles, even though the actual cache-miss latency is unchanged.

Node Layout and Prefetching

The pB$^+$-tree nodes and leafs contain the same data as in the B$^+$-tree. The data layout is slightly changed: all keys are stored before tuples and pointers. The query and insert algorithms are also modified: before accessing a leaf or a node, all of its cache-lines are prefetched. Storing keys before other data enables a binary-search to proceed as early as possible. Prefetching is performed left-to-right, such that keys are delivered to cache before tuples and pointers as only keys are needed by the binary-search.

Scans

Range scans can benefit greatly from prefetching. Range scans involve finding the first/last key of a range, and scanning the leafs in one direction until a key outside the range is found. This requires minimal computation in the tree. When computation time is small compared to the cache-miss latency, it is generally better to prefetch farther ahead then when computation time is significant. However, the node width limits how far ahead the scan is able to prefetch. It cannot easily prefetch more than one leaf ahead, since the address of the next leaf is not available until the header of the current leaf is in cache. This data-dependency is an example of the *pointer-chasing problem*. To be able to get the address of further leafs faster, the pB$^+$-tree uses a *jump-pointer array*.

Leaf Pointer Arrays

There are three ways of implementing a *jump-pointer-array*, the *internal jump-pointer array*, the *flat jump-pointer array* and the *chunked jump-pointer array*. The *flat jump-pointer array* stores pointers to leafs in a flat array. Leafs can be prefetched arbitrarily far ahead since the pointers to them are readily available in the array. The drawback of this method is that updating the array can be expensive, since on average half the array needs to be shifted on a leaf split. The problem is partly remedied in the *chunked jump-pointer array*. It works in a similar fashion, except that the array is split into chunks, and the chunks form a linked list. When a leaf is split, only the chunk which contains a pointer to it needs to be updated. Pointers within each chunk are kept as evenly distributed as possible, minimizing the number of pointers that need to be shifted on a leaf split. This method, however, sets a limit on how far ahead leafs can be prefetched. The *internal jump-pointer array* uses information already in the tree. Leaf parents already contain pointers to each leaf which can be used for leaf prefetching. During a range scan, the parent of each leaf encountered is used, thus pointers to the next few leafs are immediately available. The *internal jump-pointers* cannot handle prefetching leafs arbitrary far ahead, but this approach requires less space and has no extra update cost.

Cache Performance

The cache-performance of the pB$^+$-tree is dependent on the time it takes to prefetch a leaf or a node, and the tree height. Both of these are dependent on the node width. So, in order to compute the memory stall time, we first find the node width which gives the best performance. The relevant parameters are T_1 and T_{next}. The stall time, in terms of a full cache-miss latency, for a pB$^+$-tree with nodes of optimal width w, is found be evaluating Equation 3.6 for reasonable values of w, and selecting the one which gives the smallest value for M_{pB+}:

$$M_{pB+} = \min_{w}(H(wp, \lceil N/wt \rceil) \times (T_1 + (w-1) \times T_{next}) \times (1/T_1)) \tag{3.6}$$

For each level in the tree we assume that we incur a full cache-miss latency for the first prefetch, which costs T_1 CPU cycles, and each additional prefetch costs T_{next} CPU cycles. Note that he optimal node width w may be different for different values of N. To find an optimal node width which is adequate for a range of values for N, it is possible to simply compute the optimal node width for different values of N (each of a different magnitude) and compute their average.

3.7 The Fractal Prefetching B$^+$-Tree

General Description

The Fractal Prefetching B$^+$-tree (fpB$^+$-tree) [10] is an attempt to make the disk based B$^+$-tree cache-efficient by combining it with the pB$^+$-tree. On the outside it looks like a B$^+$-tree, with nodes the size of a disk page. However, inside each page, instead of a flat array of keys, pointers and tuples, there is a pB$^+$-tree, called the *inpage-tree*. The fpB$^+$-tree is called fractal, since it is self-similar in a sense, having a tree within a tree.

There are two approaches to implementing the fpB$^+$-tree. They differ in which data is selected into which pages. The first one, called *disk-first*, starts with a traditional B$^+$-tree. It then builds the in-page trees from the data within each page. Since an in-page tree takes more space than an array of key/pointer or key/tuple pairs, each page is not completely full before building the in-page trees. This approach makes it very straightforward to modify existing B$^+$-tree implementations into fpB$^+$-trees. The other method, called cache-first, begins by building a cache-optimized tree, similar to the pB$^+$-tree. Nodes from this tree are then placed in pages, such that I/O is minimized when using the fpB$^+$-tree. When placing nodes within the pages, there are two optimization goals. The first is to group sibling leaf nodes into the same page, to minimize I/O during range scans. The second is to group a parent and its children in the same page. These goals cannot always be achieved simultaneously. The *disk-first* approach is more relevant to this thesis, and is described further.

The *disk-first* approach is a straightforward modification of a regular B$^+$-tree. The contiguous array of key/pointer and key/tuple pairs is simply replaced by a pB$^+$-tree storing the same data. The query and update procedures for the sorted array are replaced by the corresponding procedures for the pB$^+$-tree.

The fpB$^+$-tree can also prefetch leaf pages during scan operations. This is done with an external jump-pointer array, in a similar way as the pB$^+$-tree, but at a different granularity.

Cache-Performance

In order to compute the number of stall cycles for the fpB$^+$-tree, we need to know how many leafs the inpage-tree can hold, which then decides the fpB$^+$-tree branch factor. In a similar manner to the pB$^+$-tree, we compare different node widths, and select the one which gives the best overall performance.

Let N be the number of inpage-tree nodes and leafs a single fpB$^+$-tree node can hold. We can find the maximum number of leafs L it can hold, given node width w by solving for L in Equation 3.7:

$$\sum_{i=1}^{\infty} k(i, L) \leq N < \sum_{i=1}^{\infty} k(i, L+1) \tag{3.7}$$

Where $k(i, L)$ defines how many leafs or nodes level i of a tree with L leafs has, with $i = 1$ being the leaf level (which by definition holds L leafs), and $i = 2$ is the lowest node level, and so forth. $k(i, L)$ is defined as:

$$k(i, L) = \begin{cases} L & i = 1, \\ 0 & k(i-1, L) = 1, \\ \left\lceil \frac{k(i-1, L)}{wp} \right\rceil & otherwise. \end{cases}$$

In order to find the inpage-tree node width which gives best overall cache-performance we first find L for the given node width, which in turns gives us the fpB$^+$-tree height, and thus the overall cache-performance.

3.8 The Prefetching CSB$^+$-Tree

General Description

Pointer elimination and prefetching as described in this chapter are complementary techniques [9]. It is possible to increase the node width of the CSB$^+$-tree and prefetching nodes like the pB$^+$-tree does, resulting in the "Prefetching CSB$^+$-Tree" (pCSB$^+$-tree). That way, you get better performance than the pB$^+$-tree, since the branch factor is slightly higher. The cost of splitting node groups compared to the CSB$^+$-tree are, however, even more expensive since nodes are wider.

Cache Performance

The cache-performance of the pCSB$^+$-tree can be found using the same method as pB$^+$-tree and doubling the branch factor.

3.9 Index Comparison

In this section we compare the cache-performance of the indexing methods discussed in this chapter using a simulation. We assign values to the parameters of Table 3.1 which reflect the Itanium 2 processor (which is used by Chen [8]). The Itanium 2 processor has 64-bit pointers, so we use 8 byte keys, since we assume keys are the same size as pointers. The values are these:

- Cache-line size = 128 bytes
- $T_1 = 189 cycles$
- $T_{next} = 24 cycles$
- $p = 128/(8+8) = 8 bytes$
- $t = 128/(8+0) = 16 bytes$
- $b = 32(16KB)$

Figure 3.6 shows the results. The x-axis shows the number of items in the tree while y-axis shows the time spent waiting for data to be fetched from main-memory in terms of a single cache-miss latency. Note that the x-axis is logarithmic. The number of cache misses is computed using the equations derived for each indexing method described in the previous sections.

Note that the cache-performance does not necessarily reflect the actual performance of each index, since it only measures CPU cycles wasted waiting for data to be transferred from main-memory to cache. Cycles spent doing actual work are not considered, since we assume that the CPU speed is high relative to the performance of the memory subsystem.

The B$^+$-tree has the worst cache performance. The branch factor is $32 * 8 = 256$, much higher than for the other trees, so its height increases less frequently. The average number of cache-misses when querying a leaf is 11, and 12 for nodes. The cache-performance increases by that whenever the height increases.

The T-tree grows more frequently since its branch factor is only 2, making it essentially a binary search tree. Therefore, the cache-performance decreases steadily, but it still performs better than the B$^+$-tree.

The fpB$^+$-tree has better cache-performance than the B$^+$-tree; the cost of leaf and node search in terms of cache-misses is only about 25% compared to the B$^+$-tree. The branch factor is slightly smaller, since it has to store the non-leaf levels of the inpage-tree, which is why the height increases slightly earlier than the B$^+$-tree.

Like the T-tree, the CSS- and CSB$^+$-trees have nodes the size of a single cache-line. The number of cache-misses is equal to the tree height, but since the branch factor is higher than for the T-tree, it has better cache-performance.

The pB$^+$-tree cache-performance does not change as regularly as most of the other trees. This is because the optimal node width may be different for each number of items. There

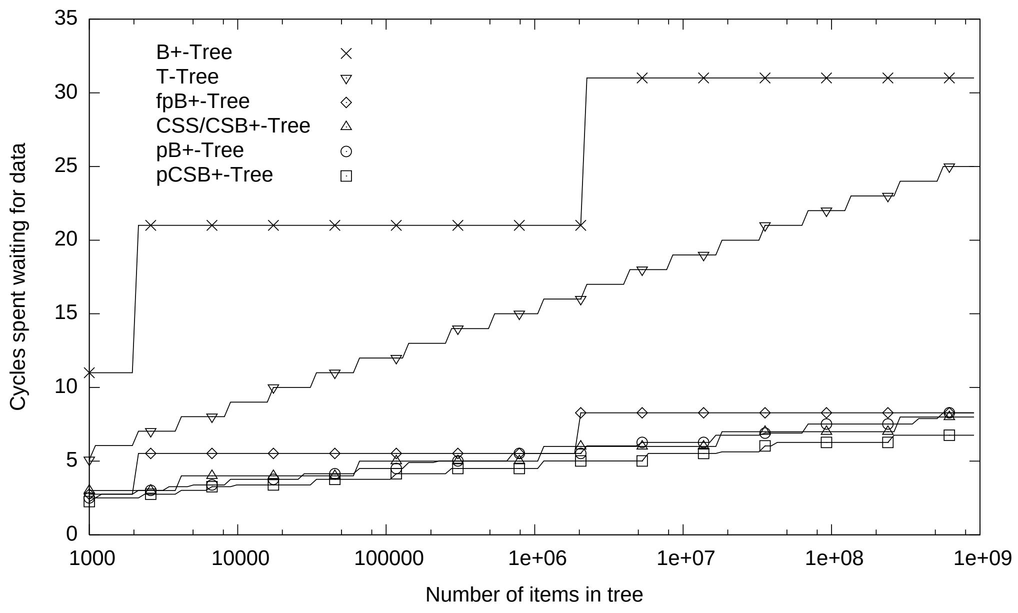


Figure 3.6: Cache-performance of different index structures for a single query. Measured in units of a full cache-miss latency.

is a compromise between tree height and cache-performance of leaf and node search, and the best combination of node width and resulting tree height is chosen. It is interested to note that the cache-performance is similar to the CSS- and CSB$^+$-trees, which cannot handle updates as efficiently.

Combining prefetching with pointer elimination results in the best cache-performance of all the trees, but by a narrow margin. In order to compute its cache-performance, we simply use the same method used for the pB$^+$-tree, but with double the branch factor, since it has no keys, and keys and pointers are of the same size.

Chapter 4

Experimental Setup

This chapter describes our experimental setup. Section 4.1 describes the hardware setup, including a discussion of the Itanium 2 processor memory subsystem. Section 4.2 describes the workloads used. Section 4.3 describes the metrics used in our experiments, and Section 4.4 describes the software setup.

4.1 Hardware Setup

The experiments are performed on a dual Itanium 2 processor server, running at 900MHz, with 4GB of main-memory. The CPUs are 2nd generation Itanium processors (code name McKinley). They run at a higher frequency than their predecessor and have lower cache latencies. The server runs Debian GNU/Linux 2.4.25-mckinley-smp.

For our experiments, the characteristics of the memory subsystem are of primary importance, so the remainder of this section gives an overview. A good reference manual for the Itanium 2 processor is available from Intel [16].

4.1.1 Cache Structure

The Itanium 2 processor has 3 levels of cache. The first-level cache is split between data and instructions, but the second- and third-level caches are unified. The caches are set-associative, with varying associativity. The details of each cache is shown in Table 4.1, including the latency of an integer load. The L1D cache uses a *write through* replacement policy, and the L2 and L3 cache use a *write back* replacement policy.

All caches use the *not-recently-used* replacement algorithm (NRU) which works as follows. Each cache-line has an associated bit which is set when it is accessed. In the case of replacement, the first cache-line with an unset bit is chosen. When all bits have been set, they are all reset, so there is always a replacement candidate.

The Itanium 2 processor has two levels of fully associative *translation lookaheadbuffers* (TLBs), with separate TLBs for instructions and data. The first level TLBs have 32 entries, and only contain translations for data in L1D and L1I. The second level TLBs have 128 entries, and contain translations for data in L2 and L3.

The miss penalty for the first level data TLB is 4 cycles. The level 2 data TLB miss penalty is 25 cycles if the data is in L2, but 31 cycles if the data is in L3. If the data is in main-memory, the penalty is 20 cycles plus the main-memory latency.

The CPU is connected to the main-memory via a double pumped 200MHz bus, which allows it to transfer data twice per bus cycle. The bus width is 128 bits, which means the bandwidth could theoretically reach 6.4GB/s.

4.1.2 Prefetching

The Itanium 2 processor instruction set includes a prefetch instruction, which has three different types of prefetching hints:

Exclusive hint: The exclusive hint controls whether the cache-line should be marked as dirty, in which case it is written back to the cache-level below when replaced. This hint should only be used when it is likely that the cache-line will be modified, e.g., when searching a leaf during tree insert.

Fault hint: The fault hint controls whether the cache-line should be fetched if the page it belongs to is not in the TLBs. The fault hint should be avoided for speculative prefetches, but should be appropriate for the index operations of the pB$^+$-tree, since the probability of using a cache-line is high.

Locality hint: The locality hint controls how high up the cache-hierarchy the cache-line is placed, as well as whether the NRU bit is set.

The *exclusive* and *fault* hints are either set or unset, but the *locality* hint has four different values, which are shown in Table 4.2. In the table, a check-mark ($\surd$) in the *Alloc* column indicates that the cache-line is brought into the corresponding cache-level, while a check-mark in the *NRU* column indicates that the NRU bit is set for that cache-line. Integer load instructions have similar locality hints, and they may set the NRU bits for cache-lines when issued, even if the cache-line was prefetched without setting the NRU bit.

Note that L1D has a different cache-line size than L2 and L3, and this has implications when prefetching with the *t1* hint. In that case the 128 byte cache-line being prefetched is brought into L2 and L3, but only the first 64 bytes are brought into L1D.

4.1.3 Data Alignment

Data alignment is very important when working with integer data on the Itanium 2 processor. All integer stores and loads must be aligned within an eight-byte window. If an unaligned integer data operation is issued, the CPU will throw an exception and call a handler inside the OS. The handler will issue an equivalent set of 1 and 2 byte operations. This means that the overhead of an unaligned load is very high compared to an aligned load. We have therefore used aligned memory allocation in our implementation, taking

Cache	Size	Line size	Integer Load Latency	Associativity
L1I	16K	64 bytes	1 cycle	4
L1D	16K	64 bytes	1 cycle	4
L2	32K	128 bytes	5 cycles	8
L3	1.5MB	128 bytes	12 cycles	12

Table 4.1: Itanium 2 processor cache size, integer load latency and associativity.

	L1D		L2		L3	
Hint	Alloc	NRU	Alloc	NRU	Alloc	NRU
t1	√	√	√	√	√	√
nt1			√	√	√	√
nt2			√		√	√
nta			√			

Table 4.2: Itanium 2 processor prefetch hints.

care to align the key- and pointer-arrays to 8-byte boundaries. Most program will not have to worry about this, since modern compilers have automatic alignment for primitive types such as integers and floating point numbers.

4.2 Workloads

Each experiment consists of two steps. First a pB$^+$-tree is constructed, either by repeated inserts or bulkloading. Second, the operation to be measured is run repeatedly; the operation may be either a point query, an insert, or a full scan of the index. We describe these steps more precisely in the following.

4.2.1 Tree construction

There are two different methods to construct trees for measurements, depending on the experiment, using insertion or bulkloading. The keys used in the tree construction can range in size from four bytes to arbitrarily long keys. In both cases, keys are generated ahead of the operation.

Keys used with repeated insertion are randomly generated. An array large enough to hold all the keys is created. The value of each byte of the key is set to the output of the C function rand() typecast to a byte. srand() is called before with a user-defined random seed.

For bulkloading, keys are generated inserted in lexicographical order. Bulkloading is performed by repeated insertions along the right-most path of the tree. The fill-factor is set by the user, to 100%, 67% or 50%. For the higher fill-factors, the node split is modified to copy less than half of the nodes to the new sibling node.

Note that pointers are always assumed to to be 8 bytes, while no data is stored in leafs, only keys.

4.2.2 Index operations

Point queries always request a record that is in the index. Query keys are chosen by inserting all keys in the tree into an array and shuffling the array. The keys are then used in the shuffled order; of more queries are issued than there are keys in the tree, the array is used multiple times.

Insert operations are performed exactly as described above for tree construction. Essentially, they are implemented with a point query, followed by an insert to the leaf.

Scans are always full index scans. They descend the index along the left-most path, and then traverse the leaf level, using the jump-pointer array for prefetching.

4.3 Performance Metrics

There are two performance metrics used in our experiments. The primary metric is the running time, measured using the *getrusage* system call.

For a more detailed analysis of the running time, we use *pfmon* (version 2) [1] and construct detailed stall cycle breakdown. Furthermore, sometimes, we have used the HPC-Toolkit for a detailed correlation of the CPU events with the source code for better understanding of the performance results.

4.3.1 Running Time

The simplest metric is the time it takes for an operation to complete. To measure this we use the *getrusage* system call. Unlike measuring the wall clock time time, it reports the user time allocated to the process. It is implemented by counting the number of timeslices the OS allocates to the process. If there are many context switches per time unit on the system, the measurement can become skewed. We address this in our experiments by making sure that there are no heavy processes running at the same time, and running each experiment multiple times. Additionally, weshow that having no other heavy processes running results in accurate measurements, by comparing the results of *getrusage* to those of *pfmon*. We also make sure that no I/O operations are performed by making sure that all data is stored in main-memory, because I/O time might not be measured since the OS could overlap performing the I/O and running other processes.

[1] Available at http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4.

4.3.2 Stall Cycle Breakdown

The Itanium 2 processor has 4 performance counters, each of which can be associated with one of over 200 events in the CPU. These can be used to group CPU cycles into different stall categories, depending on what caused the stall. Jarp [18] has suggested a methodology for this grouping, which we have adapted. The stall categories, the associated performance counters, and a description of the stall cause can be found in Table 4.3, which is borrowed from [23].

To sample the values of the performance counters we use *pfmon*. It collects the values of performance counters while running a program, and outputs their values when the measured program terminates. We run it with the parameter *-u*, which causes the CPU to only increment the counters when in user space.

One important feature of *pfmon* is that it can start profiling the first time a certain function is called. The experimental software uses this functionality by calling an empty function before an operation starts, to avoid measuring program startup time and tree construction. In order to avoid measuring the overhead of cleaning up data structures the program is forcefully terminated when the operation finishes.

It is possible to use *pfmon* instead of *getrusage* to measure the time it takes for an operation to finish, as the clock frequency of the CPU is found in the device file */proc/cpuinfo* and can be used to convert the number of cycles into seconds. In order to compare the running times given by *getrusage* and *pfmon*, as well as to verify that the stall cycle breakdown is a good representation of program activity we ran the following experiment. We bulkloaded 1M keys of varying size into a pB$^+$-tree and queried it 1M and 10M times. Both leafs and nodes were 5 cache-lines wide. The results, which are shown in Table 4.4, shows that as the error does not grow with increased running time or key size (and is never more than $\pm 1.01\%$) the two methods of measuring running time are comparable.

4.3.3 Source-Code/CPU Event correlation

The *HPC-toolkit*[2] is a set of tools used to analyze the performance of programs. It includes tools to correlate CPU events with source code using symbol table information. The Itanium 2 processor contains a *Data Event Address Register*, which contains the instruction pointer when the last memory related event occurred. When this register is sampled at regular intervals it can be used to approximate the distribution of data events across the source code, which is useful for understanding how much time is spent in different parts of the program and where cache-misses occur.

4.4 Software

The experimental software we use consists of an implementation of the pB$^+$-tree, written in C, and a set of scripts used to automate the process of running the experiments and gen-

[2] Available at http://hipersoft.cs.rice.edu/hpctoolkit.

Category: description	**Formula**
D-cache stalls: Stalls incurred waiting for data to be delivered to cache.	BE_EXE_BUBBLE_GRALL - BE_EXE_BUBBLE_GRGR + BE_L1D_FPU_BUBBLE_L1D
Instruction miss stalls: Stalls incurred waiting for instructions to be delivered to cache.	FE_BUBBLE_IMISS*
Branch misprediction: Stalls incurred when the pipeline is flushed because of a branch misprediction.	BE_FLUSH_BUBBLEḂRU + FE_BUBBLE.BUBBLE* + FE_BUBBLE.BRANCH*
RSE stalls: There are 96 registers available for stack variables, but if too many are used, a backing store simulates more registers. These stall occur when there are no free stack registers, and data is moved back and forth between the CPU and the backing store.	BE_RSE_BUBBLE.ALL
FPU stalls: Stalls in the floating point micro-pipeline.	BE_EXE_BUBBLE.FRALL + BE_L1D_FPU_BUBBLE.FPU
GR Scoreboarding: Integer register dependency stalls.	BE_EXE_BUBBLE_GRGR
Front-end Flushes: Cycles lost due to flushes in the front-end.	FE_BUBBLE_FEFLUSH*
Busy: Unstalled CPU cycles.	CPU_CYCLES minus the sum of the above

Table 4.3: A description of CPU stall cycle categories and their related events.
* Counters starting with FE_ must be scaled with the reduction factor $\frac{BACK\_END\_BUBBLE\_FE}{FE\_BUBBLE\_ALLBUT\_IBFULL}$.

Key Size	Queries	pfmon	getrusage	Difference
8B	1M	2.3852	2.3999	-0.61%
16B	1M	3.0847	3.0952	-0.34%
32B	1M	3.2904	3.2690	0.65%
64B	1M	4.2625	4.2832	-0.48%
128B	1M	6.6641	6.7017	-0.56%
8B	10M	23.8571	24.1001	-1.01%
16B	10M	30.8538	30.9927	-0.45%
32B	10M	32.3458	32.3262	0.06%
64B	10M	42.7491	42.9741	-0.52%
128B	10M	66.8485	66.9771	-0.19%

Table 4.4: Difference in measuring runtime with *getrusage* and *pfmon*.

erating graphs of the results. We refer to the set of scripts as the experimental framework. Section 4.4.1 describes the compiler and compiler flags used. Section 4.4.2 describes the pB$^+$-tree implementation and Section 4.4.3 describes the experimental framework.

4.4.1 Compiler

We use version 8.0.066 of Intel's C Compiler for Linux, which has very good support for the Itanium 2 processor. We used the following compiler flags:

`-O2` Enables non-aggressive optimizations. Recommended for most applications.

`-Ob2` Allows the compiler to control function inlining. Inlined functions are copied to the place where they are called, thus saving the overhead of a function call.

`-fno-alias` Tells the compiler to assume no aliasing in the program. Aliasing is when two pointers point to the same memory, and changing one might potentially affect data in the other, so the compiler must be careful not to reorder stores and loads in expressions containing pointers.

`-ip` Enables single-file inter-procedural optimizations.

More aggressive optimizations were tested, but resulted in a performance loss.

4.4.2 pB$^+$-Tree implementation

We have implemented the pB$^+$-tree in the C programming language. The code base is split into two parts, a library which implements the pB$^+$-tree and a command line program which sets up performs experiments on the pB$^+$-tree.

Compile Time Parameters

Many tree parameters must be defined at compile-time using macros. A small script builds binaries for all configurations of these parameters. The binary names include what parameter configuration they were built with. The parameter values are separated with underscores (_) and the relative ordering is always the same. Table 4.5 describes these parameters and the set of possible values. Parameters are listed in the same order as they would be in the binaries.

Neither the **Unrolling** nor the **Prefetch policy** parameters are applicable if prefetching is not used. The binaries built are: non-prefetching using binary and sequential search, and prefetching binaries, using all combinations of remaining parameters.

For each binary, there is a corresponding debug binary. The debug binaries are compiled without optimizations, and use many internal tests to verify the consistency of the tree during tree operations. They also include many unit tests which are used to black-box test the tree. As the name suggests, they are only useful for debugging and are much slower than the non-debug binaries.

Parameter	**Values**	**Meaning**
Prefetching	pre	Use prefetching
	<none>	Do not use prefetching
Search	bin	Use binary search
	lin	Use sequential search
Unrolling	duff	Use unrolled prefetch loop
	<none>	Use regular prefetch loop
Prefetch policy	all	Use *Linear* prefetch policy
	half	Use *Binary* prefetch policy
	half2	Use *Half-Linear* prefetch policy
	pat	Use *Half-Binary* prefetch policy

Table 4.5: Compile time parameters.

Node and Leaf Allocation

Two methods are used to allocate leafs and nodes. The first methods is to have a pool of memory and allocate memory from that. If the pool fills up, its size is doubled. Using this method, node and leaf pointers are integers, representing the index of their slot in the memory pool. The pointers can be 16, 32 or 64-bit. The second method is to let the user specify an allocation function, which could simply be malloc/free or a cache-line aligned version. Using this method, node and leaf pointers are 64-bit pointers. Whether or not to use a memory pool, and which memory pool pointer size to use, are compile-time parameters.

Using the memory pool might be useful if the tree were to be used as the inpage-tree in an fpB$^+$-tree implementation. In that case, the memory available to the tree is limited to an area inside a disk-page sized node and all pointers need to be relative.

Internally, the main difference between using the memory pool or a user-defined allocation function, is that memory-pool pointers need to be translated to physical addresses each time a leaf or a node is used, which incurs some overhead. The memory pool allocation method is not used in any experiments in this work. Instead we use the system function *memalign* for all experiments. *memalign* returns aligned memory. Allocated memory is 128 byte aligned, which is the cache-line size of the L2 and L3 caches.

Leaf/Node Structure

Each leaf consist of a 40 byte header, an array of keys, and an array of tuples, in that order. The header contains 4 fields. The number of keys, pointers to its immediate neighbors, and a union of two structs. The two structs contain information about the leaf's position inside a *flat* or a *chunked leaf pointer array*, if one of them is used for tree scans.

Each node consists of a header, an array of keys, and an array of pointers, in that order. The header contains 4 fields. The number of keys, the level the node is at, and pointers to its immediate neighbors. Of these, only the key field is strictly necessary. The other fields are used to validate the tree structure, and the right-neighbor pointer is also used by the

```
<top-level directory>
  |--code/
  |    |--Makefile
  |    |--compile.sh   <compiles different configurations of the pB+Tree>
  |    |--bin/         <compiled binaries>
  |    |--libpbtree/   <pb+tree code>
  |    |--pbtree/      <experiment code>
  |    |--temp/        <temporary compilation files>
  |
  |--experiments/
       |--sfm_main.pl             <performs a single experiment>
       |--run_opt_node_width.pl   <finds optimal node width>
       |--run_experiment.pl       <runs all experiments in a given directory>
       |--run_report.pl           <generates a report of experiment results>
       |--output/                 <experiment results>
```

Figure 4.1: Framework directory structure.

internal leaf-pointer array. The header is 24 bytes if the pointers are 64-bit, 16 bytes if they are 32-bit, and 12 bytes if they are 16-bit.

The key, tuple and pointer arrays in the leafs and nodes are 8-byte aligned, relative to the start of the node, since the overhead of performing non-aligned integer operations on the Itanium 2 processor carries heavy performance penalties as discussed in Section 4.1.3.

4.4.3 Experimental Framework

We use a small experimental framework to run experiments, gather relevant performance metrics and report results. This section gives an overview of how it works and how to use it.

The framework consists of PERL scripts which communicate with various software. The framework depends on 4 software packages for complete functionality. They are Gnuplot 4.1[3], PdfLatex[4], pfmon[5] and PERL[6]. The directory structure used is shown in Figure 4.1, with descriptions of the contents of each directory. It it split into 2 main parts. The *code* part contains all the pB$^+$-tree software and build tools while the *experiments* part contains everything else including the experiment definition scripts.

We define an experiment as a set of workloads whose results are plotted into a single graph. Once an experiment has been defined it is simple to run it and create a report with graphs. Assume the directory `exp_1` contains a file describing an experiment. The following commands are used to perform the experiment and create a report, assuming that the user starts in the top-level directory:

[3] Available at http://www.gnuplot.info/
[4] Available at http://www.tug.org/applications/pdftex/
[5] Available at http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4.
[6] Available at http://www.perl.com/

```
% cd experiments
% perl run_experiment exp_1/
% perl run_report exp_1_results/ exp1_report "Exp1 results"
```

The second step runs the experiment and places all results in the folder `exp_1_results`. The third step creates two files, `exp1_report.pdf` and `exp1_report.tar.bz2`. The first file contains the graphs showing the experimental results, and the second one contains the data files created during the experiment and all Gnuplot and Latex scripts used to make the graphs.

The framework can be downloaded at `http://datalab.ru.is/csi`. It includes the pB$^+$-tree implementation, the experiment framework, all experiment definitions used to produce the results in this work, and documentation describing the syntax of the experiment definition files.

Chapter 5

Implementation Choices

We implemented the pB$^+$-tree based on the description in [9] and adapted it to the Itanium 2 processor. During this work, we realized that we had many implementation options, which might affect performance significantly. Rather than choosing arbitrarily, we have implemented those options; this section analyzes the performance trade-offs. The options we have identified are shown in Table 5.1. As the table shows the options concern the prefetching hints, the implementation of the prefetching loop, and the implementation of the search algorithm. The table furthermore summarizes the results of our experiments; in the remainder of this section, we study each category in turn and present our performance results.

5.1 Prefetching Hints

In [8], it is argued that the faulting hint should always be used (set to "yes"). If it is not used and the cache-line address is not found in the TLB, then all prefetching instructions are canceled, and the first access incurs a TLB miss latency (see Section 4.1.1), and all cache-lines accessed incur a full cache-miss latency. It does appear to that it is beneficial to use this hint as 1) the access pattern is very predictable, and 2) the associated cost of a miss is high.

Category	**Options**	**Choice**
Fault hint	Yes / No	Yes
Locality hint	t1 / nt1 / nt2 / nta	Query: t1 Insert: nta Scan: t1
Prefetch loop	Regular / Unrolled	Unrolled
Node prefetching	Linear / Half-Linear / Binary / Half-Binary	Search: Linear Scan: Half-Lin.
Search algorithm	Linear / Binary	Binary
Comparison	Memcmp / Optimized	Optimized

Table 5.1: Implementation options and choices.

Key/Node size	Fault	Query Time	Speedup	Scan Time	Speedup
8B/4CL		2.259s		0.596s	
	√	2.270s	0.46%	0.596s	0.02%
64B/32CL		7.676s		0.833s	
	√	7.679s	0.04%	0.823s	-1.21%

Table 5.2: Prefetch fault hint speedup. 1M records, 1M queries, 10 scans.

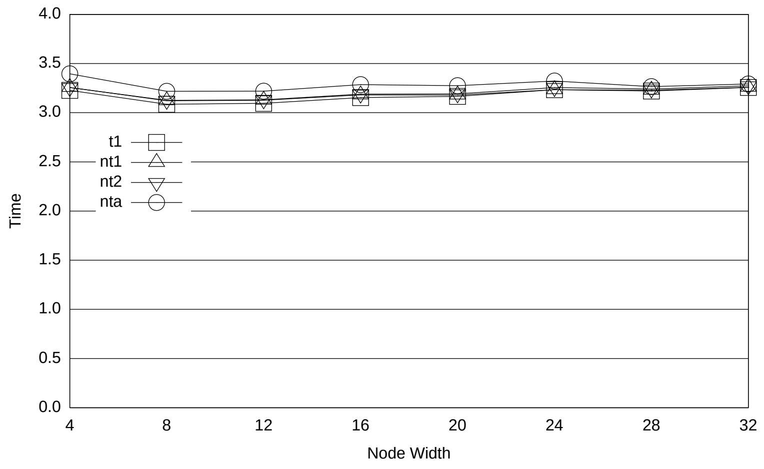


Figure 5.1: Comparison of locality hints for queries.

We set up the following experiment: We used a small pB$^+$-tree (1M keys; 8B keys; 4CL (cache-line) nodes) and a large pB$^+$-trees (1M records; 64B keys; 32CL nodes). We ran 1M queries and 10 scans over each tree, comparing the running time. Results are shown in Table 5.2. While the expectation was to see better performance with the faulting hint, particularly in the case of the scan of the large tree, the results showed no significant difference with or without the hint. As the logic for using the hint is still sound, however, we suggest to use it.

Turning to the location hints, the expectation was that each operation would have its preferred hint. We expected the *nta* hint to perform best to prefetch new nodes during splits, but worst for the other operations. The reason is that the nodes will not be read, so they done not need to be in the L1D (remember that the L1D is write-through), and it does not need to stay in cache long, so it should be biased for replacement.

The performance trade-offs of the other hints were less clear. For all these experiments we tested three key sizes, 8B, 16B and 32B, on 1M key trees, bulkloaded with fill factors 100%, 67% and 50%. Node widths used were 4CL through 32CL, using 4CL strides.

Figure 5.1 shows the response time for queries for 16B keys, as the node size is varied. As the figure shows, the *t1* hint performs best with this workload, showing savings of up to 2.6% compared to the *nt1* and *nt2* hints, and up to 7% compared to the *nta* hint. Similar savings were seen for other fill factors; across the key size/node size combinations we tested, the *t1* hint clearly performed the best.

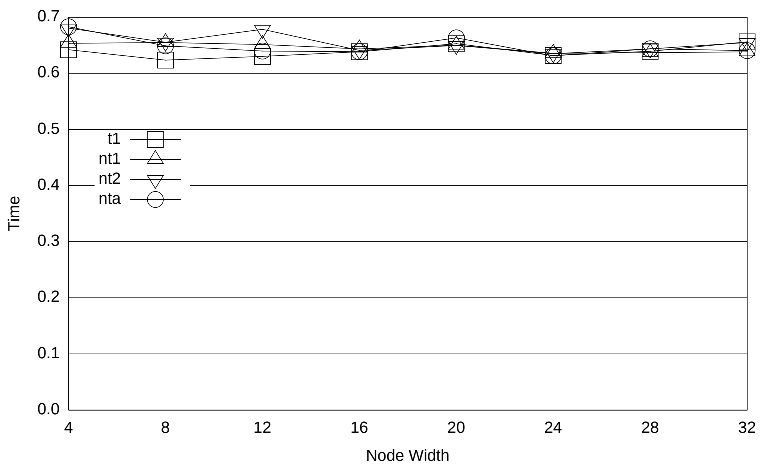


Figure 5.2: Comparison of locality hints for scan.

The insertion operation has two phases. First, a point query is used to find the correct location for the tuple. Based on the previous results, we chose to use the *t1* hint for this point query. Then the data is inserted, but since the leaf is already in cache, no prefetching is needed. In the case of node splits, however, reading new memory is necessary. For this operation, we expect the *nta* hint to be beneficial.

Turning to the performance of insertions (not shown), we did not observe any significant differences between the prefetching hints. In our experiments we did observe a few cases where the *nta* hint does indeed perform best, but only for (unrealistically) large keys and then not by a large margin. Clearly, the node splits are such a small fraction of the workload that their performance impact is negligible.

Finally, we turn to the scan operation. Figure 5.2 shows the performance of full index scans for an index with 1M keys of 16 bytes each, as the node size is varied from 4 to 32 cache-lines. For this operation, as it did for the search, the *t1* hint gives the best performance on average, but by a very small margin. The *nt1* and *nt2* hints perform about the same, and *nta* is slightly worse.

5.2 The Prefetching Loop

We now turn to the implementation of the prefetching loop itself. While the prefetching instructions do run in parallel to other processing, issuing the prefetches results in some overhead and must be efficiently implemented.

We compare a straight-forward implementation of the loop with a method called Duff's device[1] which allows unrolling loops which can handle an arbitrary number of iterations. The pseudo-code for this unrolling is shown in Figure 5.3, where the `pf()` macro performs the actual prefetching. The loop is based on a trick allowed by C syntax, whereby

[1] See http://en.wikipedia.org/wiki/Duff's_device.

```
ptr       = first_cache_line
numloops  = numcachelines / 8;
remainder = numcachelines % 8;
switch (remainder)
{
  case 0:  do {  pf(ptr); ptr += cachelinelength;
  case 7:        pf(ptr); ptr += cachelinelength;
  case 6:        pf(ptr); ptr += cachelinelength;
  case 5:        pf(ptr); ptr += cachelinelength;
  case 4:        pf(ptr); ptr += cachelinelength;
  case 3:        pf(ptr); ptr += cachelinelength;
  case 2:        pf(ptr); ptr += cachelinelength;
  case 1:        pf(ptr); ptr += cachelinelength;
           } while (--numloops > 0);
}
```

Figure 5.3: Pseudo-code for Duff's device.

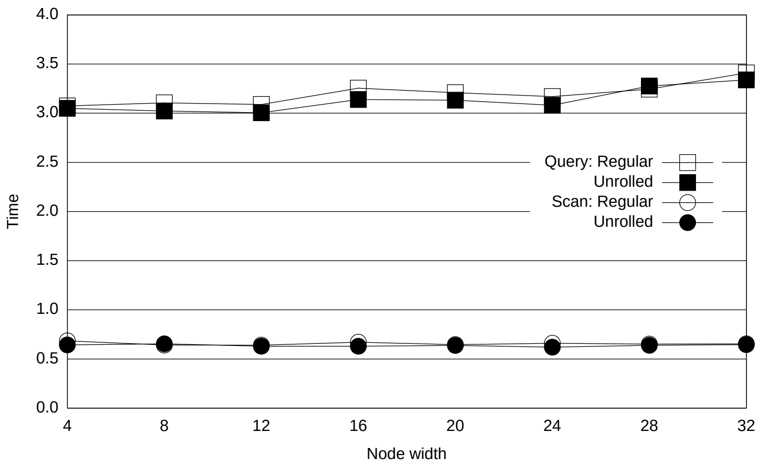


Figure 5.4: Comparison of loop unrolling options.

the switch sentence jumps into the middle of a loop of eight unrolled prefetch operations. The destination of the jump is based on the remainder of a division by 8 into the number of cache-lines to prefetch. Note that there is a precondition to the loop, that *numcachelines* must be > 0. Once the initial jump has been made, the do-loop takes over (the case labels are then ignored) and runs until every cache-line has been prefetched.

Figure 5.4 shows the performance of the regular and unrolled loops for 1M queries and 10 scans (it has very little effect on the inserts), over trees with 16B keys. As the figure shows, the unrolled loop performs better in all cases, in some cases running 4–6% faster than the original loop. The speedup for queries is 2–3% on average and up to 4% in some cases, but 2–3% on average for scans and up to 6% on some cases. We therefore recommend using the Duff's device in all cases.

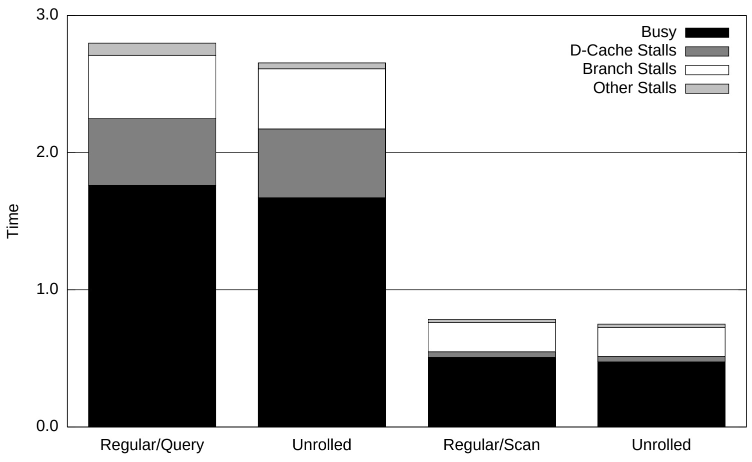


Figure 5.5: Stall cycle analysis for loop unrolling.

Figure 5.5 shows a more detailed breakdown of the costs of the queries and scans, for nodes of 16 cache-lines. The figure shows the primary benefit comes from a reduction in busy time, presumably from less loop overhead. Finally, various other stalls are reduced, for some unidentified reason. For the scans only the busy time is observably reduced.

5.3 Node Prefetching

Turning to the implementation options for node prefetching, Chen at al.[9] suggest prefetching all cache-lines of a node, left-to-right. But since a node does not necessarily use all of its cache-lines it may be more efficient to prefetch only those cache-lines which are in use. Additionally, when binary search is used, prefetching the cache-lines left-to-right might not be the most appropriate order in which to prefetch, since the reading pattern of binary-search is not sequential.

We have implemented four prefetching policies, the suggested method and three selective prefetching policies, which have in common that they prefetch only cache-lines containing data. In the following, all four policies are described, along with their benefits and drawbacks. The discussion is in terms of nodes, but it applies equally well to leafs.

Linear: The Linear policy prefetching prefetches all cache-lines of a node, left-to-right. This is the fastest way to prefetch all lines in a node. This policy, however, is oblivious to the order in which cache-lines are accessed and may prefetch cache-lines that contain no data.

Binary: This policy first reads the node header to learn how many keys the node has. It then prefetches the key array in the order dictated by a breadth-first traversal of the binary-search recursion tree for that number of keys. This provides the optimal prefetch order for binary-search. Finally, the policy prefetches the pointer array in the same order.

This policy has two performance overheads. First, it suffers a mandatory cache-miss for reading the node header before prefetching starts. This is avoided with linear prefetching. Second, since cache-lines are no longer prefetched left-to-right, Binary prefetching needs to know which cache-line to prefetch next. To save the calculations, this information is pre-calculated for all possible node contents and stored in a look-up-table, but there is still an extra indirection involved for each prefetch.

Half-Binary: This is similar to the Binary policy above. This policy first prefetches the node header, first half of key array and first half of pointer array, using a prefetch order found in a lookup-table, but without any consideration for the actual node size. All these prefetches are required, since the pB$^+$-tree is guaranteed to have at least 50% occupancy of all nodes. It then reads the header and subsequently prefetches all remaining required cache-lines in the same order as the Binary policy, excluding cache-lines which have already been prefetched.

Half-Linear: This policy prefetches the header and the first half of the key array, regardless of how many keys the node has. It then reads the header to learn how many keys are in the node. Finally it prefetches any remaining cache-lines containing keys and all cache-lines containing pointers. Cache-lines are prefetched left-to-right.

Note that since both Half-Binary and Half-Linear policies require reading the node header, they cannot issue the second batch of prefetching statements until the node header is in the cache. Furthermore, note again that all the ordering data needed for these operations is precomputed and stored in look-up-tables.

Turning to the performance of these proposed policies, we observed that the Binary policies were never significantly better than the Linear policies. There appear to be two reasons. First, the Binary policies are only applicable for queries, and not for scans. As we will discuss in a moment, more performance improvements are possible during scans. The second reason appears to be that prefetching is so fast that the overhead of the indirection is higher than the savings from the improved order. Therefore, we do not study the Binary policies further.

Turning to the Linear policies, we observed that no performance gains were seen for queries. This is understandable, since only a single node can be prefetched at a time, as prefetching can not start until the appropriate child is found. Since the Linear policy can very efficiently prefetch a full node, there is little to be gained.

For scans, on the other hand, it is possible to prefetch more than one node using the jump-pointer array, so in this case there are more potential gains from prefetching only the required data from each node. We ran the following experiment: We used a a pB$^+$-tree with 1M keys, bulkloaded using fill factors 100%, 67% and 50%, key sizes 8B, 16, 32B and 64. We scanned these trees 10 times, using leaf prefetch distances 2 and 4. We found that the Half-Linear prefetch policy only performed when the key size was 64B, and the tree was bulkloaded to 50% or 67%. Figure 5.6 shows the results for trees bulkloaded to 100% using 16 byte keys. The Linear prefetch policy clearly performs better. Figure 5.7

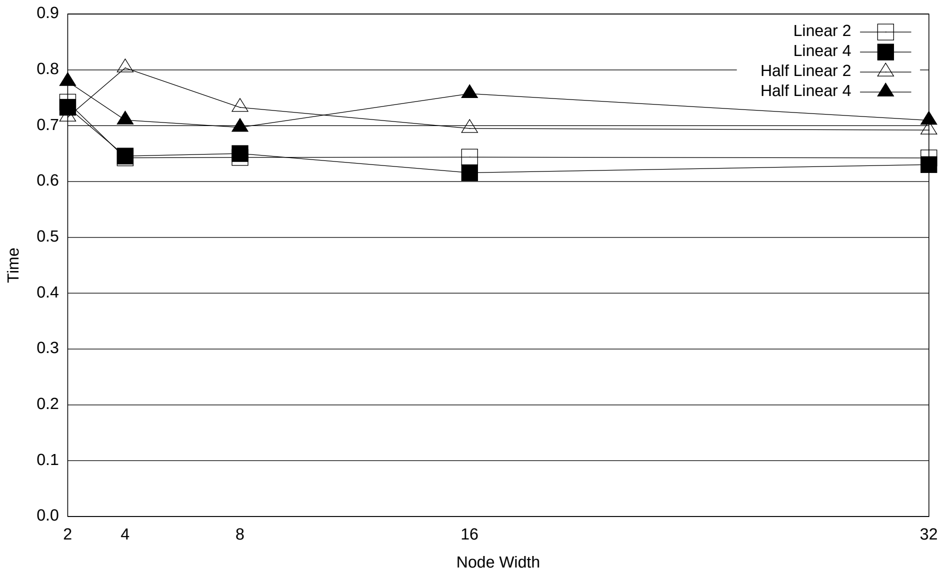


Figure 5.6: Performance of Linear prefetch policies.
1M × 16 keys, 10 Scans, 100% fill factor.

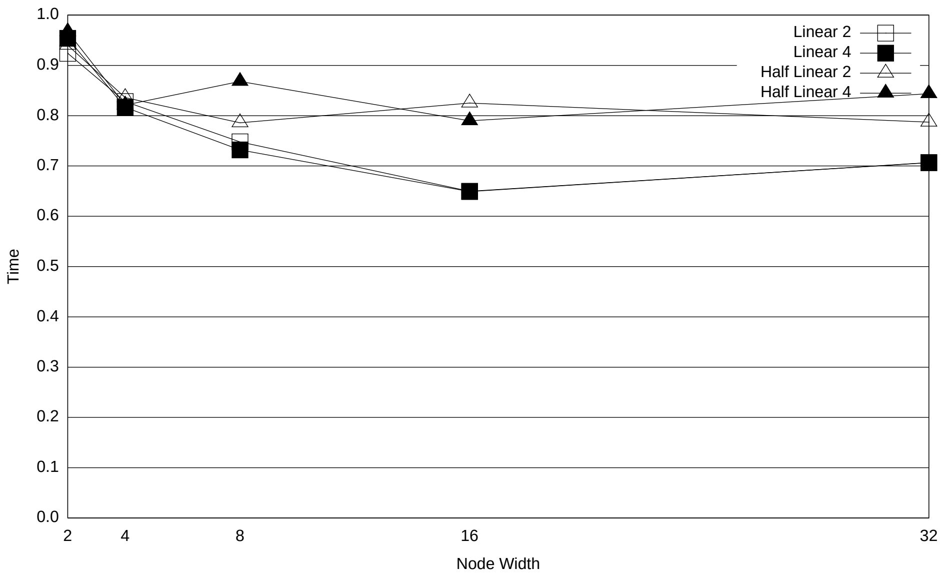


Figure 5.7: Performance of Linear prefetch policies.
1M × 32B keys, 10 Scans, 67% fill factor.

shows similar results but with a tree bulkloaded to 67% using 32 byte keys. Figure 5.8 shows the results for a tree bulkloaded to 50% using 64 byte keys, where the Half-Liner prefetch policy performs better. These results indicate that unless the potential bandwidth savings of prefetching only half the data is big, the Half-Linear policy is slower than the simpler Linear policy.

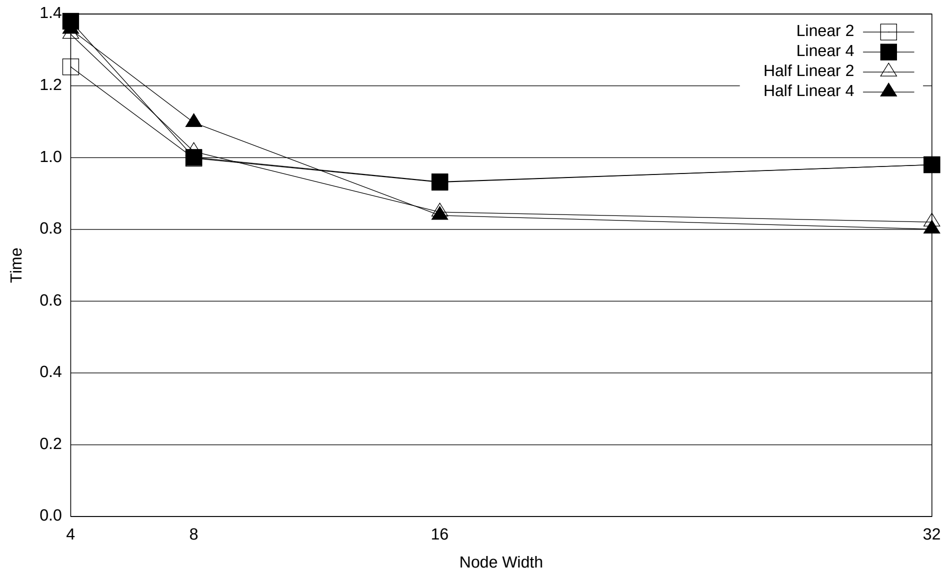


Figure 5.8: Performance of Linear prefetch policies.
1M $\times$ 64B keys, 10 Scans, 50% fill factor.

5.4 Search Algorithm and Comparison

B$^+$-trees have traditionally used binary search within the nodes and, according to [8], pB$^+$-trees have inherited that behavior. In [27], however, linear search is (surprisingly) shown to be preferable for the pCSB$^+$-tree. We therefore studied these two alternatives for our implementation.

Furthermore, especially for keys larger than eight bytes, there are alternatives for running the key comparison. One obvious option is the memcmp() function, but this operation is based on byte-wise comparison and therefore likely to be inefficient. We have therefore implemented a comparison routine that works for keys which are multiples of eight bytes. This routine always compares eight bytes at a time, by type-casting to unsigned long integers, and uses Duff's device to unroll the comparisons.

Figure 5.9 shows the performance of the linear and binary search, when used with memcmp and the optimized comparison operator. The figure shows that 1) the optimized comparison operator performs significantly better than the memcmp operator, and 2) the binary search performs much better than the linear search, especially for large nodes. In fact, we have run many other experiments comparing linear and binary search, and have never observed linear search to outperform binary search.

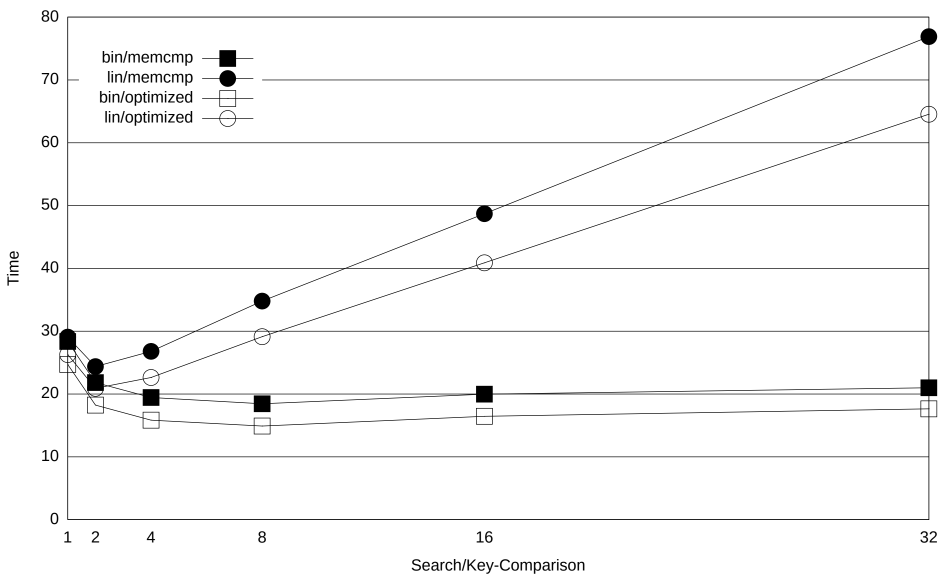


Figure 5.9: Performance of search with different search and key comparison methods. 1M $\times$ 16B keys; 1M queries; 100% fill factor.

Chapter 6

Prefetching Performance

The key idea behind the pB$^+$-tree is to use prefetching to have nodes wider than a cache-line, without incurring a full cache-latency for each cache-line accessed. Chapter 5 showed how different implementation choices affect the performance of the pB$^+$-tree; this chapter shows how the prefetching affects the overall performance.

We measure query and scan performance, but not insert performance. The reason is that most of the work of the insert operation is performed during the initial point query. Additional prefetching during inserts is only performed during leaf and node splits, which occur very rarely. We therefore feel that query performance is characteristic of insert performance.

6.1 Query Performance

To measure the effects prefetching has on query performance we measure the query performance of trees with 1M keys, constructed using repeated insertion or bulkloading using fill factors 50%, 67% and 100%, and key sizes 8B, 16B and 32B. We measure both query performance with and without prefetching using binary search, and both the *memcmp* and *optimized* key comparisons. We use the *t1* prefetching hint as recommended by Chapter 5, and the *Linear* prefetching policy.

Figure 6.1 shows the performance of 1M queries to a tree of 1M $\times$ 16B keys created using repeated insertion, using the *optimized* key comparison and binary search. Prefetching gives no benefit for node width 1CL, as expected. The benefit is at its most for node widths 4CL, 6CL, 8CL and 12CL. After that the benefit decreases somewhat. This trend of having decreased speedup after this interval of maximum speedup can be seen in all the experiments.

Table 6.1 and Table 6.2 show the average speedups gained from prefetching over all these experiments. The column *6-8-12* shows the average speedup for node-widths 6CL, 8CL and 12CL. The column *all* shows the average speedup across all tested node widths. The difference between the two columns, across all the experiments, show the same trend as noted before; increasing the node width beyond a certain point results in decreasing

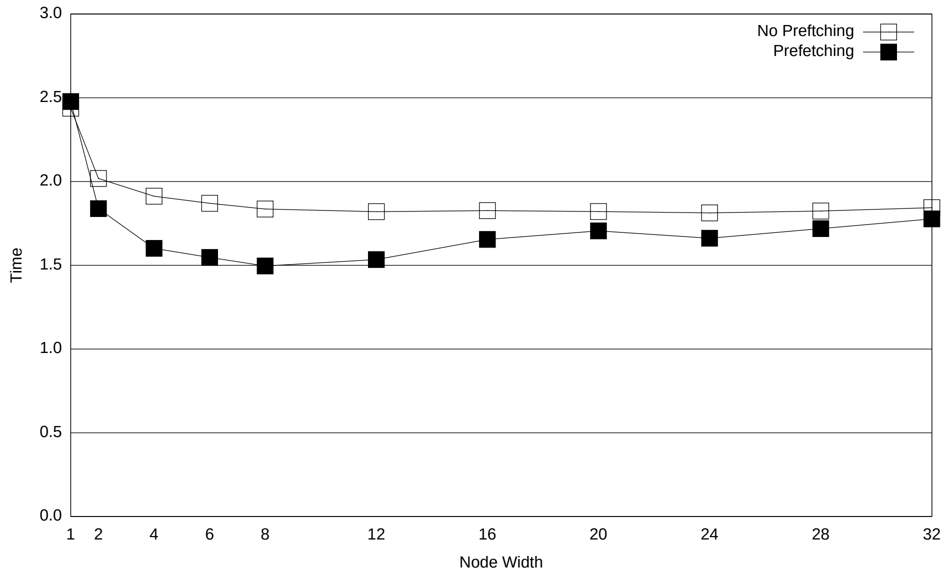


Figure 6.1: Performance gains of using prefetching.
1M × 16B keys, repeated insertion, optimized key comparison, binary search.

Key size (bytes)	8		16		32	
Node Width (cache-lines)	6-8-12	All	6-8-12	All	6-8-12	All
Insert	13.0%	7.5%	16.7%	10.9%	21.7%	13.6%
100%	10.0%	5.4%	10.7%	7.2%	11.7%	9.0%
67%	9.0%	5.5%	10.3%	7.3%	11.7%	9.7%
50%	7.3%	5.9%	9.3%	7.9%	13.0%	9.2%

Table 6.1: Prefetching speedup for different trees using memcmp key comparison. Averaged across node widths.

speedups. This is normal since at that point the tree height is only 2, and increasing the node width will only result in greater node search time.

6.2 Scan Performance

As discussed earlier, scans can benefit greatly from prefetching. This is especially true if it is possible to prefetch leafs sufficiently far ahead, since scans are more likely to be memory bandwidth bound than queries. Since there is a data dependency between leafs which limits how far ahead leafs can be prefetched, we need to have a leaf pointer array which allows us to prefetch further ahead. In this section we measure the update costs of the different ways of implementing a leaf pointer array and the performance benefits prefetching has on scans in terms of the available bandwidth used.

Key size (bytes)	8		16		32	
Node Width (cache-lines)	6-8-12	All	6-8-12	All	6-8-12	All
Insert	12.7%	6.3%	17.3%	9.8%	23.7%	13.3%
100%	14.7%	6.5%	19.7%	13.4%	25.0%	17.6%
67%	13.3%	6.6%	19.3%	13.3%	24.0%	16.8%
50%	9.3%	6.4%	16.3%	13.0%	24.7%	16.4%

Table 6.2: Prefetching speedup for different trees using optimized key comparison Averaged across node widths.

6.2.1 Leaf Pointer Array Update Cost

The leaf pointer arrays are necessary to allow scans to prefetch leafs farther ahead than just the current leaf. As described in Section 3.6, there are 3 suggested implementations. The simplest one, the *internal* leaf pointer array, uses the lowest node level, which already has pointers to all leafs. The *flat* leaf pointer array stores the pointers in one flat array outside the tree, and the *chunked* leaf pointer array stores the pointers in multiple smaller arrays (chunks), which are all linked together.

The leaf pointer arrays need to be updated during leaf splits. In order to measure the update costs for each one we set up the following experiment: we measure the operation time for different number of inserts, using key size 16B and node width 1CL, and chunk width of 4CL. The resulting tree has a small branch factor, and thus many leaf splits. The results can be seen in Table 6.3. As expected, the update cost of the *flat* leaf pointer array is prohibitive, and grows exponentially. Given this performance penalty, the *flat* leaf pointer array is not feasible and will not be discussed further. The additional cost of maintaining the *internal* leaf pointer array is effectively none. The additional cost of maintaining the chunked leaf pointer is little, since leaf splits require only the modifications of a single 4CL chunk. The last column in Table 6.3 shows the speed difference between maintaining the *chunked* and *internal* leaf pointer arrays. The *chunked* leaf pointer array maintenance overhead is never more than 6.4% of the entire update cost. Note that this experiment shows the worst case performance, and different node widths will result in fewer splits.

6.2.2 Prefetching

As mentioned earlier, the performance of scans is more likely to become memory bandwidth bound than queries. The maximum bandwidth our experiment system can theoretically achieve is 6.4 GB/s. Scans are very simple operations and mostly involve copying data, so it is interesting to see how well they utilize the available bandwidth. We set up a simple experiment, bulkloading trees with fill factor 100% and 50%, and over different key sizes using leaf prefetch distance 2 and measured the scan time. The node width used was the minimum allowed for each key size. The prefetching method used is *nt1*. In order to measure the bandwidth used, we need to know much data travels over the bus. When prefetching is enabled we use two measures, the number of cache-lines prefetched, and

Inserts	**Flat**	**Internal**	**Chunked**	**Difference**
	(in seconds)	**(in seconds)**	**(in seconds)**	
10K	0.0278	0.0195	0.0200	2.50%
20K	0.0947	0.0415	0.0439	5.88%
40K	0.375	0.0894	0.0937	4.92%
80K	2.12	0.201	0.213	6.33%
160K	16.6	0.447	0.473	5.79%
320K	92.3	0.978	1.04	6.39%
640K	Not measured	2.15	2.26	5.58%
1280K	Not measured	4.62	4.85	4.97%
2560K	Not measured	9.90	10.4	4.80%

Table 6.3: Update overhead for chunked leaf pointer array.

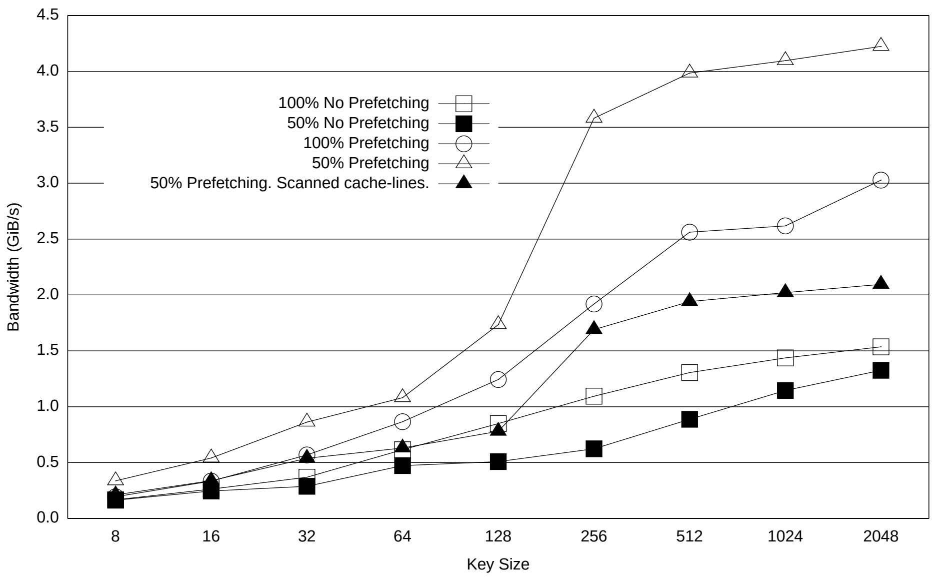


Figure 6.2: Memory bandwidth used during scans.

the number of cache-lines scanned. When prefetching is not used we only use the number of cache-lines scanned. The results can be seen in Figure 6.2.

The utilized bandwidth increases with increased key sizes, since the overhead of tree operations decreases relative to the amount of data copies. For key size 2048B and full trees we see that prefetching doubles the utilized bandwidth.

We have run benchmarks which read data from main-memory, performs minimal computation for each cache-line, and prefetches cache-lines. They show that it is possible to reach up to 6.25 GB/s bandwidth on the Itanium 2 processor, which is very close to the theoretical limit of 6.4 GB/s. Another benchmark, which is more similar to scanning, copies data from main-memory, and prefetches based on a pointer array. It can prefetch multiple consecutive cache-lines for each pointer, which mimics how the pB$^+$-tree prefetches nodes which are wider than one cache-line. It performs minimal computation on each cache-line. That benchmark has reached up to 5.8 GB/s. This benchmark shows how fast pB$^+$-tree scanning can be, given that there is very little overhead. As the results show, the bandwidth used when prefetching cache-lines of a half-full tree, is almost 75% of the measured limit. The bandwidth used when scanning a full tree is 3 GB/s for 2048B keys, which is 51.7% of the measured limit.

Chapter 7

Conclusions

In this thesis we have studied the performance of the pB$^+$-tree on the Itanium 2 processor. We have focused on the various implementation choices we faced and their effect on performance. While some of these choices have a marginal effect on performance, others affect it quite much; when all these gains are put together, the performance implications of using prefetching are considerable.

Our results have, e.g., validated the choice of binary search for the pB$^+$-tree [9], while apparently being in contrast with the results of [27]. Furthermore, we have demonstrated that careful coding of the prefetching loop and comparison operator, using Duff's device, leads to significant performance gains. Finally, we have shown which prefetching hints should be used for which operation, although the performance gains are less significant in this case.

The goal of this work is to move towards the deployment of cache-conscious structures "in the field". Future work includes further comparison with the pCSB$^+$-tree, as well as performance measurements on alternative architectures. Ultimately the goal is the integration of cache-conscious trees into a commercial quality DBMS. The best candidate is the InnoDB storage manager in the MySQL DBMS, since the source is available for free and its architecture abstracts the components which are responsible for storing data and providing access to it, making it easy to plug in different index structures.

References

[1] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[2] 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*, pages 266–277, Edinburgh, Scotland, 1999.

[3] J. Backus. Can crogramming be liberated from the von Neumann style? A Functional style and its algebra of programs. *Comm. ACM*, 21(8):613–641, Aug. 1978.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[5] M. Becker, N. Mancheril, and S. Okamoto. DBMSs on a modern processor: "Where does time go?" revisited. Technical report, CMU, 2004.

[6] P. Bonnet and I. Manolescu, editors. *Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems, 2006, Chicago, USA, June 30, 2006*. ACM, 2006.

[7] J. F. Cantin and M. D. Hill. Cache performance for selected spec cpu2000 benchmarks. *SIGARCH Computer Architecture News*, 29(4):13–18, 2001.

[8] S. Chen. *Redesigning Database Systems in Light of CPU Cache Prefetching*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[9] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–246, Santa Barbara, CA, United States, 2001.

[10] 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*, pages 157–168, Madison, Wisconsin, 2002.

[11] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[12] D. Comer. Ubiquitous $\underline{\ }$-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 2002.

[14] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[15] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1):13, Feb. 2001.

[16] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2004.

[17] J. Jannink. Implementing deletion in b+-trees. *SIGMOD Rec.*, 24(1):33–38, 1995.

[18] S. Jarp. A methodology for using the Itanium 2 performance counters for bottleneck analysis. Technical report, HP Labs, 2002.

[19] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of VLDB*, pages 294–303, San Francisco, CA, USA, 1986.

[20] P. Machanick. Approaches to Addressing the Memory Wall. Technical report, School of IT and Electrical Engineering, University of Queensland, 2002.

[21] R. Maelbrancke and H. Olivie. Optimizing jan jannink's implementation of $\underline{+}$-tree deletion. *SIGMOD Record*, 24(3):5–7, 1995.

[22] S. A. McKee. Reflections on the memory wall. In *Proceedings of the Conference on Computing Frontiers*, page 162, Ischia, Italy, 2004.

[23] M. Olsen and M. Kristensen. MySQL performance on Itanium 2. Technical report, DIKU, 2004.

[24] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2002.

[25] J. Rao and K. Ross. Making B^+-trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 475–486, Dallas, TX, USA, 2000.

[26] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of VLDB*, pages 78–89, Edinburgh, Scotland, 1999.

[27] M. Samuel, A. U. Pedersen, and P. Bonnet. Making CSB^+-trees processor conscious. In *Workshop on Data Management on New Hardware (DaMoN)*, Baltimore, MD, USA, 2005.

[28] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[29] J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[30] G. J. Ward. The radiance lighting simulation and rendering system. In *Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 459–472, New York, NY, USA, 1994.

[31] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[32] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *International Conference on Supercomputing*, pages 176–186, Santa Fe, NM, USA, 2000. ACM.

[33] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *Proceedings of VLDB*, pages 49–60, Trondheim, Norway, 2005.