Redesigning Database Systems in Light of CPU Cache Prefetching

Shimin Chen

CMU-CS-05-192

December 2005

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Anastassia Ailamaki, Co-Chair
Todd C. Mowry, Co-Chair
Christos Faloutsos
Phillip B. Gibbons
David J. DeWitt, University of Wisconsin at Madison

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation (NSF) under grant nos. CCR-0205544 and CCR-0085938, the National Aeronautics and Space Administration (NASA) under grant no. NAG2-1230, and the Microsoft Corporation through a generous fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Cache Prefetching, Database Systems, CPU Cache Performance, Data Locality Optimizations, B$^+$-Trees, Hash Joins.

To my parents Guanhu and Aili, my wife Qin, and my daughter Ada.

Abstract

Computer systems have enjoyed an exponential growth in processor speed for the past 20 years, while main memory speed has improved only moderately. Today a cache miss to main memory takes hundreds of processor cycles. Recent studies have demonstrated that on commercial databases, about 50% or more of execution time in memory is often wasted due to cache misses. In light of this problem, a number of recent studies focused on reducing the *number* of cache misses of database algorithms. In this thesis, we investigate a different approach: reducing the *impact* of cache misses through a technique called *cache prefetching*. Since prefetching for sequential array accesses has been well studied, we are interested in studying non-contiguous access patterns found in two classes of database algorithms: the B$^+$-Tree index algorithm and the hash join algorithm. We re-examine their designs with cache prefetching in mind, and combine prefetching and data locality optimizations to achieve good cache performance.

For B$^+$-Trees, we first propose and evaluate a novel main memory index structure, Prefetching B$^+$-Trees, which uses prefetching to accelerate two major access patterns of B$^+$-Tree indices: searches and range scans. We then apply our findings in the development of a novel index structure, Fractal Prefetching B$^+$-Trees, that optimizes index operations both for CPU cache performance and for disk performance in commercial database systems by intelligently embedding cache-optimized trees into disk pages.

For hash joins, we first exploit cache prefetching separately for the I/O partition phase and the join phase of the algorithm. We propose and evaluate two techniques, Group Prefetching and Software-Pipelined Prefetching, that exploit inter-tuple parallelism to overlap cache misses across the processing of multiple tuples. Then we present a novel algorithm, Inspector Joins, that exploits the free information

obtained from one pass of the hash join algorithm to improve the performance of a later pass. This new algorithm addresses the memory bandwidth sharing problem in shared-bus multiprocessor systems.

We compare our techniques against state-of-the-art cache-friendly algorithms for B$^+$-Trees and hash joins through both simulation studies and real machine experiments. Our experimental results demonstrate dramatic performance benefits of our cache prefetching enabled techniques.

Acknowledgments

I must thank my advisors, Todd Mowry and Anastassia Ailamaki, for the tremendous time, energy, and wisdom they invested in my Ph.D. education. Todd and Natassa taught me everything from choosing research topics, to performing high-quality studies, to writing papers and giving talks. Their guidance and support throughout the years are invaluable.

I am indebted to Phillip Gibbons for collaborating on my entire thesis research, for participating in our weekly discussions, and for contributing his knowledge, sharpness, and efforts to all the four publications that form the basis of this thesis.

I would like to thank the other members of my Ph.D. thesis committee, David DeWitt and Christos Faloutsos, for their thoughtful comments and invaluable suggestions that have improved the quality of the experimental results and the completeness of this thesis.

I thank Gary Valentin for collaborating on the fpB^+-Tree paper. Gary implemented jump pointer array prefetching in IBM DB2 and collected DB2 results on B^+-Tree range scan I/O performance for the paper. I am grateful to Bruce Lindsay for giving a CMU DB Seminar talk on the DB2 hash join implementation, which inspired my hash join work. My thesis work also benefitted from insightful discussions with David Lomet and Per-Åke Larson.

I would like to acknowledge Angela Brown, Kun Gao, John Griffin, Stratos Papadomanolakis, Jiri Schindler, Steve Schlosser, and Minglong Shao, for helping me set up machines for running real-machine experiments in my publications and in this thesis; Chris Colohan and Gregory Steffan, for helping me understand the internals of the “cello” simulator for my simulation studies; members of the CMU database group and STAMPede group, for giving comments on my practice talks; staff of CMU SCS facilities, for allowing me to reboot machines in the machine room after working hours; and Sharon Burks, for her administrative helps in arranging my thesis proposal and defense talks.

My summer internships expanded my research experience. I would like to thank David Lomet and Phillip Gibbons for mentoring me during my internships. I also want to thank all the people in IBM Toronto Lab, Microsoft Research Redmond, and Intel Research Pittsburgh, for making my summers productive and enjoyable.

I thank my friends in Pittsburgh, Toronto, and Seattle for their helps in my CMU graduate life and in my summer internships, and for adding a lot of funs to my life.

Finally, I must express my deepest gratitude to my family. I owe a great deal to my parents, Guanhu and Aili, who gave a life, endless love, and persistent encouragement to me. My sister, Yan, and brother-in-law, Gang, are always helping and supportive. I am deeply indebted to my dear wife, Qin, for sharing every moment of the CMU graduate life with me. Without her love, patience, encouragement, and support, it is impossible for me to complete this six and half years of long journey. Last but not least, my one-year-old daughter, Ada, motivated me to finish my thesis with her sweet voice of “baba”.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems have enjoyed an exponential growth in processor speed for the past 20 years, while DRAM main memory speed has improved only moderately [39]. Today a cache miss to main memory takes hundreds of processor cycles (e.g., about 200 cycles on Itanium 2); considering the wide issue rate (e.g., up to six instructions on Itanium 2 [44]) of modern processors, this represents a loss of about a thousand or more instructions. Unfortunately, the CPU cache hierarchy provides only a partial solution to this problem. It works effectively when the working set of a program fits in the CPU cache, as evidenced by many of the SPEC benchmark results [80]. However, for programs working on large data sets with poor data locality, the *cache-to-memory latency gap* can be a major performance bottleneck. Recent database performance studies have demonstrated that, on commercial database systems, about 50% or more of the execution time in memory is wasted due to cache misses [2, 5, 51].

In light of this problem, a number of recent research studies focused on improving the CPU cache performance of database systems [1, 13, 14, 15, 28, 31, 36, 37, 38, 53, 62, 66, 67, 75, 78, 83, 84, 90, 102, 103]. Most of these studies aimed to improve the data locality of database algorithms for reducing the *number* of cache misses. In this thesis, we investigate a different approach: reducing the *impact* of cache misses by overlapping cache miss latencies with useful computations through a technique called *cache prefetching*. Combined with data locality optimizations, this technique enables larger freedom in redesigning core database structures and algorithms, leading to better CPU cache performance.

Let us begin by understanding the challenges of optimizing CPU cache performance. At first glance, this new cache-to-memory gap may look similar to the familiar memory-to-disk gap. Main memory is used as a buffer cache for disk, while hardware SRAM caches are used as caches for DRAM main memory. Data is transferred in units of a certain size: a disk page for the memory-to-disk gap, and a cache line for the cache-to-memory gap. When SRAM caches or DRAM main memory are full and new data is requested, some cache line or memory page has to be replaced to make room for the new data. Because of this similarity, a question immediately arises: Can we simply adapt disk optimization techniques to the cache-to-memory gap and solve the entire problem?

1.1 Can We Simply Adapt Memory-to-Disk Techniques?

Because of the similarity between the two gaps, some disk optimization techniques indeed can be applied to the cache-to-memory gap. An example is the partitioning technique for the database hash join algorithm. In order to avoid expensive random disk accesses, the hash join algorithm divides its working set into pieces (a.k.a. partitions) that fit into main memory using an I/O partitioning technique [54, 59, 74, 89, 101]. Similarly, it is also a good idea to fit the working set of a program into the CPU cache, and therefore the cache-to-memory counterpart to the I/O partitioning technique can effectively avoid random memory accesses for good hash join cache performance [14, 66, 90].

However, a close examination reveals a lot of differences between the two gaps as shown in Table 1.1, several of which have fundamental impacts on the designs of optimization techniques. Moreover, the optimization targets are more complex. We cannot simply optimize for a single gap (the cache-to-memory gap) because both the cache-to-memory gap and the memory-to-disk gap are important for commercial database systems. Therefore, adapting memory-to-disk techniques does not provide a *full* solution to the cache-to-memory problem. We discuss the two reasons at length in the following.

- **The differences between the two gaps present new challenges and opportunities.** Although database systems have full control of the main memory buffer pool, CPU caches are typically managed by hardware. This is because CPU caches are performance critical from the viewpoint

Table 1.1: Comparing the Cache-to-Memory Gap and the Memory-to-Disk Gap.

Feature	Largest SRAM Cache	DRAM Main Memory
Capacity	1-10 MB	1 GB - 1 TB
Transfer Unit Size	32-128B	4-64 KB
Miss Latency	10^2-10^3 cycles	10^6-10^7 cycles
Associativity	4-16 way set associative	fully associative
Replacement Policy	variants of LRU in each set	variants of LRU
Management	hardware	software

Note: Data sources include processor manuals and textbooks [39, 44, 46, 50, 81, 91, 96].

of processors; even the largest cache is designed to be very close to the processor (e.g., a 12-cycle access latency on Itanium 2 [44]). Therefore, CPU caches cannot afford to support sophisticated software strategies such as previous proposals of data replacement techniques for main memory buffer pools [21, 48, 76].

Given the hardware replacement policy (typically some variants of LRU within a set [39]), whether a data item is in the cache is determined *implicitly* by the memory accesses seen at the cache. Although we may expect some useful data items to stay in the cache when designing an algorithm, they may well be evicted from the cache because of cache pollution, e.g., from streaming through a large amount of read-once data, or because the processor performs activities other than the target algorithm, such as executing other procedures for the same database query or running different threads for other queries. Therefore, robustness to such cache interference is a desirable property of an optimization technique.

Apart from the difference in cache management, another major difference between the cache-to-memory and the memory-to-disk gap is the extremely different transfer unit sizes: 32-128B cache lines vs. 4-64 KB disk pages. Naively, one might regard the transfer unit size simply as an adjustable parameter to optimization techniques. However, it has more profound impacts because of its relative size to the data stored. For example, if index node size is chosen to be the transfer unit size for good performance (as in B$^+$-Trees [81, 84, 91]), the resulting structures at the cache granularity have much smaller node fanouts and much higher trees. Realizing that accessing every level of a tree index incurs an expensive cache miss, researchers proposed various techniques to increase the fanout of index nodes for better cache performance [13, 53, 83, 84].

Moreover, the relationship between database records and transfer units changes. Note that database records can be 100 bytes or larger (as evidenced by the TPC benchmark setups [98]). Therefore, records can no longer be stored *inside* a transfer unit at the CPU cache granularity, but rather *span* one or a few transfer units. From our experience, understanding this subtlety is often important in designing a good solution and in analyzing the cache behavior of a program.

- **Commercial database systems require both good cache and good disk performance.** There are two primary types of relational database systems: traditional database systems and main memory database systems. The latter assume the entire database resides in main memory, which has been an important research topic [25, 57, 58] and led to commercial products such as TimesTen [97], and research systems such as Dalí [47] and Monet [71]. However, the leading commercial database systems including Oracle [77], IBM DB2 [41], and Microsoft SQL Server [70], all follow the structure of traditional database systems [3, 95]: Data is stored on disk and loaded into the main memory buffer pool before being processed. In such a database system, both the cache-to-memory gap and the memory-to-disk gap may become the performance bottleneck depending on system configurations and database workloads. Therefore, it is important to optimize for both good CPU cache performance and good disk performance. Achieving this goal often presents more challenges than putting together disk and cache optimization techniques, as evidenced by the studies that optimize data page layout for better CPU cache performance [1, 36, 82],

Summarizing the above discussions, we point out that optimizing the cache-to-memory gap is not as simple as an exercise of applying the memory-to-disk techniques. New challenges and new opportunities arise in optimizing the CPU cache performance for database systems.

1.2 Cache Optimization Strategies

After analyzing the challenges in optimizing CPU cache performance, we describe the general optimization strategies in this section. Our purpose is to provide a framework to clarify the position of our approach in the entire schemes towards bridging the cache-to-memory gap.

There are two general optimization strategies for improving CPU cache performance of a program: reducing the *number* of cache misses and reducing the *impact* of cache misses by overlapping cache misses with computations and other misses.

1.2.1 Reducing the *Number* of Cache Misses

The first general strategy is to remove cache misses of a program by improving data reference locality of the program. The idea is to fit the working set of the program into the CPU cache, thus reducing the need to load data from main memory. One way to achieve this goal is to change the order in which data items are visited in the program so that references to the same data item occur closely in time (*temporal locality*) and references to multiple data items from the same cache line occur closely in time (*spatial locality*) without incurring any additional cache misses. A well-known example is blocked matrix multiplication, which works on sub-blocks of matrices for good cache performance [39]. An alternative way to improve data reference spatial locality is to modify the placement of data items in a program by packing into the same cache lines data items that are used closely in time. This may require changes ranging from reordering fields in data structure definitions, to introducing preprocessing steps such as the partitioning step for hash joins [14, 66, 90], as discussed previously in Section 1.1. Moreover, compression techniques [13, 31, 53, 104, 83, 84] trade off the processor's processing power for more compact data representations, thus helping reduce the working set of a program.

In terms of the types of cache misses, the above discussion focuses on reducing *capacity* misses, i.e. cache misses because of a program's working set exceeding cache capacity. This is often a major cause for poor cache performance (as in matrix multiplication and hash joins). Cache misses can also result from accessing data for the first time (*cold* misses) and from the limited set associativities of CPU caches (*conflict* misses)[1] . Conflict misses are usually less of a problem when the set associativities are 8 or higher [39], which is common for high-end processors [50, 44], and the trend is to support higher associativities. However, cold misses cannot be easily removed by definition (except that compression techniques might help).

1.2.2 Reducing the *Impact* of Cache Misses

The second general strategy for optimizing CPU cache performance is to make cache misses less expensive by exploiting memory system parallelism. Modern processors allow multiple outstanding cache misses to be in flight simultaneously within the memory hierarchy. For example, the Itanium 2 system bus control logic has an 18-entry out of order queue, which allows for a maximum of 19 memory requests to be outstanding from a single Itanium 2 processor [44]. Modern processors support several mechanisms that exploit this parallelism:

- **Instruction Speculation**. In many modern processors, the hardware attempts to overlap cache misses by speculating ahead in the instruction stream. While this mechanism is useful for hiding the latency of primary data cache misses that hit in the secondary cache, the number of instructions a processor can look ahead (a.k.a. instruction window size)[2] is far too small to fully hide the latency of cache misses to main memory [26].

- **Hardware-Based Cache Prefetching**. Modern processors (e.g., Intel Pentium 4 [46] and Itanium 2 [44]) can automatically load instructions and/or data from main memory before use, thus overlapping cache miss latencies with useful computations. However, hardware-based data prefetching techniques [4] rely upon recognizing regular and predictable (e.g., strided) patterns in the data ad-

[1] Data sharing (true sharing or false sharing) between multiple threads running on different processors in a multiprocessor system may cause *coherence* cache misses. In this thesis, we mainly focus on cache misses incurred by a single thread of execution.

[2] Intel Pentium 4 [12] has an 126-instruction reorder buffer. Intel Itanium 2 [44] has an eight-stage core pipeline with a 24-instruction buffer between stage 2 and 3. Since its issue rate is up to six instructions per cycle, there can be up to 72 instructions running in the pipeline simultaneously in Itanium 2. Note that the instruction window size is roughly one order of magnitude smaller than the instruction opportunities wasted per cache miss.

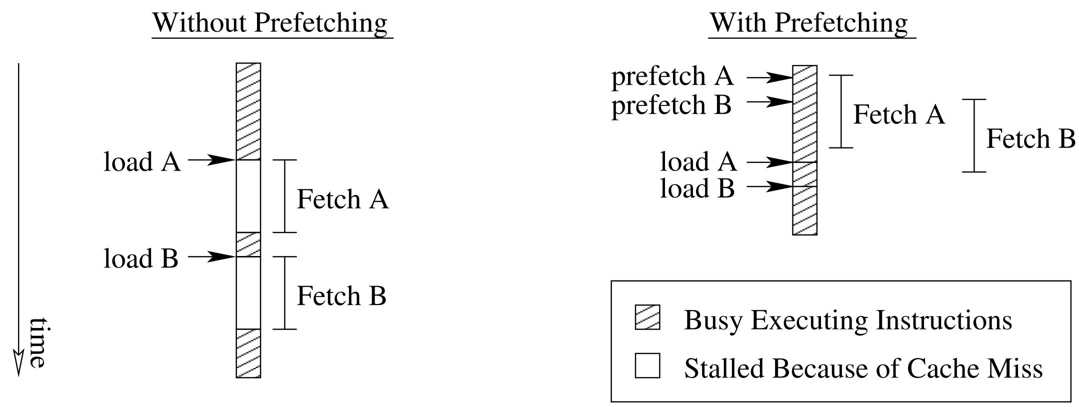


Figure 1.1: Illustration of the use of prefetch instructions to hide cache miss latencies.

dress stream, while many important database algorithms (e.g., searching a B^+-Tree index and visiting a hash table in hash joins) have less regular or rather random access patterns.

- **Software Cache Prefetching**. Modern processors (e.g., Intel Itanium 2 [44] and Pentium 4 [46]) provide *prefetch* instructions that allow software to influence the timing of cache misses. Figure 1.1 illustrates the use of prefetch instructions to hide cache miss latencies. As shown on the left, without prefetching, the processor stalls when it attempts to load two locations that are not present in the cache. As shown on the right, if we know the memory addresses of the loads early enough, we can issue prefetch instructions to read the data into the cache before the execution of the actual load instructions. In this way, the cache miss latencies can be overlapped with the execution of other useful instructions. Moreover, the first cache miss latency can be overlapped with the second. This is because the memory system can serve multiple cache misses simultaneously.

  Prefetch instructions can be regarded as light-weight, non-blocking memory load operations without register destinations. Such memory operations are performed in the background, and do not block the foreground executions. Software has the flexibility to decide what and when to prefetch. Therefore, software cache prefetching is potentially helpful for reducing the impact of any types of cache misses, including both capacity misses and cold misses, for handling different kinds of memory access patterns, and for improving all kinds of programs.

  A processor can support many different flavors of prefetch instructions. For example, both Itanium

2 and Pentium 4 support prefetch instructions to load data into different levels of caches [45, 43]. They also support special streaming prefetch instructions, which indicate that the prefetched data will be used only once and therefore should be loaded in a special way to reduce cache pollution. Moreover, Itanium 2 supports both faulting and non-faulting prefetches [45]. Non-faulting prefetches will be dropped if the virtual page table entry is missing from the TLB or if other exceptional conditions may result from executing the prefetches. In contrast, a faulting prefetch behaves more like a common load instruction; upon a TLB miss, it will load the page entry into the TLB table and continue. Therefore, faulting prefetches are specially useful if TLB misses are likely, e.g., when prefetching for random memory accesses.

- **Multiple Hardware Threads**. More and more processors support multiple hardware threads through simultaneous multithreading (SMT) [99] or single chip multiprocessors (CMP) [6, 34, 35] or a combination of the two (e.g., Intel Pentium 4 [44], IBM Power 5 [50], and Sun UltraSPARC IV [96]). When one hardware thread blocks because of a cache miss, the other hardware threads can be still running on the processor. However, it is a non-trivial task to break down a sequential program into multiple parallel threads in order to reduce the elapsed time of a single program, as evidenced by a recent study to utilize a multi-threaded and multi-core network processor for query processing [28]. Even if assuming hardware support for speculating threads [92] so that correctness is guaranteed, it still requires significant efforts to minimize data sharing across threads in order to achieve good performance [22]. Without these efforts, simply running a program on an SMT or CMP processor does not necessarily speed up the program, rather the program tends to slow down because of cache interference from other threads running on the same processor.

The two general optimization strategies described above are complementary. The first strategy is important because it reduces memory bandwidth requirement. The second strategy is important because a significant number of misses (e.g., cold misses) often still exist after applying the first strategy. The two strategies constitute a framework for bridging the cache-to-memory gap: reducing as many cache misses as possible with the first strategy, then reducing the impact of the remaining cache misses with the second strategy.

1.3 Our Approach: Redesigning Database Systems in Light of CPU Cache Prefetching

With the general optimization strategies in mind, we see that cache prefetching (software cache prefetching[3]) is a promising technique for reducing the impact of cache misses. Cache prefetching has been shown to be effective for both array-based and pointer-based codes in scientific and engineering applications [64, 65, 72, 73]. However, it has not been systematically studied for database systems before. Therefore, in this thesis, we investigate cache prefetching for improving the CPU cache performance of database systems. Unlike most previous studies (as will be described in Section 1.4), we target traditional disk-oriented database systems, which are supported by all the leading database vendors and are widely used, rather than main memory databases. We redesign core database structures and algorithms by using novel techniques enabled and inspired by cache prefetching.

Since prefetching for sequential array accesses has been well studied before and is often supported by hardware-based prefetching [4], we are interested in studying non-contiguous access patterns in database systems. Such patterns are abundant in algorithms involving tree structures and hash tables. Therefore, we focus specially on the B$^+$-Tree index [7, 8] as a representative tree implementation, and the hash join algorithm [54, 89] as a representative algorithm using hash tables. We re-examine their designs with cache prefetching in mind, and combine cache prefetching and data locality optimizations to achieve good cache performance. In the following, we describe our approaches to improving the B$^+$-Tree index and the hash join algorithm in more detail.

- *B$^+$-Trees*. B$^+$-Trees [7, 8] are used extensively for fast associative lookups throughout database systems. Compared to a binary tree, a B$^+$-Tree is a multi-way search tree. Our goal for studying B$^+$-Trees is to improve the performance of B$^+$-Trees in commercial database systems in light of prefetching. We achieve this goal in two steps. In the first step, we focus purely on the CPU cache performance of B$^+$-Trees within a main memory database environment. With the understandings obtained from the first step, we then optimize B$^+$-Trees in a traditional database system with disk

[3]Throughout the thesis, we use cache prefetching or prefetching as short terms for software cache prefetching unless otherwise noted.

I/Os in the picture. Our goal in the second step is to achieve good cache performance when most of the B$^+$-Tree is in the main memory buffer pool, and still maintain good disk performance when the B$^+$-Tree is mainly on disk. Moreover, it is interesting to analyze whether the techniques developed in the first step for the cache-to-memory gap are also applicable to the memory-to-disk gap, which will provide a deeper understanding of the differences between the two gaps.

- *Hash Joins*. Commercial hash join algorithms consist of two phases: the I/O partitioning phase and the join phase [54, 59, 74, 89, 101]. The I/O partitioning phase divides the input relations into memory-sized partitions to avoid random disk accesses because of hashing. Our goal for studying hash joins is to improve the performance of both partitioning and join phases of the hash join algorithm in commercial database systems. We also take two steps in this study. Our first step is to exploit cache prefetching for the two phases *separately*. The major challenge is to effectively issue prefetch instructions despite the random access patterns of hash table visits. Our second step is to take advantage of the two-phase structure of the hash join algorithm for further optimizing its performance. Moreover, because multiple processors can join multiple memory-sized partitions simultaneously, hash join is a good workload for studying the impact of multiple processors on the performance of prefetching algorithms.

We implement our optimization algorithms and measure the actual performance of working codes both on an Itanium 2 machine and on a simulation platform to better understand the cache behaviors of the algorithms.

1.4 Related Work

In this section, we present related work to B$^+$-Tree cache performance and hash join cache performance. Please note that some of the studies we discuss appeared after our original publications [16, 17, 18, 19] in the completion of this thesis. In the following, we mainly describe the high-level ideas. We will perform more detailed experimental comparisons between previous techniques and our proposals in Chapter 2-5.

1.4.1 Related Work on B^+-Trees

A B^+-Tree [7, 8] is a multi-way search tree, where each leaf node contains multiple index entries, and each non-leaf node has multiple children. A B^+-Tree is a balanced tree structure; all the leaf nodes occur at the same level. Leaf nodes are connected through sibling links into key order to facilitate the retrieval of a range of index entries. A B^+-Tree is also a *dynamic* index structure; it supports insertions and deletions efficiently without the need to re-build the tree periodically.

Main Memory B^+-Trees. Chilimbi, Hill, and Larus demonstrated that B^+-Trees with cache line sized nodes can outperform binary trees for memory-resident data on modern processors [20]. Likewise, B^+-Trees outperform index structures specially designed for main memory databases (in the first thrust of main memory database research assuming uniformly fast memory accesses), *i.e.* T-trees [57], on today's processors [83].

Main memory B^+-Trees have cache-line-sized nodes with small fanouts (typically 8 compared to several hundred for disk-oriented B^+-Trees with disk-page-sized nodes). Therefore, the trees are very high (e.g., 8 levels when there are 10 million 4-byte keys in trees with 64-byte nodes). Realizing that the number of expensive cache misses in a search is roughly proportional to the height of the tree, researchers aimed to improve the fanout of cache-line-sized nodes to reduce the tree levels, thus reducing the *number* of expensive misses. Rao and Ross proposed "Cache-Sensitive Search Trees" (CSS-Trees) [83] and "Cache-Sensitive B^+-Trees" (CSB^+-Trees) [84] that restrict the data layout of sibling nodes so that all (or nearly all) of the child pointers can be eliminated from the parent. This saves space in non-leaf nodes for more index entries. Bohannon, Mcllroy, and Rastogi proposed partial-key trees that store compressed keys in indices for larger fanouts [13].

Beyond improving the performance of a single search operation, researchers have studied how to improve the throughput of a large number of back-to-back searches. Zhou and Ross proposed to buffer accesses at every non-root tree node so that multiple accesses may share the cache miss of visiting a node [102]. Moreover, Cha *et al.* [15] proposed an optimistic scheme for B^+-Tree concurrency control in main memory.

CPU Cache Performance of Disk-Oriented B$^+$-Trees. In commercial database systems, B$^+$-Tree nodes are disk pages, typically 4-64KB large. Binary searches in such a large node can have very poor spatial locality; the first several probes in a binary search always access a single key out of an entire cache line, then discard the rest of the line. To reduce the *number* of cache misses of searches, Lomet described an idea, intra-node micro-indexing, in his survey of B$^+$-Tree page organization techniques [62]. It places a small array in a few cache lines of the page that indexes the remaining keys in the page. This small micro-index replaces the first several probes in a binary search, thus improving the spatial locality of a search. However, this scheme suffers from poor update performance. As part of future directions, Lomet [62] has advocated breaking up B$^+$-Tree disk pages into cache-friendly units, pointing out the challenges of finding an organization that strikes a good balance between search and insertion performance, storage utilization, and simplicity. Bender, Demaine, and Farach-Colton presented a recursive B$^+$-Tree structure that is *asymptotically* optimal, regardless of the cache line sizes and disk page sizes, but assuming no prefetching [9].

1.4.2 Related Work on Hash Joins

Hash join [54, 59, 74, 89, 101] has been studied extensively over the past two decades. It is commonly used in today's commercial database systems to implement equijoins efficiently. In its simplest form, the algorithm first builds a hash table on the smaller (*build*) relation, and then probes this hash table using records of the larger (*probe*) relation to find matches. However, the random access patterns inherent in the hashing operation have little spatial or temporal locality. When the main memory available to a hash join is too small to hold the build relation and the hash table, the simplistic algorithm suffers excessive random disk accesses. To avoid this problem, the *GRACE* hash join algorithm [54] begins by dividing the two joining relations into smaller intermediate partitions such that each build partition and its hash table can fit within main memory; pairs of memory-sized build and probe partitions are then joined separately as in the simple algorithm. This *I/O partitioning* technique limits the random accesses to objects that fit within main memory and results in nice predictable I/Os for both source relations and intermediate partitions.

A technique similar to I/O partitioning, called cache partitioning was proposed to avoid random memory accesses, thus reducing the *number* of cache misses. *Cache partitioning*, in which the joining relations are partitioned such that each build partition and its hash table can fit within the (largest) CPU cache, has been shown to be effective in improving performance in *memory-resident and main-memory* databases [14, 66, 90]. Shatdal, Kant, and Naughton showed that cache partitioning achieves 6-10% improvement for joining memory-resident relations with 100B tuples [90]. Boncz, Manegold, and Kersten proposed using multiple passes in cache partitioning to avoid cache and TLB thrashing when joining vertically-partitioned relations (essentially joining two 8B columns) in the Monet main memory database environment [14, 66]. The generated join results are actually a join index [100] containing pointers to matching pairs of records stored in vertical partitions. In order to efficiently extract the matching records, Manegold *et al.* proposed a cache conscious algorithm, called Radix-Decluster Projection that performs sophisticated locality optimizations [67].

In summary, most related work on CPU cache performance of B^+-Trees and hash joins focused on reducing the *number* of cache misses in *main memory* database environments. Very few studies tried to address their CPU cache performance in disk-oriented databases, and those that do exist did not provide extensive experimental evaluations of the proposed ideas. In contrast, we investigate cache prefetching to reduce the *impact* of cache misses, we target traditional disk-oriented database systems, which are supported by all the leading database vendors, and we present detailed performance studies of all our proposed techniques.

1.5 Contributions

The primary contributions of this thesis are the following:

- The first study that reduces the *impact* of cache misses for B^+-Trees and hash joins by exploiting cache prefetching. In addition to inserting prefetches, we redesign algorithms and data structures to make prefetching effective. Unlike most previous studies on B^+-Tree and hash join cache performance, we target traditional *disk-oriented* database systems rather than *main memory* databases because traditional disk-oriented databases are widely supported and used. For B$^+$-Trees, we address both the cache-to-memory gap and the memory-to-disk gap in the disk-oriented environment. For hash joins, we demonstrate that disk-oriented hash joins are CPU-bound with reasonable I/O bandwidth. Therefore, we mainly focus on bridging the cache-to-memory gap.

- The proposal and evaluation of a novel main memory index structure, *Prefetching B$^+$-Trees*, which uses cache prefetching to accelerate two major access patterns of B$^+$-Tree indices in the pure main memory environment: searches and range scans. Our solution has better search performance, better range scan performance, and comparable or better update performance over B$^+$-Trees with one-cache-line nodes. Moreover, we achieve better performance than CSB$^+$-Trees [84], and we show that CSB$^+$-Trees and our prefetching scheme are complementary.

- The proposal and evaluation of a novel index structure, *Fractal Prefetching B$^+$-Trees*, that optimizes index operations both for cache performance and for disk performance. We propose two different implementations of this index structure, a disk-first implementation and a cache-first implementation. Experimental results show that the disk-first implementation achieves the goals while the cache-first implementation may incur large disk overhead. We also study the effects of employing the prefetching techniques proposed in pure main memory environments for optimizing disk performance.

- The proposal and evaluation of two prefetching techniques, *Group Prefetching and Software-Pipelined Prefetching*, that exploit inter-tuple parallelism for overlapping cache misses incurred in processing a tuple across the processing of multiple tuples. Experimental results show that our techniques achieve dramatically better performance over cache partitioning and original hash joins. Moreover, our techniques are more robust than cache partitioning when there are concurrent activities in the computer system.

- The proposal and evaluation of a novel hash join algorithm, *Inspector Joins*, that exploits the free information obtained from one pass of the hash join algorithm to improve the performance of a later pass. We propose a specialized index that addresses the memory bandwidth sharing problem, and can take advantage of nearly-sorted relations. Moreover, we utilize cache prefetching to improve the robustness of Inspector Joins in the face of cache interference. Furthermore, we present an illustrative example of how Inspector Joins can use its collected statistics to select between two join phase algorithms for the given query and data. Finally, our experiments demonstrate significant performance improvement of Inspector Joins over previous state-of-the-art cache prefetching and cache partitioning algorithms.

1.6 Thesis Organization

Chapter 2 investigates cache prefetching techniques for improving B^+-Trees in main memory environments. This is the first step in our B^+-Tree study. We describe our solution for improving two major access patterns of B^+-Trees: searches and range scans. For searches, we present a novel scheme to avoid predicting and prefetching child nodes. For range scans, we solve the pointer chasing problem. We compare the performance of our solution against CSB^+-Trees [84]. We also combine our prefetching techniques with CSB^+-Trees to understand the interactions of the two schemes.

Chapter 3 describes our design of a single index structure, Fractal prefetching B+-Trees, that achieves both good cache performance and good disk performance. The basic idea is to embed B^+-Trees optimized purely for cache performance into B^+-Trees optimized purely for disk performance. Our cache-optimized B^+-Trees are based on the study in Chapter 2. This embedding process usually leads to large overflow or large underflow of disk pages. We describe two schemes, a disk-first scheme and a cache-first scheme, to solve this problem. We evaluate the performance of our solutions against B^+-Trees and micro-indexing. Moreover, we discuss the implications of applying the same prefetching techniques from Chapter 2 for improving memory-to-disk performance.

Chapter 4 exploits cache prefetching for improving hash join performance. This is the first step in our hash join study. The major difficulty in employing prefetching is that it is impossible to generate addresses for hash table accesses early enough for effective prefetching. We solve this problem by taking advantage of the large number of records to be joined and exploiting the inter-tuple parallelism.

We describe two techniques, group prefetching and software-pipelined prefetching, for improving the I/O partitioning phase and the join phase performance. Our experimental results first show that hash joins are CPU bound in a balanced server system. We then compare the performance of our solution against cache partitioning and original hash joins.

Chapter 5 first studies the impact of memory bandwidth sharing on the existing cache-friendly hash join algorithms. We see that the performances of these algorithms degrade significantly when multiple (8 or more) processors are eagerly competing for the memory bandwidth in a shared-bus multiprocessor system. To cope with this problem, we exploit the two-phase structure of the hash join algorithm. We describe our new algorithm, Inspector Joins, that examines the data in the I/O partitioning phase almost for free to generate a help structure, and uses it to accelerate the join phase of the algorithm. We describe how to combine locality optimizations and cache prefetching to achieve good performance with lower memory bandwidth requirements. Our experimental results compare Inspector Joins against the state-of-the-art cache-friendly hash join algorithms.

Finally, Chapter 6 contains a summary of the important results in this thesis, and discusses their implications.

Chapter 2

Exploiting Cache Prefetching for Main Memory B^+-Trees

2.1 Introduction

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

To illustrate the need for improving the cache performance of both search and scan on B^+-Tree indices, Figure 2.1 shows a breakdown of their simulated performance on a state-of-the-art machine. For the sake of concreteness, we pattern the memory subsystem after the Itanium 2 [44]—details are provided later in Section 2.4. The "search" experiment in Figure 2.1 looks up 10,000 random keys in a main-memory B^+-Tree index after it has been bulkloaded with 10 million keys. The "scan" experiment

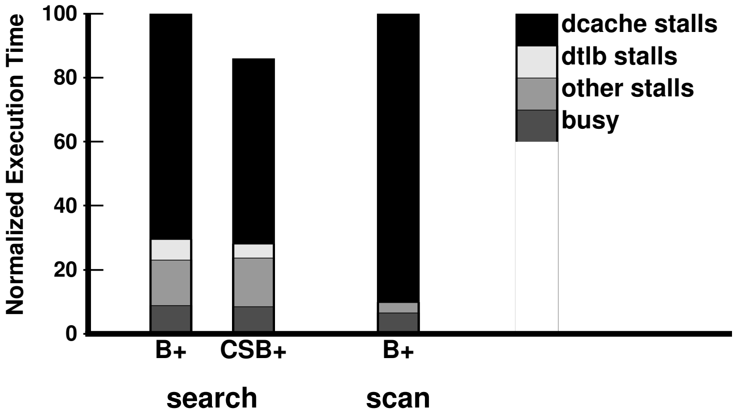


Figure 2.1: Execution time breakdown for index operations (**B+** = B^+-Trees, **CSB+** = CSB^+-Trees).

performs 100 range scan operations starting at random keys, each of which scans through 1 million ⟨`key`, `tupleID`⟩ pairs retrieving the `tupleID` values. (The results for shorter range scans—e.g., 1000 tuple scans—are similar). The B^+-Tree node size is equal to the cache line size, which is 64 bytes. Each bar in Figure 2.1 is broken down into four categories: busy time, data cache stalls, data TLB stalls, and other stalls. Both search and scan accesses on B^+-Tree indices (the bars labeled "**B+**"—we will explain the "**CSB+**" bar later) spend a significant fraction of their time—70% and 90%, respectively—stalled on data cache misses. Hence there is considerable room for improvement.

2.1.1 Previous Work on Improving the Cache Performance of Indices

In an effort to improve the cache performance of index *searches* for main-memory databases, Rao and Ross proposed two new types of index structures: "Cache-Sensitive Search Trees" (CSS-Trees) [83] and "Cache-Sensitive B^+-Trees" (CSB^+-Trees) [84]. The premise of their studies is the conventional wisdom that the optimal tree node size is equal to the *natural data transfer size*, which corresponds to the *disk page size* for disk-resident databases and the *cache line size* for main-memory databases. Because cache lines are roughly two orders of magnitude smaller than disk pages (e.g., 64 bytes vs. 4 Kbytes), the resulting index trees for main-memory databases are considerably deeper. Since the number of expensive cache misses is roughly proportional to the height of the tree, it would be desirable to somehow increase the effective fanout (also called the *branching factor*) of the tree, without paying the cost of additional cache misses that this would normally imply.

To accomplish this, Rao and Ross [83, 84] exploited the following insight: By restricting the data layout such that the location of each child node can be directly computed from the parent node's address (or a single pointer), we can eliminate all (or nearly all) of the child pointers. Assuming that keys and pointers are of the same size, this effectively doubles the fanout of cache-line-sized tree nodes, thus reducing the height of the tree and the number of cache misses. CSS-Trees [83] eliminate all child pointers, but do not support incremental updates and therefore are only suitable for read-only environments. CSB^+-Trees [84] do support updates by retaining a single pointer per non-leaf node that points to a contiguous block of its children. Although CSB^+-Trees outperform B^+-Trees on searches, they still perform significantly worse on updates [84] due to the overheads of keeping all children for a given node in sequential order within contiguous memory, especially during node splits.

Returning to Figure 2.1, the bar labeled "**CSB+**" shows the execution time of CSB^+-Trees (normalized to that of B^+-Trees) for the same index search experiment. As we see in Figure 2.1, CSB^+-Trees eliminate 18% of the data cache stall time, thus resulting in an overall speedup[1] of 1.16 for searches. While this is a significant improvement, over half of the remaining execution time is still being lost to data cache misses; hence there is significant room for further improvement. In addition, these search-oriented optimization techniques provide no benefit to *scan* accesses, which are suffering even more from data cache misses.

2.1.2 Our Approach: Prefetching B^+-Trees

In this chapter, we propose and study *Prefetching B^+-Trees* (pB^+-Trees), which use cache prefetching to reduce the amount of exposed cache miss latency. Tree-based indices such as B^+-Trees pose a major challenge for prefetching search and scan accesses since both access patterns suffer from the *pointer-chasing problem* [64]: The data dependencies through pointers make it difficult to prefetch sufficiently far ahead to limit the exposed miss latency. For index *searches*, pB^+-Trees address this problem by

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

having wider nodes than the natural data transfer size, e.g., eight vs. one cache lines. These wider nodes reduce the height of the tree, thereby decreasing the number of expensive misses when going from parent to child. The key observation is that by using cache prefetching, the wider nodes come almost for free: all of the cache lines in a wider node can be fetched almost as quickly as the single cache line of a traditional node. To accelerate index *scans*, we introduce arrays of pointers to the B^{+}-Tree leaf nodes which allow us to prefetch arbitrarily far ahead, thereby hiding the normally expensive cache misses associated with traversing the leaf nodes within the range. Of course, indices may be frequently updated. Perhaps surprisingly, we demonstrate that update times actually *decrease* with our techniques, despite any overheads associated with maintaining the wider nodes and the arrays of pointers.

The remainder of this chapter is organized as follows. Sections 2.2 and 2.3 discuss how prefetching B^{+}-Trees use cache prefetching to accelerate index searches and scans, respectively. To quantify the benefits of these techniques, we present experimental results on an Itanium 2 machine and on a simulation platform in Section 2.4. Finally, we discuss further issues and summarize the chapter in Sections 2.5 and 2.6, respectively.

2.2 Index Searches: Using Prefetching to Create Wider Nodes

A B^{+}-Tree search starts from the root. It performs a binary search in each non-leaf node to determine which child to visit next. Upon reaching a leaf node, a final binary search returns the key position. Regarding the cache behavior, we expect at least one expensive cache miss to occur each time we move down a level in the tree. Hence the number of cache misses is roughly proportional to the height of the tree (minus any nodes that might remain in the cache if the index is reused). Thus, having wider tree nodes for the sake of reducing the height of the tree might seem like a good idea. Unfortunately, in the absence of cache prefetching (i.e. when all cache misses are equally expensive and cannot be overlapped), making the tree nodes wider than the *natural data transfer size*—i.e. a cache line for main-memory databases—actually *hurts* performance rather than helps it, as has been shown in previous studies [83, 84]. The reason for this is that the number of additional cache misses at each node more than offsets the benefits of reducing the number of levels in the tree.

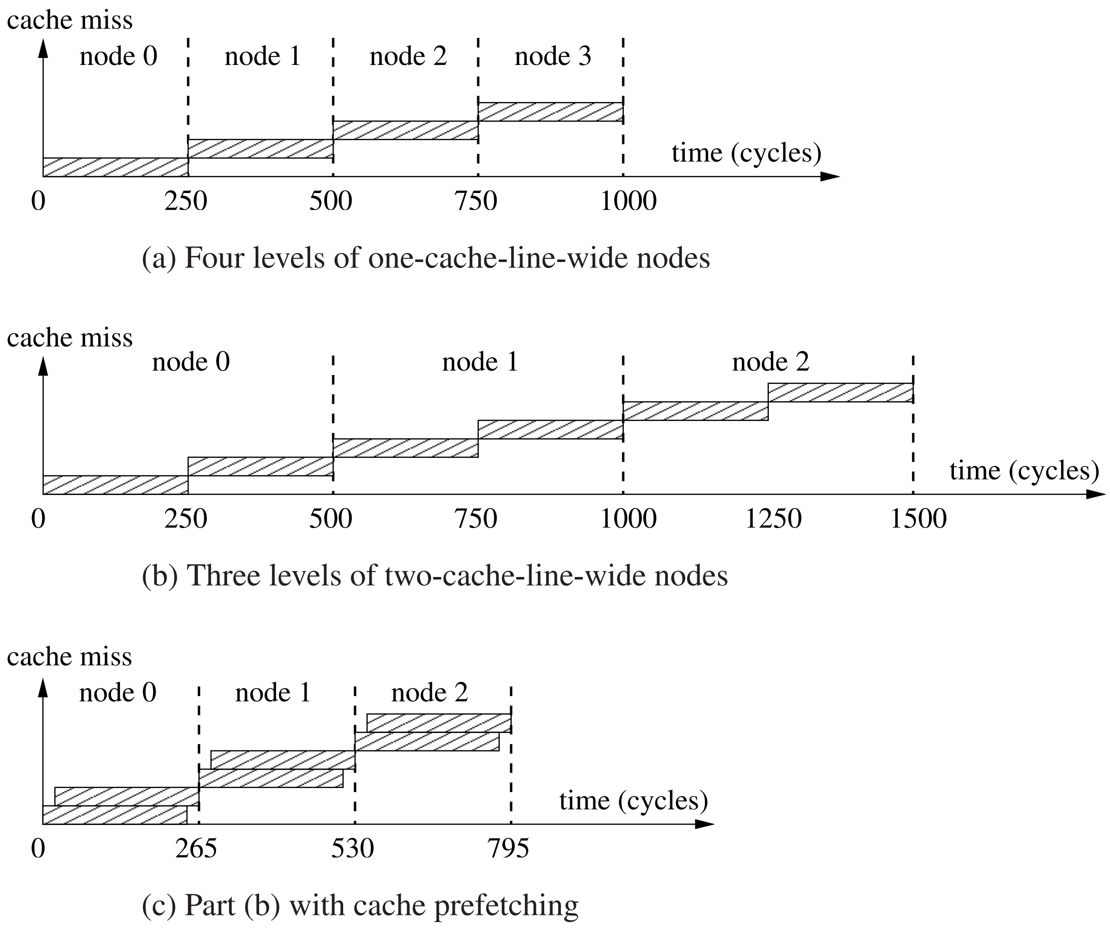


Figure 2.2: Performance of various B$^+$-Tree searches where a cache miss to memory takes 250 cycles, and a subsequent access can begin 15 cycles later (assuming no TLB misses for simplicity).

As a simple example, consider a main-memory B$^+$-Tree holding 1000 keys where the cache line size is 64 bytes and the `keys`, `child pointers`, and `tupleIDs` are all four bytes. If we limit the node size to one cache line, then the B$^+$-Tree will contain at least four levels. Figure 2.2(a) illustrates the resulting cache behavior. (In this example, we assume for simplicity there is no TLB miss . We will consider TLB misses in Section 2.2.2.) The four cache misses cost a total of 1000 cycles on our Itanium 2 machine model. If we double the node width to *two* cache lines, the height of the B$^+$-Tree can be reduced to *three* levels. However, as we see in Figure 2.2(b), this would result in *six* cache misses, thus increasing execution time by 50%.

With cache prefetching, however, it becomes possible to *hide* the latency of any miss whose address can be predicted sufficiently early. Returning to our example, if we prefetch the second half of each two-cache-line-wide tree node so that it is fetched in parallel with the first half—as illustrated in Figure 2.2(c)—we can achieve significantly *better* (rather than worse) performance compared with the one-cache-line-wide nodes in Figure 2.2(a). The extent to which the misses can be overlapped depends upon the implementation details of the memory hierarchy, but the trend is toward supporting greater parallelism. In fact, with multiple cache and memory banks and crossbar interconnects, it is possible to completely overlap multiple cache misses. Figure 2.2(c) illustrates the timing on our Itanium 2 machine model, where back-to-back misses to memory can be serviced once every 15 cycles, which is a small fraction of the overall 250 cycle miss latency. Thus, even without perfect overlap of the misses, we can still potentially achieve large performance gains (a speedup of 1.26 in this example) by creating wider than normal B^+-Tree nodes.

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

2.2.1 Modifications to the B^+-Tree Algorithm

We consider a standard B^+-Tree node structure: Each *non-leaf* node is comprised of some number f ($f \gg 1$) of `childptr` fields, $f-1$ `key` fields, and one `keynum` field that records the number of `keys` stored in the node (at most $f-1$). (All notation is summarized in Table 2.1.) Each *leaf* node is comprised of $f-1$ `key` fields, $f-1$ associated `tupleID` fields, one `keynum` field, and one `next-leaf` field that points to the next leaf node in key order. We consider for simplicity *fixed-size* `keys`, `tupleIDs`, and pointers. We also assume that `tupleIDs` and pointers are of the same size in main memory databases. Our first modification is to store the `keynum` and all of the `keys` prior to any of the pointers or `tupleIDs` in a node. This simple layout optimization allows the binary search to proceed without waiting to fetch all the pointers. Our search algorithm is a straightforward extension of the standard B^+-Tree algorithm, and we now describe only the parts that change.

Table 2.1: Terminology used throughout Chapter 2.

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

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

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

Deletion: We perform *lazy deletion* as in Rao and Ross [84]. If more than one key is in the node, we simply delete the key. When the last key in a node is deleted, we try to redistribute keys or delete the node. Since a search is also performed prior to a deletion, the entire root-to-leaf path is in the cache, and key redistribution is the only potential cause of additional misses. Therefore, before a key redistribution operation, we prefetch the sibling node from which keys are to be redistributed.

Cache prefetching can also be used to accelerate the *bulkload* of a B$^+$-Tree, which builds a B$^+$-Tree from scratch given a sorted array of $\langle$`key`, `tupleID`$\rangle$ pairs. However, because this operation is expected to occur infrequently, we focus instead on the more frequent operations of search, insertion and deletion.

2.2.2 Qualitative Analysis

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

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

Finally, an interesting consideration is to determine the optimal node size, given cache prefetching. Should nodes simply be as wide as possible? There are four system parameters that affect this answer: the latency of a full cache miss (T_1), the latency of an additional pipelined cache miss (T_{next}), the latency of a data TLB miss (T_{tlb}), and the size of the cache. As illustrated in Figure 2.3, the number of cache misses that can be served in parallel in the memory subsystem is equal to the latency of a full cache miss (T_1) divided by the additional time until another pipelined cache miss would also complete (T_{next}). We call this ratio (i.e. $\frac{T_1}{T_{\text{next}}}$) the *normalized bandwidth* (B). For example, in our Itanium 2 machine model, $T_1 = 250$ cycles, $T_{\text{next}} = 15$ cycles, and therefore $B = 16.7$. The larger the value of B, the greater the

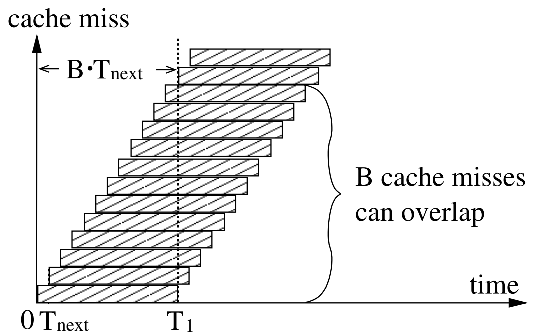


Figure 2.3: Computing the number of cache misses that can be served in parallel in the memory system.

system's ability to overlap parallel accesses, and therefore the greater likelihood of benefiting from wider index nodes. In general, we do not expect the optimal number of cache lines per node (w_{optimal}) to exceed B, since beyond that point we could have completed a binary search in a smaller node and moved down to the next level in the tree.

The third system parameter that affects the optimal node size is the latency of a data TLB miss. In our Itanium 2 machine model, $T_{\text{tlb}} = 30$ cycles. In the previous example, we ignored TLB misses for the sake of clearly explaining the wider node idea. In reality, every B$^+$-Tree node access is likely to incur a data TLB miss, for fetching the page table entry of the node in order to translate its virtual address to the physical address. The TLB can usually hold a small number of recently used page table entries. For example, in our Itanium 2 machine model, there is a 128-entry data TLB and the page size is 16KB. In other words, the TLB can cover at most 2MB memory. Therefore, B$^+$-Trees larger than 2MB are likely to suffer from TLB misses, even if the indices are used frequently. Moreover, if a B$^+$-Tree has not been used for a while, a cold search is likely to incur a TLB miss for every node on the path from root to leaf. Note that TLB misses make it more attractive to use wider nodes. In general, the larger the latency of TLB misses, the more benefit we can get by reducing the number of tree levels.

The last system parameter that potentially limits the optimal node size is the size of the cache, although in practice this does not appear to be a limitation given realistic values of B.

Optimal Node Size. Let us now consider a more quantitative analysis of the optimal node width w_{optimal}. We focus on the cost of memory accesses for computing the optimal node width because (i) this is the major cost of an index search, and (ii) the instruction cost of a search is $O(logN)$ *regardless* of the node width. A pB^{+}-Tree with N $\langle$`key`, `tupleID`$\rangle$ pairs contains at least h levels:

$$h = \left\lceil \log_f \left(\frac{N}{f-1} \right) + 1 \right\rceil, \text{ where } f = wm \tag{2.1}$$

We compute the total cost of a cold search assuming the tree is not used for a while, and no data related to the tree is cached in either the cache or in the TLB. The total cost of such a cold search is equal to the sum of the cost of visiting every node from root (level $h-1$) to leaf (level 0):

$$Totalcost = \sum_{l=0}^{h-1} cost(l) \tag{2.2}$$

After reading a `childptr`, a search can immediately start accessing the node on the next lower level. Therefore, with our data layout optimization of putting keys before `childptr`s, the cost of accessing a node depends on the position of the `childptr` to follow in the node. Let us denote the number of cache lines from the beginning of the node to the i-th `childptr` (or `tupleID`) as $lines(i)$. Then if the i-th `childptr` (or `tupleID`) is visited, the cost of accessing a node at level l is:

$$cost(l,i) = T_{\text{tlb}} + T_1 + (lines(i) - 1) \cdot T_{\text{next}} \tag{2.3}$$

For a full tree, a non-leaf node except the root node holds f `childptr`s, while a leaf node holds $f-1$ `tupleID`s. The number of children (r) of the root node is computed as follows:

$$r = \left\lceil \frac{N}{(f-1)f^{h-2}} \right\rceil \tag{2.4}$$

Then, the cost of accessing a node at level l is the average of all $cost(l,i)$ with different `childptr` positions:

$$cost(l) = \begin{cases} \frac{1}{r}\sum_{i=1}^{r} cost(l,i), & l = h-1 \\ \frac{1}{f}\sum_{i=1}^{f} cost(l,i), & l = 1,\ldots,h-2 \\ \frac{1}{f-1}\sum_{i=1}^{f-1} cost(l,i), & l = 0 \end{cases} \tag{2.5}$$

or

$$cost(l) = \begin{cases} T_{\text{tlb}} + T_1 + (\frac{1}{r}\sum_{i=1}^{r} lines(i) - 1) \cdot T_{\text{next}}, & l = h-1 \\ T_{\text{tlb}} + T_1 + (\frac{1}{f}\sum_{i=1}^{f} lines(i) - 1) \cdot T_{\text{next}}, & l = 1,\ldots,h-2 \\ T_{\text{tlb}} + T_1 + (\frac{1}{f-1}\sum_{i=1}^{f-1} lines(i) - 1) \cdot T_{\text{next}}, & l = 0 \end{cases} \tag{2.6}$$

If keys and pointers are of the same size, then the average of the `childptr` positions of a full node is approximately $\frac{3}{4}$ of the node. Therefore, the following equation holds approximately:

$$cost(l) \simeq T_{\text{tlb}} + T_1 + (\frac{3}{4}w - 1) \cdot T_{\text{next}}, \quad l < h - 1 \tag{2.7}$$

By using the above equations, we can compute the total cost of a cold search given the number of index entries and the width of a node. By computing the value of w that minimizes *totalcost*, we can find w_{optimal}. For example, in our simulations where $m = 8$, $T_1 = 250$, $T_{\text{next}} = 15$, and $T_{\text{tlb}} = 30$, $w_{\text{optimal}} = 8$ for trees with 10 million keys.

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

2.3 Index Scans: Prefetching Ahead Using Jump-Pointer Arrays

Given starting and ending keys as arguments, an index range scan returns a list of either the `tupleIDs` or the tuples themselves with keys that fall within this range. First the starting key is *searched* in the B$^+$-Tree to locate the starting leaf node. Then the scan follows the `next-leaf` pointers, visiting the leaf nodes in key order. As the scan proceeds, the `tupleIDs` (or tuples) are copied into a return buffer. This process stops when either the ending key is found or the return buffer fills up. In the latter case, the scan procedure pauses and returns the buffer to the caller (often a join node in a query execution plan), which then consumes the data and resumes the scan where it left off. Hence a range selection involves one key search and often multiple leaf node scan calls. Throughout this section, we will focus on range selections that return `tupleIDs`, although returning the tuples themselves (or other variations) is a straightforward extension of our algorithm, as we will discuss later in Section 2.5.

As we saw already in Figure 2.1, the cache performance of range scans is abysmal: 90% of execution time is being lost to data cache misses in that experiment. Figure 2.4(a) illustrates the problem: A full cache miss latency is suffered for each leaf node. A partial solution is to use the technique described in Section 2.2: If we make the leaf nodes multiple cache lines wide and prefetch each component of a leaf

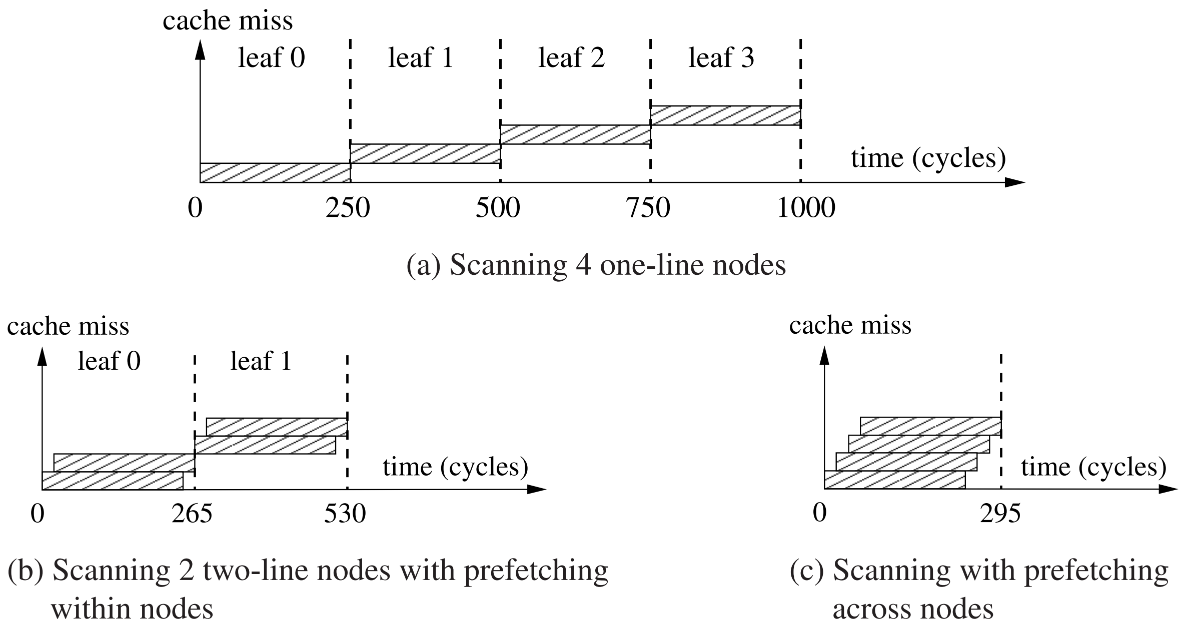


Figure 2.4: Cache behaviors of index range scans (assuming no TLB misses for simplicity).

node in parallel, we can reduce the frequency of expensive cache misses, as illustrated in Figure 2.4(b). While this is helpful, our goal is to *fully* hide the cache miss latencies to the extent permitted by the memory system, as illustrated in Figure 2.4(c). In order to achieve this goal, we must first overcome the *pointer-chasing problem*.

2.3.1 Solving the Pointer-Chasing Problem

Figure 2.5(a) illustrates the *pointer-chasing problem*, which was observed by Luk and Mowry [64, 65] in the context of prefetching pointer-linked data structures (i.e. linked-lists, trees, etc.) in general-purpose applications. Assuming that three nodes worth of computation are needed to hide the miss latency, then when node n_i in Figure 2.5(a) is visited, we would like to be launching a prefetch of node n_{i+3}. To compute the address of node n_{i+3}, we would normally follow the pointer chain through nodes n_{i+1} and n_{i+2}. However, this would incur the full miss latency to fetch n_{i+1} and then to fetch n_{i+2}, before the prefetch of n_{i+3} could be launched, thereby defeating our goal of hiding the miss latency of n_{i+3}.

Luk and Mowry proposed two solutions to the pointer-chasing problem that are applicable to linked

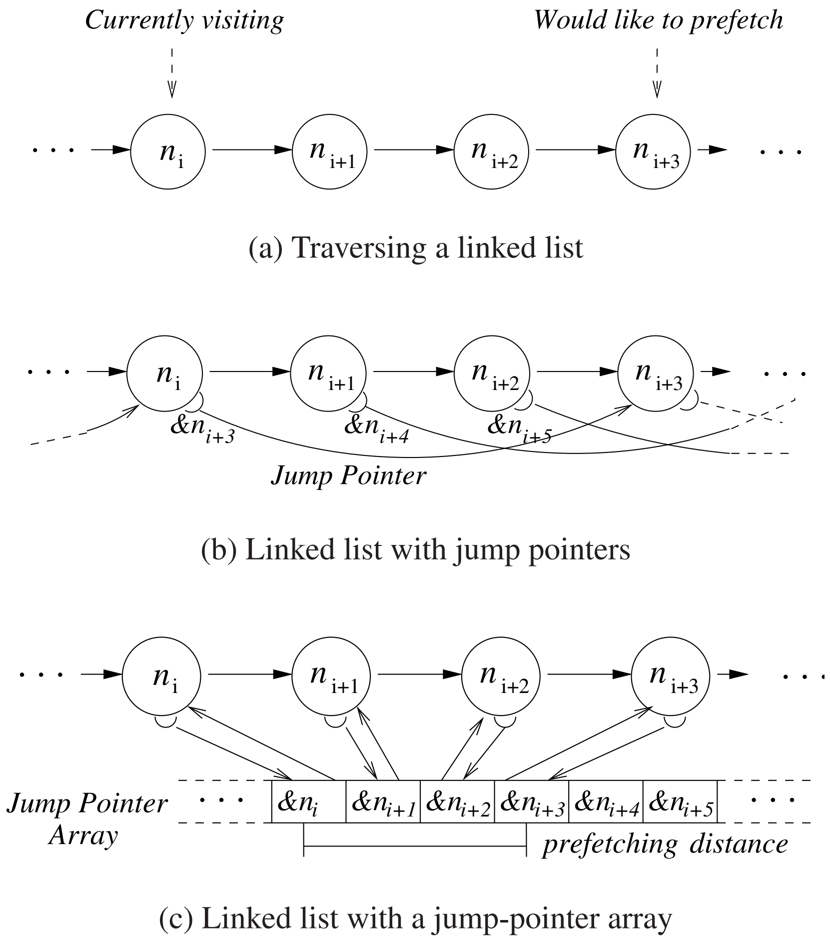


Figure 2.5: Addressing the pointer-chasing problem.

lists [64, 65]. The first scheme (*data-linearization prefetching*) involves arranging the nodes in memory such that their addresses can be trivially calculated without dereferencing any pointers. For example, if the leaf nodes of the B$^+$-Tree are arranged sequentially in contiguous memory, they would be trivial to prefetch. However, this will only work in read-only situations, and we would like to support frequent updates. The second scheme (*history-pointer prefetching*) involves creating new pointers—called *jump pointers*—which point from a node to the node that it should prefetch. For example, Figure 2.5(b) shows how node n_i could directly prefetch node n_{i+3} using three-ahead jump pointers.

In our study, we will build upon the concept of jump pointers, but customize them to the specific needs of B$^+$-Tree indices. Rather than storing jump pointers directly in the leaf nodes, we instead pull them out into a separate array, which we call the *jump-pointer array*, as illustrated in Figure 2.5(c). To initiate prefetching, a back-pointer in the starting leaf node is used to locate the leaf's position within the jump-pointer array; then based on the desired prefetching distance, an array offset is adjusted to find the address of the appropriate leaf node to prefetch. As the scan proceeds, the prefetching task simply continues to walk ahead in the jump-pointer array (which itself is also prefetched) without having to dereference the actual leaf nodes again.

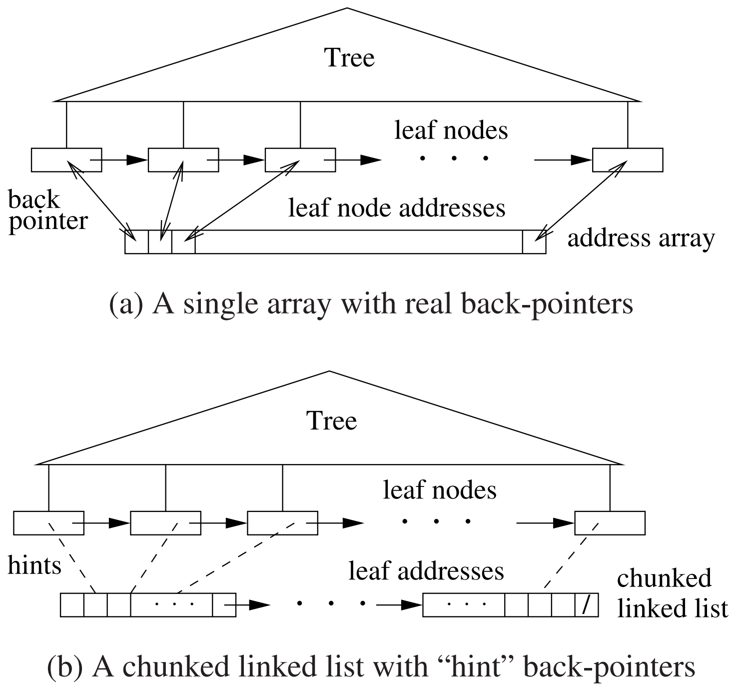
Jump-pointer arrays are more flexible than jump pointers stored directly in the leaf nodes. We can adjust the prefetching distance by simply changing the offset used within the array. This allows dynamic adaptation to changing performance conditions on a given machine, or if the code migrates to different machines. However, jump-pointer arrays do introduce additional space overhead. In Section 2.3.4, we will show that the space overhead is small. Then in Section 2.3.5, we propose to reuse internal B^+-Tree nodes to further reduce the overhead.

From an abstract perspective, one might think of the jump-pointer array as a single large, contiguous array, as illustrated in Figure 2.6(a). This would be efficient in read-only situations, but in such cases we could simply arrange the leaf nodes themselves contiguously and use *data-linearization prefetching* [64, 65]. Therefore a key issue in implementing jump-pointer arrays is to handle updates gracefully.

2.3.2 Implementing Jump-Pointer Arrays to Support Efficient Updates

Let us briefly consider the problems created by updates if we attempted to maintain the jump-pointer array as a single contiguous array as shown in Figure 2.6(a). When a leaf node is deleted, we can simply leave an empty slot in the array. However, insertions can be very expensive. When a new leaf node is *inserted*, an empty slot needs to be created in the appropriate position for the new jump pointer. If no nearby empty slots could be located, this could potentially involve copying a very large amount of data within the array in order to create the empty slot. In addition, for each jump-pointer that is moved within the array, the corresponding back-pointer from the leaf node into the array also needs to be updated, which could be very costly too. Clearly, we would not want to pay such a high cost upon insertions.

We improve upon the naïve contiguous array implementation in the following three ways. First, we



(a) A single array with real back-pointers

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

Figure 2.6: External jump-pointer arrays.

break the contiguous array into a chunked linked list—as illustrated in Figure 2.6(b)—which allows us to limit the impact of an insertion to its corresponding chunk. (We will discuss the chunk size selection later in Section 2.3.4).

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

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

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

2.3.3 Prefetching Algorithm

Having described the data structure to facilitate prefetching, we now describe our prefetching algorithm. The basic range scan algorithm consists of a loop that visits a leaf node on each iteration by following a `next-leaf` pointer. To support jump-pointer array prefetching, we add prefetches both prior to this loop (for the *startup* phase), and inside the loop (for the *steady-state* phase). Let d be the desired *prefetching distance*, in units of leaf nodes (we discuss below how to select d). During the startup phase, we issue prefetches for the first d leaf nodes.[2] These prefetches proceed in parallel, exploiting the available memory hierarchy bandwidth. During each loop iteration (i.e. in the steady-state phase), prior to visiting the current leaf node in the range scan, we prefetch the leaf node that is d nodes after the current leaf node. The goal is to ensure that by the time the basic range scan loop is ready to visit a leaf node, that node is already prefetched into the cache. With this framework in mind, we now describe further details of our algorithm.

First, in the startup phase, we must locate the jump pointer of the starting leaf node within the jump-pointer array. To do this, we follow the `hint` pointer from the starting leaf node to the chunk in the

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

chunked linked list and check to see whether the `hint` is precise—i.e. whether the `hint` points to a pointer back to the starting leaf node. If not, then we start searching within the chunk in both directions relative to the hint position until the matching position is found. As discussed earlier, the distance between the hint and the actual position appears to be small in practice.

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

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

2.3.4 Qualitative Analysis

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

Selecting the Prefetching Distance d. The *prefetching distance* (d, in units of nodes to prefetch ahead) is selected as follows. Normally this quantity is derived by dividing the expected worst-case miss latency by the computation time spent on one leaf node (similar to what has been done in other contexts [73]). However, because the computation time associated with visiting a leaf node during a range scan is quite small relative to the miss latency, we will assume that the limiting factor is the memory bandwidth. Roughly speaking, we can estimate this bandwidth-limited prefetching distance as

$$d = \left\lceil \frac{B}{w} \right\rceil, \tag{2.8}$$

where B is the normalized memory bandwidth and w is the number of cache lines per leaf node, as defined in Table 2.1. In practice, there is no problem with increasing d a bit to create some extra slack, because any prefetches that cannot proceed are simply buffered within the memory system. (Our sensitivity analysis in Section 2.4 will show that selecting a slightly larger prefetching distance results in similar range scan performance.)

Selecting the Chunk Size c. Chunks must be sufficiently large to ensure that we only need to prefetch one chunk ahead to hide the miss latency of accessing the chunks themselves. Recall that during the steady-state phase of a range scan, when we get to a new chunk, we immediately prefetch the next chunk ahead so that we can overlap its fetch time with the time it takes to prefetch the leaf nodes associated with the current chunk. Since the memory system only has enough bandwidth to initiate B cache misses during the time it takes one cache miss to complete, the chunks would clearly be large enough to hide the latency of fetching the next chunk if they contained at least B leaf pointers (there is at least one cache line access for every leaf visit). Each cache line of a full chunk can hold $2m$ leaf pointers (since there are only pointers and no keys). Chunks will be 50% full immediately after chunk splits, and each cache line of a chunk will hold m leaf pointers on average. In order to perform efficiently even in such situations, we estimate the minimum chunk size in units of cache lines as

$$c = \left\lceil \frac{B}{m} \right\rceil . \tag{2.9}$$

Another factor that could (in theory) dictate the minimum chunk size is that each chunk should contain at least d leaf pointers so that our prefetching algorithm can get sufficiently far ahead. However, since $d \leq B$ from equation (2.8), the chunk size in equation (2.9) should be sufficient. Increasing c beyond this minimum value to create some extra slack for more empty chunk slots does not hurt performance in practice, as our experimental results demonstrate later in Section 2.4.

Remarks. Given sufficient memory system bandwidth, our prefetching scheme hides the full memory latency experienced at every leaf node during range scan operations. With the data structure improvements in Section 2.3.2, we also expect good performance on updates.

However, there is a space overhead associated with the jump-pointer array. Because the jump-pointer array only contains one pointer per leaf node, the space overhead is relatively small. If keys and pointers are of the same size, a leaf node is as large as $2f$ pointers (f is defined in Table 2.1). Therefore, the

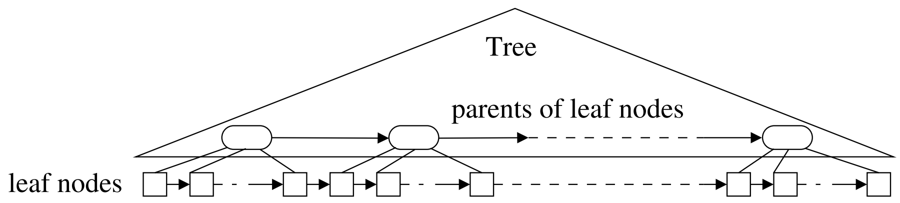


Figure 2.7: Internal jump-pointer arrays.

jump-pointer array only takes $\frac{1}{2f}$ as much space as all the leaf nodes. Given our technique described earlier in Section 2.2 for creating wider B$^+$-Tree nodes, the resulting increase in the fanout f will help reduce this overhead. However, if this space overhead is still a concern, we now describe how it can be reduced further.

2.3.5 Internal Jump-Pointer Arrays

So far we have described how a jump-pointer array can be implemented by creating a new *external* structure to store the jump pointers (as illustrated earlier in Figure 2.6). However, there is an existing structure *within* a B$^+$-Tree that already contains pointers to the leaf nodes, namely, the *parents* of the leaf nodes. The child pointers within a leaf parent node correspond to the jump pointers within a chunk of the external jump-pointer array described in Section 2.3.2. A key difference, however, is that there is no easy way to traverse these leaf parent nodes quickly enough to perform prefetching. A potential solution is to connect these leaf parent nodes together in leaf key order using linked-list pointers. (Note that this is sometimes done already for concurrency control purposes [56].)

Figure 2.7 illustrates the *internal* jump-pointer array. Note that leaf nodes do not contain back-pointers to their positions within their parents. It turns out that such pointers are not necessary for this internal implementation, because the position will be determined during the search for the starting key. If we simply retain the result of the leaf parent node's binary search, we will have the position to initiate the prefetching appropriately.

Maintaining a next pointer for a leaf parent node is fairly similar to maintaining a next pointer for a leaf node. Therefore, the bulkload, insertion and deletion algorithms can be easily obtained by extending the existing B^+-Tree algorithms. The search algorithm is the same, with the only difference being that the maximum number of pointers in a leaf parent node is reduced by one. The prefetching algorithm for range scan is similar to the one described earlier for external jump-pointer arrays, though we do not need to locate the starting leaf node within the jump-pointer array (because the position has already been recorded, as discussed above).

This approach is attractive with respect to space overhead, since the only overhead is one additional pointer per leaf parent node. The overhead of updating this pointer should be insignificant, because it only needs to be changed in the rare event that a leaf parent node splits or is deleted. Another benefit of this approach is that internal jump-pointer arrays are relatively easy to implement compared to external jump-pointer arrays because most of the maintenance work of the jump-pointer arrays is already supported through the B^+-Tree code.

One potential limitation of this approach, however, is that the length of a "chunk" in this jump-pointer array is dictated by the B^+-Tree structure, and may not be easily adjusted to satisfy large prefetch distance requirements.

In the remainder of this chapter, we will use the notations "p_eB^+-Tree" and "p_iB^+-Tree" to refer to pB^+-Trees with *external* and *internal* jump-pointer arrays, respectively.

2.4 Experimental Results

In this section, we present experimental results on an Itanium 2 machine and on a simulation platform. The simulation platform helps us better understand the cache behavior of our prefetching techniques for the following four reasons: (i) The simulator has extensive instrumentations and reports a large amount of useful information about the processor pipeline and the cache hierarchy; (ii) We have a clear understanding of how prefetch instructions are implemented in the simulator; (iii) We can easily flush caches in the simulator and study the cold cache behavior of a single search; (iv) It is easy to vary configuration parameters, such as the memory latency, to study the behavior of various techniques under potential settings in the future.

To facilitate comparisons with CSB$^+$-Trees, we present our experimental results in a *main-memory* database environment. We begin by describing the Itanium 2 machine configuration and our performance simulator. We then explain the implementation details of the index structures that we compare. The four subsections that follow present our experimental results for index searches, index range scans, updates, and operations on mature trees. Next, we present sensitivity analysis and a detailed cache performance study for a few of our earlier experiments. Finally, we study the impact of larger memory latency on our prefetching techniques.

2.4.1 Itanium 2 Machine Configuration

Throughout this thesis, we perform real machine experiments on an Itanium 2 machine unless otherwise noted. Table 2.2 lists the machine configuration parameters. Most of the information is readily available from the Linux operating system by checking the files on the "/proc" file system (e.g., "/proc/cpuinfo", "/proc/pal/cpu0/cache_info", "/proc/pal/cpu0/vm_info", and "/proc/meminfo") and by displaying version information of various commands. The machine has two 900MHz Itanium 2 processors, each with three levels of caches and two levels of TLBs. They share an 8 GB main memory. However, we only use a single CPU in our experiments unless otherwise noted. Itanium 2 supports both faulting and non-faulting prefetches [45], as described previously in Section 1.3. Since faulting prefetches will still succeed when incurring TLB misses, we use faulting prefetches ($lfetch.fault$) in our experiments.

The machine is running Linux 2.4.18 kernel with 16 KB virtual pages. There are two compilers available in the system: gcc [27] and icc [42]. (We will study the properties of the compilers through experiments and determine which compiler to use for evaluating B$^+$-Trees.) For gcc, we use its inline assembly support to insert prefetch assembly instructions into the source code. icc supports a special interface for prefetches in the form of a set of function calls; the compiler identifies such a call, and replaces it with a prefetch instruction. We use this interface for inserting prefetches. We obtain performance measurements through the perfmon library [79], which supports accesses to Itanium 2 performance counters. The measurements are for user-mode executions. We perform 30 runs for every measurement and report the average of the runs (except for measuring the latency and bandwidth parameters, which we explain

Table 2.2: Itanium 2 machine configuration.

CPU	dual-processor 900MHz Itanium 2 (McKinley, B3)
L1 Data Cache	16 KB, 64B lines, 4-way set-assoc., load lat. 1 cycle
L1 Instruction Cache	16 KB, 64B lines, 4-way set-assoc., load lat. 1 cycle
L2 Unified Cache	256 KB, 128B lines, 8-way set-assoc., load lat. 5 cycles
L3 Unified Cache	1.5 MB, 128B lines, 6-way set-assoc., load lat. 12 cycles
TLB	DTLB 1: 32 entries, fully-assoc.; ITLB 1: 32 entries, fully-assoc. DTLB 2: 128 entries, fully-assoc.; ITLB 2:128 entries, fully-assoc.
Level 2 Data TLB Miss Latency	32 cycles
Main Memory	8GB
Memory Latency (T_1)	189 cycles
Main Memory Bandwidth ($1/T_{\text{next}}$)	1 access per 24 cycles
Operating System	Linux 2.4.18 (Red Hat Linux Advanced Workstation release 2.1AW)
Page Size	16KB
Compiler	gcc: GNU project C and C++ Compiler Version 2.96 icc: Intel C++ Itanium Compiler Version 8.1
Performance Monitor Tool	kernel perfmon version: 1.0, pfmon version: 2.0

below). For the experiments in this section, all the standard deviations are within 5% of the averages. In fact, 90% of the experiments have their standard deviations within 1% of the averages.

For the rest of the configuration parameters that are not directly available, we determined their values through experimental measurements. Specifically, we measured the main memory latency, the main memory bandwidth, and the level 2 data TLB miss latency through experiments. To measure the main memory latency (T_1), we build a linked list, whose nodes are one-cache-line large and aligned on cache line boundaries. Then we perform a linked list traversal, and divide the total execution time by the number of nodes in the list to obtain a full cache miss latency (T_1). To make this measurement accurate, we perform the following three operations before an experiment: (i) randomly shuffling the nodes so that the CPU cannot easily guess the location of the next node and does experience a full cache miss at every node; (ii) flushing the CPU cache by reading a large piece of memory so that nodes are not cached;

and (iii) for each virtual memory page containing nodes of the list, keeping at least one cache line not allocated to the list and reading this cache line to warm up the TLB. (All the nodes are allocated from 64 pages so that the TLB capacity is not exceeded.)

To measure the TLB miss latency, we perform the same experiment except for the third operation in the above. Here, every node is allocated from a different virtual memory page and we do not warm up the TLB before the experiment. Therefore, every node access incurs a TLB miss and a full cache miss. By subtracting the full cache miss from the experimental result, we obtain the TLB miss latency. The measured TLB miss latency confirms the penalty listed in the Itanium 2 manual for a TLB miss that finds its page table entry in the L3 cache [44].

We will describe our experiment for measuring the memory bandwidth in Section 2.4.4 because it provides insights into the prefetching behaviors.

For the experiments that measure configuration parameters, we perform 30 runs, and take the *minimum* of the runs because a measurement can only be equal to or greater than the true latency value due to contentions (while we take *average* for B^+-Tree measurements to account for cache interference due to normal operating system activities).

2.4.2 Simulation Machine Model

Throughout this thesis, we use the same set of simulation parameters for our simulation study, as shown in Table 2.3. The memory hierarchy of the simulator is based on the Itanium 2 processor [44], while the simulator processor core pipeline is out-of-order dynamically scheduled, which models the design in most other modern processors, such as Intel Pentium 4 [12], IBM Power 5 [50], and Sun UltraSPARC IV [96]. The processor clock rate is 1.5 GHz. The simulator performs a cycle-by-cycle simulation, modeling the rich details of the processor including the pipeline, register renaming, the reorder buffer, branch prediction, and branching penalties, etc. The native integer size for the processor pipeline is 32 bits or 4 bytes. We set the integer divide latency based on our real measurement on the Itanium 2 machine. The simulator supports MIPS instructions. We generate fully-functional MIPS executables using version 2.95.2 of the gcc compiler with the “-O2” optimization flag and evaluate the CPU cache

Table 2.3: Simulation parameters.

Pipeline Parameters			
Clock Rate	1.5 GHz	Integer Multiply/Divide	4/50 cycles
Issue Width	4 instructions/cycle	All Other Integer	1 cycle
Reorder Buffer Size	128 instructions	Branch Prediction Scheme	gshare [69]
Functional Units	2 Integer, 1 Integer Divide, 2 Memory, 1 Branch, 2 FP		
Memory Parameters			
L1 Instruction Cache	16 KB, 4-way set-assoc.	Line Size	64 bytes
L1 Data Cache	16 KB, 4-way set-assoc.	Page Size	16 KB
Miss Handlers	32 for data, 2 for instruction	L1 Cache Access Latency	1 cycle
DTLB	128 entries, fully-assoc.	L2 Cache Access Latency	5 cycles
L2 Unified Cache	256 KB, 8-way set assoc.	L3 Cache Access Latency	12 cycles
L3 Unified Cache	2 MB, 8-way set assoc.	DTLB Miss Latency	30 cycles
L1-to-Memory Latency	250 cycles (plus any delays due to contention) ($T_1 = 250$)		
Memory Bandwidth	1 access per 15 cycles ($T_{\text{next}} = 15$)		

performance of user mode executions of all the algorithms that we study through detailed cycle-by-cycle simulations. Note that the simulator delivers system calls such as `read` and `write` directly to the underlying operating system without simulating the system calls. Therefore, we are able to run programs that perform I/O operations on the simulator and observe their performance of user mode executions.

Most of the memory parameters (e.g., cache sizes, associativities, cache access latencies) follow the configurations in the Itanium 2 processor, as described previously in Section 2.4.1. However, the simulator only supports a uniform cache line size across all levels of caches, while the Itanium 2 machine has two cache line sizes: 64 bytes for level 1 caches and 128 bytes for level 2 and 3 caches. We choose 64 bytes as the cache line size in our simulator.

The main memory latency parameter is based on our measurement on the Itanium 2 machine. Because the memory latency (T_1) is the number of clock cycles for loading an entire cache line, we set this value considering the processor clock rate and the cache line size in the simulator. To compute the main memory bandwidth parameter, we use the memory bandwidth supported by the Itanium 2 processor (6.4 GB/sec) [44] and take into account the clock rate and the cache line size. Moreover, the TLB miss latency is also based on our measurement on the real machine.

The simulator provides good support for cache prefetching. The simulator does not drop a prefetch when miss handlers are all busy and/or if it incurs a DTLB miss. This models the behavior of the faulting prefetch instruction ($lfetch.fault$) of Itanium 2 processor. Moreover, the simulator also supports special streaming prefetch instructions, which indicate that the prefetched data will be used only once and therefore should be loaded in a way to reduce cache pollution. We add prefetches to source code by hand, using the gcc `ASM` inline assembly macro to translate these directly into valid prefetch instructions.

2.4.3 B$^+$-Trees Studied and Implementation Details

Our experimental study compares pB$^+$-Trees of various node widths w with B$^+$-Trees and CSB$^+$-Trees. We consider both p_e^wB$^+$-Trees and p_i^wB$^+$-Trees (described earlier in Sections 2.3.2–2.3.4 and Section 2.3.5, respectively). We also consider the combination of both pB$^+$-Tree and CSB$^+$-Tree techniques, which we denote as a pCSB$^+$-Tree.

Implementation Details for Simulation Study. We implemented bulkload, search, insertion, deletion, and range scan operations for: (i) standard B$^+$-Trees; (ii) p^wB$^+$-Trees for node widths $w = 2, 3, \ldots, 16$; (iii) p_e^wB$^+$-Trees; and (iv) p_i^wB$^+$-Trees. For these latter two cases, the node width $w = 8$ was selected in the simulation study because our experiments showed that this choice resulted in the best search performance (consistent with the analytical computation in Section 2.2). We also implemented bulkload and search for CSB$^+$-Trees and pCSB$^+$-Trees. Although we did not implement insertion or deletion for CSB$^+$-Trees, we conduct similar experiments as in Rao and Ross [84] (albeit in a different memory hierarchy) to facilitate a comparison of the results. Although Rao and Ross present techniques to improve CSB$^+$-Tree search performance *within* a node [84], we only implemented standard binary search for all the trees studied because our focus is on memory performance (which is the primary bottleneck, as shown earlier in Figure 2.1).

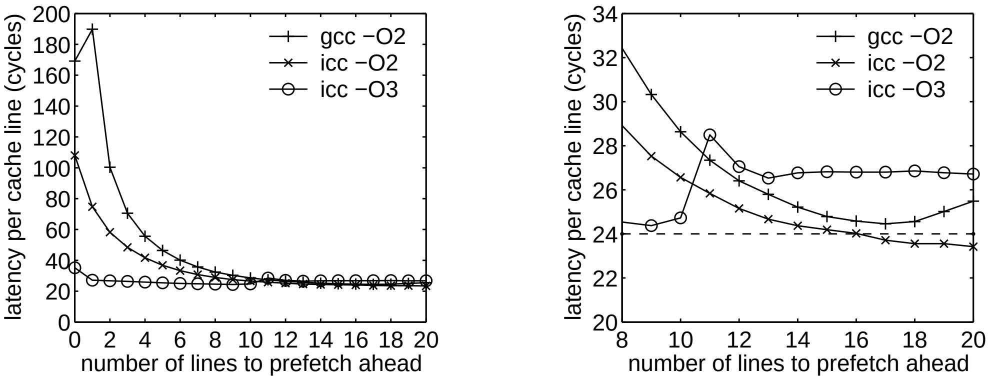
Our pB$^+$-Tree techniques improve performance over a range of `key`, `childptr`, and `tupleID` sizes. For concreteness, we report experimental results where the `keys`, `childptrs`, and `tupleIDs` are 4 bytes each, as was done in previous studies [83, 84]. As discussed in Section 2.2, we use a standard B$^+$-Tree node structure, consistent with previous studies. For the B$^+$-Tree, each node is one cache line wide (i.e. 64 bytes). Each non-leaf node contains a `keynum` field, 7 `key` fields and 8 `childptr` fields, while each leaf node contains a `keynum` field, 7 `key` fields, 7 associated `tupleID` fields, and a `next-leaf` pointer.

The nodes of the pB$^+$-Trees are the same as the B$^+$-Trees, except that they are wider. So for eight-cache-line-wide nodes, each non-leaf node is 512 bytes and contains a `keynum` field, 63 `key` fields, and 64 `childptr` fields, while each leaf node contains a `keynum` field, 63 `key` fields, 63 associated `tupleID` fields, and a `next-leaf` pointer. For the p^8_eB$^+$-Tree, non-leaf nodes have the same structure as for the pB$^+$-Tree, while each leaf node has a `hint` field and one fewer `key` and `tupleID` fields. The only difference with a p^8_iB$^+$-Tree compared to a pB$^+$-Tree is that each leaf parent node has a `next-sibling` pointer, and one fewer `key` and `childptr` fields. For the CSB$^+$-Tree and the pCSB$^+$-Tree, each non-leaf node has only one `childptr` field. For example, a CSB$^+$-Tree non-leaf node has a `keynum` field, 14 `key` fields, and a `childptr` field. All tree nodes are aligned on a 64 byte boundary when allocated.

Given the parameters in Table 2.3, one can see that the normalized memory bandwidth (B)—i.e. the number of cache misses to memory that can be serviced simultaneously—is:

$$B = \frac{T_1}{T_{\text{next}}} = \frac{250}{15} = 16.7 \tag{2.10}$$

For the p^8_eB$^+$-Tree and p^8_iB$^+$-Tree experiments, we need to select the prefetching distance (for both) and the chunk size (for the former). According to Equations (2.8) and (2.10), we select $d = \left\lceil \frac{B}{w} \right\rceil = \left\lceil \frac{16.7}{8} \right\rceil = 3$. As for the chunk size, according to Equation (2.9), we select c to be $\left\lceil \frac{B}{m} \right\rceil = \left\lceil \frac{16.7}{8} \right\rceil = 3$. (Our sensitivity analysis in Section 2.4.9 will show that selecting a slightly larger prefetching distance or chunk size to create extra slacks results in similar range scan performance.) Moreover, when bulkloading p^8_eB$^+$-Trees, we limit the external jump-pointer chunks to be at most 80% full in order to reduce the number of chunk splits due to insertions. That is, the chunks are filled with leaf node pointers up to t% full, where t is the smaller of the bulkload fill factor and 80%. (We will show insertion performance with 100% full chunks in Section 2.4.7.)



(a) The whole figure

(b) Zooming into the [8,20] region for (a)

Figure 2.8: Measuring the latency of loading an additional independent cache line (T_{next}) on the Itanium 2 machine by using cache prefetching.

Implementation Details for Itanium 2 Experiments. The index implementations are the same as in the simulation study except for the following two differences. First, the Itanium 2 machine is a 64-bit system, therefore instead of using 4-byte keys and pointers as in the simulation study, we use 8-byte keys and pointers. However, since the L3 cache line size is 128 bytes instead of 64 bytes, the number of child pointers in a one-line-wide node stays the same ($m = 8$), and therefore the node structures are the same as those in the simulation. Given the same tree size and node occupancy, the tree structures (except the external jump-pointer array) will be the same. Second, the pB$^+$-Tree algorithm parameters are different. The optimal node width for p^{w}B$^+$-Trees is 4 ($w = 4$). The other parameters are computed as follows:

$$B = \frac{T_1}{T_{\text{next}}} = \frac{189}{24} = 7.9, \quad d = \left\lceil \frac{B}{w} \right\rceil = \left\lceil \frac{7.9}{4} \right\rceil = 2, \quad c = \left\lceil \frac{B}{m} \right\rceil = \left\lceil \frac{7.9}{8} \right\rceil = 1 \tag{2.11}$$

2.4.4 A Simple Cache Prefetching Experiment: Measuring Memory Bandwidth on the Itanium 2 Machine

Figure 2.8 shows the results of a simple experiment that measures the memory bandwidth with the help of cache prefetching. This experiment verifies the cache prefetching support on the Itanium 2 machine, and provides insights into the interactions between cache prefetching and compiler optimizations.

Like the experiment for measuring the memory latency (as described in Section 2.4.1), we allocate nodes of one-cache-line size and access them. However, rather than traversing the nodes through a linked list, we build a pointer array to point to all the nodes, and read all the nodes using the pointer array. In this way, we can perform prefetching with this pointer array, which is similar to jump-pointer array prefetching. Because the execution time is minimal when prefetching fully exploits the memory bandwidth, we can obtain the T_{next}, which is the inverse of the memory bandwidth, by measuring this minimal execution time varying the number of lines to prefetch ahead. To ensure accuracy of the measurement, we perform the same three operations before an experiment as in the memory latency experiment. Moreover, we stop prefetching by the end of the array so that there is no overshooting cost, and we make sure the pointer array is in cache.

Figure 2.8(a) varies the number of nodes to prefetch ahead and reports the measured latency per cache line access for three combinations of compilers and optimization flags: “gcc -O2”, “icc -O2”, and “icc -O3”. The Itanium 2 system bus control logic has an 18-entry out of order queue, which allows for a maximum of 19 memory requests to be outstanding from a single Itanium 2 processor [44]. Therefore, we vary the prefetching distance from 0 to 20 to cover all cases. Figure 2.8(b) enlarges a portion of the whole figure to show the differences of the curves more clearly. From the figures, we see that in general all the curves decrease as the program prefetches farther ahead, thus demonstrating the effectiveness of the cache prefetching support on the Itanium 2 processor. The minimum of each curve is roughly 24 cycles. Therefore, the measured T_{next} is 24.

However, there are several large differences between the curves. The “icc -O3” curve is much closer to the minimum at the very beginning than the other two curves. However, it suddenly becomes worse when the prefetching distance is above 10, while the other two curves continue to approach the minimum. The “gcc -O2” curve jumps up when the program prefetches one node ahead.

These differences can be explained by considering the characteristics of the compilers. The icc compiler automatically inserts prefetches into its generated code. In the “-O3” level, icc performs more aggressive optimizations than in the “-O2” level. The good performance for “icc -O3” when the prefetching distance is 0 shows that the “icc -O3” handles array based accesses containing one-level indirections quite well. However, when the program prefetches more aggressively with larger prefetching distances,

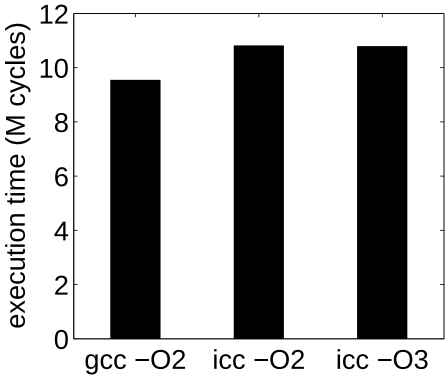


Figure 2.9: Optimal performance of the B$^+$-Tree and the p^{w}B$^+$-Tree ($w = 1, \ldots, 20$) on Itanium 2 while compiled with different compilers and optimization flags.

the prefetches inserted by the compiler interferes with the prefetches generated in the program, thus contributing to the degradation for large prefetching distances. For the gcc compiler, we insert prefetches as inline assembly instructions, which are treated as black boxes by the compiler. Therefore, a possible reason for the jump at the beginning of the "gcc -O2" curve is that the gcc compiler on Itanium 2 does not handle inline assembly instructions well. Compiler optimizations are disturbed by the inserted assembly instructions, resulting in noticeable overhead. However, as prefetches become more and more effective, the benefit from prefetching more than offsets this overhead. Therefore, the "gcc -O2" curve still approaches the same minimum value. Because of these differences, we first choose the compiler for B$^+$-Tree evaluations on the Itanium 2 machine through experiments in the following.

2.4.5 Search Performance

Choosing the Compiler for B$^+$-Tree Experiments on the Itanium 2 Machine. Figure 2.9 shows the execution time for 10,000 back-to-back random searches after bulkloading 10 million keys into trees with a 100% fill factor. The programs are compiled with three different combinations of compilers and optimization flags. We vary the node size of p^{w}B$^+$-Trees from one cache line (which is the B$^+$-Tree) to 20 cache lines. Every bar in Figure 2.9 shows the minimal execution time among all node size choices for the corresponding compiler and optimization flag. Surprisingly, the optimal performance achieved

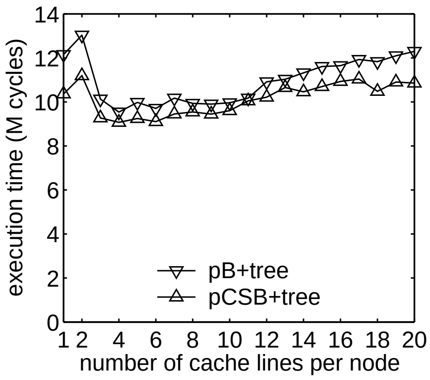
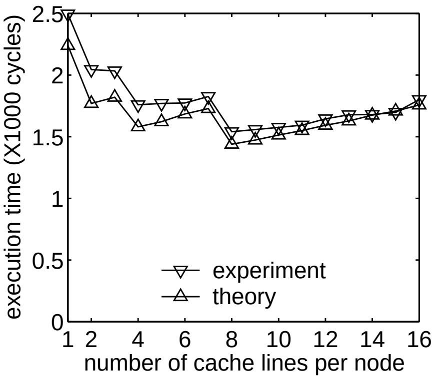


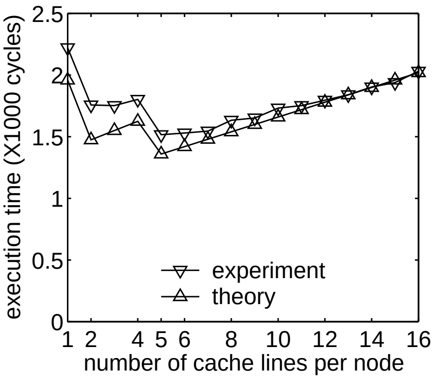
Figure 2.10: Determining the optimal node size on the Itanium 2 machine (one-cache-line points correspond to the B^+-Tree and the CSB^+-Tree).

by using “gcc -O2” is better than those by using the icc compiler. The automatically inserted prefetches by icc may interfere with our prefetching algorithm. (Unfortunately, the command line options of icc do not support disabling the insertions of prefetches on Itanium 2 machines.) Therefore, we use “gcc -O2” in all our experiments on Itanium 2 in the remaining of this section. (Note that as described in Section 2.4.2, “gcc -O2” is used for our simulation study throughout the thesis.)

Determining the Optimal Node Width on the Itanium 2 Machine. Figure 2.10 shows the execution time for the same experiment as in Figure 2.9 while varying the node width of pB^+-Trees and $pCSB^+$-Trees from 1 to 20 cache lines to determine their optimal node widths. From the figure, we see that the optimal width for pB^+-Trees is four cache lines, which is the same as computed according to Equations 2.1-2.6. However, the optimal node width (four cache lines) for $pCSB^+$-Trees is different from its computed value (two cache lines). Like the “gcc -O2” curve in Figure 2.8, the curves in Figure 2.10 all jump up at the beginning, which may result from the overhead of using inline assembly instructions in gcc. This behavior is not taken into account by the theoretical computation. When the node size increases, the benefit of prefetching more than offsets this overhead and $pCSB^+$-Trees achieves the best performance when $w = 4$. Therefore, we conclude that our theoretical computation accurately predicts the optimal node width except for two-cache-line cases on the Itanium 2 machine. If the computed value is two cache lines, we need to measure the experimental performance for nearby width choices in order to determine the optimal node width.



(a) pB+-Trees (w=1 is B+-Tree)



(b) pCSB+-Trees (w=1 is CSB+-Tree)

Figure 2.11: Comparing experimental results through simulations with theoretical costs for cold searches in trees bulkloaded 100% full with 10 million keys.

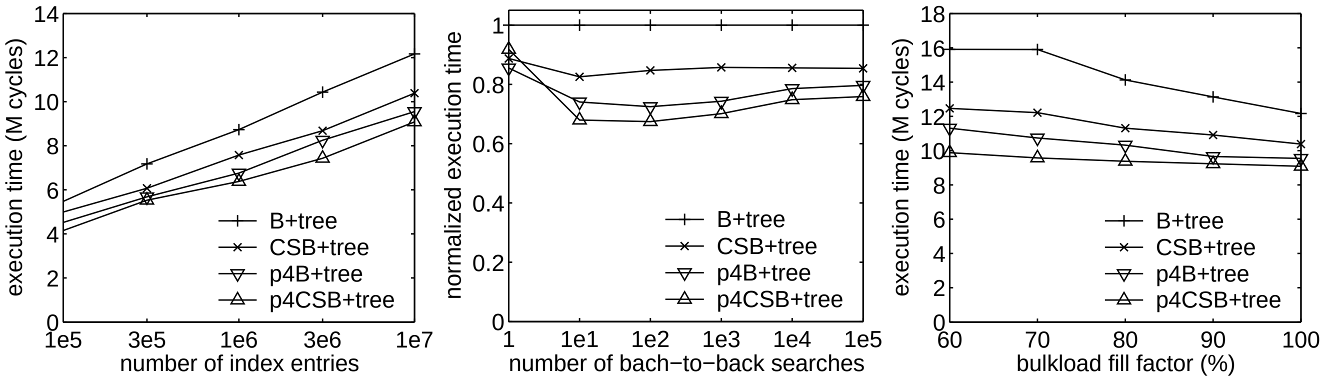
Table 2.4: The number of levels in trees with different node widths.

number of cache lines per node (w)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
number of levels of pB+-Trees	8	6	6	5	5	5	5	4	4	4	4	4	4	4	4	4
number of levels of pCSB+-Trees	7	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4

Determining the Optimal Node Width on the Simulation Platform. Figure 2.11 shows the execution time for a single cold search after bulkloading 10 million keys into trees with a 100% fill factor. As discussed in Section 2.2.2, the optimal node width is not larger than B, which is 16.7 on the simulation platform. Therefore, we vary the node width of pB+-Trees and pCSB+-Trees from 1 to 16 cache lines to determine their optimal node widths. When running the experiments, we flush the CPU caches and the TLB in the simulator before every search. We factor out the differences of search paths by reporting for every point in the figure the average execution time of 10,000 random searches.

Because Equations 2.1-2.6 compute the cost for a single search operation, this cold cache experiment enables a side-by-side comparison between the experimental and the theoretical costs in Figure 2.11. The theoretical costs of searches in pCSB+-Trees can be obtained in a similar fashion as in Equations 2.1-2.6.

Figure 2.11 shows that the theoretical results match the experimental results very well, thus verifying



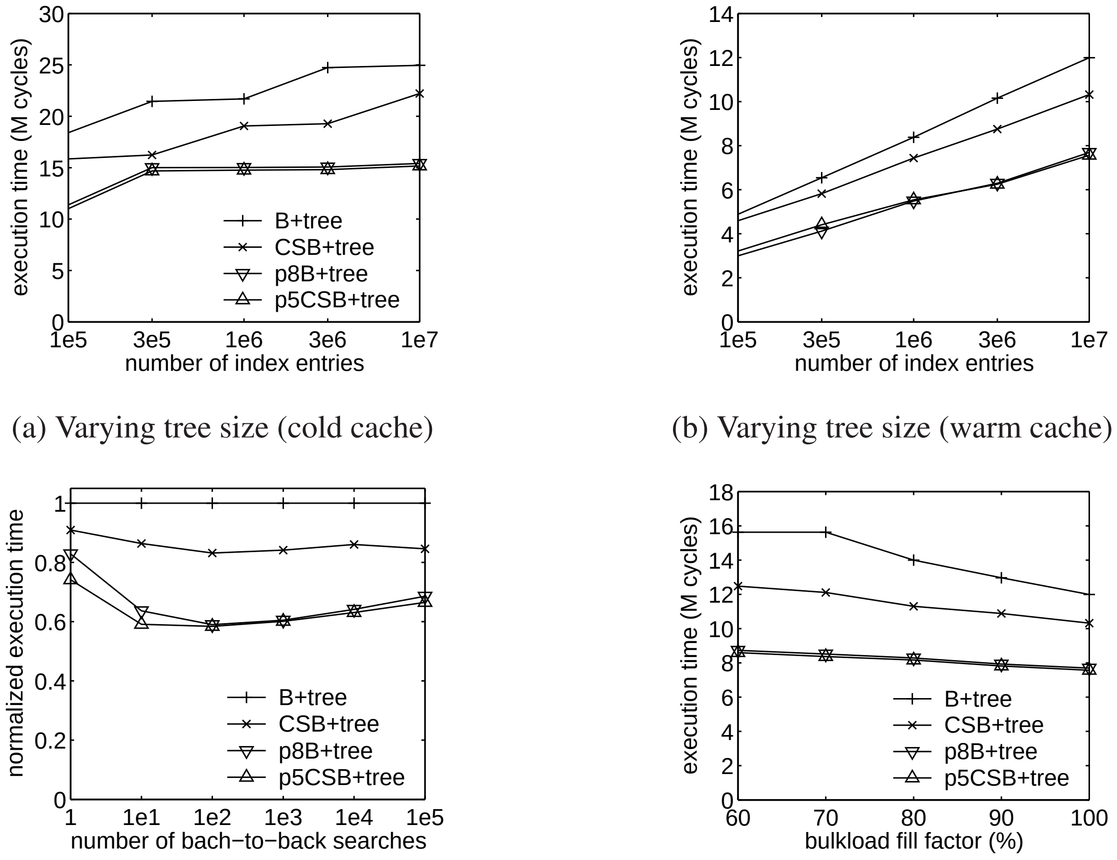
(a) Varying tree size (b) Varying number of searches (c) Varying node occupancy

Figure 2.12: Search performance on Itanium 2 (warm cache).

the correctness of our theoretical models. The curves provide insight into the index search performance. All the curves have step-like shapes, and within the same step they are tilted upwards. The large steps for a curve occur when the number of levels in the tree decreases. This can be verified by examining Table 2.4, which lists the number of levels in the tree for each data point plotted in Figure 2.11. Within a step, the search cost increases as the node size. Therefore, the minimum point occurs among the points at the beginning of the steps. From the figure, we see that the optimal width for pB$^+$-Trees is eight cache lines, and the optimal width for pCSB$^+$-Trees is five cache lines.

In the following, we vary the tree size, the node occupancy, and the number of back-to-back searches to show the performance of the optimal trees under a large varieties of settings.

Search Performance Varying the Tree Size. Figure 2.12(a), Figures 2.13(a) and (b) show the execution time of 10,000 random searches after bulkloading 1e5, 3e5, 1e6, 3e6, and 1e7 keys into the trees (nodes are 100% full except the root). In the experiments shown in Figure 2.13(a), the cache and the TLB are cleared between each search (the "cold cache" case); whereas in the experiments shown in Figure 2.12(a) and Figure 2.13(b), search operations are performed one immediately after another (the "warm cache" case). Depending on the operations performed between the searches, the real-world performance of an index search would lie in between the two extremes: closer to the warm cache case for index joins, while often closer to the cold cache case for single value selections. From these experiments,



(a) Varying tree size (cold cache)

(b) Varying tree size (warm cache)

(c) Varying number of searches (warm cache)

(d) Varying occupancy of nodes (warm cache)

Figure 2.13: Search performance through simulations.

we see that: (i) Figure 2.13(b) shows trends similar to Figure 2.12(a), which verifies our simulator model; (ii) B^+-Trees are the slowest; and (iii) The trees with wider nodes and prefetching support (pB^+-Tree, pCSB^+-Tree) both perform better than their non-prefetching counterparts (B^+-Tree, CSB^+-Tree).

On Itanium 2, the pB^+-Tree achieves 1.16-1.31X speedups over B^+-Tree and 1.05-1.20X speedups over CSB^+-Tree. Combining our prefetching scheme with the CSB^+-Tree scheme, the pCSB^+-Tree achieves even better performance (1.18-1.40X speedups over B^+-Tree). On the simulation platform, for cold cache experiments, the speedup of the pB^+-Tree over the B^+-Tree is between a factor of 1.43 to 1.64. The cold cache speedup of the pB^+-Tree over the CSB^+-Tree is between a factor of 1.08 to 1.44. Likewise, the warm cache speedups are 1.53 to 1.63 and 1.34 to 1.53, respectively. Note that the

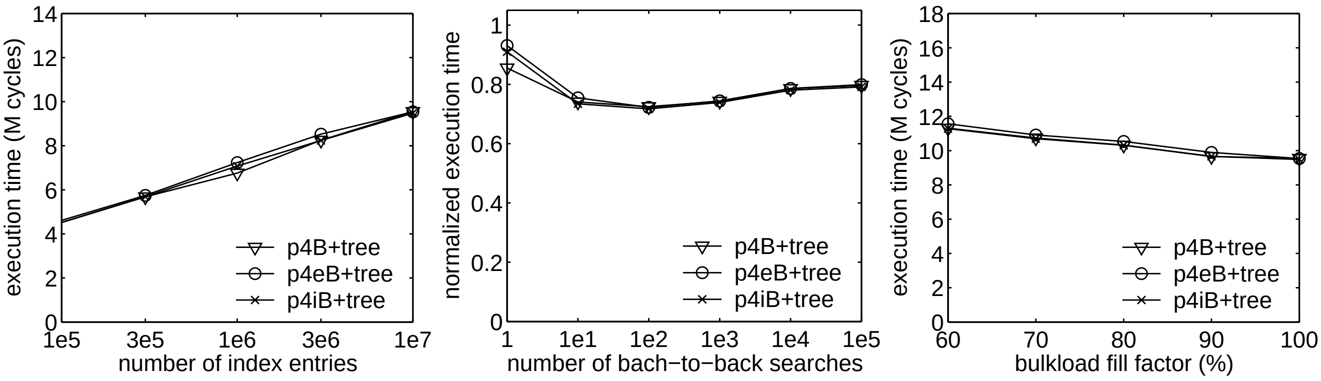
Table 2.5: The number of levels in trees for Figure 2.13(a) and (b).

Tree Type	Number of Keys				
	1e5	3e5	1e6	3e6	1e7
B^+-Tree	6	7	7	8	8
CSB^+-Tree	5	5	6	6	7
p^8B^+-Tree	3	4	4	4	4
p^5CSB^+-Tree	3	4	4	4	4

speedups are larger on the simulation platform. This is because the simulator has faster clock rate and larger memory latency, resulting in larger benefits of cache optimizations.

The trend of every single curve is clearly shown in the cold cache experiment: The curves all increase in discrete large steps, and within the same step they increase only slightly. Similar to Figure 2.11, the large steps for a curve occur when the number of levels in the tree changes. Table 2.5 shows the number of levels in the tree for each data point plotted in Figure 2.13(a) and (b). Within a step, additional leaf nodes result in more keys in the root node (the other nodes in the tree remain full), which in turn increases the cost to search the root. The step-up trend is blurred in the warm cache curves because the top levels of the tree may remain in the cache. Moreover, because the number of levels are the same, the p^8B^+-Tree and the p^5CSB^+-Tree have very similar performance. (On the Itanium 2 machine, the number of levels of p^4B^+-Tree and p^4CSB^+-Tree differs, leading to the different performance.) Therefore, we conclude that the performance gains for wider nodes stem mainly from the resulting decrease in tree height.

Search Performance Varying the Number of Back-to-Back Searches. Figure 2.12(b) and Figure 2.13(c) show the benefit of our scheme across a wide range of possible usage patterns of index searches in the warm cache case. All the trees are bulkloaded with 10 million ⟨`key`, `tupleID`⟩ pairs to be 100% full (except for the root nodes). We vary the number of random searches from 1 to 100,000, and report the execution times normalized to those of B^+-Trees. From Figure 2.12(b) and Figure 2.13(c), we see that the trees with wider nodes and prefetching support all perform better than their non-prefetching counterparts. On Itanium 2, the pB^+-Tree achieves 1.17-1.38X speedups over the B^+-Tree and 1.04-



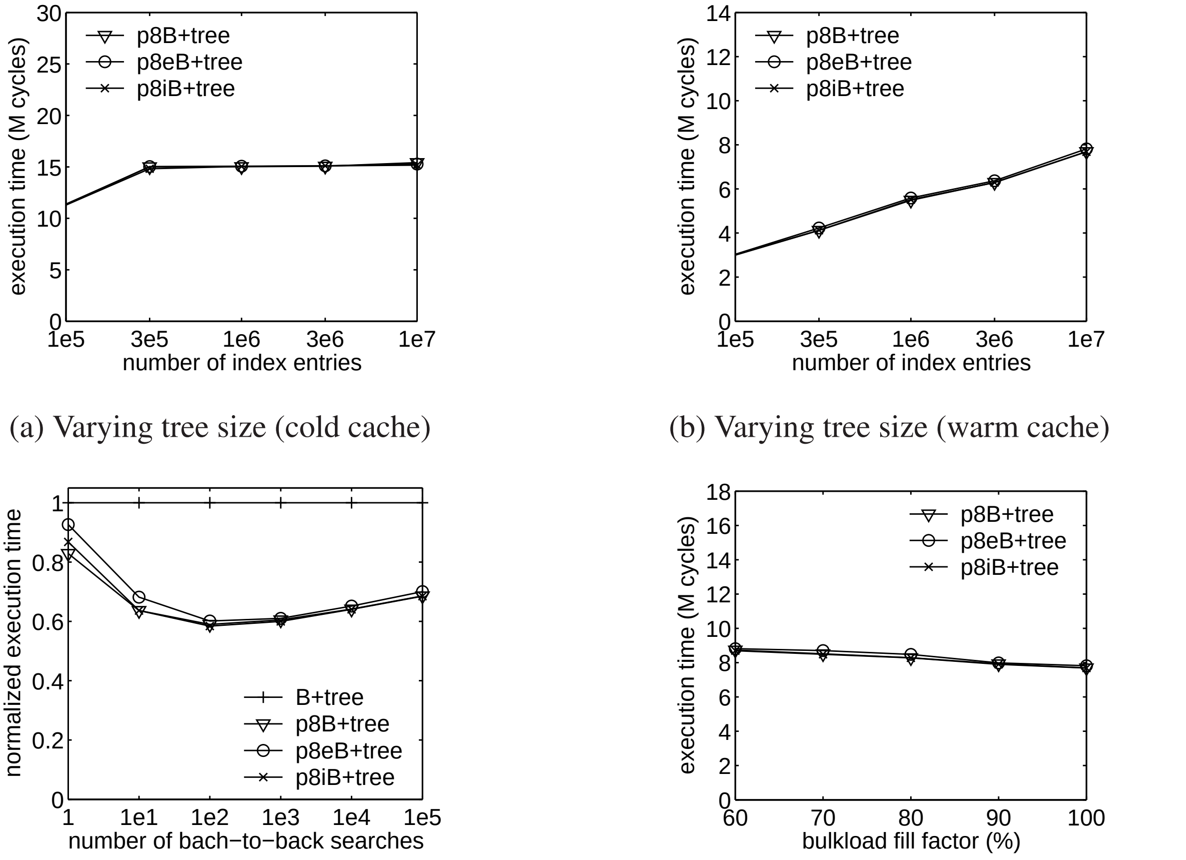
(b) Varying tree size (c) Varying number of searches (d) Varying node occupancy

Figure 2.14: Comparing the search performance of p^4B^+-Trees, $p^4_eB^+$-Trees, and $p^4_iB^+$-Trees on the Itanium 2 machine.

1.17X speedups over the CSB$^+$-Tree. On the simulation platform, the pB$^+$-Tree achieves 1.20-1.70X speedups over the B$^+$-Tree and 1.10-1.41X speedups over the CSB$^+$-Tree.

Search Performance Varying the Node Occupancy. Figure 2.12(c) and Figure 2.13(d) show the effect on search performance of varying the bulkload fill factor. All the trees are bulkloaded with 10 million ⟨`key`, `tupleID`⟩ pairs, with bulkload fill factors of 60%, 70%, 80%, 90%, and 100%. Note that the actual number of used entries in leaf nodes in an experiment is the product of the bulkload fill factor and the maximum number of slots (truncated to obtain an integer). Interestingly, the 60% and 70% fill factors result in the same number of leaf index entries for one-cache-line-node B$^+$-Trees. Therefore, the corresponding points are the same. As in the previous experiments, Figure 2.12(c) and Figure 2.13(d) shows that: (i) the B$^+$-Tree has the worst performance; (ii) the trees with wider nodes and prefetching support all perform better than their non-prefetching counterparts. On Itanium 2, the pB$^+$-Tree achieves 1.27-1.48X speedups over the B$^+$-Tree and 1.09-1.14X speedups over the CSB$^+$-Tree. On the simulation platform, the pB$^+$-Tree achieves 1.56-1.84X speedups over the B$^+$-Tree and 1.34-1.43X speedups over the CSB$^+$-Tree.

Searches on Trees with Jump-Pointer Arrays. Our next experiments determine whether the different structures for speeding up range scans have an impact on search performance. We use the optimal



(a) Varying tree size (cold cache)

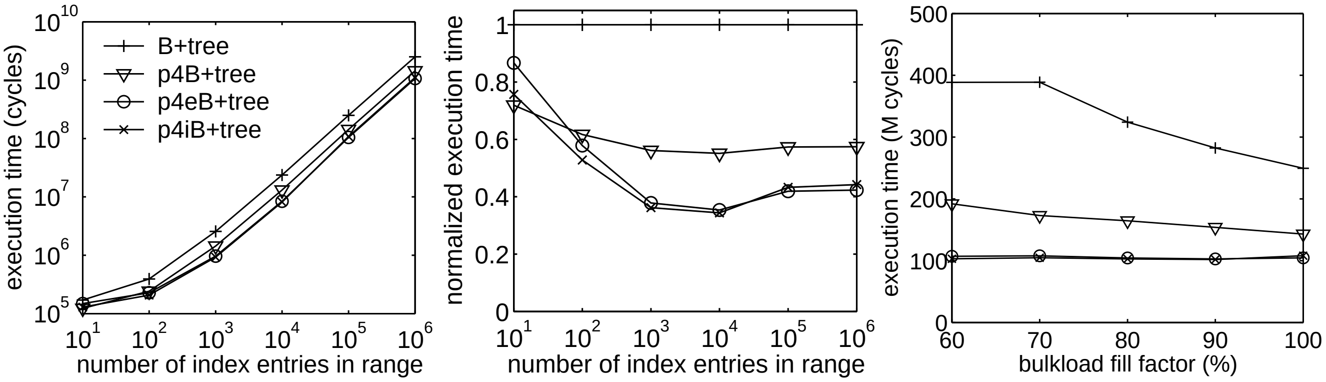
(b) Varying tree size (warm cache)

(c) Varying number of searches (warm cache)

(d) Varying occupancy of nodes (warm cache)

Figure 2.15: Comparing the search performance of p^8B^+-Trees, $p_e^8B^+$-Trees, and $p_i^8B^+$-Trees on the simulation platform.

node width for these experiments. Figure 2.14 and Figure 2.15 compare the search performance of the p^wB^+-Tree, the $p_e^wB^+$-Tree, and the $p_i^wB^+$-Tree, where $w = 4$ on Itanium 2 and $w = 8$ on the simulation platform. The same experiments as in Figure 2.12 and Figure 2.13 were performed. As described previously in Section 2.3, both the $p_e^wB^+$-Tree and the $p_i^wB^+$-Tree consume space in the tree structures relative to the p^wB^+-Tree: The maximum number of keys in leaf nodes is one fewer for the $p_e^wB^+$-Tree, and the maximum number of keys in leaf parent nodes is one fewer for the $p_i^wB^+$-Tree. Figure 2.12 and Figure 2.15 show that these differences have a negligible impact on search performance.



(a) Varying range size (wall-clock time)

(b) Varying range size (normalized time)

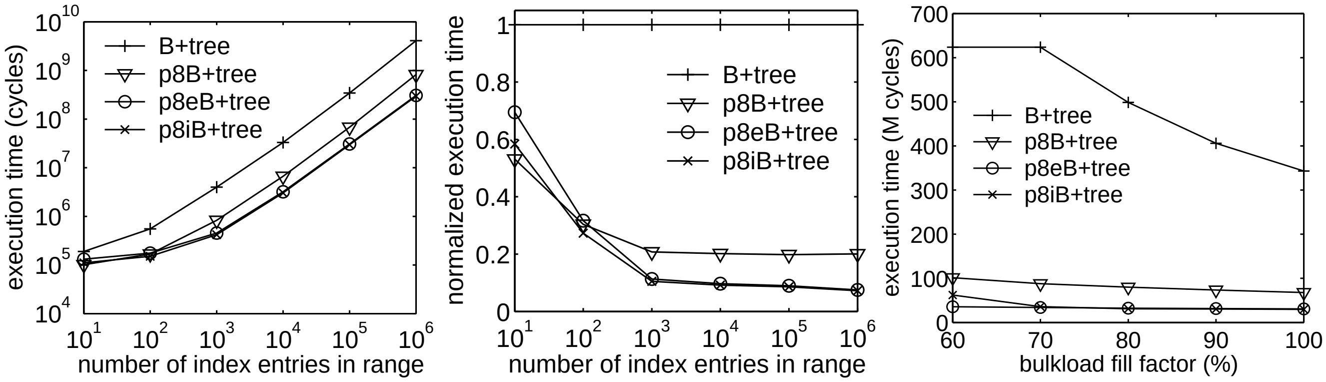
(c) Varying occupancy (wall-clock time)

Figure 2.16: Range scan performance on the Itanium 2 machine.

2.4.6 Range Scan Performance

In our next set of experiments, we evaluate the effectiveness of our techniques for improving range scan performance. We compare B^+-Trees, p^wB^+-Trees, $p_e^wB^+$-Trees, and $p_i^wB^+$-Trees, where w is the optimal node width for searches, which are presumed to occur more frequently than range scans. As discussed in Section 2.4.2, we set the prefetching distance d to 2 nodes and the chunk size c to 1 cache line on Itanium 2. On the simulation platform, we set $d = 3$ and $c = 3$.

Varying the Range Size. Figures 2.16 (a) and (b), and Figures 2.17(a) and (b) show the execution time of range scans while varying the number of `tupleIDs` to scan per request (i.e. the size of the range). Because of the large performance gains for pB^+-Trees, the execution time is shown on a logarithmic scale in Figure 2.16 (a) and Figure 2.17(a). Figure 2.16 (b) and Figure 2.17(b) show the execution times normalized to those of B^+-Trees. The trees are bulkloaded with 10 million $\langle$`key`, `tupleID`$\rangle$ pairs, using a 100% bulkload fill factor. Then 100 random starting `keys` are selected, and a range scan is requested for s `tupleIDs` starting at that starting key value, for $s =$ 1e1, 1e2, 1e3, 1e4, 1e5, and 1e6. The execution time plotted for each s is the total of the 100 range scans. Before a range scan request, the caches are cleared to more accurately reflect scenarios in which range scan requests are interleaved with other



(a) Varying range size (wall-clock time)

(b) Varying range size (normalized time)

(c) Varying occupancy (wall-clock time)

Figure 2.17: Range scan performance through simulations.

database operations or application programs (which would tend to evict any cache-resident nodes).[3]

Like the search experiments, the Itanium 2 curves and the simulation curves in the range scan experiments have similar shapes. This verifies our simulation model. As we see in Figures 2.17(a) and (b), p_eB$^+$-Trees and p_iB$^+$-Trees achieve a *factor of 8.8X to 13.8X speedups* over standard B$^+$-Trees for ranges containing 1,000 to 1 million `tupleIDs`. The figures show the contribution of both aspects of our pB$^+$-Tree design to overall performance. First, extending the node size and simultaneously prefetching all cache lines within a node while scanning (and performing the initial search)—similar to what was illustrated earlier in Figure 2.4(b)—results in a speedup of 4.8 to 5.0, as shown by the difference between pB$^+$-Trees and B$^+$-Trees in Figure 2.17 (a) and (b). Second, by using jump-pointer arrays to prefetch ahead across the (extra-wide) leaf nodes, we see an additional speedup of roughly 2 in this case, as shown by the improvement of both p_eB$^+$-Trees and p_iB$^+$-Trees over pB$^+$-Trees. Since both p_eB$^+$-Trees and p_iB$^+$-Trees achieve nearly identical performance, there does not appear to be a compelling need to build an external (rather than an internal) jump-pointer array, at least for these system parameters.

When scanning far fewer than 1,000 `tupleIDs`, however, the startup cost of our prefetching schemes

[3]The code calls a function to flush caches on the Itanium 2 machine. The function allocates a chunk of memory much larger than the CPU cache. Then it scans the memory to evict previous data from the caches and the TLBs. Note that our measurements do not include the cache flushing function.

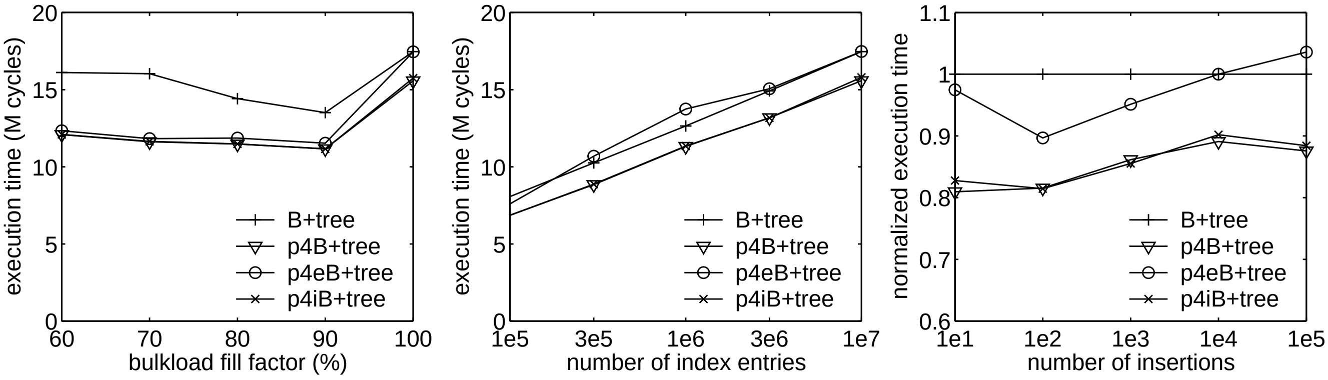
becomes noticeable. For example, when scanning only 10 `tupleIDs`, pB$^+$-Trees are only slightly faster than B$^+$-Trees, and prefetching with jump-pointer arrays is actually slower than prefetching only the extra-wide nodes. This suggests a scheme where jump-pointer arrays are only exploited for prefetching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

On the Itanium 2 machine, prefetching wider nodes achieves 1.39-1.81X speedups over the B$^+$-Tree. Jump-pointer array prefetching overlaps cache misses across leaf node accesses and achieves 2.26-2.91X speedups when there are at least 1000 `tupleIDs` in the range. For ranges containing 100 or fewer `tupleIDs`, the p$_i$B$^+$-Tree and p$_e$B$^+$-Tree achieve 1.15-1.89X speedups. The Itanium 2 speedups are smaller than the simulation for two reasons: (i) As mentioned previously, the simulation platform has larger memory latency, thus leading to larger benefits for cache optimizations; (ii) In the experiments, the leaf nodes are actually contiguous in memory after bulkloading, and therefore hardware-based prefetching mechanisms may improve the performance of B$^+$-Trees. When the nodes are not contiguous in memory, which is the typical case, the range scan speedups are much larger, as will be described in our experiments on mature trees in Section 2.4.8.

Varying the Node Occupancy. Figure 2.16 (c) and Figure 2.17(c) show the execution time of range scans while varying the bulkload fill factor. The trees are bulkloaded with 10 million $\langle$`key`, `tupleID`$\rangle$ pairs, with bulkload fill factors of 60%, 70%, 80%, 90%, and 100%. We report the total time of 100 random range scans each for 100,000 `tupleIDs`. As the bulkload fill factor decreases, the number of leaf nodes to be scanned increases (since we must skip an increasing number of empty slots), and hence our prefetching schemes achieve larger speedups. Our jump-pointer array prefetching achieves 11.1-18.3X speedups over B$^+$-Trees on the simulation platform, and 2.31-3.77X speedups on the Itanium 2 machine.

2.4.7 Update Performance

In addition to improving search and scan performance, another one of our goals is to achieve good performance on *updates*, especially since this had been a problem with earlier cache-sensitive index

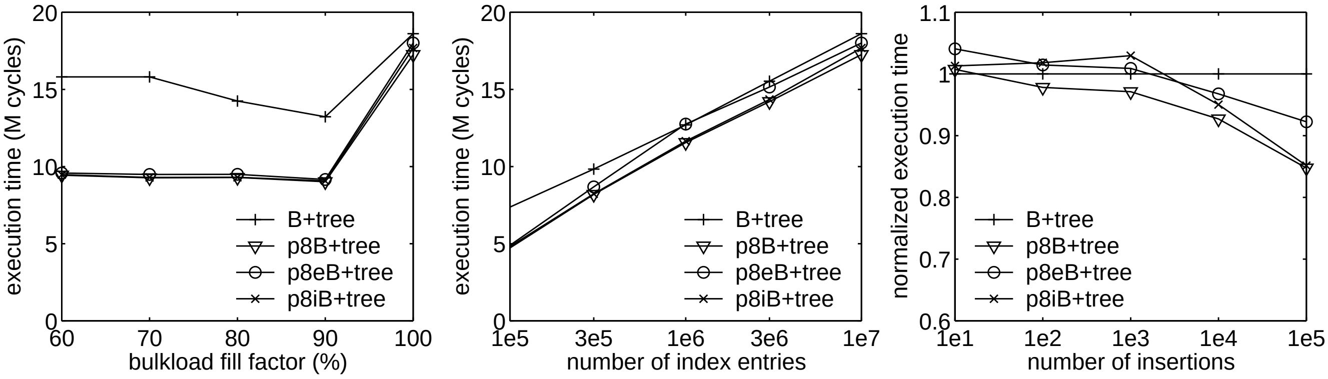


(a) Varying occupancy of tree nodes

(b) Varying tree size (trees are 100% full)

(c) Varying number of insertions (trees are 100% full)

Figure 2.18: Insertion performance on the Itanium 2 machine (warm cache).



(a) Varying occupancy of tree nodes

(b) Varying tree size (trees are 100% full)

(c) Varying number of insertions (trees are 100% full)

Figure 2.19: Insertion performance through simulations (warm cache).

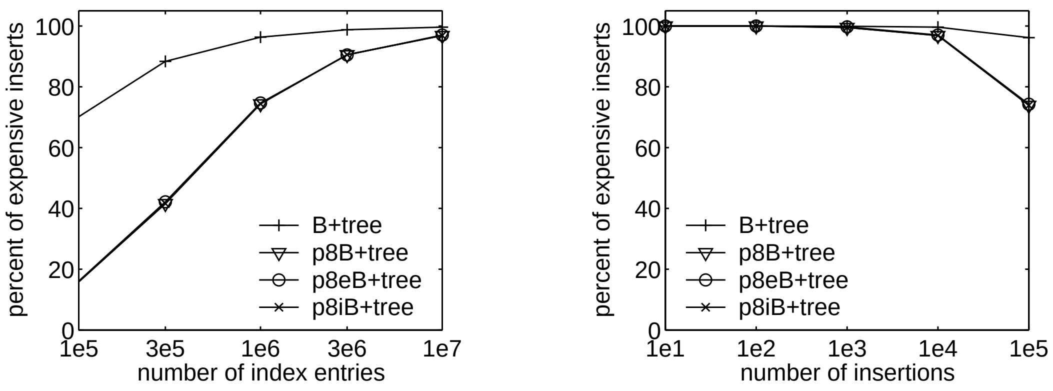
structures [83, 84]. To quantify the impact of pB$^+$-Trees on insertion and deletion performance, Figure 2.18 and Figure 2.19 show insertion performance varying the node occupancy, the tree size, and the number of random insertions, and Figure 2.22 show deletion performance varying the node occupancy and the tree size.

Insertion Performance Varying the Node Occupancy. Figure 2.18(a) and Figure 2.19(a) show the execution time for 10,000 random insertions into trees containing 10 million $\langle$`key`, `tupleID`$\rangle$ pairs with bulkload factors ranging from 60% to 100%, and with warm caches. As we see in the figure, all three pB$^+$-Tree schemes (i.e. pB$^+$-Trees, p$_e$B$^+$-Trees, and p$_i$B$^+$-Trees) perform roughly the same, and all are significantly faster than the B$^+$-Tree when the bulkload fill factor is less than 100%; they achieve 1.44-1.71X speedups over B$^+$-Trees on the simulation platform, and 1.17-1.38X speedups on Itanium 2. When the trees are not full, insertions often find empty slots in leaf nodes, and there are very few expensive node splits. Because data movement inside a leaf node does not incur cache misses, the search operation at the beginning of a insertion dominates insertion cost. Therefore, faster search operations of pB$^+$-Trees lead to significantly better insertion performance when trees are not full.

When trees are 100% full (which is the worst case and typically the less common case for insertions), the insertion performance of pB$^+$-Trees is still comparable or slightly better than B$^+$-Trees. In the following, we describe the results of two sets of experiments to better understand the insertion performance when trees are 100% full.

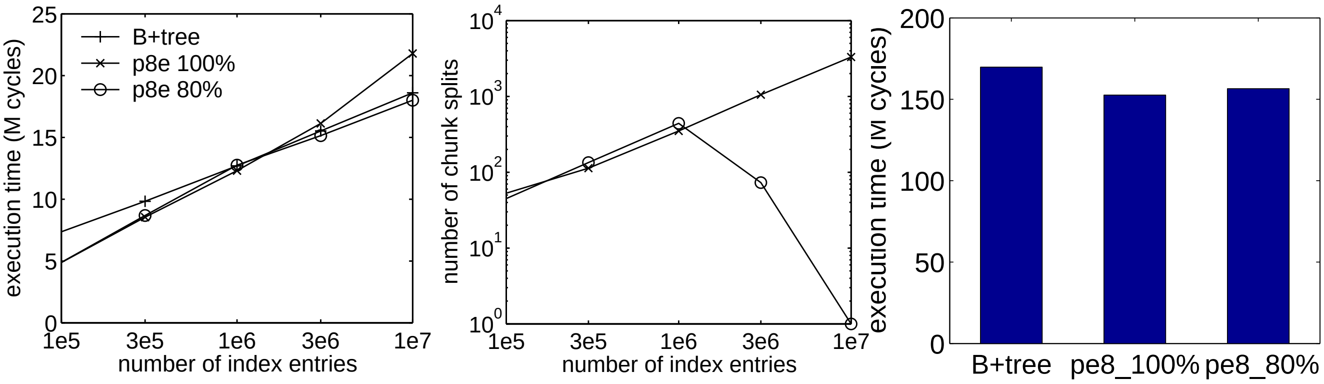
Insertion Performance When Trees Are 100% Full. Figure 2.18(b) and Figure 2.19(b) show the execution time for 10,000 random insertions into trees bulkloaded with 1e5, 3e5, 1e6, 3e6, and 1e7 ⟨`key`, `tupleID`⟩ pairs 100% full. Figure 2.18(c) and Figure 2.19(c) show the normalized execution time for i random insertions into trees bulkloaded with 10 million ⟨`key`, `tupleID`⟩ pairs 100% full, where $i =$ 1e1, 1e2, 1e3, 1e4, 1e5. Note that the "1e7" points in Figure 2.18(b) and Figure 2.19(b) and the "1e4" points in Figure 2.18(c) and Figure 2.19(c) show the results of the same experiments as the "100%" points in Figure 2.18(a) and Figure 2.19(a). From the figures, we see that all three pB$^+$-Tree schemes achieve comparable performance as B$^+$-Trees across a wide range of settings.

To further understand the performance, we depict in Figure 2.20 the percentage of insertions causing expensive node splits for the experiments in Figure 2.19(b) and (c). Comparing the performance curves with the node split curves, we can clearly see the determining effect of node splits on insertion performance when trees are 100% full. The number of node splits is affected by two major factors: the number of leaf nodes and the number of insertions. As insertions proceed, more and more leaf nodes will have splitted. Since the resulting leaf nodes from a node split have 50% empty slots, later insertions will have a larger probability of reaching a leaf node with empty slot, thus not incurring node splits. With the same



(a) Analysis for Figure 2.19(b) (b) Analysis for Figure 2.19(c)

Figure 2.20: Analyzing percentage of insertions causing node splits to understand the performance of inserting into 100% full trees.



(a) Varying tree size (1e4 insertions) (b) Number of chunk splits for (a) (c) 1e5 insertions into trees with 1e7 keys 100% full

Figure 2.21: Insertions to $p_e^8B^+$-Trees with 80% full chunks and 100% full chunks (warm cache).

number of insertions, the smaller the number of the leaf nodes, the more likely that later insertions find empty slots. From another angle, if the number of leaf nodes is fixed, the larger the number of insertions, the larger fraction of them will benefit from empty slots in leaf nodes.

Compared to B^+-Trees, our prefetching schemes have wider nodes, and thus fewer leaf nodes. Therefore, they incur significantly fewer number of node splits than B^+-Trees, as shown in Figure 2.20. The reduction is especially dramatic for the smaller trees or the larger number of insertions. For example, when trees contain 1e5 index entries, our prefetching schemes reduce roughly 80% node splits compared to B^+-Trees, thus achieving over 1.50X speedups. When the number of insertions are 1e5, even for the largest tree size (with 1e7 entries) in our experiment, our prefetching schemes reduce roughly 20% node splits compared to B^+-Trees, resulting in a 1.17X speedup for the pB^+-Trees and the p_iB^+-Trees, and a 1.08X speedup for the p_eB^+-Trees. The p_eB^+-Trees is slightly slower than the other two prefetching schemes because of chunk splits, which we quantify in the following.

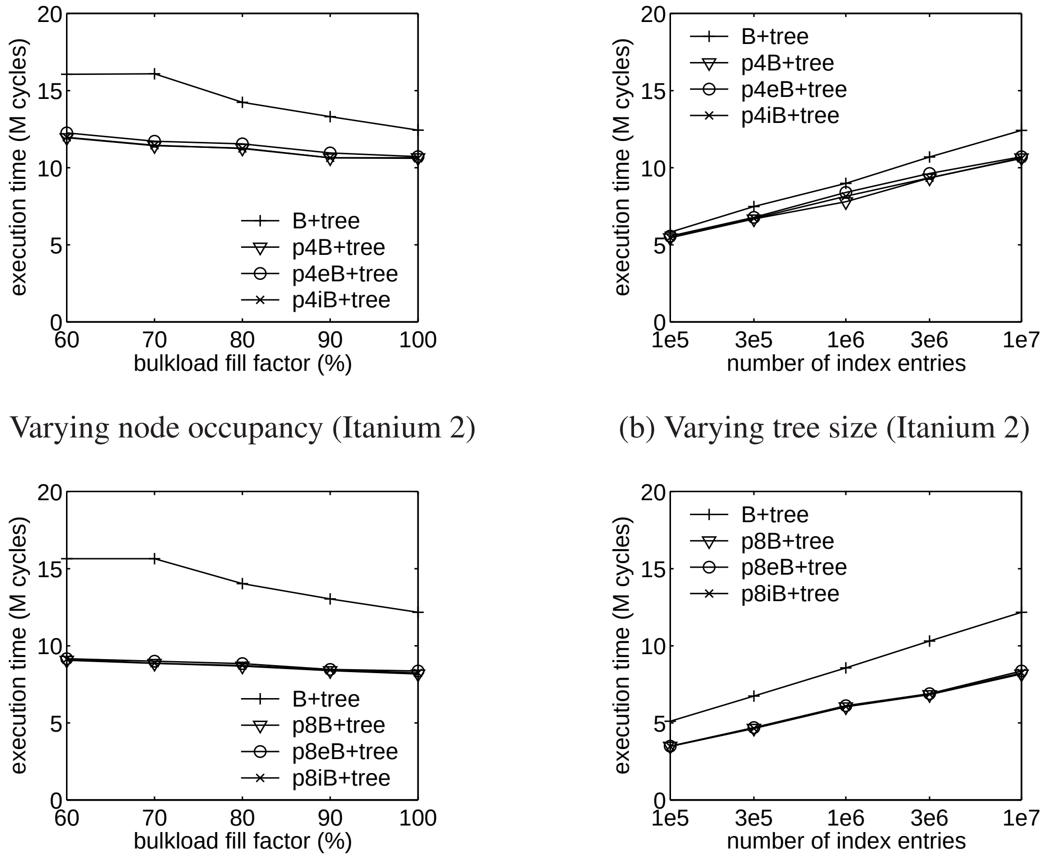
Impact of Chunk Splits on Insertion Performance. Figure 2.21 studies the impact of external jump-pointer array chunk splits on insertion performance. As described in implementation details in Section 2.4.3, p_eB^+-Trees limit the chunks to be at most 80% full upon bulkloading. Here, we compare the performance of p_eB^+-Trees with 100% full chunks (with label "p8e 100%") and the normal p_eB^+-Trees with 80% full chunks (with label "p8e 80%").

Figure 2.21(a) performs the same experiments as in Figure 2.19(b). Figure 2.21(b) depicts the number of chunk splits for the experiments in Figure 2.21(a). We can see that when the tree size is large, p_eB^+-Trees with 100% full chunks incur several orders of magnitude more chunk splits than p_eB^+-Trees with 80% full chunks. This results from two effects: (i) For the former, the larger number of node splits result in more insertions into external jump-pointer arrays, thus increasing the number of chunk splits; (ii) For the latter, the larger number of leaf nodes require larger number of chunks, which in turn leads to larger number of empty chunk slots collectively, thus decreasing the number of chunk splits.

Figure 2.21(c) shows the performance of inserting a larger number (1e5) of keys into the p_eB^+-Trees with 100% full chunks and the p_eB^+-Trees with 80% full chunks. Rather than slowing down because of chunk splits, the p_eB^+-Trees with 100% full chunks achieves better performance than B^+-Trees: The chunk split overhead is amortized across a large number of insertions.

Summarizing the findings from Figure 2.21, we conclude that it is a good idea to limit the chunks to be 80% full upon bulkloading. This avoids a large number of node splits and a large number of chunk splits to occur at the same time, thus achieving the amortizing effect with even small number of insertions.

Deletion Performance. Figure 2.22(a) and Figure 2.22(c) show the execution time of deleting 10,000 random keys from trees containing 10 million $\langle$`key`, `tupleID`$\rangle$ pairs with bulkload factors ranging from



(a) Varying node occupancy (Itanium 2)

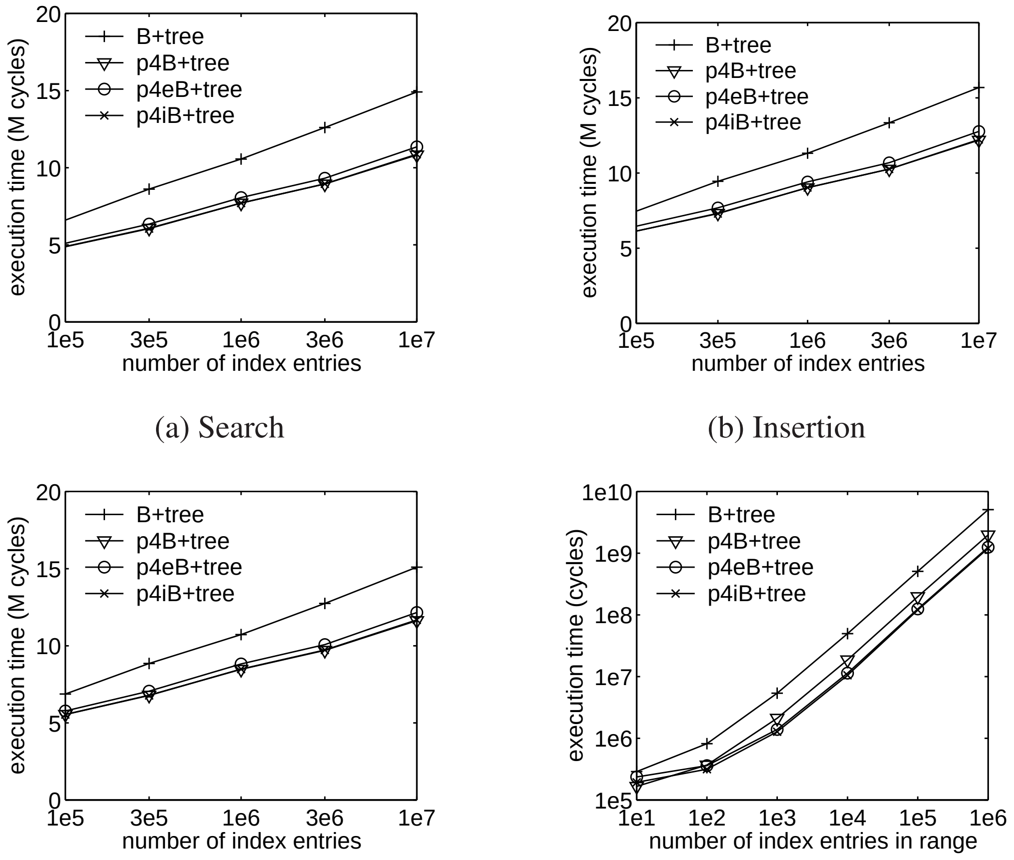
(b) Varying tree size (Itanium 2)

(a) Varying node occupancy (simulation)

(b) Varying tree size (simulation)

Figure 2.22: Deletion performance on the Itanium 2 machine and through simulations (warm cache).

60% to 100%, and with warm caches. Figure 2.22(b) and Figure 2.22(d) show the execution time for 10,000 random deletions from trees bulkloaded with 1e5, 3e5, 1e6, 3e6, and 1e7 ⟨`key`, `tupleID`⟩ pairs 100% full. Since both pB$^+$-Trees and B$^+$-Trees use lazy deletion, very few deletions result in a deleted node or a key redistribution. Hence, the deletion performance is dominated by the initial search operation. Our prefetching schemes achieve 1.40-1.77X speedups over B$^+$-Trees on the simulation platform, and 1.05-1.41X speedups on the Itanium 2 machine.



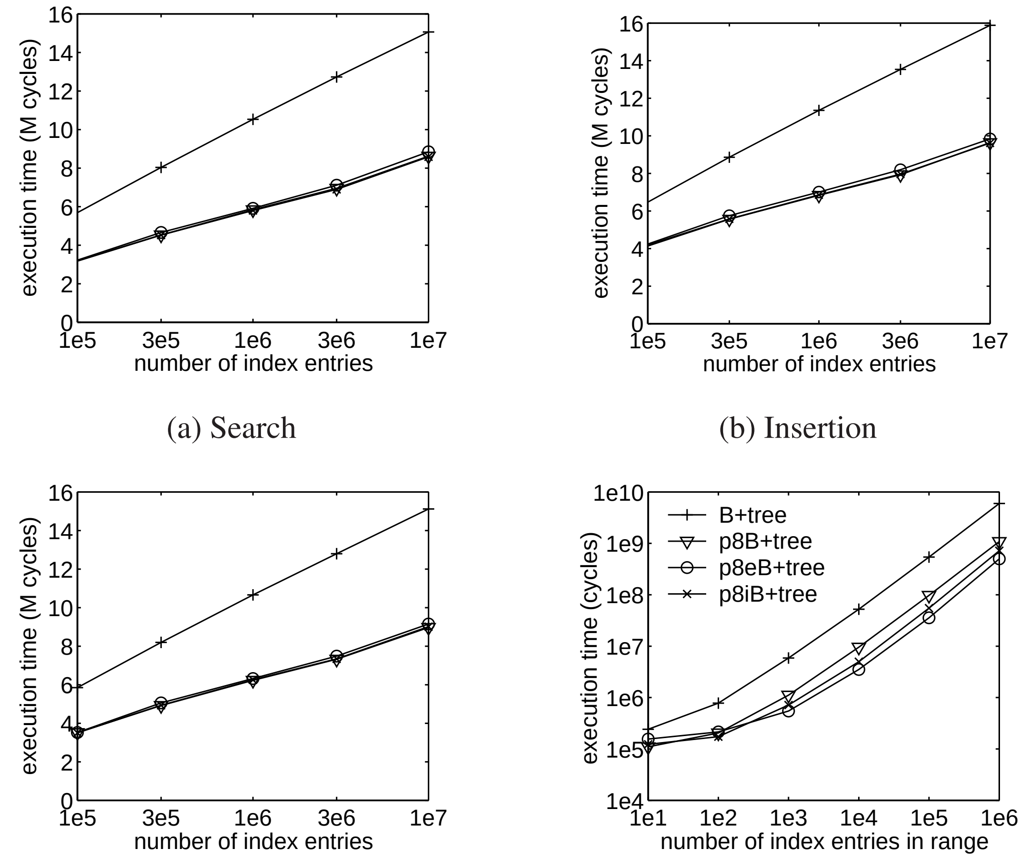
(a) Search (b) Insertion

(c) Deletion (d) Range Scan

Figure 2.23: Operations on mature trees on the Itanium 2 machine (warm cache).

2.4.8 Operations on Mature Trees

Our next set of experiments show the performance of index operations on *mature trees* [84]. To obtain a mature tree containing N index entries, we use the same method as Rao and Ross [84]: We first bulkload a tree with $10\% \cdot N$ index entries, and then insert $90\% \cdot N$ index entries. We compare the B^+-Tree, the pB^+-Tree, the p_eB^+-Tree, and the p_iB^+-Tree. Figures 2.23(a)-(c) and Figures 2.24(a)-(c) show the execution time for performing 10,000 random searches, insertions, or deletions in trees containing 1e5, 3e5, 1e6, 3e6, and 1e7 index entries. Figures 2.23(d) and Figure 2.24(d) show the execution time for range scans on matures trees with 10 million index entries starting from 100 random locations to retrieve 1e1, 1e2, 1e3, 1e4, 1e5, and 1e6 `tupleIDs`.

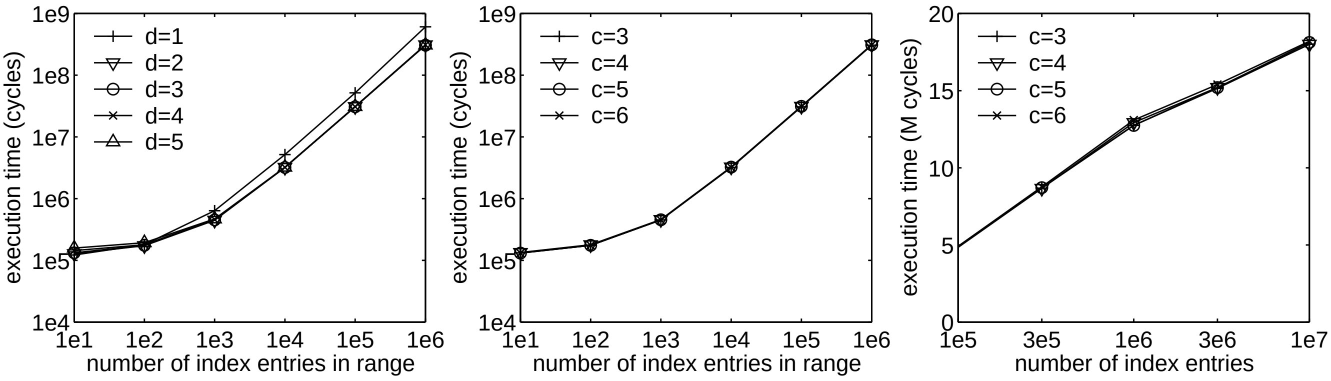


(a) Search (b) Insertion

(c) Deletion (d) Range Scan

Figure 2.24: Operations on mature trees through simulations (warm cache).

We find similar performance for mature trees as in previous experiments for trees immediately after bulkloads. On the simulation platform, our prefetching schemes achieve 1.70-1.85X speedups for searches, 1.53-1.71X speedups for insertions, 1.62-1.75X speedups for deletions. Our jump-pointer array prefetching achieve 8.2-15.2X speedups for scans over ranges larger than 1,000 `tupleIDs`, and 1.5-4.5X speedups for smaller ranges. On the Itanium 2 machine, our prefetching schemes achieve 1.22-1.42X speedups over the B^+-Tree for search, 1.04-1.30X speedups for insertion, and 1.19-1.31X speedups for deletion. For range scan, our external and internal jump-pointer array prefetching schemes achieve 1.22-2.58X speedups for smaller ranges, and 3.87-4.73X speedups when there are at least 1000 `tupleIDs` in the range. Note that the speedups for range scans on mature trees are larger than those after bulkloading because the leaf nodes are no longer contiguous in memory.



(a) Range scan varying prefetching distance

(b) Range scan varying chunk size

(c) Insertion varying chunk size

Figure 2.25: Sensitivity analysis.

We can now compare the insertion performance of pB$^+$-Trees versus the CSB$^+$-Tree. In [84], experiments on mature trees showed that the CSB$^+$-Tree could be 25% worse than the B$^+$-Tree in insertion performance. This is because the CSB$^+$-Tree requires the sibling nodes to be moved when a node splits. The pB$^+$-Trees achieve comparable of better insertion performance than the B$^+$-Tree, which is better than the CSB$^+$-Tree. Thus for modern memory systems, all three pB$^+$-Trees are significantly faster than the CSB$^+$-Tree, for all the main operations of an index structure.

2.4.9 Sensitivity Analysis

In our next set of experiments, we study the sensitivity of the p$_e$B$^+$-Tree performance gains to variations in (i) the prefetching distance d used, and (ii) the chunk size c used. Specifically, we study the sensitivity of scan performance to d and c, and the sensitivity of insertion performance to c. (Note that d and c, which are parameters for the prefetching scan, do not affect pB$^+$-Tree search performance.) In our previous experiments, $d = 3$, and $c = 3$ on the simulation platform.

Varying the Prefetching Distance. Figure 2.25(a) shows the effect on scan performance when the prefetching distance d varies from 1 to 5. The experiments are the same as in Figure 2.17(a). Clearly, $d = 1$ cannot efficiently exploit the parallelism in the memory system, and results in sub-optimal performance. $d = 2$ is actually very close to the computed value of d before applying the ceiling function, i.e. $\frac{B}{w} = \frac{16.7}{8}$. It achieves similar performance as $d = 3$, which is the computed value. Moreover, increasing d to 5 does not have adverse effects on range scan performance. Therefore, we conclude that the performance is not particularly sensitive to moderate increases in the prefetching distance.

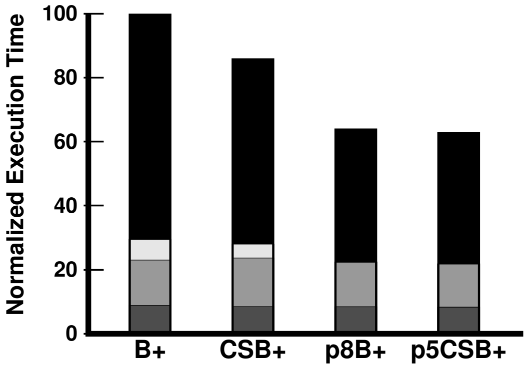
Varying the Chunk Size. Figure 2.25(b) shows the effect on scan performance when the chunk size c varies from 3 to 6. Figure 2.25(c) shows the effect of the chunk size on insertion performance. Figure 2.25(b) reports the same experiments as in Figure 2.17(a), while Figure 2.25(c) reports the same experiments as in Figure 2.19(b). We still limit the chunks in the external jump-pointer arrays to be at most 80% full upon bulkloading. From the figures, we see that increasing chunk size slightly to create some extra slacks does not incur performance degradation.

2.4.10 Cache Performance Breakdowns

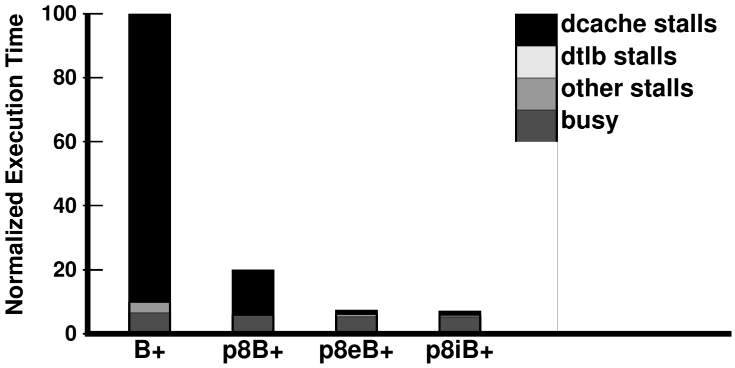
Our next set of experiments present a more detailed cache performance study, using two representative experiments: one for index search and one for index range scan. A central claim of this chapter is that the demonstrated speedups for pB$^+$-Trees are obtained by effectively limiting the exposed miss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

Our starting point is the experiments presented earlier in Figure 2.1 which illustrated the poor cache performance of existing B$^+$-Trees on index search and scan. We reproduce those results now in Figure 2.26, along with several variations of our pB$^+$-Trees. Figure 2.26(a) corresponds to the experiment shown earlier in Figure 2.13(b) with 10 million $\langle$`key`, `tupleID`$\rangle$ pairs bulk-loaded, and Figure 2.26(b) corresponds to the experiment shown earlier in Figure 2.17(a) with 1 million `tupleID`s scanned.

Each bar in Figure 2.26 represents execution time normalized to a B$^+$-Tree, and is broken down into four categories that explain what happened during all potential *graduation slots* (in the simulator). The number of graduation slots is the issue width (4 in our simulated architecture) multiplied by the number of cycles. We focus on graduation slots rather than issue slots to avoid counting speculative operations that are squashed. The bottom section (*busy*) of each bar is the number of slots where instructions actu-



(a) Search



(b) Range scan

Figure 2.26: Impact of various pB^+-Trees on the cache performance of index search and range scan.

ally graduate. The other three sections are the number of slots where there is no graduating instruction, broken down into data cache stalls, data TLB stalls, and other stalls. Specifically, the top section (*dcache stalls*) is the number of such slots that are immediately caused by the oldest instruction suffering a data cache miss, the second section (*dtlb*) is the number of slots that are caused by the oldest instruction waiting for a data TLB miss, and the third section (*other stalls*) is all other slots where instructions do not graduate. Note that the effects of L2 and L3 cache misses are included in the *dcache stalls* section. Moreover, the *dcache stalls* section is only a first-order approximation of the performance loss due to data cache misses: These delays also exacerbate subsequent data dependence stalls, thereby increasing the number of *other stalls*.

The cache performance breakdowns are generated based on our simulation results because the simulator has fine-grained instrumentations to categorize every idle graduation slot into a stall type. Note that it is difficult to generate accurate cache performance breakdowns on the Itanium 2 machine for two reasons: (i) The processor does not provide detailed information about graduation slots; (ii) Estimating the breakdowns using the number of cache misses and other event counts does not take into account the overlapping effect of these events.

As we see in Figure 2.26, pB^+-Trees significantly reduce the amount of exposed miss latency (i.e. the *dcache stalls* component of each bar). For the index search experiments, we see that while CSB^+-Trees eliminated 18% of the data cache stall time that existed with B^+-Trees, pB^+-Trees eliminate *41%*

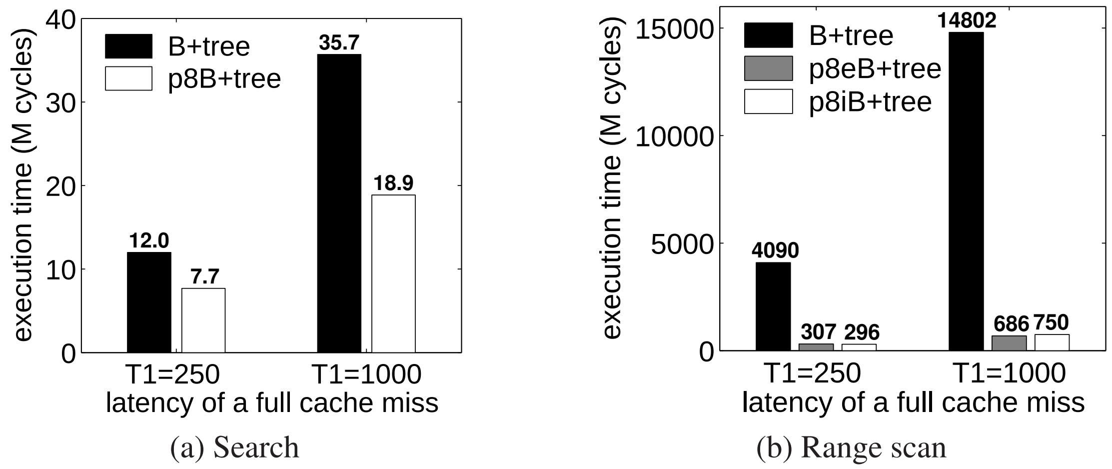


Figure 2.27: Impact of increased memory latency on the performance of index search and range scan.

of this stall time. Moreover, shallower trees because of wider nodes significantly reduce the number of data TLB misses experienced for node accesses. These two effects combined together result in an overall speedup of *1.56* (compared with 1.16 for CSB$^+$-Trees). A significant amount of data cache stall time still remains for index searches, since we still experience the full miss latency each time we move down a level in the tree (unless the node is already in the cache due to previous operations). Eliminating this remaining latency appears to be difficult, as we will discuss in the next section. In contrast, we achieve nearly ideal performance for the index range scan experiments shown in Figure 2.26(b), where both p_eB^+-Trees and p_iB^+-Trees eliminate *98%* of the original data cache stall time, resulting in an impressive *thirteen-fold overall speedup*. These results demonstrate that the pB$^+$-Tree speedups are indeed achieved primarily through a significant reduction in the exposed miss latency.

2.4.11 Impact of Larger Memory Latency

Finally, our last set of experiments investigate the impact of larger memory latency ($T_1 = 1000$) on the performance of index search and range scan. Figures 2.27(a) and (b) report the same experiments as in Figure 2.26 while varying the memory latency. We can see that the performance of B$^+$-Trees degrades dramatically as memory latency increases. In contrast, our pB$^+$-Trees exploit cache prefetching to successfully hide cache miss latencies within every node access for search, and within and across leaf node accesses for range scan, thus resulting in a more graceful performance degradation. As shown in Figure 2.27, the pB$^+$-Trees achieves a 1.89X speedup for search, and both p$_e$B$^+$-Trees and p$_i$B$^+$-Trees achieve over 19-fold speedups when the memory latency is increased to 1000 cycles.

2.5 Discussion and Related Work

We now discuss several issues related to prefetching B$^+$-Trees and the related work of prefetching B$^+$-Trees addressing these issues. Most of these studies occurred after our publication that proposed and evaluated prefetching B$^+$-Trees [18].

While our approach uses prefetching to create wider nodes for better search performance under any query and index conditions, we still suffer a full cache miss latency at each level of the tree. Unfortunately, it is very difficult to hide this cache miss latency without stronger assumptions about query patterns because of : (i) the data dependence through the child pointer; (ii) the relatively large fanout of the tree nodes; and (iii) the fact that it is equally likely that any child will be visited (assuming uniformly distributed random search keys). While one might consider prefetching the children or even the grandchildren of a node in parallel with accessing the node, there is a duality between this and simply creating wider nodes. Compared with our approach, prefetching children or grandchildren suffers from additional storage overhead for the children and grandchildren pointers. With stronger assumptions about query patterns, it may be possible to further optimize the cache performance of index searches. For example, Zhou and Ross [102] focused on situations where a large number of searches are performed in a batch and response times for the searches are less important than throughput. They proposed to buffer searches at non-root nodes so that multiple searches can share the cache misses of a single node.

Hankins and Patel [37] studied the node size of CSB$^+$-Trees on a Pentium III machine and concluded that it is desirable to use larger node size to reduce the number of tree levels because every level of the tree experiences a TLB miss and a fixed instruction overhead. Their findings corroborate our proposal for wider nodes.[4]

[4] Our experiments on the Itanium 2 machine, however, find that the optimal node size of CSB$^+$-Trees without cache prefetching is still one cache line. This may be because the Itanium 2 machine has quite different characteristics from the

The idea of increasing B^+-Tree node sizes was studied in the context of traditional disk-oriented databases. Lomet [63] proposed a technique, called elastic buckets, that allows disk-oriented B^+-Trees to use nodes of different sizes for better space utilization, which is orthogonal to our scheme of prefetching for wider nodes. Litwin and Lomet [60] proposed an access method, called bounded disorder, that increases leaf node sizes for disk-oriented B^+-Trees and builds hash tables for fast exact-match searches inside leaf nodes. While our prefetching B^+-Trees make only slight modifications to B^+-Tree algorithms, bounded disorder has to deal with complexities, such as hash bucket overflows, in leaf nodes. Moreover, searches in non-leaf nodes are not exact-matches, therefore cannot employ hash tables. In contrast, our prefetching B^+-Trees increase the size of both leaf and non-leaf nodes, thus capable of achieving larger reduction of tree heights. Furthermore, the scheme of bounded disorder can be combined with prefetching B^+-Trees for better performance.

Although we have described our range scan algorithm for the case when the `tupleIDs` are copied into a return buffer, other variations are only slightly more complex. For example, returning tuples instead of `tupleIDs` involves only the additional step of prefetching the tuple once a `tupleID` has been identified. Moreover, if the index store `tupleIDs` with duplicate keys by using separate lists for the multiple `tupleIDs`, our prefetching approach could be used to retrieve the addresses to the `tupleID` lists, then the `tupleIDs`, and finally the tuples themselves.

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

Concurrency control is also a significant issue. Cha *et al.* [15] proposed a concurrency control scheme for main memory B^+-Trees by exploiting an optimistic strategy. Since latches are typically stored in the B^+-Tree nodes, latching a node even for read-only access during a search operation involves writing to the node. This incurs expensive coherence cache misses when multiple readers are executing concurrently in a multiprocessor system. Their proposed scheme instead keeps a version number for every

Pentium III machine used in Hankins and Patel's study.

node. A read access remembers the version numbers of all the nodes it visits, and verifies whether the version numbers stay the same after the tree traversal. In this way, read accesses no longer need to perform memory writes in most cases, avoiding the coherence cache misses.

2.6 Chapter Summary

While eliminating child pointers through data layout techniques has been shown to significantly improve main memory B^+-Tree search performance, a large fraction of the execution time for a search is still spent in data cache stalls, and index insertion performance is hurt by these techniques. Moreover, the cache performance of index scan (another important B^+-Tree operation) has not been studied.

In this chapter, we explored how prefetching could be used to improve the cache performance of index search, update, and scan operations. We proposed the *Prefetching B^+-Tree* (pB^+-Tree) and evaluated its effectiveness both through simulations and on an Itanium 2 machine. Our experimental results demonstrate:

- The optimal main memory B^+-Tree node size is often wider than a cache line on a modern machine (e.g., eight lines), when prefetching is used to retrieve the pieces of a node, effectively overlapping multiple cache misses.

- Compared to the one-cache-line-node main memory B^+-Tree, our scheme of prefetching for wider nodes achieves 1.16-1.85X speedups for searches, and comparable or better update performance (up to 1.77X speedups).

- Compared to the CSB^+-Tree, our scheme achieves 1.08-1.53X speedups for searches because we can increase the fanout by more than the factor of two that CSB^+-Trees provide (e.g., by a factor of eight). Moreover, $pCSB^+$-Trees achieve even better performance on the Itanium 2 machine. Therefore, our scheme and CSB^+-Trees are complementary.

- For range scans, our jump-pointer array prefetching can effectively hide 98% of the cache miss latency suffered by one-cache-line-node main memory B^+-Trees, thus achieving a factor of up to

18.7X speedup over B^+-Trees.

- The techniques will still be effective even when the latency gap between processors and memory increases significantly in the future (e.g., by a factor of four).

In summary, the cache performance of main memory B^+-Tree indices can be greatly improved by exploiting the prefetching capabilities of state-of-the-art computer systems.

Chapter 3

Optimizing Both Cache and Disk Performance for B^+-Trees

3.1 Introduction

In Chapter 2, we have studied *prefetching B^+-Trees* for improving the CPU cache performance of main memory indices. Several recent studies[13, 84] have also considered B^+-Tree variants for indexing memory-resident data, and presented new types of B^+-Trees—*cache-sensitive B^+-Trees* [84], *partial-key B^+-Trees* [13]—that optimize for CPU cache performance by minimizing the impact of cache misses. These "cache-optimized" B^+-Trees are composed of nodes the size of a *cache line*[1] — i.e., the natural transfer size for reading from or writing to main memory. In contrast, to optimize I/O performance for indexing disk-resident data, traditional "disk-optimized" B^+-Trees are composed of nodes the size of a *disk page*—i.e., the natural transfer size for reading from or writing to disk.

Unfortunately, B^+-Trees optimized for disk suffer from poor CPU cache performance, and B^+-Trees optimized for cache suffer from poor I/O performance. This is primarily because of the large discrepancy in node sizes: Disk pages are typically 4KB–64KB while cache lines are often 32B–128B, depending on the system. Thus existing disk-optimized B^+-Trees suffer an excessive number of cache misses to

[1]In the case of *prefetching B^+-Trees* , the nodes are several cache lines wide.

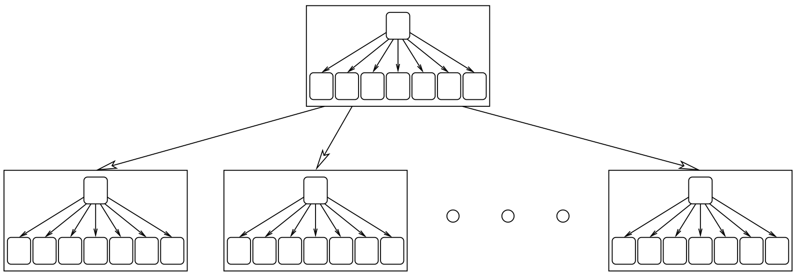


Figure 3.1: Self-similar “tree within a tree” structure.

search in a (large) node, wasting time and forcing the eviction of useful data from the cache. Likewise, existing cache-optimized B^+-Trees, in searching from the root to the desired leaf, may fetch a distinct page for each node on this path. This is a significant performance penalty, for the smaller nodes of cache-optimized B^+-Trees imply much deeper trees than in the disk-optimized cases (e.g., twice as deep). The I/O penalty for range scans on non-clustered indices of cache-optimized trees is even worse: A distinct page may be fetched for each leaf node in the range, increasing the number of disk accesses by the ratio of the node sizes (e.g., a factor of 500).

3.1.1 Our Approach: Fractal Prefetching B^+-Trees

In this chapter, we propose and evaluate *Fractal Prefetching B^+-Trees* (fpB^+-Trees), which are a new type of B^+-Tree that optimizes *both* cache and I/O performance. In a nutshell, an fpB^+-Tree is a single index structure that can be viewed at two different granularities: At a coarse granularity, it contains disk-optimized nodes that are roughly the size of a disk page, and at a fine granularity, it contains cache-optimized nodes that are roughly the size of a cache line. We refer to a fpB^+-Tree as being “fractal” because of its self-similar “tree within a tree” structure, as illustrated in Figure 3.1. The cache-optimized aspect is modeled after the *prefetching B^+-Trees* that we proposed in Chapter 2, which were shown to have the best main memory performance for fixed-size keys. (We note, however, that this general approach can be applied to any cache-optimized B^+-Tree.) In a prefetching B^+-Tree, nodes are several cache lines wide (e.g., 8—the exact number is tuned according to various memory system parameters), and cache prefetching is used so that the time to fetch a node is not much longer than the delay for a single cache miss.

We design and evaluate two approaches to implementing fpB$^+$-Trees: (i) *disk-first* and (ii) *cache-first*. In the *disk-first* approach, we start with a disk-optimized B$^+$-Tree, but then organize the keys and pointers within each page-sized node as a small tree. This *in-page tree* is a variant of the prefetching B$^+$-Tree. To pack more keys and pointers into an in-page tree, we use short in-page offsets rather than full pointers in all but the leaf nodes of an in-page tree. We also show the advantages of using different sizes for leaf versus non-leaf nodes in an in-page tree. In contrast, the *cache-first* approach starts with a cache-optimized prefetching B$^+$-Tree (ignoring disk page boundaries), and then attempts to group together these smaller nodes into page-sized nodes to optimize disk performance. Specifically, the cache-first approach seeks to place a parent and its children on the same page, and to place adjacent leaf nodes on the same page. Maintaining both structures as new keys are added and nodes split poses particular challenges. We will show how to process insertions and deletions efficiently in both disk-first and cache-first fpB$^+$-Trees. We select the optimal node sizes in both disk-first and cache-first approaches to maximize the number of entry slots in a leaf page while analytically achieving search cache performance within 10% of the best.

Ideally, both the *disk-first* and the *cache-first* approaches would achieve identical data layouts, and hence equivalent cache and I/O performance. In practice, however, the mismatch that almost always occurs between the size of a cache-optimized subtree and the size of a disk page (in addition to other implementation details such as full pointers versus page offsets) causes the disk-first and cache-first approaches to be slightly biased in favor of disk and cache performance, respectively. Despite these slight disparities, both implementations of fpB$^+$-Trees achieve dramatically better cache performance than disk-optimized B$^+$-Trees.

To accelerate range scans, fpB$^+$-Trees employ the *jump-pointer array* scheme as described previously in Section 2.3. A jump-pointer array contains the leaf node addresses of a tree, which are used in range scans to prefetch the leaf nodes, thus speeding up the scans. In Chapter 2, we have showed that this approach significantly improves CPU cache performance. In this chapter, we show it is also beneficial for I/O, by demonstrating a factor of 2.5–5 improvement in the range scan I/O performance for IBM's DB2 running on a multi-disk platform.

The remainder of this chapter is organized as follows. Section 3.2 describes how fpB$^+$-Trees enhance I/O performance. Then Section 3.3 describes how they enhance cache performance while preserving I/O performance. Section 3.4, 3.5, and 3.6 present experimental results through simulations and on real machines validating the effectiveness of fpB$^+$-Trees in optimizing both cache and disk performance. Section 3.7 and Section 3.8 discuss related work and issues related to fpB$^+$-Trees. Finally, Section 3.9 summarizes the findings in this chapter.

3.2 Optimizing I/O Performance

Fractal Prefetching B$^+$-Trees combine features of disk-optimized B$^+$-Trees and cache-optimized B$^+$-Trees to achieve the best of both structures. In this section, we describe how fpB$^+$-Trees improve I/O performance for modern database servers. In a nutshell, we consider applying to disk-resident data each of the techniques in Chapter 2 for improving the CPU cache performance for memory-resident data, which provides insight on the similarities and differences between the disk-to-memory gap and the memory-to-cache gap. We argue that while the techniques are not advantageous for search I/O performance, they can significantly improve range scan I/O performance.

Modern database servers are composed of multiple disks per processor. For example, many database TPC benchmark reports are for multiprocessor servers with 10-30 disks per processor, and hundreds of disks in all [98]. To help exploit this raw I/O parallelism, commercial database buffer managers use techniques such as sequential I/O prefetching and delayed write-back. While sequential I/O prefetching helps accelerate range scans on *clustered* indices, where the leaf nodes are contiguous on disk, it offers little or no benefit for range scans on *non-clustered* indices or for searches, which may visit random disk blocks. Our goal is to effectively exploit I/O parallelism by explicitly prefetching disk pages even when the access patterns are not sequential.

In Chapter 2, we proposed and evaluated *prefetching B^+-Trees* (pB$^+$-Trees) as a technique for enhancing CPU cache performance for index searches and range scans on memory-resident data. The question that we address now is whether those same techniques can be applied to accelerating I/O performance for disk-resident data. Since the relationship between main memory and disk for a disk-optimized tree is somewhat analogous to the relationship between CPU cache and main memory for a cache-optimized tree, one might reasonably expect the benefit of a technique to translate in at least some form across these different granularities [31]. However, because of the significant differences between these two granularities (e.g., disks are larger and slower, main memory is better suited to random access, etc.), we must carefully examine the actual effectiveness of a technique at a different granularity. In Sections 3.2.1 and 3.2.2, we consider the two aspects of pB$^+$-Trees which accelerate *searches* and *range scans*, respectively.

3.2.1 Searches: Prefetching and Node Sizes

To accelerate search performance, our pB$^+$-Tree design increased the size of a B$^+$-Tree node size to be multiple cache lines wide and prefetched all cache lines within a node before accessing it. In this way, the multiple cache misses of a single node are serviced in parallel, thereby resulting in an overall miss penalty that is only slightly larger than that of a single cache miss. The net result is that searches become faster because nodes are larger and hence trees are shallower.

For disk-resident data, the page-granularity counterpart is to increase the B$^+$-Tree node size to be a multiple of the disk page size and prefetch all pages of a node when accessing it. By placing the pages that make up a node on different disks, the multiple page requests can be serviced in parallel. For example, a 64KB node could be striped across 4 disks with 16KB page size, and read in parallel. As in the cache scenario, faster searches may result.[2]

However, there are drawbacks to applying this approach to disks. While the I/O latency is likely to improve for a single search, the I/O throughput may become worse because of the extra seeks for a node. In an OLTP (OnLine Transaction Processing) environment, multiple transactions can overlap their disk accesses, and the I/O throughput is often dominated by disk seek times; hence additional disk seeks because of multiple pages per node may degrade performance. Note that this is not a problem for cache

[2]Larger nodes can also be obtained by using multiple *contiguous* disk pages per node. However, this can be regarded as using larger disk pages, therefore it has already been considered in choosing optimal disk page sizes [32, 61].

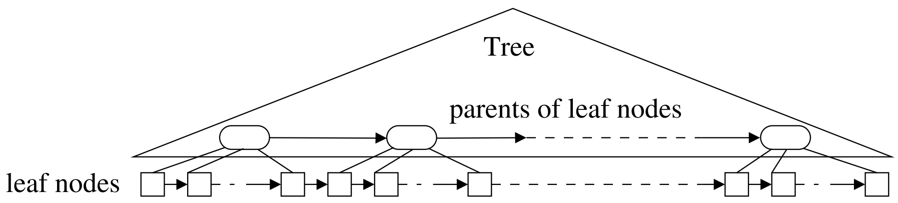


Figure 3.2: Internal jump-pointer array.

performance since only the currently executing thread can exploit its cache hierarchy bandwidth.

In a DSS (Decision Support System) environment, a server is often dedicated to a single query at a time, and hence latency determines throughput. Thus multipage-sized nodes spanning multiple disks may improve search performance. However, search times may be less important to overall DSS query times, which are often dominated by operations such as range scans, hash joins, etc. Moreover, "random" searches are often deliberately avoided by the optimizer. An indexed nested loop join may be performed by first sorting the outer relation on the join key [40, 30]. Thus each key lookup in the inner relation is usually adjacent to the last lookup, leading to an I/O access pattern that essentially traverses the tree leaf nodes in order (similar to range scans).

For these reasons, we do not advocate using multipage-sized nodes. Hence throughout this chapter, our target node size for optimizing the disk performance of fpB$^+$-Trees will be a *single* disk page.

3.2.2 Range Scans: Prefetching via Jump-Pointer Arrays

For range scan performance, we proposed a *jump-pointer array* structure in Section 2.3 that permits the leaf nodes in the range scan to be effectively prefetched. A range scan is performed by searching for the starting key of the range, then reading consecutive leaf nodes in the tree (following the sibling links between the leaf nodes) until the end key for the range is encountered. One implementation of the jump-pointer array is shown in Figure 3.2: An *internal* jump-pointer array is obtained by adding sibling pointers to each node that is a parent of leaf nodes. These leaf parents collectively contain the addresses for all leaf nodes, facilitating leaf node prefetching. By issuing a prefetch for each leaf node sufficiently far ahead of when the range scan needs the node, the cache misses for these leaf nodes are overlapped.

The same technique can be applied at page granularity to improve range scan I/O performance, by overlapping leaf page misses. It is particularly helpful in non-clustered indices and when leaf pages are *not* sequential on disks, a common scenario for frequently updated indices.[3] Note that the original technique prefetched past the end key. This overshooting cost is not a major concern at cache granularity; however, it can incur a large penalty at page granularity both because each page is more expensive to prefetch and because we must prefetch farther ahead in order to hide the larger disk latencies. To solve this problem, fpB$^+$-Trees begin by searching for both the start key and the end key, remembering the range end page. Then when prefetching using the leaf parents, we can avoid overshooting. Also note that because all the prefetched leaf pages would have also been accessed in a plain range scan, this technique does not decrease throughput.

This approach is applicable for improving the I/O performance of standard B$^+$-Trees, not just fractal ones, and as our experimental results will show, can lead to a five-fold or more speedup for large scans.

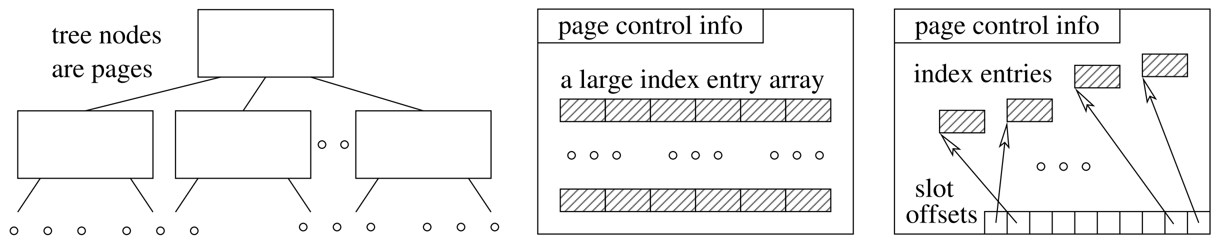
3.3 Optimizing CPU Cache Performance

In this section, we describe how fpB$^+$-Trees optimize CPU cache performance without sacrificing their I/O performance. We begin by analyzing the cache behavior for searches in traditional B$^+$-Trees in Section 3.3.1. Then we discuss the problems with previous cache-friendly approaches for traditional B$^+$-Trees in Section 3.3.2. We propose to break disk-optimized pages into cache-optimized trees and describe two approaches: *disk-first* and *cache-first*. Section 3.3.3 describes the disk-first approach, while Section 3.3.4 describes the cache-first approach, both focusing on searches and updates. Finally, in Section 3.3.5, we discuss range scans for both approaches.

3.3.1 Why Traditional B$^+$-Trees Suffer from Poor Cache Performance?

In traditional disk-optimized B$^+$-Trees, each tree node is a page (typically 4KB–64KB), as depicted in Figure 3.3(a). Figures 3.3(b) and (c) illustrate two page organization schemes for disk-optimized B$^+$-

[3]For clustered indices or when leaf pages are sequential on disks, sequential I/O prefetching can be employed instead.



(a) Tree structure (b) Page organization 1: index entry array in page (c) Page organization 2: slotted page

Figure 3.3: Disk-Optimized B^+-Tree page organizations. (An index entry is a pair of key and tupleID for leaf pages, and a pair of key and pageID for non-leaf pages.)

Trees. A small part of a page contains page control information. The two schemes differ in how the index entries are stored in the bulk of the page. In the first scheme, the bulk of the page contains a sorted array of keys, together with either the `pageID` for its child node (if the node is a non-leaf) or the `tupleID` for a tuple (if the node is a leaf), assuming fixed length keys.[4] We will refer to a key and either its `pageID` or `tupleID` as an *entry*. The second scheme avoids the contiguous array organization by using an offset array at the end of the page to point to index entries in the page. In this way, index entries can be in arbitrary locations in a page, thus reducing data movements for updates. Because of this, the slotted page organization is more common in practice. We use the first page organization in this chapter mainly for better understanding and explaining the cache behavior of other schemes.

We first examine the cache behavior of the simpler page organization with arrays to obtain insights for better understanding the cache behavior of the slotted page organization. During a search, each page on the path to the key is visited, and a binary search is performed on the very large contiguous array in the page. This binary search is quite costly in terms of cache misses. A simple example helps to illustrate this point. If the key size, `pageID` size, and `tupleID` size are all 4 bytes, an 8KB page can hold over 1000 entries. If the cache line size is 64 bytes, then a cache line can only hold 8 entries. Imagine a certain page has 1023 entries numbered 1 through 1023. To locate a key matching entry 71, a binary search will perform ten probes, for entries 512, 256, 128, 64, 96, 80, 72, 68, 70, and 71, respectively.

[4]The issues and solutions for fixed length keys are also important for variable length keys, which have their own added complications in trying to obtain good cache performance [13]. We will discuss variable length keys in Section 3.8.

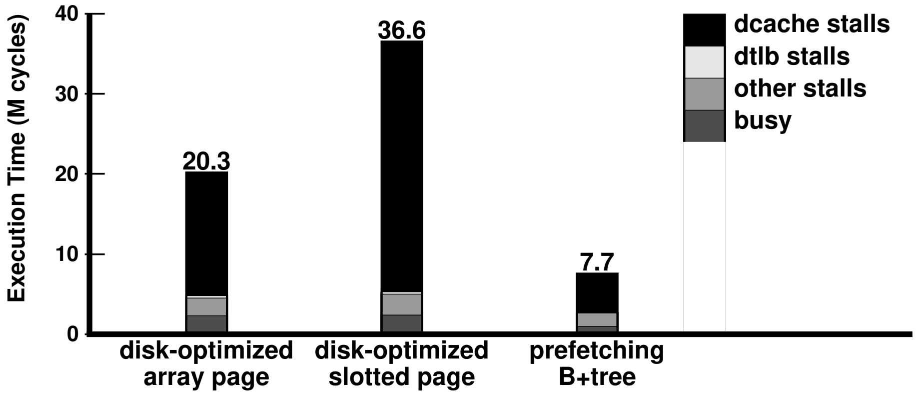


Figure 3.4: Comparing search cache performance of disk-optimized B$^+$-Trees with index entry arrays, disk-optimized B$^+$-Trees with slotted pages, and prefetching B$^+$-Trees (with eight-line-wide nodes).

Assuming that the eight entries from 65 to 72 fall within a single cache line, the first seven probes are all likely to suffer cache misses. The first six of the seven misses are especially wasteful, since each of them brings in a 64B cache line but uses only 4B of that line. Only when the binary search finally gets down to within a cache line are more data in a cache line used. This lack of spatial locality makes binary search on a very large array suffer from poor cache performance.

This poor spatial locality is further aggravated in the slotted page organization for two reasons. First, suppose for a moment the index entries in a slotted page are stored in key order. Then a binary search in the slotted page essentially visits two arrays, the slot offset array and the index entry "array", experiencing poor spatial localities at both arrays. Second, if the index entries are out of order, then the spacial locality within a cache line disappears. Even if the previous step tests a key immediately next to the current key in key order, their actual locations in the page can be far apart. Therefore, every key comparison is likely to incur an expensive cache miss.

Figure 3.4 compares the performance of disk-optimized B$^+$-Trees with cache-optimized prefetching B$^+$-Trees for searches. The figure shows the simulated execution time for performing 10,000 random searches after each tree has been bulkloaded with 10 million keys on a memory system similar to Itanium 2. (The simulator is described previously in Section 2.4.2.) Execution time is broken down into busy time, data cache stalls, data TLB stalls, and other stalls. As we see in Figure 3.4, disk-optimized B$^+$-

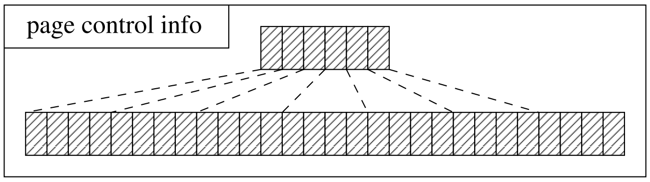


Figure 3.5: Illustration of micro-indexing.

Trees spend significantly more time stalled on data cache misses than prefetching B+-Trees.[5] Moreover, the cache performance of the slotted page organization is significantly worse than that of the array page organization. Interestingly, TLB stalls have little impact on the performance of disk-optimized B+-Trees because index entries are stored in pages.

3.3.2 Previous Approach to Improving B+-Tree Cache Performance

One approach that was briefly mentioned by Lomet [62] is *micro-indexing*, which is illustrated in Figure 3.5. The idea behind micro-indexing is that the first key of every cache line in the array can be copied into a smaller array, such as keys $1, 9, 17, \ldots, 1017$ in the example above. These 128 keys are searched first to find the cache line that completes the search (thus reducing the number of cache misses to five in the example). Unfortunately this approach has to use a contiguous array to store index entries. Like disk-optimized B+-Trees with index entry arrays, it suffers poor update performance. In order to insert an entry into a sorted array, half of the page (on average) must be copied to make room for the new entry. To make matters worse, the optimal disk page size for B+-Trees is increasing with disk technology trends [32, 61], making the above problems even more serious in the future.

To realize good cache performance for all B+-Tree operations, we look to cache-optimized B+-Trees as a model and propose to break disk-sized pages into cache-optimized nodes. This is the guiding principle behind fpB+-Trees. In the following, we propose and evaluate two approaches for embedding cache-optimized trees into disk-optimized B+-Tree pages: *disk-first* and *cache-first*.

[5]The extra "busy" time for disk-optimized B+-Trees is due to the instruction overhead associated with buffer pool management.

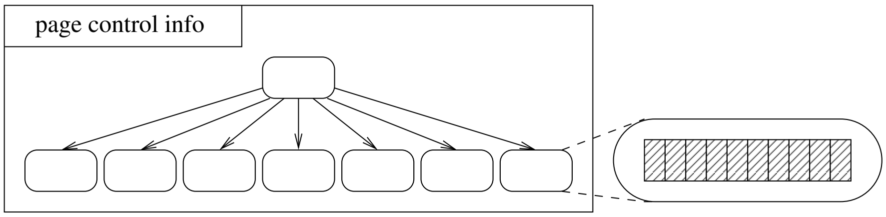


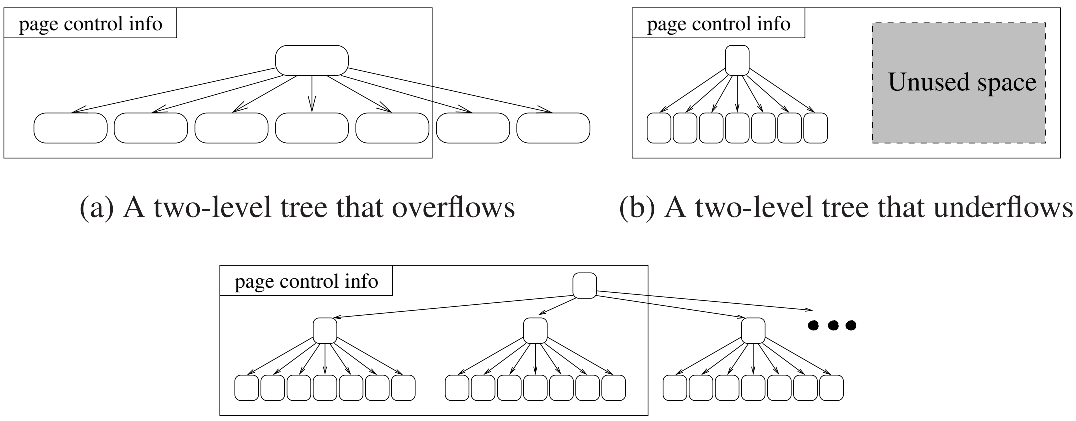
Figure 3.6: Disk-First fpB$^+$-Tree: a cache-optimized tree inside each page.

3.3.3 Disk-First fpB$^+$-Trees

Disk-First fpB$^+$-Trees start with a disk-optimized B$^+$-Tree, but then organize the keys and pointers in each page-sized node into a cache-optimized tree, which we call an *in-page tree*, as shown in Figure 3.6. Our in-page trees are modeled after pB$^+$-Trees, because they were shown to have the best cache performance for memory-resident data with fixed-length keys. The approach, however, can be applied to any cache-optimized tree.

As in a pB$^+$-Tree, an fpB$^+$-Tree in-page tree has nodes that are aligned on cache line boundaries. Each in-page node is several cache lines wide. When an in-page node is to be visited as part of a search, all the cache lines comprising the node are *prefetched*. That is, the prefetch requests for these lines are issued one after another without waiting for the earlier ones to complete. Let T_1 denote the full latency of a cache miss and T_{next} denote the latency of an additional pipelined cache miss. Then $T_1 + (w-1) \cdot T_{\text{next}}$ is the cost for servicing all the cache misses for a node with w cache lines. Because on modern processors, T_{next} is much less than T_1, this cost is only modestly larger than the cost for fetching one cache line. On the other hand, having multiple cache lines per node increases its fan-out, and hence can reduce the height of the in-page tree, resulting in better overall performance, as detailed previously in Chapter 2.

Disk-First fpB$^+$-Trees have two kinds of in-page nodes: leaf nodes and non-leaf nodes. Their roles in the overall tree (the disk-optimized view) are very different. While in-page non-leaf nodes contain pointers to other in-page nodes *within the same page*, in-page leaf nodes contain pointers to nodes *external to their in-page tree*. Thus, for in-page non-leaf nodes, we pack more entries into each node by using short in-page offsets instead of full pointers. Because all in-page nodes are aligned on cache line boundaries, the offsets can be implemented as a node's starting cache line number in the page. For



(a) A two-level tree that overflows (b) A two-level tree that underflows

(c) Adding a third level to the tree in (b) causes an overflow

Figure 3.7: The node size mismatch problem.

example, if the cache line is 64 bytes, then a 2 byte offset can support page sizes up to 4MB. On the other hand, in-page leaf nodes contain child `pageIDs` if the page is not a leaf in the overall tree, and `tupleIDs` if the page is a leaf.

The Node Size Mismatch Problem. Considering cache performance only, there is an optimal in-page node size, determined by memory system parameters and key and pointer sizes. Ideally, in-page trees based on this optimal size fit tightly within a page. However, the optimal page size is determined by I/O parameters and disk and memory prices [32, 61]. Thus there is likely a mismatch between the two sizes, as depicted in Figure 3.7. Figure 3.7(a) shows an overflow scenario in which a two-level tree with cache-optimal node sizes fails to fit within the page. Figure 3.7(b) shows an underflow scenario in which a two-level tree with cache-optimal node sizes occupies only a small portion of a page, but a three-level tree, as depicted in Figure 3.7(c), overflows the page. Note that because the fanout of a cache-optimized node is on the order of 10s, the unused space in the underflow scenario can be over 90% of the entire page. Thus, in most cases, we must give up on having trees with cache-optimal node sizes, in order to fit within the page. (Section 3.3.4 describes an alternative "cache-first" approach that instead gives up on having the cache-optimized trees fit nicely within page boundaries.)

Determining Optimal In-page Node Sizes. Our goals are to optimize search performance and to

(a) Use smaller nodes when overflow (b) Use larger nodes when underflow

Figure 3.8: Fitting cache-optimized trees in a page.

maximize page fan-out for I/O performance. To solve the node size mismatch problem, we give up using cache-optimal node sizes in disk-first fpB$^+$-Trees. In addition, we propose to allow different node sizes for different levels of the in-page tree. As shown in Figure 3.8, to combat overflow, we can reduce the root node (or restrict its fan-out) as in Figure 3.8(a). Similarly, to combat underflow, we can extend the root node so that it can have more children, as in Figure 3.8(b).

However, allowing arbitrarily many sizes in the same tree will make index operations too complicated. To keep operations manageable, noting that we already have to deal with different non-leaf and leaf node structures, we instead develop an approach that permits two node sizes for in-page trees: one for leaf nodes and one for non-leaf nodes. As we shall see, this flexibility is sufficient to achieve our goals.

At a high-level, there are three variables that we can adjust to achieve the goals: the number of levels in the in-page tree (denoted L), the number of cache lines of the non-leaf nodes (denoted w) and the number of cache lines of the leaf nodes (denoted x). Here we determine the optimal node sizes for an in-page tree, given the hardware parameters and the page size. Assume we know T_1 is the full latency of a cache miss, and T_{next} is the latency of an additional pipelined (prefetched) cache miss. Then the cost of searching through an L level in-page tree is

$$cost = (L-1)[T_1 + (w-1)T_{next}] + T_1 + (x-1)T_{next} \tag{3.1}$$

We want to select L, w, and x so as to minimize *cost* while maximizing page fan-out.

However, these two goals are conflicting. Moreover, we observed experimentally that because of fixed costs such as instruction overhead, small variations in *cost* resulted in similar search performance. Thus, we combine the two optimization goals into one goal $\mathcal{G}$: maximize the page fan-out while maintaining the analytical search cost to be within 10% of the optimal.

Now we simply enumerate all the reasonable combinations of w and x (e.g., 1-32 lines, thus $32^2 = 1024$ combinations). For each combination, we compute the maximum L that utilizes the most space in the page, which in turn allows *cost* and fan-out to be computed. Then we can apply $\mathcal{G}$ and find the optimal node widths. Table 3.1 in Section 3.4 and Table 3.2 in Section 3.5 depict the optimal node widths used in our experiments. Note that the optimal decision is made only once when creating an index. Therefore, the cost of enumeration is small.

Modifications to Index Operations.

Bulkload. Bulkloading a tree now has operations at two granularities. At a page granularity, we follow the common B+-Tree bulkload algorithm with the maximum fan-out computed by our previous computations. Inside each page, we bulkload an in-page tree using a similar bulkload algorithm. For in-page trees of leaf pages, we try to distribute entries across all in-page leaf nodes so that insertions are more likely to find empty slots. But for non-leaf pages, we simply pack entries into one in-page leaf node after another. We maintain a linked list of all in-page leaf nodes of leaf pages in the tree, in order.

Search. Two granularities, but straightforward.

Insertion. Insertion is also composed of operations at two granularities. If there are empty slots in the in-page leaf node, we insert the entry into the sorted array for the node, by copying the array entries with larger key values to make room for the new entry. Otherwise, we need to split the leaf node into two. We first try to allocate new nodes in the page. If there is no space for splitting up the in-page tree, but the total number of entries in the page is still far fewer than the page maximum fan-out, we reorganize the in-page tree and insert the entry to avoid expensive page splits. But if the total number of entries is quite close to the maximum fan-out (fewer than an empty slot per in-page leaf node), we split the page by copying half of the in-page leaf nodes to a new page and then rebuilding the two in-page trees in their respective pages.

Deletion. Deletion is simply a search followed by a lazy deletion of an entry in a leaf node, in which we copy the array entries with larger key values to keep the array contiguous, but we do not merge leaf nodes that become half empty.

3.3.4 Cache-First fpB$^+$-Trees

Cache-First fpB$^+$-Trees start with a cache-optimized B$^+$-Tree, ignoring page boundaries, and then try to intelligently place the cache-optimized nodes into disk pages. The tree node has the common structure of a cache-optimized B$^+$-Tree node: A leaf node contains an array of keys and `tupleIDs`, while a non-leaf node contains an array of keys and pointers. However, the pointers in non-leaf nodes are different. Since the nodes are to be put into disk pages, a pointer is a combination of a `pageID` and an offset in the page, which allows us to follow the `pageID` to retrieve a disk page and then visit a node in the page by its offset. Nodes are aligned on cache line boundaries, so the in-page offset is the node's starting cache line number in the page

We begin by describing how to place nodes into disk pages in a way that will minimize the structure's impact on disk I/O performance. Then we present our bulkload, insertion, search, and deletion algorithms for cache-first fpB$^+$-Trees.

Node Placement. There are two goals in node placement: (i) group sibling leaf nodes together into the same page so that range scans incur fewer disk operations, and (ii) place a parent node and its children into the same page so that searches only need one disk operation for a parent and its child.

To satisfy the first goal, we designate certain pages as leaf pages, which contain only leaf nodes. The leaf nodes in the same leaf page are siblings. This design ensures good range scan I/O performance.

Clearly, the second goal cannot be satisfied for all nodes, because only a limited number of nodes fit within a page. Moreover, the node size mismatch problem (recall Figure 3.7) means that placing a parent and its children in a page usually results in either an overflow or an underflow for that page. We can often transform a large underflow into an overflow by placing the grandchildren, the great grandchildren, and so on in the same page, until we incur either only a modest underflow (in which case we are satisfied with the placement) or an overflow (see Figures 3.7(b) and (c)).

There are two approaches for dealing with the overflow. First, an overflowed child can be placed into its own page to become the top-level node in that page. We then seek to place its children in the same page. This aggressive placement helps minimize disk accesses on searches. Second, an overflowed child

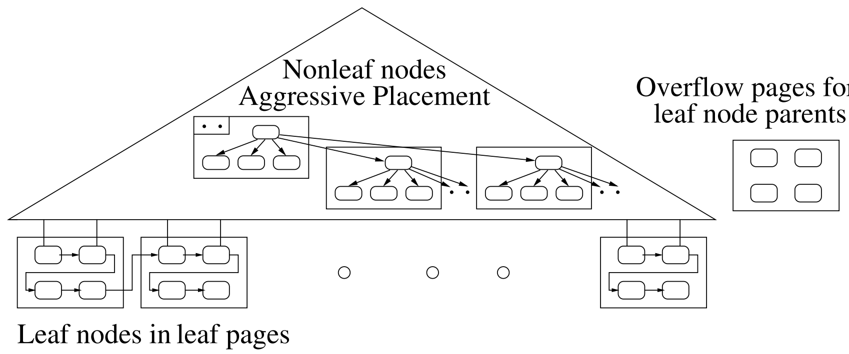


Figure 3.9: Cache-First fpB$^+$-Tree design.

can be stored in special overflow pages. This is the only reasonable solution for overflowed leaf parent nodes, because their children are stored in leaf pages.

Our node placement scheme is summarized in Figure 3.9. For non-leaf nodes, we use the aggressive node placement for good search performance, except for leaf parents, which use overflow pages. Leaf nodes are stored in leaf-only pages, for good range scan performance.

Algorithms. When creating the index, we determine the optimal node widths for cache performance by applying the same optimization goal $\mathcal{G}$ used in the disk-first approach. The search cost can be computed similar to prefetching B$^+$-Trees, which is described previously in Section 2.2.2. The only difference is that searches in cache-first fpB$^+$-Trees incur fewer number of TLB misses due to the aggressive placement of nodes into pages. The average number of page accesses can be estimated given the tree structure. Table 3.1 in Section 3.4 depicts the optimal node widths used in our experiments.

We now consider each of the index operations.

- **Bulkload.** We focus on how to achieve the node placement depicted in Figure 3.9. Leaf nodes are simply placed consecutively in leaf pages, and linked together with sibling links, as shown in the figure. Non-leaf nodes are placed according to the aggressive placement scheme, as follows.

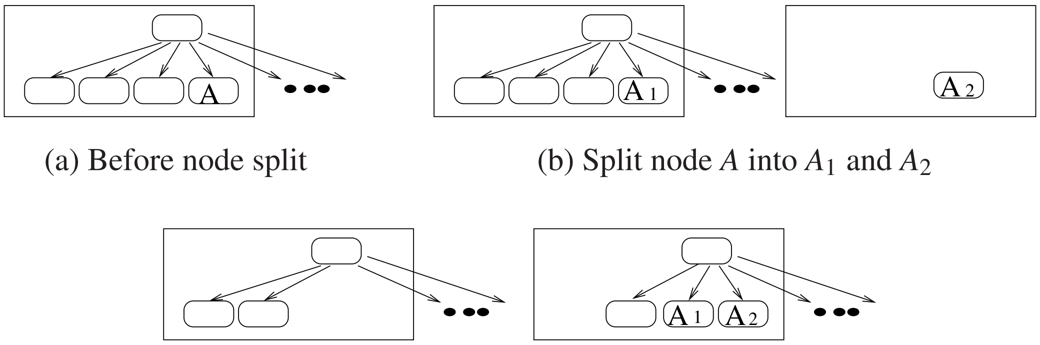
  First, we compute (i) the maximum number of levels of a full subtree that fit within a page, and (ii) the resulting underflow for such a subtree, i.e., how many additional nodes fit within the page. For example, if each node in the full subtree has 69 children, but a page can hold only 23 nodes, then only one level fits completely and the resulting underflow is 22 nodes. We create a bitmap with one bit for each child (69 bits in our example), and set a bit for each child that is to be placed with the parent (22 bits in our example, if we are bulkloading 100% full). We spread these set bits as evenly as possible within the bitmap.

As we bulkload nodes into a page, we keep track of each node's relative level in the page, denoted its *in-page level*. The in-page level is stored in the node header. The top level node in the page has in-page level 0. To place a non-leaf node, we increment its parent's in-page level. If the resulting level is less than the maximum number of in-page levels, the non-leaf node is placed in the same page as its parent, as it is part of the full subtree. If the resulting level is equal to the maximum number of in-page levels, the node is placed in the same page if the corresponding bit in the bit mask is set. If it is not set, the non-leaf node is allocated as the top level node in a new page, unless the node is a leaf parent node, in which case it is placed into an overflow page.

- **Insertion.** For insertion, if there are empty slots in the leaf node, the new entry is simply inserted. Otherwise, the leaf node needs to be split into two. If the leaf page still has spare node space, the new leaf node is allocated within the same page. Otherwise, we split the leaf page by moving the second half of the leaf nodes to a new page and updating the corresponding child pointers in their parents. (To do this, we maintain in every leaf page a back pointer to the parent node of the first leaf node in the page, and we connect all leaf parent nodes through sibling links.) Having performed the page granularity split, we now perform the cache granularity split, by splitting the leaf node within its page.

  After a leaf node split, we need to insert an entry into its parent node. If the parent is full, it must first be split. For leaf parent nodes, the new node may be allocated from overflow pages. But if further splits up the tree are necessary, each new node must be allocated according to our aggressive placement scheme.

  Figure 3.10 helps illustrate the challenges. We need to split node A, a non-leaf node whose children are non-leaf nodes, into two nodes A_1 and A_2, but there is no space in A's page for the additional node. As shown in Figure 3.10(b), a naïve approach is to allocate a new page for A_2. However,



(a) Before node split

(b) Split node A into A_1 and A_2

(c) Split the page into two

Figure 3.10: Non-leaf node splits.

A_2's children are half of A's children, which are all top level nodes in other pages. Thus either A_2 is the only node in the new page, which is bad for I/O performance and space utilization, or we must move A_2's children up into A_2's page, which necessitates promoting A_2's grandchildren to top level nodes on their own pages, and so on. Instead, to avoid the drawbacks of both these options, we split A's page into two, as shown in Figure 3.10(c).

- **Search.** Search is quite straightforward. One detail is worth noting. Each time the search proceeds from a parent to one of its children, we compare the `pageID` of the child pointer with that of the parent page. If the child is in the same page, we can directly access the node in the page without retrieving the page from the buffer manager.

- **Deletion.** Similar to disk-first fpB+-Trees.

3.3.5 Improving Range Scan Cache Performance

For range scans, we employ jump-pointer array prefetching, as described in Section 3.2.2, for both I/O and cache performance. We now highlight some of the details.

In disk-first fpB+-Trees, both leaf pages and leaf parent pages have in-page trees. For I/O prefetching, we build an internal jump-pointer array by adding sibling links between all in-page leaf nodes that are in leaf parent pages, because collectively these nodes point to all the leaf pages. For cache prefetching,

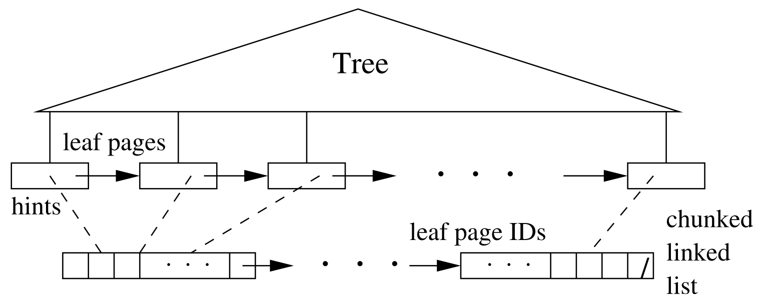


Figure 3.11: External jump-pointer array.

we build a second internal jump-pointer array by adding sibling links between all in-page leaf parent nodes that are in leaf pages, because collectively these nodes point to all the leaf nodes of the overall tree (i.e., all in-page nodes containing `tupleIDs`). In both jump-pointer arrays, sibling links within a page are implemented as page offsets and stored in the nodes, while sibling links across page boundaries are implemented as `pageIDs` and stored in the page headers.

In cache-first fpB$^+$-Trees, leaf pages contain only leaf nodes, while leaf parent pages can be either in the aggressive placement area or in overflow pages. Thus at both the page and cache granularities, sibling links between leaf parents may frequently cross page boundaries (e.g., a sequence of consecutive leaf parents may be in distinct overlap pages). Thus the internal jump-pointer array approach is not well suited for cache-first fpB$^+$-Trees. Instead, as shown in Figure 3.11, we maintain an *external* jump-pointer array that contains the page IDs for all the leaf pages, in order to perform I/O prefetching. Similarly, for cache prefetching, we could maintain in each leaf page header an external jump-pointer array, which contains the addresses of all nodes within the page. Instead, we observe that our in-page space management structure indicates which slots within a page contain nodes, and hence we can use it to prefetch all the leaf nodes in a page before doing a range scan inside the page.

3.4 Cache Performance through Simulations

In Chapter 2, we have verified our simulation model by comparing the simulation results with real machine results on the Itanium 2 machine. Since the simulation platform provides more instrumentations and mechanisms for us to better understand the cache behavior of a program, we take a different approach to presenting the results in this chapter. We first perform detailed cycle-by-cycle simulations to evaluate and understand the cache performance of fpB^+-Trees in Section 3.4. Then in Section 3.5, we verify our findings on the Itanium 2 machine. Section 3.6 evaluates the I/O performance of fpB^+-Trees.

3.4.1 Experimental Setup

We evaluate the CPU cache performance of fpB^+-Trees through detailed simulations of fully-functional executables running on a state-of-the-art machine. The simulator is described previously in Section 2.4.2. The memory hierarchy of the simulator is based on the Itanium 2 processor [44].

We implemented five index structures: (i) disk-optimized B^+-Trees with slotted pages, (ii) disk-optimized B^+-Trees with arrays in pages, (iii) micro-indexing, (iv) disk-first fpB^+-Trees, and (v) cache-first fpB^+-Trees. For disk-optimized B^+-Trees, our two implementations differ in their page structures. In disk-optimized B^+-Trees with slotted pages, a page contains a slot offset array at the end, which points to every index entry (a key and a `pageID` or `tupleID`) in the page. Index entries are not necessarily contiguous in a page. In contrast, the second implementation of disk-optimized B^+-Trees contains a large contiguous key array, and a contiguous array of `pageIDs` or `tupleIDs` in every page. Our experiments will show that while the former implementation has lower update cost, it has worse search performance. Since the slotted page organization is more common in practice, we use this tree structure as the baseline for all our comparisons. We study the implementation with arrays in pages mainly for the purpose of better understanding the behavior of the slotted page implementation and micro-indexing.

We implemented bulkload, search, insertion, deletion, and range scan operations for all the trees (range scans for micro-indexing was not explicitly implemented because its behavior is similar to that of disk-optimized B^+-Trees with arrays in pages). We implemented the index structures on top of our buffer manager, which uses the CLOCK algorithm to do page replacement.

We use 4 byte keys (except in Section 3.4.7, where we use 20 byte keys), 4 byte `pageIDs`, 4 byte `tupleIDs`, and 2 byte in-page offsets. We partitioned keys and pointers into separate arrays in all tree nodes (except for disk-optimized B^+-Trees with slotted pages) for better cache performance [31, 62].

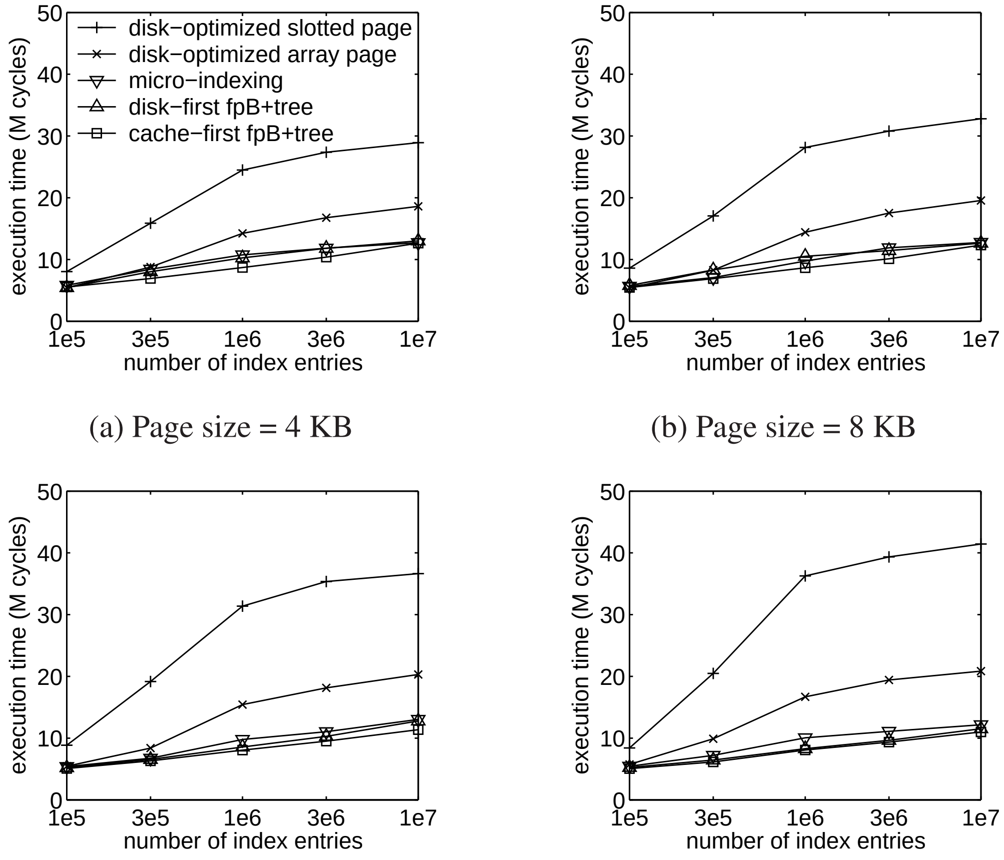
Table 3.1: Optimal width selections (4 byte keys, $T_1 = 250$, $T_{\text{next}} = 15$).

	Disk-First fpB$^+$-Trees				Cache-First fpB$^+$-Trees			Micro-Indexing		
page size	non-leaf node	leaf node	page fan-out	$\frac{\text{cost}}{\text{optimal}}$	node size	page fan-out	$\frac{\text{cost}}{\text{optimal}}$	subarray size	page fan-out	$\frac{\text{cost}}{\text{optimal}}$
4KB	64	384	470	1.00	576	497	1.01	128	496	1.01
8KB	192	256	961	1.01	576	994	1.01	192	1008	1.01
16KB	128	768	1995	1.05	960	2023	1.00	320	2032	1.02
32KB	256	832	4017	1.02	960	4046	1.00	320	4064	1.01

Disk-First fpB$^+$-Trees have 2 byte in-page pointers in non-leaf nodes and 4 byte pointers in leaf nodes, while cache-first fpB$^+$-Trees have 6 byte pointers combining `pageIDs` and in-page offsets in non-leaf nodes. We performed experiments for page sizes of 4KB, 8KB, 16KB, and 32KB, which covers the range of page sizes in most of today's database systems. As shown in Table 3.1, we computed optimal node widths for fpB$^+$-Trees using $T_1 = 250$ and $T_{next} = 15$ when key size is 4 bytes. The slowdowns relative to the optimal performance show the measurements in our experiments, as will be described in Section 3.4.2. We can see that the optimal width selections all satisfy our optimization criterion $\mathcal{G}$, i.e. maximizing leaf page fan-out while achieving within 10% of the best search performance.

In our micro-indexing implementation, a tree page contains a header, a micro-index, a key array, and a pointer array. The micro-index is formed by dividing the key array into sub-arrays of the same size and copying the first keys of the sub-arrays into the micro-index. A search in a page first looks up the micro-index to decide which sub-array to go to and then searches that sub-array. For better performance, we require the sub-array size to be a multiple of the cache line size (if applicable) and align the key array at cache line boundaries. To improve the performance of micro-indexing, we employ pB$^+$-Tree-like prefetching for micro-indices, key sub-arrays, and pointer sub-arrays. Insertion and deletion follow the algorithms of disk-optimized B$^+$-Trees, but then rebuild the affected parts of the micro-index. As shown in Table 3.1, we computed the optimal sub-array sizes for micro-indexing based on the same optimal criterion $\mathcal{G}$ as advocated for fpB$^+$-Trees.

We try to avoid conflict cache misses in the buffer manager between buffer control structures and



(a) Page size = 4 KB

(b) Page size = 8 KB

(c) Page size = 16 KB

(d) Page size = 32 KB

Figure 3.12: 10,000 random searches in trees that are 100% full.

buffer pool pages. The control structures are allocated from the buffer pool itself, and only those buffer pages that do not conflict with the control structures will be used. In fpB$^+$-Trees, putting top-level in-page nodes at the same in-page position would cause cache conflicts among them. So we instead place them at different locations determined by a function of the `pageIDs`.

3.4.2 Search Cache Performance through Simulations

Varying the Number of Entries in Leaf Pages. Figures 3.12 and 3.13 show the execution times of 10,000 random searches after bulkloading 1e5, 3e5, 1e6, 3e6, and 1e7 keys into the trees (nodes are

100% full except the root). All caches are cleared before the first search, and then the searches are performed one immediately after another. The four plots in Figure 3.12 show search performance when the database page sizes are 4KB, 8KB, 16KB, and 32KB, respectively. The fpB$^+$-Trees and micro-indexing use the corresponding optimal widths in Table 3.1. From the figures, we see that the cache-sensitive schemes, fpB$^+$-Trees and micro-indexing, all perform significantly better than disk-optimized B$^+$-Trees. Compared to disk-optimized B$^+$-Trees with slotted nodes, the cache-sensitive schemes achieve speedups between 1.38 and 4.49 at all points and between 2.22 and 4.49 when the trees contain at least 1 million entries. Moreover, comparing the two disk-optimized B$^+$-Tree implementations, the slotted page organization is significantly worse because searches incur cache misses at both slot offset arrays and index entries. Furthermore, comparing the three cache-sensitive schemes, we find their performance more or less similar. The two fpB$^+$-Tree schemes are slightly better than micro-indexing, and the cache-first fpB$^+$-Tree is the best of all three cache-sensitive schemes in most experiments.

When the page size increases from 4KB to 32KB, the performance of disk-optimized B$^+$-Trees becomes slightly worse. While larger leaf pages cause more cache misses at the leaf level, this cost is partially compensated by the savings at the non-leaf levels: Trees become shallower and/or root nodes have fewer entries. At the same time, fpB$^+$-Trees and micro-indexing perform better because larger page sizes leave more room for optimization. With the two trends, we see larger speedups over disk-optimized B$^+$-Trees with slotted nodes: over 2.81 for 16KB pages, and over 3.41 for 32KB pages, when trees contain at least 1 million entries.

Varying Node Width. Figure 3.13 compares the performance of different node widths for fpB$^+$-Trees and micro-indexing when the page size is 16KB. Our optimal criterion is to maximize leaf page fan-out while keeping analytical search performance within 10% of the best. Figure 3.13 confirms that our selected trees indeed achieve search performance very close to the best among the node choices. The experiments reported here correspond to the experiments in Figure 3.12(c) with 10 million keys in trees. Figures 3.13(a)–(c) show the node sizes for minimal execution times, the optimal widths selected, and all the other node sizes that have analytical search performance within 10% of the best. As shown in Figure 3.13, our selected optimal trees all perform within 5% of the best. (Table 3.1 also reports the

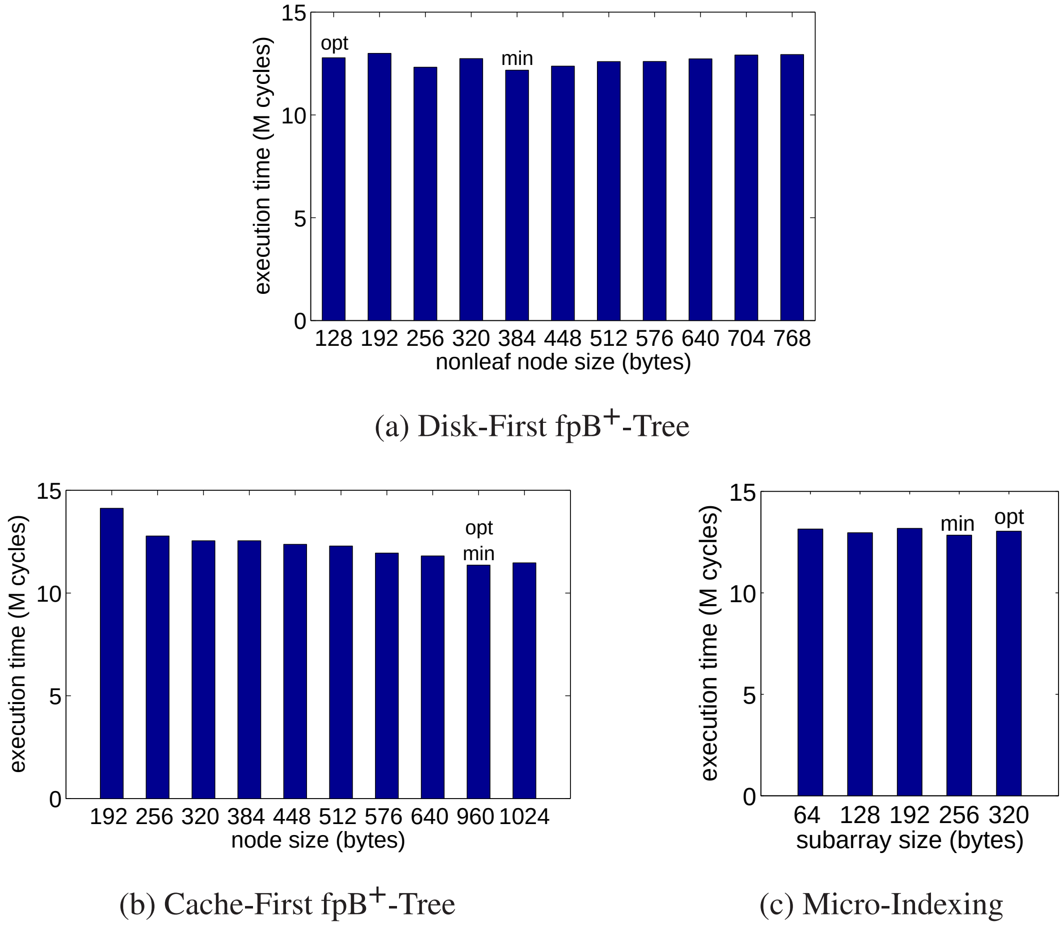


Figure 3.13: Optimal width selection when page size is 16 KB. ("**min**": the width achieving the minimal execution time; "**opt**" : the selected optimal width given the optimal criterion.)

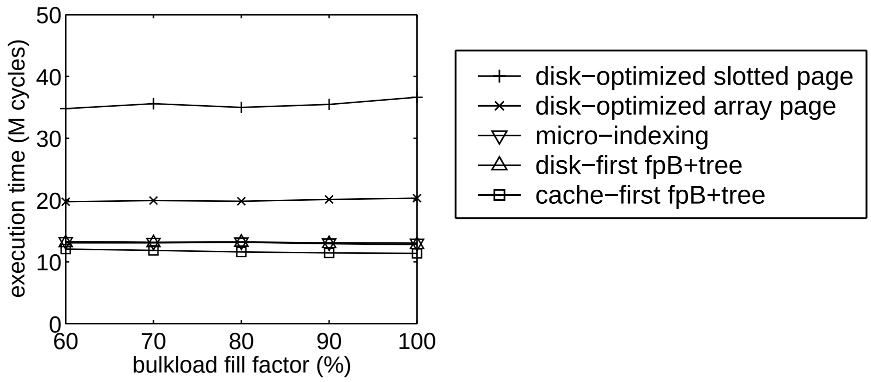


Figure 3.14: Search varying node occupancy (10 million keys, 16KB pages, 10,000 searches).

search cost of the selected optimal trees over the best search performance when the database page size is 4KB, 8KB, and 32KB.) Therefore, in the experiments that follow, we use the optimal node sizes given in Table 3.1.

Varying Node Occupancy. In Figure 3.14, we varied the 1e7-entry experiments in Figure 3.12(c) with bulkload fill factors ranging from 60% to 100%. Compared with disk-optimized B^+-Trees with slotted pages, fpB^+-Trees and micro-indexing achieve speedups between 2.62 and 3.23.

3.4.3 Insertion Cache Performance through Simulations

Figure 3.15 shows the insertion performance in three different settings. The experiments all measured the execution times for inserting 10,000 random keys after bulkloads, while varying the bulkload fill factor, the numbers of entries in leaf pages, and the page size. The fpB^+-Trees achieve up to a 3.83 speedup over disk-optimized B^+-Trees with slotted pages, and up to a 20-fold speedup over disk-optimized B^+-Trees with index entry arrays, while micro-indexing performs almost as poorly as disk-optimized B^+-Trees with index entry arrays.

Figure 3.15(a) compares insertion performance of trees from 60% to 100% full containing 10 million keys. Compared to the two schemes that store large contiguous arrays of index entries in pages (disk-optimized B^+-Trees with index entry arrays and micro-indexing), the other schemes avoid the large data movement cost for updates and perform dramatically better when trees are between 60% and 90% full. The fpB^+-Trees achieve 13 to 20-fold speedups over disk-optimized B^+-Trees with index entry arrays between 60% and 90%, while for 100% full trees, they are over 2.0 times better.

Interestingly, the curves have extremely different shapes: Those of disk-optimized B^+-Trees with index entry arrays and micro-indexing increase from 60% to 90% but drop at the 100% point, while the curves of fpB^+-Trees and disk-optimized B^+-Trees with slotted pages stay flat at first but jump dramatically at the 100% point. These effects can be explained by the combination of two factors: data movement and page splits. When trees are 60% to 90% full, insertions usually find empty slots and the major operation after searching where the key belongs is to move the key and pointer arrays in

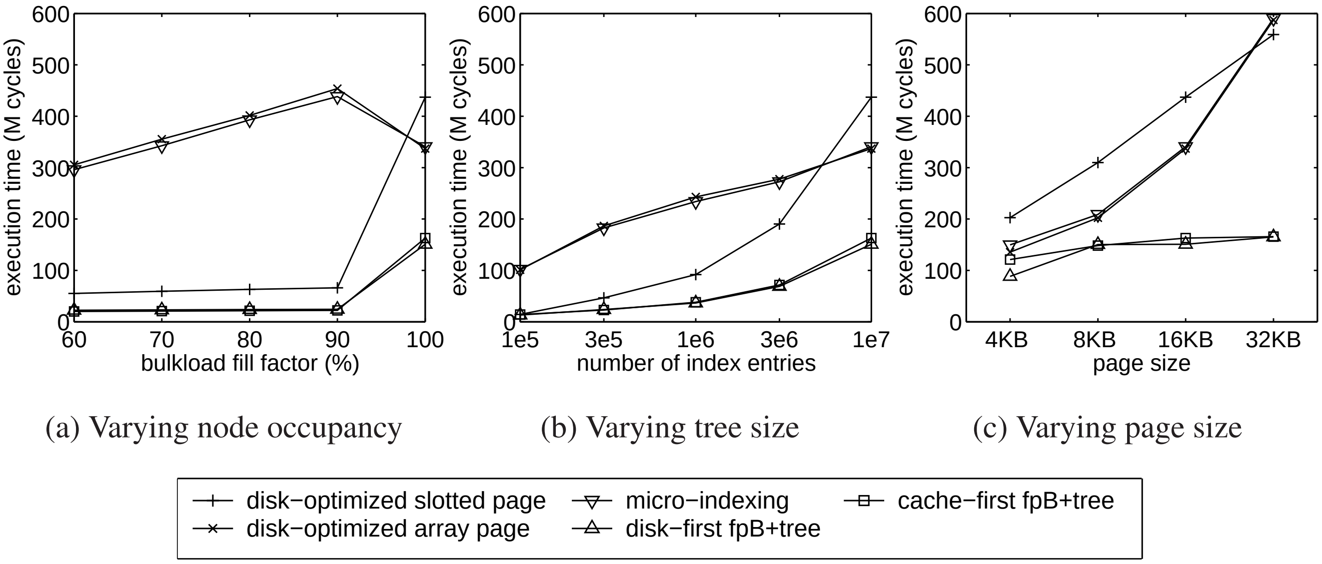


Figure 3.15: Insertion cache performance (default parameters: 10 million keys, 100% full, 16 KB pages, 10,000 insertions).

order to insert the new entry. When large contiguous arrays are stored in pages, this data movement is by far the dominant cost. As the occupied portions of the arrays grow from 60% to 90%, this cost increases, resulting in slower insertion times. In fpB^{+}-Trees, we reduced the data movement cost by using smaller cache-optimized nodes, resulting in 13 to 20-fold speedups. Data movement has become much less costly than search, leading to the flat curves up through 90% full. Similarly, the curve of slotted page organization is also flat until 90% full because it reduces data movement by using one level of indirection through the slot offset arrays. However, the use of slot offset arrays results in poor CPU cache performance for search. Compared to disk-optimized B^{+}-Trees with slotted pages, fpB^{+}-Trees achieve a speedup of over 2.46 because of faster searches.

When the trees are 100% full, insertions cause frequent page splits. In disk-optimized B^{+}-Trees with slotted pages and fpB^{+}-Trees, the cost of a page split is far more than the previous data movement cost, resulting in the large jump seen in the curves. In schemes that store large contiguous arrays in pages, however, the page split cost is comparable to copying half of a page, which is the average data movement cost for inserting into an almost full page. But later insertions may hit half empty pages (just split) and hence incur less data movement, resulting in faster insertion times at the 100% point. Moreover, disk-optimized B^+-Trees with slotted pages have to access each index entry through the slot offset array, while the other schemes all directly move a number of index entries at a time. This additional cost of indirection makes the performance of the slotted page organization even worse than the index entry array page organization for inserting into 100% full trees.

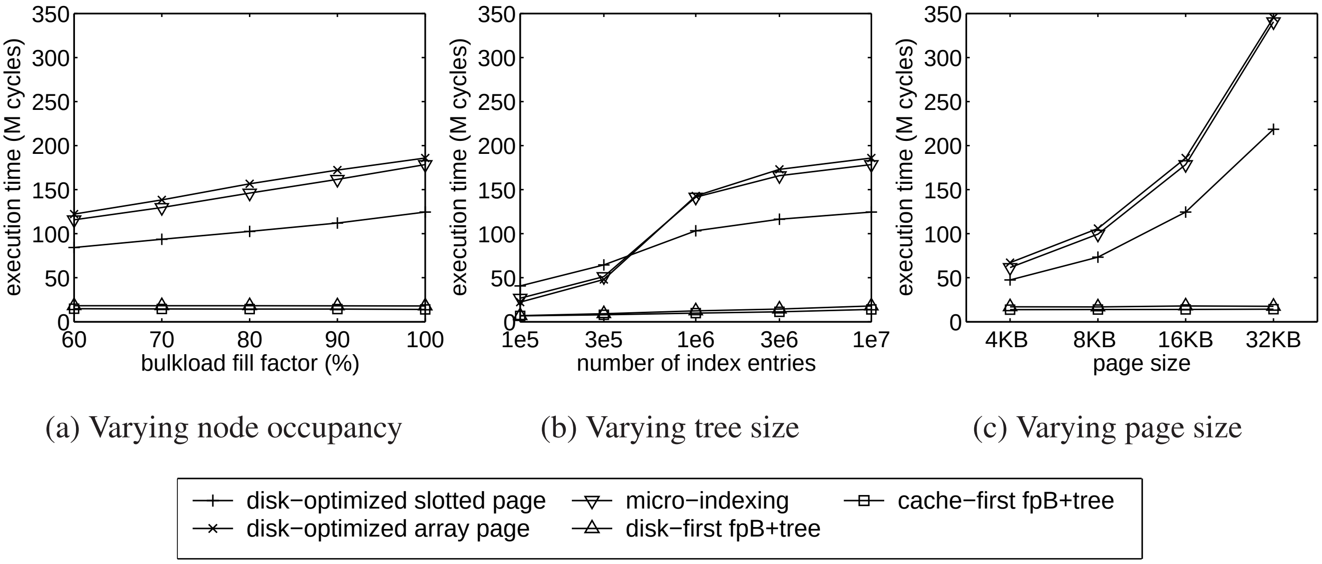
Figure 3.15(b) shows insertion performance on full trees of different sizes. Compared to disk-optimized B^+-Trees with slotted pages, fpB^+-Trees achieve speedups up to 2.90 when the number of entries in leaf pages is increased from 1e5 to 1e7. Compared to disk-optimized B^+-Trees with index entry arrays, fpB^+-Trees achieve speedups from 8.16 to 2.07. This decrease in speedup is caused by the increasing number of page splits (from 48 to 1631 leaf page splits for disk-optimized B^+-Trees with index entry arrays, and similar trends for other indices). As argued above, larger number of page splits have a much greater performance impact on fpB^+-Trees than on indices using large contiguous arrays, leading to the speedup decrease.

Figure 3.15(c) compares the insertion performance varying page sizes when trees are 100% full. As the page size grows, the execution times of disk-optimized B^+-Trees and micro-indexing degrade dramatically because of the combined effects of larger data movement and larger page split costs. In fpB^+-Trees, though page split costs also increase, search and data movement costs only change slightly, because with larger page sizes comes the advantages of larger optimal node widths. Therefore, the curves of fpB^+-Trees increase only slightly in Figure 3.15(c). In Figure 3.15(c), fpB^+-Trees achieve 1.12–3.83 speedups over the two disk-optimized B^+-Trees.

Comparing the two fpB^+-Trees, we see they have similar insertion performance. Sometimes cache-first fpB^+-Trees perform worse than disk-first fpB^+-Trees. This is primarily because of the more complicated node/page split operations in cache-first fpB^+-Trees, as discussed in Section 3.3.4.

3.4.4 Deletion Cache Performance through Simulations

Deletions are implemented as lazy deletions in all the indices. A search is followed by a data movement operation to remove the deleted entry, but we do not merge underflowed pages or nodes. Figure 3.16



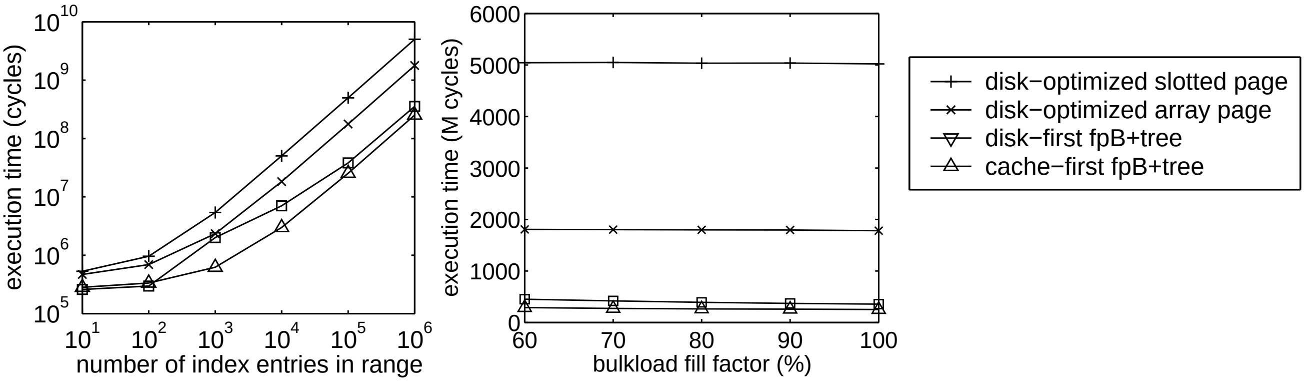
(a) Varying node occupancy (b) Varying tree size (c) Varying page size

Figure 3.16: Deletion cache performance (default parameters: 10 million keys, 100% full, 16 KB pages, 10,000 deletions).

evaluates deletion performance (for 10,000 random deletions) in three settings: (a) varying the bulkload fill factor when the page size is 16KB, (b) varying the number of index entries when trees are 100% full, and (c) varying the page sizes when the trees are 100% full. The dominant cost in disk-optimized B^+-Trees with index entry arrays and micro-indexing is the data movement cost, which increases as the bulkload factor increases, the page size grows, and the tree height or root page occupancy grows as the tree size increases. For disk-optimized B^+-Trees with slotted pages, slot offset arrays must be kept contiguous. The size of the slot offset arrays increases as the page size, leading to increasing data movement cost for deletions. However, the search and data movement costs of fpB^+-Trees only change slightly. So the fpB^+-Trees achieve 3.2–24.2 fold speedups over disk-optimized B^+-Trees with index entry arrays, and 2.8–15.3 fold speedups over disk-optimized B^+-Trees with slotted pages.

3.4.5 Range Scan Cache Performance through Simulations

Figures 3.17(a) and (b) compare the range scan cache performance of fpB^+-Trees and disk-optimized B^+-Trees in two settings: (a) varying the range size from 10 to 1 million index entries, and (b) varying



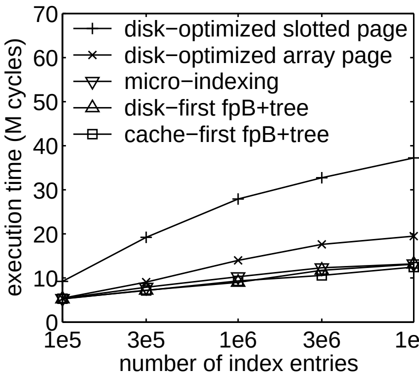
(a) Varying range size (b) Varying node occupancy

Figure 3.17: Range scan performance (default parameters: 10 million keys in trees, 100% full, 16 KB pages, scanning 1 million keys).

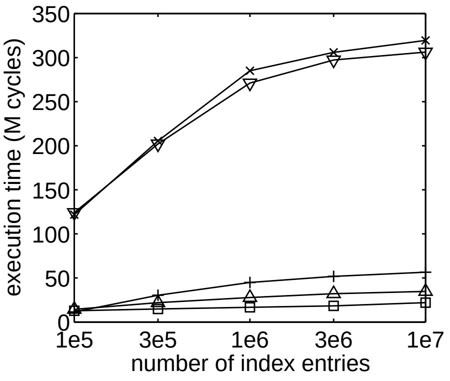
the bulkload fill factor from 60% to 100%. The trees are bulkloaded with 10 million keys. We generate 100 random start keys, for each computing an end key such that the range spans precisely the specified number of index entries, and then perform these 100 range scans one after another. From the figures, we see that the two disk-optimized B^+-Tree curves are very different: The slotted page organization is up to 2.8-fold slower than the array page organization. This is because the slotted page organization pays the indirection cost when retrieving index entries within a page during a range scan. Compared to the disk-optimized B^+-Trees with slotted pages, the disk-first and cache-first fpB^+-Trees achieve speedups between 1.9 and 20.0.

3.4.6 Mature Tree Cache Performance

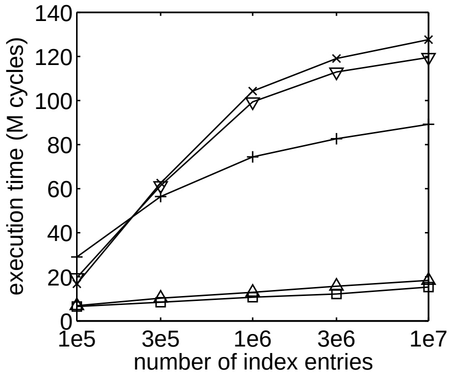
We performed a set of experiments with mature trees created by bulkloading 10% of index entries and then inserting the remaining 90%. We vary the number of index entries in the mature trees from 1e5 to 1e7 for the search, insertion, and deletion experiments, as shown in Figures 3.18(a)-(c). For range scans, we vary the range size from 10 index entries to 1 million index entries, as shown in Figure 3.18(d). From the figures, we find performance gains similar to those shown in our previous experiments with bulkloaded trees. Compared to disk-optimized B^+-Trees with slotted pages, fpB^+-Trees improve search



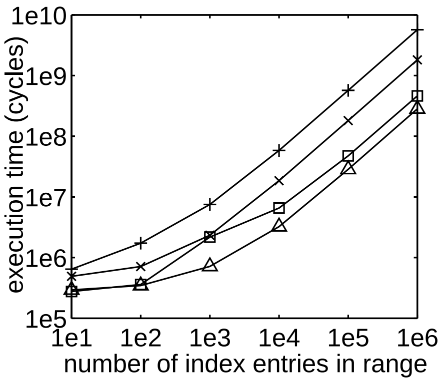
(a) Search



(b) Insertion



(c) Deletion



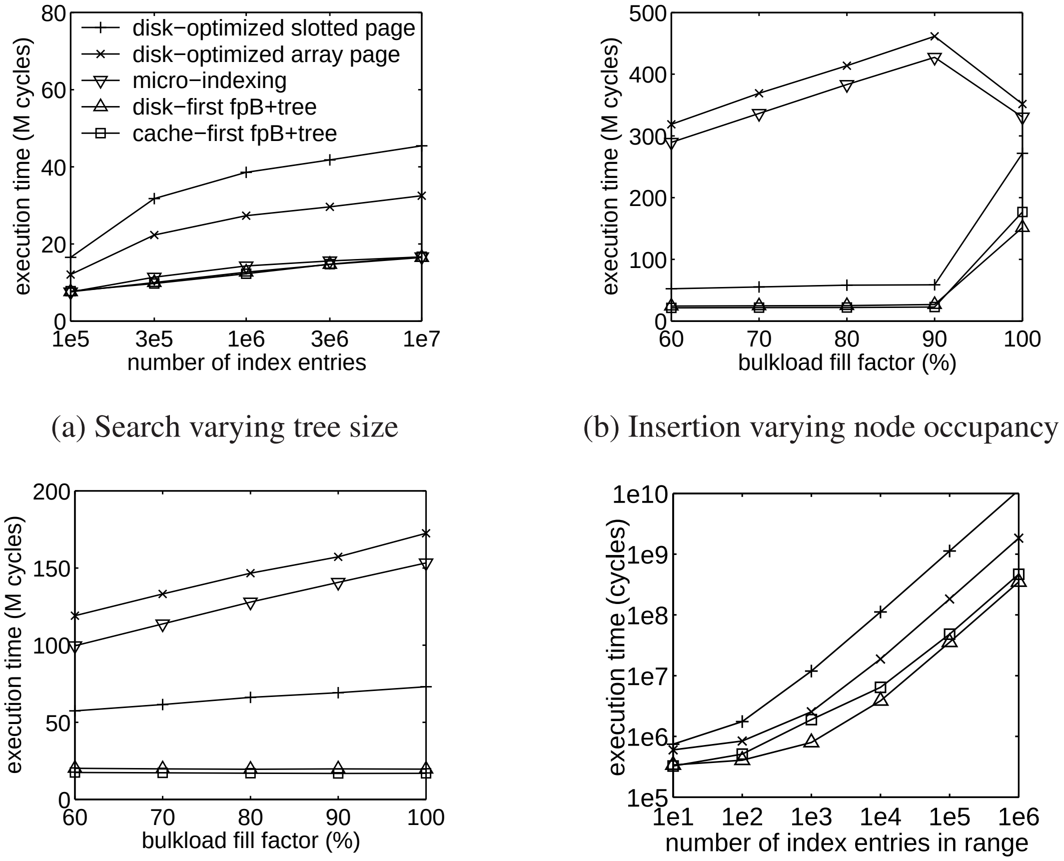
(d) Range scan

Figure 3.18: Mature tree cache performance.

performance by a factor of 1.7–3.1, insertion performance by a factor up to 2.83, deletion performance by a factor of 4.2–6.9, and range scan performance by a factor of 2.2–20.1.

3.4.7 Results with Larger Key Size

Finally, we present our experimental results with larger keys. Figures 3.19(a)-(d) show the index cache performance when 20-byte keys are used, and the experiments correspond to those (with 4-byte keys) shown previously in Figure 3.12(c), Figure 3.15(a), Figure 3.16(a), and Figure 3.17(a), respectively. Comparing the corresponding figures, we see that experiments with larger keys show similar perfor-



(a) Search varying tree size

(b) Insertion varying node occupancy

(c) Deletion varying node occupancy

(d) Range scan varyingrange size

Figure 3.19: Operations with 20B keys and 16KB pages. (Disk-First fpB$^+$-Tree: non-leaf node=64B, leaf node=384B; Cache-First fpB$^+$-Tree: node size=576B; Micro-Indexing: subarray size = 128B)

mance gains to our previous experiments. Compared to disk-optimized B$^+$-Trees with slotted pages, fpB$^+$-Trees improve search by a factor of 2.1–3.3, insertion by a factor of 1.5–2.6, deletion by a factor of 2.9–4.3, and range scan by a factor of 2.2–32.5.

3.5 Cache Performance on an Itanium 2 Machine

After analyzing the experimental results through simulations, in this section, we present our experimental results on the Itanium 2 machine to verify the findings in the previous section. We begin by describing our experimental setup. Then we present the results on search, insertion, deletion, range scan, and mature tree operations. Finally, we compare the simulation and Itanium 2 results.

3.5.1 Experimental Setup

We use the same Itanium 2 machine for real machine experiments as in Chapter 2. (Its configuration parameters are detailed previously in Section 2.4.1.) We study three index structures in the real machine experiments: (i) disk-optimized B^{+}-Trees with slotted pages, (ii) micro-indexing, and (iii) disk-first fpB^{+}-Trees. Note that in the simulation study, we use disk-optimized B^{+}-Trees with index entry arrays to better understand the performance of the other schemes. However, our major purpose in this section is to verify the performance benefits of our schemes. Therefore, we only compare with the most common implementation of disk-optimized B^{+}-Trees. Moreover, we also do not show the performance of cache-first fpB^{+}-Trees because their cache performance has been shown to be similar to that of disk-optimized B^{+}-Trees. (Cache-First fpB^{+}-Trees may incur large I/O overhead and therefore less attractive, as will be described in Section 3.6.)

In our experiments, we use eight-byte keys for easy comparison with our previous experiments with prefetching B^{+}-Trees. However, unlike experiments for main memory indices, we use four-byte `pageIDs` or `tupleIDs` instead of eight-byte pointers. The index structures discussed in this chapter do not directly contain memory addresses as pointers. Instead, they contain ID values determined by the database storage manager. Ideally, these physical IDs are kept unchanged when a database is ported across different platforms. Therefore, their sizes should not be affected by the memory address size of the Itanium 2 machine. The other aspects of the index structures follow the descriptions in the simulation experiments in Section 3.4.1.

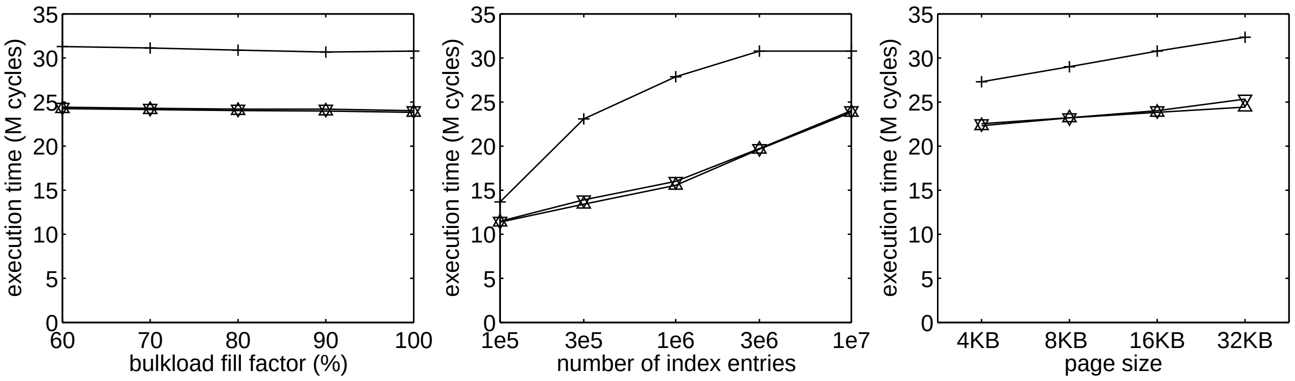
Table 3.2 shows the computed optimal node widths for disk-first fpB^{+}-Trees and micro-indexing using $T_1 = 189$ and $T_{next} = 24$ when key size is 8 bytes. The slowdowns relative to the optimal performance show the measurements in our experiments. We can see that like our simulation experiments, the optimal width selections all satisfy our optimization criterion $\mathcal{G}$, i.e. maximizing leaf page fan-out while achieving within 10% of the best search performance.

Table 3.2: Optimal width selections on the Itanium 2 machine (8 byte keys, $T_1 = 189$, $T_{\text{next}} = 24$).

	Disk-First fpB$^+$-Trees				Micro-Indexing		
page size	non-leaf node	leaf node	page fan-out	$\frac{\text{cost}}{\text{optimal}}$	subarray size	page fan-out	$\frac{\text{cost}}{\text{optimal}}$
4KB	128	384	310	1.00	128	320	1.00
8KB	128	640	636	1.01	384	672	1.01
16KB	256	640	1325	1.00	384	1344	1.01
32KB	384	896	2664	1.00	768	2704	1.00

Since data movement operations occur frequently in all the indices, we examined a few choices of implementing them on the Itanium 2 machine. There are three types of data movements in the indices: (i) moving data to a different page during page splits; (ii) moving data for a few bytes toward lower memory addresses; and (iii) moving data for a few bytes to-wards higher memory addresses. The latter two cases occur when removing an entry from or inserting an entry into an array. Since the in-page offsets (for both disk-optimized B$^+$-Trees with slotted pages and disk-first fpB$^+$-Trees) are two-bytes, we would like to support data movements with a granularity of at least 2 bytes (if not 1 byte). We decide to use the library function "memcpy" because it is even faster than copying with 2-byte integers. However, "memcpy" can only deal with the first two types of data movements. Although the third type can be handled by "memmove", its performance is significantly worse than "memcpy". Surprisingly, it is more efficient to copy the source data to a temporary buffer and then copy it to the destination using two "memcpy" calls. Therefore, we decide to use this simple implementation of two "memcpy" calls for the third type of data movements. However, this choice result in an additional "memcpy" call for inserting keys into disk-first fpB$^+$-Trees but deleting keys from disk-optimized B$^+$-Trees with slotted pages (the slots are in reverse key order). As a result, the insertion speedups of our schemes are smaller than those of deletion under similar settings, as will be described further in Section 3.5.4 and 3.5.6.

For each of the point in the figures that follow, we performed 30 runs and measured the user-mode execution times in cycles using the perfmon library. Note that the instruction overhead of buffer management is included in all the measurements. We compute and report the average of the 30 runs for each



(a) Varying node occupancy (b) Varying tree size (c) Varying page size

Figure 3.20: Search cache performance on the Itanium 2 machine (default parameters: 10 million keys, 100% full, 16 KB pages, 10,000 searches).

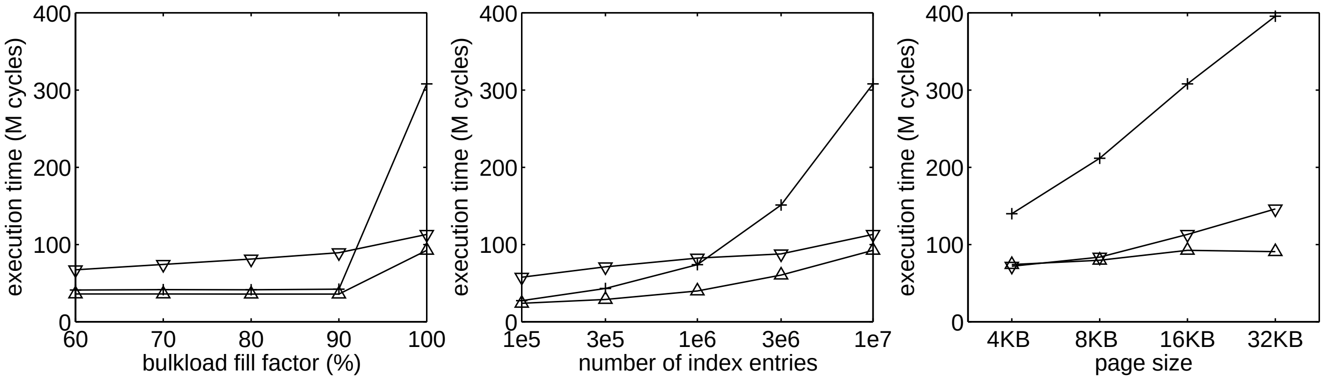
experiment. The standard deviations of the experiments are all within 5% of the averages, and over 90% of the experiments have their standard deviations within 1% of the averages.

3.5.2 Search Performance on Itanium 2

Figures 3.20(a)–(c) show the cache performance of 10,000 random searches varying the node occupancy from 60% to 100% for trees with 10 million keys, varying the tree size from 1e5 to 1e7 with trees that are 100% full, and varying the page size from 4KB to 32KB for trees with 10 million keys. The default parameters describe trees that are common across the three sets of experiments. From the figures, we see that the cache-sensitive schemes, disk-first fpB$^+$-Trees and micro-indexing, perform significantly better than disk-optimized B$^+$-Trees. Compared to disk-optimized B$^+$-Trees with slotted nodes, our disk-first fpB$^+$-Trees achieve speedups between 1.20 and 1.79 at all points.

3.5.3 Insertion Performance on Itanium 2

Figures 3.21(a)–(c) show the performance of inserting 10,000 random keys into the indices while varying node occupancy, tree size, and page size. The experiments correspond to those in the simulation experi-



(a) Varying node occupancy (b) Varying tree size (c) Varying page size

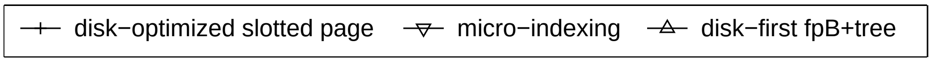
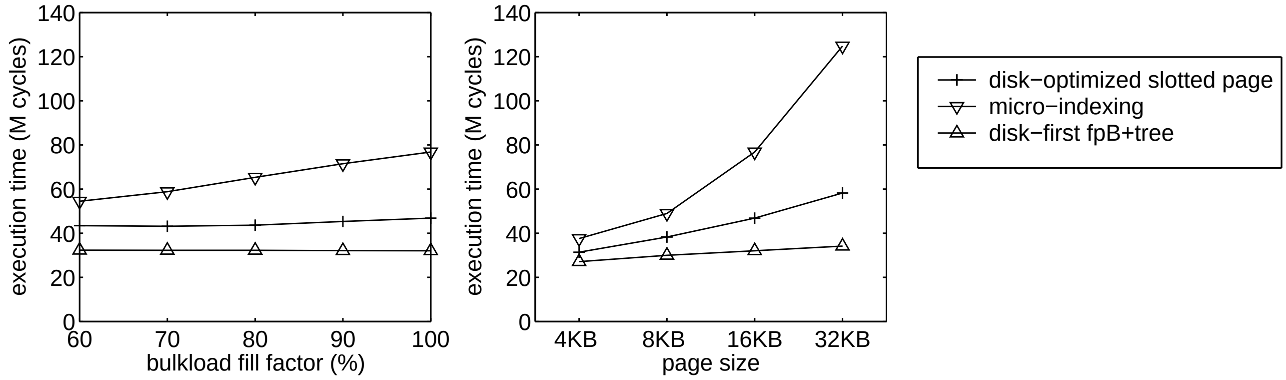


Figure 3.21: Insertion cache performance on the Itanium 2 machine (default parameters: 10 million keys, 100% full, 16 KB pages, 10,000 insertions).



(a) Varying node occupancy (b) Varying page size

Figure 3.22: Deletion cache performance on the Itanium 2 machine (default parameters: 10 million keys, 100% full, 16 KB pages, 10,000 deletions).

ments as shown in Figures 3.15(a)–(c). Comparing the real machine results with the simulation results, we see that the real machine curves follow similar trends except that the gap between micro-indexing and disk-first fpB$^+$-Trees is smaller. This means that the data movement cost for a large contiguous array is less significant on the Itanium 2 machine. We used "memcpy" for moving large chunks of data in our implementations. A possible explanation is that because the data movement has sequential access patterns, either hardware or software prefetches (by the "memcpy" implementation) can be easily employed for improving its performance.

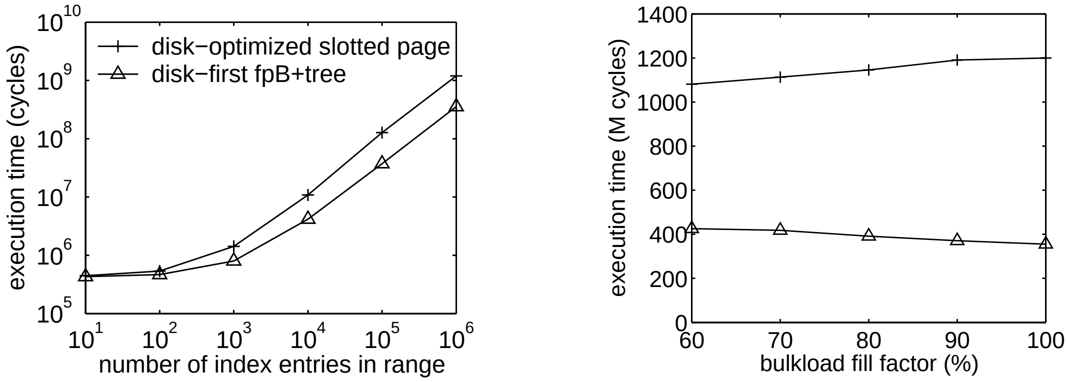
When the trees are between 60% and 90% full, the performance of disk-optimized B$^+$-Trees and disk-first fpB$^+$-Trees are similar. However, when the trees are 100% full, disk-optimized B$^+$-Trees with slotted pages suffer from the overhead of indirection through the slot offset arrays during page splits. Compared to disk-optimized B$^+$-Trees with slotted pages, our disk-first fpB$^+$-Trees achieve 1.14–1.17X speedups when trees are at most 90% full, and up to 4.36X speedups when trees are full.

3.5.4 Deletion Performance on Itanium 2

Figures 3.22(a) and (b) show the execution times of deleting 10,000 random keys from the indices varying the bulkload fill factor from 60% to 100%, and varying the page size from 4KB to 32KB. The trees are built in the same way as for Figures 3.16(a) and (c) in the simulation study. As we can see from the figures, the curves all follow trends similar to those in the simulation study. Compared to disk-optimized B$^+$-Trees with slotted pages, our disk-first fpB$^+$-Trees achieve 1.16–1.70X speedups. Moreover, compared to the insertion performance with trees that are at most 90% full, we can see that the speedups for deletion are larger than those for insertions. This is mainly because of the implementation of data movement operations, as described previously in Section 3.5.1.

3.5.5 Range Scan Performance on Itanium 2

Figures 3.23(a) and (b) show the real machine range scan performance varying the range size and the node occupancy. The figures report the same experiments as shown in Figures 3.17(a) and (b) in the



(a) Varying range size (b) Varying node occupancy

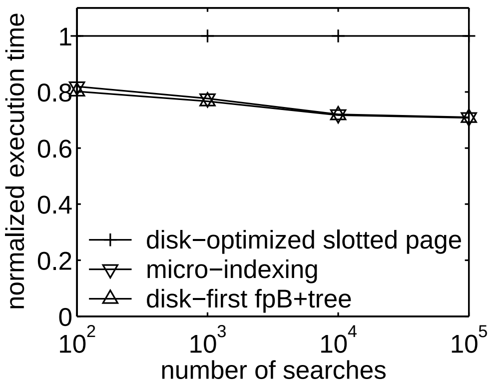
Figure 3.23: Range scan cache performance on the Itanium 2 machine (default parameters: 10 million keys in trees, 100% full, 16 KB pages, scanning 1 million keys).

simulation study. Compared to disk-optimized B^+-Trees with slotted pages, our disk-first fpB^+-Trees achieve 1.03–1.25X speedups for ranges up to 100 entries, and 1.78–3.38X speedups for larger ranges containing 1000 or more index entries.

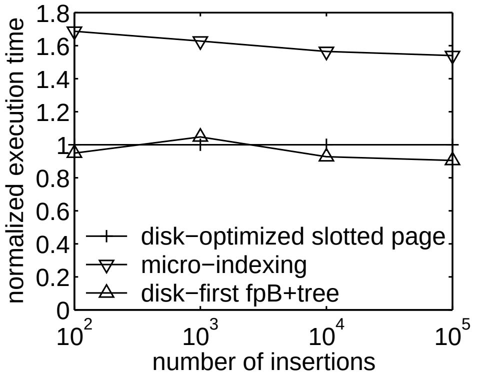
3.5.6 Operations on Mature Trees

Next, we present the experimental results for operations on mature trees. Figures 3.24(a)–(c) show the execution times normalized to those of disk-optimized B^+-Trees for search, insertion, and deletion operations while varying the number of operations from 1e2 to 1e5. The indices are created by bulkloading 1 million index entries and then inserting 9 million index entries. Compared to disk-optimized B^+-Trees with slotted pages, our disk-first fpB^+-Trees improve search by a factor of 1.25–1.41, achieve similar insertion performance, and improve deletion by a factor of 1.28–1.44. (We see the impact of the implementation of data movement operations again. Given an implementation that has equally good performance for moving data toward both directions, the insertion performance would be better.)

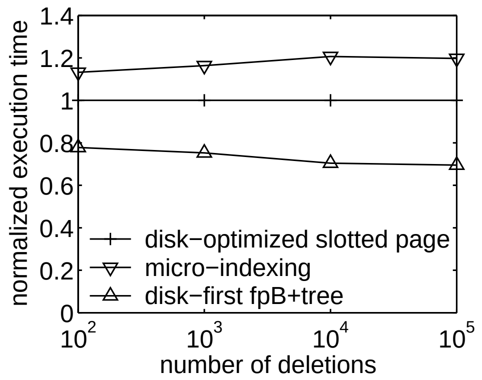
Figure 3.24(d) shows the execution times of scanning 100 random ranges while varying the range size from 10 index entries to 1 million index entries. The experiments correspond to those shown in Figure 3.23(a). Compared to disk-optimized B^+-Trees with slotted pages, our disk-first fpB^+-Trees



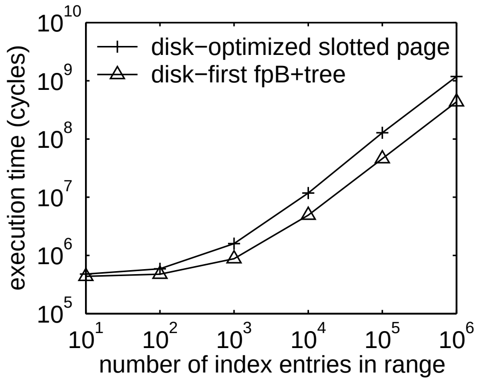
(a) Search



(b) Insertion



(c) Deletion



(d) Range scan

Figure 3.24: Operations on mature trees on the Itanium 2 machine (default parameters: 10 million keys in trees and 16 KB pages).

achieve 1.09–1.24X speedups for smaller ranges that contain up to 100 index entries, and 1.82–2.71X speedups for larger ranges that contain 1000 or more index entries.

3.5.7 Comparing Simulation and Itanium 2 Results

Finally, we compare the simulation and Itanium 2 results. Comparing Figure 3.12 and Figure 3.14 with Figure 3.20 for search performance, comparing Figure 3.15 with Figure 3.21 for insertion performance, comparing Figure 3.16 with Figure 3.22 for deletion performance, and comparing Figure 3.17 with Figure 3.23 for range scan performance, we see that the Itanium 2 curves all show similar trends to the simulation curves, which verifies the simulation results. (The speedups on the Itanium 2 machine are comparable or smaller than the speedups in the simulation study. This is mainly because of the larger memory latency in the simulator setting, which leads to larger benefits for cache optimizations.) Therefore, we conclude that our experimental results on both the simulation platform and the Itanium 2 machine demonstrate significant benefits of the fpB$^+$-Trees.

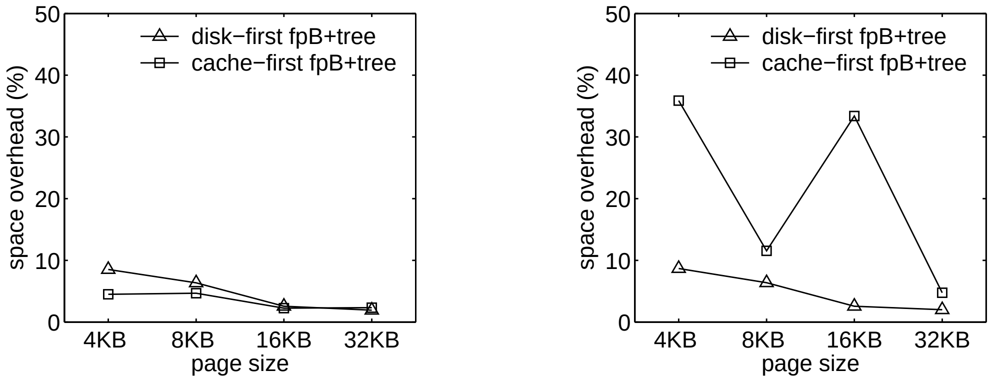
3.6 I/O Performance and Space Overhead

We evaluate the I/O performance of the fpB$^+$-Trees through experiments on real machines. To study the I/O performance of searches, we executed random searches, and then counted the number of I/O accesses (i.e., the number of buffer pool misses). For searches, the I/O time is dominated by the number of I/Os, because there is little overlap in accessing the pages in a search. To study the I/O performance of range scans, we executed random range scans on the Itanium 2 machine using up to 8 disks. Furthermore, we evaluate the I/O performance of range scans in a commercial DBMS: We implemented our jump-pointer array scheme for disk-optimized B$^+$-Trees within IBM DB2, and executed range scan queries on DB2.

3.6.1 Space Overhead

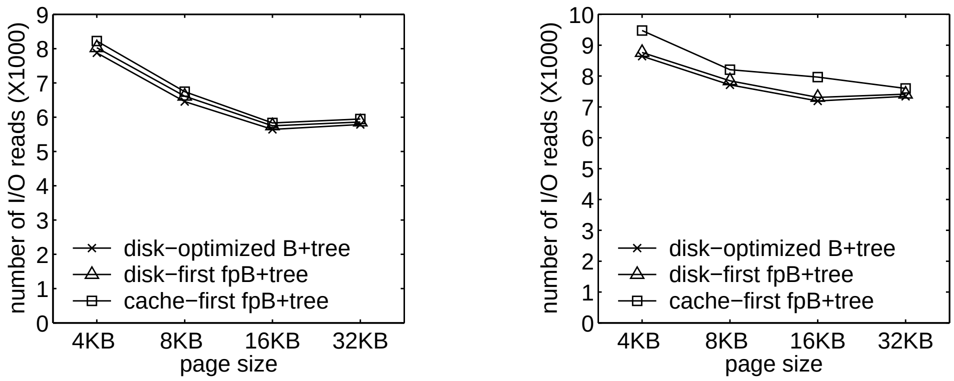
Figure 3.25 shows the space overhead[6] of the fpB$^+$-Trees compared to disk-optimized B$^+$-Trees for a range of page sizes, depicting two (extreme) scenarios: (i) immediately after bulkloading the trees 100% full, and (ii) after inserting 9 million keys into trees bulkloaded with 1 million keys. We see that in each of these scenarios, disk-first fpB$^+$-Trees incur less than a 9% overhead. In cache-first fpB$^+$-Trees, the space overhead is less than 5% under scenario (a), even better than disk-first fpB$^+$-Trees. This is because the leaf pages in cache-first fpB$^+$-Trees only contain in-page leaf nodes, while disk-first fpB$^+$-Trees build in-page trees (containing non-leaf and leaf nodes) in leaf pages. However, for the mature tree scenarios, the space overheads of the cache-first fpB$^+$-Tree can grow to 36%, because of the difficulties in maintaining effective placement of nodes within pages over many insertions.

[6] Space Overhead $= \dfrac{\text{number of pages in the index}}{\text{number of pages in a disk-optimized B}^+\text{-Tree}} - 1$



(a) After bulkloading trees 100% full (b) Mature trees

Figure 3.25: Space overhead.



(a) Search after bulkload (b) Search in mature trees

Figure 3.26: Number of disk accesses for searches.

Figure 3.25 also shows that as the page size grows, the space overhead of disk-first fpB^+-Trees decreases because larger pages allow more freedom when optimizing in-page node widths.

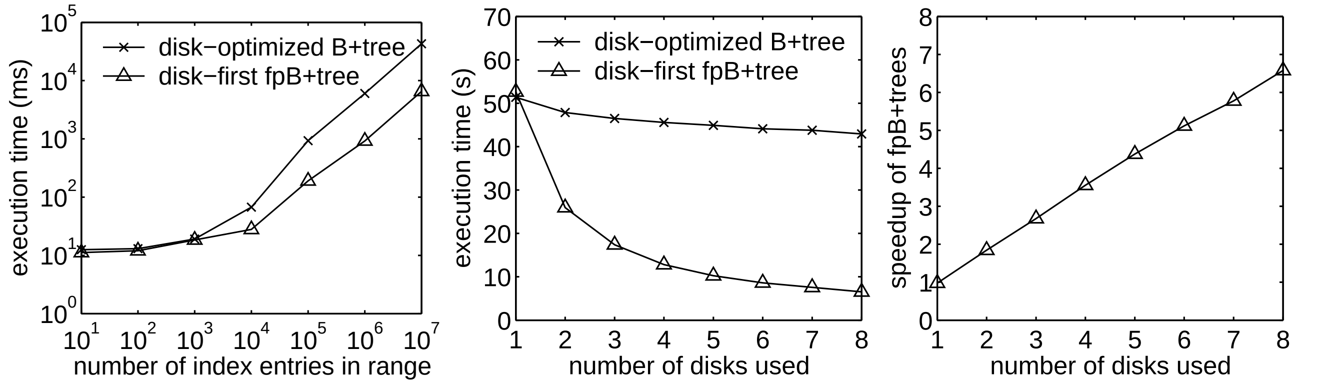
3.6.2 Search Disk Performance

Figure 3.26 shows the search I/O performance of fpB^+-Trees. The figure reports the number of I/O page reads that miss the buffer pool when searching 10,000 random keys in trees containing 10 million keys. The buffer pool was cleared before every experiment. We see that for all page sizes, disk-first fpB^+-Trees perform close to that of disk-optimized B^+-Trees, accessing less than 3% more pages. However, cache-first fpB^+-Trees may access up to 11% more pages. After looking into the experiments, we determined that the extra cost is incurred mainly when accessing leaf parent nodes in overflow pages. For example in the 4KB case in Figure 3.26(a), the fan-out of a non-leaf node is 57 and a page can contain part of a two-level tree. But only 6 out of the 57 children can reside on the same page as a node itself. Therefore even if all the parents of the leaf parent nodes are top-level nodes, 51 out of every 57 leaf parent nodes will still be placed in overflow pages, leading to many more page reads than disk-optimized B^+-Trees. However, as page sizes grow, this problem is alleviated and the performance of cache-first fpB^+-Trees gets better, as can be seen for the 32KB points.

3.6.3 Range Scan Disk Performance

Unlike our search experiments, which counted the number of I/O accesses, our range scan I/O performance experiments measure running times on the Itanium 2 machine (described earlier in Section 2.4.1) using up to 8 disks. Each of the eight disks is a 15,000 rpm Seagate Cheetah 15K ST336754LW with an average seek time of 3.6 ms, a track-to-track seek time of 0.2 ms, and an average rotational latency of 2 ms. These values give us a rough idea of the cost of a random disk access, the dominating operation of range scans in *non-clustered* indices (which are the focus of this study).

We implemented a buffer pool manager using the POSIX thread (pthread) library. We imitate raw disk partitions by allocating a large file on each disk used in an experiment and managing the mapping from `pageIDs` to file offsets ourselves. The buffer manager has a dedicated worker thread for each of the disks, which obtains I/O requests from a per-disk request queue and performs I/O operations on behalf of the main thread. To perform an I/O operation, the main thread computes the target disk based on the `pageID`, and puts an I/O request into the corresponding request queue. For synchronous I/O operations (e.g., reading an index page), the main thread blocks until it receives a completion notification from the appropriate worker thread. For prefetch requests, the main thread continues execution immediately after enqueueing the request. The worker thread, however, performs a synchronous I/O read system call and blocks on behalf of the main thread in the background. Later when the main thread attempts to access



(a) Execution time vs. range size (when 8 disks are used)

(b) Time vs. number of disks (10 million entries in range)

(c) Speedup vs. number of disks (10 million entries in range)

Figure 3.27: Range scan disk performance.

the prefetched page, it checks whether the valid flag of the destination buffer has been set by the worker thread. If not, then the I/O prefetch has not yet completed, and therefore the main thread will block until it receives the I/O completion notification from the worker thread.

We performed the following three operations to ensure that both the operating system file cache and the caches on disk are cold before every run in our experiments: (i) we restricted the Linux operating system to use only the lower 1GB of main memory by setting a boot flag; (ii) we allocated a 1GB memory buffer that was written and then read; and (iii) we read separate 128MB files from each of the eight disks.

We measure elapsed real times via `gettimeofday`. Each reported data point is the average of 30 runs; the standard deviation is less than 3% of the average in all cases. Moreover, for smaller ranges containing 10,000 or fewer entries, we perform 100 disjoint range scans in every run in order to reduce measurement variances. The reported points are computed by dividing the total execution times by 100.

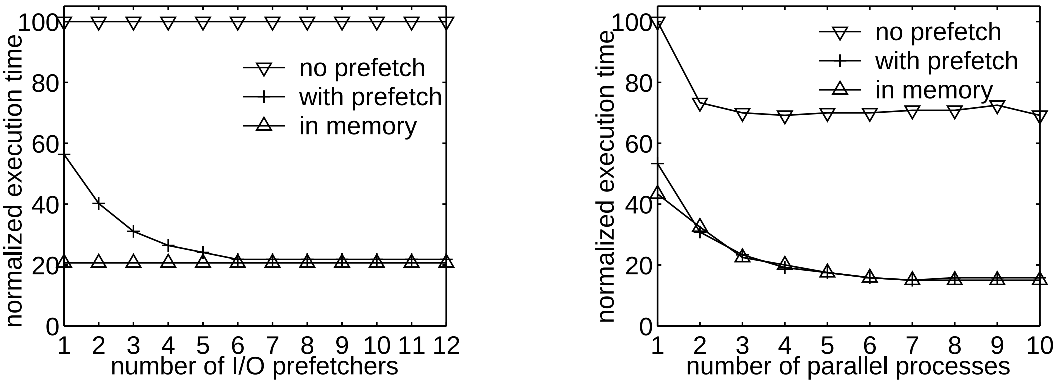
To set up the experiments in this subsection, we bulkloaded the trees 100% full with 45 million (8-byte) keys and then inserted 5 million keys to make the trees mature. Index pages are 16KB large. We striped the index pages of a tree across 1-8 disks depending on the experiment.

Figure 3.27 compares the range scan I/O performance of disk-first fpB$^+$-Trees and disk-optimized

B^+-Trees. Figure 3.27(a) shows the execution time in milliseconds for range scans using 8 disks, where the starting keys are selected at random and the size of the resulting range varies from 10 index entries to 10 million index entries. For small ranges (10-1000 entries), the execution times for the two trees are indistinguishable because the ranges often fit within a single disk page. For larger ranges (10,000 entries and up), which spans multiple pages, the disk-first fpB^+-Tree with jump-pointer array I/O prefetching provides a significant improvement over the disk-optimized B^+-Tree. Even for the 10,000-entry case, which scans only about 10 pages, the fpB^+-Tree achieves a speedup of 2.41 over the disk-optimized B^+-Tree. Better still, for large scans of 1 million and 10 million entries, the disk-first fpB^+-Tree is *6.52-6.58 times faster* than the disk-optimized B^+-Tree.

In our implementation of jump pointer array I/O prefetching, we avoid overshooting the end of a range by searching for the end key and recording the end leaf `pageID`. From the small range results in Figure 3.27(a), we see that this technique is quite effective. Note that this technique potentially incurs an additional I/O for loading the end leaf page. This is not a problem for two reasons: (i) for large ranges, this additional I/O overhead is negligible; (ii) for small ranges, the end leaf page tends to still be in the buffer pool so that the range scan operation does not load the end leaf page again when accessing it.

Figure 3.27(b) shows the execution time in seconds for scanning 10 million entries, varying the number of disks. Figure 3.27(c) shows the speedups corresponding to the experiments in Figure 3.27(b). Because of the data dependence in following the sibling links, disk-optimized B^+-Trees read only one leaf page at a time, and the I/Os for leaf pages are not overlapped. In contrast, jump pointer array prefetching enables the disk-first fpB^+-Trees to prefetch leaf pages sufficiently early. This ability leads to significant benefits: a 6.58-fold speedup with 8 disks. Note that the speedup almost doubles when doubling the number of disks. Since the I/O bandwidth used in the 8-disk experiments is only 33.3 MB/s out of the 320MB/s maximum bandwidth supported by a single SCSI controller, the speedup is likely to increase if more disks are added to the system.



(a) Varying number of prefetchers (SMP degree = 9)

(b) Varying SMP degree (Number of prefetchers = 8)

Figure 3.28: Impact of jump-pointer array prefetching on the range scan performance of DB2.

3.6.4 Range Scan Disk Performance on a Commercial DBMS

Finally, we evaluate the impact of jump-pointer array prefetching for range scans on a commercial DBMS. We implemented our jump-pointer array scheme for disk-optimized B+-Trees within IBM's DB2 Universal Database[7] . Because DB2's index structures support reverse scans and SMP scan parallelism, we added links in both directions, and at all levels of the tree. These links are adjusted at every non-leaf page split and page merge.

We performed experiments on an IBM 7015-R30 machine (from the RS/6000 line) with 8 processors, 80 SSA disks, and 2GB of memory, running the AIX operating system. We populated a 12.8 GB table across 80 raw partitions (i.e., 160 MB per partition) using 10 concurrent processes to insert a total of roughly 50 million rows of random data of the form `(int,int,char(20),int,char(512))`. An index was created using the three integer columns; its initial size was less than 1 GB, but it grows through page splits. We used the query `SELECT COUNT(*) FROM DATA`, which is answered using the index. Figure 3.28 shows the results of these experiments.

[7]Notices, Trademarks, Service Marks and Disclaimers: The information contained in this publication does not include any product warranties, and any statements provided in this document should not be interpreted as such. The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries: IBM, DB2, DB2 Universal Database. Other company, product or service names may be the trademarks or service marks of others.

As we see in Figure 3.28, our results on an industrial-strength DBMS are surprisingly good (2.5-5.0 speedups). The top curves in both figures are for the plain range scan implementation without jump-pointer array prefetching. The bottom curves show the situation when the leaf pages to be scanned are already in memory; this provides a limit to the possible performance improvements. Figure 3.28(a) shows that the performance of jump-pointer array prefetching increases with the number of I/O prefetchers, until the maximum performance is nearly reached. Figure 3.28(b) shows that increasing the degree of parallelism increases the query performance, which again tracks the maximum performance curve.

3.7 Related Work

A number of recent studies have demonstrated the importance of optimizing the cache performance of a DBMS [1, 2, 5]. B^+-Trees have been discussed in this regard, including several recent survey papers [31, 62]. This thesis, however, is the first to propose a B^+-Tree index structure that effectively optimizes both CPU cache and disk performance on modern processors, for each of the basic B^+-Tree operations: searches, range scans, insertions, and deletions.

Lomet [61] presented techniques for selecting the optimal page size for B^+-Trees when considering buffer pool performance for disk-resident data. To achieve both good disk and good cache performance, we propose to fit a cache-optimized tree into every disk-optimal B^+-Tree page. However, this usually results in an overflow or a large underflow of a page, as described in Section 3.3. We presented a disk-first solution and a cache-first solution to solve this node size mismatch problem in this chapter.

Lomet's recent survey of B^+-Tree techniques [62] mentioned the idea of intra-node micro-indexing: i.e., placing a small array in a few cache lines of the page that indices the remaining keys in the page. While it appears that this idea had not been pursued in any detail before, we compare its performance against fpB^+-Trees later in our experimental results. We observe that while micro-indexing achieves good search performance (often comparable to fpB^+-Trees), it suffers from poor update performance. As part of future directions, Lomet [62] has independently advocated breaking up B^+-Tree disk pages into cache-friendly units, pointing out the challenges of finding an organization that strikes a good balance between search and insertion performance, storage utilization, and simplicity. We believe that fpB$^+$-Trees achieve this balance.

Graefe and Larson [31] presented a survey of techniques for improving the CPU cache performance of B$^+$-Tree indices. They discussed a number of techniques, such as order-preserving compressions, and prefix and suffix truncation that stores the common prefix and suffix of all the keys in a node only once,that are complementary to our study, and could be incorporated into fpB$^+$-Trees. Bender *et al.* [9] present a recursive B$^+$-Tree structure that is *asymptotically* optimal, regardless of the cache line sizes and disk page sizes, but assuming no prefetching.

3.8 Discussion

Variable Length Keys. We have focused on improving performance for B$^+$-Trees with fixed length keys. For variable length keys, the common index structure employs the slotted page organization. However, as shown in our experiments, disk-optimized B$^+$-Trees with slotted pages have poor search cache performance and range scan cache performance. The structure shown in Figure 3.29 has been suggested to have better cache performance in [31, 62]. The main idea is to extract fixed length prefixes of the variable length keys to form a prefix array along with byte offsets pointing to the rest of index entries, which includes the rest of the keys and `pageIDs` or `tupleIDs`. If the prefixes distinguish keys well (e.g., after applying common prefix extraction), key comparison will mostly use the prefixes without accessing the rest of the keys. And we essentially get back to the contiguous array structure. Better performance results from the savings in key comparison and the spatial locality of the key prefix array.

We can employ fpB$^+$-Trees to improve the performance of B$^+$-Trees with variable length keys. Since prefetching allows cache optimized nodes to be much larger than a cache line, we will be able to put large variable length index entries in cache optimized nodes and then embed the nodes into disk-optimized B$^+$-Trees in the disk-first way, as shown conceptually in Figure 3.30. The actual fan-out of a node *varies* due to the variable sizes of its index entries. Alternatively, we can replace the key prefix array in Figure 3.29 with an in-page cache-optimized tree to further improve search and update performance.

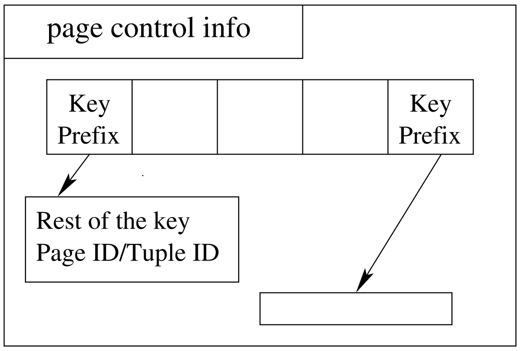


Figure 3.29: Extracting key prefixes into an offset array.

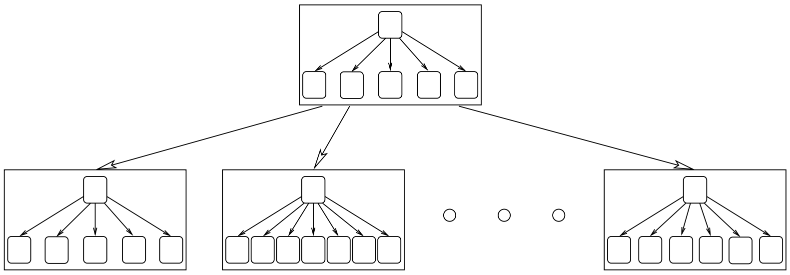


Figure 3.30: fpB$^+$-Trees for variable length keys.

Prefetching for Range Scans. For I/O range scan prefetching, we build jump-pointer arrays. An alternative approach traverses the non-leaf pages of a tree, retrieving leaf `pageIDs` for prefetching. The path from root page to the current leaf parent page can be recorded temporarily and with the help of the child pointers in non-leaf pages, one can sweep the path across the tree from the beginning to the end of the range. Although this approach saves the effort of building additional data structures, it becomes complicated when there are more than three levels in trees.

Concurrency Control. Our disk-first fpB$^+$-Trees can be viewed as a page organization scheme. All the modifications are within a page. Therefore, the original concurrency control scheme for disk-optimized B$^+$-Trees can be still employed for disk-first fpB$^+$-Trees. Moreover, the sibling links for implementing jump-pointer arrays have already been advocated for concurrency control purposes [56].

3.9 Chapter Summary

Previous studies on improving index performance have focused either on optimizing the cache performance of memory-resident databases, or else optimizing the I/O performance of disk-resident databases. What has been lacking prior to this study is an index structure that achieves good performance for both of these important levels of the memory hierarchy. Our experimental results demonstrate that *Fractal Prefetching B^+-Trees* are such a solution:

- Compared with disk-optimized B^+-Trees with slotted pages, Fractal Prefetching B^+-Trees achieve 1.20-4.49X speedups for search cache performance, up to a factor of 20-fold speedup for range scan cache performance, and up to a 15-fold speedup for update cache performance.
- Compared with disk-optimized B^+-Trees with slotted pages, disk-first fpB^+-Trees have low space overhead (less than 9%) and low I/O overhead (less than 3%) for searches. Because cache-first fpB^+-Trees may have some large I/O overhead, we recommend in general the *disk-first* approach.
- Jump pointer array I/O prefetching achieves up to a five-fold improvement in the I/O performance of range scans on a commercial DBMS (DB2).

In summary, by effectively addressing the complete memory hierarchy, fpB^+-Trees are a practical solution for improving DBMS performance.

Chapter 4

Improving Hash Join Performance through Prefetching

4.1 Introduction

Hash join [54, 59, 74, 89, 101] has been studied extensively over the past two decades, and it is commonly used in today's commercial database systems to implement equijoins efficiently. In its simplest form, the algorithm first builds a hash table on the smaller (*build*) relation, and then probes this hash table using tuples of the larger (*probe*) relation to find matches. However, the random access patterns inherent in the hashing operation have little spatial or temporal locality. When the main memory available to a hash join is too small to hold the build relation and the hash table, the simplistic algorithm suffers excessive random disk accesses. To avoid this problem, the *GRACE* hash join algorithm [54] begins by partitioning the two joining relations such that each build partition and its hash table can fit within memory; pairs of build and probe partitions are then joined separately as in the simple algorithm. This *I/O partitioning* technique limits the random accesses to objects that fit within main memory and results in nice predictable I/Os for every source relation and intermediate partition. Because it is straightforward to predict the next disk address for individual relation and partition, I/O prefetching can be exploited effectively to hide I/O latencies. As a result, the I/O costs no longer dominate. For example, our experiments on an Itanium 2 machine show that a hash join of two several GB relations is CPU-bound with

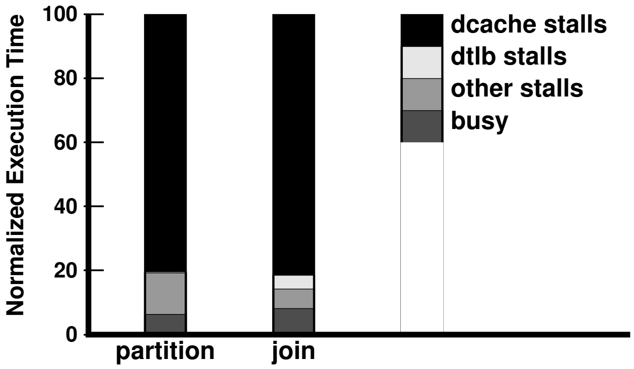


Figure 4.1: User-mode execution time breakdown for hash join.

five or seven disks depending on whether output tuples are consumed in memory or written to disk, and it becomes increasingly CPU-bound with each additional disk, as will be shown in Section 4.7.

4.1.1 Hash Joins Suffer from CPU Cache Stalls

So where *do* hash joins spend most of their time? Previous studies have demonstrated that hash joins can suffer from excessive CPU cache stalls [14, 66, 90]. The lack of spatial or temporal locality means the GRACE hash join algorithm cannot take advantage of the multiple levels of CPU cache in modern processors, and hence it repeatedly suffers the full latency to main memory during building and probing. Figure 4.1 provides a breakdown of the simulated user-level performance on our simulation platform. The memory hierarchy of the simulator is modeled after the Itanium 2 machine [44], as described in Section 2.4.2. The "partition" experiment divides a 200MB build relation and a 400MB probe relation into 800 partitions, while the "join" experiment joins a 50MB build partition with a 100MB probe partition. Each bar is broken down into four categories: busy time, data cache stalls, data TLB miss stalls, and other stalls. As we see in Figure 4.1, both the *partition* and *join* phases spend a significant fraction of their time—80% and 81%, respectively—stalled on data cache misses!

Given the success of I/O partitioning in avoiding random *disk* accesses, the obvious question is whether a similar technique can be used to avoid random *memory* accesses. *Cache partitioning*, in which the joining relations are partitioned such that each build partition and its hash table can fit within the (largest) CPU cache, has been shown to be effective in improving performance in memory-resident and main-memory databases [14, 66, 90]. However, cache partitioning suffers from two important practical limitations. First, for traditional disk-oriented databases, generating cache-sized partitions while scanning from disk requires a large number of concurrently active partitions. Experiences with the IBM DB2 have shown that storage managers can handle only hundreds of active partitions per join [59]. Given a 2 MB CPU cache and (optimistically) 1000 partitions, the *maximum relation size that can be handled is only 2 GB*. Beyond that hard limit, any cache partitioning must be done using additional passes through the data — as will be shown in Section 4.7, this results in up to 89% slowdown compared to the techniques we propose. Second, cache partitioning assumes *exclusive use* of the cache, but this assumption is unlikely to be valid in an environment with multiple ongoing activities. Once the cache is too busy with other requests to effectively retain its partition, *the performance may degrade significantly* (up to 78% in the experiments in Section 4.7). Hence, we would like to explore an alternative technique that does not suffer from these limitations.

4.1.2 Our Approach: Cache Prefetching

Rather than trying to *avoid* CPU cache misses by building tiny (cache-sized) hash tables, we instead propose to exploit cache prefetching to *hide* the cache miss latency associated with accessing normal (memory-sized) hash tables, by overlapping these cache misses with computation.

Challenges in Applying Prefetching to Hash Join. A naïve approach to prefetching for hash join might simply try to hide the latency within the processing of a single tuple. For example, to improve hash table probing performance, one might try to prefetch hash bucket headers, hash buckets, build tuples, etc. Unfortunately, such an approach would have little benefit because later memory references often depend upon previous ones (via pointer dereferences). Existing techniques for overcoming this *pointer-chasing problem* [64] will not work because the randomness of hashing makes it impossible to predict the memory locations to be prefetched.

The good news is that although there are many dependencies *within* the processing of a single tuple, dependencies are less common *across* subsequent tuples due to the random nature of hashing. Hence our approach is to exploit *inter-tuple parallelism* to overlap the cache misses of one tuple with the computation and cache misses associated with other tuples.

A natural question is whether either the hardware or the compiler could accomplish this inter-tuple cache prefetching automatically; if so, we would not need to modify the hash join software. Unfortunately, the answer is no. As described in Section 1.2.2, hardware-based prefetching techniques [4] rely upon recognizing regular and predictable (e.g., strided) patterns in the data address stream, but the inter-tuple hash table probes do not exhibit such behavior. In many modern processors, the hardware also attempts to overlap cache misses by speculating ahead in the instruction stream. However, as described in Section 1.2.2, although this approach is useful for hiding the latency of primary data cache misses that hit in the secondary cache, the instruction window size is often a magnitude smaller than the instructions wasted due to a cache miss to main memory, and is even smaller compared with the amount of processing required for a single tuple. While our prefetching approaches (described below) are inspired by compiler-based scheduling techniques, existing compiler techniques for scheduling prefetches [64, 73] cannot handle the ambiguous data dependencies present in the hash join code (as will be discussed in detail in Sections 4.4.4 and 4.5.3).

Overcoming these Challenges. To effectively hide the cache miss latencies in hash join, we propose and evaluate two new prefetching techniques: *group prefetching* and *software-pipelined prefetching*. For group prefetching, we apply modified forms of compiler transformations called *strip mining* and *loop distribution* (illustrated later in Section 4.4) to restructure the code such that hash probe accesses resulting from groups of G consecutive probe tuples can be pipelined.[1] The potential drawback of group prefetching is that cache miss stalls can still occur during the transition between groups. Hence our second prefetching scheme leverages a compiler scheduling technique called *software pipelining* [55] to avoid these intermittent stalls.

A key challenge that required us to extend existing compiler-based techniques in both cases is that although we expect dependencies across tuples to be unlikely, they are still possible, and we must take them into account to preserve correctness. If we did this conservatively (as the compiler would), it would

[1] In our experimental set-up in Section 4.7, $G = 25$ is optimal for hash table probing.

severely limit our potential performance gain. Hence we optimistically schedule the code assuming that there are no inter-tuple dependencies, but we perform some extra bookkeeping at runtime to check whether dependencies actually occur. If so, we temporarily stall the consumer of the dependence until it can be safely resolved. Additional challenges arose from the multiple levels of indirection and multiple code paths in hash table probing.

A surprising result in our study is that contrary to the conventional wisdom in the compiler optimization community that software pipelining outperforms strip mining, group prefetching appears to be more attractive than software-pipelined prefetching for hash joins. A key reason for this difference is that the code in the hash join loop is far more complex than the typical loop body of a numeric application (where software pipelining is more commonly used [55]).

The chapter is organized as follows. Section 4.2 discusses the related work in greater detail. Section 4.3 analyzes the dependencies in the join phase, the more complicated of the two phases, while Sections 4.4 and 4.5 use group prefetching and software-pipelined prefetching to improve the join phase performance. Section 4.6 discusses prefetching for the partition phase. Experimental results appear in Section 4.7. Finally, Section 4.8 summarizes this chapter.

4.2 Related Work

Since the GRACE hash join algorithm was first introduced [54], many refinements of this algorithm have been proposed for the sake of avoiding I/O by keeping as many intermediate partitions in memory as possible [29, 59, 74, 89, 101]. All of these hash join algorithms, however, share two common building blocks: (i) *partitioning* and (ii) *joining* with in-memory hash tables. To cleanly separate these two phases, we use GRACE as our baseline algorithm throughout this chapter. We point out, however, that our techniques should be directly applicable to the other hash join algorithms.

Several papers have developed techniques to improve the cache performance of hash joins [14, 66, 90]. Shatdal *et al.* showed that cache partitioning achieved 6-10% improvement for joining memory-resident relations with 100B tuples [90]. Boncz, Manegold and Kersten proposed using multiple passes in cache partitioning to avoid cache and TLB thrashing [14, 66]. They showed large performance improvements on real machines for joining vertically-partitioned relations in the Monet main memory database, under exclusive use of the CPU caches. They considered neither disk-oriented databases, more traditional physical layouts, multiple activities trashing the cache, nor the use of prefetching. They also proposed a variety of code optimizations (e.g., using shift-based hash functions) to reduce CPU time; these optimizations may be beneficial for our techniques as well.

As mentioned earlier, software prefetching has been used successfully in other scenarios [18, 19, 64, 73]. While software pipelining has been used to schedule prefetches in array-based programs [73], we have extended that approach to deal with more complex data structures, multiple code paths, and the read-write conflicts present in hash join.

Previous work demonstrated that TLB misses may degrade performance [14, 66], particularly when TLB misses are handled by software. However, the vast majority of modern processors (including Intel Pentium 4 [12] and Itanium 2 [44]) handle TLB misses in hardware. Moreover, TLB prefetching [85] can be supported by treating TLB misses caused by prefetches as normal TLB misses. For example, Intel Itanium 2 supports faulting prefetch instructions (*lfetch.fault* [44]), which can incur TLB misses and automatically load TLB entries. Hence, using our prefetching techniques, we can overlap TLB misses with computation, minimizing TLB stall time.

4.3 Dependencies in the Join Phase

In this section, we analyze the dependencies in a hash table visit in the join phase. Our purpose is to show why a naïve prefetching algorithm would fail. We study a concrete implementation of the in-memory hash table, as shown in Figure 4.2. The hash table consists of an array of hash buckets, each composed of a header and (possibly) an array of hash cells pointed to by the header. A hash cell represents a build tuple hashed to the bucket. It contains the tuple pointer and a fixed-length (e.g., 4 byte) hash code computed from the join key, which serves as a filter for the actual key comparisons. When a hash bucket contains only a single entry, the single hash cell is stored directly in the bucket header. When two or

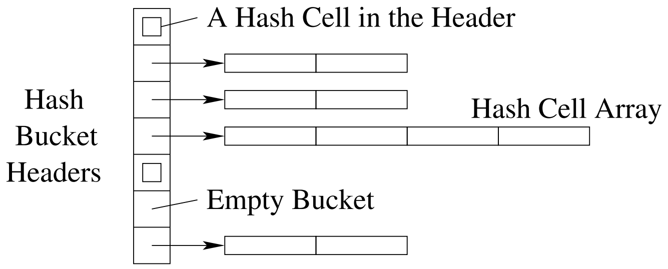


Figure 4.2: An in-memory hash table structure.

more tuples are hashed to the bucket, a hash cell array is allocated. When the array is full and a new tuple is to be hashed to the same bucket, a new array doubling the capacity is allocated and existing cells are copied to the new array.

A naïve prefetching algorithm would try to hide cache miss latencies *within* a single hash table visit by prefetching for potential cache misses, including hash bucket headers, hash cell arrays, and/or build tuples. However, this approach would fail because there are a lot of dependencies in a hash table visit. For example, the memory address of the bucket header is determined by the hashing computation. The address of the hash cell array is stored in the bucket header. The memory reference for a build tuple is dependent on the corresponding hash cell (in a probe). These dependencies essentially form a critical path; a previous computation or memory reference generates the memory address of the next reference, and must finish before the next one can start. Addresses would be generated too late for prefetching to hide miss latencies. Moreover, the randomness of hashing makes it almost impossible to predict memory addresses for hash table visits. These arguments are true for all hash-based structures.[2] Therefore, applying prefetching to the join phase algorithm is not a straightforward task.

4.4 Group Prefetching

Although dependencies *within* a hash table visit prevent effective prefetching, the join phase algorithm processes a large number of tuples and dependencies are less common *across* subsequent tuples due to

[2]The structure in Figure 4.2 improves upon chained bucket hashing, which uses a linked list of hash cells in a bucket. It avoids the pointer chasing problem of linked lists [65, 18].

the randomness of hashing. Therefore, our approach is to exploit *inter-tuple parallelism* to overlap cache miss latencies of one tuple with computations and miss latencies of other tuples. To ensure correctness, we must systematically intermix multiple hash table visits, reorder their memory references, and issue prefetches early enough. In this section, we propose group prefetching to achieve these goals.

4.4.1 Group Prefetching for a Simplified Probing Algorithm

We use a simplified probing algorithm to describe the idea of group prefetching. As shown in Figure 4.3(a), the algorithm assumes that all hash buckets have hash cell arrays and every probe tuple matches exactly one build tuple. It performs a probe per loop iteration.

As shown in Figure 4.3(b), the group prefetching algorithm combines multiple iterations of the original loop into a single loop body, and rearranges the probe operations into stages[3] . Each stage performs one computation or memory reference on the critical path for all the tuples in the group and then issues prefetch instructions for the memory references of the next stage. For example, the first stage computes the hash bucket number for every tuple and issues prefetch instructions for the hash bucket headers, which will be visited in the second stage. In this way, the cache miss to read the hash bucket header of a probe will be overlapped with hashing computations and cache misses for other probes. Prefetching is used similarly in the other stages except the last stage. Note that the dependent memory operations of the same probe are still performed one after another as before. However, the memory operations of different probes are now overlapped.

4.4.2 Understanding Group Prefetching

To better understand group prefetching, we generalize the previous algorithms of Figure 4.3(a) and (b) in Figure 4.3(c) and (d). Suppose we need to process N independent elements. For each element i, we need to make k dependent memory references, $m_i^1, m_i^2, \cdots, m_i^k$. As shown in Figure 4.3(c), a straightforward algorithm processes an element per loop iteration. The loop body is naturally divided into $k+1$ stages

[3] Technically, what we do are modified forms of compiler transformations called *strip-mining* and *loop distribution* [52].

```
foreach tuple in probe partition
{
    compute hash bucket number;
    visit the hash bucket header;
    visit the hash cell array;
    visit the matching build tuple to
      compare keys and produce output tuple;
}
```

(a) A simplified probing algorithm

```
for i=0 to N-1 do
{
    code 0;
    visit (m_i^1); code 1;
    visit (m_i^2); code 2;
    ... ...
    visit (m_i^k); code k;
}
```

(c) Processing an element per iteration

```
foreach group of tuples in probe partition
{
    foreach tuple in the group {
      compute hash bucket number;
      prefetch the target bucket header;
    }
    foreach tuple in the group {
      visit the hash bucket header;
      prefetch the hash cell array;
    }
    foreach tuple in the group {
      visit the hash cell array;
      prefetch the matching build tuple;
    }
    foreach tuple in the group {
      visit the matching build tuple to
        compare keys and produce output tuple;
    }
}
```

(b) Group prefetching for simplified probing

```
for j=0 to N-1 step G do
{
    for i=j to j+G-1 do {
      code 0;
      prefetch (m_i^1);
    }
    for i=j to j+G-1 do {
      visit (m_i^1); code 1;
      prefetch (m_i^2);
    }
    for i=j to j+G-1 do {
      visit (m_i^2); code 2;
      prefetch (m_i^3);
    }
    ... ...
    for i=j to j+G-1 do {
      visit (m_i^k); code k;
    }
}
```

(d) General group prefetching algorithm

Figure 4.3: Group prefetching.

Table 4.1: Terminology used throughout Chapter 4.

Name	Definition
k	# of dependent memory references for an element
G	group size in group prefetching
D	prefetch distance in software-pipelined prefetching
T_1	full latency of a cache miss
T_{next}	latency of an additional pipelined cache miss
C_l	execution time for code l, $l = 0, 1, \ldots, k$

by the k memory references. *Code 0* (if exists) computes the first memory address m_i^1. *Code 1* uses the contents in m_i^1 to compute the second memory address m_i^2. Generally *code l* uses the contents in m_i^l to compute the memory address m_i^{l+1}, where $l = 1, \cdots, k-1$. Finally, *code k* performs some processing using the contents in m_i^k. If every memory reference m_i^l incurs a cache miss, the algorithm will suffer from kN expensive, fully exposed cache misses.

Since the elements are independent of each other, we can use group prefetching to overlap cache miss latencies across multiple elements, as shown in Figure 4.3(d). The group prefetching algorithm combines the processing of G elements into a single loop body. It processes *code l* for all the elements in the group before moving on to *code* $l+1$. As soon as an address is computed, the algorithm issues a prefetch for the corresponding memory location in order to overlap the reference across the processing of other elements.

Now we determine the condition for fully hiding all cache miss latencies. Suppose the execution time of *code l* is C_l, the full latency of fetching a cache line from main memory is T_1, and the additional latency of fetching the next cache line in parallel is T_{next}, which is the inverse of memory bandwidth. (Table 4.1 shows the terminology used throughout the chapter.) Assume every m_i^l incurs a cache miss and there are no cache conflicts. Note that we use these assumptions only to simplify the derivation of the conditions. Our experimental evaluations include all the possible effects of locality and conflicts in hash joins. Then, the sufficient condition for fully hiding all cache miss latencies is as follows:

$$\begin{cases} (G-1)\cdot C_0 \geq T_1 \\ (G-1)\cdot \max\{C_l, T_{next}\} \geq T_1,\ l = 1, 2, \cdots, k \end{cases} \tag{4.1}$$

We will give the proof for the condition in the next subsection. For an intuitive explanation, let us focus on the first element in a group, element j. The prefetch for m_j^1 is overlapped with the processing of the remaining $G-1$ elements at code stage 0. The first inequality ensures that the processing of the remaining $G-1$ elements takes longer time than a single memory reference so that the prefetched memory reference will complete before the visit operation for m_j^1 in code stage 1. Similarly, the prefetch for m_j^{l+1} is overlapped with the processing of the remaining $G-1$ elements at code stage l. The second inequality ensures that the memory reference latency is fully hidden. Note that T_{next} corresponds to the memory bandwidth consumption of the visit operations of the remaining $G-1$ elements. In the proof, we also show that memory access latencies for other elements are fully hidden by simple combinations of the inequalities.

We can always choose a G large enough to satisfy the second inequality since T_{next} is always greater than 0. However, when *code 0* is empty, m_j^1 can not be fully hidden. Fortunately, in the previous simplified probing algorithm, *code 0* computes the hash bucket number and is not empty. Therefore, we can choose a G to hide all the cache miss penalties.

In the above, cache conflict misses are ignored for simplicity of analysis. However, we will show in Section 4.7 that conflict miss is a problem when G is too large. Therefore, among all possible G's that satisfy the above inequalities, we should choose the smallest to minimize the number of concurrent prefetches and conflict miss penalty.

4.4.3 Critical Path Analysis for Group Prefetching

In the following, we use critical path analysis to study the processing of a group, i.e. an iteration of the outer loop in Figure 4.3(d). For simplicity of analysis, we assume that every m_i^l incurs a cache miss and there are no cache conflicts among the memory references in a group. Figure 4.4 shows the graph for critical path analysis. A vertex represents an event. An edge from vertex A to B indicates that event B depends on event A and the weight of the edge is the minimal delay. (For simplicity, zero weights are not shown in the graph.) The run time of a loop iteration corresponds to the length of the critical path in the graph, i.e. the longest weighted path from the start to the end.

The graph is constructed as follows. We use three kinds of vertices:

- **P vertex**: the execution of a prefetch instruction
- **C vertex**: the start of *code 0*
- **VC vertex**: the start of a visit and *code l* ($l = 1, 2, \cdots, k$)

Vertex subscripts indicate the elements being processed. Their superscripts correspond to the memory addresses in the program for P vertices, and to the code stage for C and VC vertices. In Figure 4.4, a row of vertices corresponds to an inner loop that executes a code stage for all the elements in a group. We use three kinds of edges:

- **Instruction flow edges**: They go from left to right in every row and from top to bottom across rows. For example, code 0 for element j (vertex C_j^0) and the prefetch for m_j^1 (vertex P_j^1) are executed before code 0 for element j+1 (vertex C_{j+1}^0) and prefetch for m_{j+1}^1 (vertex P_{j+1}^1). The second inner loop (the second row) starts after the first inner loop finishes. We assume that code l takes a fixed amount of time C_l to execute, which is shown as weights of outgoing edges from C and VC vertices. The instruction overhead of the visit and the following prefetch instruction is also included in it. So the other instruction flow edges have zero weights.
- **Latency edges**: an edge from a P vertex to the corresponding VC vertex represents the prefetched memory reference with full latency T as its weight.
- **Bandwidth edges**: an edge between VC vertices represents memory bandwidth. Usually an additional (independent) cache miss can not be fully overlapped with the previous one. It takes T_{next} more time to finish, which is the inverse of memory bandwidth. [4]

Now we consider the critical path of the graph. If we ignore for a moment all latency edges, the graph becomes clear and simple: All paths go from left to right in a row and from top to bottom from the start

[4]The bandwidth edges are not between the P vertices because prefetch instructions only put requests into a buffer and it is the actual memory visits that wait for the operations to finish.

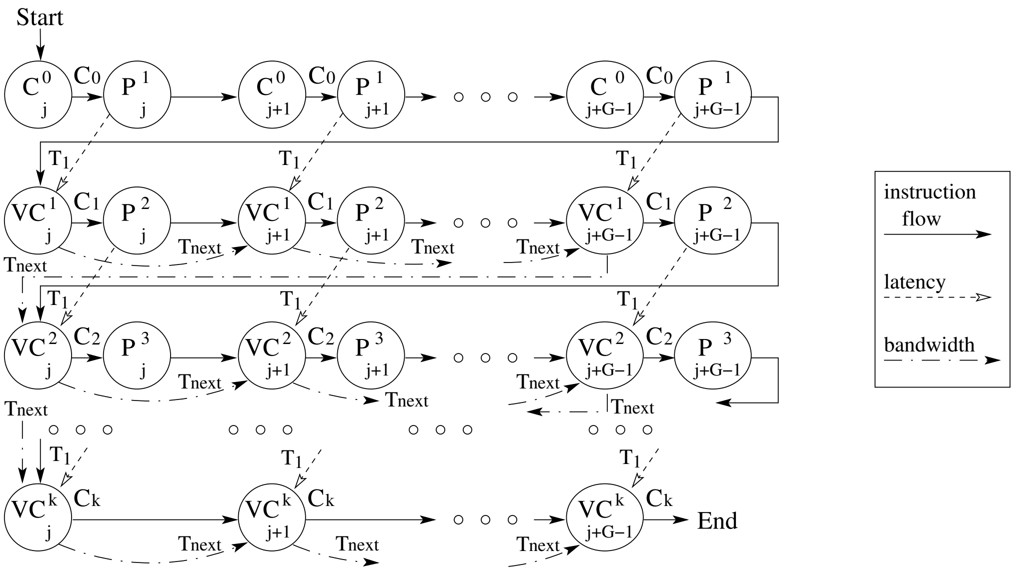


Figure 4.4: Critical path analysis for an iteration of the outer loop body in Figure 4.3(d).

to the end; alternative paths are all local between instruction flow edges and bandwidth edges. Since the critical path is the longest path, we can ignore an edge if there is a longer path connecting the same vertices. Intuitively, we can choose a large G so that latency edges are shorter than the paths along rows and they can be ignored. In this situation, the critical path of the graph is the longest path *along the rows*.

We would like to derive the condition to fully hide all cache miss latencies. If all cache miss latencies are hidden, all latency edges will not be on the critical path, vice versa. Therefore, it is equivalent to derive the condition to ensure that all latency edges are shorter than paths along rows. We have the following theorem.

Theorem 1. *The following condition is sufficient for fully hiding all cache miss latencies in the general group prefetching algorithm:*

$$\begin{cases} (G-1)\cdot C_0 \geq T \\ (G-1)\cdot \max\{C_l, T_{next}\} \geq T,\ l = 1, 2, \cdots, k \end{cases}$$

Proof. The first inequality ensures that the first latency edge from row 0, i.e. the edge from vertex P_j^1 to vertex VC_j^1 in the graph, is shorter than the path along row 0. The second inequality ensures that the first latency edge from row l in the graph, i.e. the edge from vertex P_j^{l+1} to vertex VC_j^{l+1}, is shorter than the corresponding path along row l, where $l = 1, 2, \cdots, k-1$. Note that the inequality when $l = k$ is used only in the proof below.

For the other latency edges, we can prove they are shorter than the paths along rows with a simple combination of the two inequalities. For the x-th latency edge from row 0, i.e. the edge from vertex P^1_{j+x-1} to vertex VC^1_{j+x-1}, the length of the path along the row is as follows:

$$\begin{aligned}
\text{Path Length} &= (G-x)\cdot C_0 + (x-1)\cdot \max\{C_1, T_{next}\} \\
&= [(G-x)\cdot(G-1)\cdot C_0 + (x-1)\cdot(G-1)\cdot \max\{C_1, T_{next}\}]/(G-1) \\
&\geq [(G-x)\cdot T + (x-1)\cdot T]/(G-1) \\
&= T
\end{aligned}$$

For the x-th latency edge from row l, i.e. the edge from vertex P^{l+1}_{j+x-1} to vertex VC^{l+1}_{j+x-1}, where $l = 1, 2, \cdots, k-1$, the length of the path along the row is as follows:

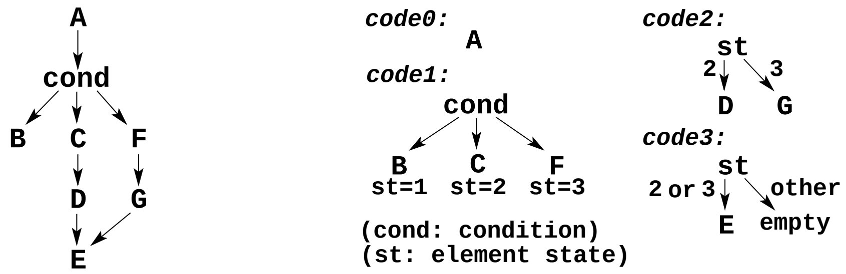
$$\begin{aligned}
\text{Path Length} &= (G-x)\cdot \max\{C_l, T_{next}\} + (x-1)\cdot \max\{C_{l+1}, T_{next}\} \\
&= [(G-x)\cdot(G-1)\cdot \max\{C_l, T_{next}\} + (x-1)\cdot(G-1)\cdot \max\{C_{l+1}, T_{next}\}]/(G-1) \\
&\geq [(G-x)\cdot T + (x-1)\cdot T]/(G-1) \\
&= T
\end{aligned}$$

Therefore, when the two inequalities are satisfied, all latency edges are shorter than the corresponding paths along rows and all cache miss latencies are fully hidden. $\square$

4.4.4 Dealing with Complexities

Previous research showed how to prefetch for two dependent memory references for array-based codes [72]. Our group prefetching algorithm solves the problem of prefetching for an arbitrary fixed number k of dependent memory references.

We have implemented group prefetching for both hash table building and probing. In contrast to the simplified probing algorithm, the actual probing algorithm contains multiple code paths: There could be zero or multiple matches, hash buckets could be empty, and there may not be a hash cell array in a bucket. To cope with this complexity, we keep state information for the G tuples of a group. We divide



(a) An example (b) Code stages in group prefetching

Figure 4.5: Dealing with multiple code paths.

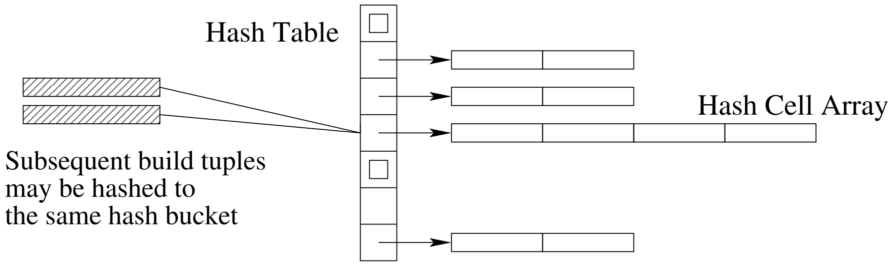
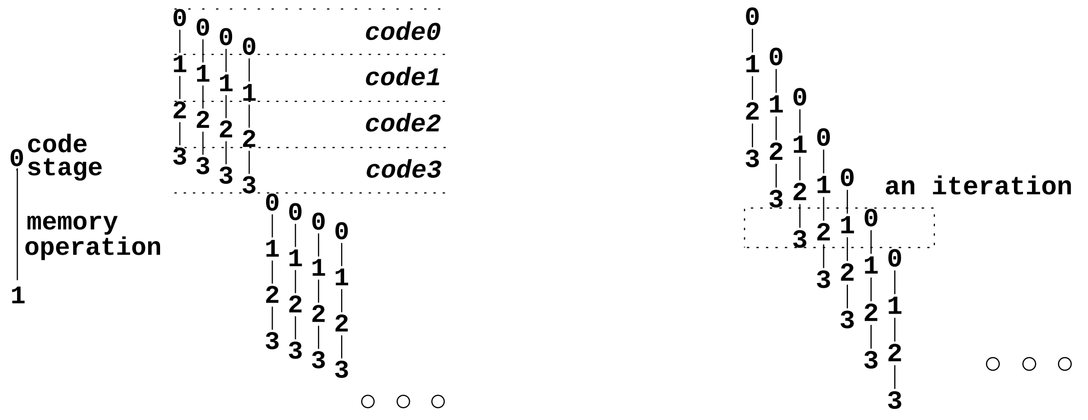


Figure 4.6: A read-write conflict.

each possible code path into code pieces on the boundaries of dependent memory references. Then we combine the code pieces at the same position of different code paths into a single stage using conditional tests on the tuple states. Figure 4.5 illustrates the idea of this process. Note that the common starting point of all code paths is in *code 0*. The first code piece including a branch sets the state of an element. Then subsequent code stages test the state and execute the code pieces for the corresponding code paths. The total number of stages ($k+1$) is the largest number of code pieces along any original code path.

When multiple independent cache lines are visited at a stage (e.g., to visit multiple build tuples), our algorithm issues multiple independent prefetches in the previous stage.

The group prefetching algorithm must also cope with read-write conflicts. Though quite unlikely, it is possible that two build tuples in a group may be hashed into the same bucket, as illustrated in Figure 4.6. However, in our algorithm, hash table visits are interleaved and no longer atomic. Therefore, a race condition could arise; the second tuple might see an inconsistent hash bucket being changed by the first one. Note that this complexity occurs because of the read-write nature of hash table building. To cope



(a) Group prefetching ($G = 4$) (b) Software-pipelined prefetching ($D = 1$)

Figure 4.7: Intuitive pictures of the prefetching schemes.

with this problem, we set a busy flag in a hash bucket header before inserting a tuple. If a tuple is to be inserted into a busy bucket, we delay its processing until the end of the group prefetching loop body. At this natural group boundary, the previous access to the busy hash bucket must have finished. Interestingly, the previous access has also warmed up the cache for the bucket header and hash cell array, so we insert the delayed tuple without prefetching. Note that the algorithm can deal with any number of delayed tuples (to tolerate skews in the key distribution).

4.5 Software-Pipelined Prefetching

In this section, we describe our technique of exploiting software pipelining to schedule prefetches for hash joins. We then compare our two prefetching schemes.

Figure 4.7 illustrates the difference between group prefetching and software-pipelined prefetching intuitively. Group prefetching hides cache miss latencies within a group of elements and there is no overlapping memory operation between groups. In contrast, software-pipelined prefetching combines different code stages of different elements into an iteration and hides latencies across iterations. It keeps running without gaps and therefore may potentially achieve better performance.

```
prologue;
for j=0 to N-3D-1 do
{
   tuple j+3D:
     compute hash bucket number;
     prefetch the target bucket header;

   tuple j+2D:
     visit the hash bucket header;
     prefetch the hash cell array;

   tuple j+D:
     visit the hash cell array;
     prefetch the matching build tuple;

   tuple j:
     visit the matching build tuple to
       compare keys and produce output tuple;
}
epilogue;
```

(a) Software-pipelined prefetching for simplified probing

```
prologue;
for j=0 to N-kD-1 do
{
   i=j+kD;
   code 0 for element i;
   prefetch (m_i^1);

   i=j+(k-1)D;
   visit (m_i^1); code 1 for element i;
   prefetch (m_i^2);

   i=j+(k-2)D;
   visit (m_i^2); code 2 for element i;
   prefetch (m_i^3);

   ... ...

   i=j;
   visit (m_i^k); code k for element i;
}
epilogue;
```

(b) General software-pipelined prefetching

Figure 4.8: Software-pipelined prefetching.

4.5.1 Understanding Software-pipelined Prefetching

Figure 4.8(a) shows the software-pipelined prefetching for the simplified probing algorithm. The subsequent stages for a particular tuple are processed D iterations away. (D is called the *prefetch distance* [72].) Figure 4.7(b) depicts the intuitive picture when $D = 1$. Suppose the left-most line in the dotted rectangle corresponds to tuple j. Then, an iteration combines the processing of stage 0 for tuple $j+3D$, stage 1 for tuple $j+2D$, stage 2 for tuple $j+D$, and stage 3 for tuple j.

Figure 4.8(b) shows the generalized algorithm for software-pipelined prefetching. In the steady state, the pipeline has $k+1$ stages. The loop body processes a different element for every stage. The subsequent stages for a particular element are processed D iterations away. Intuitively, if we make the intervals between code stages for the same element sufficiently large, we will be able to hide cache miss

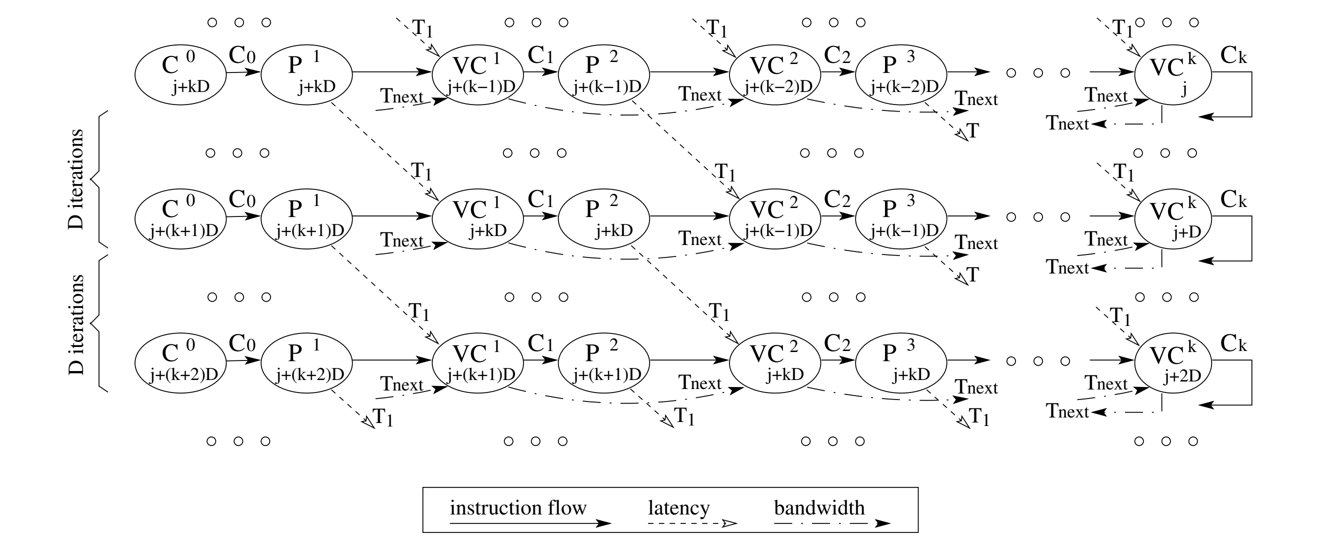


Figure 4.9: Critical path analysis for software-pipelined prefetching (steady state).

latencies. Under the same assumption as in Section 4.4.2, the sufficient condition for hiding all cache miss latencies in the steady state is as follows. (We will derive this condition in the next subsection.)

$$D \cdot (max\{C_0 + C_k, T_{next}\} + \sum_{l=1}^{k-1} \max\{C_l, T_{next}\}) \geq T \tag{4.2}$$

We can always choose a D sufficiently large to satisfy this condition. In our experiments in Section 4.7, we will show that conflict miss is a problem when D is too large. Therefore, similar to group prefetching, we should choose the smallest D to minimize the number of concurrent prefetches.

4.5.2 Critical Path Analysis for Software-pipelined Prefetching

We perform critical path analysis using Figure 4.9. The graph is constructed in the same way as Figure 4.4, though a row here corresponds to a single loop iteration in the general software-pipelined prefetching algorithm. Instruction flow edges are still from left to right in a row and from top to bottom across rows. Focusing on the latency edges, we can see the processing of the subsequent stages of an element. Two subsequent stages of the same element are processed in two separate rows that are D iterations away.

If the paths along the rows are longer, the latency edges can be ignored because they are not on the critical path and the cache miss latencies are fully hidden. The sufficient condition for hiding all cache miss latencies is given in the following theorem.

Theorem 2. *The following condition is sufficient for fully hiding all cache miss latencies in the general software-pipelined prefetching algorithm:*

$$D \cdot (max\{C_0 + C_k, T_{next}\} + \sum_{l=1}^{k-1} \max\{C_l, T_{next}\}) \geq T$$

Proof. The left-hand side of the inequality is the total path length of D rows in Figure 4.9. Clearly, when this length is greater than or equal to the weight of a latency edge, latency edges can be ignored in critical path analysis and all cache miss latencies are fully hidden. □

4.5.3 Dealing with Complexities

We have implemented software-pipelined prefetching by modifying our group prefetching algorithm. The code stages are kept almost unchanged. To apply the general model in Figure 4.8(b), we use a circular array for state information; the index j in the general model is implemented as the array index. We choose the array size to be a power of 2 and use bit mask operation for modular index computation to reduce overhead. Moreover, since *code 0* and *code k* of the same element is processed kD iterations away, we ensure the array size is at least $kD+1$.

The read-write conflict problem in hash table building is solved in a more sophisticated way. Since there is no place (like the end of a group in group prefetching) to conveniently process all the conflicts, we have to deal with the conflicts in the pipeline stages themselves. We build a waiting queue for each busy hash bucket. The hash bucket header contains the array index of the tuple updating the bucket. The state information of a tuple contains a pointer to the next tuple waiting for the same bucket. When a tuple is to be inserted into a busy bucket, it is appended to the waiting queue. When we finish hashing a tuple, we check its waiting queue. If the queue is not empty, we record the array index of the first waiting tuple in the bucket header, and perform the previous code stages for it. When this tuple gets to the last stage, it will handle the next tuple in the waiting queue if it exists.

4.5.4 Group vs. Software-pipelined Prefetching

Both prefetching schemes try to increase the interval between a prefetch and the corresponding visit, in order to hide cache miss latency. According to the sufficient conditions, software-pipelined prefetching can always hide all miss latencies, while group prefetching achieves this only when *code 0* is not empty (as is the case of the join phase). When *code 0* is empty, the first cache miss cannot be hidden. However, with a large group of elements, the amortized performance impact can be small.

In practice, group prefetching is easier to implement. The natural group boundary provides a place to do any necessary "clean-up" processing (e.g., for read-write conflicts). Moreover, the join phase can pause at group boundaries and send outputs to the parent operator to support pipelined query processing. Although a software pipeline may also be paused, the restart costs will diminish its performance advantage. Furthermore, software-pipelined prefetching has larger bookkeeping overhead because it uses modular index operations and because it maintains larger amount of state information (such as the waiting queue for handling read-write conflicts).

4.6 Prefetching for the Partition Phase

Having studied how to prefetch for the join phase of the hash join algorithm, in this section, we discuss prefetching for the partition phase. In the partition phase, an input relation is divided into multiple output partitions by hashing on the join keys. Typically, the algorithm keeps in main memory an input page for the input relation and an output page for every intermediate partition. The algorithm processes every input page, and examines every input tuple in an input page. It computes the partition number from the tuple join key. Then it extracts (projects) the columns of the input tuple relevant to the database query performing the hash join operation and copies them to the target output buffer page. When an output buffer page is full, the algorithm writes it out to the corresponding partition and allocates a new page.

While partitioning tables for a hash join, commercial database systems often construct a filter to quickly discard probe tuples that do not have any matches [59]. Such filters may improve join performance significantly when a large number of probe tuples do not match any tuples in the build relation (e.g., there is a predicate on the build relation in a foreign-key join). Typically, a join key is mapped to several bit positions in the filter. While partitioning the build relation, the algorithm sets the bit positions for every build tuple in the filter. Then the algorithm checks the bit positions for every probe tuple. If some bits are not set, the probe join key does not match any existing build join keys. (Filters will be discussed in more detail in Chapter 5, where we extend the existing filter scheme to collect more information for better join performance.)

Like the join phase, the I/O partitioning phase employs hashing: It computes the partition number and the bit positions in the filter of a tuple by hashing on the tuple's join key. Because of the randomness of hashing, the resulting memory addresses are difficult to predict. Moreover, the processing of a tuple also needs to make several dependent memory references, whereas the processing of subsequent tuples are mostly independent due to the randomness of hashing. Therefore, we employ group prefetching and software-pipelined prefetching for the I/O partitioning phase.

There are read-write conflicts in visiting the output buffers. Imagine that two tuples are hashed to the same partition. When processing the second tuple, the algorithm may find that the output buffer has no space and needs to be written out. However, it is possible that the data from the first tuple has not been copied into the output buffer yet because of the reorganization of processing. To solve this problem, in group prefetching, we wait until the end of the loop body to write out the buffer and process the second tuple. In software-pipelined prefetching, we use waiting queues similar to those for hash table building in the join phase. Note that setting bits in the filter does not incur read-write conflicts because the operations are idempotent.

4.7 Experimental Results

In this section, we present experimental results to quantify the benefits of our cache prefetching techniques. We begin by describing the experimental setups. Next, we show that hash join is CPU bound with reasonable I/O bandwidth. After that, we focus on the user-mode CPU cache performance of hash joins. Similar to Chapter 3, we perform both simulations and Itanium 2 experiments to understand the user-mode cache behaviors of the prefetching techniques. Finally, we evaluate the impact of our cache prefetching techniques on the elapsed real times of hash joins with disk I/Os.

4.7.1 Experimental Setup

Implementation Details. We have implemented our own hash join engine. For real machine I/O experiments, we implemented a buffer manager that stripes pages across multiple disks and performs I/O prefetching with background worker threads. For CPU cache performance studies, we store relations and intermediate partitions as disk files for simplicity. We employ the slotted page structure and support fixed length and variable length attributes in tuples. Schemas and statistics are kept in separate description files for simplicity, the latter of which are used to compute hash table sizes and numbers of partitions.

Our baseline algorithm is the GRACE hash join algorithm [54]. The in-memory hash table structure follows Figure 4.2 in Section 4.3. A simple XOR and shift based hash function is used to convert join keys of any length to 4-byte hash codes. Typically the same hash codes are used in both the partition and the join phase. Partition numbers in the partition phase are the hash codes modulo the total number of partitions. Hash bucket numbers in the join phase are the hash codes modulo the hash table size. Our algorithms ensure that the hash table size is a relative prime to the number of partitions. Because the same hash codes are used in both phases, we avoid the memory access and computational overheads of reading the join keys and hashing them a second time, by storing hash codes in the page slot area in the intermediate partitions and reusing them in the join phase. Note that changing the page structure of intermediate partitions is relatively easy because the partitions are only used in hash joins.

We implemented three prefetching schemes for both the partition phase and the join phase algorithm: simple prefetching, group prefetching, and software-pipelined prefetching. As suggested by the name, simple prefetching uses straightforward prefetching techniques, such as prefetching an entire input page after a disk read. In our simulation study, we use simple prefetching as an enhanced baseline in order to show the additional benefit achieved using our more sophisticated prefetching schemes. We use gcc and insert prefetch instructions into C++ source code using inline ASM macros. In our cache performance study on the Itanium 2 machine, we compare the performance of both gcc and icc generated executables. As will be shown in Section 4.7.6, executables generated by icc are significantly faster for all techniques (including the baseline and cache partitioning), and the best performance is achieved with "icc -O3". Therefore, we compile with "icc -O3" for our Itanium 2 experiments. Note that "icc -O3" already enhances a program by automatically (aggressively) inserting prefetches. In fact, we find that the icc generated baseline achieves even slightly better performance than the simple prefetching approach. Therefore, we omit the simple prefetching curves when presenting Itanium 2 results.

Cache Partitioning. Cache partitioning generates cache-sized build partitions so that every build partition and its hash table can fit in cache, greatly reducing the cache misses in the join phase. It has been shown to be effective in main-memory and memory-resident database environments [14, 90]. We have implemented the cache partitioning algorithm for disk-oriented database environments. The algorithm partitions twice: The I/O partition phase generates memory-sized partitions, which are subsequently partitioned again in memory as a preprocessing step for the join phase.

Experimental Design. In our experiments, we assume a fixed amount of memory (50 MB) is allocated for joining a pair of build and probe partitions in the join phase and the partition phase generates partitions that will tightly fit in this memory [5] . That is, in the baseline and our prefetching schemes, a build partition and its hash table fit tightly in the available memory. In the cache partitioning scheme, the partition sizes are also computed to satisfy the algorithm constraints and best utilize available memory.

Build relations and probe relations have the same schemas: A tuple consists of a 4-byte join key and a fixed-length payload. We believe that selection and projection are orthogonal issues to our study and we do not perform these operations in our experiments. An output tuple contains all the fields of the matching build and probe tuples. The join keys are randomly generated. A build tuple may match zero or more probe tuples and a probe tuple may match zero or one build tuple. In our experiments, we vary the tuple size, the number of probe tuples matching a build tuple, and the percentage of tuples that have matches, in order to show the benefits of our solutions in various situations.

[5] The memory to cache size ratio is 50:2 for the simulation study, and it is 50:1.5 for the Itanium 2 machine. This ratio corresponds to the ratio of the hash table size (including build tuples) over the cache size, which is large enough to reflect the typical hash join cache behavior.

Measurement Methodology. We first measure GRACE hash join performance on the Itanium 2 machine with multiple disks to show that hash join is CPU-bound with reasonable I/O bandwidth. Therefore, it is important to study hash join cache performance.

We then evaluate the CPU cache performance (of user mode executions) of all the schemes through cycle-by-cycle simulations in order to better understand their cache behaviors. As described previously in Section 2.4.2, the memory hierarchy of the simulator is based on the Itanium 2 processor [44].

Finally, we verify the findings from the simulation study by running experiments on the Itanium 2 machine. The machine configuration is described previously in Section 2.4.1. We first focus on the CPU cache performance and measure the user mode execution times of all the schemes with the perfmon library on the Itanium 2 machine. Then we measure elapsed times to show the benefits of our prefetching algorithms while I/O operations are also in the picture. We measure the total elapsed time of an operation with a single `gettimeofday` system call. Since the resolution of the `gettimeofday` system call is 1 micro second on the Itanium 2 machine, we are able to measure the waiting time of an individual I/O request with the `gettimeofday` system call. We sum up the individual stall time measurements to obtain the reported I/O stall times. Because of this measurement methodology, the standard deviations of the I/O stall times are larger compared to the user-mode cache performance and the total elapsed time.

For every Itanium 2 experiment, we perform 30 runs and report the averages. For the user-mode cache performance measurements, the standard deviations are within 1% of the averages in all cases. For the total elapsed real time measurements, the standard deviations are within 5% of the averages in all cases. For the I/O stall times, the standard deviations are either within 10% of the averages or less than 1 second in all but one case where the standard deviation is 16.6% of the average.[6] Note that our performance comparisons are based on the more stable measurements of user-mode cache performance and elapsed real times.

[6] This is an experiment using 8 disks on the Itanium 2 machine. Since the eighth disk contains the root partition and the swap partition, the measurements with 8 disks tend to have larger variances than those with 1-7 disks.

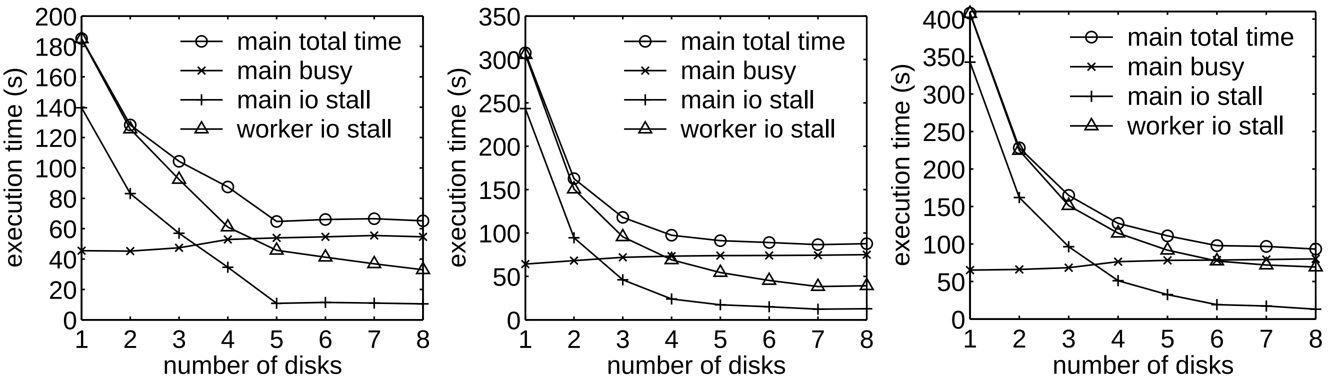
4.7.2 Is Hash Join I/O-Bound or CPU-Bound?

Our first set of experiments study whether hash joins are I/O-bound or CPU-bound. We measure the performance of GRACE hash joins on an Itanium 2 machine using up to 8 SCSI disks. The machine configuration and the buffer pool manager implementation are described previously in Section 3.6.3. In addition, the following two details of our buffer manager implementation are also relevant to these hash join experiments: (i) relations are striped across all of the disks used in an experiment in 256KB stripe units;[7] and (ii) worker threads perform background writing on behalf of the main thread. A worker thread calls `fdatasync` to flush any pages that may be cached in the file system cache after every k write operations ($k = 128$ in our experiments).

To be conservative, we would like to focus on the worst-case scenario where no intermediate partitions are cached in the main memory, thereby resulting in the maximum I/O demand for hash joins. Hence we measure the performance of the partition phase and the join phase in separate runs, and we ensure that the file system and disk caches are cold before every run using the techniques described earlier in Section 3.6.3. Depending on the queries, the join output tuples may either be written to disk or consumed in main memory by the parent operator; we perform experiments to evaluate both of these cases.

Figure 4.10 shows the performance of the partition phase and the join phase of joining a 2GB build relation with a 4GB probe relation varying the number of disks used. Tuples are 100 bytes. The algorithm generates 57 intermediate partitions so that a build partition and its hash table consume up to 50 MB of memory in the join phase. To better understand the elapsed times, we show four curves in every figure. The *main total* time is the elapsed real time of an algorithm phase. It is broken down into the *main busy* time and the *main io stall* time. The *main io stall* time is the time that the main thread spends either (i) waiting for an I/O completion notification from a worker thread, or (ii) waiting for an empty queue slot to enqueue an I/O request. The *main busy* time is computed by subtracting the *main io stall* time from the *main total* time; it is an approximation of the user-mode execution time. The *worker io stall* time is the maximum of the I/O stall times of individual worker threads.

[7] This models the typical data layout in commercial database systems. For example, the size of a stripe unit (a.k.a. extent) in IBM DB2 is between 8KB and 8MB [40]. By default, an extent in IBM DB2 contains 32 pages. Depending on the page size, the default extent can be 128KB, 256KB, 512KB, or 1MB large.



(a) Partition phase performance

(b) Join phase performance (outputs consumed in memory)

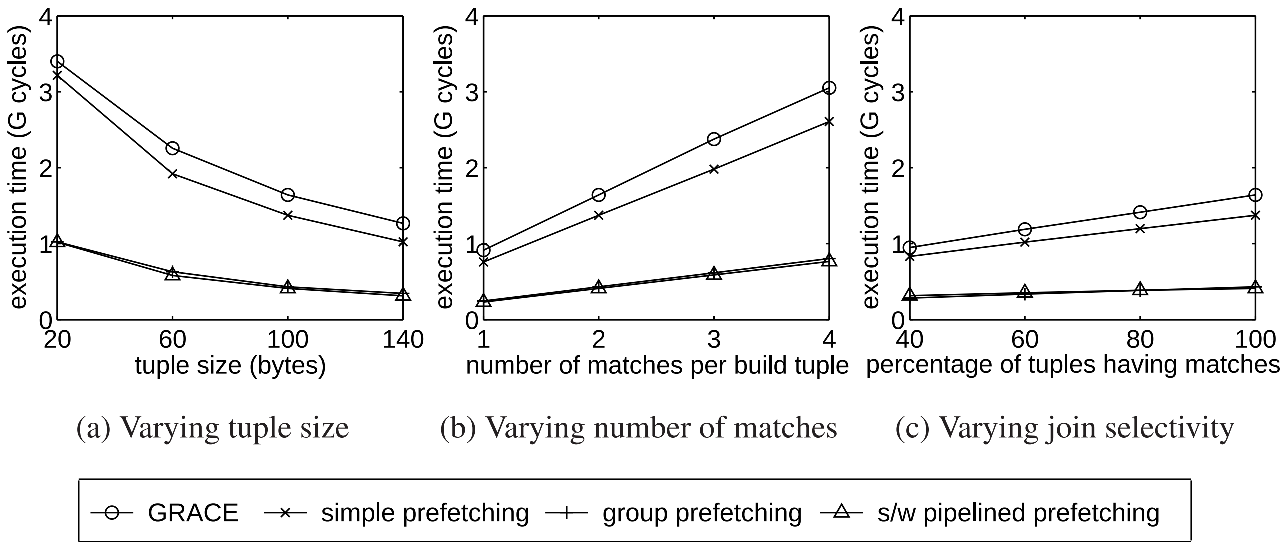
(c) Join phase performance (outputs written to disk)

Figure 4.10: Hash join is CPU-bound with reasonable I/O bandwidth.

As shown in Figure 4.10, the *worker io stall* time decreases dramatically as the number of disks increases, since this in turn decreases the number of I/O operations per disk. In contrast, the *main busy* time stays roughly the same across all of the experiments. This is because the memory and computational operations in the hash join do not depend on the number of disks. Combining the two trends, we see that hash joins are I/O-bound with a small number of disks (e.g. up to four disks). As more and more disks are added, hash joins gradually become CPU-bound.

As shown in Figure 4.10(a) and Figure 4.10(b), the partition phase and the join phase with outputs consumed in memory are CPU-bound with five or more disks: The *main busy* time is significantly larger than the *worker io stall* time, and the *main total* time becomes flat. As shown in Figure 4.10(c), the join phase with outputs written to disk becomes CPU-bound when seven disks are used.[8] Note that it is reasonable to use five or seven disks on the Itanium 2 machine because there are typically 10 disks per processor on a balanced DB server [98]. Therefore, we conclude that on the Itanium 2 machine, hash joins are CPU-bound with reasonable I/O bandwidth. The gap between the *main busy* time and the *worker io stall* time highlights the opportunity for reducing the total time by improving the hash join CPU performance.

[8]Although the curve markers seem to overlap, this claim is supported by experimental results in Section 4.7.7, which demonstrate that cache prefetching improves the performance in this case.



(a) Varying tuple size (b) Varying number of matches (c) Varying join selectivity

Figure 4.11: Join phase performance.

4.7.3 Join Phase Performance through Simulations

Figure 4.11 shows the join phase performance through simulations of the baseline and the prefetching schemes while varying the tuple size, the ratio of probe relation size to build relation size, and the percentage of tuples that have matches. The experiments model the processing of a pair of partitions in the join phase. In all experiments, the build partition fits tightly in the 50MB memory. By default, tuples are 100 bytes and every build tuple matches two probe tuples. As shown in the figure, group and software-pipelined prefetching achieve 3.02-4.04X speedups over the GRACE hash join. Because the central part of the join phase algorithm is hash table visiting, simple prefetching only obtains marginal benefit, a 1.06-1.24X speedup over the baseline. By exploiting the inter-tuple parallelism, group and software-pipelined prefetching achieve additional 2.65-3.40X speedups over the simple prefetching scheme.

In Figure 4.11(a), as we increase the tuple size from 20 bytes to 140 bytes, the number of tuples in the fixed sized partition decreases, leading to the decreasing trend of the curves. In Figure 4.11(b) and (c), the total number of matches increases as we increase the number of matches per build tuple or the percentage of tuples having matches. This explains the upward trends. Moreover, the probe partition size also increases in Figure 4.11(b), contributing to the much steeper curves than those in Figure 4.11(c).

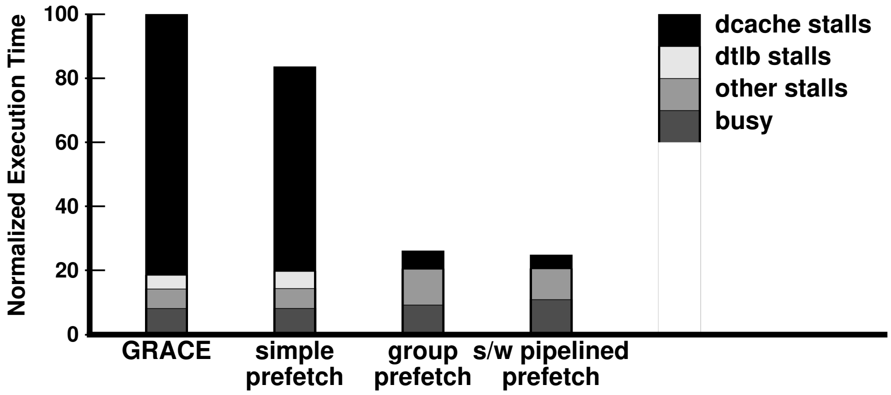
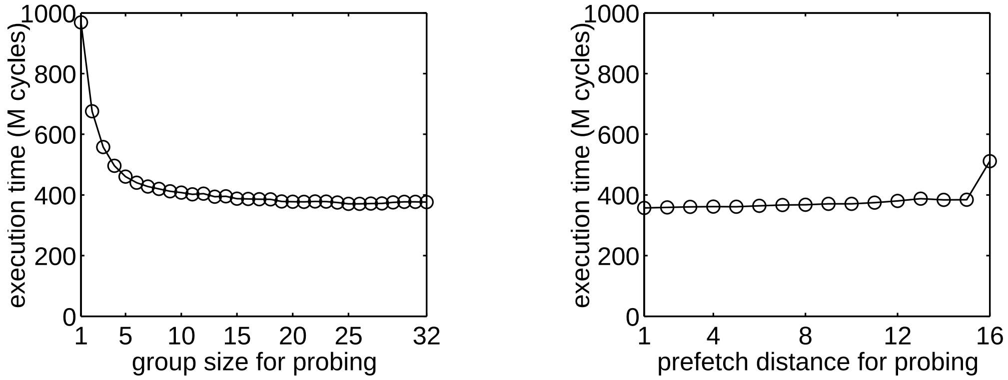


Figure 4.12: Execution time breakdown for join phase performance (Figure 4.11(a), 100B tuples).

Figure 4.12 shows the execution time breakdowns for the 100-byte points in Figure 4.11(a). The GRACE bar is shown as the "join" bar previously in Figure 4.1. Group prefetching and software-pipelined prefetching indeed successfully hide most of the data cache miss latencies. The simulator outputs confirm that the remaining data cache misses are mostly L1 cache misses but L2 hits or L1 and L2 misses but L3 hits. The (transformation, bookkeeping, and prefetching) overheads of the techniques lead to larger portions of busy times. The busy portion of the software-pipelined prefetching bar is larger than that of the group prefetching bar because of its more complicated implementation. Interestingly, other stalls also increase. A possible reason is that some secondary causes of stalls show up when the data cache stalls are reduced.

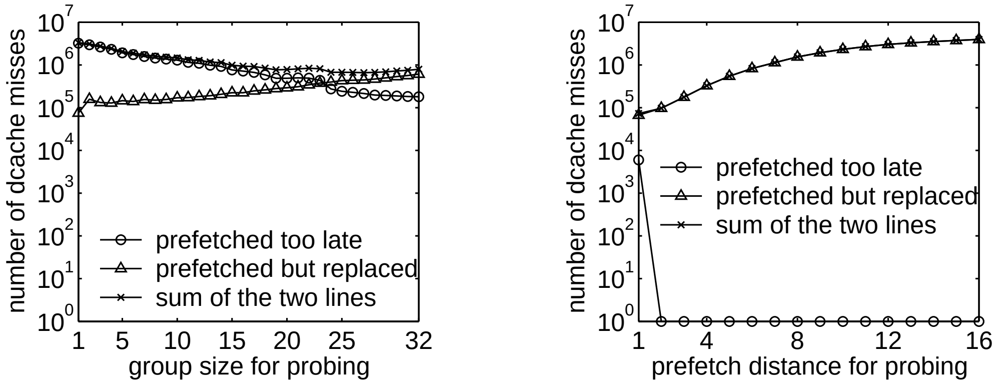
Join Performance Varying Algorithm Parameters. Figures 4.13(a) and (b) show the relationship between the cache performance and the parameters of our prefetching algorithms. We perform the same experiment as in Figure 4.11(a) when tuples are 100 bytes. We show the tuning results for only the hash table probing loop here but the curves for the hash table building loop have similar shapes. The optimal values for the hash table probing loop are $G = 25$ and $D = 1$, which are used in all simulation experiments unless otherwise noted.

From Figures 4.13(a) and (b), we see that both curves have large flat segments: A lot of parameter choices achieve near-optimal performance. In other words, our prefetching algorithms are quite robust against parameter choices. Therefore, the algorithm parameters can be pre-set for a range of machine configurations, and do not necessarily require tuning on every machine.



(a) Group prefetching (b) Software-pipelined prefetching

Figure 4.13: Tuning parameters of cache prefetching schemes for hash table probing in the join phase.



(a) Group prefetching (b) Software-pipelined prefetching

Figure 4.14: Analyzing prefetched memory references that still incur data cache misses to understand the tuning curves of the join phase.

From Figures 4.13(a) and (b), we see that performance becomes worse when the parameters are extremely small or extremely large. According to our models, the group size and the prefetch distance must be large enough to hide cache miss latencies. This explains the poor performance with small parameters. To verify this point and to understand the cases with large parameters, we analyze the breakdowns of prefetched memory references that still incur L1 data cache misses in Figure 4.14. We obtain the information from the statistics of the simulator, which tracks prefetches and memory references visiting the same cache lines[9] . As shown in Figures 4.14(a) and (b), the prefetched references that are still missing

[9]The simulator only matches a prefetch with the first reference after the prefetch visiting the same cache line. Therefore,

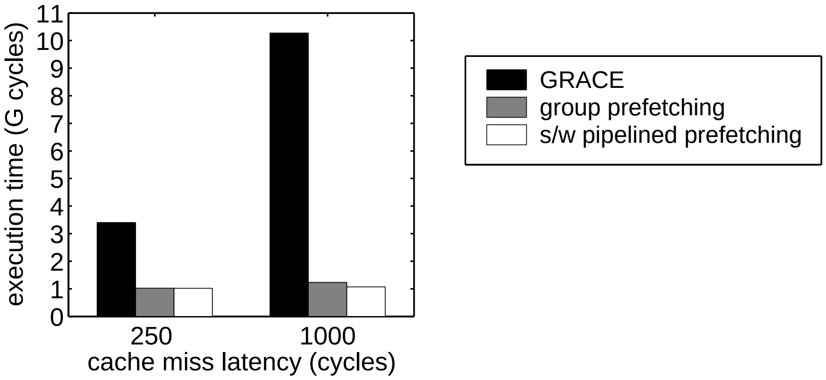


Figure 4.15: Join phase performance varying memory latency.

from the L1 data cache can be divided into two categories: references that are prefetched too late to hide all cache miss latencies, and references that are prefetched too early and already replaced from the data cache. The sum of the two curves show the total prefetched references that still incur data cache misses. From the figures, we can clearly see the trend: As group size increases, fewer and fewer references are prefetched too late, but more and more references are prefetched too early and already replaced. This trend explains the curve shapes in Figures 4.13(a) and (b). The poor performance when $D = 16$ is mainly due to prefetching too early, while the poor performance for small G values is mainly because prefetches are issued too late to hide cache miss latencies.

Join Performance Varying Memory Latency. Figure 4.15 shows the join phase performance when the memory latency T_1 is set to 250 cycles (default value) and 1000 cycles in the simulator. The optimal parameters are $G = 25$ and $D = 5$ for hash table probing when the memory latency is 1000 cycles. As shown in the figure, the execution time of GRACE hash join increases dramatically as the memory latency increases. In contrast, the execution times of both group and software-pipelined prefetching only increase slightly, thus achieving 8.3-9.6X speedups over GRACE hash join. This means that the prefetching algorithms will still keep up when the processor/memory speed gap increases even more (4 times in our experiments) as expected to happen in the future by the technology trend.

later references to the same cache line are not counted as prefetched references in the simulator statistics and not included in Figure 4.14. Nevertheless, the curves should show the correct trends

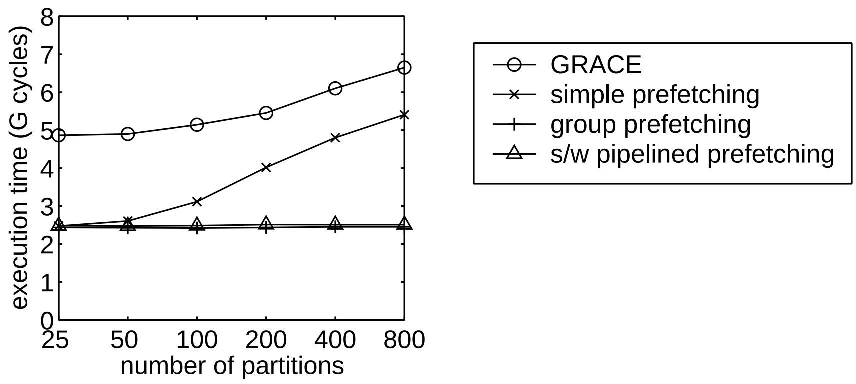


Figure 4.16: Partition phase performance.

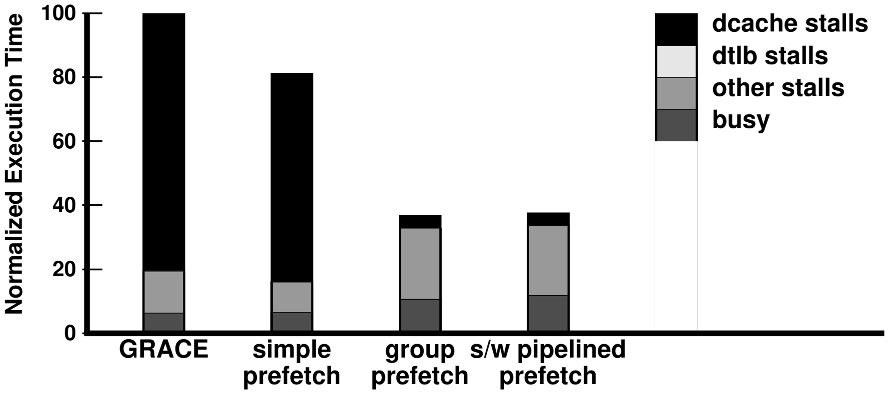


Figure 4.17: Execution time breakdown for Figure 4.16 with 800 partitions.

4.7.4 Partition Phase Performance through Simulations

Figure 4.16 shows the partition phase performance partitioning a 200MB build relation and a 400MB probe relation through simulations. We vary the number of partitions from 25 to 800, and the tuple size is 100 bytes. (Unlike all the other experiments, the generated partitions may be much smaller than 50 MB.) As shown in the figure, we see that as the number of partitions increases, the simple approach of prefetching all input and output pages and assume they stay in the CPU cache is less and less effective, while our two prefetching schemes maintain the same level of performance. Compared to the GRACE baseline, our prefetching schemes achieve 1.96-2.71X speedups for the partition phase.

Figure 4.17 shows the execution time breakdown for Figure 4.16 where 800 partitions are generated.

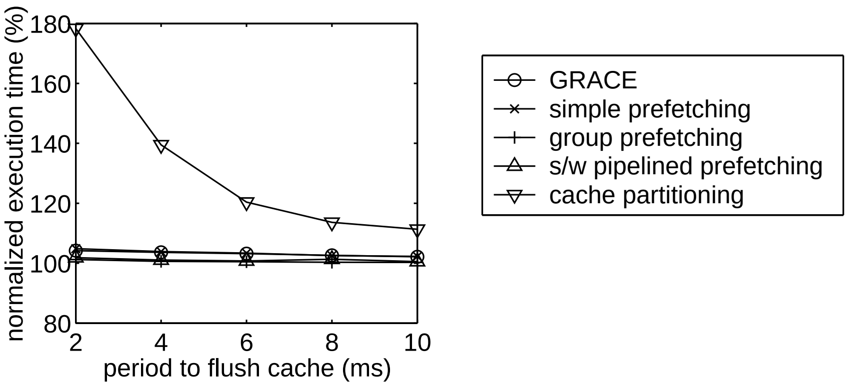
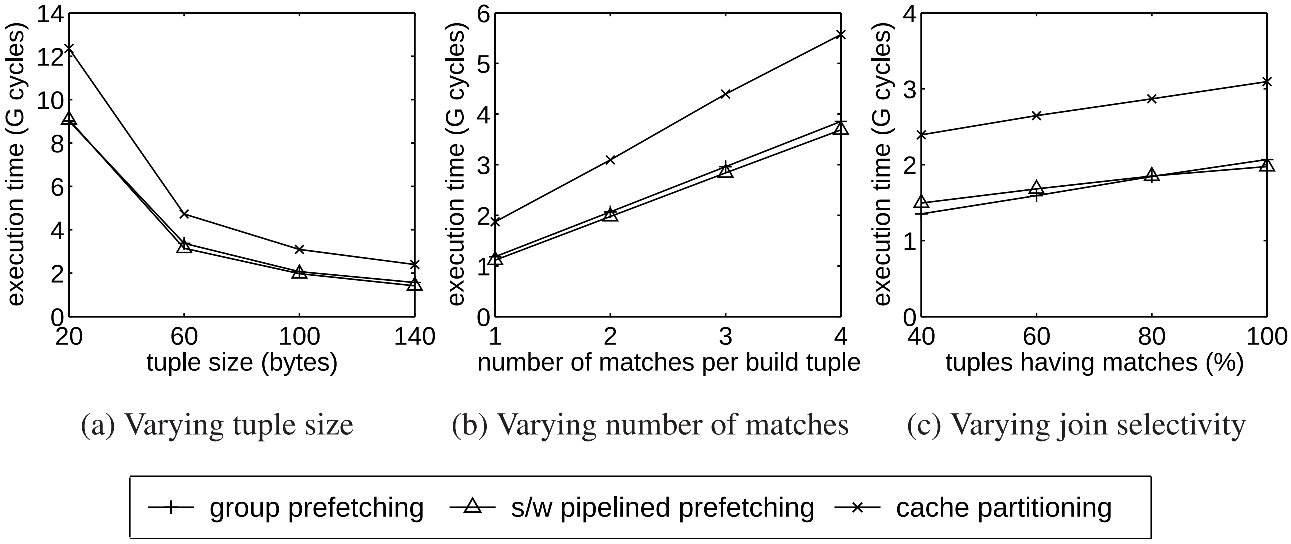


Figure 4.18: Impact of cache flushing on the different techniques.

Group prefetching and software-pipelined prefetching successfully hide most of the data cache miss latencies. Similar to Figure 4.12, the busy portion of the group prefetching bar is larger than that of the GRACE bar, and the busy portion of the software-pipelined prefetching bar is even larger, showing the instruction overhead of the prefetching schemes.

4.7.5 Comparison with Cache Partitioning

Robustness. Cache partitioning assumes exclusive use of the cache, which is unlikely to be valid in a dynamic environment with multiple concurrent activities. Although a smaller "effective" cache size can be used, cache conflicts may still be a big problem and cause poor performance. Figure 4.18 shows the performance degradation of all the schemes when the cache is periodically flushed, which is the worst case interference. We vary the period to flush the cache from 2 ms to 10 ms. A reported point of an algorithm is the execution time of the algorithm with cache flush over the execution time of the same algorithm without cache flush. Therefore, "100" corresponds to the join phase execution time when there is no cache flush. As shown in Figure 4.18, cache partitioning suffers from 11-78% performance degradation. Although the figure shows the worst-case cache interference, it certainly reflects the robustness problem of cache partitioning. In contrast, our prefetching schemes do not assume hash tables and build partitions remain in the cache. As shown in the figure, they are very robust against even frequent cache flushes.



(a) Varying tuple size (b) Varying number of matches (c) Varying join selectivity

Figure 4.19: Re-partitioning cost of cache partitioning. (Default parameters: 200 MB build relation, 400 MB probe relation, 100 B tuples, every build tuple matches two probe tuples.)

Re-partitioning Cost. The number of I/O partitions is upper bounded by the available memory of the partition phase and by the requirements of the storage manager. Experiences with the IBM DB2 have shown that storage managers can handle only hundreds of active partitions per hash join [59]. Given a 2 MB CPU cache and (optimistically) 1000 partitions, the maximum relation size that can be handled through a single partition pass to generate cache-sized partitions is 2 GB. Beyond this limit, it is necessary to employ an additional partition pass to produce cache-sized partitions. We study this re-partitioning cost with several sets of experiments as shown in Figures 4.19(a)-(c).

Figures 4.19(a)-(c) compare the join phase performance of our prefetching schemes with cache partitioning. Note that the re-partitioning step is usually performed immediately before the join phase in main memory, and therefore we can regard it as a preprocessing step in the join phase. Moreover, we employ simple prefetching in the join phase to enhance the cache partitioning scheme wherever possible.

Figure 4.19(a) shows the join phase execution times of joining a 200 MB build relation with a 400 MB probe relation through simulations. Every build tuple matches two probe tuples. We increase the tuple size from 20 bytes to 140 bytes, which results in decreasing numbers of tuples in the relations and therefore the downward trends of the curves. Figure 4.19(b) varies the number of matches per build tuple from 1 match to 4 matches for the 100-byte experiments in Figure 4.19(a). Figure 4.19(c) varies the percentage of build and probe tuples having matches from 100% to 40%. The "100%" points correspond to the 100-byte points in Figure 4.19(a). As shown in the figures, the re-partitioning overhead makes cache partitioning 36–77% slower than the prefetching schemes. Therefore, we conclude that the re-partitioning step significantly slows down cache partitioning compared to group prefetching and software-pipelined prefetching.

4.7.6 User Mode CPU Cache Performance on an Itanium 2 Machine

In this subsection, we present our experimental results for hash join cache performance on the Itanium 2 machine. We first determine the compiler and optimization levels to use for our hash join experiments. Figure 4.20 shows the execution times of joining a 50 MB build partition and a 100 MB probe partition in memory for all the schemes compiled with different compilers and optimization flags. The tuples are 100 bytes, and every build tuple matches two probe tuples. The group prefetching and software-pipelined prefetching bars show the execution times with the optimal parameters tuned for the particular compiler and optimization levels.

From Figure 4.20, we can see that executables generated by icc are significantly faster than those generated by gcc. Moreover, the two optimization levels of icc achieve similar performance. Because the best performance of all schemes is achieved with "icc -O3", we choose "icc -O3" to compile our code in the experiments that follow. Note that "icc -O3" automatically inserts prefetches into the generated executables, thus enhancing the GRACE and cache partitioning schemes. Our prefetching schemes achieve significantly better performance even compared with the compiler-enhanced GRACE and cache partitioning schemes.

Join Phase Performance. Figures 4.21(a)-(c) show the join phase user mode cache performance of all the schemes while varying the tuple size, the ratio of probe relation size to build relation size, and the percentage of tuples that have matches. These experiments correspond to Figures 4.11(a)-(c) in the simulation study. In order to perform cache partitioning, we relax the limitation of 50 MB available memory,

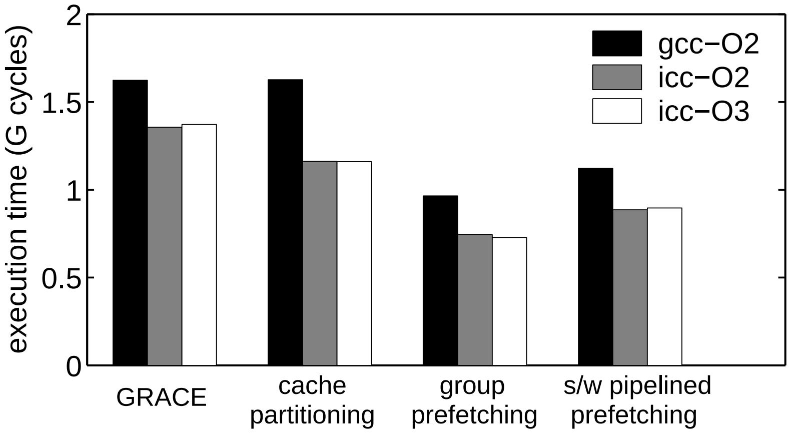


Figure 4.20: Choosing the compiler and optimization levels for hash join cache performance study on the Itanium 2 machine.

and allocate more memory to hold the probe partition as well as the build partition in memory. However, even with this favorable treatment for cache partitioning, our prefetching schemes are still significantly better. As shown in Figures 4.21(a)-(c), group prefetching and software-pipelined prefetching achieve 1.65-2.18X and 1.29-1.69X speedups over the GRACE hash join. Compared to cache partitioning, group prefetching and software-pipelined prefetching achieve 1.52-1.89X and 1.18-1.47X speedups.

Comparing Figures 4.21(a)-(c) and Figures 4.11(a)-(c) in the simulation study, we can see a major difference: Software-pipelined prefetching is significantly worse than group prefetching. A possible reason is that the instruction overhead of implementing software-pipelined prefetching shows up. To verify this point, we compare the number of retired instructions of all the schemes in Figure 4.22. Clearly, both group prefetching and software-pipelined prefetching execute more instructions for code transformation and prefetching than GRACE hash join. Software-pipelined prefetching incurs more instruction overhead, executing 12–15% more instructions than group prefetching. Moreover, cache partitioning executes 43–53% more instructions than GRACE hash join because of the additional partitioning step.

Figures 4.23(a) and (b) show the relationship between the cache performance and the parameters of our prefetching algorithms. We perform the same experiment as in Figure 4.21(a) when tuples are 100 bytes. The optimal values for probing are $G = 14$ and $D = 1$. These values are used in all the experiments

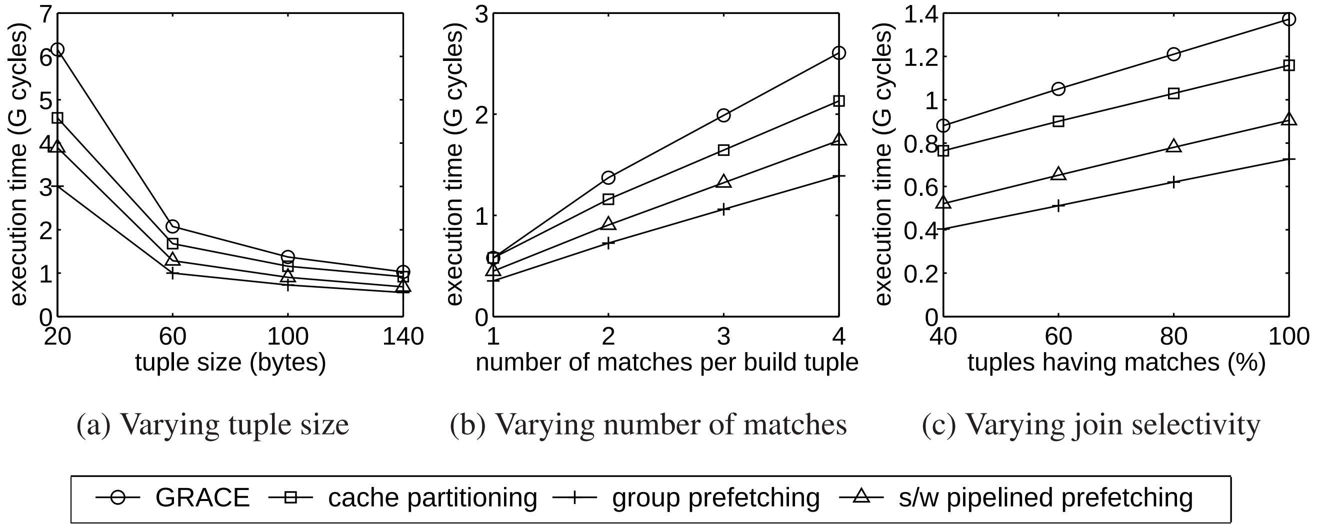


Figure 4.21: Join phase cache performance on Itanium 2.

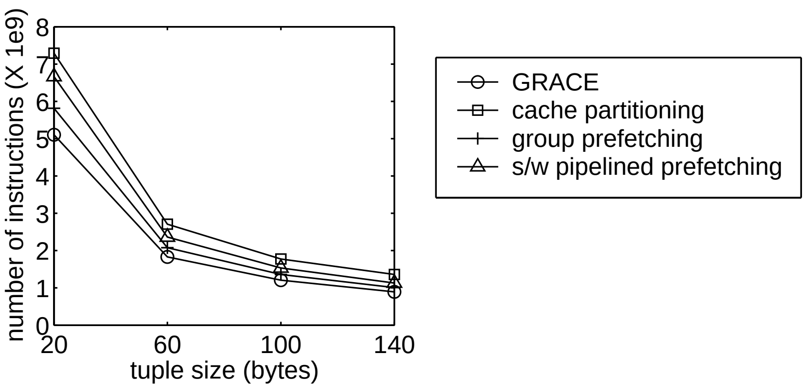
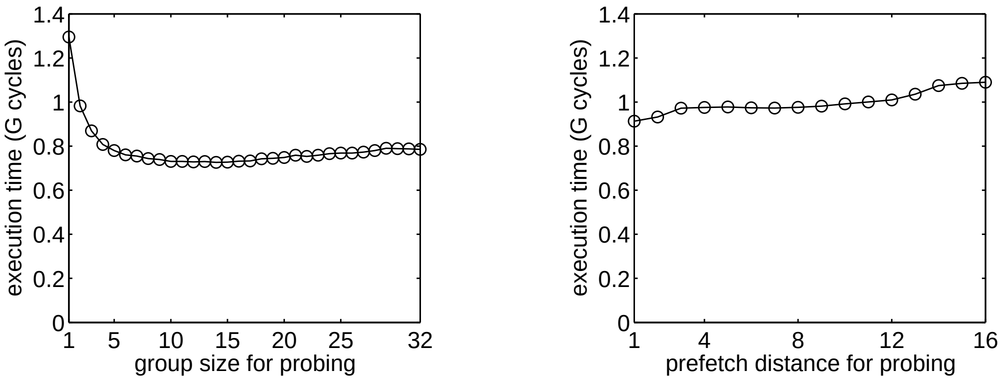


Figure 4.22: Number of retired IA64 instructions for Figure 4.21(a).

shown in Figure 4.21. Similar to what we find in simulation study, we can see large flat segments for both tuning curves, demonstrating the robustness of our prefetching algorithm against parameter choices.

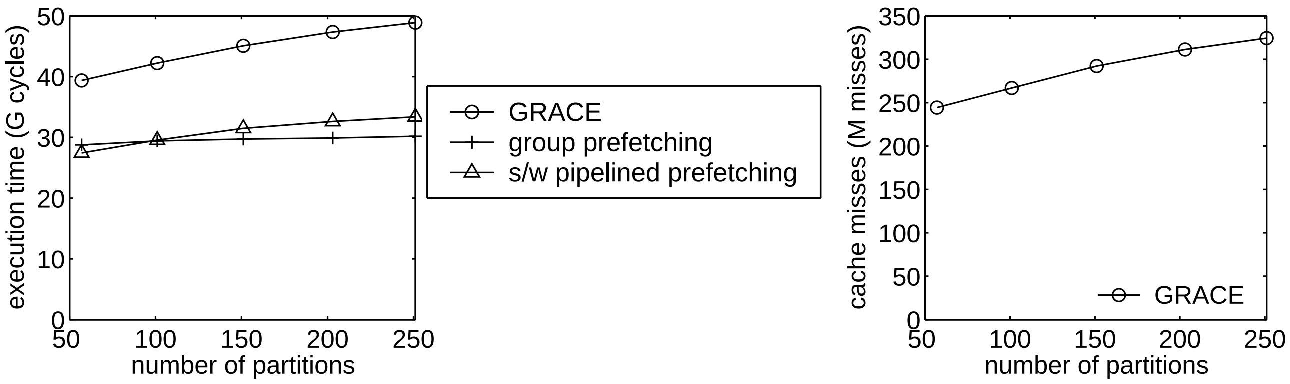
Partition Phase Performance. Figure 4.24(a) shows the user mode execution times of partitioning a 2 GB build relation and a 4 GB probe relation into 57, 100, 150, 200, and 250 partitions.[10] The tuple size is 100 bytes. We see that the GRACE hash join degrades significantly as the number of partitions

[10] 57 is selected to ensure that partitions fit in main memory. The others are chosen arbitrarily for understanding the effects of larger number of partitions.



(a) Group prefetching (b) Software-pipelined prefetching

Figure 4.23: Tuning parameters of group and software-pipelined prefetching for hash table probing in the join phase on the Itanium 2 machine.



(a) Varying number of partitions (b) L3 cache misses

Figure 4.24: Partition phase performance on the Itanium 2 machine.

increases. As shown in Figure 4.24(b), this degradation is mainly resulted from the increasing number of L3 cache misses. Note that the number of memory references per tuple does not change. However, because the number of output buffers increases, the memory references are more likely to miss the CPU cache. Automatically inserted prefetches by the icc compiler do not solve the problem. In contrast, our prefetching algorithms exploit the inter-tuple parallelism to overlap cache misses across the processing of multiple tuples. The performance of our schemes almost stays the same. Compared to the GRACE join, group prefetching and software-pipelined prefetching achieve 1.37-1.62X and 1.43-1.46X speedups.

4.7.7 Execution Times on the Itanium 2 Machine with Disk I/Os

In this subsection, we study the impact of our cache prefetching techniques on the elapsed real times of hash join operations running on the Itanium 2 machine with disk I/Os. We perform the same set of experiments as in Section 4.7.2 (joining a 2GB build relation and a 4GB probe relation) while varying the tuple size and the number of intermediate partitions. We use seven disks in these experiments.[11]

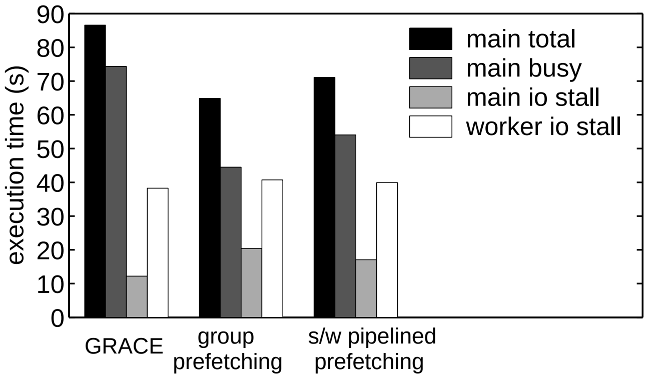
Figures 4.25-4.27 compare our two cache prefetching techniques with the GRACE hash join. The figures show a group of four bars for each experiment. These bars correspond to the four curves described previously in Section 4.7.2. The points for the GRACE join with 100B tuples correspond to the seven disk points shown previously in Figure 4.10. Note that in Figure 4.27, the numbers of partitions, 57 and 113, are chosen automatically by the hash join algorithm (as described earlier in Section 4.7.1) so that a build partition and its hash table consume up to 50 MB of main memory in the join phase. We also measure the performance of generating 250 partitions to better understand the results.

As shown in Figure 4.25-4.27, our group prefetching scheme achieves 1.12-1.84X speedups for the join phase and 1.06-1.60X speedups for the partition phase over the GRACE join algorithm. Our software-pipelined prefetching achieves 1.12-1.45X speedups for the join phase and 1.06-1.51X speedups for the partition phase.

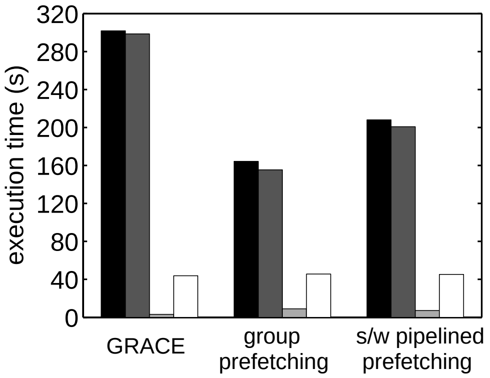
For the experiments, we break down the times in the same way as in Section 4.7.2. Comparing the three groups of bars in each figure, we see that as expected, the *worker io stall* times stay roughly the same, while our cache prefetching techniques successfully reduce the *main busy* times, thus leading to the reduction of the elapsed real times. Note that our implementation of the buffer pool manager is straightforward and without extensive performance tuning. As a result, in some experiments the *main io stall* times increase rather than staying the same, partially offsetting the benefits of reduced *main busy* time. Despite using this relatively simple buffer manager implementation, however, our cache prefetching techniques still achieve non-trivial performance gains.

Comparing the overall speedups as the tuple sizes and the numbers of partitions vary, we see that the

[11] The eighth disk contains the root partition and swap partition. We find that using seven disks instead of eight reduces the variance of the measurements.

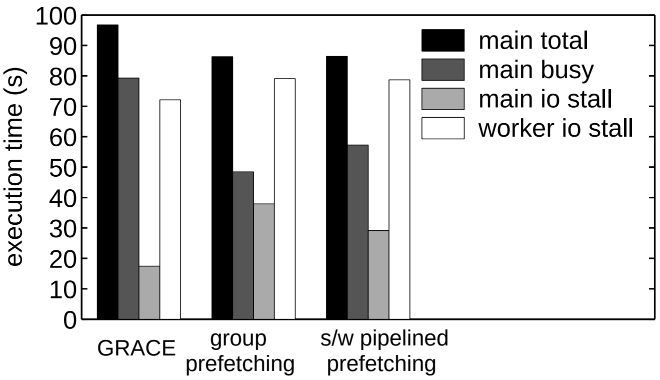


(a) 100B tuples (group prefetching: 1.33X speedup, s/w pipelined prefetching: 1.22X speedup)

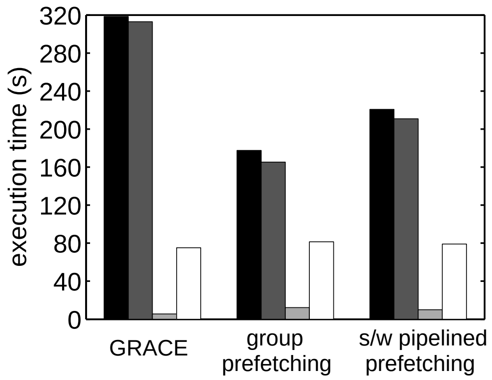


(b) 20B tuples (group prefetching: 1.84X speedup, s/w pipelined prefetching: 1.45X speedup)

Figure 4.25: Join phase performance with I/Os when output tuples are consumed in main memory.

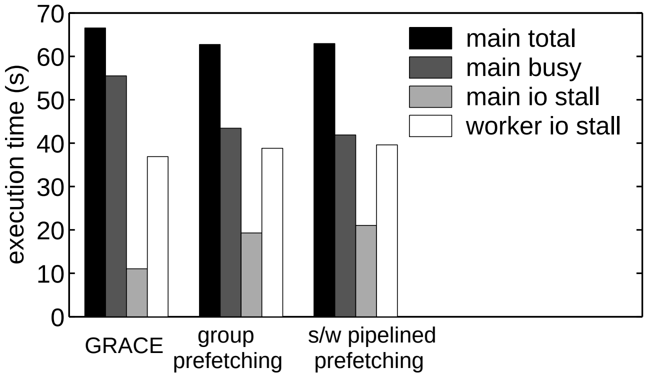


(a) 100B tuples (group prefetching: 1.12X speedup, s/w pipelined prefetching: 1.12X speedup)

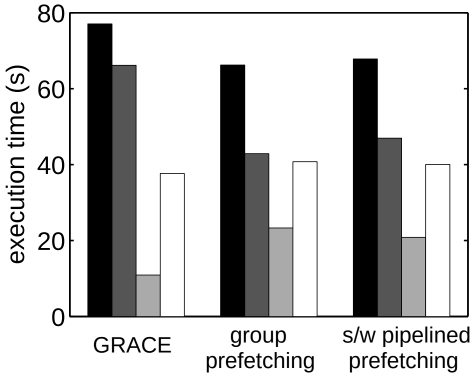


(b) 20B tuples (group prefetching: 1.79X speedup, s/w pipelined prefetching: 1.44X speedup)

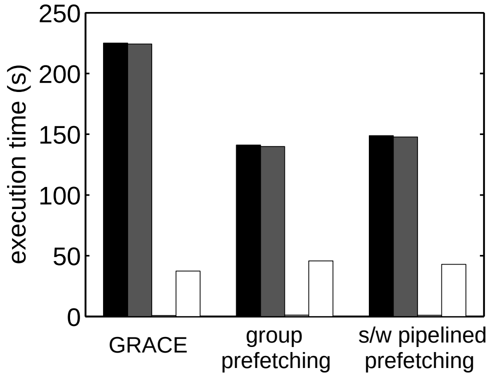
Figure 4.26: Join phase performance with I/Os when output tuples are written to disk.



(a) 100B tuples, 57 partitions (group prefetching: 1.06X speedup, s/w pipelined prefetch: 1.06X speedup)



(b) 100B tuples, 250 partitions (group prefetching: 1.16X speedup, s/w pipelined prefetching: 1.14X speedup)



(c) 20B tuples, 113 partitions (group prefetching: 1.60X speedup, s/w pipelined prefetching: 1.51X speedup)

Figure 4.27: I/O partitioning phase performance with I/Os.

speedups are larger for the join operations that use 20B tuples or produce larger number of partitions. This is because hash joins are more CPU-intensive in these situations. Compared with 100B tuples, there are roughly five times as many 20B tuples to be processed per disk page by hash joins. Larger numbers of partitions require more output buffer space in the I/O partitioning phase, thus incurring more cache misses. Hence as we see in Figures 4.25-4.27, the gap between the *main busy* time and the *worker io stall* time is larger in these experiments, thus leading to a larger potential benefit for CPU cache optimizations.

In summary, we observe that our cache prefetching techniques successfully reduce the elapsed real times of hash joins on the Itanium 2 machine with disk I/Os.

4.8 Chapter Summary

While prefetching is a promising technique for improving CPU cache performance, applying it to the hash join algorithm is not straightforward due to the dependencies within the processing of a single tuple and the randomness of hashing. In this chapter, we have explored the potential for exploiting *inter-tuple* parallelism to schedule prefetches effectively.

Our prefetching techniques—*group prefetching* and *software-pipelined prefetching*—systematically reorder the memory references of hash joins and schedule prefetches so that cache miss latencies in the processing of a tuple can be overlapped with computation and miss latencies of other tuples. We developed generalized models to better understand these techniques and successfully overcame the complexities involved with prefetching the hash join algorithm. Our experimental results both through simulations and on an Itanium 2 machine demonstrate:

- Compared with GRACE and simple prefetching approaches, our cache prefetching techniques achieve 1.29-4.04X speedups for the join phase cache performance and 1.37-3.49X speedups for the partition phase cache performance. Comparing the elapsed real times when I/Os are in the picture, our cache prefetching techniques achieve 1.12-1.84X speedups for the join phase and 1.06-1.60X speedups for the partition phase.

- Compared with cache partitioning, our cache prefetching schemes do not suffer from the large re-partitioning cost, which makes cache partitioning 36-77% slower than our schemes.

- Unlike cache partitioning, our cache prefetching schemes are robust against even the worst-case cache interference.

- Our prefetching schemes achieve good performance for a large varieties of joining conditions.

- The techniques will still be effective even when the latency gap between processors and memory increases significantly in the future (e.g., by a factor of four).

In summary, our group prefetching and software-pipelined prefetching techniques can effectively improve the CPU cache performance of the join phase and the partition phase of hash joins. Moreover, we believe that our techniques can improve other hash-based algorithms such as hash-based group-by and aggregation algorithms, and other algorithms that have inter-element parallelism.

Chapter 5

Inspector Joins

5.1 Introduction

Our ability to minimize the execution time of queries often depends upon the quality of the information we have about the underlying data and the existence of suitable indices on that data. Thus, database management systems (DBMS) maintain various statistics and indices on each relation, which fuel all of the optimizer's decisions. Because it is not feasible to maintain statistics and indices specific to every query, the DBMS must rely on *general* statistics and indices on the relations in order to optimize and process *specific* queries, often resulting in incorrect decisions and ineffective access methods. This problem is particularly acute for join queries, where (i) characteristics of the join result often must be inferred from statistics on the individual input relations and (ii) it is impractical to maintain indices suitable for all join query and predicate combinations. In this chapter, we address this problem in the context of hash joins, one of the most frequent join algorithms.

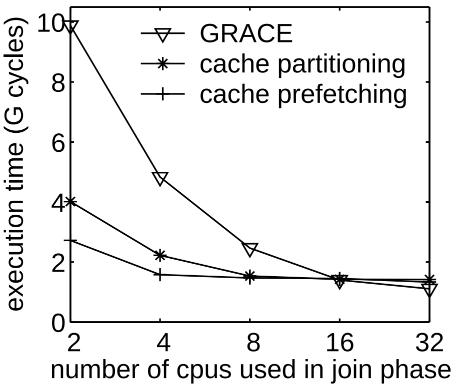
Our key observation is that because hash-based join algorithms visit all the data in the I/O partitioning phase before they produce their first output tuple, we have the opportunity to *inspect* the data during this earlier pass and then use this knowledge to optimize the subsequent join phase of the algorithm. In particular, we show how statistics and specialized indices, *specific to the given query on the given data*, can be used to significantly reduce the primary performance bottleneck in hash joins, namely, the poor CPU cache performance caused by the random memory accesses when building and probing hash tables [14, 16, 66, 90]. We show that our approach in this chapter, which we call *Inspector Joins*, matches or exceeds the performance of state-of-the-art cache-friendly hash join algorithms, including cache partitioning [14, 66, 90], and cache prefetching algorithms that we propose and evaluate in Chapter 4. Inspector joins are specially targeted at shared-bus multi-processor systems. When run on eight or more processors, Inspector Joins offer significant (1.09–1.75X) speedups over the previous algorithms. Moreover, the specialized indices created by inspector joins are particularly well-suited to two common join scenarios: foreign key joins and joins between two nearly-sorted relations.[1]

5.1.1 Previous Cache-Friendly Approaches

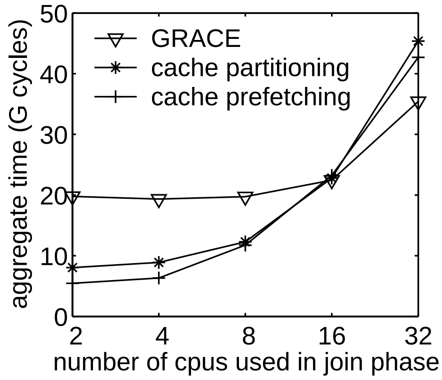
There are two state-of-the-art cache friendly approaches for hash joins: cache partitioning and cache prefetching. Given a pair of build and probe partitions in the join phase, cache partitioning [14, 66, 90] recursively divides the two memory-sized partitions into cache-sized sub-partitions so that a build sub-partition and its hash table fit into the CPU cache, thus reducing the number of cache misses caused by hash table visits. However, the re-partition cost is so significant that cache partitioning is up to 89% worse than cache prefetching for foreign key joins, as described in Chapter 4. Moreover, cache partitioning is sensitive to cache interference by other concurrent activities in the system because it assumes exclusive use of the cache. Cache prefetching exploits memory system parallelism in today's processors and uses software prefetch instructions to overlap cache misses with computation. The cache prefetching techniques are effective only when there is sufficient memory bandwidth. However, modern database servers typically run on multiprocessor systems. In an SMP (symmetric multiprocessing) system, the entire memory bandwidth is shared across all the processors. Because cache prefetching essentially trades off bandwidth for reduced execution time, its benefit gradually disappears as more and more processors eagerly compete for the limited memory bandwidth.

Figure 5.1 shows the join phase performance of joining a 500MB build relation with a 2GB probe

[1] Joins between nearly-sorted relations arise, for example, in the TPC-H benchmark, where the lineitem table and the orders table are nearly sorted on the (joining) order keys. We also observe joins between nearly-sorted relations in a commercial workload.



(a) Execution time varying the number of CPUs



(b) Aggregate time of all CPUs used in join phase

Figure 5.1: Impact of memory bandwidth sharing on join phase performance in an SMP system.

relation, varying the number of CPUs used in the join phase. In each experiment, the number of I/O partitions generated is a multiple of the number of CPUs. Then the same join phase algorithm is run on every CPU to process different partitions in parallel. (Please see Section 5.6.1 for setup details.) As shown in Figure 5.1(a), both cache partitioning and cache prefetching perform significantly better than the original GRACE hash join. Cache partitioning is worse than cache prefetching because of the re-partition cost. The effect of memory bandwidth sharing is more clearly shown in Figure 5.1(b), which reports the total aggregate time of all CPUs for the join phase for the same experiment. We can see that the benefit of cache prefetching gradually disappears as more and more processors are competing for the memory bandwidth. Cache prefetching becomes even worse than the GRACE hash join when there are 16 processors or more. Interestingly, the GRACE hash join also suffers from memory bandwidth sharing when there are 32 processors.

5.1.2 The Inspector Join Approach

To achieve good performance even when memory bandwidth is limited, we need to reduce the *number* of cache misses of the join phase algorithm, in addition to applying prefetching techniques to hide cache miss latencies. Our approach exploits the multi-pass structure of the hash join algorithm. During the I/O partitioning phase, inspector joins create a special multi-filter-based index with little overhead; this index will enable us to have “in-place” cache-sized sub-partitions of the build table. Unlike cache partitioning our approach reduces the number of cache misses without moving tuples around. The join phase, which we refer to as a *cache-stationary join phase* because of its in-place nature, is performed using the index.

Our cache-stationary join phase is specially designed for joins with nearly unique build join keys, which include primary-foreign key joins, the majority of all the real-world joins. On the other hand, if probe tuples frequently match multiple build tuples in a given join query, the cache-stationary join phase is not the best choice. An inspector join can detect this condition during its inspection and switch to use a different join phase algorithm (see Section 5.6.5 for details). Moreover, as mentioned above, inspector joins can detect nearly-sorted relations (after any predicates being applied before the join). Our initial intuition was that a sort-merge based join phase algorithm should be applied in this case. However, surprisingly, the cache-stationary join phase performs equally well, due to the effectiveness of its multi-filter-based index.

The chapter is organized as follows. Section 5.2 discusses related work. Section 5.3 illustrates the high level ideas in our solution. Section 5.4 and 5.5 describe our algorithms in detail. Section 5.6 presents our experimental results. Finally, Section 5.7 summarizes the chapter.

5.2 Related Work

Hash join cache performance. Hash join has been studied extensively over the past two decades [29, 54, 59, 89]. Recent studies focus on the CPU cache performance of hash joins. Shatdal *et al.* show that cache partitioning achieves 6-10% improvement for joining memory-resident relations with 100B tuples [90]. Boncz, Manegold and Kersten propose using multiple passes in cache partitioning to avoid cache and TLB thrashing when joining vertically-partitioned relations (essentially joining two 8B columns) [14, 66]. However, we show in Chapter 4 that when the tuple size is 20B or larger, the re-partition cost of cache partitioning is so significant that cache partitioning is up to 89% worse than cache prefetching. In Chapter 4, we exploit the inter-tuple parallelism to overlap the cache misses of a tuple with the processing of multiple tuples. We propose and evaluate two prefetching algorithms, group prefetching and software-pipelined prefetching. However, as shown in Figure 5.1, the performance of cache prefetching degrades significantly when more and more CPUs are eagerly competing for the memory bandwidth in a multiprocessor system. Therefore, in this chapter, we exploit information collected in the I/O partitioning phase to fit address-range-based sub-partitions in cache, thus reducing the number of cache misses without incurring additional copying cost. Our approach is effective for tuples that are 20B or more. For smaller tuples, we revert to cache partitioning.

Inspection concept. Several studies exploit information collected while processing queries previously submitted to the DBMS: reusing partial query results in multi-query optimization [86], maintaining and using materialized views [10], creating and using join indices [100], and collecting up-to-date statistics for future query optimizations [94]. Unlike these studies the inspection and use of the information in our approach are specific to a single query. Therefore, we avoid the complexities of deciding what information to keep and how to reuse data across multiple related queries. Moreover, our approach is effective for any join query and predicate combinations.

Dynamic re-optimization techniques augment query plans with special operators that collect statistics about the actual data during the execution of a query [49, 68]. If the operator detects that the actual statistics deviate considerably from the optimizer's estimates, the current execution plan is stopped and a new plan is used for the remainder of the query. Compared to the *global* re-optimization of query plans, our inspection approach can be regarded as a *complementary, local* optimization technique inside the hash join operator. When hash joins are used in the execution plan, our inspection approach creates specialized indices to enable the novel cache-stationary optimization and allows informed choice of join phase algorithms. Because the indices and informed choice account for which tuples will actually join as well as their physical layout within the intermediate partitions, this functionality cannot be achieved by operators *outside* the join operator.

5.3 Inspector Joins: Overview

In this section, we describe (i) how we create the multi-filters as a result of data inspection, (ii) how we minimize the number of cache misses without moving any tuples around, (iii) how we exploit cache

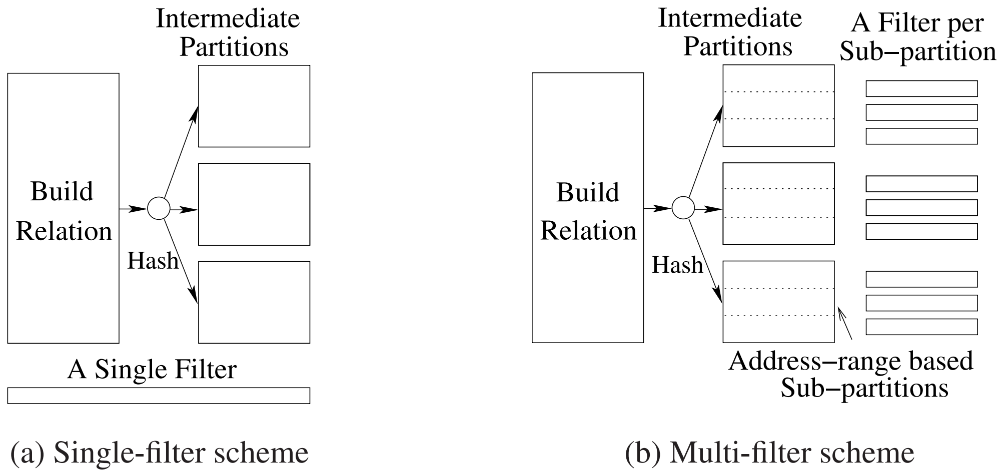


Figure 5.2: Using multiple filters to inspect the data.

prefetching to hide the remaining cache misses and to improve robustness against cache interference, and (iv) how we choose join phase algorithms based on obtained information about the data.

5.3.1 Inspecting the Data: Multi-Filters

While partitioning tables for a hash join, commercial database systems often construct a filter to quickly discard probe tuples that do not have any matches [59]. Such filters may improve join performance significantly when a large number of probe tuples do not match any tuples in the build relation (e.g., there is a predicate on the build relation in a foreign-key join). As shown in Figure 5.2(a), a single filter is computed from all the build tuples to approximately represent all the join attribute values in the build relation. Testing a value against the filter is conservative: While a negative answer means that the value is not in the filter, false positives may occur with a low probability. (Bloom filters, detailed in Section 5.4, are a typical example.) When partitioning the probe relation, the algorithm tests every probe tuple against the filter. If the test result is negative for a tuple, the algorithm simply drops the tuple, thus saving the cost of writing it to disk and processing it in the join phase.

Instead of using a single large filter that represents the entire build relation, the inspector join creates multiple shorter filters (illustrated in Figure 5.2(b)), each representing a disjoint subset of build tuples.

Testing a probe tuple against the filters will (conservatively) show which subsets the probe tuple has matches in. The build relation subsets are address-range-based sub-partitions; that is, a subset represents all build tuples in K consecutive pages in a build partition. K is chosen to make the sub-partition fit in the cache in the join phase, as will be described in Section 5.3.2.

The inspector join builds the set of small filters by inspecting the build relation during the partitioning phase. To keep track of the sub-partition boundaries, we use a page counter for every partition. Then, every build tuple is used to compute the filter corresponding to the sub-partition the tuple belongs to. Note that the multi-filter scheme tests filters differently than the single-filter scheme. For every probe tuple, after computing its destination partition, the algorithm checks the join attribute value in the tuple against *all* the filters in the partition. The algorithm drops the probe tuple only if *all* filter tests for all sub-partitions are negative. The positive tests show which sub-partition(s) may contain matching build tuples of the probe tuple, and this information is used in the join phase of the inspector algorithm. Section 5.4 demonstrates that our multi-filter scheme incurs the same number of cache misses as the single-filter scheme during the inspection and filter-construction phase, and it can achieve the same aggregate false positive rate with moderate memory space requirements.

5.3.2 Improving Locality for Stationary Tuples

During the join phase, the inspector join algorithm knows which probe tuples match every address-range-based sub-partition of the build relation, and therefore processes tuples one sub-partition at a time. For every sub-partition, the algorithm builds a cache-resident hash table on the build tuples, and probes it with all the probe tuples associated with this sub-partition. We ensure that the build tuples of a sub-partition and its hash table fit into the cache by choosing the number of pages per build sub-partition, K, as follows:

$$K \cdot P + K \cdot n \cdot H \leq C \tag{5.1}$$

The variables used above and throughout the chapter are summarized in Table 5.1.

[2]Note that we usually set C to be a fraction (e.g., 0.5) of the total cache size so that call stacks and other frequently used data structures can stay in the cache as well.

Table 5.1: Terminology used throughout Chapter 5.

Name	Definition
P	page size (in bytes)
N	number of build tuples in build relation
n	number of build tuples per partition page
H	number of bytes in hash table for every key
C	effective cache size (in bytes)[2]
L	cache line size (in bytes)
K	number of build pages per sub-partition
S	number of sub-partitions per build partition
bpk	number of bits per key for a single Bloom filter
fpr	Bloom filter false positive rate

Figure 5.3 compares the cache behaviors of all the join-phase algorithms that we are considering. As shown in Figure 5.3(a), the GRACE algorithm joins memory-sized partitions. It builds an in-memory hash table on all the build tuples, then probes this hash table using every tuple in the probe partition to find matches. Because of the inherent randomness of hashing, accesses to the hash table have little temporal or spatial locality. Since the build partition and its hash table are typically much larger than the CPU cache size, these random accesses often incur expensive cache misses, resulting in poor CPU cache performance.

Figure 5.3(a) shows that the cache prefetching algorithms perform the join in the same way as the GRACE algorithm [54]. The prefetching algorithms do not reduce the number of cache misses; rather, they use prefetch instructions to hide the latency of cache misses when repeatedly visiting the hash table. To achieve this, they rely on sufficient memory bandwidth to quickly service cache misses and prefetch requests. When multiple processors aggressively compete for the limited main memory bandwidth, however, the performance of the prefetching algorithms is likely to degrade significantly, as shown previously in Figure 5.1.

Figure 5.3(b) illustrates how cache partitioning joins pairs of cache-sized partitions to avoid excessive cache misses because of hash table visits. The algorithm recursively partitions memory-sized partitions

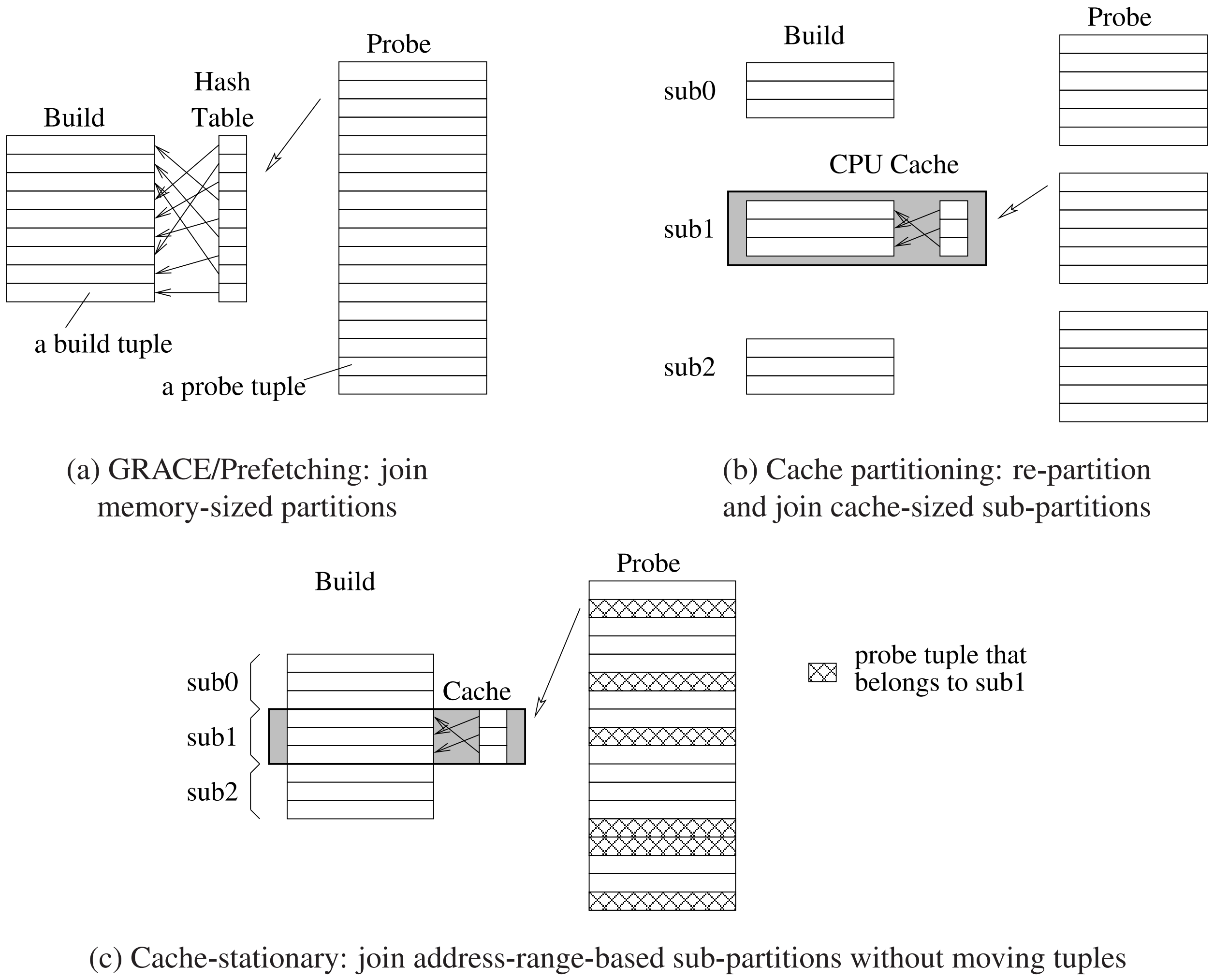


Figure 5.3: Comparing the cache behaviors of different join phase algorithms.

into cache-sized sub-partitions, and then joins the sub-partitions using cache-resident hash tables. Essentially, cache partitioning visits every tuple at least once more than the GRACE algorithm, thereby incurring significant re-partition cost, as shown previously in Figure 5.1.

Figure 5.3(c) shows the cache behavior of the cache-stationary join phase of inspector join. It reads memory-sized partitions into memory and processes one cache-sized partition at a time, avoiding the cache misses caused by hash table visits. It simply visits the consecutive pages of a build sub-partition to build a hash table. Random memory accesses are avoided while building the hash table because the build sub-partition and the hash table fit into the cache. Since the algorithm already knows which probe tuples may have matches in the given build sub-partition, it can directly visit these probe tuples in place without moving them. Compared to cache partitioning, inspector joins eliminate unnecessary cache misses without moving any tuples, thereby avoid the excessive re-partitioning overhead. The algorithm almost never revisits probe tuples when join attribute values in the build relation are unique (or almost unique). Values in the build relation are unique, for instance, in foreign-key joins, which constitute most of the real-world joins. (As detailed below, the inspector join verifies the assumption and selects one of the other join phase algorithms when the assumption does not hold.) Moreover, the algorithm utilizes cache prefetching to further hide the latency for the probe tuples, as we describe below.

5.3.3 Exploiting Cache Prefetching

We exploit cache prefetching techniques in addition to using cache-sized sub-partitions for two reasons. First, cache prefetching can hide the latency of the remaining cache misses, such as the cold cache misses that bring a build sub-partition and its hash table into the CPU cache, and the cache misses for accessing the probe tuples. Second, cache prefetching can improve the robustness of our algorithm when there is interference with other processes running concurrently in the system. As shown previously in Section 4.7.5, cache partitioning performance degrades significantly when the CPU cache is flushed every 2-10 ms, which is comparable to typical thread scheduling time. To cope with this problem, we issue prefetch instructions as a safety net for important data items that should be kept in the cache, such as the build tuples in a build sub-partition. If the data item is in cache, there is no noticeable penalty. On the other hand, if the data item has been evicted from cache, the prefetch instruction brings it back into the cache significantly earlier, making this approach worthwhile. In a sense, we use double measures to maximize cache performance when accessing important data items.

5.3.4 Choosing the Best Join Phase Algorithm

Based on the statistics collected from the actual data in the partition and inspection phase, inspector joins can choose the join phase algorithm best suited to the given query. For example, we detect duplicate build keys by counting the number of sub-partitions each probe tuple matches. Since a probe tuple must be tested against all the possible matching sub-partitions for correctness, the execution time of the cache-stationary join phase of the inspector join increases with the number of duplicate build keys. When the number of sub-partitions a probe tuple matches on average is above a threshold, inspector joins select a different join phase algorithm, as will be shown in Section 5.6.5.

Our inspection approach can also detect relations that are nearly-sorted on the join key. Our initial intuition is that a sort-merge based join phase should be applied in this case. To verify our intuition, we implemented an inspection mechanism to detect nearly-sorted tuples. The basic idea is to keep tuples that are out of order in a memory buffer when partitioning an input relation. The input is nearly sorted if the memory buffer does not overflow when all the tuples are read. At this point, all the intermediate partitions contain in-order tuples. We then partition the (small number of) out-of-order tuples and store them separately from the in-order tuples. In the join phase, given four inputs per partition (out-of-order and in-order build and probe inputs), the sort-merge algorithm first sorts the out-of-order inputs and then merges all four inputs to find matching tuples. Surprisingly, we find in our experiments (in Section 5.6.5) that the cache-stationary join phase performs as well as the sort-merge implementation.

5.4 I/O Partition and Inspection Phase

In this section, we begin by introducing a typical filter implementation: Bloom filters. Then, we discuss the memory space requirement of our multi-filter scheme, and we illustrate how our scheme achieves the same number of cache misses as the single-filter scheme. Finally, we describe the I/O partition and inspection algorithm that uses the multi-filter scheme to determine the matching sub-partition information for probe tuples.

5.4.1 Bloom Filters: Background

A Bloom filter represents a set of keys and supports membership tests [11]. As shown in Figure 5.4, a Bloom filter consists of a bit vector and d independent hash functions, $h_0, h_1, \ldots, h_{d-1}$ ($d = 3$ in the

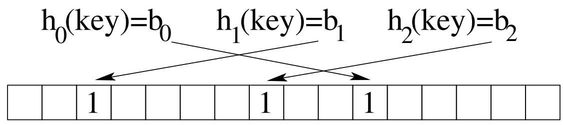


Figure 5.4: A Bloom filter with three hash functions.

Table 5.2: Number of Bloom filter bits per key ($d = 3$).

false positive rate (fpr)	0.1	0.05	0.01	0.005	0.001	0.0005
number of bits per key (bpk)	4.808	6.529	12.364	15.997	28.474	36.277

figure). To represent a set of keys, we first initialize all the bits in the bit vector to be 0. Then, for every key, we compute d bit positions using the hash functions and set the bits to 1 in the bit vector. (A bit may be set multiple times by multiple keys.)

To check whether a test key exists in the set of known keys, we compute d bit positions for the test key using the hash functions and check the bits in the bit vector. If some of the d bits are 0, the set of known keys can not contain the test key. If all of the d bits are 1, the test key may or may not exist in the set of known keys. Therefore, Bloom filter tests may generate false positives but may never generate false negative results.

Intuitively, the larger the Bloom filter vector size, the smaller the probability that a test generates a false positive, which is called the false positive rate. In fact, the false positive rate fpr and the number of bits per key bpk of the bit vector are closely related to each other [11]:

$$fpr \approx (1 - e^{-d/bpk})^d, \text{when bit vector size} \gg 1 \tag{5.2}$$

Table 5.2 shows the bpk values for various fpr. In this chapter, we only consider Bloom filters with $d = 3$.[3] We point out, however, that our algorithm works for any choice of d.

[3]Previous studies computed the optimal d for the purpose of minimizing filter sizes [23, 88, 93]. The computed optimal size (e.g., 11 when $bpk = 16$) can be much larger than 3. However, the major concern in hash join algorithms is to reduce the hashing cost and the number of memory references for setting and testing filters. Therefore, commercial hash join algorithms often choose a small d [59]. Section 5.4.2 shows that the resulting space overhead is still relatively modest.

Table 5.3: Total filter size varying tuple size (1 GB build relation, $fpr = 0.05$, $S = 50$).

tuple size	20B	60B	100B	140B
number of build tuples	50M	16.7M	10M	7.1M
single-filter	40.8MB	13.6MB	8.2MB	5.8MB
multi-filter	178.0MB	59.4MB	35.6MB	25.3MB

5.4.2 Memory Space Requirement

In a single filter scheme, the total size of the filter in bytes can be computed as follows, where N is the total number of build tuples (assuming that keys are unique):

$$total\_filter\_size_{single} = bpk \cdot N/8 \tag{5.3}$$

Our multi-filter scheme constructs a filter per sub-partition in every build partition. However, the filters represent disjoint subsets of build tuples; every build tuple belongs to one and only one sub-partition. Therefore, every build tuple is still represented by a single filter. Let bpk' be the number of bits per key for an individual filter. Then the total filter size of the multi-filter scheme is:

$$total\_filter\_size_{multi} = bpk' \cdot N/8 \tag{5.4}$$

We can quantify the increase in memory space by using the ratio between the filter sizes of the multi-filter and the single-filter schemes:

$$space\_increase\_ratio = \frac{total\_filter\_size_{multi}}{total\_filter\_size_{single}} = \frac{bpk'}{bpk} \tag{5.5}$$

To obtain bpk', we need to first compute the false positive rate fpr' for an individual filter in the multi-filter scheme. Suppose there are S sub-partitions per build partition. Then, a probe tuple will be checked against all the S filters in the partition to which the probe tuple is hashed. If any filter test is positive, the join phase algorithm has to join the probe tuple with the corresponding build sub-partition for matches. In order to keep the number of additional probes caused by false positives the same as the single-filter scheme, the single-filter scheme fpr and the individual fpr' of the multi-filter scheme should satisfy:

$$fpr' = fpr/S \tag{5.6}$$

For example, if the single-filter scheme's fpr is 0.05, we can compute the space increase ratio as follows. Since $fpr' = fpr/S$, $fpr' = 0.001$ if $S = 50$. Then, $bpk = 6.529$ and $bpk' = 28.474$, according to Table 5.2. Therefore, $space\_increase\_ratio$ is 4.4. Similarly, if $S = 100$, we can compute that $space\_increase\_ratio$ is 5.6.

Table 5.3 compares the filter size of the multi-filter scheme with the single-filter scheme when the aggregate false positive rate is 0.05 and there are 50 sub-partitions per partition.[4] The build relation is 1GB large, and we vary the tuple size from 20 to 140 bytes. We can see that the space requirement is moderate when the tuple size is greater than or equal to 100B, which is typical in most real-world applications. Even if the tuple size is as small as 20B, the memory requirement of 178MB can still be satisfied easily in today's database servers.[5]

5.4.3 Minimizing the Number of Cache Misses

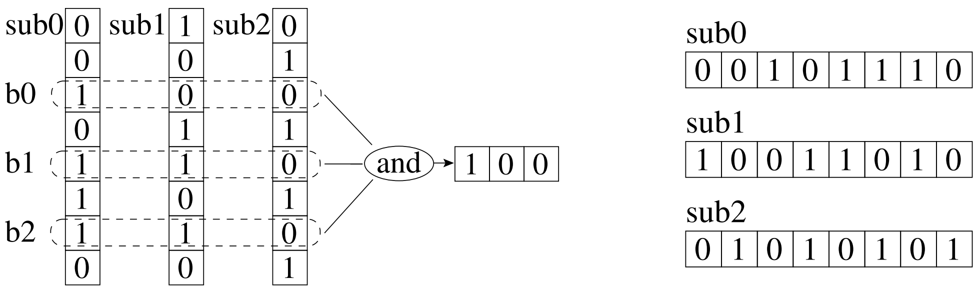
The single-filter scheme writes three bits in the Bloom filter for every build tuple. For every probe tuple, it reads three bits in the Bloom filter. Since the bit positions are random because of the independent hash functions, the single-filter scheme potentially incurs three cache misses for every build tuple and for every probe tuple, assuming the total filter size is larger than the CPU cache size. (We do not use our algorithm if the relation is so small that the computed single filter size is smaller than cache, but the total size of the multiple filters may be larger than cache.)

In the multi-filter scheme, a build tuple is still represented by a single filter corresponding to its sub-partition. Therefore, the multi-filter scheme still writes three bits for every build tuple, incurring the same number of cache misses as the multi-filter scheme.

However, the multi-filter scheme checks S filters for every probe tuple, where S is the number of sub-partitions per partition. We ensure that the filters are of the same size. Given a probe tuple, the

[4] $S = 50$ is a reasonable choice. Even if the cache size is as small as 1MB, and the I/O partition phase can produce up to 500 partitions (limited by the capability of the storage manager), it allows the build relation size to be as large as 25GB.

[5] Hash join may choose to hold intermediate partition pages in memory. Therefore, the above additional memory space requirement may result in extra I/Os. However, hash join is CPU bound with reasonable I/O bandwidth (as shown previously in Chapter 4), and this is a minor effect.



(a) Vertical layout (b) Horizontal layout

Figure 5.5: Layouts of multiple Bloom filters.

multi-filter scheme still computes the same number of bit positions as the single-filter scheme, then it simply checks the same bit positions for all S filters. However, the filter tests may incur $3S$ cache misses, which is much more than the single-filter scheme.

This problem can be solved by laying out the filters vertically for every partition. The idea was proposed previously in the context of searching text databases with bit-sliced signature files on disk [24]. In the following, we employ this idea in testing main-memory-based filters. Moreover, we describe how we get around the problem of creating vertical filters when the number of filters (thus the number of bits per row) is unknown.

As shown in Figure 5.5(a), the bits at the same bit position in all the filters of a partition are consecutive in memory. That is, the first bits of all the filters are stored together, which are followed by the second bits of all the filters, so on so forth. Note that the cache line size is typically 32B to 128B, or 256-1024 bits, which is much larger than the number of filters per partition S. Therefore, under the vertical layout, we can read the bits of a given position from all the filters while incurring only a *single* cache miss. In this way, the multi-filter scheme can keep the number of cache misses the same as the single-filter scheme for testing a probe tuple.

Figure 5.5(a) shows that we can test all the filters for a given probe tuple using a bit operation under the vertical layout. We simply compute a bit-wise AND operation of the b_0 bits, the b_1 bits, and the b_2 bits. A 1 in the result means all three bits for the corresponding filter are 1. Therefore, a 1/0 resulting bit means a positive/negative test result for the corresponding filter.

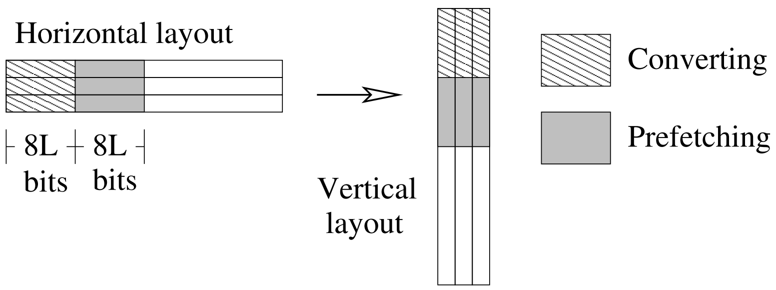


Figure 5.6: Horizontal to vertical layout conversion.

A new problem occurs when we lay out the filters vertically: New filters can not be easily allocated and the number of filters in a partition must be determined before allocating the memory space for the vertical filters. Since the actual partition size may vary due to data skew, using the maximal possible number of sub-partitions may waste a lot of memory space.

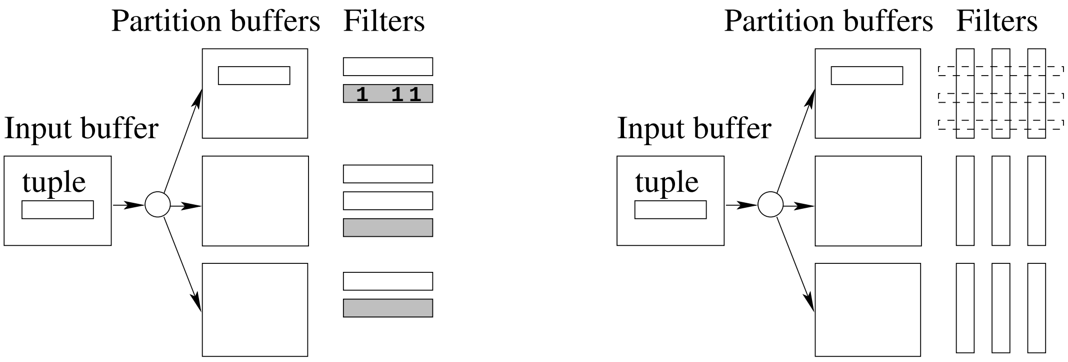
We solve this dynamic allocation problem by using horizontal layout when partitioning the build relations and generating the filters, as shown in Figure 5.5(b). Then, we convert the horizontal layout into an equivalent vertical layout before partitioning the probe relation.

Figure 5.6 illustrates the conversion algorithm. Horizontal filters are allocated at cache line boundaries. We transpose the filters one block at a time. Every block consists of a cache line ($8L$ bits) for all the filters. The source cache lines of different filters in the horizontal layout are not contiguous in memory, while the destination block is a continuous chunk of memory. Every outer-loop iteration of the algorithm prefetches the next source and destination blocks in addition to converting the current block. In this way, we hide most of the cache miss latency of accessing the source and destination filters.

5.4.4 Partition and Inspection Phase Algorithm

The algorithm consists of the following three steps:

1. Partition build relation and compute horizontal filters;
2. Convert horizontal filters to vertical layout;
3. Partition probe relation and test vertical filters.



(a) Partition build relation (b) Partition probe relation

Figure 5.7: I/O partition and inspection phase algorithm.

In the above, we have described the algorithm for step 2. This subsection focuses on the other two steps in the algorithm.

As shown in Figure 5.7(a), step 1 allocates an input buffer for the build relation and an output buffer for every intermediate partition. It uses horizontal filters. Each partition keeps a page counter for the outgoing pages. When the counter equals to K, the number of pages per sub-partition, a new filter is allocated from a memory pool and the counter is reset to 0. For every build tuple, the algorithm extracts the join attribute to compute a 32-bit hash code. It determines the partition number by using the hash code and copies the tuple (with projection if needed) to the output buffer. The algorithm also computes and sets the three bit positions of the current horizontal filter. For better cache performance, we employ group prefetching as described in Chapter 4. The only difference is the addition of prefetching for the Bloom filter positions. Moreover, a tuple's hash code is stored in the page slot area to save hash code computation in the join phase (as described in Chapter 4).[6]

As shown in Figure 5.7(b), Step 3 is similar to Step 1 with the following differences. First, the algorithm tests every probe tuple against the set of vertical filters in the tuple's partition. A tuple is dropped when all the resulting bits are 0. Second, positive results show which sub-partitions may contain matching tuples for the given probe tuple. The sub-partition ID(s) is recorded in the slot area of the same

[6]A build partition page slot consists of a 4B hash code and a 2B tuple offset. Every two slots are combined together to align the hash codes at 4B boundaries.

output page containing the tuple.[7] In most cases, a single sub-partition ID is found. Note that slots may be of variable size now. This is not a problem since the probe slots will only be visited sequentially (in the counting sort step) in the join phase algorithm, as will be described in Section 5.5. Third, the number of probe tuples associated with each sub-partition is counted, which is used (in the counting sort step) in the join phase algorithm.

5.5 Cache-Stationary Join Phase

The join phase algorithm consists of the following steps:

1. Read build and probe partitions into main memory;
2. Extract per-sub-partition probe tuple pointers;
3. Join each pair of build and probe sub-partitions.

By using the sub-partition information collected in the partition and inspection phase, the algorithm achieves good cache performance without copying any tuples. The sub-partition information is stored in the order of probe tuples in the probe intermediate partitions. However, Step 3 visits all the probe tuples of a single sub-partition and then moves on to the next sub-partition. It requires the sub-partition information in the order of sub-partition IDs. Therefore, probe tuple sub-partition information has to be sorted before use. In the following, we first describe how Step 2 performs counting sort, then discuss the use of prefetching to improve performance and robustness in Step 3.

5.5.1 Counting Sort

The algorithm knows the number of sub-partitions and the number of probe tuples associated with each sub-partition; the latter is collected in the I/O partition phase. Therefore, we can use counting sort, which

[7]From high address to low address, a probe partition page slot consists of a 4B hash code, a 2B tuple offset, a 1B number of sub-partitions, a sequence of sub-partition IDs each taking 1B. We align slots on 4B boundaries and a slot takes 8B when there is a single sub-partition ID.

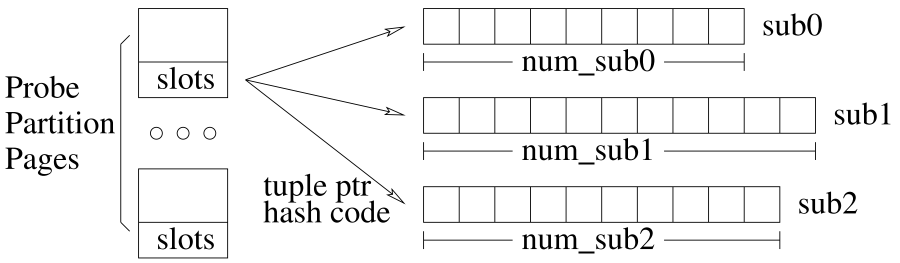


Figure 5.8: Extracting probe tuple information for every sub-partition using counting sort.

is a fast O(N) algorithm, for extracting probe tuple information (the probe tuple pointers and hash codes) for every sub-partition.

As shown in Figure 5.8, for every sub-partition, we allocate an array, whose size is equal to the number of probe tuples associated with the sub-partition. The algorithm visits the slot area of all the probe partition pages sequentially. For every slot, it computes the tuple address using the tuple offset. Then the algorithm copies the tuple address and the hash code to the destination array(s) that are specified by the sub-partition ID(s) recorded in the page slot. Assuming the build join attribute values are mostly unique, there is often a single sub-partition ID for a probe tuple, and the tuple address and hash code are only copied once. After processing all the probe page slots, the algorithm obtains an array of (tuple pointer, hash code) pairs for every sub-partition. Note that the tuples themselves are not visited nor copied in the counting sort.

We use cache prefetching to hide the cache miss latency of reading page slots and writing to the destination arrays. We keep a pointer to the next probe page and issue prefetches for the next page slot area while processing the slot area of the current page. Similarly, for every destination array, we keep a pointer to the next cache line starting address. We issue a prefetch instruction for the next cache line before we start using the current cache line in the array.

5.5.2 Exploiting Prefetching in the Join Step

For every pair of build and probe sub-partitions, the algorithm first constructs a hash table. (We assume the same hash table structure as in Chapter 4.) Since the hash codes are stored in the build page slot area,

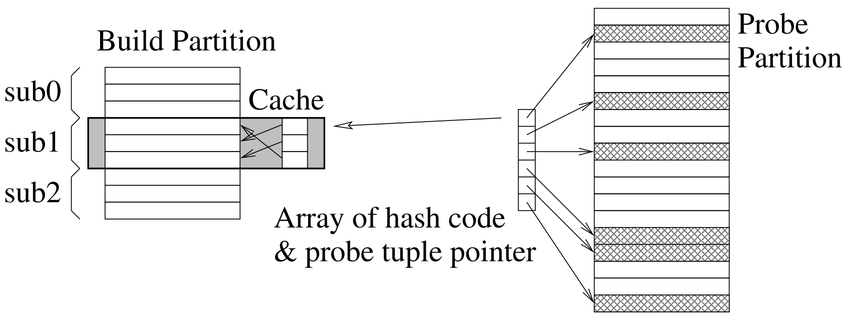


Figure 5.9: Joining a pair of build and probe sub-partitions.

the algorithm does not need to access the actual build tuples for creating the hash table. However, we expect the build tuples to be used frequently during probing. Therefore, we issue prefetch instructions for all build tuples of the sub-partition to bring them into cache, as shown in Figure 5.9.

After building the hash table, the algorithm visits the array containing the probe tuple pointers and hash codes of the sub-partition, as shown in Figure 5.9. It probes the hash table with hash codes in the array. If a probe is successful, the algorithm visits the probe tuple and the build tuple to verify that their join attributes are actually equal. It produces an output tuple for a true match.

We issue prefetches for the probe tuples and for the array containing probe tuple information. Here, we use a special kind of prefetch instruction, non-temporal prefetches, which are supported by Intel Itanium 2 and Pentium 4 architectures [43, 45]. Non-temporal prefetches are used to read cache lines that do not have temporal locality; the cache lines are supposed to be used only once. Therefore, cache lines read by non-temporal prefetches ignore the LRU states in the cache, and they go to a particular location in the corresponding cache set, thus minimizing cache pollution by the prefetched line. Since we aim to keep the build sub-partition and the hash table in cache, minimizing the cache pollution caused by visiting other structures is exactly what we want.

To prefetch the array containing probe tuple pointers and probe hash codes, we keep a pointer p to the cache line in the array that are $dist$ lines ahead of the current cache line ($dist = 20$ in our implementation). Suppose there are m pairs of pointers and hash codes in every cache line. The algorithm first issues prefetches for the first $dist$ lines and then sets p to the beginning of $dist+1$ line. Whenever the algorithm finishes processing m probe tuples, it issues a prefetch for the cache line pointed by p and increases p by a cache line. The algorithm checks p against the end of the array to stop prefetching. To prefetch for the probe tuples, we use a buffer to temporarily store the pairs of pointers pointing to the build and probe tuples that correspond to successful hash table probes. When this buffer is full, we visit these tuple pairs using software-pipelined prefetching.

Finally, we improve the robustness of our algorithm by issuing prefetches for the build tuples while prefetching for the probe tuples. In most cases, the build tuples are already in cache, and these prefetches do not have effects. However, if the build tuples are replaced somehow, the prefetches can bring the build tuples back into the cache quickly. We do not prefetch the hash table for the same purpose because it requires larger changes to the algorithm and therefore may incur significant run-time cost.

5.6 Experimental Results

In this section, we evaluate the CPU cache performance of our inspector joins against the cache prefetching and cache partitioning algorithms. Moreover, in Section 5.6.5, we exploit the inspection approach to detect situations where there are duplicate build keys or where relations are nearly sorted, and choose the best join phase algorithm.

5.6.1 Experimental Setup

Implementation Details. We implemented five hash join algorithms: group prefetching, software-pipelined prefetching, cache partitioning, enhanced cache partitioning with advanced prefetching support, and our inspector join algorithm. We store relations and intermediate partitions as disk files, and the join algorithms are implemented as stand-alone programs that read and write relations in disk files. We keep schemas and statistics in separate description files for simplicity. Statistics on the relations about the number of pages and the number of tuples are used to compute hash table sizes, numbers of partitions, and Bloom filter sizes.

Our cache prefetching implementations extend the cache prefetching algorithms described in Chapter 4 with Bloom filter support. The algorithms utilize a single Bloom filter for removing probe tuples having no matches. We add prefetches for the Bloom filter to the group and software-pipelined prefetching algorithm in the I/O partition phase. In our experiments, we find that the performance results of the two prefetching algorithms are very similar. To simplify presentation, we only show the group prefetching curves, which are labeled as "*cache prefetching*".

The two cache partitioning algorithms both use the group prefetching implementation for the I/O partition phase; they perform re-partition and join cache-sized sub-partitions in the join phase. The enhanced cache partitioning performs advanced prefetching similar to that of the inspector join for joining a pair of cache-sized sub-partitions. It also performs advanced prefetching to reduce the re-partition cost. This algorithm serves as a stronger competitor to our algorithm. In the figures that follow, enhanced cache partitioning is labeled as "*enhanced cpart*".

In every experiment, the number of I/O partitions generated is a multiple of the number of CPUs. Then the same join phase algorithm is run on every CPU to process different partitions in parallel. The partition phase algorithms take advantage of multiple CPUs by conceptually cutting input relations into equal-sized chunks and partitioning one chunk on every CPU. Every processor generates the same number of partition outputs. The *i*-th build partition will conceptually consist of the *i*-th build output generated by every processor. The probe partitions are generated similarly. Every CPU will build its own filter(s) based on the build tuples it sees. After partitioning the build relation, the generated filters are merged. For the single-filter scheme, all filters are OR-ed together to get a single filter. For the multi-filter scheme, different CPUs actually generate horizontal filters for different sub-partitions. Therefore, the algorithm can directly perform horizontal to vertical filter conversion. Then, the same filter(s) is shared across all the CPUs for testing probe tuples.

Finally, we report speedups when we compare inspector joins with another algorithm running on *the same number of CPUs*. Note that the speedups do not compare our algorithm running on multiple CPUs (e.g., 8 CPUs) with another algorithm running on a single CPU, and therefore should not be interpreted as the entire benefits of using multiple CPUs. Rather, the speedups show the additional benefits over advanced baseline algorithms that already run on multiple CPUs, while the entire benefits of using multiple CPUs can be obtained easily by examining the figures and noting that the wall-clock time is roughly equal to the aggregate time divided by the number of CPUs.

Experimental Design. We use the same relation schema as in Chapter 4: A tuple consists of a 4-byte randomly generated join key and a fixed-length payload. An output tuple contains all the fields of the matching build and probe tuples. In all the experiments except those in Section 5.6.5, a probe tuple can match zero or one build tuple, and a build tuple may match one or more probe tuples. We test the performance of our solution in various situations by varying the tuple size, the number of probe tuples matching a build tuple (which is the ratio between probe and build relation sizes), and the percentage of probe tuples that have matches. We vary the latter from 5% to 100% to model the effects of a selection on a build attribute different from the join attribute.

In all our experiments, we assume the available memory size for the join phase is 50MB and the cache size is 2MB, which follow the settings in Chapter 4. Note that when multiple join instances are running on multiple processors, the actual memory allocated is 50MB multiplied by the number of instances. For example, in the case of 32 CPUs, the total memory used for the join phase is 1600MB. The Bloom filter false positive rate (fpr) for the cache prefetching algorithm, and the two cache partitioning algorithms is set to be 0.05. The individual Bloom filter false positive rate (fpr') for our inspector join algorithm is set to be 0.001.

Measurement Methodology. We evaluate the CPU cache performance (of user mode executions) of the algorithms on a dual-processor Itanium 2 machine and through detailed cycle-by-cycle simulations. The Itanium 2 machine configuration is described previously in Section 2.4.1. We compile the algorithms with "icc -O3" and measure user-mode cache performance with the perfmon library. Each reported data point is the average of 30 runs; the standard deviation is less than 1% of the average in all cases.

The simulation parameters are described previously in Section 2.4.2. The only difference is that we simulate a single thread of execution in Chapter 2-4, while we simulate multiple (up to 32) execution streams on multiple processors sharing the same memory bus and main memory in this section.

As will be shown in Section 5.6.2, the main memory bandwidth is sufficient for the algorithms when

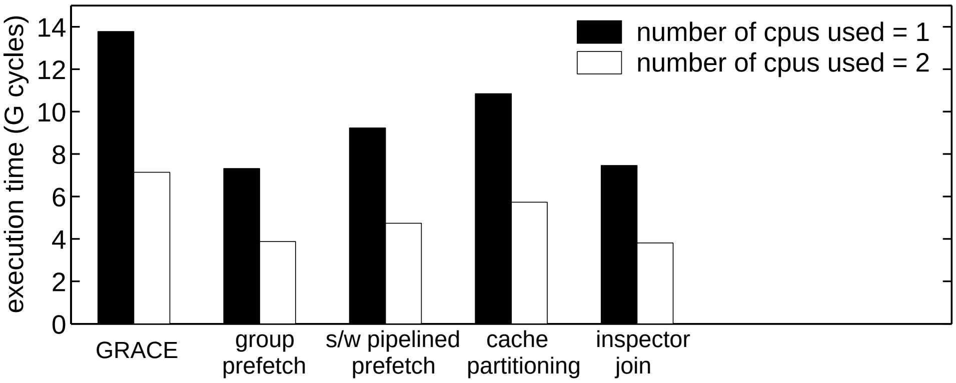


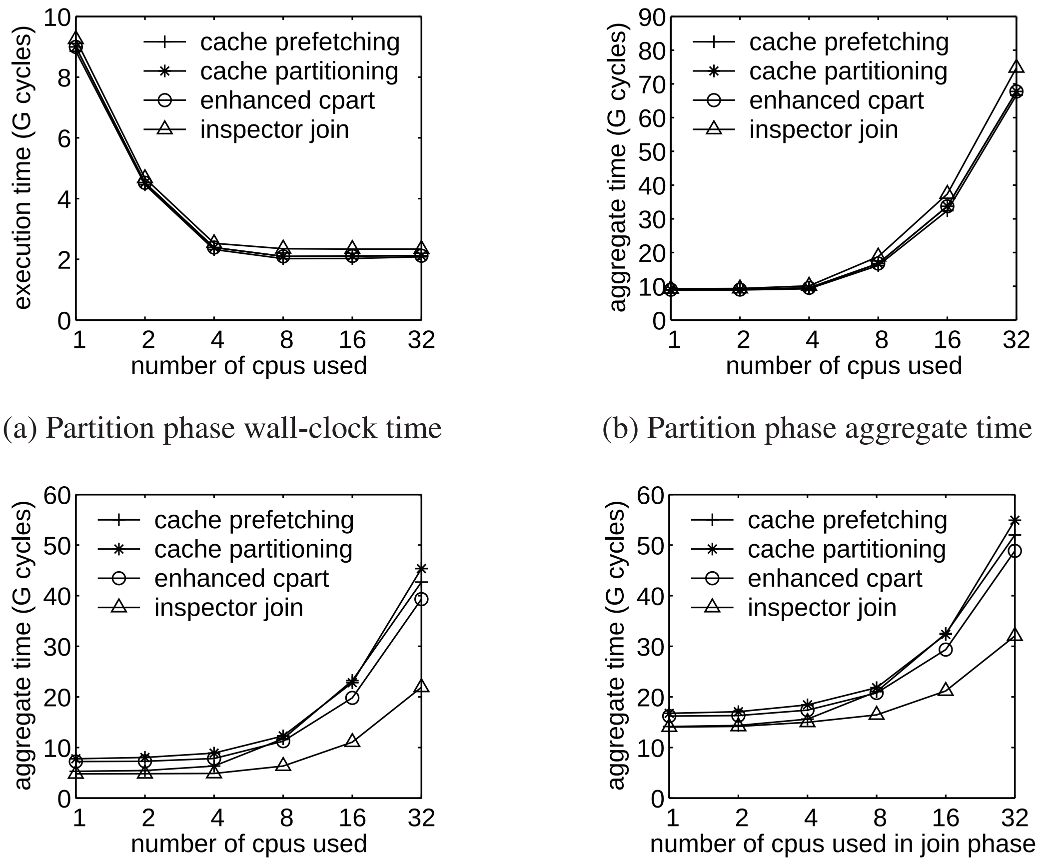
Figure 5.10: Join phase user-mode time varying the number of CPUs used on the Itanium 2 machine.

only two processors are used on both the Itanium 2 machine and our simulated machine model. However, previous cache-friendly algorithms degrade significantly when eight or more CPUs are used. Therefore, the majority of the experimental results in this section are obtained through simulations modeling more than two CPUs.

5.6.2 Varying the Number of CPUs

Figure 5.10 and Figure 5.11 compare the performance of the algorithms while varying the number of CPUs. The experiments join a 500MB build relation and a 2GB probe relation. The tuple size is 100B. 50% of the probe tuples have no matches and every build tuple matches 2 probe tuples.

Figure 5.10 shows the user-mode cache performance while we use one or two CPUs on the dual-processor Itanium 2 machine. We see that all the algorithms achieve near linear speedup when doubling the CPUs used; there is little impact of memory bandwidth sharing on the performance of the algorithms. By using simulations, we are able to model computer systems with larger number of CPUs. As shown in Figure 5.11, our simulation results support similar observations when less than 4 CPUs are used. However, when 8 or more CPUs are used, both cache prefetching and cache partitioning degrade significantly due to the sharing of the memory bandwidth. Because of this capability of simulating a large number of processors, we will focus on simulation results in the rest of Section 5.6.



(a) Partition phase wall-clock time

(b) Partition phase aggregate time

(c) Join phase aggregate time

(d) Total aggregate time (up to 4 CPUs in (a))

Figure 5.11: Varying the number of CPUs used through simulations.

Figure 5.11(a) shows the partition phase wall-clock time, and Figure 5.11(b) shows the aggregate execution time on all CPUs used in the partition phase. We see that all the partition phase curves are very similar. Compared to the other schemes, the inspector join incurs a slight overhead. (The ratio between the partition phase execution times of the best algorithm and the inspector join is 0.86-0.97.) This is mainly because of the computational cost of converting horizontal filters into vertical filters and testing a set of filters. The most costly operation is extracting the bit positions of 1's from a bit vector in both conversion and filter testing. This overhead will become less significant as processors are getting faster. As shown in Figure 5.11(a), all the curves become flat after the 4-CPU case. Therefore, all the following experiments use up to 4 CPUs in the partition phase. (If the join phase uses p CPUs, then the partition phase uses p CPUs when $p \leq 4$, and uses 4 CPUs when $p > 4$.)

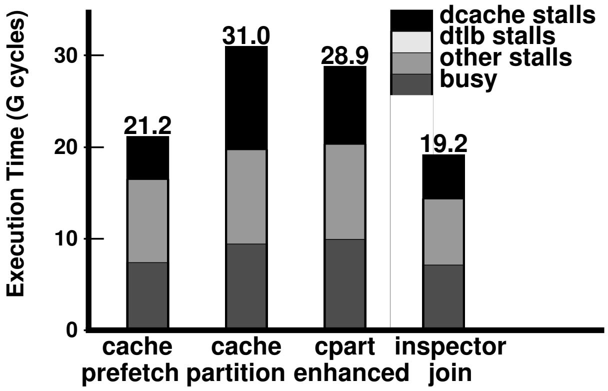
Figure 5.11(c) shows the aggregate time on all CPUs for the join phase. The cache prefetching and cache partitioning curves are the same as in Figure 5.1(b). Our inspector join is the best. Because of the memory bandwidth sharing effect, the cache prefetching curve degrades significantly when there are 8 or more CPUs. Since our algorithm combines locality optimizations and cache prefetching, it is less sensitive to bandwidth contention. Compared to the cache prefetching algorithms, our inspector join algorithm achieves 1.86-2.35X speedups when 8 or more CPUs are used in the join phase.

As shown in Figure 5.11(c), the two cache partitioning algorithms are worse than the cache prefetching algorithm and our inspector join when there are less than 4 CPUs. This is mainly because of the large re-partition overhead, which consists of more than 36% of their join phase execution times. The enhanced algorithm is always better than the original algorithm, which verifies the effectiveness of the applied prefetching techniques. As the number of CPUs increase, the enhanced algorithm becomes significantly better than the cache prefetching algorithms because it utilizes cache-sized sub-partitions to reduce the number of cache misses. However, it still degrades quickly beyond 4 CPUs. This is mainly because the re-partition step is quite sensitive to memory bandwidth sharing. Compared to the enhanced cache partitioning algorithm, our inspector join achieves 1.50-1.79X speedups with 1-32 CPUs.

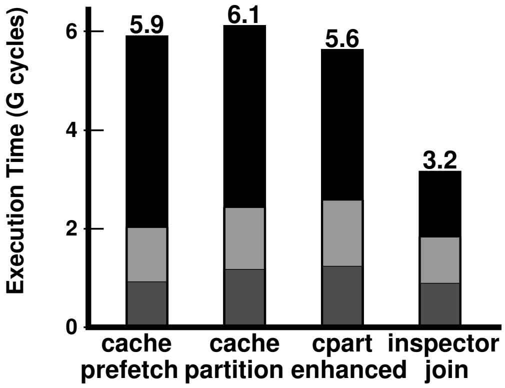
Figure 5.11(d) shows the aggregate performance of both phases using up to 4 CPUs in the partition phase. When there are 8 or more CPUs, inspector join achieves 1.26-1.75X speedups over the cache prefetching algorithm and the enhanced cache partitioning algorithm.[8]

Figure 5.12 shows the CPU time breakdowns for the join phase of the algorithms. The breakdowns are for the tasks running on CPU 0 in the system. The Y axis shows the execution time. Every bar is broken down into four categories: CPU busy time, stalls due to data cache misses (including the effect of L2 misses), stalls due to data TLB misses, and other resource stalls. Comparing Figure 5.12(a) and (b), we can see that the fractions of data cache stalls for the three left bars increase dramatically. This clearly shows the impact of memory bandwidth sharing on the performance. In contrast, our cache-

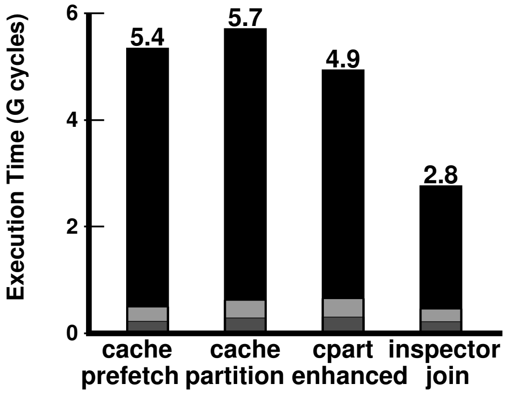
[8] Because we compare inspector joins against cache prefetching and cache partitioning algorithms, the above figures omit the GRACE hash join curves for clarity. Compared to the GRACE hash join algorithm, our inspector join achieves 1.62-4.14X speedups for the join phase and 1.75-2.83X speedups for the entire hash join with 1-32 CPUs. The speedups at 32 CPUs are 1.62X and 1.75X, respectively.



(a) num CPUs used = 1



(b) num CPUs used = 8



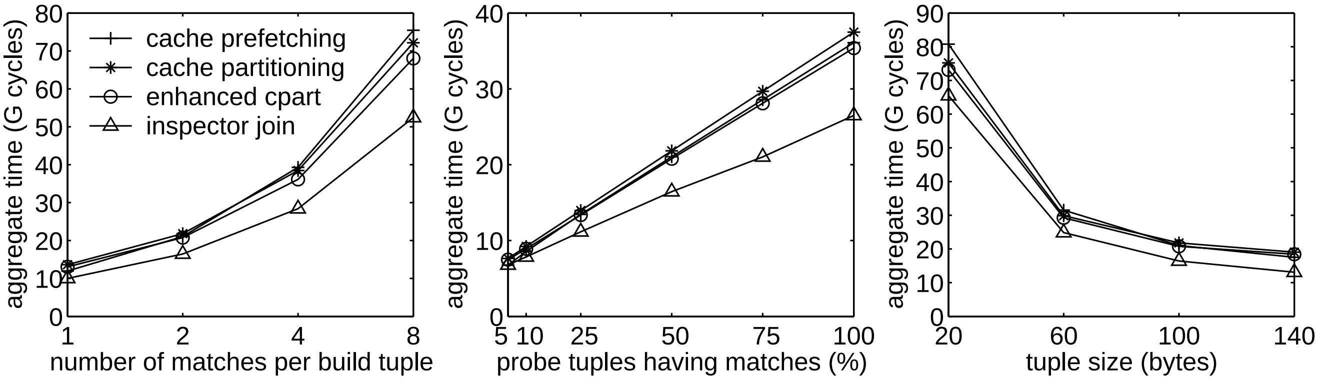
(c) num CPUs used = 32

Figure 5.12: Join phase CPU time breakdowns for CPU 0.

stationary algorithm (of the inspector join) achieves quite good cache performance. At 32 CPUs, cache stalls dominate all bars, as shown in Figure 5.12(c). Even in this case, our cache-stationary algorithm is better than the other algorithms.

5.6.3 Varying Other Parameters

Figure 5.13 shows the benefits of our inspector join algorithm over cache prefetching and cache partitioning while varying the number of probe tuples matching a build tuple (which is the ratio between probe and build relation sizes), the percentage of probe tuples that have matches, and the tuple size. All



(a) Varying the number of matches per build tuple

(b) Varying the percentage of probe tuples having matches

(c) Varying the tuple size

Figure 5.13: Aggregate execution time varying three parameters when 8 CPUs are used in the join phase.

the experiments use 8 CPUs in the join phase. The three figures share a common set of experiments, which correspond to the 8-CPU points in Figure 5.11.

Figure 5.11(a) varies the number of probe tuples matching every build tuple from 1 to 8 (while keeping the build relation size fixed). Essentially, we vary the ratio of probe relation size to build relation size from 1 to 8. Figure 5.11(b) varies the percentage of probe tuples having matches (while keeping the probe relation size fixed). Figure 5.11(c) varies the tuple size from 20B to 140B (while keeping the build relation size fixed). Note that the number of tuples decreases as the size of the tuple increases. Therefore, all the curves have the downward shape. Note that in the 20B experiments, a cache line of 64B contains multiple probe tuples. Since the cache-stationary join visits probe tuples non-sequentially, it may incur multiple cache misses for every cache line in the probe partition. However, our inspector join with cache-stationary join phase is still the best even for the 20B experiments.

In all the experiments, we can see that our inspector join algorithm is the best. For all the experiments except the 5% points in Figure 5.11(b)[9] , our inspector join achieves 1.09–1.44X speedups compared to the cache prefetching algorithm and the enhanced cache partitioning algorithm.

[9]When there are only 5% probe tuples having matches, the aggregate join phase execution time only consists of 9–23% of the total aggregate execution time. Therefore, the difference is small among all the algorithms optimizing the join phase performance.

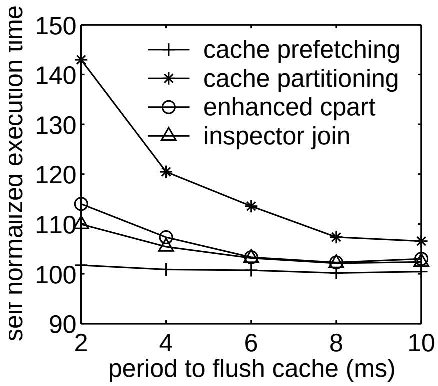


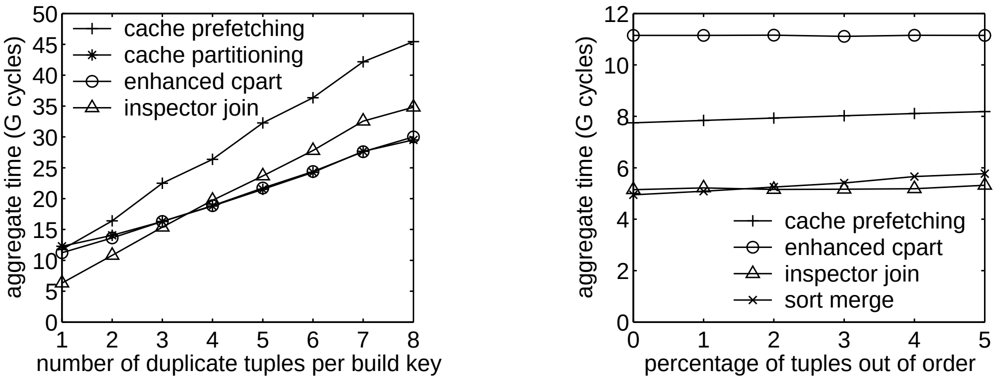
Figure 5.14: Robustness against cache interference (join phase performance *self-normalized* to the performance of the same algorithm without cache flushing, num CPUs used=1).

5.6.4 Robustness of the Algorithms

Figure 5.14 shows the performance degradation of all the algorithms when the cache is periodically flushed, which is the worst case interference. We vary the period to flush the cache from 2 ms to 10 ms, and report the execution times self normalized to the no flush case. That is, “100” corresponds to the join phase execution time when there is no cache flush.

The cache prefetching algorithm sees at most 2% performance degradation because of cache flushes. It is very robust because it does not assume that any large data structures stay in the cache. In contrast, the original cache partitioning algorithm assumes the exclusive use of the cache, and suffers from a 7-43% performance degradation for the cache flushes. Like the original cache partitioning, our inspector join algorithm and the enhanced cache partitioning algorithm both try to keep a build sub-partition and its hash table in the cache. To improve robustness, both of the algorithms perform prefetching for build tuples.[10] As shown in Figure 5.14, this technique effectively reduces the performance degradation to 2-14%, which is a 2-4X improvement compared to the original cache partitioning.

[10] As we do not prefetch for the hash table, we expect to pay higher cost than pure prefetching schemes when the cache is flushed. Prefetching for the hash table is much more complicated than prefetching only for build tuples, and may incur more run-time overhead for normal execution.



(a) Duplicate build keys

(b) Nearly sorted relations

Figure 5.15: Exploiting the inspection mechanism.

5.6.5 Choosing the Best Join Phase Algorithm

By default, our inspector join algorithm uses the cache-stationary join phase. However, our inspection approach can detect situations where cache-stationary join phase is not the best algorithm and choose a different one. Figure 5.15(a) varies the number of duplicate build tuples per build join attribute value. The duplicate-free points correspond to the 8-CPU points in Figure 5.11(c). We see that the default cache-stationary join phase of the inspector algorithm is still the best until 3 duplicates per key. However, when there are 4 duplicates per key, enhanced cache partitioning gets better. The prefetching algorithm needs to visit multiple build tuples for every probe tuple in the duplicate key cases. Since the visits are all cache misses, the performance of the prefetching algorithm suffers significantly.

As discussed in Section 5.3.4, our inspection approach detects duplicate keys by counting the average number of sub-partitions matching every probe tuple. To compute this value, we sum up the number of matching probe tuples of every sub-partition (which has already been maintained to enable the counting sort). We then divide this value by the total number of probe tuples in the partition to obtain the average. Therefore, the additional overhead for this computation is only S integer additions plus 1 integer division, where S is the number of sub-partitions in the partition. Compared to the partition aggregate time, which is on the order of 10^9 cycles, this additional overhead is almost zero. When a probe tuple on average matches 4 or more sub-partitions in a partition, our inspection approach chooses enhanced cache partitioning as the join phase algorithm. Thus the actual inspector join performance tracks the best of the *inspector join* and *enhanced cpart* curves in the figure. The speedup increases as the number of duplicates. For the case with 8 duplicate keys, our inspection approach achieves 1.16X speedups over the cache-stationary join phase.

Figure 5.15(b) shows the performance when the source relations are nearly sorted. We vary the percentage of tuples out of order from 0% (fully sorted) to 5%. For the fully sorted case, we sort the input relations for the 8-CPU points in Figure 5.11(c). Then we randomly choose 1%–5% of tuples and randomly change their locations to generate the other test cases.

As shown in Figure 5.15(b), the sort-merge algorithm performs the best as expected for the fully sorted case. However, to our surprise, the inspector join performs equally well. The reason is that for the fully sorted case, the build tuples in a build sub-partition are sorted, and the corresponding probe tuples are contiguous in the probe partition. Therefore, the cache-stationary join phase essentially visits both the build and the probe partitions sequentially. Since the hash table is kept in cache, the cache behavior of the cache-stationary join phase is the same as the sort merge join, which only merges two in-order inputs. However, when more and more tuples are out of order, the sort merge join pays increasingly more cost to sort out-of-order tuples. In contrast, the inspector algorithm pays only a slight overhead to visit some probe tuples non-sequentially. Therefore, the inspector join becomes better than the sort merge join when 3% or more tuples are out of order.

5.7 Chapter Summary

In this chapter, we have proposed and evaluated *inspector joins*, which exploit the fact that during the I/O partitioning phase of a hash-based join, we have an almost free opportunity to inspect the actual properties of the data that will be revisited later during the join phase. We use this "inspection" information in two ways. First, we use this information to accelerate a new type of cache-optimized join phase algorithm. The cache-stationary join phase algorithm is especially useful when the join is run in parallel on a multiprocessor, since it consumes less of the precious main memory bandwidth than existing state-of-the-art schemes. Second, information obtained through inspection can be used to choose a join phase algorithm that is best suited to the data. For example, inspector joins can choose enhanced cache partitioning as the join phase algorithm when a probe tuple on average matches 4 or more sub-partitions. Our experimental results demonstrate:

- Inspector joins offer speedups of 1.09–1.75X over the best existing cache-friendly hash join algorithms (i.e. cache prefetching and cache partitioning) when running on 8, 16, or 32 processors, with the advantage growing with the number of processors.
- Inspector joins are effective under a large range of joining conditions (e.g., various tuple sizes, various fractions of tuples with matches, etc.).
- Cache prefetching improves the robustness of inspector joins against cache interference.
- The inspection mechanism can be used to select among multiple join phase algorithms for the given query and data.

In summary, Inspector Joins are well-suited for modern multi-processor database servers. Moreover, we believe the inspection concept can be potentially exploited in other multi-pass algorithms (such as external sorting [29]).

Chapter 6

Conclusions

Widening exponentially, the *cache-to-memory* latency gap has become one of the major performance bottlenecks for database systems. While a great many recent database studies focused on reducing the *number* of cache misses for database algorithms, little research effort has exploited cache prefetching for reducing the *impact* of cache misses for database systems. In this thesis, we present the first systematic study of *cache prefetching* for improving database CPU cache performance. Combining cache prefetching and data locality optimization techniques, we redesigned the B$^+$-Tree index structure as a representative tree structure and the hash join algorithm as a representative algorithm employing hash tables. Both our simulation studies and real machine experiments support the following conclusions:

- **Exploiting Cache Prefetching for Main Memory B$^+$-Trees**. For index search operations, we find that contrary to conventional wisdom, the optimal B$^+$-Tree node size on a modern machine is often *wider* than the natural data transfer size (i.e. a cache line), since we can use prefetching to fetch each piece of a node simultaneously. Prefetching wider nodes offers the following advantages relative to B$^+$-Trees and CSB$^+$-Trees: (i) better search performance (1.16-1.85X speedups over B$^+$-Trees and 1.08-1.53X speedups over CSB$^+$-Trees) because wider nodes effectively reduce the height of the trees and this scheme can increase the fanout by more than the factor of two that CSB$^+$-Trees provide (e.g., by a factor of eight); (ii) comparable or better performance on updates (up to 1.77X speedups over B$^+$-Trees) because of the improved search speed and the decreased frequency of node splits due to wider nodes; (iii) no fundamental changes to the original B^+-Tree data structures or algorithms. In addition, we find that our scheme is *complementary* to CSB^+-Trees. For (non-clustered) index range scan operations, our results demonstrate that jump-pointer array prefetching can effectively hide 98% of the cache miss latency suffered by range scans, thus resulting in *a factor of up to 18.7X speedup* over a range of scan lengths.

- **Optimizing Both Cache and Disk Performance for B^+-Trees in Commercial DBMSs**. We find that *Fractal Prefetching B^+-Trees*, which embed cache-optimized trees into disk pages, is an effective solution for simultaneously achieving both good cache and good disk performance for B^+-Trees. Because directly combining the trees of the two granularities often result in the node size mismatch problem, we developed two solutions for solving this problem: the *disk-first* and the *cache-first* implementations of fpB^+-Trees. Since the cache-first implementation may incur large disk overhead, we recommend in general the disk-first implementation. Compared with disk-optimized B^+-Trees with slotted pages, disk-first fpB^+-Trees achieve the following advantages with only a slight I/O overhead: (i) *1.20-4.49X speedups for search* because of the improved spatial locality inside each disk page; (ii) up to a factor of 20-fold improvement for range scans because of prefetching; and (iii) up to a 15-fold improvement for updates by avoiding slot indirection cost and large data movement with cache-optimized nodes. Moreover, our detailed analyses of applying the cache prefetching techniques of pB^+-Trees to the memory-to-disk gap show that while prefetching wider nodes may be less attractive because of the increased number of disk seek operations, jump-pointer array I/O prefetching leads to large performance gains for (non-clustered) index range scans. In particular, we demonstrate a *2-5X* improvement for index range scans in an industrial-strength commercial DBMS (IBM's DB2).

- **Improving Hash Join Performance through Prefetching**. We find that while it is difficult to prefetch within the processing of a single tuple because of the random accesses to the hash table and because of the dependences between consecutive memory references, it is a good idea to exploit *inter-tuple parallelism* to overlap cache miss latencies across the processing of multiple tuples. Our proposals, *group prefetching* and *software-pipelined prefetching*, present two systematic ways to reorganize the hash join program to effectively attain this goal. Compared with GRACE and simple prefetching approaches, our prefetching techniques achieve 1.29-4.04X speedups for the join phase and 1.37-3.49X speedups for the partition phase for user mode cache performance. Comparing the elapsed real times on an Itanium 2 machine with multiple disks, the two cache prefetching techniques achieve 1.12-1.84X speedups for the join phase and 1.06-1.60X speedups for the partition phase. Moreover, they are 36-77% faster than cache partitioning on large relations and do not require exclusive use of the cache to be effective. Finally, extensive comparisons between group prefetching and software-pipelined prefetching demonstrate that contrary to the conventional wisdom in the compiler optimization community, group prefetching can be significantly faster (1.23-1.30X speedups for the join phase performance on the Itanium 2 machine) than software-pipelined prefetching because of less instruction overhead.

- **Inspector Joins**. We find that it is an effective approach to exploit the free information obtained from one pass of the hash join algorithm to improve the performance of a later pass. Our multi-filter scheme allows the extraction of approximate matching information between the two joining relations almost for free during the I/O partitioning phase. Later, this information is used in the join phase to generate cache-sized sub-partitions without tuple copying. This optimization scheme both addresses the memory sharing problem in a shared-bus multi-processor system and eliminates the re-partitioning cost of the cache partitioning technique. Moreover, we demonstrate that cache prefetching can effectively improve the robustness of inspector joins in the face of cache interference. Furthermore, we present illustrative examples of how inspector joins can use its collected statistics to select among multiple join phase algorithms for the given query and data. Finally, our experiments demonstrate that as we run on 8 or more processors, inspector joins achieve 1.09–1.75X speedups over previous state-of-the-art cache prefetching and cache partitioning algorithms, with the speedup increasing as the number of processors increases.

Summarizing the above specific findings from our B^+-Tree and hash join studies, we draw the following general conclusions:

- Cache prefetching is an effective optimization technique that reduces the *impact* of cache misses for improving the CPU cache performance of B^+-Trees and hash joins.

- Combined with data locality optimizations, cache prefetching provides larger freedom in redesigning data structures and algorithms, thus leading to better performance, as demonstrated in fractal prefetching B^+-Trees and inspector joins.

- Cache prefetching techniques are robust against cache interference, and they can be employed to improve the robustness of an algorithm, as demonstrated in both pieces of our hash join work.

Finally, the techniques we developed for B^+-Trees and hash joins are potentially applicable to other tree-based index structures (e.g., spatial indices [33, 87]) and other hash-based algorithms (e.g., hash-based group-by and aggregation algorithms [29]).

Bibliography

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th VLDB*, pages 169–180, Sept. 2001.

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

[3] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[4] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91*, pages 176–186, Nov. 1991.

[5] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th ISCA*, pages 3–14, June 1998.

[6] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th ISCA*, pages 282–293, June 2000.

[7] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[8] Rudolf Bayer and Mario Schkolnick. Concurrency of Operations on B-Trees. *Acta Informatica*, 9:1–21, 1977.

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

[10] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the 1986 SIGMOD Conference*, pages 61–71, May 1986.

[11] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13(7):422–426, 1970.

[12] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K.S. Venkatraman. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 08(01):1–17, Feb. 2004.

[13] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. Improving Main-Memory Index Performance with Partial Key Information. In *Proceedings of the 2001 SIGMOD Conference*, pages 163–174, May 2001.

[14] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th VLDB*, pages 54–65, Sept. 1999.

[15] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th VLDB*, pages 181–190, Sept. 2001.

[16] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th ICDE*, pages 116–127, March 2004.

[17] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Inspector Joins. In *Proceedings of the 31st VLDB*, Aug.-Sep. 2005.

[18] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving Index Performance through Prefetching. In *Proceedings of the 2001 SIGMOD Conference*, pages 235–246, May 2001.

[19] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal Prefetching B^+-Trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the 2002 SIGMOD Conference*, pages 157–168, June 2002.

[20] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-Conscious Structure Layout. In *Proceedings of PLDI '99*, pages 1–12, May 1999.

[21] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th VLDB*, pages 127–141, Aug. 1985.

[22] Christopher B. Colohan, Anastassia Ailamaki, J. Gregory Steffan, and Todd C. Mowry. Optimistic Intra-Transaction Parallelism on Chip Multiprocessors. In *Proceedings of the 31st VLDB*, pages 73–84, Aug.-Sept. 2005.

[23] Christos Faloutsos. Signature files: Design and Performance Comparison of Some Signature Extraction Methods. In *Proceedings of the 1985 SIGMOD Conference*, pages 63–82, May 1985.

[24] Christos Faloutsos and Raphael Chan. Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison. In *Proceedings of the 14th VLDB*, pages 280–293, Aug.-Sept. 1988.

[25] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, Dec. 1992.

[26] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th ISCA*, pages 162–171, May 1999.

[27] GNU project C and C++ compiler. http://www.gnu.org/software/gcc/gcc.html.

[28] Brian T. Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating Database Operations Using a Network Processor. In *Proceedings of the First International Workshop on Data Management on New Hardware (DaMoN 2005)*, June 2005.

[29] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[30] Goetz Graefe. The Value of Merge-Join and Hash-Join in SQL Server. In *Proceedings of the 25th VLDB*, pages 250–253, Sept. 1999.

[31] Goetz Graefe and Per-Åke Larson. B-tree Indexes and CPU Caches. In *Proceedings of the 17th ICDE*, pages 349–358, April 2001.

[32] Jim Gray and Goetz Graefe. The Five-Minute Rule Ten Years Later. *ACM SIGMOD Record*, 26(4):63–68, Dec. 1997.

[33] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 SIGMOD Conference*, pages 47–57, June 1984.

[34] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael K. Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[35] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[36] Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *Proceedings of the 29th VLDB*, pages 417–428, Sept. 2003.

[37] Richard A. Hankins and Jignesh M. Patel. Effect of Node Size on the Performance of Cache-Conscious B^+-trees. In *Proceedings of the 2003 SIGMETRICS Conference*, pages 283–294, June 2003.

[38] Stavros Harizopoulos and Anastassia Ailamaki. STEPS towards Cache-resident Transaction Processing. In *Proceedings of the 30th VLDB*, pages 660–671, Aug.-Sep. 2004.

[39] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[40] IBM Corp. *IBM DB2 Universal Database Administration Guide Version 8.2*. 2004.

[41] IBM DB2 Universal Database. http://www.ibm.com/software/data/db2/.

[42] Intel C++ Compiler.
http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/index.htm.

[43] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual, Volumn 2B:Instruction Set Reference N-Z*. Order Number: 253667.

[44] Intel Corp. *Intel Itanium 2 Processor Reference Manual For Software Development and Optimization*. Order Number: 251110-003.

[45] Intel Corp. *Intel Itanium Architecture Software Developer's Manual*. Order Number: 245317-004.

[46] Intel Corp. *Intel Pentium4 and Intel Xeon Processor Optimization*. Order Number: 248966-007.

[47] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In *Proceedings of the 20th VLDB*, pages 48–59, Sept. 1994.

[48] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th VLDB*, pages 439–450, Sept. 1994.

[49] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of the 1998 SIGMOD Conference*, pages 106–117, June 1998.

[50] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar./Apr. 2004.

[51] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 25th ISCA*, pages 15–26, June 1998.

[52] Ken Kennedy and Kathryn S. McKinley. Loop Distribution With Arbitrary Control Flow. In *Proceedings of Supercomputing'90*, pages 407–416, Nov. 1990.

[53] Kihong Kim, Sang Kyun Cha, and Keunjoo Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proceedings of the 2001 SIGMOD Conference*, pages 139–150, May 2001.

[54] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing*, 1(1):63–74, 1983.

[55] Monica S. Lam. *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon University, May 1987.

[56] Philip L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6(4):650–670, Dec. 1981.

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

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

[59] Bruce Lindsay. Hash Joins in DB2 UDB: the Inside Story. *Carnegie Mellon DB Seminar*, March 2002.

[60] Witold Litwin and David B. Lomet. The Bounded Disorder Access Method. In *Proceedings of the 2nd ICDE*, pages 38–48, Feb. 1986.

[61] David Lomet. B-tree Page Size when Caching is Considered. *ACM SIGMOD Record*, 27(3):28–32, Sep. 1998.

[62] David Lomet. The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. *ACM SIGMOD Record*, 30(3):64–69, Sep. 2001.

[63] David B. Lomet. Partial Expansions for File Organizations with an Index. *ACM Transactions on Database Systems*, 12(1):65–84, 1987.

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

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

[66] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Proceedings of the 26th VLDB*, pages 339–350, Sept. 2000.

[67] Stefan Manegold, Peter A. Boncz, Niels Nes, and Martin Kersten. Cache-Conscious Radix-Decluster Projections. In *Proceedings of the 30th VLDB*, pages 684–695, Aug.-Sept. 2004.

[68] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust Query Processing through Progressive Optimization. In *Proceedings of the 2004 SIGMOD Conference*, pages 659–670, June 2004.

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

[70] Microsoft SQL Server. http://www.microsoft.com/sql/default.mspx.

[71] Monet Main Memory Database. http://sourceforge.net/projects/monetdb/.

[72] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

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

[74] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the 14th VLDB*, pages 468–478, Aug. 1988.

[75] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the 1994 SIGMOD Conference*, pages 233–242, May 1994.

[76] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 SIGMOD Conference*, pages 297–306, May 1993.

[77] Oracle Database. http://www.oracle.com/database/index.html.

[78] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proceedings of the 17th ICDE*, pages 567–574, Apr. 2001.

[79] Perfmon Project. http://www.hpl.hp.com/research/linux/perfmon/index.php4.

[80] Published SPEC Benchmark Results. http://www.spec.org/results.html.

[81] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2003.

[82] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A Case for Fractured Mirrors. In *Proceedings of the 28th VLDB*, pages 430–441, Aug. 2002.

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

[84] Jun Rao and Kenneth A. Ross. Making B^{+}-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 SIGMOD Conference*, pages 475–486, May 2000.

[85] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *Proceedings of the 27th ISCA*, pages 117–127, May 2000.

[86] Timos K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th VLDB*, pages 507–518, Sept. 1987.

[88] Dennis G. Severance and Guy M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems*, 1(3):256–267, 1976.

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

[90] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th VLDB*, pages 510–521, Sept. 1994.

[91] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, New York, New York, 5th edition, 2005.

[92] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th ISCA*, pages 1–12, June 2000.

[93] S. Stiassny. Mathematical Analysis of Various Superimposed Coding Methods. *American Documentation*, 11(2):155–169, Feb. 1960.

[94] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th VLDB*, pages 19–28, Sept. 2001.

[95] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, Sept. 1976.

[96] Sun Microsystems. *UltraSPARC IV Processor Architecture Overview*. Technical Whitepaper, Version 1.0, Feb. 2004.

[97] TimesTen Main Memory Database System. http://www.timesten.com/.

[98] TPC Benchmarks. http://www.tpc.org/.

[99] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd ISCA*, pages 392–403, June 1995.

[100] Patrick Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[101] Hansjörg Zeller and Jim Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In *Proceedings of the 16th VLDB*, pages 186–197, Aug. 1990.

[102] Jingren Zhou and Kenneth A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *Proceedings of the 29th VLDB*, pages 405–416, Sept. 2003.

[103] Jingren Zhou and Kenneth A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *Proceedings of the 2004 SIGMOD Conference*, pages 191–202, June 2004.

[104] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–342, 1977.