Improving Hash Join Performance through Prefetching

SHIMIN CHEN
Intel Research Pittsburgh
ANASTASSIA AILAMAKI
Carnegie Mellon University
PHILLIP B. GIBBONS
Intel Research Pittsburgh
TODD C. MOWRY
Carnegie Mellon University and Intel Research Pittsburgh

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 80% of its user time stalled on CPU cache misses, and explores the use of CPU cache *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 1.29–4.04X speedups for the join phase and 1.37–3.49X speedups for the partition phase over GRACE and simple prefetching approaches. Moreover, compared with previous cache-aware approaches (i.e. cache partitioning), the schemes are at least 36% faster on large relations and do not require exclusive use of the CPU cache to be effective. Finally, comparing the elapsed real times when disk I/Os are in the picture, our cache prefetching schemes achieve 1.12-1.84X speedups for the join phase and 1.06-1.60X speedups for the partition phase over the GRACE hash join algorithm.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; D.1.m [**Programming Techniques**]: Miscellaneous

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Hash join, CPU cache performance, CPU cache prefetching, Group prefetching, Software-pipelined prefetching

1. INTRODUCTION

Hash join [Kitsuregawa et al. 1983; Shapiro 1986; Nakayama et al. 1988; Zeller and Gray 1990] 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

An earlier version [Chen et al. 2004] of this paper appeared in the *20th International Conference on Data Engineering (ICDE 2004)*. This paper mainly expands on the following aspects: (i) Section 8.3 and Section 8.4 are newly added that report both user-mode performance and elapsed times with disk I/Os on an Itanium 2 machine, while only user-mode simulation results were shown in the previous version; (ii) Section 4.3 and Section 5.2 are newly added to use critical path analyses to prove Conditions (1) and (2); (iii) Simulation results in Section 8.2 are updated with up-to-date simulation parameters; (iv) Section 8.1 is completely rewritten with results obtained on the Itanium 2 machine; (v) Section 9 is newly added to discuss practical implementation issues for DBMSs.

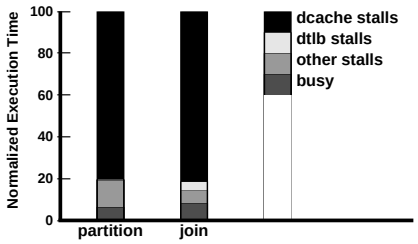


Fig. 1. User-mode execution time breakdown for hash join.

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 [Kitsuregawa et al. 1983] 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 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 8.1.

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 [Shatdal et al. 1994; Boncz et al. 1999; Manegold et al. 2000]. 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 1 provides a breakdown of the simulated user-level performance on our simulation platform (which will be described in detail in Section 7 and Section 8.2). 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 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 [Shatdal et al. 1994; Boncz et al. 1999; Manegold et al. 2000]. 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 [Lindsay 2002]. 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 8, 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% as shown in Section 8.2). Hence, we 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 exploit cache prefetching to *hide* the cache miss latency associated with accessing normal (memory-sized) hash tables, by overlapping these cache misses with computation. Modern processors allow multiple cache misses to be in flight simultaneously in the memory hierarchy (e.g., 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 [Intel Corporation 2004]), 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 [Mowry et al. 1992] and pointer-based programs [Luk and Mowry 1996], but it has not been applied to hash joins.

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* [Luk and Mowry 1996] 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 [Baer and Chen 1991] 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, although this approach is useful for hiding the latency of primary data cache misses that hit in the secondary cache, the instruction window size (typically 64-128 entries) is often a magnitude smaller than the instructions wasted due to a cache miss to main memory (e.g. 200-300 cycle latency multiplied by 4-6 instruction issue slots per cycle), 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 [Luk and Mowry 1996; Mowry et al. 1992] cannot handle the ambiguous data dependencies present in the hash join code (as will be 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* [Lam 1987] 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 [Lam 1987]).

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, for user-mode performance, the techniques achieve 1.29–4.04X speedups for the join phase and 1.37–3.49X speedups for the partition phase over GRACE and simple prefetching approaches. Moreover, they are at least 36% faster than cache partitioning on large relations and do

[1] In our experimental set-up in Section 8, $G = 25$ is optimal for hash table probing.

not require exclusive use of the cache to be effective. Furthermore, we make extensive comparisons between group prefetching and software-pipelined prefetching, demonstrating that group prefetching is up to 30% faster than software-pipelined prefetching. Finally, we present experiments measuring elapsed real times with disk I/Os and demonstrate the effectiveness of our cache prefetching schemes for disk-oriented hash joins: our schemes achieve 1.12-1.84X speedups for the join phase and 1.06-1.60X speedups for the partition phase over the GRACE hash join algorithm.

The rest of the paper is organized as follows. Section 2 discusses the related work in more detail. Section 3 analyzes the dependencies in the join phase, the more complicated of the two phases, while Sections 4 and 5 propose group prefetching and software-pipelined prefetching to improve the join phase performance. Section 6 discusses prefetching for the partition phase. Section 7 describes experimental setups and methodologies. Experimental results appear in Section 8. Section 9 discusses practical issues in implementing the prefetching techniques in DBMSs. Finally, Section 10 concludes the paper.

2. RELATED WORK

Since the GRACE hash join algorithm was first introduced [Kitsuregawa et al. 1983], 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 [Shapiro 1986; Nakayama et al. 1988; Zeller and Gray 1990; Graefe 1993]. 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 paper. 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 [Shatdal et al. 1994; Boncz et al. 1999; Manegold et al. 2000]. Shatdal *et al.* showed that cache partitioning achieved 6-10% improvement for joining memory-resident relations with 100B tuples [Shatdal et al. 1994]. Boncz, Manegold and Kersten proposed using multiple passes in cache partitioning to avoid cache and TLB thrashing [Boncz et al. 1999; Manegold et al. 2000]. 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., shift-based hash computation) 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 [Mowry et al. 1992; Luk and Mowry 1996; Chen et al. 2001; Chen et al. 2002]. While software pipelining has been used to schedule prefetches in array-based programs [Mowry et al. 1992], 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 showed that TLB misses may degrade performance [Boncz et al. 1999; Manegold et al. 2000], particularly when they are handled by software. However, the majority of modern processors (e.g. x86 [Intel Corporation 2006] and Itanium 2 [Intel Corporation 2004]) handle TLB misses in hardware. Moreover, TLB prefetching [Saulsbury et al. 2000] can be supported by treating TLB misses caused by prefetches as normal TLB misses. For example, faulting prefetch instructions on Itanium 2 (*lfetch.fault* [Intel

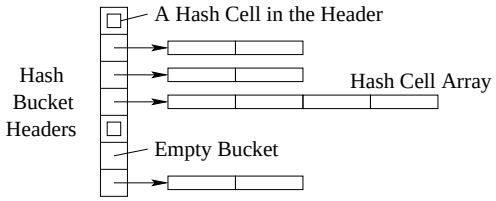


Fig. 2. An in-memory hash table structure.

Corporation 2004]) 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. (Section 9 discusses TLB misses and other practical issues in more detail).

After the conference version of this paper [Chen et al. 2004] was published, several recent studies have adopted our schemes of exploiting inter-tuple parallelism to improve hash join performance. Gold *et al.* [Gold et al. 2005] implemented hash join probe operations on the Intel IXP2400 network processor and demonstrated that using multiple hardware threads and cores to exploit inter-tuple parallelism can achieve up to 5X speedups over a single thread, and 2.5X speedups over a baseline implementation on Pentium 4. Zhou *et al.* [Zhou et al. 2005] proposed and evaluated a scheme to use helper hardware threads to perform faulting prefetches for improving database performance. Their hash join implementation was based on our schemes to exploit inter-tuple parallelism. Chen *et al.* [Chen et al. 2005] proposed an inspector join algorithm to achieve the benefits of cache partitioning with almost zero re-partitioning overhead. They exploited our prefetching schemes in the partition phase and used the schemes to avoid conflict misses in the join phase.

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 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 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. Therefore, addresses would be generated

too late for prefetching to hide miss latencies. In addition, predicting memory addresses for hash table visits is almost impossible because of the randomness of hashing. Note that these arguments hold for all hash-based structures.[2] 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 issue prefetches early enough. In this section, we propose group prefetching to achieve these goals.

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[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.

4.2 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 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 naturally 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.

[2]The structure in Figure 2 improves upon chained bucket hashing, which uses a linked list of hash cells in a bucket. Here, we avoid the pointer chasing problem of linked lists [Luk and Mowry 1999; Chen et al. 2001].

[3]Technically, what we do are modified forms of compiler transformations called *strip-mining* and *loop distribution* [Kennedy and McKinley 1990].

```
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

Fig. 3. Group prefetching.

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 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 I shows the terminology used throughout the paper.) 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:

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

The proof of this condition is given 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

Table I. Terminology used throughout the 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_1	full latency of a cache miss
T_{next}	latency of an additional pipelined cache miss
C_l	execution time for code l, $l = 0, 1, \ldots, k$

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 8 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. For P vertices, their superscripts correspond to the memory addresses in the program; for C and VC vertices, the superscripts are the code stage. 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, there is an instruction flow edge from vertex C_j^0 (code 0 for element j) to vertex P_j^1 (prefetch for m_j^1) with weight C_0; edge P_j^1 to C_{j+1}^0 means that

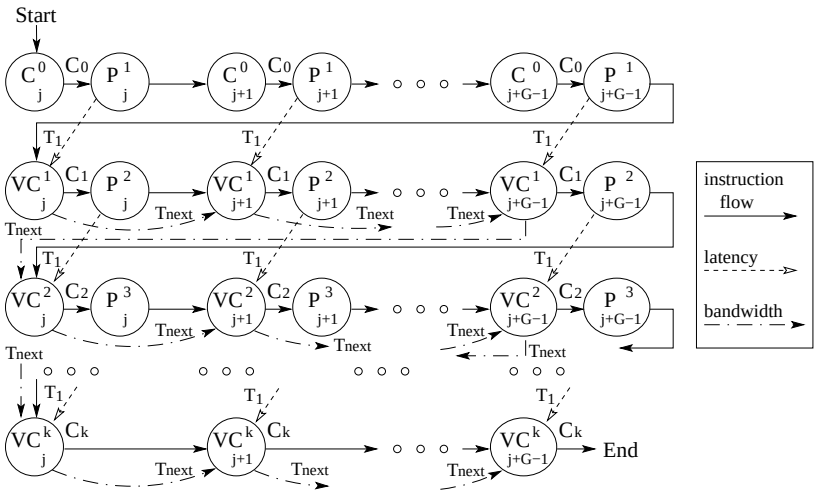


Fig. 4. Critical path analysis for an iteration of the outer loop body in Figure 3(d).

prefetch for m_j^1 is executed before code 0 for element j+1; and 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_1 as its weight.

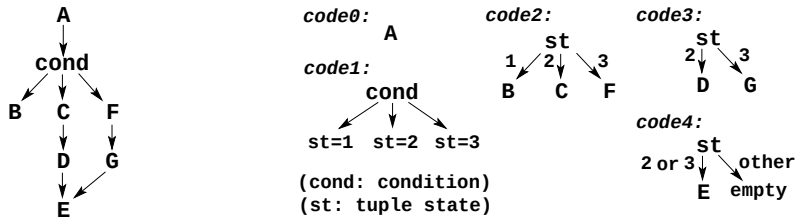
—**Bandwidth edges**: an edge between VC vertices represents memory bandwidth. Usually an additional (independent) cache miss can not be fully overlapped with the previous one. It takes T_{next} more time to finish, which is the inverse of memory bandwidth.

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, and 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:*

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



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

Fig. 5. Dealing with multiple code paths.

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_{j+x-1}^1 to vertex VC_{j+x-1}^1, the length of the path along the row is as follows:

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

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

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

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 [Mowry 1994]. 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 illustrates the idea of this process. Edges in Figure 5(a) denote the control flow and data dependencies. Figure 5(b) shows the code stages for group

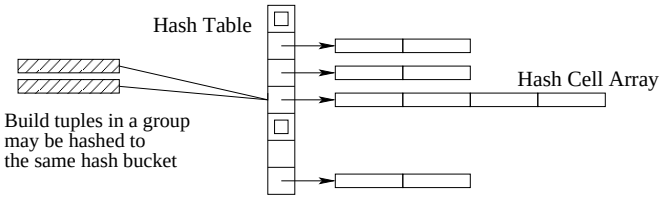


Fig. 6. A read-write conflict.

prefetching. The common starting point of all code paths is in *code 0*. *code 1* performs the conditional test (e.g. checking whether the hash bucket is empty, contains a single entry in the bucket header, or contains a hash cell array), then records the outcome in the state of the tuple. Then subsequent code stages test the tuple state and execute the code pieces for the corresponding code paths. Moreover, *code l* tests the tuple state to determine the code path of *code l+1* and issue the corresponding prefetch instructions. The total number of stages ($k+1$) is the largest number of code pieces (determined by the number of dependent memory accesses) 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 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 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, 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).

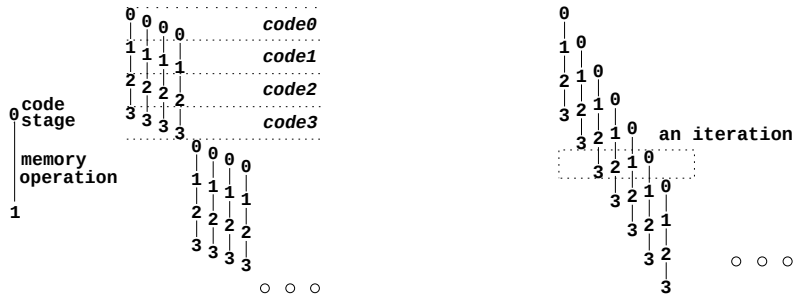
5. SOFTWARE-PIPELINED PREFETCHING

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 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.

5.1 Understanding Software-Pipelined Prefetching

Figure 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* [Mowry 1994].) Figure 7(b) depicts the intuitive picture when $D = 1$.



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

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

Fig. 7. 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

Fig. 8. Software-pipelined prefetching.

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 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 distances 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.)

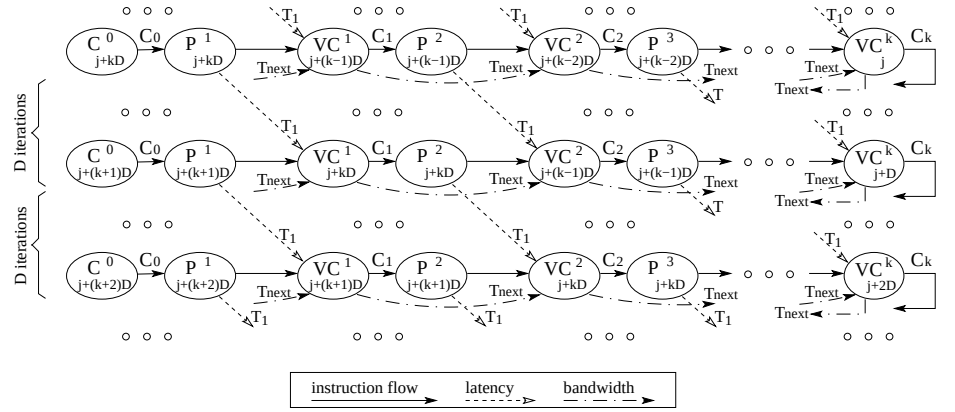


Fig. 9. Critical path analysis for software-pipelined prefetching (steady state).

$$D \cdot (max\{C_0 + C_k, T_{next}\} + \sum_{l=1}^{k-1} \max\{C_l, T_{next}\}) \geq T \quad (2)$$

We can always choose a D sufficiently large to satisfy this condition. In our experiments in Section 8, 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 9. 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 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 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. □

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 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.

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).

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.

Like the join phase, the I/O partitioning phase employs hashing: It computes the partition number 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 may be read-write conflicts in visiting the output buffers. Imagine that two tuples that are close in the input relation happen to be hashed to the same output buffer. 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 at this moment the data from the first tuple has not yet been copied into the output buffer because we have reorganized the processing. To solve this problem, in group prefetching, we delay the processing of the second tuple until the end of the loop body, where we write out the buffer and process the delayed tuple. In software-pipelined prefetching, we use waiting queues similar to those used in hash table building in the join phase.

7. EXPERIMENTAL SETUP

7.1 Measurement Methodology

We evaluate our prefetching schemes through both cycle-accurate simulations and real machine experiments on an Itanium 2 machine. We model a wider range of modern processors with the simulation study. Moreover, simulation allows us to flexibly instrument the processor pipeline and the cache hierarchy to better understand the results. Furthermore, we can model future machine configurations by varying the simulation parameters. We verify the trends learned from the simulation study with user-mode performance results on the Itanium 2 machine. Finally, we measure total elapsed time with disk I/Os on the Itanium 2 machine to show the benefits of our schemes on disk-oriented hash joins.

Itanium 2 Configuration for Real-Machine Experiments. Table II lists the Itanium 2 machine configuration parameters. The machine has two 900MHz Itanium 2 McKinley processors, each with three levels of caches and two levels of TLBs. They share an 8 GB main memory. However, we used only the lower 1GB memory in our experiments (more details in Section 8.1). Most of the cache hierarchy parameters are described in the Itanium 2 manual [Intel Corporation 2004]. We measure the DTLB miss latency, main memory latency (T_1), and main memory bandwidth(T_{next}) through experiments [Chen 2005]. Note that 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 [Intel Corporation 2004].

Itanium 2 supports both faulting and non-faulting prefetches. Non-faulting prefetches are dropped if the prefetches cause exceptions, such as TLB misses. In contrast, a faulting prefetch behaves similar to a load instruction without destination register; upon a TLB miss, it will load the page entry into the TLB table and continue. Since hash table visits are likely to cause TLB misses, we choose faulting prefetches in our experiments. Prefetching can be implemented with the two compilers available on the machine: gcc and icc. gcc supports inserting prefetch instructions as inline assembly code, while icc supports a special function-call-like interface for prefetches. In Section 8.3, we compare the hash join performance using different compilers and optimization levels. The decision is to use "icc -O3" for all the Itanium 2 experiments.

The machine is running Linux 2.4.18 kernel with 16 KB virtual pages. We measure user-mode performance by using the perfmon library [Perfmon Project] to access the Itanium 2 performance counters. We measure the total elapsed times with disk I/Os using the `gettimeofday` system call. We perform 30 runs and report the averages. For the user-mode cache performance measurements, the standard deviations are within 1% of the

Table II. 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. DTLB 2 Miss Latency: 32 cycles
Main Memory	8GB (only the lower 1GB used), Latency (T_1): 189 cycles, Bandwidth ($1/T_{next}$): 1 access per 24 cycles
Disks	8 SCSI Seagate Cheetah 15K ST336754LW disks, 15000 rpm, average seek time: 3.6 ms, average rotational latency: 2 ms
Operating System	Linux 2.4.18 (Red Hat Linux Advanced Workstation release 2.1AW)
Page Size	16KB
Compiler	Intel C++ Itanium Compiler Version 8.1, icc -O3
Measurement	user-mode performance: kernel perfmon version 1.0, pfmon version 2.0 total elapsed time with I/Os: `gettimeofday()`

Table III. Parameters for simulation study.

Pipeline Parameters			
Clock Rate	1.5 GHz	Integer Multiply	4 cycles
Issue Width	4 instructions/cycle	Integer Divide	50 cycles
Reorder Buffer Size	128 instructions	All Other Integer	1 cycle
Branch Prediction	gshare [McFarling 1993]		
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 ($T_{next} = 15$)		

averages in all cases. For the total elapsed real time measurements, the standard deviations are within 5% of the averages in all cases.

Machine Model for Simulation Study. Table III shows the parameters for the simulation study. The simulator models a generic out-of-order super-scalar processor, which is the model in most modern processors (other than Itanium 2), such as Intel Pentium 4 [Intel Corporation 2004], IBM Power 5 [Kalla et al. 2004], and Sun UltraSPARC IV [Sun Microsystems]. It performs a cycle-by-cycle simulation, modeling the rich details of the processor including the pipeline, register renaming, branch prediction, and branching penalties, etc. The simulator supports the MIPS instruction set and executes gcc-generated executables. The simulator simulates only user-mode executions; it delivers system calls such as `read` and `write` directly to the underlying operating system.

Because CPU cache performance is the major factor in hash join user-mode performance and the memory hierarchy of Itanium 2 is representative of modern server processors, we model the memory hierarchy of the Itanium 2 machine in the simulator. Most of the memory parameters (e.g., cache sizes, associativities, cache access latencies) follow the Itanium 2 configuration as described in Table II. Moreover, the simulator supports faulting

prefetches. However, unlike Itanium 2, the simulator only supports a uniform cache line size across all levels of caches. We choose 64 bytes as the cache line size, and adjust the memory latency (T_1) and the memory bandwidth (T_{next}) accordingly.

7.2 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.

GRACE Hash Join. Our baseline algorithm is the GRACE hash join algorithm [Kitsuregawa et al. 1983]. The in-memory hash table structure is described previously in 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.[4] Our algorithms choose the hash table size to be a relative prime to the number of partitions and to be larger than the number of build tuples to be hashed. In this way, a hash bucket typically contains only one or two build tuple entries, leading to minimal search cost within a hash bucket. 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. This optimization is employed in all the schemes we implemented

Prefetching Schemes. 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. We implement simple prefetching as an enhanced baseline in order to show the additional benefit achieved using our more sophisticated prefetching schemes. On the Itanium 2 machine,"icc -O3" enhances a program by automatically (aggressively) inserting prefetches. In fact, we find that the icc generated baseline achieves slightly better performance than the simple prefetching approach. Therefore, we only show simple prefetching curves for the simulation study but 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

[4]We make no assumptions on the join key distribution, and therefore choose the more general modulo division operation rather than setting the hash table size to be a power of 2 and using the bit mask operation [Manegold et al. 2000]. We believe that the latter technique requires certain key distribution (e.g. uniform) for maintaining the quality of hash computation. In general, reducing the cost C of a code stage may shorten the critical path in Figure 4 (Figure 9), which requires larger G (D) and potentially makes our prefetching algorithms faster.

database environments [Shatdal et al. 1994; Boncz et al. 1999]. 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 (using the same hash codes).

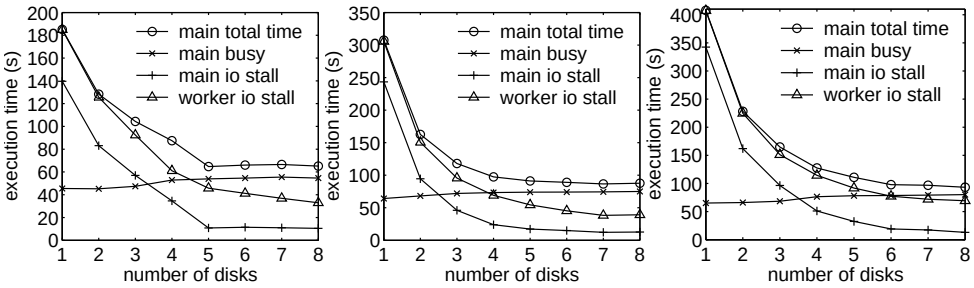
Buffer Manager for Experiments with Disk I/Os. We implemented the buffer pool manager using the POSIX thread (pthread) library. Given a set of disks, the buffer pool manager stripes all the relations across all of the disks in 256KB stripe units;[5] and it imitates raw disk partitions by allocating a large file on each disk and managing the mapping between `pageID`s and file offsets. For each disk, the buffer manager maintains a request queue and runs a dedicated worker thread. To perform an I/O operation with page `pageID`, the main thread computes the target disk ID *i* and the disk file offset from the `pageID`. It then appends a request into the the *i*-th request queue. If the I/O operation is synchronous, the main thread blocks till it receives a completion notification from the worker thread *i*. For prefetch requests (e.g. reading the next pages in the input relations for hash joins), the main thread continues without waiting for the request to complete. The worker thread, however, performs a synchronous I/O read and blocks on behalf of the main thread in the background. Later when the main thread attempts to access 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. For asynchronous I/O write operations, worker threads perform background writing on behalf of the main thread. Moreover, worker threads call `fdatasync` periodically (every 128 write operations in our experiments) to flush any pages that may be cached in the file system cache.

7.3 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 [6]. 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. We join a 2GB build relation with a 4GB probe relation in the Itanium 2 experiments, while we join a 200MB build relation with a 400MB probe relation in the simulation study (which is limited by simulation time). In our

[5]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 [IBM Corporation 2004]. 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.
[6]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.



(a) Partition phase performance (b) Join phase performance (outputs consumed in memory) (c) Join phase performance (outputs written to disk)

Fig. 10. Hash join is CPU-bound with reasonable I/O bandwidth.

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.

8. PERFORMANCE EVALUATION

In this section, we present experimental results to quantify the benefits of our cache prefetching techniques. We begin by showing that hash join is CPU bound with reasonable I/O bandwidth. Next, we study the user-mode CPU cache performance of hash joins through both simulations and Itanium 2 experiments. Finally, we evaluate the impact of our cache prefetching techniques on the elapsed real times of hash joins with disk I/Os.

8.1 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 the Itanium 2 machine using up to 8 SCSI disks. We use the multi-threaded buffer manager as described in Section 7.2.

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 by performing the following three operations: (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 dummy files from each of the eight disks.

Figure 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. Depending on the queries, the join output may either be written to disk or consumed in main memory by the parent operator; we perform experiments to evaluate both cases. 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 approximates the time to do hash join work in memory. The *worker io stall* time is the largest I/O stall time of individual worker threads.

As shown in Figure 10, the *worker io stall* time decreases dramatically as the number of disks increases and the number of I/O operations per disk decreases. 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 only I/O-bound when the number of disks is small (e.g. ≤ 4 disks). As more and more disks are added, hash joins gradually become CPU-bound.

As shown in Figure 10(a) and Figure 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 10(c), the join phase with outputs written to disk becomes CPU-bound when seven disks are used.[7] 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 [TPC Benchmarks]. 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. Section 8.4 will show that our prefetching schemes can reduce the elapsed total time of both partition and join phases with disk I/Os.

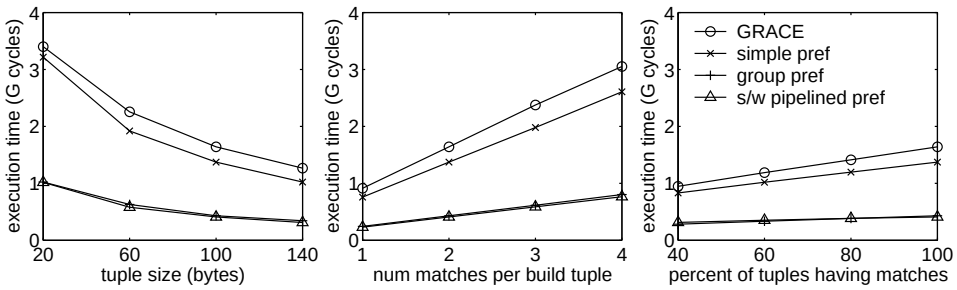
8.2 User-Mode CPU Cache Performance through Simulations

Join Phase Performance. We compare the join phase performance of the baseline algorithm and the three prefetching schemes through simulations in Figure 11. 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 Figure 11, while varying the tuple size, the ratio of probe relation size to build relation size, and the percentage of tuples that have matches, group and software-pipelined prefetching achieve 3.02-4.04X speedups over the GRACE hash join. On the other hand, simple prefetching only obtains a 1.06-1.24X speedup over the baseline, because it does not improve the central part of the join phase algorithm—hash table visiting. Compared with simple prefetching, group and software-pipelined prefetching achieve additional 2.65-3.40X speedups.

The curve trends of the sub-figures are expected. In Figure 11(a), as the tuple size increases from 20 to 140 bytes, the number of tuples in the fixed sized partition decreases, leading to the decreasing trend. In Figure 11(b) and (c), the total number of matches increases as 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 11(b), contributing to the much steeper curves than those in Figure 11(c).

Join Phase Execution Time Breakdowns. We show the execution time breakdowns in Figure 12. Each bar is broken down into four categories that explain what happened

[7]Although the curve markers seem to overlap, this claim is supported by experimental results in Section 8.4, which demonstrate that cache prefetching improves the performance in this case.



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

Fig. 11. Join phase user-mode performance through simulations (joining a pair of build and probe partitions).

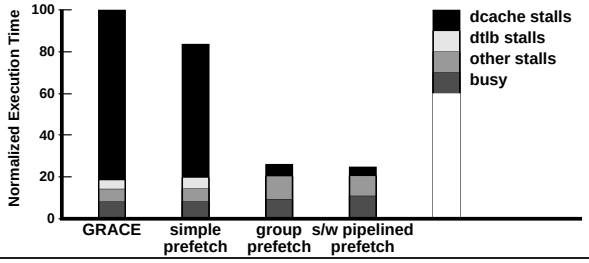


Fig. 12. Execution time breakdown for join phase performance (Figure 11(a), 100B tuples).

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 actually 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 stalled instruction suffering a data cache miss, the second section (*dtlb*) is the number of slots that are caused by the oldest stalled 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 often difficult to generate such accurate cache performance breakdowns on a real 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.

Figure 12 corresponds to the 100-byte points in Figure 11(a). The GRACE bar is shown as the "join" bar previously in Figure 1. We see that group prefetching and software-

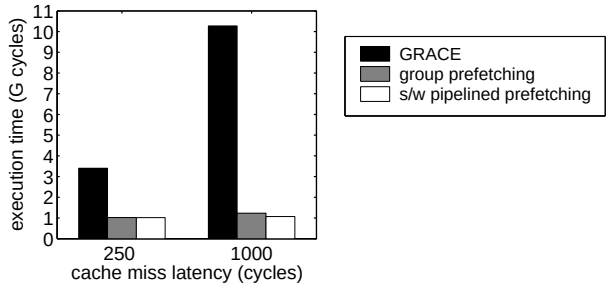


Fig. 13. Join phase user-mode performance varying memory latency. (The 250-cycle results are the same as the 20B results in Figure 11(a).)

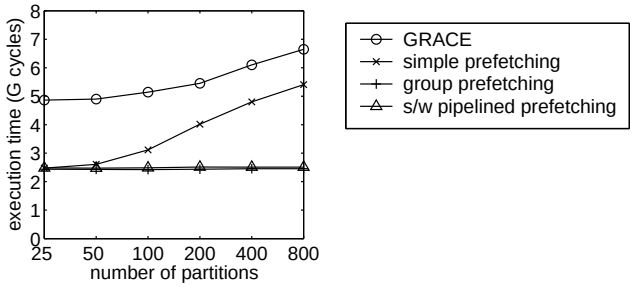


Fig. 14. Partition phase user-mode performance through simulations.

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 Memory Latency. Figure 13 shows the join phase performance when the memory latency T_1 is set to 250 cycles (default value) and 1000 cycles in the simulator. We see that 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 increase only slightly, thus achieving 8.3-9.6X speedups over GRACE hash join. We conclude that the prefetching algorithms are effective even when the processor/memory speed gap increases dramatically (e.g. by a factor of four).

Partition Phase Performance. Figure 14 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 fix the tuple size as 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

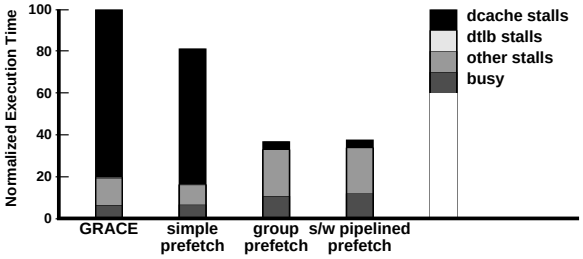


Fig. 15. Execution time breakdown for Figure 14 with 800 partitions.

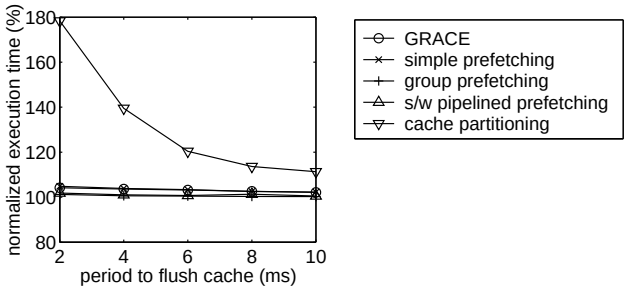


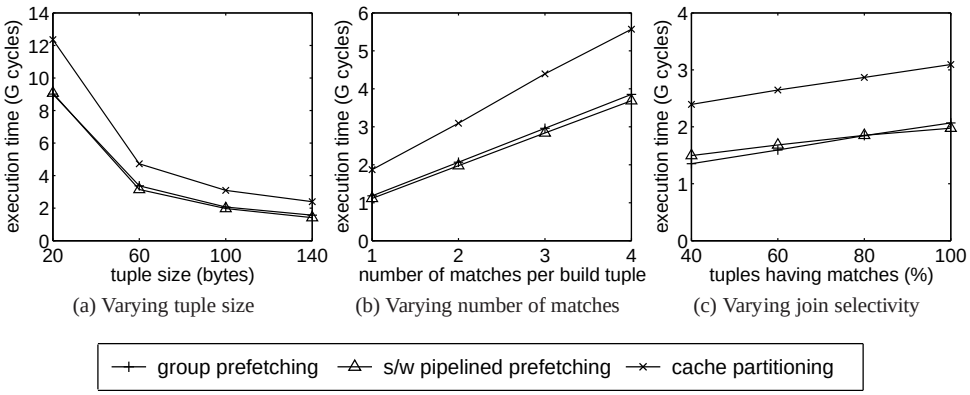
Fig. 16. Impact of cache flushing on the different techniques.

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 GRACE, our prefetching schemes achieve 1.96-2.71X speedups.

Figure 15 shows the execution time breakdown for Figure 14 where 800 partitions are generated. Group prefetching and software-pipelined prefetching successfully hide most of the data cache miss latencies. Similar to Figure 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.

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. In Figure 16, we show the performance degradation of all the schemes when the cache is periodically flushed, modeling the worst case interference. We vary the period to flush the cache from 2 ms to 10 ms, and report the execution time normalized to the performance when running an algorithm without cache flushes. As shown in Figure 16, 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.

Comparison with Cache Partitioning: Re-Partitioning Cost. The number of I/O partitions is upper bounded by the available memory of the partition phase and by the require-



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

Fig. 17. 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.)

ments of the storage manager. Experiences with the IBM DB2 have shown that storage managers can handle up to hundreds of active partitions per hash join [Lindsay 2002]. 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 17(a)-(c). 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 17(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 17(b) varies the number of matches per build tuple from 1 match to 4 matches for the 100-byte experiments in Figure 17(a). Figure 17(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 17(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.

8.3 User-Mode CPU Cache Performance on the Itanium 2 Machine

Choosing the Compiler and Optimization Level. In this subsection, we present our experimental results for hash join user-mode performance on the Itanium 2 machine. We first determine the compiler and optimization level for our experiments. Figure 18 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.[8]

[8]A group/software-pipelined prefetching bar shows the optimal result after tuning for the configuration.

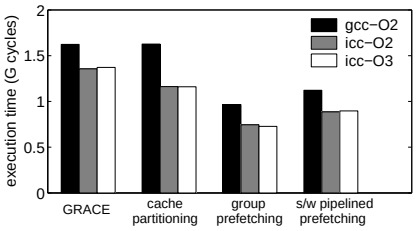
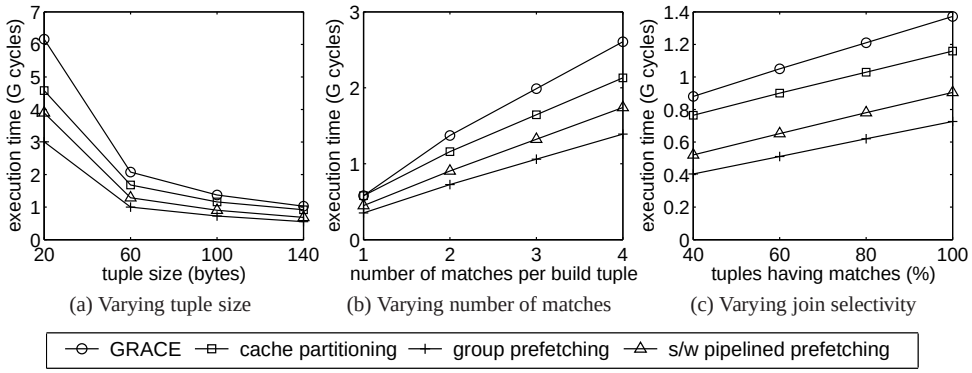


Fig. 18. Choosing the compiler and optimization level for hash join study on the Itanium 2 machine.



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

Fig. 19. Join phase user-mode performance on Itanium 2 (joining a pair of build and probe partitions).

The tuples are 100 bytes, and every build tuple matches two probe tuples. From Figure 18, 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 on the Itanium 2 machine.

Note that "icc -O3" automatically inserts software prefetch instructions into the generated executables for improving performance. Therefore, the compiler-enhanced GRACE join subsumes simple prefetching, and we do not report separate simple prefetching results on Itanium 2. Moreover, the cache partitioning scheme is also enhanced with compiler-inserted prefetches, which makes a stronger competitor to compare against.

Join Phase Performance. Figure 19 shows the join phase 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 the simulation study in Figure 11. For cache partitioning, we relax the limitation of 50 MB available memory, 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 Figure 19, 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.

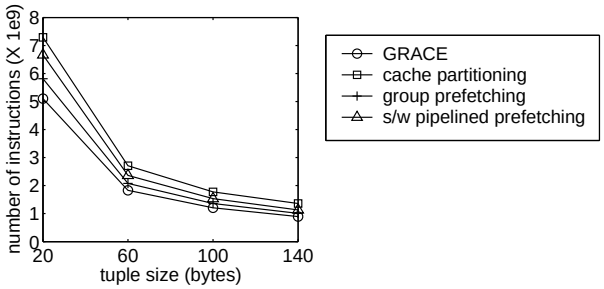
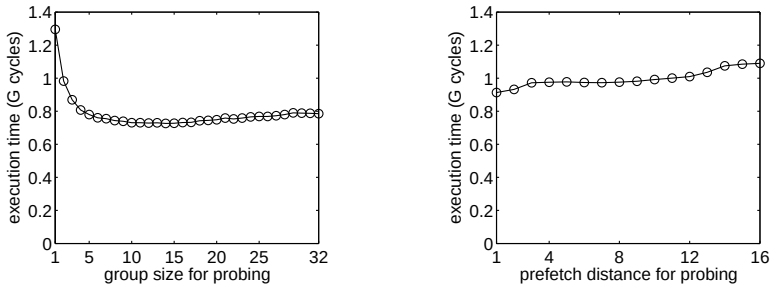


Fig. 20. Number of retired IA64 instructions for Figure 19(a).

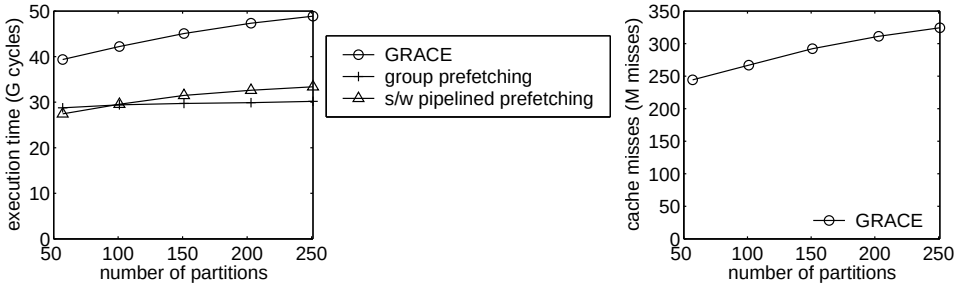


(a) Group prefetching (b) Software-pipelined prefetching

Fig. 21. Tuning parameters of group and software-pipelined prefetching for hash table probing in the join phase on the Itanium 2 machine.

Comparing Figure 19 and Figure 11 in the simulation study, we can see a major difference: Software-pipelined prefetching is significantly worse than group prefetching; group prefetching is 23-30% faster than software-pipelined prefetching in Figure 19. We examine the results by comparing the number of retired instructions of all the schemes in Figure 20. 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.

Algorithm Parameter Tuning. Figures 21(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 19(a) when tuples are 100 bytes. Here, we focus on the performance variance for only the hash table probing loop but the curves for the hash table building loop have similar shapes. The optimal values for probing are $G = 14$ and $D = 1$. These values are used in all the experiments shown in Figure 19. From the figure, 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 may be pre-set for a range of machine configurations.



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

Fig. 22. Partition phase user-mode performance on the Itanium 2 machine.

Partition Phase Performance. Figure 22(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.[9] The tuple size is 100 bytes. We see that the GRACE hash join degrades significantly as the number of partitions increases. While the number of memory references and instructions for processing a tuple does not change, Figure 22(b) shows that the number of L3 cache misses increases dramatically for GRACE join. This is because larger number of output buffers leads to larger chances for a memory reference to miss the CPU cache. While the GRACE join is enhanced by "icc -O3", automatically inserted prefetches 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.

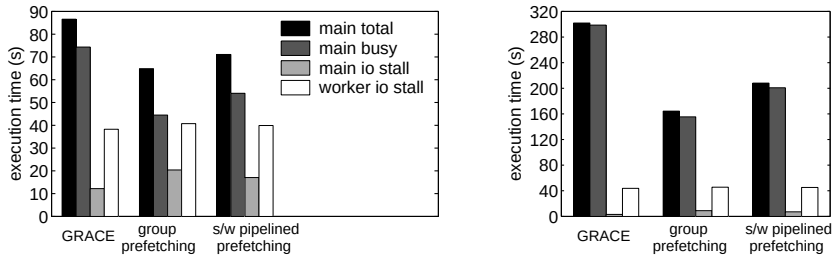
8.4 Execution Times on the Itanium 2 Machine with Disk I/Os

In this subsection, we study the impact of our cache prefetching techniques on the elapsed real times of hash join operations with disk I/Os. We perform the same set of experiments as in Section 8.1 (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.[10] Figures 23-25 compare our two cache prefetching schemes with the GRACE hash join. The experiments for the GRACE join with 100B tuples correspond to the seven disk points shown previously in Figure 10. Note that in Figure 25, the numbers of partitions, 57 and 113, are chosen automatically by the hash join algorithm 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 23-25, 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.

[9] 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.

[10] 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)

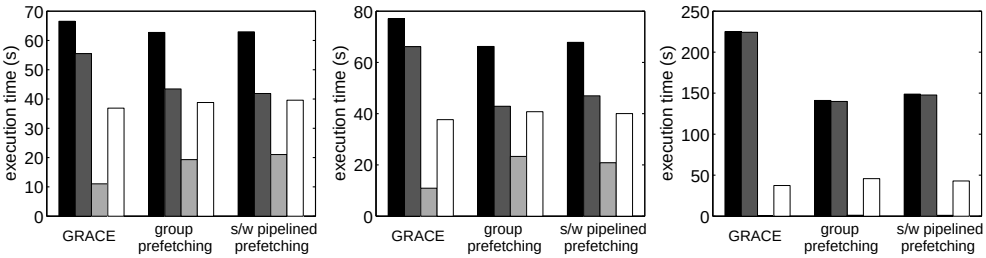
(b) 20B tuples (group prefetching: 1.84X speedup, s/w pipelined prefetching: 1.45X speedup)

Fig. 23. 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)

Fig. 24. Join phase performance with I/Os when output tuples are written to disk.



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

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

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

Fig. 25. Partition phase performance with I/Os.

The figures show a group of four bars for each experiment. These bars correspond to the four curves described previously in Section 8.1. 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 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 shown in Figures 23-25, the gap between the *main busy* time and the *worker io stall* time is larger, 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.

9. DISCUSSION

Several practical issues may arise when implementing our prefetching techniques in a production DBMS that targets multiple architectures and is distributed as binaries. First, the syntax of prefetch instructions is often different across architectures and compilers. We can define a set of macros for each pair of architecture and compiler combinations to hide the difference, then write code using the macros. This is the approach in our implementation.

Second, some architectures (e.g. existing x86 processors) do not support faulting prefetches that can succeed regardless of TLB misses. Two techniques can address this problem: (i) use (dummy) load instructions as faulting prefetches, which is correct because all the memory references that are prefetched will actually occur in our algorithms; (ii) use large virtual page sizes to reduce TLB misses, which is widely supported (e.g. on x86 [Intel Corporation 2006], SPARC [McDougall 2004], and POWER 5 [Hepkin 2006]). POWER 5 supports 16GB virtual page size, which can be sufficient for holding the in-memory data of most hash joins. Different x86 and SPARC processors support 2MB-256MB virtual page sizes. Given 4MB pages and a 32 entry TLB, if the hash table structure fits in 128MB, we can issue non-faulting prefetches for hash bucket headers and hash cell arrays, while using dummy loads for prefetching actual tuples.

Third, several architectures require software to explicitly manage the caches (e.g. the Cell Broadband Engine [Kahle et al. 2005] and network processors [Gold et al. 2005]). As shown in [Gold et al. 2005], our prefetching techniques can be well adapted for preloading data into the explicitly managed caches.

Fourth, pre-set parameters for the group size and the prefetch distance may be suboptimal on machines with very different configurations (e.g. memory speed). The solution is to perform a calibration test during DBMS installation to determine the best parameters.

Finally, as a last resort, if the above does not work, a DBMS can fall back to the original hash join algorithm either at compile time for specific architectures or at installation time based on the calibration results.

10. CONCLUSION

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 the techniques and successfully overcame the complexities involved with the hash join algorithm.

We performed detailed experimental studies through simulations and on an Itanium 2 machine focusing on both user-mode CPU cache performance and real elapsed times with disk I/Os. Our experimental results demonstrated that hash join performance can be improved dramatically by using our group prefetching and software-pipelined prefetching techniques. Moreover, the techniques will still be effective even when the speed gap between processors and memory increases significantly (e.g., by a factor of four). 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

This research is supported by a grant from the NSF. We wish to thank D. J. DeWitt for insightful comments. The third author thanks P. Bohannon, S. Ganguly, H. F. Korth, and P. P. S. Narayan for helpful discussions. We thank the anonymous reviewers for their comments and suggestions.

REFERENCES

BAER, J.-L. AND CHEN, T.-F. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91*. Albuquerque, NM, USA, 176–186.

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

CHEN, S. 2005. Redesigning Database Systems in Light of CPU Cache Prefetching. Ph.D. thesis, Carnegie Mellon University.

CHEN, S., AILAMAKI, A., GIBBONS, P. B., AND MOWRY, T. C. 2004. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering*. Boston, MA, USA, 116–127.

CHEN, S., AILAMAKI, A., GIBBONS, P. B., AND MOWRY, T. C. 2005. Inspector Joins. In *Proceedings of the 31st International Conference on Very Large Data Bases*. Trondheim, Norway, 817–828.

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

CHEN, S., GIBBONS, P. B., MOWRY, T. C., AND VALENTIN, G. 2002. Fractal Prefetching B^+-Trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. Madison, WI, USA, 157–168.

GOLD, B. T., AILAMAKI, A., HUSTON, L., AND FALSAFI, B. 2005. Accelerating Database Operations Using a Network Processor. In *Proceedings of the First International Workshop on Data Management on New Hardware (DaMoN 2005)*. Baltimore, MD, USA.

GRAEFE, G. 1993. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys 25*, 2, 73–170.

HEPKIN, D. 2006. Guide to Multiple Page Size Support on AIX 5L Version 5.3. http://www-03.ibm.com/servers/aix/whitepapers/multiple_page.pdf.

IBM CORPORATION. 2004. *IBM DB2 Universal Database Administration Guide Version 8.2*.

Intel Corporation 2004. *Intel Itanium 2 Processor Reference Manual For Software Development and Optimization*. Intel Corporation. Order Number: 251110-003.

Intel Corporation 2006. *IA-32 Intel Architecture Software Developer's Manual (Volumn 3A and 3B): System Programming Guide*. Intel Corporation.

KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JONHNS, C. R., MAEURER, T. R., AND SHIPPY, D. 2005. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development 49*, 4/5 (July/Sept.), 589–604.

KALLA, R. N., SINHAROY, B., AND TENDLER, J. M. 2004. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro 24*, 2 (Mar./Apr.), 40–47.

KENNEDY, K. AND MCKINLEY, K. S. 1990. Loop Distribution With Arbitrary Control Flow. In *Proceedings of Supercomputing'90*. New York, NY, USA, 407–416.

KITSUREGAWA, M., TANAKA, H., AND MOTO-OKA, T. 1983. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing 1*, 1, 63–74.

LAM, M. S. 1987. A Systolic Array Optimizing Compiler. Ph.D. thesis, Carnegie Mellon University.

LINDSAY, B. 2002. Hash Joins in DB2 UDB: the Inside Story. *Carnegie Mellon DB Seminar*.

LUK, C.-K. AND MOWRY, T. C. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA, USA, 222–233.

LUK, C.-K. AND MOWRY, T. C. 1999. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Transactions on Computers 48*, 2 (Feb.), 134–141.

MANEGOLD, S., BONCZ, P. A., AND KERSTEN, M. L. 2000. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Proceedings of the 26th International Conference on Very Large Data Bases*. Cairo, Egypt, 339–350.

MCDOUGALL, R. 2004. Supporting Multiple Page Sizes in the Solaris Operating System. http://www.sun.com/blueprints/0304/817-5917.pdf.

MCFARLING, S. 1993. Combining Branch Predictors. Tech. Rep. WRL Technical Note TN-36, Digital Equipment Corporation. June.

MOWRY, T. C. 1994. Tolerating Latency Through Software-Controlled Data Prefetching. Ph.D. thesis, Stanford University.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, MA, USA, 62–73.

NAKAYAMA, M., KITSUREGAWA, M., AND TAKAGI, M. 1988. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the 14th International Conference on Very Large Data Bases*. Los Angeles, CA, USA, 468–478.

PERFMON PROJECT. http://www.hpl.hp.com/research/linux/perfmon/index.php4.

SAULSBURY, A., DAHLGREN, F., AND STENSTRÖM, P. 2000. Recency-based TLB preloading. In *Proceedings of the 27th International Symposium on Computer Architecture*. Vancouver, BC, Canada, 117–127.

SHAPIRO, L. D. 1986. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems 11*, 3, 239–264.

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

Sun Microsystems. *UltraSPARC IV Processor Architecture Overview*. Sun Microsystems. Technical Whitepaper, Version 1.0, Feb. 2004.

TPC BENCHMARKS. http://www.tpc.org/.

ZELLER, H. AND GRAY, J. 1990. An Adaptive Hash Join Algorithm for Multiuser Environments. In *Proceedings of the 16th International Conference on Very Large Data Bases*. Brisbane, Queensland, Australia, 186–197.

ZHOU, J., CIESLEWICZ, J., ROSS, K. A., AND SHAH, M. 2005. Improving Database Performance on Simultaneous Multithreading Processors. In *Proceedings of the 31st International Conference on Very Large Data Bases*. Trondheim, Norway, 49–60.