Effect of Node Size on the Performance of Cache-Conscious Indices

Richard A. Hankins
Univ of Michigan
hankinsr@eecs.umich.edu

Jignesh M. Patel
Univ of Michigan
jignesh@eecs.umich.edu

Abstract

In main-memory environments, the number of processor cache misses has a critical impact on the performance of the system. Cache-conscious indices are designed to improve the performance of main-memory indices 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 instruction count, which play a significant role in determining the overall performance of the index. Using analytical models and a detailed experimental evaluation, we investigate the effect of the index's node size on two common cache-conscious indices: a cache-conscious B+-tree (CSB+-tree), and a cache-conscious extendible hash index. We show that using node sizes much larger than the cache line size can result in better search performance for the CSB+-tree. For the hash index, reducing the number of overflow chains is the key to improving search performance, even if it requires using a node size that is much larger than the cache line size. Extensive experimental evaluation demonstrates that these node size choices are valid for a variety of data distributions, for range searches on the CSB+-tree, and can also be used to speed up the execution of traditional hash-based query operators that use memory-resident indices internally.

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 database servers with large main memory configurations, data sets and entire databases primarily reside in main memory, and it is expected that in the future all but the largest data sets will be resident in main memory [2].

Traditional databases are designed to reduce the number of disk accesses primarily because 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 of accessing data from the main memory [1, 5, 23, 25, 27]. 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 cache 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 [1]. 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.

A frequently performed operation in database systems is evaluating equality-based searches. Given the importance of equality-based searches, a number of cache-conscious index structures have been proposed to speedup this operation [4, 20, 24, 25]. 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 [20] and chained bucket hashing [24]. 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 tree-based 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 hash-based 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 hash tables [14]. As we demonstrate, this choice is often suboptimal for cache-conscious access methods when running on modern processors.

In this paper, we analyze the effect of node size on the performance of two popular cache-conscious indices: a cache-conscious B$^+$-tree (CSB$^+$-tree) [25] and a modified extendible hash index. This paper makes the following contributions:

- Using analytical models, 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. For the CSB$^+$-tree, we show that changing the node size can have contrary effects on the number of cache misses and the instruction counts.

- This paper provides a detailed experimental study examining the effect of node size on the performance of the two index structures. We show that when executing equality and range searches on the CSB$^+$-tree, a node size of 512 bytes or larger generally results in better performance. Compared to the conventional node size, a larger node size can improve the performance of the CSB$^+$-tree by **19%** (24% speedup) for equality searches, and **48%** (94% speedup) for range searches on the Intel Pentium III.[1] We also show that these results hold when indexing attribute values that have a skewed distribution.

  For the hash index, our experiments demonstrate that the number of buckets in the overflow chains has a critical impact on the performance of the index. The index search performance improves significantly as the number of buckets in the overflow chains decreases. The index performs best when the node size is such that there are no overflow chains, regardless of the cache line size. For the extendible hash index with a moderate directory overhead (around 5%), a node size of a few hundred bytes improves the performance of the index by **42%** (72% speedup) over an index that uses a conventional node size.

- We also evaluate the effect of node size on the space required to store the index in memory. We show that for the CSB$^+$-tree, a large node size is not only more time efficient but it is also more space efficient.

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

For the hash index, a larger directory generally results in better performance as it reduces the length of overflow chains, which critically impacts the performance of the index. In fact, if the distribution of the input data is known, and if there are no limits on the amount of memory that can be used for storing the index, the optimal node size is a cache line size with a directory size that produces no overflow chains. Because this criteria could result in prohibitively large directory sizes, it may be desirable to put an upper limit on the size of the hash directory. We show that for a given hash directory size, a good choice for the bucket size is one that minimizes the number of overflow chains, even if this requires using a bucket size that is significantly larger than the cache line size.

- Finally, we experimentally evaluate the choice of node size for the hash index when used in hash-based query operations, such as a join operation. We show that our criteria for node sizes can also be used to improve the performance of these operations.

The remainder of this paper is organized as follows: Section 2 briefly describes the CSB$^+$-tree and the extendible hash index. Section 3 describes the analytical models for both indices. Section 4 presents the experimental results of operations on the two indices, and Section 5 discusses related work. Finally, we present our conclusions and directions for future work in Section 6.

2 Index Structure Descriptions

CSB$^+$-tree

A CSB$^+$-tree is an adaptation of the ubiquitous B$^+$-tree for main memory databases [25]. In a CSB$^+$-tree, a non-leaf node contains a single pointer and a list of keys, instead of $\langle key, childpointer \rangle$ pairs. The single pointer references a group of children, where the number of children in the group is equal to the number of keys plus one. A child node is referenced by adding an offset, based on a key's position, to the group pointer. By eliminating the child node pointers in the non-leaf nodes, additional keys can be stored resulting in more efficient use of a cache line. We use the CSB$^+$-tree in this study because it has been shown to outperform other tree-based cache-conscious indices and traditional indices in memory-resident databases [25].

Extendible Hashing

Of the various dynamic hash indexing structures that have been proposed, extendible hashing has been shown to outperform other indexing structures in main memory databases [20]. Consequently, we chose a cache-sensitive version of the extendible hash index. A traditional extendible hash index consists of an expandable directory of buckets, with each bucket containing a $\langle key, recordID \rangle$ pair. An index into the bucket directory is calculated by applying a hash function on the search key. The directory entry points to a chain of buckets that contain the $\langle key, recordID \rangle$ pairs. On insertion, if the desired bucket is full, the bucket directory is expanded, and the entries in the full bucket are re-distributed. For duplicates, overflow buckets are created to hold the entries.

We have made two modifications to the traditional extendible hash index. First, we have imposed an upper bound on the size of the directory. After the directory reaches this upper bound, we no longer double the directory on bucket overflows. Such limits are commonly used in practice since without such limits the space required for the directory structure can be prohibitively large [20, 24, 29].

The second modification is that the entries are kept in sorted order within a bucket, which allows using a binary search within a node. We assume that searching is much more frequent than inserts or deletes, so this modification favors the searches at the expense of updates. (An evaluation of the performance benefits of this second modification is presented in [16].) To simplify the description of both index structures, we will use the words buckets and nodes interchangeably to refer to the buckets in the hash index.

Variable	Description	Value
Common Parameters		
cpi	processor clock cycles per instruction executed	0.63 (Pentium III)
$miss\_latency$	processor clock cycles per L2 cache miss	78 (Pentium III)
$pred\_penalty$	processor clock cycles to correct a mis-predicted branch	15 (Pentium III)
ϵ	max. number of entries in a node	varies
f	fill percentage of a node	varies
l	number of cache lines in a node	varies
σ	cardinality of the index	varies
q	number of queries	varies
CSB$^+$-tree Parameters		
bf	branching factor	varies
h	height of the CSB$^+$-tree	varies
I_{search}	instructions to compare a key and select the next position in the binary search	5
I_{trav}	instructions per node traversal	30
Hash Index Parameters		
C	average overflow chain length	varies
I_{dir}	instructions per directory access	20
I_{search}	instructions to compare a key and select the next position in the binary search	5
I_{trav}	instructions per bucket traversal	30

Table 1: Model Parameters

3 Index Structure Analysis

In this section, analytical models of the two index structures are used to examine the effect of node size on the performance of equality searches. This section is arranged as follows. We first present a simple execution time model based on the mix of instructions executed, the number of data cache misses, and the number of mis-predicted branches that occur in an operation. We then model the instructions executed, cache misses, and branch mis-prediction events that occur during an equality search operation on both the CSB$^+$-tree and the hash indexes. Using the event models and the execution time model, we analyze the performance of both indexes as the node size varies. In addition, we use the CSB$^+$-tree model to examine the additional effects of concurrency control logic on the performance of the index.

The following analysis relies on the parameter definitions shown in Table 1. The values for the parameters shown in Table 1 are *estimates* based upon our implementation of the indices. These values may change across different processors and implementations. The analytical model that we develop here is not dependent on particular values, and in fact can be used to study the performance effects of modifying these parameter values.

3.1 Execution Time Model

We model the cost of executing an index search as a function of three variables: the instruction count (I), the number of data cache misses (M), and the number of branch mis-predictions (B). There are additional factors that contribute to the total execution time, including instruction cache misses [1]. Based on a profile of our system, the instruction cache misses contribute less than 0.5% to the overall execution time, and are ignored in the analysis that follows. The execution time model is shown in Equation 1:

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

```
/* Event Loop */
for (i = 0; i < Queries; i++) {
    address = origin + random offset
    /* De-reference address, generating a cache miss */
    val = *address
    for (j = 0; j < Instructions; j++) {
        /* computation involving "val" */
    }
}
```

Figure 1: Profile Code

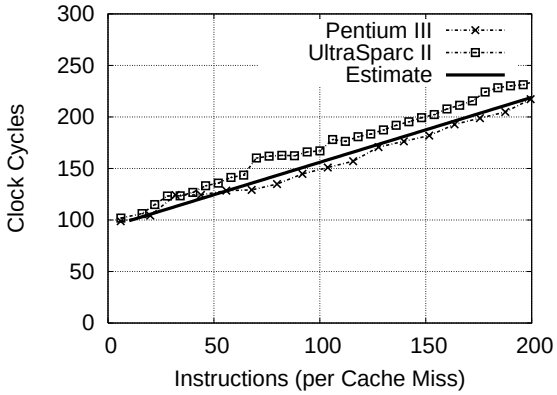

Figure 2: Execution Time Profile

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, and $pred\_penalty$ is the cost of incorrectly predicting a branch path. The cpi, $miss\_latency$, and $pred\_penalty$ costs are in processor clock cycles.

Estimates for the cpi, $miss\_latency$, and $pred\_penalty$ parameters can usually be extracted from a processor's design manual. For example, the Intel Pentium III can retire up to three instructions per clock cycle, an L2 cache miss requires approximately 11–14 memory bus cycles[2] to satisfy, and the cost of mis-predicting a branch is about 10–15 clock cycles [18]. Modern processors are very complex, so these estimates do not necessarily reflect the actual time for retiring instructions and satisfying a cache miss. For example, modern processors can execute instructions out of order, and efficiently process multiple outstanding cache requests. At the same time, some instructions require more time to retire than others, and stalls on cache misses can cause the instruction pipeline to flush, further delaying the retirement of instructions. Thus calculating the cpi is particularly hard since, for the Intel Pentium III, it can range from a best case value of 0.33 cycles per instruction to the worst case value of 14, or more, cycles per instruction (the number of stages in the instruction pipeline for Pentium III is 14, leading to the "theoritical" worst case).

To more accurately estimate these parameters, we employ a simple experiment. Our goal in this experiment is to obtain more accurate values for the cpi and $miss\_latency$ parameters when executing code similar to an index search operation. To simplify the experiment, we assume that the branch mis-prediction penalty reported in the processor specifications is fairly accurate. This assumption is reasonable as the branch mis-prediction penalty is mostly a result of flushing the instruction pipeline inside the processor, and unlike the other two parameters, the manual provides tight upper and lower bounds on the branch miss prediction costs.

For the experiment, we constructed a loop that reads four-byte values from a randomly selected address within a large memory block, ensuring that the selected address is not being reused. In this way, we generate one cache miss per read operation. After a single read operation, we execute a variable number of instructions, thereby gradually increasing the number of instructions per cache miss. In addition, a branch mis-prediction event is generated at the termination of the variable instruction loop. This series of operations roughly simulates the operations involved in an index search. The pseudo-code for this experiment is shown in Figure 1.

The result of the experiment performed on an Intel Pentium III processor and a Sun UltraSPARC-II processor is shown in Figure 2. Using the execution profile of a particular processor, the cpi and $miss\_latency$ parameters can be calculated by performing a least squares fit of the execution time equation in Equation 1.

[2]The memory bus runs at 100MHz, so each memory bus clock cycle is actually six clock cycles for a 600MHz processor. This translates to an L2 cache miss latency of approximately 66-84 processor clock cycles.

For example, based on the execution time profile of the Intel Pentium III, the cpi parameter is 0.63 cycles per instruction and the $miss\_latency$ parameter is 78 cycles per cache miss, assuming a branch mis-prediction penalty of 15 clock cycles. Function 1 with the more accurate parameter values is also plotted in Figure 2.

To reiterate, extracting the parameters from the processor profile is simply a more accurate estimate of how the processor acts under these types of workloads. These parameters can be taken from the processor manual, but the performance model will be less accurate with respect to the actual implementation.

3.2 CSB$^+$-tree

In this section, we present an analytical model of the search operation on the CSB$^+$-tree. We first present a model to compute the average number of cache misses, followed by a model to compute the average number of instructions that are executed.

During a 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 a binary key-search, m_{node}, can be computed as:

$$m_{node} = \lceil \log_2(l+1) \rceil . \tag{2}$$

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

$$h = \left\lceil \log_{bf} \left(\frac{\sigma}{f * \epsilon} \right) + 1 \right\rceil , \tag{3}$$

where the maximum number of keys in a node, ϵ, can be computed by dividing the size of a node by the size of an index key. A CSB$^+$-tree node stores a child pointer, a count, and a number of key values, each of which is four bytes in our implementation.

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 the effect of cached node data, we apply Cardenas's formula [7, 28] *at each level* of the tree, recognizing that data at each level of the tree has a non-uniform probability of being cached[3]. It is important to note that the processor cache can sometimes be flushed by the operating system, 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. Consequently, many queries are satisfied between cache flushes.

Cardenas's formula, shown in Equation 4, 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 cache lines for a given level of the tree.

$$X_D(\lambda, q) = \lambda * (1 - (1 - 1/\lambda)^q) \tag{4}$$

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.

$$m_{btree} = \frac{\sum_{i=1}^{h} X_D(\lambda_i, q * m_{node})}{q} \tag{5}$$

[3]We use Cardenas's formula instead of Yao's formula because we are interested in modeling queries with replacement.

In this equation, λ_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:

$$\lambda_i = \begin{cases} bf_{root} * bf^{i-2} * l & \text{if } i > 1 \\ l & \text{if } i = 1 \text{ (root)} \end{cases} \quad (6)$$

$$bf_{root} = \frac{L}{bf^{h-2}} \quad (7)$$

$$bf = \epsilon * f + 1 \quad (8)$$

In the above equation, 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 5 accounts for compulsory misses, but does not incorporate conflict misses or capacity misses in the cache [17]. For a first-order approximation, we assume a large cache where these effects are not significant.

The number of instructions executed at each level includes the binary key-search, plus the cost of a child-node traversal. Equation 9 predicts the number of instructions executed.

$$i_{btree} = h * I_{search} * \log_2(f * \epsilon + 1) + h * I_{trav}, \quad (9)$$

In Equation 9, 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 accessing the child node. The values for these parameters are estimates of the actual implementation, and are given in Table 1.

The number of mis-predicted branches is proportional to the number of binary search operations executed. 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. Equation 10 estimates the number of mis-predicted branches.

$$b_{btree} = \frac{h * \log_2(f * \epsilon + 1)}{2} \quad (10)$$

In Equation 10, the number of mis-predicted branches is 50% of the branches taken in the binary search of each node. The overall cost of an index scan is calculated by substituting m_{btree}, i_{btree}, and b_{btree} for M, I, and B respectively in Equation 1.

3.2.1 Analysis

Figure 3 shows the average number of cache misses (Eq. 5) incurred during an equality search for various node sizes. From Figure 3, we observe that the CSB$^+$-tree shows the best cache utilization at small node sizes, with the node size range of 64–256 bytes incurring within 5% of the minimum number of cache misses. At a node size of 32 bytes, which is the L2 cache line size for the Pentium III, 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, and the cache utilization is poor. At a node size of 64 bytes, which is the L2 cache line size for the UltraSPARC-II, a leaf node can contain a maximum of six entries, sharply decreasing the height of the tree and significantly improving the number of cache misses incurred in 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.

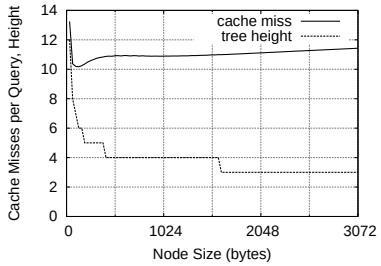

Figure 3: CSB$^+$-tree, Cache Misses per Query ($q = 10,000, f = 0.67$)

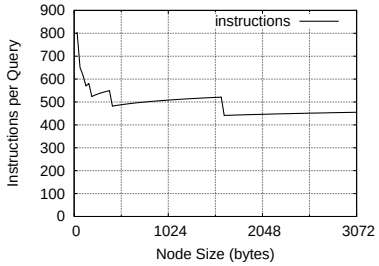

Figure 4: CSB$^+$-tree, Instructions per Query ($q = 10,000, f = 0.67$)

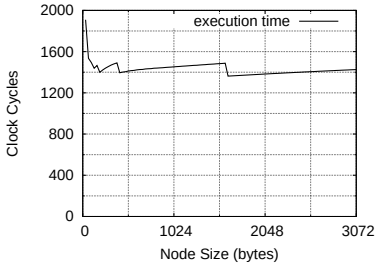

Figure 5: CSB$^+$-tree, Execution Time per Query ($q = 10,000, f = 0.67$)

Figure 4 shows the average number of instructions executed during an equality search for various node sizes (Eq. 9). 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, whereas the binary search cost increases at a gradual rate. Consequently, for the larger node sizes there are fewer instructions executed per query.

Figure 5 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, 5, and 9. As the node size decreases, the cache miss latency is contributing less to the overall execution time while the instructions are contributing much more. Therefore, while the cache performance may be optimal for small node sizes, the execution time is adversely affected by the high instruction count. For the CSB$^+$-tree, the minimum number of cache misses occurs at a node size of 128 bytes, and the minimum number of instructions executed occurs at 1632 bytes. The predicted optimal node size is 1632 bytes, performing 26% faster over a node size of 32 bytes, which is the L2 cache line size for the Intel Pentium III.

The number of branch mis-predictions (Eq. 10) follows a trend similar to the number of instructions executed, contributing a fairly constant 17–19% to the overall execution time, over the entire range of node sizes. In the interest of space, the branch mis-prediction performance is not presented here, but can be found in the full-length version of this paper [16].

To determine the effects of adding concurrency control to the CSB$^+$-tree, we simulate the cost of page level locking on the search performance of the CSB$^+$-tree. Summarizing the results, we found that the overhead of page level locking lowers the search performance by a fairly constant amount over the range of node sizes examined. The interested reader can find the results of this analysis in the full-length version of this paper [16].

3.3 Extendible Hashing

An equality search on the hash index has three essential operations: the directory lookup, bucket chain traversal, and bucket search. To determine a candidate bucket chain, a hash function is performed on the search key, providing an index into the bucket directory. Once the starting bucket in the bucket chain is located, each overflow bucket is searched for matching entries.

For each access of the directory, a data cache miss will be incurred, assuming the directory entry size is less than the length of a cache line. Because there is no ordering across the overflow chain, each overflow bucket must be searched for the desired entry, incurring the cost of a binary search for each bucket. The total number of cache misses, m_{hash}, incurred per index query is:

$$m_{hash} \quad = \quad 1 + C * \lceil \log_2(l+1) \rceil \,, \tag{11}$$

where C is the average overflow chain length.

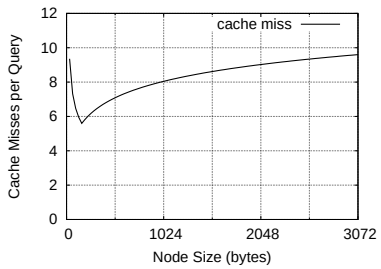

Figure 6: Hash Index, Cache Misses per Query ($q = 10,000$)

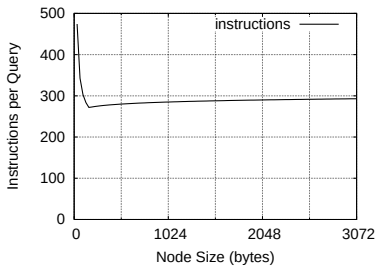

Figure 7: Hash Index, Instructions per Query ($q = 10,000$)

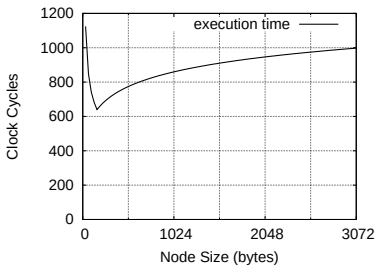

Figure 8: Hash Index, Execution Time per Query ($q = 10,000$)

The instruction count model, Equation 12, includes the instructions needed to: read the bucket directory, perform a binary search inside a bucket, and traverse to the next bucket in the overflow chain.

$$i_{hash} = I_{dir} + C * (I_{search} * \log_2(\epsilon + 1) + I_{trav}) \tag{12}$$

Values for these costs are given in Table 1.

Similar to the CSB$^+$-tree, the number of mis-predicted branches is proportional to the number of binary search operations executed. Equation 13 estimates the number of mis-predicted branches.

$$b_{hash} = \frac{C * \log_2(\epsilon + 1)}{2} \tag{13}$$

In Equation 13, the number of mis-predicted branches is 50% of the branches taken in the binary search of the buckets. As in the model for the CSB$^+$-tree, execution time is calculated using Equations 1, 11, 12, and 13.

3.3.1 Analysis

Figure 6 shows the average number of cache misses per query (Eq. 11) for various node sizes. Because we have an upper limit on the size of the hash directory, small bucket sizes cause the average length of the overflow chain to be greater than one. As the bucket size increases, the average length of the overflow chain decreases, resulting in fewer cache misses. The number of cache misses reaches its minimum point when the bucket size is such that there are no overflow chains (160 bytes in the Figure). As the bucket size continues to increase, the number of cache misses increases because of the binary search on the larger buckets. Note that the directory structure only doubles when a bucket overflows, so the directory may not reach its maximum size for large bucket sizes.

Figure 7 shows the number of instructions executed (Eq. 12) for various node sizes. Again, there is a sharp drop in the number of instructions as the length of the bucket chain is reduced to one. The reduction in instructions is a result of traversing fewer overflow buckets. After the sharp decrease in instructions executed, there is a steady increase in the number of instructions, starting at a node size of 160 bytes, due to the binary search on the larger buckets.

Figure 8 plots the effect of node sizes on the overall execution time, produced using Equations 1, 11, 12, and 13. For small node sizes, most of the execution time is spent traversing the overflow buckets. For large node sizes, the majority of time is spent performing entry searches. The optimal node size range for the hash index is in the the range 160–192 bytes, where the overflow chains are reduced to one bucket. The model demonstrates that small node sizes, on the order of a cache line size, can create additional structural inefficiencies, such as long overflow chains, that far outweigh any benefit from improved cache line utilization.

For the hash index, the maximum permitted size of the directory has a critical impact on the number of overflow chains (and consequently on the overall performance). We used the model to investigate the effect of the directory size. As the size of the directory increases, the performance of the index generally improves. However, larger directories require more storage space, and often it is desirable to put an upper limit on this space [20, 24, 29]. Given a limit on the directory size, we have used the model to show that the optimal performance point occurs when the node size is such that there are no overflow chains, regardless of the size of the node with respect to the cache line size. In the interest of space, we omit these results here, and refer to the full length version of this paper [16]; however, we will return to this topic in the experimental section where we experimentally demonstrate this effect.

The number of branch mis-predictions (Eq. 13) follows a trend similar to the number of instructions executed, contributing approximately 5–8% to the overall execution time over the entire range of node sizes. In the interest of space, the branch mis-prediction performance is not presented here, but can be found in the full-length version of this paper [16].

3.4 Analytical Model Discussions

The execution time results in this section are based on the Intel Pentium III architecture, but the analytical models can easily be modified for other processors. The analytical models we present are a first-order approximation of the performance of the indices, and while more detailed models can be created, the experimental section will confirm that these simple models are fairly accurate in capturing the overall trends.

4 Experimental Evaluation

In this section, we present experimental results based on an implementation of both the CSB$^+$-tree and the extendible hash index in an experimental main memory database system, called Quickstep, that we are currently building.

4.1 Index Implementation Details

The indices were implemented in an actual database system, and as a result, system abstractions may have induced additional performance overhead for some of the index operations. For example, our system allocates memory in pages that can be saved to disk or transferred outside of the system. We also utilize a buffer manager to manage caching of pages in main memory. The Quickstep 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 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, we always kept the entire data set pinned in main memory. Within the system constraints, we tried to optimize the indices as much as possible. The indices and database system were coded in C++, and the system was compiled with the *gcc* compiler from GNU with all optimizations turned on.

We implemented the *full* CSB$^+$-tree as described in [25]. All nodes were allocated in pages of memory, with groups of nodes allocated on contiguous pages of memory if necessary. All pages were pinned in memory, and direct memory pointers were used to minimize the system overhead. All keys are four bytes, and all $\langle key, RID \rangle$ entries are eight bytes.

For the hash index, all $\langle key, RID \rangle$ entries are eight bytes. The overhead for the hash directory is typically designed to be between 1–10% [29], so unless stated otherwise, the maximum directory size is set at 512K entries, or 4MB, which is approximately a 5% space overhead for ten million index entries. Similar

to the CSB$^+$-tree, the hash buckets are allocated in pages of memory, and direct memory pointers were used to reference the buckets.

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 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 Linux, kernel version 2.4.13. To access the event counters on the processor, we used the PAPI library [6].

We also verified these results on a 450MHz Sun UltraSPARC-II with 1024MB of main memory, running SunOS version 5.8. The UltraSPARC-II also 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 4MB unified data/instruction, with a 64 byte line size. The results of the UltraSPARC-II are similar to the results obtained for the Pentium III, and the same conclusions as presented in this section can be drawn for the UltraSPARC-II. However, the cache line size for the SPARC processor is twice the line size used in the Pentium processor, and 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 [16].

For the experiments that are considered in this section, the implementation does not incur any index locking overhead. We have also analyzed the performance impact of having concurrency control on the CSB$^+$-tree grouping nodes into pages using the organization technique presented in [10]. With this orgaization, multiple levels of the index are stored in a page, which amortizes the cost of a page lock over a number of levels in the index. Our experimental evaluation of the locking overhead shows that the cost of locking is fairly constant across a wide range of node sizes. These results can be found in [16].

The events that we measured include execution time, L2 cache misses, instructions executed, L1 instruction cache misses, and branch mis-predictions. The L1 instruction cache misses contribute an insignificant percentage of overall execution time (less than 0.5%), and are not discussed in the analysis of the experiments. For the CSB$^+$-tree and the hash index, the branch mis-predictions costs are fairly constant across all the node sizes, contributing about 16% and 5% to the overall execution time, respectively. In addition, record retrieval and output operations are not included in our measurements of the index performance.

4.3 Data Set and Queries

For the data set, we used the Wisconsin Benchmark's [3, 13] 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, unless stated otherwise, the indexes are queried ten thousand times. We divided the measured events by the total number of queries to calculate an average event per query, and reported them in this section (so each data point in a graph is an average of ten thousand queries). We also ran experiments with much larger numbers of queries, and relations with different cardinalities. The per query performance results presented here remained essentially the same; for brevity we omit these results here, and refer the interested reader to [16].

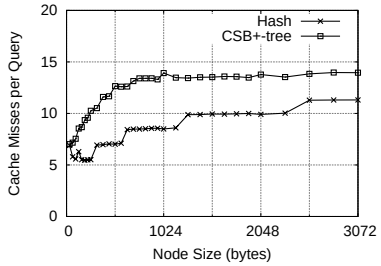

Figure 9: Equality Searches: Cache Misses

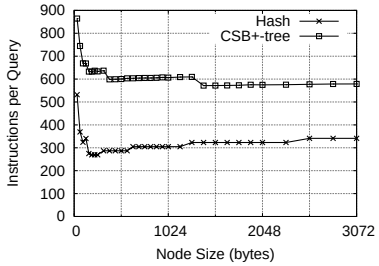

Figure 10: Equality Searches: Instructions

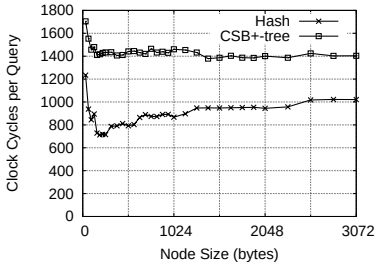

Figure 11: Equality Searches: Execution Time

4.4 Measuring Performance

In analyzing the experiments, we will frequently refer to the optimal range of node sizes for a given performance metric. For our analysis, this range is defined as the node sizes that are within 5% of the optimal node size 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: $\frac{performance\ before}{performance\ after} - 1\%$. When reporting the performance results, the performance improvement will be presented first, with the percentage speedup presented second and in parenthesis.

4.5 Expt 1: Search Performance

To analyze the effect of node size on the search performance of both indexes, we first present an analysis of equality searches on both indexes over a wide range of node sizes. Since the CSB$^+$-tree also supports range searches, we also present the effects of the node size on the performance of CSB$^+$-tree range searches.

4.5.1 Equality Search

In the first experiment, both indexes were constructed on the *unique1* attribute in the scaled TENK relation (cardinality 10M). The number of cache misses, number of instructions, and the execution time are reported on a per query basis in Figures 9, 10 and 11 respectively.

Figure 9 shows the average number of cache misses per query for both the CSB$^+$-tree and the hash index. 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 nodes sizes become larger, the binary search within each node increasingly contributes to the number of cache misses.

We also observe in Figure 9 that the hash index experiences its fewest cache misses for the bucket size range of 160–256 bytes. For smaller bucket sizes, the hash index has long overflow chains, resulting in a large number of cache misses. When the bucket size is 168 bytes, there are no overflow chains in the hash index, and the hash index has the best performance. After this point, there are no overflow buckets, and the number of cache misses increases due to the binary search on the increasing bucket sizes.

Figure 10 shows the average number of instructions per query for the two index structures. As the node size increases for the CSB$^+$-tree, the height of the tree decreases, causing the instruction count to drop. The jumps that are seen in Figure 10 for the CSB$^+$-tree correspond to changes in the height of the index structure. For a given height, as the node size increases, the cost of the binary search increases, causing a gradual increase in the instruction count. The instruction count is at its minimum within the range of 1408–3072 bytes.

For the hash index, the instruction count in Figure 10 sharply decreases until there are no overflow buckets, which happens when the node size is 168 bytes. After this point, as the node size increases, the

instruction count increases gradually due to the binary search on the increasing node sizes. The instruction count is near its minimum point within the range of 160–256 bytes.

Figure 11 shows the execution time, measured as clock cycles per query, for the two indices. As the figure shows, the CSB$^+$-tree's best performance occurs at node sizes greater than **160 bytes**. The hash index's best performance occurs for node sizes in the range of **160–256 bytes**, corresponding to the node size that results in no overflow buckets. Both of these ranges are much larger than the cache line size, which is 32 bytes. For the CSB$^+$-tree, using a node size within the range 256–512 bytes can improve the performance of the index by 17% (21% speedup) over the performance when the node size is 32 bytes. Using a node size in the range of 1280–3072 bytes can improve the performance of the index up to **19% (24% speedup)** over the performance when the node size is 32 bytes. Similarly for the hash index, using a node size in the range of 160–256 bytes improves the performance of the index up to **42% (72% speedup)** over a node size of 32 bytes.

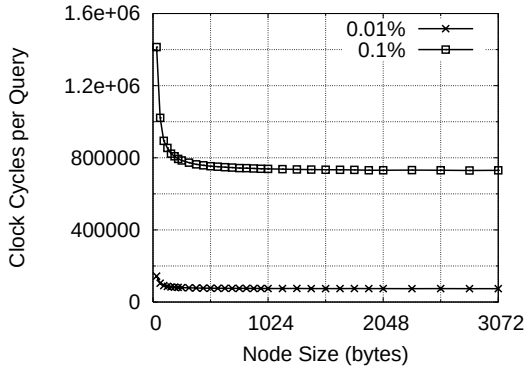

Figure 12: Pentium III Range Searches: CSB$^+$-tree, Execution Time

4.5.2 Range Search

In this experiment, we assess the performance of the CSB$^+$-tree when evaluating range search queries for two ranges: 0.01% and 0.1% of the *unique1* attribute in the TENK relation scaled to ten million entries. The performances of 1.0% and 10.0% range scans are similar to the results presented here, and in the interest of space, we omit these results and direct the reader to the full-length version of this paper [16].

The range search on the CSB$^+$-tree index proceeds as follows. The tree is traversed to find the first leaf-node entry that matches the range's lower boundary. All subsequent entries are then found by sequentially scanning the leaf nodes. The search terminates on the first leaf-node entry that is above the range's upper boundary.

The execution time results for the range search experiment are shown in Figure 12. As was the case with equality searches, we observe that using a node size in the range of 256–512 bytes performs up to 47% (88% speedup) faster than a CSB$^+$-tree designed with the conventional node size equal to a cache line size (32 bytes for the Pentium III). Using a node size larger than 512 bytes improves the range search performance up to **48% (94% speedup)** over a CSB$^+$-tree designed with the conventional node size.

4.6 Expt 2: Space Requirements

In this experiment, we investigate the effect of the node size on the space required to store the CSB$^+$-tree index. We do not consider the space required for the hash index in this experiment, as our hash index is an extendible hash index whose directory size increases or decreases depending on the number of inserted entries, resulting in a fairly constant space requirement across different node sizes. We discuss the issue of the directory size in the next experiment.

The CSB$^+$-tree 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. For this experiment only, the entries were inserted in sorted order to obtain a consistent overall fill factor, creating nodes that are only half full in almost all cases. The space requirements of the CSB$^+$-tree versus node size in Figure 13 highlights an important problem with using small node sizes for memory constrained environments. As expected, small node sizes result in very deep trees, requiring substantially more space to represent. From the figure, a node size of 512 bytes requires 55% less space than a node size of 32 bytes, and requires 27% less space than a node size of 64 bytes. Node sizes larger than 512 bytes show moderate improvements in space utilization, requiring 57% less space than a node size of 32 bytes.

4.7 Expt 3: Directory Size

For the hash index, the performance of the index depends on the number of buckets in the overflow chains, which in turn depends on the cardinality, the distribution of data values, and the size of the directory. In this experiment, we examine the performance of the hash index for a variety of directory sizes. The index is constructed on the *unique1* attribute of the TENK relation scaled to ten million entries.

Figure 14 plots the execution time of equality searches on hash indexes with various directory sizes. The directory sizes ranged from 1% to 40%, which corresponds to a range of entries from 128K to 4096K entries. For a given directory size, the performance is best for the smallest bucket size that produces no overflow chains. From the figure, we also observe that the performance of the index improves as the directory size increases. In addition, the larger the directory size, the smaller the size of the node that results in best performance. Unfortunately, a very large directory may be space prohibitive. In choosing a directory size, often the rule of thumb is to use between 1% and 10% of the space for the directory [29]. As shown in the figure, the optimal node size increases from 96 bytes to 704 bytes as the directory size decreases from 10% to 1% respectively. An additional insight illustrated by the experiment is that when choosing the node size, *it is better to err on the side of larger nodes, if the data distribution or cardinality is not known accurately in advance*.

4.8 Expt 4: Larger Key Size

In the previous experiments, we analyzed the search performance of indexes constructed on 4-byte keys. We now examine the performance of equality search on both indexes when constructed on 8-byte keys. For this experiment, both indices were constructed on the *unique1* attribute of the TENK relation, scaled to ten

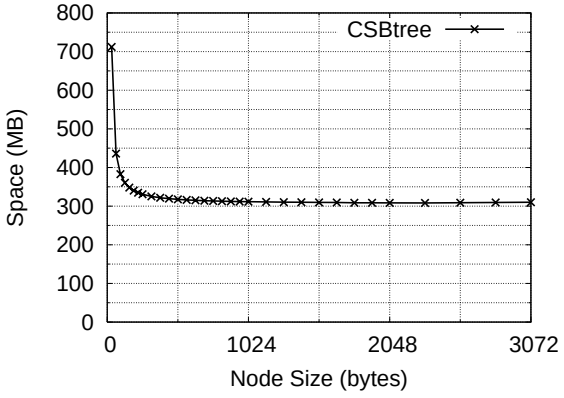

Figure 13: Space Requirements

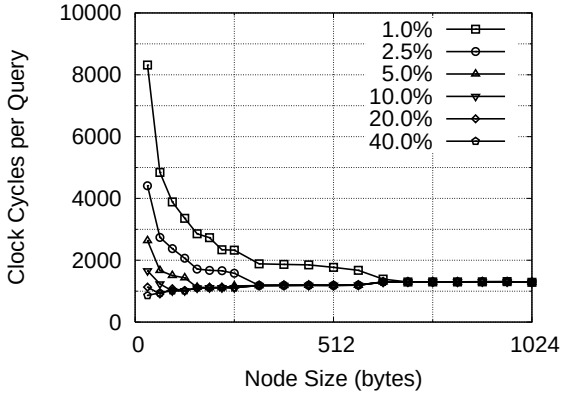

Figure 14: Directory Sizes: Hash Index, Execution Time

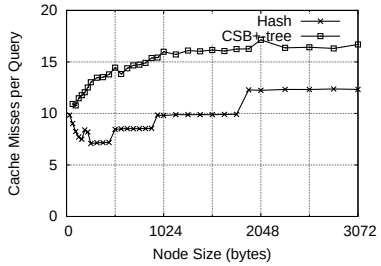

Figure 15: Equality Search with Non-Integer Keys: Cache Misses

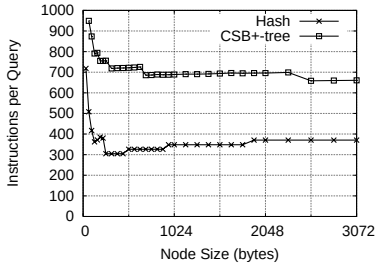

Figure 16: Equality Search with Non-Integer Keys: Instructions

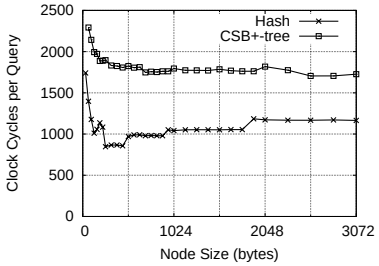

Figure 17: Equality Search with Non-Integer Keys: Execution Time

million entries. We store each attribute value as type double precision float to examine the performance of each index for a key size of 8-bytes.

Figure 15 shows the number of cache misses for both indexes. In the figure, the CSB$^+$-tree and hash indexes demonstrate similar trends in cache miss behavior compared to the same indexes with 4-byte keys (Fig. 9). However, for very small node sizes, the number of cache misses are almost double their previous values (with 4-byte keys) since each node now contains fewer entries.

Figure 16 shows the number of instructions executed for both indexes. The performance of both indexes is similar to their performance when indexing 4-byte keys.

Figure 17 shows the number of processor clock cycles executed for both indexes. Again, the performance of both indexes is similar to their performance when indexing 4-byte keys. Because the number of cache misses are double the number of cache misses when indexing 4-byte keys, the contribution of the cache misses to the overall execution time increases. Even so, the execution time continues to follow the trend in instructions executed.

From Figure 17, the CSB$^+$-tree experiences its optimal performance for node sizes larger than 1024 bytes, performing up to 26% (34% speedup) faster than a node size of 64 bytes. (For a node size of 32 bytes, the cache line size on the Pentium III, the CSB$^+$-tree exceeded the alloted main-memory space, and is not presented in the performance figures; the performance difference between the optimal and the "conventional" is expected to be larger.) A node size of 512 bytes performs 20% (26% speedup) better than at a node size of 64 bytes. From Figure 17, the hash index experiences its optimal performance for node sizes in the range of 256–448 bytes, performing up to 51% (106% speedup) faster than a node size of 32 bytes. *This experiment 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.*

4.9 Expt 5: Equality Search with Duplicates

In this experiment, both indices are loaded with ten million integers which are drawn from a set of five-hundred thousand distinct integer values. These data sets follow a Zipfian distribution [30] with a specific *skew parameter*. For a skew of 0, there are approximately twenty duplicates for each value. As the skew parameter increases, some values are duplicated many more times than others. The query set for this experiment included one query for each of the distinct values. In the interest of space, we only present the results for data sets with skews of 0 and 1.0.

Figures 18, 19, and 20 show the effect of duplicates on the performance of the CSB$^+$-tree for various node sizes. Additional cache misses now result from scanning duplicate elements in the leaf nodes, increasing the number of cache misses per query compared to the equality search in Figure 9. The instruction count per query also increases as the leaf nodes are scanned for all matching elements. The final execution time in Figure 20 shows node sizes in the range of 256–512 bytes perform up to 49% (95% speedup) better than

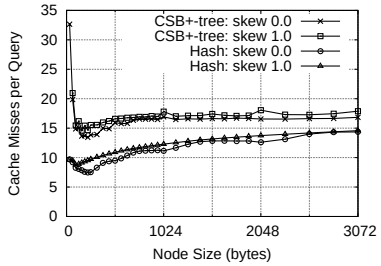

Figure 18: Equality Search with Duplicates: Cache Misses

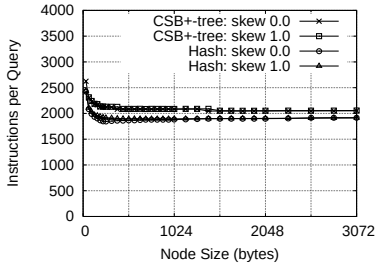

Figure 19: Equality Search with Duplicates: Instructions

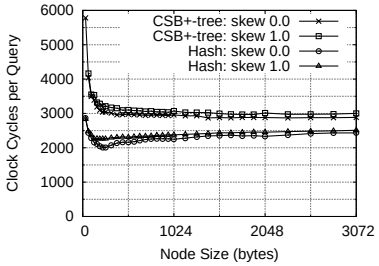

Figure 20: Equality Search with Duplicates: Execution Time

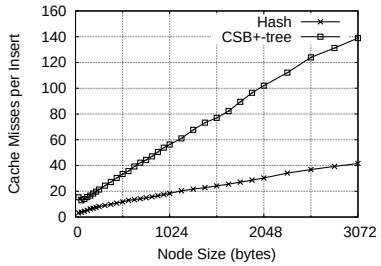

Figure 21: Insertions: Cache Misses

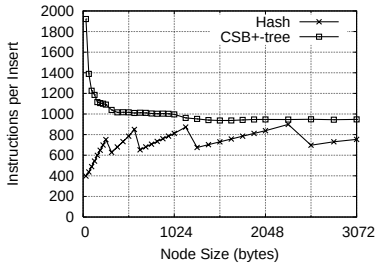

Figure 22: Insertions: Instructions

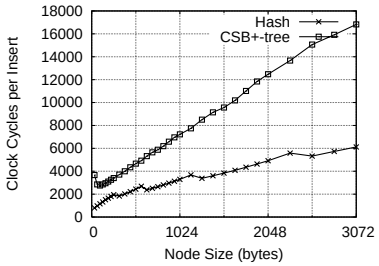

Figure 23: Insertions: Clock Cycles

a node size equal to the cache line size of 32 bytes. Node sizes larger than 512 bytes perform up to 50% (100% speedup) better than a node size equal to 32 bytes.

The effects of duplicates on the hash index are also shown in Figures 18, 19, and 20. For very small bucket sizes, the hash index incurs a large number of cache misses due to long overflow chains. As the bucket size increases, the number of cache misses drops significantly because the number of overflow buckets is reduced. After a point, however, increasing the bucket size does not have a large effect on the number of overflow chains because of the duplicate entries. As the bucket size becomes larger, the number of cache misses increases because of the binary search within each bucket. From Figure 20, we see that for a hash index with uniformly distributed duplicate entries, bucket sizes in the range of 192–320 bytes are optimal. For a hash index with a skewed distribution of duplicate values, bucket sizes in the range of 96–896 bytes are optimal. The optimal range of node sizes is larger when indexing skewed duplicates because some highly duplicated keys benefit from larger node sizes, while key values with only a few duplicates benefit from smaller bucket sizes.

4.10 Expt 6: Insert Performance

In this experiment, we evaluate the effect of node size on the performance of the indices when inserting ten million unique, unsorted integers into an empty index. For the integer values, we used the *unique1* attribute of the TENK relation scaled to ten million entries.

Insertion performance for both indices is shown in Figures 21, 22, and 23. From the cache misses plotted in Figure 21, we observe that the number of cache misses incurred by both indices increases as the node size increases because the amount of data that must be moved on inserts is proportional to the node size. As expected, the CSB$^+$-tree shows a much larger number of cache misses than the hash index since there are a number of node splits.

For the instruction count shown in Figure 22, as the CSB$^+$-tree node size increases, there are fewer levels to traverse, reducing the instruction count. For the hash index, as the node size increases, the directory

split costs increases rapidly. The stair-step pattern of the instruction count for the hash index is due to the directory splits.

The execution time is presented in Figure 23. From the figure, as the node size increases, the execution time also increases, following the cache miss pattern closely in both cases. For both indexes, smaller node sizes perform much better than larger node sizes. From the figure, a CSB$^+$-tree with a node size of 512 bytes performs 27% (21% slowdown) slower than when using a node size of 32 bytes. Node sizes larger than 512 bytes perform increasingly worse.

From Figure 23, the hash index experiences its best performance at a node size of 32 bytes. The performance of the hash index decreases as the node size increases. It is important to note that the hash index keeps entries in sorted order inside of the buckets. As described in Section 2, this modification was made to benefit the search operation over insertions.

4.11 Expt 7: Join Operator Performance

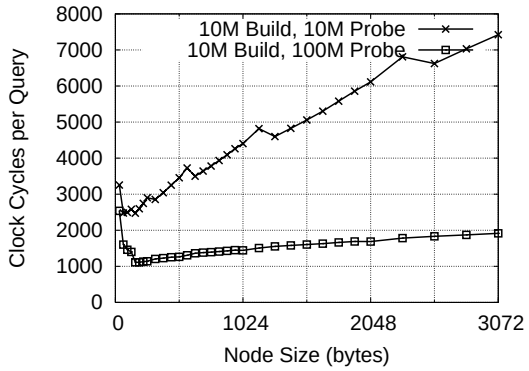

Figure 24: Hash Join: Execution time (10M Build Relation, 10M/100M Probe Relation)

In this experiment, we evaluate the performance of the hash index when used in a query operation such as a hash join operator [12, 26]. In this operator, a hash index is first built using tuples from the *build relation*, and then probed with tuples from the *probe relation*. We use two queries in this experiment: In the first query, both the build and probe relations are created on the unsorted *unique1* attribute of the Wisconsin TENK relation scaled to ten million entries, and in the second query, the probe relation is created on the unsorted *unique1* attribute of the Wisconsin TENK relation scaled to one hundred million entries. Figure 24 plots the results for this experiment. For the first query, since inserts are more expensive than searches, the execution time closely follows the hash performance during inserts, discussed in Section 4.10. Consequently, smaller node sizes result in better performance.

For the second query, the overall performance follows the index search behavior closely. As seen in the figure, node sizes around 192 bytes perform 56% (128% speedup) faster than when the node size is set to the cache line size of 32 bytes. We note that when evaluating a hash join operation, the database query optimizer typically chooses the build relation as the relation that has fewer tuples, so the second query may be more representative of a typical join operation.

4.12 Discussions

In summary, we have experimentally demonstrated that, for the CSB$^+$-tree index, node sizes of 512 bytes and larger generally perform much better on searches than the conventional node size. Node sizes larger than 512 bytes generally perform up to 19% (24% speedup) faster for equality searches and 48% (94% speedup) faster for range searches when compared to the performance at a node size equal to a cache line size (on the Pentium III). However, large node sizes suffer poor insert performance, as a CSB$^+$-tree with a node size of

512 bytes performs 27% (21% slowdown) slower than a CSB$^+$-tree with a node size of 32 bytes. Node sizes larger than 512 bytes show increasingly worse insert performance. Consequently, larger node sizes are most effective in improving the search performance when updates are rare. In addition, we have shown that the conventional node size demonstrates poor space utilization, consuming up to 124% more space than a node size of 512 bytes. For memory constrained systems, small node sizes may not be feasible.

For the extendible hash index, we have demonstrated the importance of reducing the length of the overflow chains. We have shown that the hash index's bucket size should be chosen to eliminate bucket chains, and not chosen based simply on the cache line size. Choosing the bucket size in this way improved performance by 42% (72% speedup) for equality searches, and 56% (128% speedup) when the hash index is used in a hash join operation.

5 Related Work

A number of previous studies have identified the critical impact that processor cache misses have on the performance of modern database systems, which typically run on servers with large main memory configurations [1, 21, 23, 27]. To remedy this performance bottleneck, a number of popular database algorithms and access methods have been adapted to improve their cache utilization [5, 14, 23, 27]. Main memory database systems have also paid considerable attention to efficient indexing techniques, and the work by Lehman and Carey [20] is one of the earliest. The authors investigate 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 the effectiveness of index structures in main memory environment by considering the performance impact of cache misses. In [24], 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 on updates to the data. The authors also investigated dynamic indexing techniques in the main-memory environment in [25], 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 [4] and for architectures that support prefetching [9].

In a recent paper, Chen, Gibbons and Mowry [9] 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. We recognize that node size influences both the number of cache misses and the instruction count, and that a node size can be chosen that balances these two factors, improving overall performance.

Chen, Gibbons, Mowry, and Valentin also propose a version of their prefetching B+-tree optimized for disk pages, called Fractal pB+-trees [10]. 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, called the CR-Tree [19]. The CR-Tree improves the cache utilization of the traditional R-Tree by reducing the space required to store the minimum bounding rectangles, or MBRs. Through an analytical model and experimental results, the authors show that node sizes larger than a cache line size generally outperform node sizes approaching the cache line size. In addition, as the query selectivity, data cardinality, and data dimensionality increase, the optimal node size is also found to increase. Our work in this paper compliments their work on multi-dimensional indices by presenting an analytical model and experimental results for the single-dimensional CSB$^+$-tree. In this paper, we have shown that node sizes larger than the cache line size are optimal for single-dimensional indices as well.

Cha, Hwang, Kim, and Kwon have also analyzed the performance of the CSB$^+$-tree and the traditional B+-Tree when incorporating concurrency control logic and used in a shared memory multi-processor system [8]. The authors found that a node size of twice the cache line size is optimal for the CSB$^+$-tree. As the description of the experiment that supports this conclusion is sparse, there may be many reasons why their conclusion contradicts our findings. 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 [22], which follows the work of Gray and Graefe [15]. 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. [11] examines how the compiler can change the layout of data structures to improve cache-behavior of the program. They propose compiler optimization techniques to optimize the layout of data structures used in coding Microsoft's SQL Server, improving the performance of SQL Server by 1–2%. Since the proposed technique is a general purpose compiler technique, the authors do not consider changing the database algorithms or implementations.

6 Conclusion

In this paper, we investigate the performance of two main memory indexing structures, namely a recently proposed cache-sensitive B+-tree index, the CSB$^+$-tree, and a main memory extendible hash index. We introduce first-order analytical models for the index structures. From the analysis of these models, we demonstrate that using the common design heuristic of setting the node size equal to the cache line size for cache-conscious index structures is often suboptimal. We show that both cache misses and instruction count must be balanced to achieve optimal index performance.

We also report results from extensive experimentation on both of these index structures. 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. Larger node sizes generally perform better for searches, but suffer poor insert performance. For the hash index, the number of buckets in overflow chains has a critical impact on the performance of the index. The smallest bucket size that results in an index with no overflow chains generally has the best performance, regardless of the processor cache line size. Another critical parameter for the hash index is the size of the directory. Larger directories improve the performance of the index, but take up more space.

We also experimentally demonstrate that these results hold when the relation being indexed has duplicate key values, larger key sizes, and for range search queries evaluated using the CSB$^+$-tree. The results that we present in the paper can be applied to main memory databases and even traditional databases that use main

memory type of index structures for operations such as hash joins. 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 few of the more popular microprocessors, the results of this paper should be used cautiously when applying to implementations running on other processors. However, our results are centered around the observation that, in modern processors, the overall performance of an index structure depends on the number of cache misses and the instructions that are executed. Our analytical model captures these characteristics using a simple model which can easily be adapted for other architectures.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. of 25th Int'l Conference on Very Large Data Bases, Edinburgh*, pages 266–277, September 1999.

[2] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The Asilomar Report on Database Research. *SIGMOD Record*, 27(4):74–80, 1998.

[3] D. Bitton and C. Turbyfill. A Retrospective on the Wisconsin Benchmark. In *Readings in Database Systems*, pages 422–441. Morgan Kaufmann, 1994.

[4] P. Bohannon, P. McIlroy, and R. Rastogi. Main-Memory Index Structures with Fixed-Size Partial Keys. In *SIGMOD Conference*, 2001.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of 25th Int'l Conference on Very Large Data Bases, Edinburgh*, pages 54–65, September 1999.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The Int'l Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

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

[8] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *The VLDB Journal*, pages 181–190, 2001.

[9] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance through Prefetching. In *Proc. of ACM SIGMOD Int'l Conference on Management of Data, Santa Barbara*, pages 235–246, May 2001.

[10] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B+trees: Optimizing Both Cache and Disk Performance. In *Proc. of ACM SIGMOD Int'l Conference on Management of Data, Madison*, May 2002.

[11] T. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *PLDI*, May 1999.

[12] D. DeWitt, R. Katz, F. Ohlken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Databases. *SIGMOD Record*, 14(2):1–8, 1984.

[13] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, 1993.

[14] G. Graefe, R. Bunker, and S. Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *Proc. of 24th Int'l Conference on Very Large Data Bases, New York City*, pages 86–97, August 1998.

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

[16] R. A. Hankins and J. M. Patel. Effect of Node Size on the Performance of Cache-Conscious Indices. *Extended Report, http://www.eecs.umich.edu/quickstep/publ/ccindices.pdf*, 2002.

[17] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1998.

[18] Intel Corporation. Intel Architecture Optimization Reference Manual.

[19] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 139–150. ACM Press, 2001.

[20] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Twelfth Int'l Conference on Very Large Data Bases, Kyoto*, pages 294–303, August 1986.

[21] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *ISCA*, pages 39–50, 1998.

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

[23] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. AlphaSort: A Cache-Sensitive Parallel External Sort. *VLDB Journal*, 4(4):603–627, 1995.

[24] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proc. of 25th Int'l Conference on Very Large Data Bases, Edinburgh*, pages 78–89, September 1999.

[25] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proc. of the 2000 ACM SIGMOD Int'l Conference on Management of Data, Dallas*, pages 475–486, May 2000.

[26] L. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, Oct. 1986.

[27] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of 20th Int'l Conference on Very Large Data Bases, Santiago de Chile*, pages 510–521, September 1994.

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

[29] H. Zeller and J. Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In *16th Int'l Conference on Very Large Data Bases, Brisbane*, pages 186–197, August 1990.

[30] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley (Reading MA)., 1949.