Improving Hash Join Performance through Prefetching

Shimin Chen Anastassia Ailamaki Phillip B. Gibbons[†]
Todd C. Mowry

October 2003

CMU-CS-03-157

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Intel Research Pittsburgh, 417 South Craig St., Pittsburgh, PA 15213, USA.

Abstract

Hash join algorithms suffer from extensive CPU cache stalls. This paper shows that the standard hash join algorithm for disk-oriented databases (i.e. GRACE) spends over 73% of its user time stalled on CPU cache misses, and explores the use of *prefetching* to improve its cache performance. Applying prefetching to hash joins is complicated by the data dependencies, multiple code paths, and inherent randomness of hashing. We present two techniques, *group prefetching* and *software-pipelined prefetching*, that overcome these complications. These schemes achieve 2.0–2.9X speedups for the join phase and 1.4–2.6X speedups for the partition phase over GRACE and simple prefetching approaches. Compared with previous cache-aware approaches (i.e. cache partitioning), the schemes are at least 50% faster on large relations and do not require exclusive use of the CPU cache to be effective.

Keywords: Cache performance, databases, hash join, group prefetching, software-pipelined prefetching

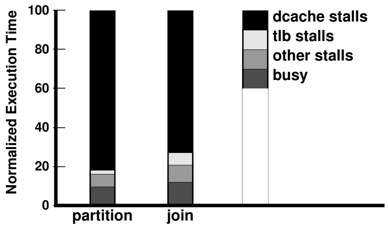


Figure 1: Execution time breakdown for hash join.

1 Introduction

Hash join [10, 14, 16, 23, 27, 29] 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 from excessive random disk accesses. To avoid this problem, the *GRACE* hash join algorithm [14] 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 sequential I/Os for source relations and intermediate partitions. As a result, the I/O costs no longer dominate. For example, our experiments on a quad-processor Pentium III show that a hash join of two several GB relations is already CPU-bound with only 4 disks, and it becomes increasingly CPU bound with each additional disk (details in Section 7).

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 [5, 19, 28]. 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 from the full latency to main memory during building and probing. Figure 1 provides a breakdown of the simulated user-level performance on a state-of-the-art machine (details in Section 7). The "partition" experiment divides a 1GB 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, TLB miss stalls, and other stalls. As we see in Figure 1, both the *partition* and *join* phases spend a significant fraction of their time—82% and 73%, 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 two 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 database environments [5, 19, 28]. However, cache partitioning suffers from two important practical limitations. First, for traditional disk-oriented databases, partitioning into 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 hash join [16]. Given a 1MB CPU cache and (optimistically) 1000 partitions, the *maximum relation size that can be handled is only 1 GB*. Beyond that hard limit, any cache partitioning must be done using additional passes through the data — as will be shown in Section 7, this results in at least a 50% 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 67% in the experiments in Section 7). Hence, we would like to explore an alternative technique that does not suffer from these limitations.

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 *hide* the cache miss latency associated with accessing normal (memory-sized) hash tables, by overlapping these misses with computation. Modern processors allow multiple cache misses to be in flight simultaneously in the memory hierarchy (e.g., the Compaq ES40 [8] supports 32 in-flight loads, 32 in-flight stores, and 8 outstanding off-chip cache misses per processor), and the trend has been toward supporting more and more simultaneous misses. To enable software to fully exploit this parallelism, modern processors also provide explicit *prefetch* instructions for moving data into the cache ahead of its use. Software-based prefetching has been successfully applied in the past to array-based programs [22], pointer-based programs [17], and database B$^+$-Trees [6, 7], but it has not been applied to hash joins.

Challenges in Applying Prefetching to Hash Join. A naive 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* [17] 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. Hardware-based prefetching techniques [2] 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; while this approach is useful for hiding the latency of primary data cache misses that hit in the secondary cache, the amount of lookahead buffering (in the *reorder buffers*) is far too small to fully hide the latency of cache misses to main memory [9] (e.g., 128 vs. 600 entries for the Compaq ES40 [8]), 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 [17, 22] cannot handle the ambigous data dependencies present in the hash join code (as discussed in detail in Sections 4.4 and 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) 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* [15] 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 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 [15]).

[1] In our experimental set-up in Section 7, $G = 19$ is optimal.

1.3 Contributions of This Paper

This paper makes the following contributions. First, to our knowledge, this is the first study to explore how prefetching can be used to accelerate both the join and partition phases of hash join by exploiting inter-tuple parallelism. Second, we propose two prefetching techniques, *group prefetching* and *software-pipelined prefetching*, and show how they can be applied to significantly improve hash join performance. Overall, the techniques achieve 2.0–2.9X speedups for the join phase and 1.4–2.6X speedups for the partition phase over GRACE and simple prefetching approaches. Moreover, they are at least 50% faster than cache partitioning on large relations and do not require exclusive use of the cache to be effective. Third, we make extensive comparisons between group prefetching and software-pipelined prefetching. Finally, we develop generalized models to understand group and software-pipelined prefetching.

The paper is organized as follows. Section 2 discusses the related work in greater detail. Section 3 analyzes the dependencies in the join phase, the more complicated of the two phases, while Sections 4 and 5 use group prefetching and software-pipelined prefetching to improve the join phase performance. Section 6 discusses prefetching for the partition phase. Experimental results appear in Section 7 and conclusions in Section 8.

2 Related Work

Since the GRACE hash join algorithm was first introduced [14], 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 [10, 16, 23, 27, 29]. All of these hash join algorithms, however, share two common building blocks: (1) *partitioning* and (2) *joining* with in-memory hash tables. To cleanly separate these two phases, we use GRACE as our baseline algorithm throughout this paper. We point out, however, that our techniques should be directly applicable to the other hash join algorithms.

CPU cache performance has been identified as a major performance bottleneck for database systems [1, 3, 12] and many recent studies have focused on improving the cache performance of core database algorithms [4, 5, 6, 7, 24, 25, 28]. Several papers have developed techniques to improve the cache performance of hash joins [5, 19, 28]. Shatdal *et al.* showed that cache partitioning achieved 6-10% improvement for joining memory-resident relations with 100B tuples [28]. Boncz, Manegold and Kersten proposed using multiple passes in cache partitioning to avoid cache and TLB thrashing [5, 19]. 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 [6, 7, 17, 22]. While software pipelining has been used to schedule prefetches in array-based programs [22], 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 [5, 19], particularly when TLB misses are handled by software. Since the vast majority of modern processors (including those from Intel) handle TLB misses in hardware, we model hardware-based TLB miss handling in our simulations. In addition, our simulator supports TLB prefetching [26] by treating TLB misses caused by prefetches as normal TLB misses. Hence, using our prefetching techniques, the TLB misses are overlapped with computation, minimizing TLB stall time.

3 Dependencies in the Join Phase

In this section, we analyze the dependencies in a hash table visit in the join phase to show why a naive prefetching algorithm would fail. We study a concrete implementation of the in-memory hash table, as shown in Figure 2. It 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.[2] A single hash cell is put into the bucket header. When more tuples are hashed to the bucket, a hash cell array is allocated, the size of which can be dynamically increased.

[2] Hash codes are usually good distinguishers of the join keys.

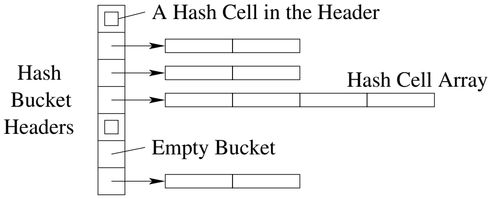


Figure 2: An in-memory hash table structure

A naive 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.[3] Therefore, applying prefetching to the join phase algorithm is not a straightforward task.

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 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 schedule prefetch instructions early enough. In this section, we propose group prefetching to achieve these objectives.

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 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 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[4]. 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.

4.2 Understanding Group Prefetching

To better understand group prefetching, we generalize the previous algorithms of Figure 3(a) and (b) in Figure 3(c) and (d). Suppose we need to process N independent elements. For each element i, we need to make k dependent

[3] The structure in Figure 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 [18, 6].

[4] Technically, what we do are modified forms of compiler transformations called *strip-mining* and *loop distribution* [13].

```
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

```
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 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

```
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 3: Group prefetching

memory references, $m_i^1, m_i^2, \cdots, m_i^k$. As shown in Figure 3(c), a straightforward algorithm processes an element per loop iteration. The loop body is natually divided into $k+1$ stages 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 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 instruction for the memory location so that the reference will be overlapped 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, and the additional latency of fetching the next cache line in parallel is T_{next}, which is the inverse of memory bandwidth. (Table 1 shows the terminology used throughout the paper.) Assume every m_i^l incurs a cache miss and there are no cache conflicts [5]. Then, the sufficient condition for fully hiding all cache miss latencies is as follows:

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

We will give the proof for the condition in the next subsection. For an intuitive explanation, let us focus on the

[5] We use these assumptions only to simplify the derivation of the conditions. Note that our experimental evaluations include all the possible effects of locality and conflicts in hash joins.

Table 1: Terminology used throughout this paper.

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

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 references 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 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.

4.3 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 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 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, 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

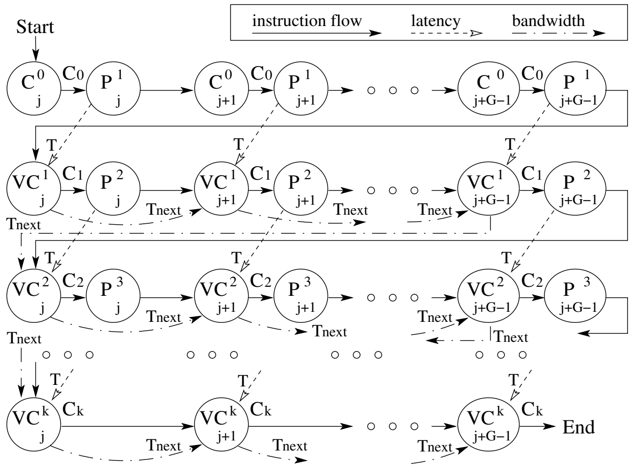


Figure 4: Critical path analysis for an iteration of the outer loop body in Figure 3(d)

inverse of memory bandwidth. [6]

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 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:*

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

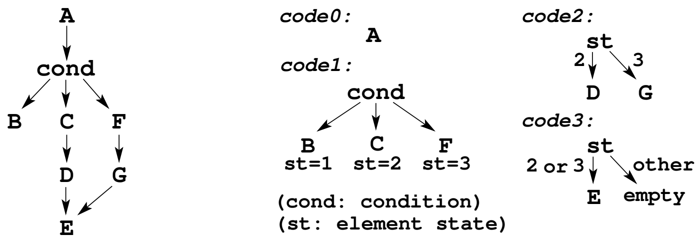
Proof. The first inequality ensures that the first latency edge from row 0, i.e. the edge from vertex P_j^1 to vertex VC_j^1 in the graph, is shorter than the path along row 0. The second inequality ensures that the first latency edge from row l in the graph, i.e. the edge from vertex P_j^{l+1} to vertex 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 xth latency edge from row 0, i.e. the edge from vertex P_{j+x-1}^1 to vertex VC_{j+x-1}^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 xth latency edge from row l, i.e. the edge from vertex P_{j+x-1}^{l+1} to vertex VC_{j+x-1}^{l+1}, where $l = 1, 2, \cdots, k-1$,

[6] 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.



(a) An example

(b) Code stages in group prefetching

Figure 5: Dealing with multiple code paths.

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. □

4.4 Dealing with Complexities

Previous research showed how to prefetch for two dependent memory references for array-based codes [21]. Our group prefetching algorithm solves the problem of prefetching for an arbitrary fixed number k of dependent memory references.

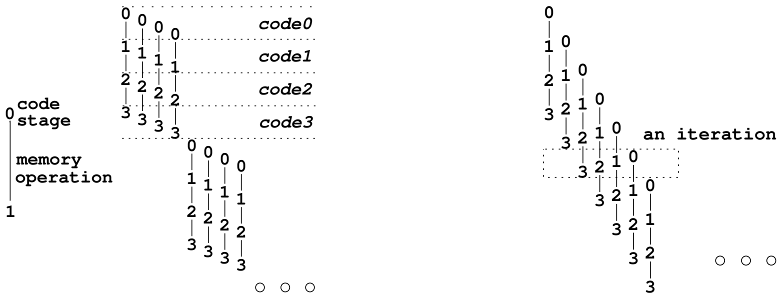
We have implemented group prefetching for both hash table building and probing. In contrast to the simplified probing algorithm, the actual probing algorithm contains multiple code paths: there could be zero or multiple matches, hash buckets could be empty, and there may not be a hash cell array in a bucket. To cope with this complexity, we keep state information for the G tuples of a group. We divide 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 5 shows 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 group prefetching 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. 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 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. The algorithm can deal with any number of delayed tuples (to tolerate skews in the key distribution).

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.



(a) Group prefetching ($G = 4$)

(b) Software-pipelined prefetching ($D = 1$)

Figure 6: Intuitive pictures of the prefetching schemes

```
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 7: Software-pipelined prefetching

Figure 6 shows 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.

5.1 Understanding Software-pipelined Prefetching

Figure 7(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* [21].) Figure 6(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 7(b) shows the generalized algorithm for software-pipelined prefetching. In the steady state, the pipeline

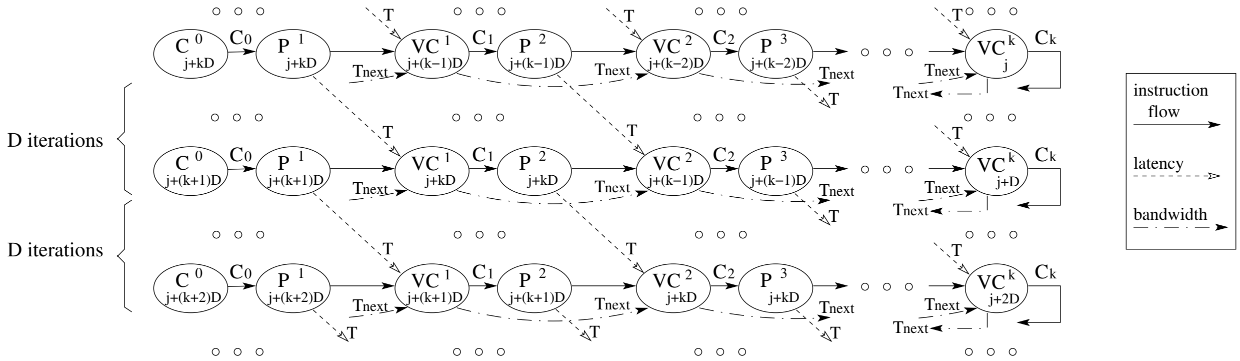


Figure 8: Critical path analysis for software-pipelined prefetching (steady state)

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 latencies. Under the same assumption as in Section 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.)

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

We can always choose a D sufficiently large to satisfy this condition. In our experiments in Section 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.

5.2 Critical Path Analysis for Software-pipelined Prefetching

We perform critical path analysis using Figure 8. The graph is constructed in the same way as Figure 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 D iterations away.

If the paths along the rows are longer, the latency edges can be ignored 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 8. 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. □

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 7(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.

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 of its use of modular index operations and its larger maintained state (such as the waiting queue for read-write conflicts).

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 an output buffer per partition and an input buffer are allocated in main memory. Disk pages from the input relation are streamed through the input buffer. Every input tuple is examined. Its partition number is computed from the join key. The relevant columns of the input tuple are then extracted (projected) and copied to the target output buffer. When an output buffer is full, it is written out.

Clearly, we should employ different prefetching techniques depending on the number of partitions generated. If the number of partitions is small enough so that all the buffers and relevant data structures fit in *cache*, we only need to prefetch for the input page to bring the input data into cache faster after every disk page read. This constitutes our simple prefetching scheme for the partition phase.

When the number of partitions is large, however, there could be cache thrashing during the partition phase; every output buffer visit may incur a cache miss. Similar to the join phase, the processing of a tuple needs to make several dependent memory references, whereas the processings of subsequent tuples are mostly independent due to the randomness of hashing. Therefore, we employ group prefetching and software-pipelined prefetching under this situation.

Note that there are read-write conficts 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.

7 Experimental Results

In this section, we show that hash join is CPU bound through real-machine experiments. We then evaluate the CPU cache performance of our prefetching techniques by simulation.

7.1 Experiment Setup

Implementation Details. We have implemented our own hash join engine. For real machine experiments, we implemented a buffer manager that stripes pages across multiple disks and performs I/O prefetching with background worker threads. For CPU performance simulation studies, we store relations and intermediate partitions as disk files for simplicity. We employ 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 by the hash join algorithms to compute numbers of partitions and hash table sizes.

Our baseline algorithm is the GRACE hash join algorithm [27]. The in-memory hash table structure follows Figure 2 in Section 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. Since the same hash codes are used in both phases, computing them twice could be expensive in computational overhead and in memory access overhead to read join keys. Therefore, we made a simple optimization: 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 since 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 tries straightforward ways to employ prefetching, such as prefetching an entire input page after a disk read. We use simple prefetching as an enhanced baseline and show how much more benefit we can achieve by using more sophiscated prefetching schemes. Prefetch instructions are inserted into C++ source codes with gcc inline ASM macros.

Cache Partitioning. Cache partitioning generates cache-sized build partitions so that every build partition and its hash table can fit in cache and cache misses in the join phase can be greatly reduced. It has been shown to be effective in main-memory and memory-resident database environments [5, 28].

We have implemented two cache partitioning algorithms for disk-oriented databases, which will be compared with our prefetching techniques in Section 7.5. First, we can increase the number of partitions and generate cache-sized partitions directly in the I/O partition phase. Second, we partition twice: the I/O partition phase generates memory-sized partitions, and then they are partitioned again in memory as a preprocessing step for the join phase. We call the first scheme "direct cache" and the second "two-step cache".

Experiment Design. In all our experiments (except for Figure 14(a)), we assume the available memory size for the join phase is 50MB [7]. The partition phase produces partitions to fully utilize the available memory. That is, in the baseline and our prefetching schemes, a build partition and its hash table fit tightly in the memory. In the cache partitioning schemes, 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 to show the benefits of our solutions in various situations.

Measurement Methodology. We first measure GRACE hash join performance on a real 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.

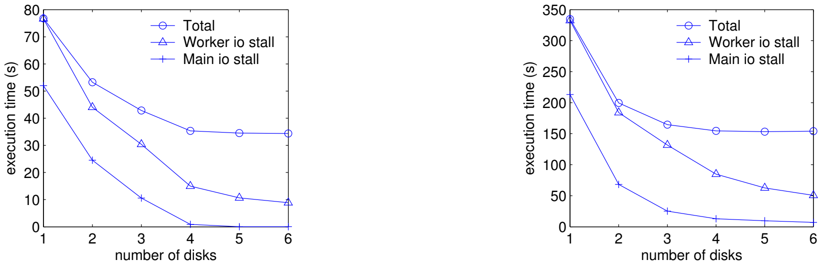
We then evaluate the CPU cache performance (of user mode executions) of all the schemes through simulation in order to get good prefetching support. We generate fully-functional executables with gcc and run the programs with detailed cycle-by-cycle simulations. The simulator models a dynamically-scheduled, superscalar processor running at a clock rate of 1 GHz. The memory hierarchy is based on the Compaq ES40 [8]. Since Alpha processor only supports software-simulated integer divide, we use integer divide latency from the Intel Pentium4 [11]. The simulator does not drop prefetches when miss handlers are all busy. Moreover, the simulator supports TLB prefetching [26] by treating

[7] This is the memory allocated for joining a pair of build and probe partitions. The memory size is limited by the simulation environment. However, the memory to cache size ratio is set to be 50:1, which is reasonable for hash joins on a modern computer system. Therefore, we expect the experiments to reflect real-world hash join cache behaviors.

Table 2: Simulation parameters

Processor pipeline parameters	
Clock Rate	1 GHz
Issue Width	4 insts/cycle
Functional Units	2 Integer, 1 integer divide, 2 Memory, 1 Branch, 2 FP
Reorder Buffer Size	128 insts
Integer Multiply/Divide	15/56 cycles
All Other Integer	1 cycle
Branch Prediction Scheme	gshare [20]

Memory parameters	
Line Size	64 bytes
Primary Instruction Cache	64 KB, 2-way set-assoc.
Primary Data Cache	64 KB, 4-way set-assoc.
Miss Handlers	32 for data, 2 for inst.
DTLB	64 entries, fully-assoc.
DTLB Miss Handlers	1
Page Size	8 KB
Unified Secondary Cache	1 MB, 4-way set-assoc.
Primary-to-Secondary Miss Latency	15 cycles (plus any delays caused by contention)
DTLB Miss Latency	20 cycles
Primary-to-Memory Miss Latency	150 cycles (plus any delays caused by contention)
Main Memory Bandwidth	1 access per 10 cycles



(a) Partition Performance

(b) Join Performance

Figure 9: Hash join is CPU-bound with reasonable I/O bandwidth

TLB misses caused by prefetches as normal TLB misses. Therefore, our prefetching schemes can overlap TLB miss latencies with computations and cache misses. Important simulator parameters are shown in Table 2.

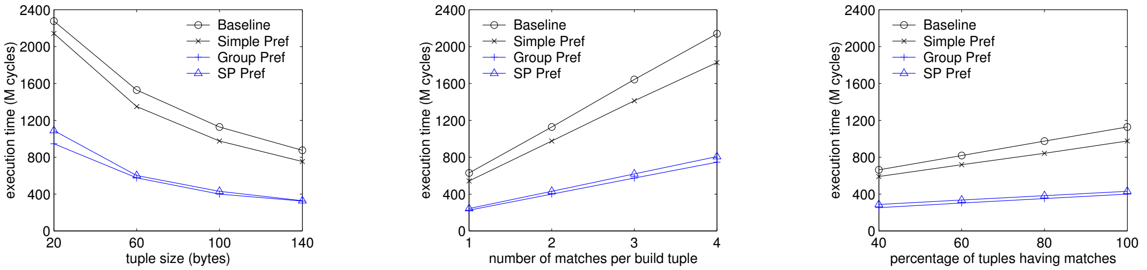
7.2 Is Hash Join I/O-Bound or CPU-Bound?

Figure 9 shows the performance of GRACE hash join on a machine running Linux 2.4.18 with four 550MHz PentiumIII processors and 512MB RAM. Our experiments use 6 Seagate Cheetah X15 36LP SCSI disks, each with a maximum transfer rate of 68 MBytes/sec. We imitate raw disk partitions by allocating a large file on each disk and managing the mapping from page IDs to file offsets ourselves. To get good I/O performance, we stripe a relation across all the disks with 256KB units. Our buffer manager has a dedicated worker thread for each of the disks, which performs I/O operations on behalf of the main hash join thread. The buffer manager implements I/O prefetching and background writing so that I/O operations can be overlapped with computations as much as possible. We measure total elapsed times with gettimeofday() system call and I/O stall times with processor cycle counter and PAPI package. All the reported points are the average of 10 measurements with standard deviations less than 10% of the averages or less than 1 second.

The experiments join a 1.5GB build relation with a 3GB probe relation by producing 31 intermediate partitions for both relations. Tuples are all 100 bytes long with 4 byte join keys. Figure 9(a) shows the partition phase performance for the build relation, and Figure 9(b) shows the join phase performance of joining all the build and probe partitions.

The figures vary the number of disks used and report the total elapsed time for the operations, the maximum I/O stall time of all the background worker threads, and the stall time of the main thread waiting for workers. The worker I/O stall time shows the time to finish all the I/Os in background, which decreases dramatically as the number of disks increases. With four or more disks, hash join is clearly CPU-bound; the total elapsed time becomes flat, and the main thread spends less than 10% of the total time waiting for worker threads. Since there are typically 10 disks per processor on a balanced DB server, we expect that hash join is CPU-bound in a large number of real-world systems.

Moreover, the gap between the top and the middle curves leaves large room for reducing the total time by improving CPU performance, for example, room for 3-fold potential improvement when there are 6 disks.



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

Figure 10: Join phase performance

7.3 Join Phase Performance

Figure 10 shows the join phase performance of the baseline and the prefetching schemes while varying the tuple size, the number of probe tuples matching a build tuple, 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. The three sets of experiments share a pivot point: tuples are 100B long and every build tuple matches two probe tuples. As shown in the figure, group prefetching and software pipelined prefetching achieve 2.4-2.9X and 2.1-2.7X speedups over the baseline algorithm, respectively. Since the central part of the join phase algorithm is hash table visit, simple prefetching only obtains marginal benefit, a 1.1-1.2X speedup over the baseline. By exploring the inter-tuple parallelism, group and software-pipelined prefetching achieve additional 2.3-2.5X and 2.0-2.3X speedups over the simple prefetching scheme, respectively.

In Figure 10(a), as we increase the tuple size, the number of tuples processed decreases, leading to the decreasing trend of the curves. In Figure 10(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 10(b), contributing to the much steeper curves than those in Figure 10(c).

Figure 11 shows the execution time breakdowns for Figure 10(a) when tuples are 100B long. The baseline case is shown as the "join" bar in Figure 1. Group prefetching and software pipelined prefetching indeed successfully hide most of the data cache miss latencies. The simulator outputs confirm that the remaining cache misses are mostly L1 cache misses (but L2 hits) due to conflicts. The (transformation, book keeping, and prefetching) overheads of the techniques lead to larger portions of busy times. Software-pipelined prefetching is more costly than group prefetching. Interestingly, other stalls also increase. A possible reason is that some secondary causes of stalls show up when the data cache stalls are reduced.

Figure 12 shows the relationship between the cache performance and the parameters of the prefetching algorithms. We perform the same experiment as in Figure 10(a) when tuple size is 20B. We only show the tuning results for the probing loop here but the curves for the building loop have similar shapes. The experiments in Figure 10 all use the optimal parameters: $G = 19$ and $D = 1$ for probing.

The top curves in Figure 12 show the performance when the memory latency T is set to 1000 cycles in the simulator. The optimal points shift right; larger group size and prefetch distance are needed to hide the increased latencies, as expected by our models. Interestingly, software-pipelined prefetching becomes better than group prefetching. More importantly, software-pipelined prefetching achieves similar performance when we change T from 150 to 1000 cycles. This means that the prefetching algorithm will still keep up when the processor/memory speed gap increases even more (6 times in our experiments) as expected to happen in the future by the technology trend.

The curves all have concave shapes; performance becomes worse when the parameters are too small or too 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 when the parameters are small. To verify this and to understand the performance with large parameters, we analyze the breakdowns of cache misses in Figure 13. The bars correspond to the optimal

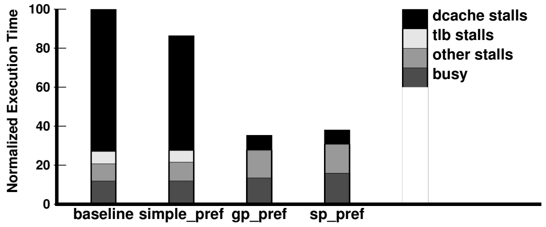
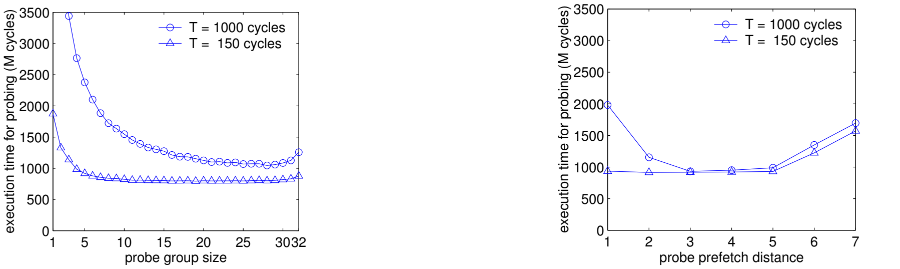


Figure 11: Execution time breakdown for join phase performance (Figure 10(a), 100B tuples)



(a) group prefetching

(b) software-pipelined prefetching

Figure 12: Tuning parameters of group prefetching and software pipelined prefetching for join phase

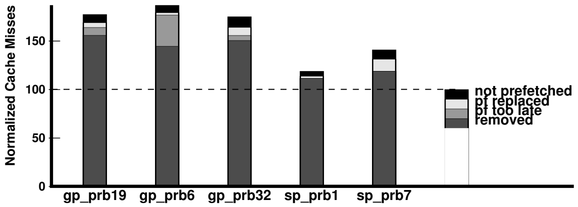


Figure 13: Breakdowns of cache misses to understand the tuning curves of join phase

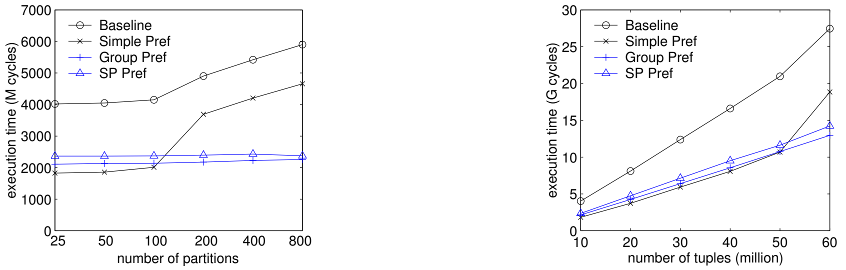
points, too-small points, and too-large points from the $T = 150$ curves in Figure 12.

The bars are normalized to the number of cache misses in the baseline GRACE case (100 in the figure). The total heights of the bars correspond to the number of cache misses when we apply the code transformations but without inserting any prefetches. When we insert the prefetches, a lot of cache misses disappear, which are captured by the bottom part of the bars. The other categories are i) "pf too late", i.e. prefetching is too late to hide all the latency; ii) "pf replaced", i.e. prefetching is too early and the prefetched cache line has already been replaced from the cache by other memory references or prefetches; iii) "not prefetched".

From Figure 13, the too-small case shows a large "pf too late" portion. This confirms the above analysis. In the too-large cases, a lot of prefetches have been replaced, meaning that the poor performance is caused by cache conflicts. The larger the parameters, the more prefetches and other memory references are executed between a prefetch and its visit instruction, and therefore the larger chance that a prefetch is replaced from cache.

7.4 Partition Phase Performance

Figure 14(a) shows the partition phase performance varying the number of partitions from 25 to 800. The source relation has 10 million 100-byte tuples. The figure is divided into two different regions. When partition number is 25, 50, and 100, simple prefetching achieves the best performance. However, when the number of partitions becomes larger, the performance of simple prefetching deteriorates dramatically, while group prefetching and software-



(a) varing number of partitions

(b) varying relation size

Figure 14: Partition phase performance

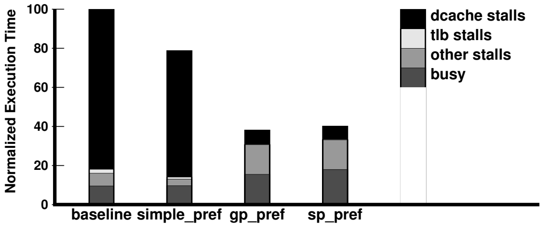
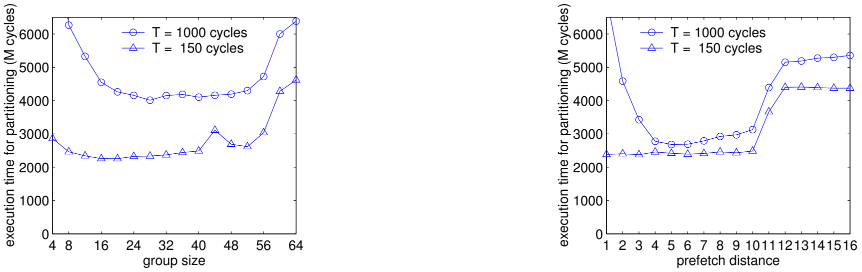


Figure 15: Execution time breakdown for partition phase performance (Figure 14(a), 800 partitions)



(a) group prefetching

(b) software-pipelined prefetching

Figure 16: Tuning parameters of group prefetching and software pipelined prefetching for partition phase

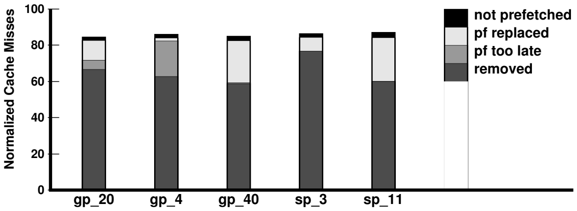


Figure 17: Breakdowns of cache misses to understand the tuning curves of partition phase

pipelined prefetching win. Since 1MB L2 cache can hold 128 pages of 8KB each, the output buffers (and other miscelleous data structures) fit in L2 cache in the left region, in which more sophisticated prefetchings with larger overhead are not necessary. However, when the output buffers can not fit in cache as in the right region, simple prefetching suffers from excessive cache misses. Group prefetching and software-pipelined prefetching exploit inter-

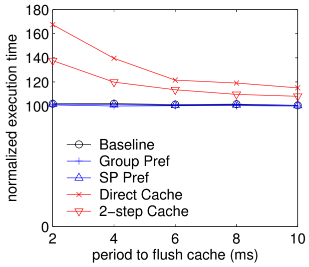
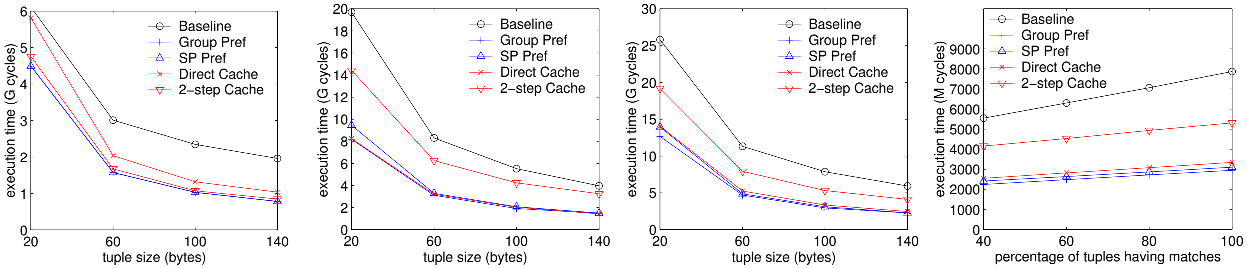


Figure 18: Impact of cache flushing on the different techniques.



(a) I/O partition phase of (c) using combined prefetching

(b) Join phase of (c)

(c) Overall performance (varying tuple size)

(d) Overall performance (varying join selectivity)

Figure 19: Comparisons with cache partitioning when it applies

tuple parallelism to successfully hide most of the cache miss latencies.

Figure 14(b) varies the number of 100B tuples in the input relation while keeping the partition size the same (to fit tightly in 50MB memory). Hence the number of partitions also increases; 26, 51, 76, 102, 127, and 152 partitions are generated, respectively. Essentially the graph shows the same tradeoff as Figure 14(a) but in a more natural setting.

To get the best of both situations, we choose the prefetching algorithm based on the cache size and the number of partitions. Overall, this combined prefetching achieves 1.9-2.6X speedups over the baseline.

Figure 15 shows the execution time breakdown for Figure 14(a) where 800 partitions are generated. Group prefetching and software pipelined prefetching successfully hide most of the data cache miss latencies. Figure 16 shows the relationships between parameters and the cache performance of group prefetching and software-pipelined prefetching. Figure 17 analyzes the reasons for the poor performance when parameters are too small or too large. We see similar curve shapes and trends as in the join phase.

7.5 Comparison with Cache Partitioning

Problems with Large Relations. 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 [16]. Given a 1 MB CPU cache and (optimistically) 1000 partitions, the maximum relation size for "direct cache" is only 1 GB. "Two-step cache" solves this problem by introducing an additional partition pass. However, this additional copying cost results in 50-150% slowdown compared to our prefetching schemes.

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 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 cache from 10ms to 2ms in our simulator. "100" corresponds to the join phase execution time when there is no cache flush. *Direct cache* and *2-step cache* suffer from 15-67% and 8-38% performance degradation, respectively. The reason that *2-step cache* suffers from less degradation is that cache flushes may occur during in-memory partition operations and be less harmful. 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 in cache. As shown in the figure, they are very robust against even cache flushes.

Experiments when "direct cache" also applies. Figure 19 compares our prefetching schemes with cache partitioning. Note that the I/O partition phases of all schemes use the combined prefetching scheme discussed above; the major differences of the curves are the different numbers of I/O partitions generated. The second partition step in "two-step cache" is shown as part of the join phase performance. Moreover, we employ prefetching in the join phase to enhance the cache partitioning schemes wherever possible.

Figure 19(a)-(c) show experiments joining a 200MB build relation with a 400MB probe relation. Every build tuple matches two probe tuples. We increase the tuple size, which results in decreasing number of tuples in the relations and the downward trends of the curves. "Direct cache" achieves the best performance in the join phase by avoiding most cache misses. However, it suffers from larger overheads in the partition phase for generating much more partitions. "Two-step cache" suffers from the overhead of the additional partition step and is 50-150% worse than the prefetching schemes. Overall, our prefetching schemes are the best (slightly better than "direct cache"). In Figure 19(d), we keep the tuple size to be 100B and vary the percentage of tuples that have matches. Again, we see similar trends as in Figure 19(c). In Figure 19, our prefetching techniques achieve 1.4-2.5X speedups for the partition phase, 2.1-2.9X speedups for the join phase, and 1.9-2.7X speedups overall compared to the baseline algorithm.[8]

8 Conclusions

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 paper, 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 results demonstrated that hash join cache performance can be improved dramatically through prefetching. More interestingly, the techniques will still be effective even when the speed gap between processors and memory increases significantly in the future (e.g., by a factor of six). 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.

Acknowledgments

The third author thanks P. Bohannon, S. Ganguly, H. F. Korth, and P. P. S. Narayan for helpful discussions.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th VLDB*, pages 266–277, Sept. 1999.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.

[3] L. A. Barroso, K. Gharachorloo, and E. D. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th ISCA*, pages 3–14, June 1998.

[8] The experiments shown in the prefetching curves generate the same number partitions as in the baseline curve.

[4] P. Bohannon, P. McIlroy, and R. Rastogi. Improving Main-Memory Index Performance with Partial Key Information. In *Proceedings of the SIGMOD 2001 Conference*, May 2001.

[5] P. A. Boncz, S. Manegold, and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th VLDB*, pages 54–65, Sept. 1999.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance through Prefetching. In *Proceedings of the SIGMOD 2001 Conference*, pages 235–246, May 2001.

[7] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B^+-Trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the SIGMOD 2002 Conference*, June 2002.

[8] Z. Cvetanovic and R. E. Kessler. Performance Analysis of the Alpha 21264-Based Compaq ES40 System. In *Proceedings of the 27th ISCA*, pages 192–202, June 2000.

[9] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *the 26th ISCA*, May 1999.

[10] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[11] Intel Corp. *Intel Pentium4 and Intel Xeon Processor Optimization*. Order Number: 248966-007.

[12] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 25th ISCA*, pages 15–26, June 1998.

[13] K. Kennedy and K. McKinley. Loop Distribution With Arbitrary Control Flow. In *Proceedings of the 1990 Conference on Supercomputing*, pages 407–416, 1990.

[14] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing*, 1(1):63–74, 1983.

[15] M. S. Lam. *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon University, May 1987.

[16] B. Lindsay. Hash Joins in DB2 UDB: the Inside Story. *Carnegie Mellon DB Seminar*, Mar 2002.

[17] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th ASPLOS*, pages 222–233, Oct. 1996.

[18] C.-K. Luk and T. C. Mowry. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Transactions on Computers*, 48(2):134–141, Feb. 1999.

[19] S. Manegold, P. A. Boncz, and M. L. Kersten. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Proceedings of the 26th VLDB*, Sept. 2000.

[20] S. McFarling. Combining Branch Predictors. Technical Report WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[21] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Mar. 1994.

[22] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th ASPLOS*, pages 62–73, Oct. 1992.

[23] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the 14th VLDB*, pages 468–478, Aug. 1988.

[24] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the 1994 SIGMOD Conference*, pages 233–242, May 1994.

[25] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the SIGMOD 2000 Conference*, pages 475–486, May 2000.

[26] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *Proceedings of the 27th ISCA*, pages 117–127, May 2000.

[27] L. D Shapiro. Join Processing in Database Systems with Large Main Memories. *TODS*, 11(3):239–264, 1986.

[28] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th VLDB*, pages 510–521, Sept. 1994.

[29] H. Zeller and J. Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In *Proceedings of the 16th VLDB*, pages 186–197, Aug. 1990.