Effect of Node Size on the Performance of Cache-Conscious B$^+$-trees

Richard A. Hankins
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122, USA.
hankinsr@eecs.umich.edu

Jignesh M. Patel
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122, USA.
jignesh@eecs.umich.edu

ABSTRACT

In main-memory databases, the number of processor cache misses has a critical impact on the performance of the system. Cache-conscious indices are designed to improve performance by reducing the number of processor cache misses that are incurred during a search operation. Conventional wisdom suggests that the index's node size should be equal to the cache line size in order to minimize the number of cache misses and improve performance. As we show in this paper, this design choice ignores additional effects, such as the number of instructions executed and the number of TLB misses, which play a significant role in determining the overall performance. To capture the impact of node size on the performance of a cache-conscious B+-tree (CSB+-tree), we first develop an analytical model based on the fundamental components of the search process. This model is then validated with an actual implementation, demonstrating that the model is accurate. Both the analytical model and experiments confirm that using node sizes much larger than the cache line size can result in better search performance for the CSB+-tree.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design; I.6 [**Simulation and Modeling**]: Miscellaneous; E.1 [**Data Structures**]: *Trees*

General Terms

Algorithms, Measurement, Performance, Design

Keywords

Index, Cache-Conscious, B$^+$-tree

1. INTRODUCTION

Systems with large main memory configurations are becoming more prevalent, due in large part to the decreasing price and the increasing capacity of random access memory chips. Consequently, it is economical and desirable to configure database servers with large amounts of main memory; in such configurations, the entire database, or most of the frequently accessed parts of the database, resides in main memory. As this trend continues, it is expected that, in the future, all but the largest data sets will be resident in main memory [3].

Traditional databases are designed to reduce the number of disk accesses, since accessing data on the disk is orders of magnitude more expensive than accessing data in main memory. With data sets becoming resident in main memory, the new performance bottleneck is the latency in accessing data from the main memory [2,6,25, 27, 28]. Since accessing data in main memory is "expensive" relative to the processor speeds, modern processors make use of *processor caches*. A processor cache is a block of low-latency memory that sits between the processor and main memory, and stores the contents of the most recently accessed memory addresses. Latency in retrieving data from the cache is one to two orders of magnitude smaller than the latency in retrieving data from the main memory; therefore, careful utilization of the processor caches can result in large overall performance improvements.

Modern processors typically have separate caches for instructions and data, with multiple levels of these caches. The first two levels of data cache memory are denoted as L1-D cache and L2-D cache respectively. It has been proven that database systems experience a significant number of L2-D cache misses, and these misses contribute substantially to the overall execution time [2]. In this paper, we are primarily concerned with L2 data cache-misses, and for the rest of the paper we will simply refer to these as cache misses.

In addition to cache misses, another significant factor that can impact the performance of main-memory database systems is the number of *TLB misses*. TLB misses can occur when a virtual memory address is translated into a physical memory address. Within a process, both data and code segments are allocated to locations in the process's virtual address space. However, at the hardware-level, all memory accesses use physical addresses. To access the actual item (data or instruction), the operating system must translate the virtual address of the item to its physical memory address. These address translations are stored in a *page table* that resides in main-memory. An entry in the page table maps a virtual page address to the physical page address. For every address translation request, the page in which that address resides is identified, and the corresponding page table entry is retrieved. To improve the performance of this operation, a *translation look-aside buffer* (TLB) is used to cache the most recent address translations. Main-memory is accessed only if the address translation is not found in the TLB. As with processor cache misses, TLB misses are expensive as they can require accessing the main memory.

A frequently performed operation in database systems is evalu-

SIGMETRICS'03, June 10–14, 2003, San Diego, California, USA.

ating equality-based searches. Given the importance of equality-based searches, a number of cache-conscious index structures have been proposed to speedup this operation [5, 22, 26, 27]. These cache-conscious access methods include tree-based access methods, such as the CSS-trees, CSB$^+$-tree, and T-trees; and hash-based access methods, such as extendible hashing [22] and chained bucket hashing [26]. These techniques primarily focus on arranging the data in the access method's data structure to reduce the number of cache misses, and produce significant improvements over the traditional disk-based indices.

A design decision that is consistently used in cache-conscious indices is defining the node size to be equal to the size of the L2 data cache line. This is analogous to defining the node size to be equal to the disk page size in traditional disk-based database environments. On the first access to a node, its entire content is copied from main memory into the cache. All subsequent accesses to this node can be satisfied by reading data from the processor cache, thereby avoiding the long latency associated with reading data from main memory. Even query processing algorithms, such as hash-based join and aggregate operations, have used this idea of setting the node size equal to the cache line size for constructing their internal in-memory indices [17]. As we demonstrate, this choice is often suboptimal for cache-conscious access methods when running on modern processors.

While both tree-based and hash-based indexing techniques are of interest within the context of memory-resident databases, in this paper we concentrate on the more popular of these indexing structures, namely the B$^+$-tree. (See the technical report [19] for analysis and evaluation of hash-based cache-conscious indices).

The disk-based B$^+$-tree has been shown to have poor processor cache performance, and modifications to the B$^+$-tree have been shown to dramatically improve the performance of B$^+$-tree in main-memory environments. The modified indexing structure is called the CSB$^+$-tree [27], and it is the indexing structure that we examine in this paper. We analyze the effect of node size on the CSB$^+$-tree and make the following contributions:

- Using a first-order analytical model of the search performance, we show that the conventional choice of setting the node size equal to the cache line size is often suboptimal. This design choice focuses on reducing the number of cache misses, but ignores the effect on the number of instructions that are executed, the number of conditional branches mispredicted, and the number of TLB misses.
- Using a simple first-order model for the space requirements, we show that a large node size is not only more time efficient, but it is also much more space efficient.
- Using an actual implementation of the index, we validate both the search performance model and the space model. We show that when executing equality searches, a node size of 512 bytes or larger generally results in better performance than the conventional cache-conscious node size equal to the cache line size. Compared to the conventional node size, a larger node size can improve the performance of the CSB$^+$-tree by **17%** (1.21 speedup) for equality searches on the Intel Pentium III.[1] Larger node sizes are also found to improve the space utilization by up to **57%** compared to a CSB$^+$-tree constructed using the conventional node size.

The remainder of this paper is organized as follows: Section 2 briefly describes the CSB$^+$-tree index. Section 3 presents analytical models for the index. Section 4 presents results based on an actual implementation of the index. Section 5 discusses related work, and finally, we present our conclusions and directions for future work in Section 6.

[1]These commonly used performance metrics are defined in Section 4.4

2. CSB$^+$-tree

A CSB$^+$-tree [27] is an adaptation of the ubiquitous B$^+$-tree for main memory databases [27]. The CSB$^+$-tree is an important data structure for memory-resident databases as it has been shown to outperform other cache-conscious, tree-based indices as well as traditional, tree-based indices in memory-resident databases [27].

Figure 1 shows an example of a CSB$^+$-tree. Like the B$^+$-tree, data is stored in the leaf nodes. These data entries are of the form: $\langle key, pointer \rangle$, where key is the key value for the record pointed to by $pointer$. In a regular B$^+$-tree, a non-leaf node has a similar structure except that the $pointer$ points to another node in the index. key values in non-leaf nodes guide the search operation to the appropriate leaf node(s). The key difference between the CSB$^+$-tree and the B$^+$-tree is in the structure of the non-leaf nodes. In a non-leaf node of the CSB$^+$-tree, there is only *one* pointer. This pointer references the start of a collection of child nodes, called a *group*, where each node in the *group* is allocated contiguously in memory. The parent node only points to the head of the group; thus, the address of a child node can be computed from this pointer and the ordinal number of the child in the parent node. By eliminating the child node pointers in the non-leaf nodes, additional keys can be stored which results in more efficient use of the space used by a non-leaf node. Node size is typically set to the cache line size.

To illustrate the use of the index during a search operation, consider the retrieval of the record identifier (RID) associated with the key value 3 on the index shown in Figure 1. The search operation first begins at the root node *A*, where there is only one key value, 10. The search value, 3, is less than 10 so the search continues to the first child node, *B*. The search key, 3, is greater than 2 but less than 4, indicating the search should continue to the second child node, *E*. *E* is a leaf node which contains a number of $\langle key, pointer \rangle$ pairs. A binary search inside node *E* is then used to find the key value 3, and the corresponding pointers identify the records that match the search. If required, the pointers are followed to retrieve the actual records. In main-memory databases, following the pointer is equivalent to dereferencing a memory address.

3. INDEX STRUCTURE ANALYSIS

In this section, an analytical model of the CSB$^+$-tree's search performance is presented. The model is then used to examine the effect of the node size on the search performance. Next, an analytical model of the index's space requirements is introduced, and then used to analyze the effect of node size on the space required to store the index.

The following analytical models rely on the input parameter definitions shown in Table 1 and the model parameters shown in Table 2. The values for the architecture and index parameters in Table 2 are *estimates* based upon our implementation of the CSB$^+$-tree executing on the Pentium III architecture. While only a single architecture is examined in this section, the analytical models are not dependent on particular values, allowing one to study the performance effects of modifying these parameter values.

3.1 Analytical Model for Execution Time

The cost of executing an index search is modeled as a function of four variables: the instruction count (I), the number of data cache

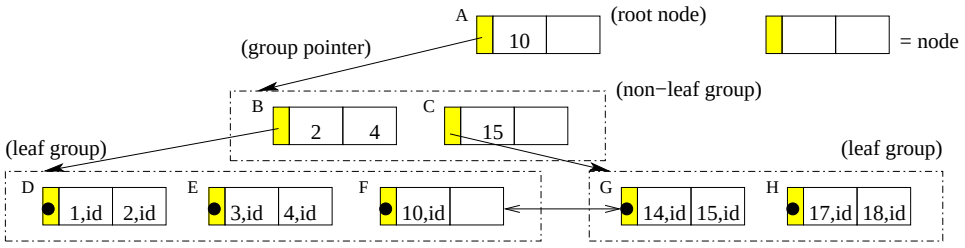

Figure 1: CSB$^+$-tree

Variable	Description	Value
	Input Parameters	
$node\_sz$	size of an index node (in bytes)	varies
q	number of queries	varies
σ	cardinality of the index	varies
	Output Parameters	
M_{btree}	number of cache misses incurred in searching the CSB$^+$-tree	Eq. 10
M_{spec}	number of cache misses incurred in searching the CSB$^+$-tree — speculative	Eq. 14
I_{btree}	number of instructions executed in searching the CSB$^+$-tree	Eq. 15
B_{btree}	number of branch paths incorrectly predicted in searching the CSB$^+$-tree	Eq. 16
T_{btree}	number of TLB misses incurred in searching the CSB$^+$-tree	Eq. 18
$space$	space required to construct a CSB$^+$-tree (in bytes)	Eq. 19
$space_{full}$	space required to construct the "full" CSB$^+$-tree (in bytes)	Eq. 21

Table 1: Input and Output Parameters for the Model

misses (M), the number of branch mispredictions (B), and the number of TLB misses (T). There are additional factors that contribute to the execution time, including instruction cache misses [2]. Instruction cache misses can become a significant component of the overall execution time, especially if the query workload requires executing code paths with large footprints. For this study, the L1 instruction cache effects are ignored because we only execute queries on a single index structure, and the code footprint fits well within the L1 instruction cache. Our assumption is supported by empirical measurements showing that the instruction cache misses contribute less than 0.5% to the overall execution time.

An additional factor that may affect execution time is out-of-order execution. Out-of-order execution enables a processor to potentially hide a portion of the observed stall time by exploiting instruction-level parallelism. Given that the primary operation in searching the CSB$^+$-tree is a binary search, which has a short code path and contains data dependencies, there is little opportunity to exploit instruction-level parallelism; therefore, we ignore out-of-order effects.

The execution time model is shown in Equation 1:

$$t = I * cpi + M * miss\_latency + B * pred\_penalty + T * tlb\_penalty \quad (1)$$

In Equation 1, t is the total execution time, in processor clock cycles, cpi is the cost of executing an instruction, $miss\_latency$ is the cost of servicing a cache miss, $pred\_penalty$ is the cost of incorrectly predicting a branch path, and $tlb\_penalty$ is the cost of retrieving a TLB entry from the page table in main-memory. The cpi, $miss\_latency$, $pred\_penalty$, and $tlb\_latency$ costs are all in processor clock cycles. Estimates for the cpi, $miss\_latency$, $pred\_penalty$, and $tlb\_penalty$ parameters can usually be extracted from a processor's design manual. The values for these architecture parameters for a Pentium III are shown in Table 2. We note that the cpi parameter was estimated based on the observed behavior over a number of actual index searches. Also, the TLB miss penalty is assumed to be the same as the L2 cache miss penalty, which ignores the effects of caching page table entries in the processor caches. As we show in our model, the majority of TLB misses are compulsory misses that occur at the lowest levels of the tree, so the page table entries are less likely to be cached. In addition, the TLB miss penalty that we have chosen is very conservative, so page table entries that are found in the cache are not severely penalized with the full TLB miss latency.

3.1.1 Cache Miss Component

During an equality search operation, cache misses are incurred as the search proceeds down the index tree, performing a binary key-search at each level. The number of cache misses for this binary key-search, m_{node}, can be computed as:

$$l = \left\lceil \frac{node\_size}{lsz} \right\rceil \quad (2)$$

$$k = \left\lfloor f * \frac{node\_sz - nlmeta\_sz}{key\_sz} \right\rfloor \quad (3)$$

$$k_{init} = \frac{lsz - nlmeta\_sz}{key\_sz} \quad (4)$$

$$m_{node} = \begin{cases} \log_2(l+1) + (1 - \frac{k_{init}}{k}), & \text{if } k > k_{init} \\ \log_2(l+1), & \text{if } k <= k_{init} \end{cases} \quad (5)$$

Each non-leaf node contains a child pointer, a key count, and a list of *key* values. The size of the child pointer and key count metadata is represented by the $nlmeta\_sz$ term. The number of cache

Variable	Description	Value
Architecture Parameters (Pentium III)		
cpi	processor clock cycles per instruction executed	0.63
lsz	size of a cache line (in bytes)	32
$miss\_latency$	processor clock cycles per L2 cache miss	75
$page\_sz$	size of a page in main-memory (in bytes)	4096
$pred\_penalty$	branch misprediction penalty in processor clock cycles	15
tlb_cap	number of entries in the TLB	64
$tlb\_penalty$	processor clock cycles to retrieve a TLB entry	75
Index Parameters		
I_{search}	instructions to compare a key and select the next position in the binary search	6
I_{trav}	instructions per node traversal	30
$entry\_sz$	size of an entry in a leaf node (in bytes)	8
f	fill percentage of a node	0.67
key_sz	size of an index key (in bytes)	4
$nlmeta\_sz$	size of the non-leaf's metadata (in bytes)	8
$lmeta\_sz$	size of the leaf's metadata (in bytes)	12
Computed Values		
l	number of cache lines spanned by a node	Eq. 2
k	number of keys in a non-leaf node	Eq. 3
k_{init}	number of keys in the first cache line of a node	Eq. 4
m_{node}	number of cache misses incurred in a binary search of a node	Eq. 5
L	number of leaf nodes in the index	Eq. 6
bf	branching factor	Eq. 7
h	height of the tree	Eq. 8
bf_{root}	branching factor of the root node	Eq. 11
λ_i	number of nodes in a single level of the index	Eq. 12
m_{spec}	number of cache misses incurred in a binary search of a node — speculative	Eq. 13
p_i	number of main-memory pages containing a single level of the index	Eq. 17
bf_{max}	maximum branching factor	Eq. 20

Table 2: Internal Parameters for the Model

misses incurred by a binary search inside a node (Eq. 5) depends on the number of cache lines spanned by the node (Eq. 2), plus an additional cost to read the node's child pointer and key count data. Because a node's key entries begin immediately after the meta-data, the cost of accessing the meta-data is adjusted based on the probability of revisiting the cache line during the search. The probability of revisiting the cache line containing the first few keys (Eq. 4) depends on the total number of keys in the node (Eq. 3).

The number of cache misses incurred as the search operation traverses down the tree is bounded by the height of the tree, h, which is computed as follows:

$$L = \frac{\sigma}{\left\lfloor f * \frac{node\_sz - lmeta\_sz}{entry\_sz} \right\rfloor} \quad (6)$$

$$bf = k + 1 \quad (7)$$

$$h = \lceil \log_{bf} L \rceil + 1. \quad (8)$$

The height of the tree (Eq. 8) is dependent on the number of leaf nodes (Eq. 6) in the index as well as the branching factor (Eq. 7.) The number of leaf nodes, L, is computed by dividing the cardinality, σ, by the number of entries per leaf node. Each leaf node contains a count, pointers to both the right and left sibling groups, and a list of $\langle key, rid \rangle$ pairs. The entry count and pointers are considered meta-data and are accounted for by the term $lmeta\_sz$. The branching factor, bf, is the number of children a non-leaf node can reference, and is dependent on the number of keys that are contained in the non-leaf nodes (Eq. 3).

On the first traversal of the tree, each node access will incur a compulsory cache miss. However, on subsequent traversals (queries), nodes near the root of the tree will have a high probability of being found in the processor cache, while the leaf nodes will have a substantially lower probability. To model this effect, we apply Cardenas's formula [8] *at each level* of the tree since the probability of a node being cached depends on the number of nodes at that level, which in turn depends on the level of that node.

We note that this problem is very similar to estimating the number of page misses in a buffer pool, and databases frequently use Yao's formula for this purpose [13]. Yao's formula [32] computes the average number of pages, or blocks, accessed during query evaluation. Since the records that satisfy a query are unique, Yao's formula assumes that records in the blocks will not be revisited. In the case of modeling cache misses, Cardenas's formula is more appropriate because we are interested in predicting the number of unique blocks accessed with the possibility of replacement. In other words, the data contained within a block may be revisited during subsequent queries.

In using Cardenas's formula, we are modeling a system of continuously running queries, i.e. the steady state behavior. It is important to note that the processor cache can be diluted by other programs, removing highly accessed data from the cache. In using Cardenas's formula to account for the actual cache misses during a search, we assume that the database application is a high priority process, and that the interference from the OS or other applications is marginal.

Continuing with the description of the model, Cardenas's formula, shown in Equation 9, predicts the number of unique blocks that are visited, X_D, for a given number of queries, q, on a given number of data blocks, λ; in our case λ is the total number of nodes for a given level of the tree.

$$X_D(\lambda, q) = \lambda * (1 - (1 - 1/\lambda)^q) \tag{9}$$

The total number of cache misses, M_{btree}, is modeled as the sum of the expected number of unique cache misses at each level of the tree.

$$M_{btree} = \frac{\sum_{i=1}^{h} X_D(l * \lambda_i, q * m_{node})}{q} \tag{10}$$

In Equation 10, $l * \lambda_i$ is the number of cache lines spanned by all the nodes at level i of the tree, and $q * m_{node}$ is the total number of cache lines accessed for q queries at each level i. λ_i can be estimated as follows:

$$bf_{root} = \frac{L}{bf^{h-2}} \tag{11}$$

$$\lambda_i = \begin{cases} bf_{root} * bf^{i-2} & \text{if } i > 1 \\ 1 & \text{if } i = 1 \text{ (root)} \end{cases} \tag{12}$$

In Equation 12, bf_{root} is the branching factor of the root node, and bf is the average fanout of a node. We treat the root node as a special case because the number of entries in the root node greatly influences the number nodes at each level of the tree. Without this special case, we could get a very different value of λ for each level of the tree, leading to an overestimation of the number of cache misses.

Equation 10 accounts for compulsory misses, but does not incorporate conflict misses or capacity misses in the cache [20]. For a first-order approximation, we assume a large cache where these effects are not significant. The experimental results in Section 4 confirm the accuracy of this simplification, but in future work we plan to address this with a more detailed cache model.

Modern processors use branch predictors to predict the target of a branch instruction and speculatively execute instructions from the predicted target. The actual target of the branch instruction is computed several cycles later, during the execution of the branch instruction. If the actual target does not match the predicted target, the processor pipeline is flushed and execution resumes at the correct target. During this speculative phase, a load instruction might initiate a cache miss. Once the branch instruction is executed, and if the branch target was predicted incorrectly, the cache miss due to the load instruction will no longer remain on the critical path of execution. As a result, a cache miss that results from speculative execution may be registered by the processor's event counter, even though the full cache miss latency will not be present in the execution time. It is important to account for these speculative cache miss events to correctly estimate the observed stall time due to cache misses. Equation 13 modifies the binary-search cost-equation (Eq. 5) to account for speculative memory accesses.

$$m_{spec} = \begin{cases} 1.5 * \log_2(l+1) - 0.5 + (1 - \frac{k_{init}}{k}) & \text{if } k > k_{init} \\ 1.5 * \log_2(l+1) - 0.5 & \text{if } k <= k_{init} \end{cases} \tag{13}$$

Equation 13 assumes that the search key is random, and consequently, the processor mispredicts the search direction half of the time. Based on this assumption, the number of cache line accessed is equal to the number of non-speculative cache reads due to the binary search, plus an additional 50% due to branch misprediction. When including speculative memory accesses, Equation 14 can be used to predict the number of cache miss events that an out-of-order capable processor is likely to register.

$$M_{spec} = \frac{\sum_{i=1}^{h} X_D(l * \lambda_i, q * m_{spec})}{q} \tag{14}$$

3.1.2 Instruction Component

The second component of the analytical model is the number of instructions executed during a search operation. The number of instructions executed includes a binary key-search of the entire data, plus the cost of a child-node traversal. Equation 15 predicts the number of instructions executed.

$$I_{btree} = \log_2(\sigma) * I_{search} + h * I_{trav}. \tag{15}$$

In Equation 15, I_{search} is the number of instructions required to evaluate a key and select the next evaluation position in the binary search, and I_{trav} is a fixed cost for traversing to the next node in the tree. The values for these parameters are estimates based on the actual implementation, and values for these parameters are shown in Table 2.

3.1.3 Branch Prediction Component

The third component of the model is an estimate of the number of conditional branches that are incorrectly predicted by the architecture. The number of mispredicted branches is proportional to the number of binary search operations executed plus the conditional branch that ends the binary search of a node. On each key comparison in a binary search, the next key examined may be before or after the present key, with each path having an equal probability of being taken. Therefore, the processor has an equal probability of predicting the correct or incorrect path. Also, the processor can not accurately predict the conditional branch that is taken to end the search. Consequently, the number of mispredicted branches, B_{btree}, is estimated to be 50% of the branch instructions that are executed during the binary search of the data, plus one misprediction per node searched to represent the end of the search loop.

$$B_{btree} = h + \frac{\log_2(\sigma)}{2} \tag{16}$$

3.1.4 TLB Component

The fourth and final component of the model is an estimate of the number of misses in the TLB. The number of TLB misses depends on the number of unique physical memory pages that are accessed during the search. Index nodes are contained in physical pages of memory, where each page may hold one or more nodes. In traversing the index, each node's physical memory address must be translated from the virtual memory address. If an index is large, then the TLB may not be able to cache all the address translations for the index, and expensive TLB cache misses will be incurred during an index search.

The TLB is typically much smaller than the level 2 cache, e.g. the data TLB on the Pentium III has only 64 entries, so capacity misses must be taken into account. Since each query accesses exactly one page at each level of the index, the TLB entries can be viewed as a number of smaller independent caches, each with a cache of tlb_cap/h entries. This model assumes a fully associative TLB cache using an LRU replacement policy. The probability of experiencing a TLB miss can then be calculated as the probability

of not hitting the same page in tlb_cap/h attempts.

$$p_i = \left\lceil \frac{node\_sz}{page\_sz} * \lambda_i \right\rceil \quad (17)$$

$$T_{btree} = h - \sum_{i=1}^{h} \left(1 - \frac{1}{p_i}\right)^{\frac{tlb\_cap}{h}} \quad (18)$$

Equation 18 calculates the average number of TLB misses per query by subtracting the average number of pages that are found in the TLB from the number of TLB requests. The probability of a page-table entry being found in the TLB is dependent on the number of memory pages at each level, p_i. In deriving Equation 18, the height of the index is assumed to be less than the number of TLB entries, and the number of queries is much greater than the number of TLB entries. The model may underestimate the actual number of TLB misses because the operating system may flush the TLB on page table changes and context switches.

3.1.5 Overall Cost

The overall cost of an index scan is calculated by substituting M_{btree}, I_{btree}, B_{btree}, and T_{btree} for M, I, B, and T respectively in Equation 1. We use M_{btree} rather than M_{spec}, since the speculative cache misses will not incur the entire cache miss latency.

Using the four component models, as well as the execution model, the effect of the node size on the performance of equality search can now be analyzed.

3.1.6 Analysis of Node Size Effects on Equality Search

Using the model presented in the previous section, the effects of node size of the performance of equality searches can now be determined. For this analysis, the underlying architecture is assumed to be an Intel Pentium III (Table 2), which is the same as our experimental machine; the cardinality of the index is set to 10 million, and the number of search queries to 10 thousand.

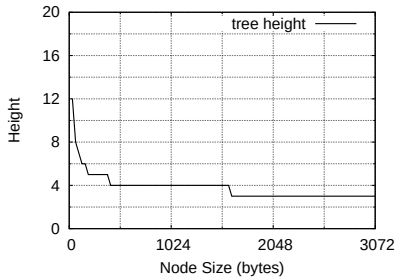

Figure 2: Index Height

Figure 2 shows the height of the CSB$^+$-tree as the node size increases. As expected, a small node size creates an index with a large height, and as the node size grows larger than a few cache lines in size, the height of the index decreases dramatically.

Figure 3 shows the average number of cache misses incurred during an equality search for various node sizes. In this figure, we plot the average number of cache-miss events with and without the misses incurred due to speculative execution. In the figure, the plot labeled *non-speculative* corresponds to Eq. 10, and the plot labeled speculative corresponds to Eq. 14. From Figure 3, we observe that the CSB$^+$-tree shows its best cache behavior at small node sizes, with a node size equal to the cache line size (32 bytes) experiencing the minimum number of cache misses. At a node size of 32 bytes, the leaf node can only hold a maximum of two entries, with four bytes remaining unused. Because of the small node size, the tree is high. At a node size of 64 bytes, a leaf node can contain a maximum of six entries, sharply decreasing the height of the tree but increasing the number of cache misses incurred to traverse the tree. As the node size increases, cache performance begins to suffer as the binary search inside of the node causes poor cache line utilization.

Figure 4 shows the average number of instructions executed during an equality search, for various node sizes (Eq. 15). For node sizes in the range of 32–96 bytes, there is a large traversal cost due to the height of the tree. As the size of the node increases, the traversal cost decreases rapidly. Consequently, for the larger node sizes, there are fewer instructions executed per query.

The number of branch mispredictions (Eq. 16), presented in Figure 5, follows a trend similar to the number of instructions executed. As the node size increases, the height of the tree decreases, and the mispredictions that occur per node access is reduced. The cost of the mispredictions contributes approximately 14–18% to the overall execution time, over the entire range of node sizes.

The number of TLB misses (Eq. 18) in Figure 6 also follows the height of the tree. The number of TLB misses is highest at small node sizes because of the large number of unique pages that are accessed. At larger node sizes, there are fewer TLB misses because more work is being performed per page access. An index constructed with a node size of 512 bytes exhibits 67% fewer TLB misses than at a node size of 32 bytes.

Figure 7 shows the effect of the node size on the overall execution time of equality search on the CSB$^+$-tree. This graph is plotted using Equations 1, 10, 15, 16, and 18. As the node size increases, the cache miss latency is contributing more to the overall execution time while the instructions, branch mispredictions, and TLB miss latency are contributing much less to the overall execution time; therefore, while the cache performance may be optimal for small node sizes, the execution time is adversely affected by the high instruction count, branch mispredictions, and TLB misses. From the figures, the minimum number of cache misses occurs at a node size of 32 bytes, and the minimum number of instructions executed occurs at sizes of 1632 bytes and larger. The number of branch mispredictions and the number of TLB misses continue to decline as the node size increases, since the tree is becoming more shallow. The resulting execution time is poor for small node sizes, quickly improves as the node size approaches 512 bytes, and then remains fairly constant for node sizes larger than 512 bytes. *The predicted optimal node size is 1696 bytes, performing 27% faster over a node size of 32 bytes, which is the L2 cache line size for the Intel Pentium III, and the current conventional choice for a node size.*

3.2 Analytical Model for Space

The total amount of space required to construct a CSB$^+$-tree is calculated by summing the number of nodes at each level, and then multiplying the sum by the node size, as shown in Equation 19.

$$space = node\_sz \sum_{i=1}^{h} \lambda_i \quad (19)$$

In Equation 19, $node\_sz$ is the node size, and λ_i is the number of nodes at each level of the tree. While $space$ accounts for the memory allocated to nodes that contain actual entries, a *full* CSB$^+$-tree allocates memory for the entire *group* of nodes, regardless of the parent node's fill factor. The space required for a full CSB$^+$-tree

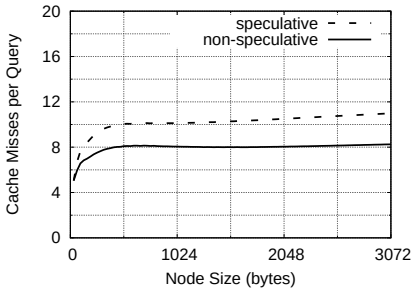

Figure 3: Cache Misses

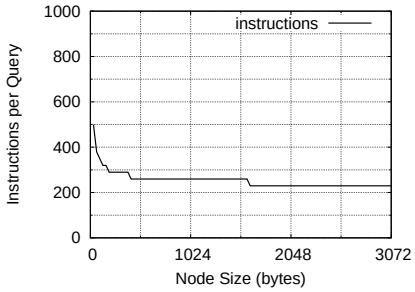

Figure 4: Instructions Executed

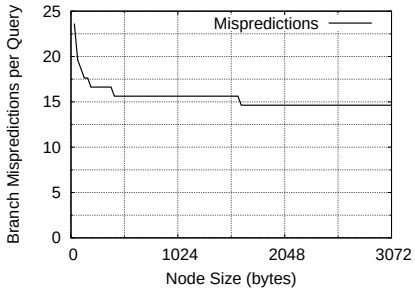

Figure 5: Branch Mispredictions

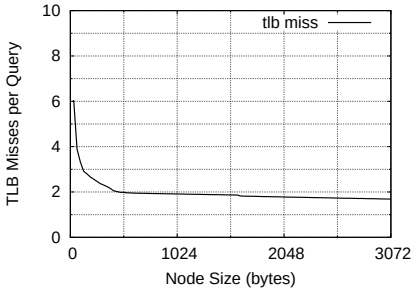

Figure 6: TLB Misses

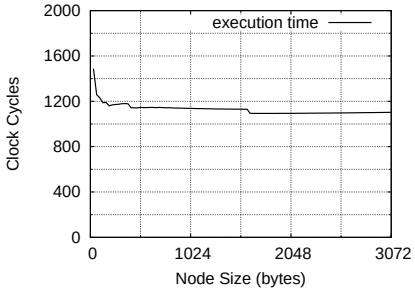

Figure 7: Execution Time

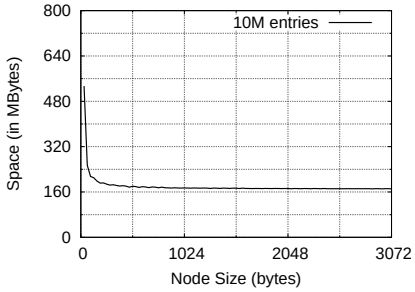

Figure 8: Memory Required

is shown in Equation 21.

$$bf_{max} = 2 * \left\lfloor \frac{node\_sz - nlmeta\_sz}{2 * key\_sz} \right\rfloor + 1 \qquad (20)$$

$$space_{full} = \frac{bf_{max}}{bf} * space \qquad (21)$$

Equation 21 uses the ratio of $bf_{max} : bf$ to relate the number of nodes allocated in a group to the number that actually contain entries, where bf_{max} is the maximum branching factor of a node.

Figure 8 plots the space required for a "full" CSB$^+$-tree to store 10 million entries. From the figure, node sizes larger than a cache line size create trees that require much less space. *A node size of 512 bytes requires 66% less space than a node size of 32 bytes, and 29% less space than a node size of 64 bytes.*

3.3 Analytical Model Application

We have applied the performance model to analyze the effects of constructing the CSB$^+$-tree with larger keys, and to analyze the performance effects of using a simple concurrency control protocol with the index. In the interest of space, only a summary of these results is presented here, and we refer the interested reader to the full-length version of this paper [19].

To model the search performance when indexing keys larger than four bytes (such as doubles), we simply modify the parameter key_sz. Analysis of the model demonstrates that, with larger keys, the conventional choices are even further off from the optimal performance point, and the benefits of using larger node sizes are even greater.

The model has also been applied to examine the effects of a simple concurrency control protocol. As shown in [19], the overall effect on the model is similar to increasing the overhead per node, I_{trav}. Consequently, the performance benefits of using larger node sizes continue to hold even when a concurrency control protocol is employed.

4. EXPERIMENTAL EVALUATION

In this section, we present experimental results based on an implementation of the CSB$^+$-tree in a main memory database system, named *Quickstep*, that we are currently building.

4.1 Index Implementation Details

Since the index was implemented in an actual database system, system abstractions may have induced additional performance overhead in some of the index operations. For example, our system allocates memory in pages that can then be saved to disk, and we utilize a buffer manager to manage page caching in main memory. The Quickstep database buffer manager allows pinning entire relations or indices in main memory (if there is enough memory available), and maps the entire disk image to a contiguous space in virtual memory. In this mode, all of the data is pinned in the main memory, and disk pointers are *swizzled* to direct memory pointers; thereby, reducing node traversal overheads. In the experiments presented in this section, the entire data set is pinned in main memory. The index and database system were coded in C++, and the system was compiled with the GNU gcc compiler with all optimizations turned on.

The *full* CSB$^+$-tree, as described in [27], was implemented. All nodes were allocated in pages of memory, with groups of nodes allocated on contiguous pages of memory if necessary. All keys are four bytes, and all $\langle key, RID \rangle$ entries are eight bytes.

4.2 Experimental Setup

The experiments were performed on a 600MHz Intel Pentium III with 768MB of main memory. The Pentium III has a two level cache hierarchy. The first level consists of a 16KB data cache and a 16KB instruction cache. The second level cache is a 512KB unified data/instruction cache. All caches are 4-way, set associative, with a 32 byte line size. The Pentium III also has two TLBs, one for code pages (i-TLB) and one for data pages (d-TLB). The i-TLB can hold 32 translation entries, while the d-TLB can hold 64

entries. Both TLBs are 4-way set associative. The Pentium III provides two hardware counters for measuring processor events, such as the number of cache misses and the number of instructions executed. The operating system used on this machine was Debian Linux, kernel version 2.4.18. To access the event counters on the processor, we used the PAPI library [7].

We also verified these results on a 450MHz Sun UltraSPARC-II with 1024MB of main memory, running SunOS version 5.8. The cache line size for the UltraSPARC-II is twice the line size used in the Pentium III processor. When comparing the relative performance improvements over the "default" case of using a node size equal to the cache line size, the relative performance improvements are roughly half of that observed for the Pentium III. In the interest of space, we only present the experimental results using the Pentium III here, and refer the interested reader to [19].

The events measured include execution time, L2 cache misses, instructions executed, L1 instruction cache misses, and branch mispredictions. The L1 instruction cache misses are an insignificant percentage of overall execution time (less than 0.5%), and are not discussed in the analysis of the experiments. The Pentium III does not expose the data TLB misses as measurable events. To measure the number of TLB misses, we wrote a TLB simulator that estimated the number of TLB misses from a trace of the memory references. The simulator modeled the capacity and associativity of the Pentium III d-TLB, and assumed an LRU replacement policy.

Concurrency control, record retrieval, and output operations are not included in our measurements of the index performance. Record retrieval and output costs remain constant for the range of index node sizes, while the effects of concurrency control have been examined separately, and can be found in the full version of this paper [19].

4.3 Data Set and Queries

For the data set, we used the Wisconsin Benchmark's [4, 14] TENK relation scaled to ten million entries, and indexed on the *unique1* attribute, which is a candidate key in this relation. Unless stated otherwise, the *unique1* values are inserted into the index in unsorted order.

In each of the experiments reported below, the index is queried ten thousand times. Each query is an equality search for a key value that is randomly selected from the ten million possible entries. The measured events are divided by the total number of queries to calculate an average event per query, so each data point in a graph is an average of ten thousand queries. Experiments with much larger numbers of queries, and relations with different cardinalities were also analyzed. The per query performance results presented here remained essentially the same; for brevity, we omit these results, and refer the interested reader to [19].

4.4 Measuring Performance

In analyzing the experiments, we will frequently refer to the optimal range of node sizes for a given performance metric; this range is defined as the node sizes that result in performance that is within 5% of optimal for the particular metric. To quantify performance improvement, we use the percentage improvement metric given by:

$$\frac{performance\,before - performance\,after}{performance\,before}\%$$

A second frequently used metric is *speedup*, given by:

$$\frac{performance\,before}{performance\,after}$$

When reporting the performance results, the performance improvement will be presented first, with the speedup presented second and in parenthesis.

4.5 Experiment 1: Equality Search Performance

To analyze the effect of node size on the search performance of the index, we first present an analysis of equality searches over a wide range of node sizes. In this first experiment, the CSB$^+$-tree index was constructed on the *unique1* attribute in the scaled TENK relation (cardinality 10M).

Figure 9 shows the average number of cache misses per query for the CSB$^+$-tree index, including speculative cache misses. From the figure, we observe that the CSB$^+$-tree experiences the fewest cache misses for the node sizes ranging from 32–64 bytes. As expected, cache misses are minimized at small node sizes. However, as the node sizes become larger, the binary search within each node increasingly contributes to the number of cache misses.

Figure 10 shows the average number of instructions per query for the index. As the node size increases for the CSB$^+$-tree, the height of the tree decreases, causing the instruction count to drop. The jumps that occur in Figure 10 correspond to changes in the height of the index structure. The instruction count is at its minimum within the range of 1408–3072 bytes.

The number of branch mispredictions, shown in Figure 11, is fairly constant for node sizes of 512 bytes and greater. For much smaller node sizes, the number of mispredictions increase due to the increased number of node traversals that occur.

Figure 12 shows the number of TLB misses per query. As discussed in Section 4.2, the Pentium III does not expose the data TLB misses, so this figure is derived using a simulation. As expected, the number of TLB misses varies with the height of the index. A node size of 512 bytes exhibits 73% fewer TLB misses than at a node size of 32 bytes, and node sizes larger than 512 bytes incur up to 80% fewer TLB misses than at a node size of 32 bytes.

Figure 13 shows the execution time, measured as clock cycles per query. As the figure shows, the CSB$^+$-tree's best performance occurs at node sizes greater than **160 bytes**. Node sizes in this range are much larger than the cache line size, which is 32 bytes. Using a node size within the range 256–512 bytes can improve the performance of the index by 17% (1.21 speedup) over the performance when the node size is 32 bytes. After a node size of 512 bytes, the performance of the index very gradually improves. Using a node size in the range of 1280–3072 bytes can improve the performance of the index up to **19% (1.24 speedup)** over the performance when the node size is 32 bytes.

Comparing Figures 9 – 13 from the experimental evaluation to Figures 3 – 7 from the analytical model, we can see that the analytical model accurately predicts the actual empirical behavior. As stated before, the experiments were performed in the context of a database system, and consequently include additional performance overhead that is not predicted by the analytical model. For example, before evaluating the index search operation, the query must be parsed, which requires additional instructions, and also incurs one data cache miss to read the literal in the query predicate. This overhead is reflected as a constant value added to the actual number of cache misses incurred (Fig. 9) and the actual number instructions executed (Fig. 10), and subsequently affects the overall execution time (Fig. 13).

4.6 Experiment 2: Space Requirements

In this experiment, we investigate the effect of the node size on the space required to store the CSB$^+$-tree index. The CSB$^+$-tree

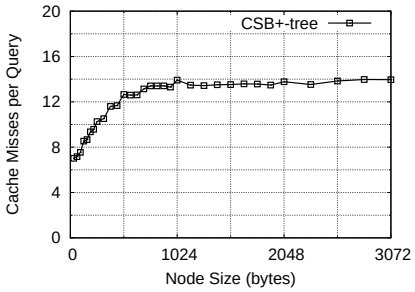

Figure 9: Cache Misses

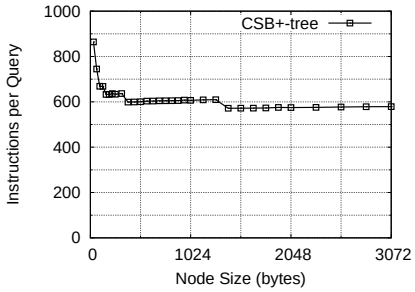

Figure 10: Instructions Executed

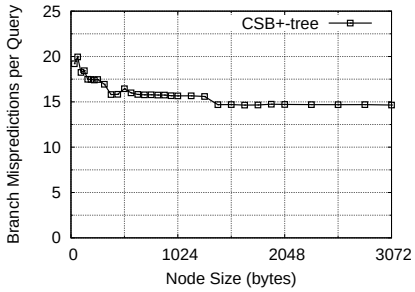

Figure 11: Branch Mispredictions

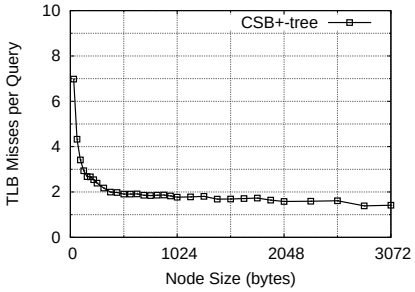

Figure 12: TLB Misses (Simulated)

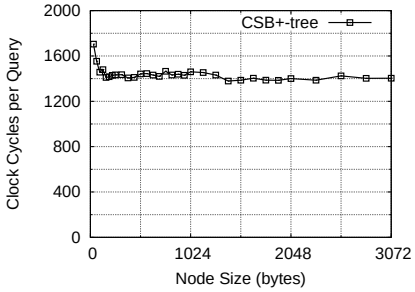

Figure 13: Execution Time

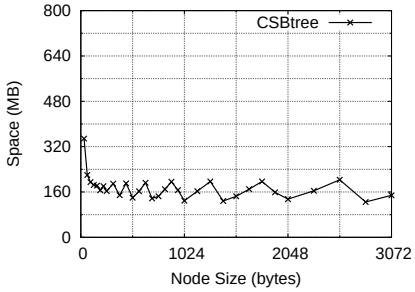

Figure 14: Space Requirements

implementation was the *full* CSB$^+$-tree, where space is allocated for an entire group of nodes regardless of whether each node contains entries. As in the previous experiment, we used the TENK relation scaled to ten million entries, and indexed on the *unique1* attribute. The space requirements of the CSB$^+$-tree versus node size in Figure 14 highlights an important problem with using small node sizes for memory constrained environments: small node sizes result in very deep trees, requiring substantially more space to represent. From the figure, a node size of 512 bytes requires 60% less space than a node size of 32 bytes, and requires 36% less space than a node size of 64 bytes. Node sizes larger than 512 bytes show moderate improvements in space utilization, requiring between 42–64% less space than a node size of 32 bytes. The saw-tooth shape in Figure 14 is due to small changes in the fill factor of the nodes in the index. This fill factor can vary from 0.5 to 1, but the analytical model does not exhibit this saw-tooth form because the fill factor is modeled as a constant of 0.67, which is the average fill factor that is typically observed in practice [31]. While the fill factor varies in the experimental evaluation, the space requirements fluctuate around 163MB, which compares favorably to the 171MB that is predicted by the analytical model.

4.7 Additional Experiments

We have performed several additional experiments to evaluate the effect of node size under a variety of different conditions. In the interest of space, we only summarize these results, and refer the interested reader to [19].

First, we examined the performance of range searches for a variety of node sizes. The results show that for the Pentium III, using a node size larger than 512 bytes improves the range search performance by up to 48% (1.94 speedup) over a CSB$^+$-tree constructed with the conventional node size.

Second, we examined the performance of equality search on a CSB$^+$-tree constructed with keys of size eight bytes (the key size for a double precision number). The CSB$^+$-tree experienced its optimal performance with node sizes larger than 1024 bytes, performing up to 26% (1.34 speedup) faster than at a node size of 64 bytes. These results demonstrate that, with larger keys, the conventional choice in node size is even further from the optimal performance point, and the benefits of using larger node sizes are even greater.

Third, we evaluated the impact of concurrency control on the equality search performance of the CSB$^+$-tree. This experiment shows that the optimal range of node sizes remains 160-bytes and larger, and results in a performance improvement of 17% (1.21 speedup) over a node size of 32 bytes.

Fourth, the performance of the CSB$^+$-tree index containing duplicate keys was examined. For this experiment, the index was loaded with ten million integers which were drawn from a set of five-hundred thousand distinct integer values. These values followed a Zipfian distribution where the *skew* parameter varied between 0 and 1. This experiment shows that, when the index contains skewed data, node sizes much larger than the cache line size performed optimally. Specifically, node sizes in the range of 256–512 bytes performed up to 49% (1.95 speedup) faster than a node size equal to the cache line size of 32 bytes. Node sizes larger than 512 bytes perform up to 50% (2.00 speedup) faster than a node size equal to 32 bytes.

Finally, we evaluated the effect of node size on the performance of the CSB$^+$-tree when inserting ten million unique, unsorted integers into an empty index. As the node size increased, the execution time also increased due to the larger cost in splitting nodes. Smaller node sizes performed much better than larger node sizes. A CSB$^+$-tree with a node size of 512 bytes performed 27% (21% slowdown) slower than an index with a node size of 32 bytes. Node sizes larger than 512 bytes performed increasingly worse. While node sizes much larger than a cache line size perform worse than small node sizes, workloads that have many more search operations relative to inserts (the common case in practice) will not be greatly affected by this poor insert performance.

4.8 Discussion

In summary, we have experimentally demonstrated that the first-order models presented in Section 3 are accurate. The experimental data shows that node sizes larger than 512 bytes perform up to 17% (1.21 speedup) faster for equality searches, as compared to the performance of an index with a node size equal to the cache line size. In addition, we have shown that the conventional node size demonstrates poor space utilization, consuming up to 149% more space than a node size of 512 bytes.

5. RELATED WORK

A number of previous studies have identified the critical influence that processor cache misses have on the performance of modern database systems, which typically run on servers with large main-memory configurations [2, 23, 25, 28]. To remedy this performance bottleneck, a number of popular database algorithms and access methods have been adapted to improve their cache utilization [6, 17, 25, 28]. Main-memory database systems have also paid considerable attention to efficient indexing techniques. The earliest work in this area by Lehman and Carey [22] investigates various hash-based and tree-based indexing structures for main memory databases. They also propose a new indexing structure called the T-tree that is shown to be very effective in main-memory environments. The paper did not consider cache behavior, primarily because processors at that time did not have sophisticated processor caches.

Rao and Ross recently rekindled interest in main-memory indices by considering the performance impact of cache misses. In [26], they investigate main memory indexing techniques for static data, proposing the Cache Sensitive Search tree, or CSS-tree, to improve the processor cache utilization during a search. In the analysis of the CSS-tree, the authors conclude that a node size equal to a cache line size is optimal in most cases. A limitation of the CSS-tree is that it is a static index structure, and must be entirely rebuilt upon *any* updates to the data. The authors also investigated dynamic indexing techniques in the main-memory environment in [27], proposing a cache-conscious variation of the traditional B+-tree, called the CSB$^+$-tree. The CSB$^+$-tree eliminates child node pointers in the non-leaf nodes, allowing additional keys to be stored in a node which improves cache line utilization. Analogous to the traditional B+-tree where a node size is equal to a disk page to minimize the number of page accesses during a search, the node size for the CSB$^+$-tree is set equal to a processor cache line to minimize the number of cache misses. The work by Rao and Ross has been extended in recent years in a number of different ways, including handling variable key length attributes efficiently [5] and for architectures that support prefetching [10].

In a recent paper, Chen, Gibbons and Mowry [10] examined the cache behavior of B+trees and CSB$^+$-trees. They conclude that the CSB$^+$-trees produce very deep trees which cause many cache misses as a search traverses down the tree. They propose a prefetching-based solution, in which the node size of a B+tree is larger than the cache line size, and special prefetching instructions are manually inserted into the B+tree code to prefetch cache lines and avoid stalling the processor. We also recommend larger node sizes for the CSB$^+$-tree, but our recommendation is not based on using special hardware prefetch instructions. Rather, we recognize that node size influences a number of different factors besides cache misses, and that overall performance is improved by carefully considering the effect of node size on all these factors.

Chen, Gibbons, Mowry, and Valentin also propose a version of their prefetching B+-tree optimized for disk pages, called Fractal pB+-trees [11]. In this work, the authors show how the pB+-tree index can be efficiently constructed onto disk pages, which are generally much larger in size than the index node. This paper nicely demonstrates the practical implications of utilizing a cache-sensitive main-memory index in a disk-based environment.

Kim, Cha, and Kwon recently proposed a cache-conscious modification to the traditional R-Tree index, which is used for indexing spatial data types such as points and polygons. The indexing structure that they propose is called the CR-Tree [21], which improves the cache utilization of the traditional R-Tree by reducing the space required to store the keys. The authors show that node sizes larger than a cache line size generally outperform node sizes approaching the cache line size. Our work in this paper compliments their work on spatial indices by presenting an analytical model and experimental results for the single-dimensional CSB$^+$-tree.

Cha, Hwang, Kim, and Kwon have also analyzed the performance of the CSB$^+$-tree and the traditional B+-Tree when incorporating concurrency control logic for use in a shared memory multiprocessor system [9]. The authors found that a node size of twice the cache line size is optimal for the CSB$^+$-tree. The authors' experiments were conducted on a Sun Enterprise 5500 server with eight UltraSPARC-II processors, so twice the cache line size translated into a node size of 128 bytes. As the description of the experiment that supports this conclusion is sparse, it is unclear why their conclusions modestly contradict our findings that even larger node size result in better performance. Our work provides a more complete analytical and experimental evaluation of the CSB$^+$-tree on single processor systems, and demonstrates that node sizes larger than the cache line size are optimal for both equality searches and range searches.

Choosing an optimal node size for a B+-tree in a traditional disk-bound database system has been the focus of a paper by Lomet [24], which follows the work of Gray and Graefe [18]. Lomet shows that from the performance perspective, large page sizes for B+-trees are better because they amortize the cost of going to the disk and also produce shallower trees. Our analysis presents an important parallel from the perspective of the processor data cache misses.

Chilimbi et al. [12] examines how the compiler can change the layout of data structures to improve cache-behavior of the program. They demonstrate that such compiler optimization techniques can improve the performance of Microsoft's SQL Server by 1–2%. Since the proposed technique is a general purpose compiler technique, the authors do not consider changing the database implementation.

There has been work on modeling cache misses for sparse matrix operations [15] as well as general processor cache modeling equations [16]. This work provides a more complete model of the cache hardware, including modeling capacity and conflict misses in their workloads. Our motivation for using Cardenas's formula was to model the first order effects of our particular workload, which are the most significant factors for the CSB$^+$-tree.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we investigate the performance of a popular cache-sensitive B+-tree index, called the CSB$^+$-tree. For this investigation, we introduce a first-order analytical model for the index structure. From the analysis of the model, we demonstrate that using the common design heuristic of setting the node size equal to the cache line size for this index structure is often suboptimal. We show that cache misses, instruction count, and TLB misses must be balanced to achieve the optimal index performance. We also report results from extensive experimentation on the CSB$^+$-tree index structure.

The experiments show that for the CSB$^+$-tree, a node size of 512 bytes or larger performs well across a wide range of search queries.

The results that we present in the paper can be easily applied to main memory databases and even to traditional databases that use main memory type of index structures for search operations. In most well structured systems, node size is typically a constant in the code or a database configuration parameter, and changing the node size is fairly straightforward. Thus we expect that, for many systems, using the results of this paper is likely to be an easy way to improve index's search performance.

Since we have only experimentally validated our results for a couple of the more popular microprocessors, the results of this paper should be used cautiously when applying to implementations running on other processors. However, our conclusions are centered around the observation that, in modern processors, the overall performance of an index structure depends on the number of cache misses, the instructions that are executed, the number of mispredicted conditional branches, and the number of TLB misses. Our analytical model captures these characteristics using a simple model which can easily be adapted for other architectures.

This paper has only focused on the equality search operation, and in the future we plan on extending the analytical model to analyze range search operations. The complexity lies in modeling the number of cache misses incurred during the linear leaf-node scan. For example, if the number of cache lines accessed during the leaf-node scan approaches, or surpasses, the capacity of the processor cache, all non-leaf node cache lines will be flushed. The result is that every access will result in a cache miss, virtually eliminating cache line re-use. To illustrate this scenario, a 0.65% range search of ten million entries will flush a 512 KB L2 cache after a single query. For very small range selectivities, the previous performance model for equality searches may suffice. However, for range selectivities that result in small, but non-negligible, fractions of the processor cache being replaced, it may be necessary to incorporate a more complete cache model to accurately capture the performance of the workload, similar to the work of Doallo et al. [15] for sparse matrix operations

We also plan on using the model to investigate the effects of the processor architecture on the performance of the CSB$^+$-tree. This study will include varying such architectural parameters as the main-memory latency, cache-line size, page size, etc. Some preliminary results on the effects of varying the architectural parameters can be found in [19].

There is a rich literature base examining performance models for caches in SMP environments [1, 29, 30]. However, the main focus of these previous papers has been on the performance impact of various cache invalidation protocols. In some studies [29], the effect of cache block size is also considered. In our current work, we only consider single processor machines, and consequently do not need these performance models for SMPs. However, as part of our future work, we will consider the effect of node sizes on CSB$^+$-trees in SMP environments with a mixed workload of updates and read-only queries. The SMP cache models will be directly applicable in that investigation.

7. ACKNOWLEDGEMENTS

This research was supported by the National Science Foundation under grant IIS-0093059. We would like to thank Murali Annavaram and Steve Reinhardt for their valuable comments on earlier drafts of this paper.

8. REFERENCES

[1] ADVE, S. V., ADVE, V. S., HILL, M. D., AND VERNON, M. K. Comparison of Hardware and Software Cache Coherence Schemes. *ACM SIGARCH Computer Architecture News 19*, 3 (1991), 298–308.

[2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK* (September 1999), pp. 266–277.

[3] BERNSTEIN, P. A., BRODIE, M. L., CERI, S., DEWITT, D. J., FRANKLIN, M. J., GARCIA-MOLINA, H., GRAY, J., HELD, G., HELLERSTEIN, J. M., JAGADISH, H. V., LESK, M., MAIER, D., NAUGHTON, J. F., PIRAHESH, H., STONEBRAKER, M., AND ULLMAN, J. D. The Asilomar Report on Database Research. *SIGMOD Record 27*, 4 (1998), 74–80.

[4] BITTON, D., DEWITT, D. J., AND TURBYFILL, C. Benchmarking database systems a systematic approach. In *Proceedings of the 9th International Conference on Very Large Data Bases, Florence, Italy.* (1983), pp. 8–19.

[5] BOHANNON, P., MCILROY, P., AND RASTOGI, R. Main-Memory Index Structures with Fixed-Size Partial Keys. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA* (May 2001), pp. 163–174.

[6] BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK* (September 1999), pp. 54–65.

[7] BROWNE, S., DONGARRA, J., GARNER, N., HO, G., AND MUCCI, P. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications 14*, 3 (2000), 189–204.

[8] CARDENAS, A. Analysis and Performance of Inverted Data Base Structures. *Communications of the ACM 18*, 5 (May 1975), 253–264.

[9] CHA, S. K., HWANG, S., KIM, K., AND KWON, K. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy* (2001).

[10] CHEN, S., GIBBONS, P. B., AND MOWRY, T. C. Improving Index Performance through Prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA* (May 2001), pp. 235–246.

[11] CHEN, S., GIBBONS, P. B., MOWRY, T. C., AND VALENTIN, G. Fractal Prefetching B+trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, WI, USA* (May 2002), pp. 157–168.

[12] CHILIMBI, T., HILL, M. D., AND LARUS, J. R. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language*

Design and Implementation, Atlanta, Georgia, USA (May 1999), pp. 1–12.

[13] CHOU, H.-T., AND DEWITT, D. J. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, Sweden* (Aug. 1985), Morgan Kaufmann, pp. 127–141.

[14] DEWITT, D. J. The Wisconsin Benchmark: Past, Present, and Future. In *The Benchmark Handbook for Database and Transaction Systems* (1993), J. Gray, Ed., Morgan Kaufmann.

[15] DOALLO, R., FRAGUELA, B., AND ZAPATA, E. Direct Mapped Cache Performance Modeling for Sparse Matrix Operations. In *7th EUROMICRO Workshop on Parallel and Distributed Processing* (February 1999), pp. 331–338.

[16] GHOSH, S., MARTONOSI, M., AND MALIK, S. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the 11th International Conference on Supercomputing* (1997), ACM Press, pp. 317–324.

[17] GRAEFE, G., BUNKER, R., AND COOPER, S. Hash Joins and Hash Teams in Microsoft SQL Server. In *Proceedings of the 24th International Conference on Very Large Data Bases, New York City, New York, USA* (August 1998), pp. 86–97.

[18] GRAY, J., AND GRAEFE, G. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *SIGMOD Record 26*, 4 (1997).

[19] HANKINS, R. A., AND PATEL, J. M. Effect of Node Size on the Performance of Cache-Conscious Indices. *Extended Report, http://www.eecs.umich.edu/quickstep/publ/ccindices.pdf* (2003).

[20] HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1998.

[21] KIM, K., CHA, S. K., AND KWON, K. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA* (May 2001), pp. 139–150.

[22] LEHMAN, T. J., AND CAREY, M. J. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan* (August 1986), pp. 294–303.

[23] LO, J. L., BARROSO, L. A., EGGERS, S. J., GHARACHORLOO, K., LEVY, H. M., AND PAREKH, S. S. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, Barcelona, Spain* (1998), pp. 39–50.

[24] LOMET, D. B-tree Page Size When Caching is Considered. *SIGMOD Record 27*, 3 (1998).

[25] NYBERG, C., BARCLAY, T., CVETANOVIC, Z., GRAY, J., AND LOMET, D. B. AlphaSort: A Cache-Sensitive Parallel External Sort. *VLDB Journal 4*, 4 (1995), 603–627.

[26] RAO, J., AND ROSS, K. A. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK* (September 1999), pp. 78–89.

[27] RAO, J., AND ROSS, K. A. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA* (May 2000), pp. 475–486.

[28] SHATDAL, A., KANT, C., AND NAUGHTON, J. F. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile* (September 1994), pp. 510–521.

[29] VERNON, M., AND HOLLIDAY, M. A. Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets. *Performance Eval. Rev. (USA), Spec. Issue 14*, 1 (May 1986), 9–17.

[30] VERNON, M. K., JOG, R., AND SOHI, G. S. Performance Analysis of Hierarchical Cache-Consistent Multiprocessors. *Performance Evaluation 9*, 4 (1989), 287–302.

[31] YAO, A. On Random 2-3 Trees. *Acta Informatica 9* (1978), 159–170.

[32] YAO, S. Approximating Block Accesses in Database Organization. *Communications of the ACM 20*, 4 (Apr. 1977), 260–261.