Hash-Join Algorithms on Modern Multithreaded Computer Architectures

Philip Garcia Henry F. Korth

Department of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015 USA
{philipgar,hfk}@lehigh.edu

Abstract

As the performance gap between main memory and modern processors widens, database algorithms must be adapted to be "architecture-aware" for optimal performance. We address this issue using the computation of hash-join to study the impact of simultaneous multithreading (SMT) and main-memory latency (cache misses) on system performance.

Prior work [7] has studied cache misses on a simulation based on the Compaq ES40. Our results are obtained by measuring the performance of actual hardware (Intel Pentium and Xeon, and AMD Opteron) first for the single-threaded version of the hash-join algorithm used in the prior work and a new version designed for multiple threads.

We found that hardware prefetching for main-memory data into CPU cache as implemented in the actual architectures we tested reduces significantly the real-world benefit of software prefetching (contrary to prior work on simulated systems). We found that SMT achieved significant speedup for our thread-aware hash-join algorithm when compared with a single-threaded execution on the same single processor. Software prefetching proved beneficial in this environment.

1 Introduction

As microprocessors increase in speed and transistor densities increase, computer architects have devised new strategies to maximize the performance of these chips. These changes in design often require a reexamination of the strategies used by database-system designers in order to utilize the capabilities of the processor fully. Faster and larger main memories combined with access latencies that have actually increased when measured in processor cycles have fueled the need to rethink query-processing strategies[5, 3].

Significant progress has been made in optimizing database algorithms for the main-memory bottleneck[5, 21]. More recent research has aimed at using software prefetch instructions to speed up database operations[7, 9, 8].

Our work builds on the prefetching scheme used by Chen et al. in [7]. Their research has shown that significant gains can be had through the use of software prefetching in the hash-join algorithm. However, these experiments have been conducted primarily in software-simulated processors that do not fully take into account the actions taken by modern computer architectures when a prefetch instruction is issued.

We have built upon their research through the following:

- Running the experiments on real hardware.

Lehigh University Technical Report
LU-CSE-05-001

- Obtaining results that show that main-memory latency is not the only bottleneck in the system.

- Designing and testing a version of the algorithm that can take full advantage of modern multithreaded computer processors.

Our system is designed to run on currently available computer architectures including those that vary significantly from that used in [7]. Computer architects have been pushing the limits of what a single thread can run in a clock cycle. Therefore, they are using explicit forms of parallelism such as simultaneous multithreading (SMT) and single-chip multiprocessors [6]. We chose a multi-threaded design for those reasons.

Our study was conducted primarily on an Intel Pentium 4 Prescott machine based on the Intel NetBurst microarchitecture[14, 4]. Intel's Prescott microprocessor is an out-of-order, speculative, superscalar processor with deep pipelines. It is also the first general-purpose microprocessor to contain a form of SMT (referred to as Hyper-Threading in Intel documents)[14, 4, 18]. SMT is a technology that allows multiple threads to share the processor's execution units[18, 10, 22]. We measured the effectiveness of using software prefetching on this architecture and designed a version of the hash-join algorithm to take advantage of the SMT capabilities on the chip.

Our examination discovered the following about latent operations:

- Prior computer-simulation models have overestimated savings achieved through software prefetching.

- Previous research on using software prefetching has not accounted fully for the effects of built-in hardware prefetch units.

- Software prefetching is a useful tool when used under the proper circumstances.

- SMT can be used to speed up latent operations significantly.

We obtained the following results concerning the hash-join algorithm as described in [7]

- When combined, SMT and software prefetching can speedup hash-join operations dramatically.

- A simple division of tasks among threads can result in a large speedup of the hash-join algorithm.

- Copying memory can consume a large portion of the runtime when performing a hash join due to the limited instruction-level parallelism (ILP) caused by L1 cache latency.

2 Prior Work

In our research, we followed the example used by Chen et al.[7] and implemented a hash-join algorithm to test the ability of prefetching to hide latency on modern computer hardware. Their study presented many new ideas in how to best implement the hash-join operation.

Previous work has shown that the standard GRACE hash-join algorithm [16] is not an optimal solution for main-memory databases. One of the first attempts to create a more cache-aware hash join was done by Shatdal et al. in [21]. In [7] the authors followed Shatdal et al.'s methodology and created a direct-cache (DC) implementation of the hash join where each partition and hash table fits within the processor's L2 cache. However, [7] showed the direct-cache hash join to be infeasible in real systems (without the overhead of two-pass partitioning). They also showed that generating cache-sized relations requires extra overhead in the partitioning phase.

In order to speed up hash join, [7] used software prefetch instructions to hide the main-memory latency. They discussed two separate methods to implement prefetching: group-prefetching (GPF) as well as software-pipelining (SP).

Group prefetching is the simpler of the two algorithms. This method of prefetching first partitions the operation being performed into distinct stages that are separated based on memory accesses. Then the algorithm selects a group of tuples on which to operate. For each tuple in the group, the algorithm first does the computation required by the stage for that tuple and then issues a prefetch for the memory needed to process

the tuple in the next stage. Once the algorithm has done this for each tuple in the group, it proceeds to the next stage and repeats the process. After completing the final stage for the group, the algorithm selects another group of tuples and proceeds as before.

Software-pipelining is a slightly more complex form of prefetching, but potentially can hide more of the latency found in stages that do little computation. Software pipelining, like group prefetching, partitions the operation being performed into a pipeline of N distinct stages that are split based on memory accesses, so $O(t)$ becomes $O_1(O_2(\ldots(O_N(t))\ldots))$. We choose a prefetch distance D that defines how far in advance we prefetch a tuple's next memory access. Unlike group-prefetching, software-pipelining does not operate on groups of tuples at a time.

Each iteration proceeds in stages with each stage operating on a different tuple. Intuitively, after initial startup, the iteration operates as follows. In the first stage, it performs pipeline operation O_N on tuple x_i and subsequently prefetches the memory that tuple x_i will need when we run pipeline operation O_{N-1} on it. In the following stage, the algorithm performs operation O_{N-1} on tuple x_{i-D}. This continues until the last stage where the algorithm finishes operating on tuple $x_{i-D*(N-1)}$. At this point the algorithm proceeds to the next iteration, where it starts by processing tuple x_{i+D}. The algorithm follows this pattern until all of the tuples have been processed. Chen et al.[7] show that the optimal value of D is the smallest value that can hide all of the cache-miss latency.

Our system uses these same approaches to help hide main-memory's latency. However, our system differs from [7] in that it is based on a main-memory database model and does not take disk I/O into account. We chose this model because it has been shown that with a sufficient number of disk arms, disk I/O is not a bottleneck[7]. Our system also differs in that we designed a multithreaded algorithm to take advantage of the explicit parallelism found in SMT architectures, whereas prior work addressed unithread-execution cache optimization.

All of our tests were performed on a GNU/Linux platform and were compiled using gcc 3.3 with -O3, sse2 instructions enabled, and optimized for the specific machine (i.e. march=prescott or march=pentium4).

R	Build relation (400 MB)
r	Current tuple in R
R^i	Partition i of relation R
S	Probe relation (200 MB)
s	Current tuple in S
S^i	Partition i of relation S
T	Output relation
P_x	Pointer to tuple x
H_x	Hash value of tuple x (32 bits)
B_x	Hash bucket for hash value x in current hash table
b	Current entry in hash table bucket
b.P	Pointer contained in bucket entry b
b.H	Hash value contained in bucket entry b
HT_i	Hash table for i'th partition
2^α	Number of partitions
2^β	Number of hash buckets/partition
$\alpha + \beta \leq 32$	

Table 1: Variables used in algorithms

We ran our experiments on a variety of hardware platforms including an Intel Xeon 3.0 GHz (Northwood core), and an Intel Pentium 4 Prescott 2.8 GHz. We also ran the single-threaded algorithm on an AMD Opteron 1.6 GHz system for comparison. Results for this machine are available in Appendix C. The Northwood Xeon ignores a prefetch request if it results in a translation look-aside buffer (TLB) miss (this is likely to happen as there are 64 entries in the TLB, resulting in an effective window of $64 * 4$KB = 512KB of prefetchable memory locations)[13, 4]. We have therefore focused on the results obtained using the Prescott platform, as it supports the most interesting architectural features.

3 Partitioning

Our partitioning method is shown in Figure 1. We partitioned both a 400MB probe relation and a 200MB build relation. The relations were cre-

```
PARTITION(S)
1  for each s in S
2    do read tuple s from S
3       h ← bits 1—α of H_s
4       Append s,H_s to Relation S^h
```

Figure 1: Partitioning algorithm.

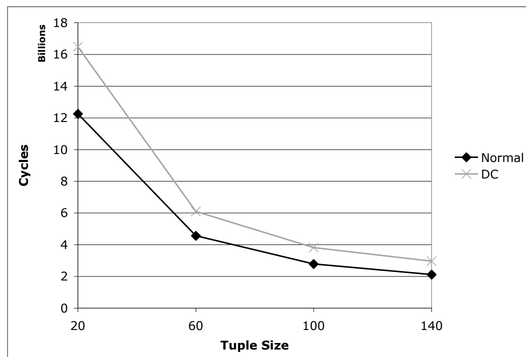


Figure 2: Partition cycles on P4 Prescott.

ated such that every tuple in the build relation matches exactly two tuples in the probe relation. Both relation schemas contained a variable-sized data field as well as a 10 byte key. The number of tuples in the relation varied with the tuple size in order to create a relation of the desired size. We chose these values in order to compare our results directly with those obtained by [7].

For the "normal" (without prefetching enabled) runs we split the relations into 8 partitions resulting in partition relations of approximately 50MB for the probe relation and 25MB for the build relation. For the direct-cache (DC) experiments we split the initial relations into 512 partitions (approximately 780KB and 390KB per partition). We chose these set sizes because when combined with the hash table, they are able to fit comfortably within the system's 1MB cache.

During the partitioning phase of the algorithm we followed [7] and implemented a simple software-prefetching scheme that prefetched the next tuple to be partitioned. However, we did not attempt to implement software-pipelining or group-prefetching as the simple prefetch scheme showed no change in overall performance. This results from the fact that the Prescott processor's hardware prefetch logic is capable of tracking multiple memory streams and can therefore prefetch the memory in advance without explicit prefetch instructions[4, 14].

Figure 2 shows the runtimes of the partition phase for both the normal runs and the direct-cache runs. It has been broken down to show the runtimes based on the tuple size. When using a tuple size of 100 bytes (we henceforth use this tuple size for comparisons unless we explicitly state otherwise) the Pentium 4 required 2.8 billion cycles to partition the relation using normal partition sizes and 3.8 billion cycles to generate cache-sized partitions.

Upon examination of these numbers we see that the processor does not manage to achieve its maximum theoretical main-memory bandwidth of 3.2GB/s. We determined that the partitioning phase of the algorithm is dominated by the time it takes to run *memcpy*. This is due to *memcpy*'s poor instruction-level parallelism, combined with L1 cache misses and L1 cache-access latency.

Assuming that the overhead to start and stop the partitioning phase is small in comparison to the total runtime, the processor spends approximately $\frac{2.8\ billion\ cycles}{6\ million\ tuples} = 470$ cycles processing each 100-byte tuple in the partitioning phase. The following equation shows the time spent in *memcpy* using the variables found in Table 2.

$$T_{copy}(N) = 2 * \left(T_{L2} * \left(1 + \frac{N}{L_1} \right) + T_{L1} * \frac{N}{4} \right) \quad (1)$$

Equation 1 shows the formula used to calculate $T_{copy}(N)$ or the time required to copy N bytes of data across L2 cache assuming that none of the data exists in the L1 cache, and that N is not very large (less than 1KB).[1] However, because the partitioning phase reads and writes the data in-order, the data loaded into L1 cache when processing tuple x_i will be used by tuple x_{i+1}[2]. Therefore this equation can be modified to the

[1] For large values of N, *memcpy* can run optimized routines that allow data to be accessed much faster than this equation shows.

[2] In the case of the output partitions, the data loaded when writing to partition s may not be used by tuple x_{i+1}, but when the number of partitions is small the data will likely still be in cache when it is requested by tuple x_{i+n}.

Variable	Description	P4	Opteron
Tm	Cycles to service a single cache miss (assumes a TLB miss)	250	150
Tn	Additional cycles to service a second (consecutive) cache miss	200	50
T_{L1}	Cycles required to retrieve 4 bytes from L1 cache	4	3
T_{L2}	Cycles required to move data from L2 cache into L1 cache	28	17
N	Number of bytes accessed	100	100
L_2	L2 cache line size (bytes)	128	64
L_1	L1 cache line size (bytes)	64	64
T_N	Time to read N bytes from main-memory into L2 cache.		
$T_{copy}(N)$	Time required to copy N bytes (assuming everything is in L2 cache)		

Table 2: Cache-related variables

following form which ignores the extra capacity cache miss that would occur when loading the first byte of the data[3].

$$T_{copy}(N) = 2 * \left(T_{L2} * \frac{N}{L_1} + T_{L1} * \frac{N}{4} \right) \quad (2)$$

Equation 2 shows that it takes approximately 285 cycles to copy a tuple. This works out to approximately 61% of the total cycles required to process a tuple in the partitioning stage. Upon further examination of the partitioning algorithm, we noticed that it takes approximately 125 additional cycles to generate the key's hash, perform bounds-checking, handle pointer arithmetic and account for the loop's overhead. When adding these to the cycles needed to copy the tuple we find that we have accounted for over 88% of the partition phase's runtime without taking into account main-memory latency.

This shows that the combination of *memcpy*'s poor ILP as well as L1 cache-access latency and L1 cache misses combine to create the primary bottleneck in the partitioning phase of the algorithm. This, combined with the processor's built-in hardware prefetch logic, explains why the addition of software-prefetch instructions had no impact on overall partition phase performance.

The runtimes that we obtained for the algorithm are similar to those generated in [7] when prefetching was enabled. Their experiments showed partitioning times of ∼1s for the prefetch-optimized runs. While their simulated processor varied significantly from ours in terms of speed, it is interesting to note that the memory bandwidth of the Pentium 4 is significantly less then that of the Compaq ES40 used in [7] (3.2GB/s versus 5.2GB/s). Examining these details it follows intuitively that our algorithm made greater use of the memory bandwidth available to us and was not limited by data cache misses, but rather limited by the speed of in-cache copy operations required to perform the partitioning.

[3] The first half of the expression is the cost of cache misses; the second is the cost of L1 cache access (in 4-byte units).

Build(S^i)

1 Create a new hash table B
2 **for each** s in S^i
3 **do** $h \leftarrow$ bits α—β of H_s
4 **insert** P_s,H_s into first free cell in bucket B_h
5 **return** B

Figure 3: Algorithm for constructing hash table.

4 Build Phase

The hash-table generation algorithm is shown in Figure 3. Build times for the partitioned relations are relatively small; taking only 10% of the total time it takes to both build and probe the relations (when prefetching is disabled).

Our build phase uses group prefetching similar to that used by [7], where it was shown to speed up the build portion of the hash join. While [7] discussed the necessity of preventing the rare condition of a read-write conflict when checking to add new values to the hash table in order to ensure correctness, we have found that this addition to the algorithm is unnecessary if we are willing to accept the miss cost when an entry is already

found in the hash table[4].

We found that the optimal runtime was obtained when there was less than 1 tuple per hash-table entry. We therefore chose to prefetch only the first entry in the hash table and accept the miss penalty if the bucket was already full (as this should be a relatively rare occurrence). This method allowed us to avoid making useless prefetches to null pointers, and ensured correctness as the writing of tuples to the table was handled serially. Had we not done this the check for read-write conflicts would require additional code that would potentially slow down the operation.

When we added the prefetch logic to the build algorithm we obtained a minimal improvement in the overall speed of the application. The speedup was negligible because the build phase uses a very small percentage of overall time, and because the time spent processing each tuple was minimal, resulting in little useful work to overlap with the prefetch attempt. Accesses to the partitioned relation were already prefetched by the hardware's prefetch logic so attempts to prefetch them would be worthless.

Because of this we chose to implement only the group prefetching version of the build algorithm and not software-pipelining because the added benefit of software-pipelining would likely be negligible.

5 Probe Phase

The algorithm for probing the hash tables is outlined in Figure 4. We chose to implement the normal probe algorithm, the direct-cache approach as well as group prefetching (GPF) and software pipelining (SP). We also implemented a combination of DC and SP (DC-SP).

Figure 5 details the total runtimes (in cycles) for the build and probe phases of the hash join. The probing process requires many random memory accesses resulting in many data-cache misses. The memory addresses of these misses can be precomputed in advance as [7] has shown.

[4] The read-write conflict exists because if two tuples x_i and x_{i+n} (where tuple x_{i+n} is in the same group as x_i) both hash to bucket B_h, then when the pipeline calculates and prefetches the final memory address for the tuples they would be the same.

PROBE(S^i, R^i, B)

1 **for each** r in R^i
2 **do for each** entry b in bucket B_h
3 **do if** $(b.H = r.H$
  $\& \ (*b.P).key = r.key)$
4 **then**
5 **append** r,$*(b.P)$ to T
6 **return** T

Figure 4: Algorithm for probing hash table.

This allows large speedups to be obtained by using prefetching's ability to hide main-memory latency. Our results showed a speedup of between 40% and 55% on the Pentium 4 Prescott when using software-prefetch instructions. The speedups varied with the tuple size yielding larger speedups for smaller tuple sizes.

5.1 Analysis of a Probe

It is interesting to note that software pipelining obtained slightly better results than group prefetching[5]. This is in contrast to the best results obtained in [7]. We attribute this to two reasons. The first is because the cache-miss latency on the Prescott processors is large (∼250 cycles). Chen et al.[7] also observed that software pipelining outperforms group prefetching when the memory latency is very large (1000 cycles). Our results can also be attributed to the fact that some stages in the group-prefetching algorithm do very little "useful" or non-latent work and are incapable of overlapping all of the main-memory latency with useful instructions.

Each probe required approximately 1950 cycles to execute when no prefetching was enabled (using standard partition sizes). When we enabled software pipelining, each probe required only 1200 cycles; yielding a savings of about 750 cycles/probe.

By examining the probe algorithm detailed in Figure 4, we see that there are two distinct points where we must access a "random" memory location that is likely to result in a cache miss. The

[5] For the software-pipelining algorithm we used a D value of 1 that we obtained experimentally and mathematically to be the best possible D value under the test conditions.

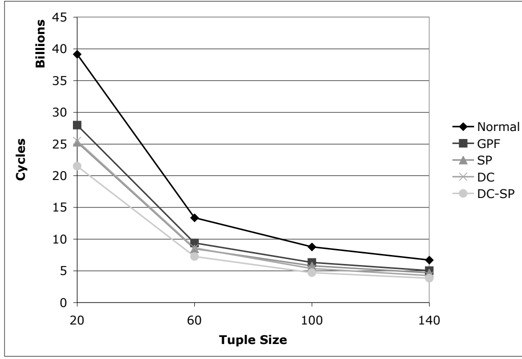


Figure 5: Total build and probe cycles on a P4

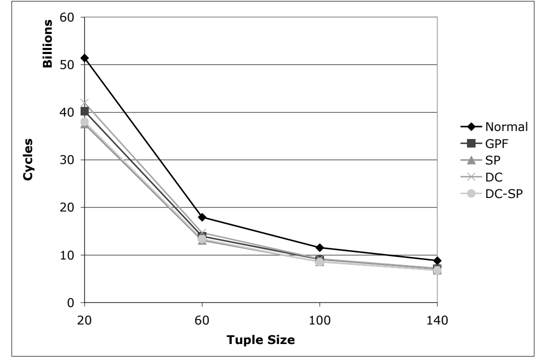


Figure 6: Cycles for entire hash join on a P4.

first cache miss occurs on line 2 of the probe algorithm. This miss occurs when we look up the current entry in the hash table and subsequently read an 8-byte bucket header that consists of a 4-byte tuple pointer and another 4-byte overflow pointer. We are assuming that these 8 bytes do not cross any cache-line boundary so the cost of this memory access is simply Tm (see Table 2).

The second cache miss is significantly more expensive then the first one, because it can span multiple cache lines. This miss occurs on line 3 of the probe algorithm with subsequent misses possibly occurring on line 5. These misses occur when we read in the N-byte tuple from main memory. This tuple will often cross cache-line boundaries and therefore takes T_N cycles:[6]

$$T_N = Tm + Tn * \left(\left\lfloor \frac{|N - L_2|}{L_2} \right\rfloor + \frac{N \bmod L_2}{L_2} \right) \quad (3)$$

Equation 3 takes into account the first cache miss (which will also likely cause a TLB miss) as well as subsequent misses when reading in the next line (note: subsequent misses can also be much faster than the first miss due to spatial locality within main memory as well as the hardware's data-prefetch buffer[4]). On our test platform these equations result in ~660 cycles of latency for prefetching to hide. However our results have shown prefetching on the Pentium 4 to save us ~750 cycles per iteration.

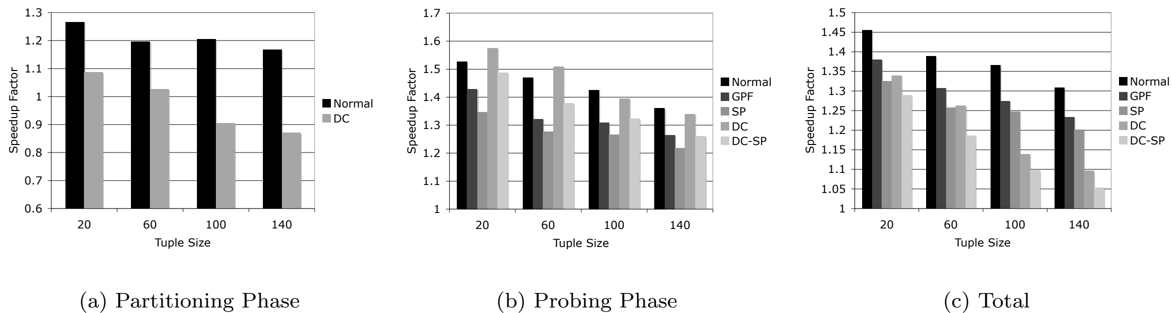
[6]It is safe to assume that the entry in the build table is not found in the processor's cache because the build partition is of size 25MB while our L2 cache is only 1MB in size.

The remaining savings can be attributed to the prefetch instructions forcing the cache to evict the output tuples to main memory at an earlier time as well as the time required to move the entire cacheline into L2 cache (based on the bus bandwidth). While each cache miss requires Tm or Tn cycles to start moving data into L2 cache, it requires even more time for the rest of the data to finish streaming into cache (i.e. $Tm + N/BusBandwidth$).

The probing algorithm spends much of the time on actions other than waiting for main memory. Equation 1 calculates the time it takes to copy data from the input tuples to the output tuples across the L2 caches (assuming L1 cache misses). This time is rather significant taking ~ 600 cycles. Much of the rest of the time spent can be characterized as waiting to read the probe tuple from memory (a bandwidth limitation problem) as well as the time unnecessarily spent reading the output tuple from memory and subsequently evicting it from the cache.

5.2 Comparison with Prior Work

The hardware simulated in [7] differs from the hardware that currently is available on the market. Their processor model assumes when there are multiple outstanding cache misses the first miss requires Tm cycles while subsequent misses require the inverse of the memory bandwidth (i.e. T_{next}). The fact that their performance did not degrade significantly when Tm was increased suggests that their value for T_{next} does not increase with Tm.



(a) Partitioning Phase (b) Probing Phase (c) Total

Figure 7: Speedups for the SMT version of the algorithm

This assumption requires that memory accesses are fully pipelined operations. This assumption is only partially valid (the time that is spent while the requests are waiting on the memory controller allows partial overlap, and multiple outstanding requests), however modern computer systems are still limited by the memory bus and cannot allow very many concurrent memory accesses (if the data is located in the same physical DRAM then simultaneous access becomes impossible with current technologies). These details, though perhaps too complex for a tractable simulation, are readily apparent in our experiments using actual hardware.

Our observation of current hardware suggests that we can overlap memory requests with useful work. However the gains from prefetching cannot increase beyond a threshold as each cycle that is being saved from a memory operation must have a matching cycle of useful work in order to hide the latency. Assuming that T_{next} is roughly the same as Tm, we are not able to obtain the speedups of over 2x that [7] observed on their simulated processor.

6 Multithreaded Algorithm

One of the goals in designing our hash-join algorithm was to allow multiple threads to execute concurrently. This parallelism can be exploited by current and future multithreaded and/or multicore architectures such as those by Sun (Niagara architecture), AMD, Intel and IBM[2, 1, 11, 12]. We chose to limit our experiments to only two threads because Intel's NetBurst architecture currently supports a form of SMT called *Hyper-Threading* that allows at most two threads to execute at the same time[14, 18, 4].

SMT is a technology that allows multiple threads to execute inside the same processor core at the same time (sharing functional units, caches, etc. between the threads). SMT has been shown to be highly effective when dealing with high-latency applications[13, 17, 20, 19]. One way SMT can achieve this is by allowing one thread to run at nearly full speed while another thread is stalled, waiting for data (whether from main memory, L2, or L1 cache). It also enables the two threads to share the processor's function subunits when running calculations. However, because the hash-join algorithm is not a computation-intensive process, most of the benefit obtained from it is through the overlap of memory instructions[13, 10, 23].

The hash-join algorithm does not fully lend itself to parallelization within the partitioning phase. Without more explicit synchronization constructs that could potentially impose cache-misses of their own, we cannot start building the hash table or probing the build relations until both the build and probe relations are finished being partitioned. In our algorithm, one thread must sit idle for a significant portion of the partitioning time (in our tests the build relation was exactly half the size of the probe relation and thus took approximately half as long to partition). We designed the build/probe phase of our algorithm such that each thread obtains the next available partition and proceed to build a hash table and subsequently probe it. This resulted in a simple but effective separation of tasks that required a minimum of synchronization.

6.1 Multithreaded Speedup

Figure 7 shows the speedup factors obtained when we enabled SMT. These results show that most of the speedup obtained is found in the probe phase of the algorithm. This is not surprising due to the limited parallelism within the partitioning phase. Figure 7(a) shows that the partition phase yields speedups between 17% and 27% for "normal-sized" partitions (The anomalies found in the DC result are covered in detail in Section 6.2).

Our results show that the speedups are greater for smaller tuple sizes. This makes sense because, for smaller tuples, we must calculate more information per byte moved (e.g. calculating hash values for every 20 bytes rather then 140 bytes) generating more cycles with which we can overlap integer and load/store operations. These numbers are on par with the speedups obtained in [20] when simultaneously running a memory-based microkernel and an integer microkernel on a Pentium 4 with Hyper-Threading.

The probe phase offers significantly more room for SMT-enabled processors to improve performance because the probe process is more naturally split up among threads and requires much movement of data, creating latencies that SMT can mitigate. During this portion of the algorithm, we obtain speedups between 36% and 53% for the normal algorithm and speedups between 22% and 35% for the software-pipelined algorithm. The normal algorithm obtained a larger speedup because cache misses are more common in the non-prefetch-enabled version of the hash join, and thus there is greater opportunity for SMT to overlap a cache miss on one thread with useful work in the other.

For the combined execution, Figure 7(c) shows that SMT is particularly useful in helping to speed up non-prefetch-optimized hash-join algorithms. Figure 8 shows the execution times for both single and multi-threaded execution with and without prefetching. Enabling both SP and SMT results in a speedup of between 56% and 82% depending on the tuple size.

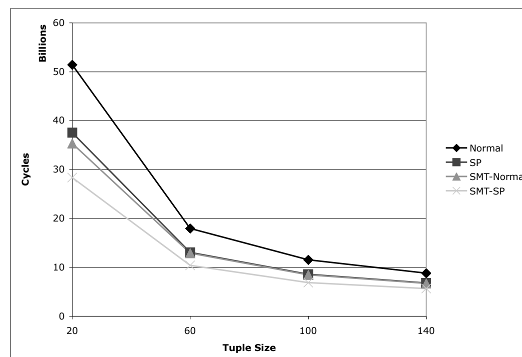


Figure 8: Total Runtimes

6.2 Direct-Cache Results

The direct-cache approach results in suboptimal performance and, as Figure 7(a) shows, enabling SMT actually made the partition phase slower for the direct-cache measurements (speedup factor $<$ 1). The reason for this is that each thread can use only half of the cache, so each partition in DC must be only half as big. This doubled the number of partitions.

While the direct-cache approach obtained significant speedups during the probe phase, we found that the partition phase became prohibitively more expensive when we increased the number of partition to 2^{10} in order to maintain a direct-cache algorithm. Because of the extra overhead in the partitioning phase, our results showed that running two threads using SP (without cache-sized partitions) is the fastest method to perform the hash-join algorithm. Even when we enabled both software pipelining and cache-sized partitions, we were not able to obtain runtimes as fast as those obtained when we used the SP algorithm on standard partition sizes due to the excessive time spent during the partitioning phase[7].

The DC approach results in suboptimal performance. As [7] observed, it is unlikely that a conventional database system could effectively handle the number of partitions generated by the DC partitioning algorithm. We found that software prefetching is more effective than using the conventional cache-sized partition approach to speed

[7]As figure 6 shows, for the single threaded case DC-SP ran at almost the exact same speed as the base SP algorithm.

up the hash-join algorithm. Our results for multithreading agree with the those obtained by [7] for the unithreaded case.

7 Future Work

While our work has shown that the benefits of using software prefetch instructions have often been overestimated in software simulations, it has shown that software prefetch instructions are highly useful and help to eliminate the main-memory bottleneck that exists in many database operations such as the hash join.

However, our work has also shown that eliminating the stalls due to main-memory latency can expose new bottlenecks that can hinder a program's performance. One of the most important bottlenecks uncovered is the time required to copy data across a processor's cache. This speed is partially limited by current implementations of the *memcpy* function. *memcpy* can be speeded up through the use of multiword transfer instructions (as found in SSE2 instructions and 64-bit integer operations), producing a *memcpy* that optimally copies small blocks of memory (<1KB). However, taking advantage of these optimizations would require either aligning the data on 16- (for SSE2 instructions) or 8- (for 64-bit **int** instructions) byte boundaries, or creating special cases to handle the smaller unaligned data transfers required.

While we attempted to use special purpose *memcpy* routines in our code none of them resulted in an appreciable speedup for the small data transfers we used in the hash join. Modifying the data structures used in the hash join to help ease the use of these techniques would also result in a significant reduction in portability of the system to other platforms. We have therefore left restructuring of data-access code to future work.

Another potential optimization that could be made within the partitioning and probing phases is to write to memory using non-temporal stores. Non-temporal stores allow the processor to write data to main memory without first bringing the current value of the memory to be overwritten into the processor's cache. This helps to conserve bus bandwidth and allows for more efficient writes. While the Pentium 4 has support for non-temporal stores[15], it is designed for writing entire cache-lines at once. This means that for tuples significantly smaller than a cache line, a buffering system would need to be implemented. It is questionable how much performance this would gain us (if any) due to the overhead of buffer management, so we have also left this particular optimization for future work.

Another promising direction for future work is a pipelined version of the algorithm, particularly for future architectures that require extensive thread level parallelism for optimal execution (such as Sun's upcoming Niagara and Rock architectures)[2]. Hash join may benefit also from SIMD approaches as proposed by [24].

8 Conclusion

In conclusion, our work has shown that the hash-join algorithm can be optimized heavily for usage with modern processors. Our research has shown that while software prefetching is not as useful as previous computer simulations have shown[7], the use of software-prefetching algorithms can greatly speed up the execution of the hash join.

Our work has also shown that a multithreaded implementation of hash join running on an SMT processor offers dramatic performance improvements. Our discussion of these results indicate that one should expect similar impact on workloads that have an abundance of memory access instructions—not only those that result in cache misses, but also when main-memory latency has been hidden through other means such as software prefetching.

References

[1] AMD to release first x86 multi-core processors mid-2005. `www.amd.com/us-en/-0,,3715_11787,00.html`.

[2] Throughput computing. `/www.sun.com/-processors/throughput/faqs.html`.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proc. of the 25th International Conference on Very Large Data Bases*, 1999.

[4] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, (Q1):4–15, 2002.

[5] P. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proc. of the 25th International Conference on Very Large Data Bases*, 1999.

[6] D. Burger and J. R. Goodman. Billion-transistor architectures: There and back again. *IEEE Computer*, 37:22–28, Mar. 2004.

[7] S. Chen, A. Ailamki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *IEEE International Conference on Data Engineering*, 2004.

[8] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *ACM SIGMOD International Conference on the Management of Data*, May 2001.

[9] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefeching B^+-trees: Optimizing both cache and disk performance. In *ACM SIGMOD International Conference on the Management of Data*, June 2002.

[10] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[11] I. D. Forum. Intel outlines platform innovations for more manageable, balanced and secure enterprise computing. `www.intel.com/pressroom/archive/-releases/20040218corp.htm`.

[12] M. Funk. Simultaneous multi-threading (SMT) on eServer iSeries POWER5 processors. `www-1.ibm.com/servers/eserver/-iseries/perfmgmt/pdf/SMT.pdf`.

[13] P. Garcia and H. F. Korth. Sort algorithms that exploit multithreaded architectures. 2005.

[14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1), 2001.

[15] Intel. *Intel Pentium 4 Processor Optimization*, 2001.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. In *New Generation Computing*, volume 1, pages 63–74, 1983.

[17] J. J. Lo, L. A. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. Technical report, Compaq, July 1998.

[18] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, (Q1):4–15, 2002.

[19] L. K. McDowell, S. J. Eggers, and S. D. Gribble. Improving server software support for simultaneous multithreaded processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[20] V. K. Reddy, A. M. Sule, and A. V. Anantaraman. Hyper-threading on the Pentium 4, December 2002.

[21] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proc. of the 20th International Conference on Very Large Data Bases*, pages 510–521. Morgan Kaufmann Publishers Inc., 1994.

[22] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22nd Annual International Symposium on Computer Architecture*, June 1995.

[23] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 191–202, 1996.

[24] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proc. ACM SIGMOD International Conference on the Management of Data*, June 2002.

A Explanation of Equations

A.1 Equation 3

Equation 3 shows the number of cycles required for the processor to read N bytes from memory (assuming all N bytes are not in any levels of cache, and the address of N is not found in the translation look aside buffer). The equation is simplistic in its form and just shows that the cost to read N into memory is simply Tm (the cost of the first cache miss including address translation) plus Tn (the time required to service a second miss) multiplied by the number of misses that will occur (subtracting out the miss from the first one). $\left\lfloor \frac{N-L_2}{L_2} \right\rfloor$ denotes the number of extra required misses (regardless of where in the cache line the value is and $\frac{N \bmod L_2}{L_2}$ denotes the probability that the last portion will require an extra cache miss because it straddles a cacheline. This equation assumes that the tuples are randomly scattered in memory and not even multiples of the cache line size. This is a valid assumption as most datasets are not of even cache line multiples.

A.2 Equations 1 and 2

Equation 1 details how long it takes to copy N bytes on a processor assuming that both the source and destination memory are already loaded into the processor's L2 cache. It also assumes that reading is equally as expensive as writing to the cache and that the operations cannot overlap. The first half of the equation $T_{L2} * \left(1 + \frac{N}{L_1}\right)$ calculates how much time must be spent servicing L1 cache misses. The number of L1 cache misses that occur is $1 + \left\lfloor \frac{N}{L_1} \right\rfloor + \frac{N \bmod L_1}{L_1}$ where $1 + \left\lfloor \frac{N}{L_1} \right\rfloor$ represents the number of required cache misses and $\frac{N \bmod L_1}{L_1}$ represents the probability that the data will cross a cacheline boundary. This can then be simplified into the expression above.

However this equation must be modified if the data is being copied sequentially (i.e. read 100 bytes from location memory location M_1, write it to memory location M_2, subsequently read 100 bytes from $M_1 + 100$ and write to memory location $M_2 + 100$. Under this scenario it should take the same length of time as simply copying 200 bytes (as the L1 cache is shared between the two transfers). Therefore the equation can be modified such that the number of L1 cache misses is only $\frac{N}{L_1}$ due to the sharing of L1 cache between memory accesses.

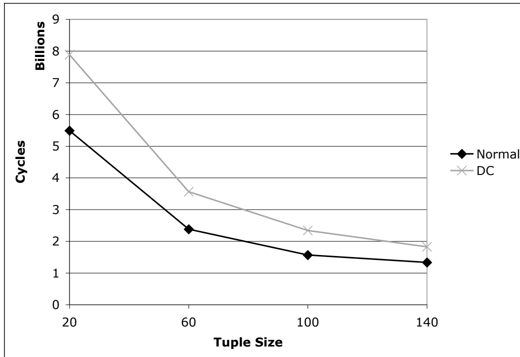
The second portion of the equation is simply the time required to read or write the data from or to L1 cache, assuming that data is read/written in 4 byte quantities. In our calculations we assumed that we had to move 190 bytes (90 bytes of data for both the build and probe tuple as well as the 10 byte join key).

B Choice of D in the software-pipelining algorithm

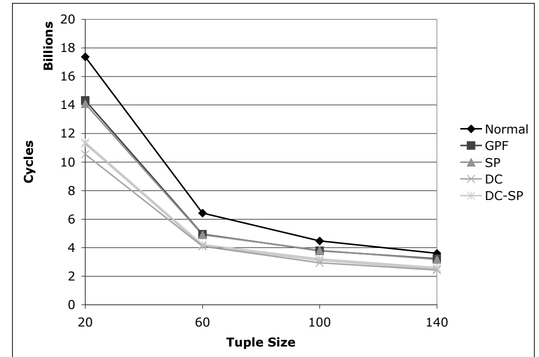
In choosing an optimal value of D for software pipelining we need the smallest value that is large enough to hide all of the latency associated with the memory operation. In order to obtain this value Chen et. al proposed the following equation:

$$D \cdot (max\{C_0 + C_k, T_{next}\} + \sum_{l=1}^{k-1} max\{C_l, T_{next}\}) \geq T \quad (4)$$

Where C_i represents the time required to process step i, and T_{next} represents the time required to prefetch an additional cache-line. However in our experiments we found the value of T_{next} to be roughly the same as T. By applying this equation



(a) Partition phase



(b) Build/Probe phase

Figure 9: Cycles required for the hash join on the AMD Opteron platform

we get:

$$D \cdot (max\{C_0 + C_k, T\} + \sum_{l=1}^{k-1} max\{C_l, T\}) \geq T \quad (5)$$

This statement effectively says that in order to fully hide all the latency $D \cdot \sum_{l=0}^{k} \leq D \cdot \sum_{l=0}^{k} T$. Because of this the value of D can be factored out which means that the smallest value of D is trivally 1.

C AMD Opteron Results

In examining the runtimes on the Opteron (Figures 9(a) and 9(b)), it is obvious that the hash join is performed in fewer cycles than on the NetBurst architecture. This is due to the Opteron's much lower clock speed (1.6GHz)[8]. The Opteron's results show that this processor is able to extract much greater instruction-level parallelism from the code. However it is interesting to note that the Opteron's runtimes do not vary greatly in terms of seconds(for the unoptimized runs) from those of the Pentium 4.

The probe phase's performance on the Opteron (Figure 9(b)) showed much smaller speedups than those obtained on the Pentium 4 when we enabled prefetching (we saw speedups of between only 13% and 21% for software prefetching on the Opteron). They also show software prefetching's inability to hide main-memory latency fully when compared to the direct-cache approach.

It is interesting to note that while the Opteron can outperform the Pentium 4 when no prefetching or multithreading optimizations are made (despite the Opteron's significantly lower clock speed), once these optimizations are enabled, the Pentium 4 outperforms the Opteron significantly.

Because the Opteron did not allow us to obtain as interesting results through the use of software prefetching, and because the single processor version of the Opteron offers no chance to exploit thread-level parallelism, we chose to focus our results on the Pentium 4 and presented those numbers throughout the paper.

[8]note: the Opteron was run in 32 bit mode so 64 bit extensions have not been enabled