Towards pB^{+}-trees in the Field: Implementation Choices and Performance

Árni Már Jónsson
Reykjavík University
Ofanleiti 2
IS-109 Reykjavík
Iceland
arnij@ru.is

Björn Þór Jónsson
Reykjavík University
Ofanleiti 2
IS-109 Reykjavík
Iceland
bjorn@ru.is

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 paper 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.

1. INTRODUCTION

Modern computers are based on the von Neumann architecture [16], 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 [6]. 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 [10].

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 [15]. 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 from cache (e.g., on Pentium 4 a cache miss can cost as many as 276 CPU cycles whereas reading data from L1 cache takes only 2 cycles [17]). 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.

Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB 2006), June 30, 2006, Chicago, Illinois, USA
Copyright 2006 ACM 1-59593-463-4 ...$5.00.

1.1 Cache-Conscious Indices

Database Management Systems (DBMSs) are not excempt from the main memory bottleneck. In their seminal paper, Ailamaki et al. [1] 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 [2], 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 [4], triggering the development of many cache-conscious indices. The CSS-tree [13] and CSB^{+}-tree [12] 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 [4] has nodes that are several cache-lines in width and uses prefetching to avoid preceived cache latency during index traversal. In [14] 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 [14], the CSB^{+}-tree was implemented and measured using the Itanium 2 processor. In [3] 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.

This paper 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. In contrast to [3], however, we focus on various implementation choices for the pB$^+$-tree and their effect on its performance.

1.2 Overview of the Paper

The remainder of this paper is organized as follows. Section 2 reviews the cache-conscious indices, in particular the pB$^+$-tree. Section 3 presents our experimental setup, while Section 4 desribes the implementation choices and their performance implications. Section 5 gives our conclusions.

2. CACHE-SENSITIVE INDICES

In this section we first review the state-of-the-art of cache-conscious indices and then describe the pB$^+$-tree, which is the subject of our performance study, in more detail.

2.1 Background

The most common database index is the B$^+$-tree. It is designed to minimize the number of disk operations, by having nodes the size of a disk page, but has been found to have poor cache performance [4]. Many cache-conscious (i.e., optimized with cache-memory in mind) variants of the B$^+$-tree have therefore been proposed.

Rao and Ross have proposed the Cache-Sensitive Search tree (CSS-tree) [13], as well as the Cache-Sensitive B$^+$-tree (CSB$^+$-tree) [12]. These data structures incur fewer cache-misses than the B$^+$-tree by eliminating (most) child pointers, thus placing more data into each cache-line. Both the CSS-tree and the CSB$^+$-tree have nodes the size of a cache-line, making the number of cache-misses incurred during search equal to the height of the tree.

In order to compute the addresses of children without the use of pointers, nodes are laid out in memory in a very strict manner. The CSS-tree is stored consecutively in memory, with each level of nodes stored right after the level above, with the exception that the leafs are stored consecutively, regardless of their level in the tree. Since the size of each level is known, the position of each node can be easily computed without storing any pointers. The CSS-tree does not support incremental updates, as the tree must be completely rebuilt, making the structure unsuitable for use with DBMSs.

The CSB$^+$-tree is more flexible, while retaining most of the performance benefits of the CSS-tree. The CSB$^+$-tree stores one pointer per node, which points to its first child. The children of each node are then stored consecutively in memory, allowing the location of any child to be determined using this single pointer. The CSB$^+$-tree does support incremental updates, but during node splits the whole set of siblings must be shifted to make room for the new node.

Chen et al. have proposed the Prefetching B$^+$-tree (pB$^+$-tree) [4] and the Fractal Prefetching B$^+$-tree (fpB$^+$-tree) [5]. The pB$^+$-tree is nearly identical to the B$^+$-tree, but with smaller nodes that are multiple cache-lines wide. Before a node is used, all of its cache-lines are prefetched. The wider nodes allow greater fan-out than with the CSB$^+$-tree. If enough cache-lines can be prefetched in parallel, the overhead of prefetching each node is more than offset by the reduction in tree height. Further details are presented in Section 2.2.

The Fractal Prefetching B$^+$-tree (fpB$^+$-tree) [5] replaces each node of a regular disk-optimized B$^+$-tree with a pB$^+$-tree (a tree inside a tree), thus optimizing both for main-memory and disk performance.

Samuel, Pederson and Bonnet have recently demonstrated that the pointer elimination technique of the CSB$^+$-tree and the wide nodes and associated prefetching of the pB$^+$-tree are complementary. They implemented the CSB$^+$-tree with wider nodes and node prefetching (pCSB$^+$-trees) and showed that siginficant performance gains are seen [14] over regular CSB$^+$-trees. pCSB$^+$-trees and pB$^+$-trees were compared in [3] and shown to perform similarly.

2.2 The pB$^+$-tree

The pB$^+$-tree [4] is a cache-conscious main-memory index, which has essentially the same structure as the B$^+$-tree, except that it has much smaller nodes. In contrast with the CSB$^+$-tree, however, each node has a width of multiple cache-lines, resulting in higher fan-out and lower trees.

The pB$^+$-tree prefetches all cache-lines of a node before it is used. Since the CPU can prefetch many cache-lines in parellel with other processing, the internal search of the node does not suffer a full cache-miss latency for each cache-line. As cache-lines are prefetched left-to-right, the internal format of nodes is changed slightly, such that all keys are stored together in the "left half", while the pointers (or records) are stored in the "right half". The keys are used much more than the pointers, so it makes sense to prefetche them first.

To give an idea of the potential benefits of prefetching, the Itanium-2 CPU McKinley [7] has a cache-miss latency of 288 cycles, and can have up to 16 outstanding cache-misses. If 16 cache-lines are prefetched, the first line can be delivered in 288 cycles and the last one in 558 cycles, or slightly less than two cache-miss latencies. Thus a node with a width of 16 cache-lines can be prefetched entirely in the time it takes to read two cache-lines.

While only a single node can be prefetched at a time during point queries (and inserts) as the search must always determine the correct child node to descend to, index scans have a more predictable access pattern, traversing the index leafs in order. If the optimal prefetching distance is greater than the node-width it is not possible, without further modifications, to prefetch sufficiently far ahead as to fully utilize the available memory bandwidth. This is because the leafs form a linked list and the address of the next leaf is not available until the sibling-pointer of the current leaf is in the cache. This problem is referred to as the "pointer-chasing" problem and can be solved by storing a separate "jump-pointer array", thus avoiding the dependency between neighbouring leafs.

The proposed implementation of the jump-pointer array involves splitting the jump-pointer array into chunks, with the chunks forming a linked list themselves. Only pointers within a single chunk need to be shifted after a leaf split. The leafs store a back pointer to allow for quickly finding the starting point for the prefetching. Some further optimizations are proposed; see [4] for details.

3. EXPERIMENTAL SETUP

In this section, we describe our experimental setup. Section 3.1 describes the Itanium 2 processor used in our experiments, while Section 3.2 details the compiler used and the associated flags. Section 3.3 describes the workloads used in our experiments and Section 3.4 the metrics used for evaluation. This section list the most relevant information; the software and experimental framework used, as well as more detailed information on how to reproduce the results, is presented in [9] and available at http://datalab.ru.is/csi.

Cache	Size	Cache-Line	Integer Load Latency	Associa-tivity
L1I	16K	64 bytes	1 cycle	4
L1D	16K	64 bytes	1 cycle	4
L2	256K	128 bytes	5 cycles	8
L3	1.5MB	128 bytes	12 cycles	12

Table 1: Itanium 2 processor cache structure.

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

Table 2: Itanium 2 prefetching locality hints.

3.1 The Itanium 2 processor

The experiments are performed on a dual CPU Itanium 2 server, running at 900Mhz, with 4GB of main-memory. The CPUs are 2nd generation Itanium CPUs (code name McKinley). These 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 caching subsystem are of primary importance, so the remainder of this section gives an overview.[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 are shown in Table 1.

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. If no cache-line with an unset bit is found, then all bits are reset.

Prefetching

The Itanium 2 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 reading the new sibling node during a node split.

Fault hint: The fault hint controls whether the cache-line should be fetched if the page it belongs to is not in the TLB. The fault hint should be avoided for speculative prefetches, but should be appropriate for the index operations of the pB$^+$-tree, which have a predictable access pattern.

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 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.

[1]A good reference manual for the Itanium 2 processor is available from Intel [7].

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

Data Alignment

Data alignment is very important when working with integer data on the Itanium 2. 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. 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 care to align the key- and pointer-arrays to 8-byte boundaries.

3.2 Compiler

We use version 8.0.066 of Intel's C Compiler (*icc*) for Linux, which has very good support for the Itanium 2 processor. While the *gcc* compiler was preferred in [3], we found *icc* to give superior performance in our early experiments. Furthermore, it performed much better in our experiments with the STREAM benchmark.[2] Finally, the *icc* compiler was also preferred in [14], which used the same computer as we have used. We used the following compiler flags:

`-O2` Enables non-agressive 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 interprocedural optimizations.

More aggressive optimizations were tested, but resulted in performance losses.

3.3 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.

[2]Available at ftp://ftp.cs.virginia.edu/pub/stream/.

Category	Formula	Meaning
D-cache stalls	BE_EXE_BUBBLE_GRALL - BE_EXE_BUBBLE_GRGR + BE_L1D_FPU_BUBBLE_L1D	Stalls incurred waiting for data to be delivered to cache.
Instruction miss stalls	FE_BUBBLE_IMISS*	Stalls incurred waiting for instructions to be delivered to cache.
Branch misprediction	BE_FLUSH_BUBBLEBRU + FE_BUBBLE.BUBBLE* + FE_BUBBLE.BRANCH*	Stalls incurred when the pipeline is flushed because of a branch misprediction.
RSE stalls	BE_RSE_BUBBLE.ALL	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.
FPU stalls	BE_EXE_BUBBLE.FRALL + BE_L1D_FPU_BUBBLE.FPU	Stalls in the floating point micropipeline.
GR Scoreboarding	BE_EXE_BUBBLE_GRGR	Integer register dependancy stalls.
Front-end Flushes	FE_BUBBLE_FEFLUSH*	Cycles lost due to flushes in the front-end.
Busy	CPU_CYCLES minus the sum of the above	Unstalled CPU cycles.

Table 3: A description of CPU stall cycle categories and their related events (borrowed from [11]).
*** Counters starting with FE_ must be scaled with the reduction factor** $\frac{\textbf{BACK\_END\_BUBBLE\_FE}}{\textbf{FE\_BUBBLE\_ALLBUT\_IBFULL}}$

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 operations.

Keys used with repeated insertions 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 and 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 node to the new sibling node.

Note that pointers are always assumed to be 8 bytes, while no data is stored in the leafs, only keys. Keys are never padded; if keys must be 8 byte aligned for performance reasons, their size should be a multiple of 8 bytes.

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; if 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.

3.4 Performance Metrics

The primary performance metric used in our performance study is the running time of each operation, measured using the *getrusage* system call which has a resolution of 10 ms.

For more detailed analysis of the running time, we use *pfmon (version 2.0)* and construct a detailed stall cycle breakdown, described below. Furthermore, we have sometimes used the HPC-toolkit for a detailed correlation of the CPU events with the source code for better understanding of the performance results.

Each measuremnt is the average of 14 or 30 runs, with the 2 or 5 highest and lowest results removed, respectively.

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 [8] has suggested a methodology for this grouping, which we have adopted. The stall categories, the associated performance counters, and a description of the stall cause can be found in Table 3, which is borrowed from [11].

To sample the values of the performance counters we use a tool called *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 instructs it 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. Our experimental software uses this functionality by calling an empty function just before an operation starts, to avoid measuring the overhead of the 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 the 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

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: Comparison of *getrusage* and *pfmon*. [1M records; 5 CL nodes.]

results, which are shown in Table 4, show that as the error does not grow with increased running time or key size (and is never more than $\pm1.01\%$) the two methods of measuring running time are comparable.

Source-Code/CPU Event correlation

The *HPC-toolkit*[3] is a set of tools used to analyse 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 occured. 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. IMPLEMENTATION CHOICES

We implemented the pB$^+$-tree based on the description in [4] 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 choose arbitrarily, we have implemented those options; this section analyses the performance tradeoffs. The options we have identified are shown in Table 5. 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.

4.1 Prefetching Hints

In [3], 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 the prefetching command is cancelled and subsequent accesses to the node will all incur full cache latency. It does appear to make sense to use this hint as 1) the access pattern is very predictable, and 2) the associated cost of a miss is high.

We set up the following experiment: We used a small pB$^+$-tree (1M keys; 8B keys; 4CL (cache-line) nodes) and a large pB$^+$-tree (1M records; 64B keys; 32CL nodes). We ran 1M queries and 10 scans over each tree, comparing the running time. While the expectation was to see better performance with the faulting hint, particularly in the case of

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

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: Implementation options and choices.

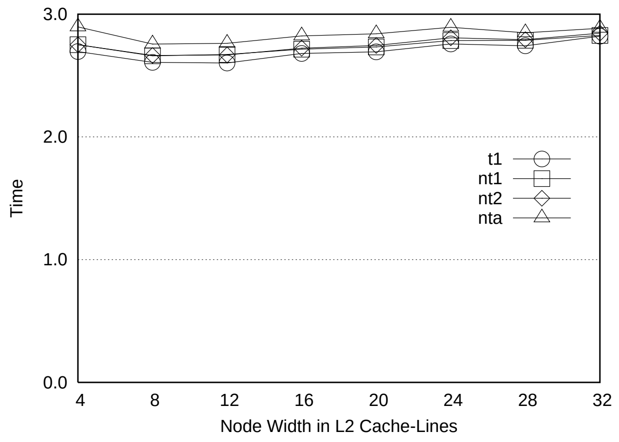

Figure 1: Comparison of locality hints for queries. [1M records; 1M queries; 16B key; 67% fill f.]

the scan of the large tree, the results showed no 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 locality hints, the expectation was that each operation would have its preferred hint. We expected the *nta* hint to perform best for the node insertion part of the insert operation, but worst for the other operations. The performance tradeoffs of the other hints were less clear.

Figure 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.

The insertion operation has two phases. First, a point query is used to find the correct location for the record. 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 2 shows

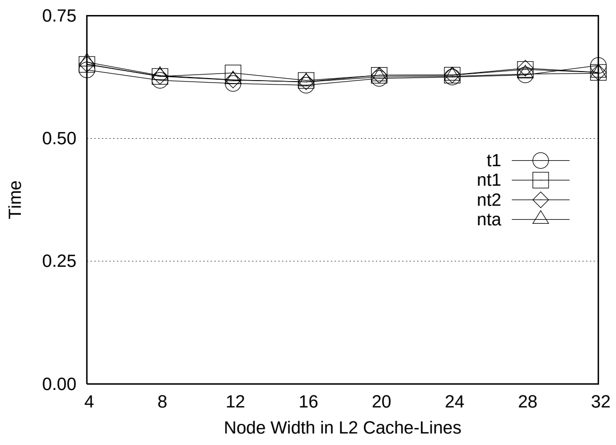

Figure 2: Comparison of locality hints for scans. [1M records; 10 scans; 16B key; 67% fill f.]

```
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 3: Pseudo-code for Duff's device.

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—about 1–2% better in most cases. The *t1* hint performs best in most other cases we have seen, although the difference is always quite small.

4.2 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[4] which allows unrolling loops of arbitrary length. The pseudo-code for this unrolling is shown in Figure 3, where the `pf()` macro performs the actual prefetching. The loop is based on a trick allowed by C syntax, whereby 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 4 shows the performance of the regular and unrolled loops for 1M queries and 10 scans (it has no additional effect on the inserts), over trees with 16B keys. As the figure shows, the unrolled loop performs better in all cases, in most cases running 4–5% faster than the original loop. We therefore recommend using the Duff's device in all cases.

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

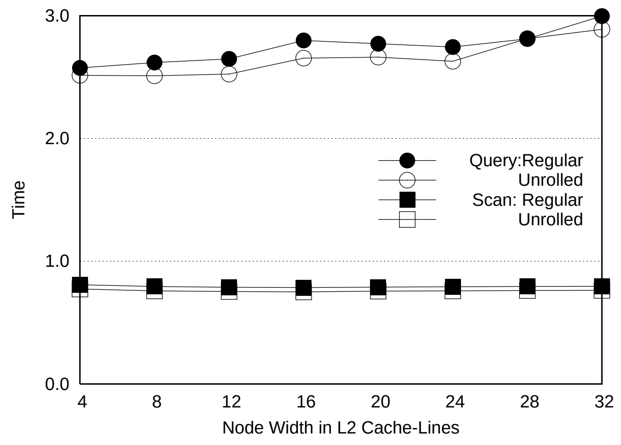

Figure 4: Comparison of loop unrolling options. [1M records; 1M Qs.&10 Scans; 16B key; 100% fill f.]

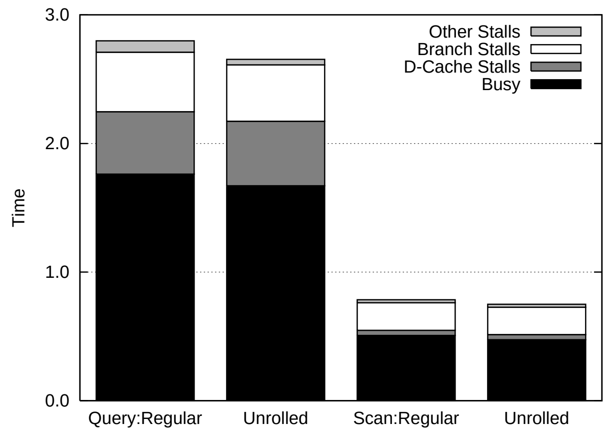

Figure 5: Stall cycle analysis for loop unrolling. [1M rec.; 1M Qs.&10 Scans; 16B k.; 16CL n.; 100% ff.]

Figure 5 shows a more detailed breakdown of the costs of the queries and scans, for nodes of 16 cache-lines. The figure shows that the primary benefit for both query and scan comes from a reduction in busy time because of the reduced loop overhead. The percentage of branch stalls does not decrease since the number of loop iterations is still very predictable. Various other stalls are reduced, for some unidentified reason.

4.3 Node Prefetching

Turning to the implementation options for node prefetching, Chen at al. suggest prefetching all cache-lines of a node, left-to-right. But since a node does not necesserily 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, when nodes are not full.

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 best overall 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 precalculated for all possible number of keys in nodes and leafs and stored in a look-up-table. There is however 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 further prefetching statements until the node header is in the cache. Note also 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.

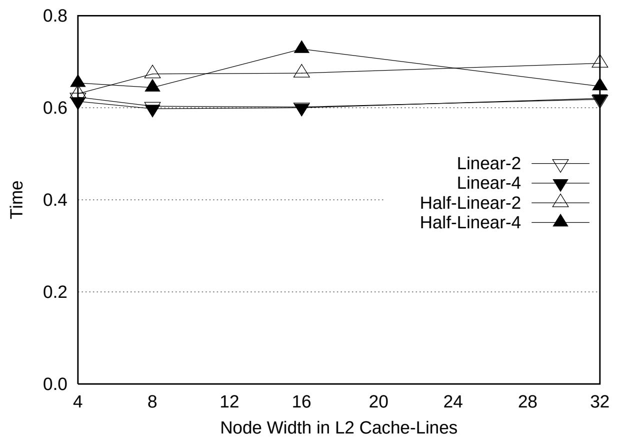

Figure 6: Performance of Linear prefetch. policies. [1M rec.; 10 Scans; 8B keys; 67% fill f.]

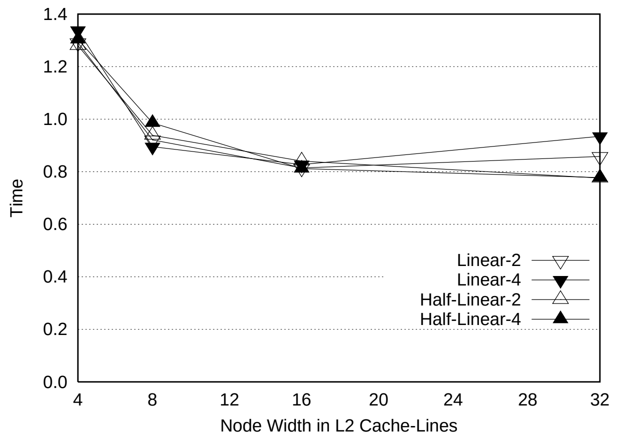

Figure 7: Performance of Linear prefetch. policies. [1M rec.; 10 Scans; 64B keys; 67% fill f.]

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 must always wait for the appropriate child to be found. Since the Linear policy can very efficiently prefetch a full node, there is not much performance 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. Figure 6 shows the performance of the two Linear policies with different prefetching distances of two and four nodes, respectively, when scanning an index with 8 byte keys, which is two-thirds full. The figure shows that there are no performance benefits using the Half-Linear policy in this case.

Turning to Figure 7, which shows the same policies, but this time for larger search keys of 64 bytes, we observe that while the Linear policy is slightly better with small nodes sizes, the Half-Linear policy is now faster for larger node sizes. Overall, we have found that the Half-Linear policy only outperforms the Linear policy when key size is at least 64 bytes and the fill factor is no more than 67%.

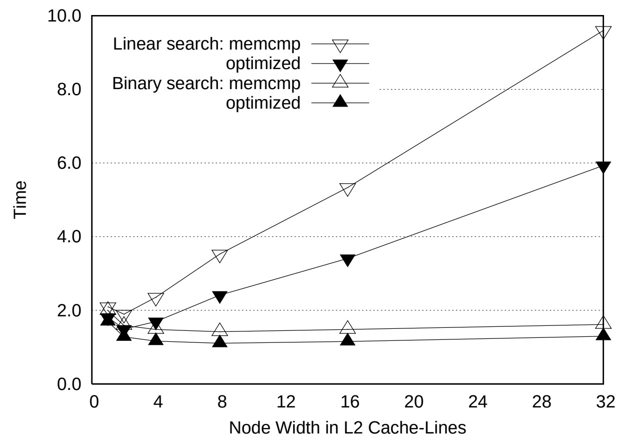

Figure 8: Performance of search and comparison. [1M records; 1M Qs.; 16B key; 100% fill f.]

4.4 Search Algorithm and Comparison

B$^+$-trees have traditionally used binary search within the nodes and, according to [3], pB$^+$-trees have inherited that behavior. In [14], 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 8 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.

5. CONCLUSIONS

In this paper 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 effects 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, they become significant.

Our results have, e.g., validated the choice of binary search for the pB$^+$-tree [4], while apparently being in contrast with the results of [14]. 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.

As outlined in the introduction, this paper is part of a trend towards the deployment of cache-conscious structures "in the field". Future work includes a further comparison with the pCSB$^+$-tree, performance measurements on alternative architectures, and ultimately the integration of the pB$^+$-tree, and the fpB$^+$-tree, into the MySQL DBMS.

6. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of VLDB*, pages 266–277, Edinburgh, Scotland, 1999.

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

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

[4] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of ACM SIGMOD*, pages 235–246, Santa Barbara, California, United States, 2001.

[5] 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.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2002.

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

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

[9] Á. M. Jónsson. Performance considerations for the prefetching B$^+$-tree. Master's thesis, Reykjavík University, Reykjavík, Iceland, 2006. In preparation.

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

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

[12] J. Rao and K. Ross. Making B$^+$-trees cache conscious in main memory. In *Proceedings of ACM SIGMOD*, pages 475–486, Dallas, TX, USA, 2000.

[13] 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.

[14] 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.

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

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

[17] 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.