

Exploiting Multithreaded Architectures to Improve the Hash Join Operation

Layali Rashid, Wessam M. Hassanein and Moustafa A. Hammad*

Department of Electrical and Computer Engineering,*Department of Computer Science
University of Calgary

{lrashid@ucalgary.ca, hassanein@enel.ucalgary.ca, hammad@cpsc.ucalgary.ca}

ABSTRACT

As database management systems gain importance in our everyday life, it is essential to have efficient implementations of important database operations such as the hash join. Improvements in processor architectures including simultaneous multithreaded architectures and Chip Multiprocessors have opened opportunities for taking advantage of the new multithreaded hardware. Recently, several efforts have been done to enhance database performance through architecture-aware data management. In this paper, we present a new architecture-aware hash join (AA_HJ) algorithm for main memory database systems, where all the data resides in memory. AA_HJ relies on sharing critical structures at the cache level, and distributing the load evenly between threads. Our timing results show a performance improvement up to 2.9x for the Intel® Pentium® 4 HT and up to 4.6x on the Intel® Quad Xeon® Dual-Core machine, compared to single-threaded hash join. The L2 load miss rate is reduced by up 82%.

1. INTRODUCTION

As information management becomes an integral part of our everyday life, database management systems (DBMSs) gain further importance as a critical commercial application. The performance of DBMSs has been less than optimal due to their poor memory performance [3]. Main memory database systems (MMDB) [26] suffer from large cache misses and low CPU utilization [1]. At the hardware level, multithreaded architectures are considered among the significant advances in processor architectures. In Simultaneous Multithreaded architectures (SMT) [23] multiple threads execute concurrently sharing the same hardware. Whereas in Chip Multiprocessors (CMP) [27]; one chip contains multiple processor cores usually sharing the second level cache and the bus. Such sharing can result in either performance improvements (e.g., one thread prefetching data for another) or performance degradation (e.g., two threads conflict in the shared caches). Exploiting multithreaded architectures create new opportunities for improving essential DBMSs operations. Hash join (an optimized join operation that uses hash table data structures) is one of the most important operations commonly used in current commercial DBMSs [22]. Therefore, revisiting the join implementation to take advantage of state-of-the-art hardware improvements is an important direction to boost the performance of DBMSs. Significant work has been done on the improvement of hash join operations ([4], [6], [9], [16], [20], [21], [24], [25]). Parallel hash Join has been extensively examined by Shatdal [20] for SMP architectures. Shatdal [20] also presented a hybrid between hash join algorithm designed for shared-nothing multiprocessors and SMP systems. Database operations have been investigated on SMT architectures in many papers ([9], [15], [18],

[24]), including hash join operations ([9], [24]). In [9] Garcia and Korth conclude that prefetching techniques in [4] are useful for the probing phase only in the hash join on real SMT hardware. J. Zhou et al in [24] use a helper-thread approach to exploit the two threads available in an SMT architecture. Work on database operations and CMP architectures include [28] in which researchers evaluate the On-Line Transaction Processing (OLTP) benchmark TPC-C and the decision-support database benchmark TPC-H on a CMP simulator. They find that most stalls are due to data misses mainly in the Level 2 cache. In [29] Colohan et al. use speculative threads to parallelize database queries for a CMP 4-processors simulator, and achieve speedups ranging from 36% up to 74% for some TPC-C transactions. Other work on tuning software on CMP environments include [3], which presents a general theoretical justification of upper and lower bounds on cache misses for a system consisting of p processors with shared memory hierarchy. However, previous work does not take advantage of the sharing in the underlying hardware structures, has less than optimal work division among threads, and does not provide insights into the memory hierarchy performance. In this paper we present the following main contributions: (1) we analyze and study the different phases of traditional hash join algorithms using one of the most popular join algorithms (the Grace Algorithm [13]) and highlight existing problems. (2) We apply improvements to the different hash join phases to enhance their single thread performance. (3) We propose a multithreaded hash join algorithm that takes advantage of the underlying multithreaded architecture by *sharing* data between threads in the same processor. Thus, reducing cache conflicts and using one thread to prefetch data for the other. We refer to our algorithm as an Architecture-Aware Hash Join (AA_HJ). (4) We show that our proposed algorithm can be easily integrated with the recent (yet orthogonal) improvements to the single threaded hash join operation to achieve high performance. In particular, we take advantage of the software group prefetching technique proposed by [4]. To the best of our knowledge, no other work has proposed a multithreaded hash join algorithm that takes advantage of the underlying SMT and CMP hardware. In this paper we conduct our experiments on the Intel® Pentium® 4 HT (SMT, dual-threaded) processor and the Intel® Quad Xeon® Dual Core server (combination of SMT, CMP and Symmetric processors (SMP), up to 16 threads). On the first machine we achieve speedups ranging from 2.1 to 2.9 times compared to the Grace hash join. While on the second machine our speedups range from 2 to 4.6 times depending on the tuple size. The rest of this paper is organized as follows: Section 2 describes the concepts of databases and hash join. In Section 3 we present the details of our proposed dual-threaded and multi-threaded AA_HJ. Section 4 describes the experimental methodology. In Section 5 we present the timing and memory-

characterizing results on the Intel® Pentium® 4 HT processor for the dual-threaded AA_HJ. In Section 6 we show the results on the Intel® Quad Xeon® Dual Core server for more than two threads and characterize the hardware performance for AA_HJ and analyze its memory behavior using the Intel® VTune Performance Analyzer. Finally, conclusions are presented in Section 7.

2. BACKGROUND

This section introduces database management systems (DBMSs) and hash join operations [2]. The relational database management system (RDBMS) model is the traditional DBMS originally presented by Edgar F. Codd [6]. RDBMS is a tabular representation of a database, where records (tuples) represent the rows and attributes represent the columns. As an example Figure 1 has three relational tables.

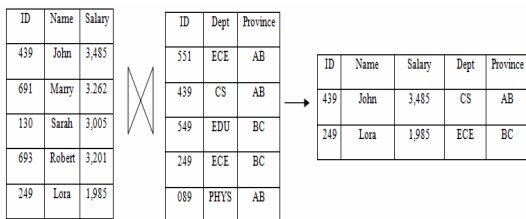


Figure 1: Database Join

Queries initiated to the RDBMS include retrieving tuples that satisfy some conditions, updating, and deleting tuples. Some queries request data that exists in two relations (tables), Figure 1 shows an example of joining two tables. The datasets are organized such that some employees have their names and salaries stored in one table, while the departments and provinces are stored in another table. To retrieve all the data for any employee whose ID is in both tables, we perform a natural-join. Natural join is one variation of a join in which we ask to retrieve all tuples from both relations whose join-key (ID in Figure 1) matches. This is one of the most popular types of joins. In its simplest form, joining two relations can be processed by two nested loops, where the outer loop reads a tuple from the large relation, and the inner loop scans the smaller relation looking for tuples with keys equal to that for the outer tuple. A more efficient (and the most popular) implementation for the join query is the hash join which is shown in Figure 2. In a hash join, a hash table is constructed from the smaller relation (usually called R or *build relation*).

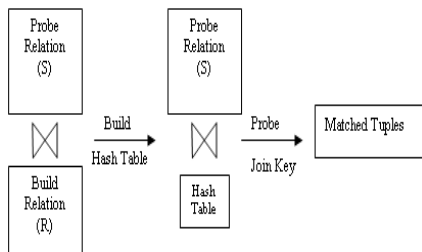


Figure 2: Hash Natural-join Process

Next, tuples are probed from the larger relation (usually called S or *probe relation*) one by one using the hash table. A hash table structure is shown in Figure 3. It is an array of buckets, where each bucket has a pointer to a linked list of cells. Each cell has a pointer to a tuple in the build relation, and a hash value generated from the

joining key of this tuple. After building the hash table, the probe relations' tuples are read one by one. For each S tuple read, the joining key hash value is computed, and then the bucket number is calculated from the hash value. The proper bucket (cells array) is accessed, and each cell's hash value is compared against the S tuple's hash value for a match. If a match occurs, the pointer in that cell is dereferenced so as to load the build relation R tuple, whose key will be compared against the probe S tuple's key for a match. If we have a match then both the build and probe tuples are projected into the output buffer. A hash join requires random accesses to the hash table during the probing phase, and random accesses to the R-relation to retrieve the matched tuples. To reduce the memory access latency resulting from these random accesses, previous efforts have concentrated on storing the data tables as close to the CPU as possible. For disk-resident databases (DRDBs) ([2], [8]) both the R and S-relations are partitioned into clusters (partitions) that fit in the main memory.

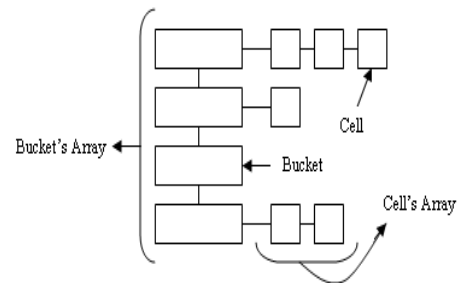


Figure 3: Hash Table Structure

This algorithm is widely known as the "Grace Hash Join". While for MMDB, a similar partition-based approach called "cache partitioning" (a.k.a. Direct Cache, DC) is used. In DC partitioning ([4], [9], [14], [17], [21]) the R and S-relations are partitioned into clusters such that each R cluster and its corresponding hash table fit in the highest level cache (largest cache) in the machine. This is done prior to any hash join processing. The partition-based hash join algorithm is shown in Figure 4.

```

partition R into R0, R1, ..., Rn-1
partition S into S0, S1, ..., Sn-1
for i = 0 until i = n-1
    use Ri to build hash-tablei
for i = 0 until i = n-1
    probe Si using hash-tablei

```

Figure 4: Hash Join Base Algorithm

In the parallel hash join [16] both relations are partitioned among the available processors p in a multi-processor system, for example. This is done by dividing the S and R-relation into p clusters (blocks), such that each cluster has approximately the same number of tuples. Then each processor uses its R-relation-cluster to build one global hash table. Multiple writes to the same memory location are synchronized by latches. In the final step of the parallel hash join, each processor probes its cluster using the global hash table.

3. ARCHITECTURE AWARE HASH JOIN

In this section we propose a dual-threaded architecture-aware hash join (AA_HJ) database operation, then we extend it to use more than two threads. Our algorithm takes advantage of the following two main features in SMT architectures: (1) two threads are

available to run simultaneously, (2) the full memory hierarchy is shared between these two threads (i.e. the cache sharing feature of SMT architectures). MMDB systems suffer from high L2 cache miss rates and therefore, reducing/hiding the memory access latency is an important performance factor for hash join operations.

The Build Index Partition Phase: We use the OpenMP library ([7], [19]) to initiate two threads, where each thread is assigned a unique ID. To minimize thread creation and killing overhead, we initiate the two threads only once when the hash join begins, and kill the threads only when the join is completed. Our algorithm starts by creating structures to hold the R-relation index clusters (partitions) for each thread. Each entry in the index structures consists of 8Bytes; 4Bytes for the tuple index, which is a pointer to the tuple in the R-relation, and 4Bytes to store the hash value for that tuple. We partition the R-relation by first splitting it between the two threads, such that the first thread processes the first half ($R_0-R_{(n/2)-1}$) and the second thread processes the second half ($R_{n/2}-R_{n-1}$). The R-relation is accessed sequentially by each thread. Therefore, the hardware prefetcher is able to capture the memory address patterns and prefetch the needed data. This eliminates the need for explicit software prefetch instructions. Each thread in this stage reads a tuple from its half and calculates the tuple's key mod number of clusters that belong to this thread. Therefore, it chooses the cluster where it should store the tuple. The thread saves the tuple's pointer together with its hash value, which is calculated from the tuple's key. We are using 1024 clusters for the index partition. This generates L1 cache size clusters (L1 cache size is 64KByte). Later in this section we explain an automated method to determine numbers of clusters prior to actual execution.

The Build and the Probe Index Partition Phase: Before we begin this stage, we make sure that both threads finish the build index partition phase completely by using a barrier synchronization pragma. Our hash tables are described in Figure 3. We study several possible multithreaded implementations. 1) Use the two threads simultaneously, each building a hash table. This approach resulted in contention over the cache between the two threads hash tables. Thus resulting in cache misses for most accesses in the two hash tables and highly degrading performance. 2) Use the two threads to build the same hash table simultaneously. We use atomic synchronization pragmas to restrict writing to the same memory location to one thread at a time.

```

for i = 0 until i = total-number-of-clusters/2
    for j = 0 until j = thread0.Build-clusteri.number-of-entries-1
        insert thread0.Build-clusteri.tuplej into hash-tablei
    for k = 0 until k = thread1.Build-clusteri.number-of-entries-1
        insert thread1.Build-clusteri.tuplek into hash-tablei

```

Figure 5: AA_HJ Build Phase Executed by one Thread

However, this type of synchronization limits the performance of the two threads, resulting in slowdowns rather than speedups. 3) Devoting one thread to create the hash tables of the build phase and use the second thread to perform the S-relation index partitioning phase simultaneously. This method gives us the best performance and therefore, is our method of choice.

The build phase algorithm is shown in Figure 5. Each two clusters generate one hash table, where both of these two clusters have the same key-range. For example, both thread₀.Build-cluster₁ (cluster₁ generated by thread₀ from the first phase) and thread₁.Build-cluster₁ (cluster₁ generated by thread₁ from the first phase) generate hash-table₁ in Figure 5. While the first thread is building the hash tables, we use the second thread to perform the S-relation index partitioning simultaneously. The R-relation structures will be accessed repeatedly to probe tuples in the probe phase, thus they need to fit in one of our caches. While for S-relation, each tuple will be read once during the probing phase to search for its match, so the S-relation clusters do not need to fit in the caches. Also, since tuples are read sequentially, the hardware prefetcher is able to prefetch the S-relation tuples. Each entry in the S-relation clusters has a similar form to that used for the R-relation clusters. We create two sets of clusters, one for each thread. The first set of clusters store the indexes resulting from tuples ranging from 0 to $(n/2)-1$, where n is the total number of tuples in the S-relation. While the second set of clusters stores indexes from $(n/2)$ to $n-1$. Therefore, each key-range has two clusters, one from the first S-relation half and the other from the second half. The algorithm used for the S-relation indexing phase is shown in Figure 6 (where S means S-relation).

```

x=0
do{
    read S.tuplex
    z = appropriate-cluster-number depending on S.tuplex.key
    insert S.tuplex into thread0.Probe-clusterz
    read S.tuplex+(n/2)
    z = appropriate-cluster-number depending on S.tuplex+n/2.key
    insert S.tuplex+(n/2) into thread1.Probe-clusterz
    increment x by 1
} while ( x < n/2 )

```

Figure 6: AA_HJ Probe Index Partitioning Phase Executed by one Thread

The Probe Phase: As the probing phase uses both the hash tables and the S-relation clusters, we can not begin this phase until both threads of the previous phase are done. Thus, a barrier pragma is implemented between the two phases. One of the large challenges for the probe phase is the random accesses to the hash table whenever there is search for a potential match. As described in Section 2: Figure 3, each access to the hash table will result in a sequence of pointers dereferenced. The probe phase begins by accessing the appropriate bucket, reading the cell array's pointer, accessing the cell array and dereferencing every cell's pointer so as to read this tuple's key and test for a match with the probed tuple. Consequently, the goal of optimizing this phase concentrates on proposing a solution for the sequence of random accesses to the hash tables. Architectural Aware Hash Join (AA_HJ) controls both threads such that each thread is probing tuples from its cluster whose key-range is similar to another cluster that is being probed by the other thread concurrently. As an example, in Figure 7, we show the process of generating four clusters from the S-relation in the S-relation index partitioning phase by Thread1. Thread2 will be busy in hash tables building (not shown in the figure). Next, in the probe phase the two clusters that belong to the same key-range are probed by the two threads simultaneously and one hash table is visited

during each key-range’s iteration. To prevent race conditions that might arise from one thread probing its cluster faster than the other thread, we divide each key-range probe iteration with a barrier pragma from the other iterations. However, since keys are randomly distributed throughout the S-relation, each cluster from thread₀’s set of clusters will result in almost the same number of matches as those resulted from the corresponding cluster from thread₁’s set of clusters. Thus, probing both clusters requires the same time. The pseudo code for our algorithm is shown in Figure 8. The term “number-of-clusters” refers to the total number of clusters generated from the S-relation. Since both threads are using the same hash table concurrently in each iteration, one thread will serve as an implicit hash table-prefetcher for the other thread while it is probing its own tuples.

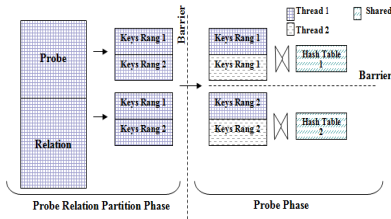


Figure 7: AA_HJ S-Relation Partitioning and Probing Phases

This is because each hash table fits in the L1-cache therefore, once it is fetched, it remains cache-resident until the next iteration, where another hash table is prefetched. The original S-relation is not accessed sequentially any more because of our index partitioning, therefore the hardware prefetcher will not be as useful. To solve this problem, we use explicit prefetch instructions to prefetch the next tuple in the cluster before we begin to process the current tuple. We find that prefetching one tuple ahead is enough to overlap the memory access latency for the tuple.

```

for i = 0 until i = number-of-clusters/2
  if (thread0)
    for j = 0 until j = thread0.Probe-clusteri.number-
of-entries
      prefetch thread0.Probe-clusteri.tuplej+1
      use hash-tablei to probe thread0.Probe-
clusteri.tuplej
    else
      for k = 0 until k = thread1.Probe-clusteri.number-
of-entries
        prefetch thread1.Probe-clusteri.tuplek+1
        use hash-tablei to probe thread1.Probe-
clusteri.tuplek
      pragma barrier

```

Figure 8: AA_HJ Multithreaded Probing Algorithm

This is because each prefetch instruction in the Intel[®] Pentium[®] 4 loads two cache lines and the largest tuple size we study is 140Bytes.

Extending AA_HJ for more than two threads: We now present a scalable form of AA_HJ that exploits more than two threads. Our new version of AA_HJ is capable of utilizing various types of multithreading including SMP (Symmetric Multiprocessors), CMP

and SMT. Follows is a description of the changes we have made to the dual-thread AA_HJ:

R-relation index partition: Assume that the R-relation has R_n tuples. Also, assume that the number of available threads in the platform is t , t includes any threads resulting from the SMT, the CMP and the SMP architectures, where $t = \text{number of processor chips} \times \text{number of cores per chip} \times \text{number of SMT threads per processor core}$. For example, if a system has four processor chips, each processor is a quad core, each core is 2-threads SMT, then $t = 4 \times 4 \times 2 = 32$ threads. Each thread t_i ($i = 0, 1 \dots t-1$) is assigned R_n/t tuples. The remaining tuples after this division will be added to the last thread. An index partitioning similar to the one described in earlier in this section is executed by each thread. By the end of this phase any thread will have a set of clusters c . A c_i ($i = 0, 1 \dots \text{limit}-1$) stands for a key-range as described earlier. The value of *limit* depends on the following observations: (1) the total size of clusters for any key-range must be small enough to allow both the hash table and its R-clusters to fit in the L2 cache. This is because we are planning for each four threads in a chip to share one hash table. (2) During the probe phase some space in the L2 cache should be reserved for a few tuples from the S cluster. The tuples from the S-relation are used only once, so this space is intended to be a temporary storage for tuples prefetched manually. (3) Some space should be reserved for the operating system processes. Taking all these factors into account, we use $(R\text{-relation-size} + \text{hash tables'sizes})/\text{limit} < \text{L2 cache size}$ to calculate *limit*. Since it is difficult to estimate the hash table size prior to the hash join (hash table size ranges between 22MByte up to 150MByte in our case) we use its worst case, where hash table is just above half the R-relation size (150MByte). Therefore *limit* is measured as follows $(250 + 150) / \text{limit} < 2$, which results in *limit* > 200. We choose *limit* to be 256 clusters.

Build Phase and S-Relation Index Partition Phase: In the second step, the thread with the smallest identifier builds the hash tables. Simultaneously, other threads will index-partition the S-relation as described in earlier in this section. The thread with the next smallest identifier will generate two sets of clusters instead of one, to compensate for the thread building the hash tables.

Probe Phase: During this phase, a constructive cache-level sharing is maintained by directing all four threads of each dual-SMT-core to probe one key-range using one hash table. Recall that any thread generates a set of clusters from phase two, with a cluster for each key-range. Therefore, t clusters exist for each key range. A probing thread in a core will process $t/4$ clusters. Again, GP is incorporated with this phase code to eliminate cold misses. Keeping in mind that this is the most expensive phase in the hash join operation, we provide several optimizations including: (1) the load is almost perfectly balanced between threads. Given that our keys are uniformly distributed, the sizes of clusters are very close. (2) Having four threads repeatedly visiting the same memory structure (hash table), will highly increase temporal and spatial locality.

4. EXPERIMENTAL METHODOLOGY

We run our algorithms on two multithreaded machines. The first (Machine 1) is a 3.4GHz Intel[®] Pentium[®] 4 processor with hyper-threading technology (HT, Intel’s dual thread SMT architecture [11]). The second (Machine 2) is the Intel[®] Xeon[®] Quad Processors for PowerEdge 6800, each processor is a dual-core, each core is HT. General specifications for both machines are shown in Table 1.

Both systems have L2 unified cache with 128Bytes cache lines. We use the Scientific Linux version 4.1 operating system which is based on the Redhat Linux Enterprise version 4.0. We implemented all algorithms in C, and we use the Intel® C++ Compiler for Linux version 9.1 [10] with maximum optimizations. We use the built-in OpenMP C/C++ library [19] version 2.5 (as implemented in the Intel® C++ Compiler) to initiate multiple threads in our multi-threaded codes. We repeat each run three times, remove the outliers, and take the average. Timing and memory measurements are done through our program using functions such as `gettimeofday()`. A warm up run is done prior to any measurements to load the relations into main memory.

Table 1: Machines Specifications

	Machine 1	Machine 2
Processor(s)	Pentium® 4 with HT	Quad Xeon®, PowerEdge 6800
L1 data Cache	64Kbyte	64KByte/core
L2 Cache	2MByte	2MByte/processor
Main Memory	1GByte 533MHZ DDR2	4GByte 400MHZ DDR2
Clock Speed	3.4 GHz	2.66 GHz
Hard Drive	160GByte	300GByte

We choose to implement our own version from hash join rather than using the database benchmarks (e.g. TPC-C) to prevent the impact of DBMS overhead from unseen activities. These activities might include query planner, query optimizer, etc. For Machine 1 we use a 50MByte build relation and a 100MByte probe relation. We choose these sizes to make sure that our relations, in addition to any large intermediate structures needed by the code, fit in our 1GByte main memory.

Table 2: Number of Tuples for Machine 1 and 2

Tuple Size (Byte)	Build Relation in Machine 1	Probe Relation in Machine 1	Build Relation in Machine 2	Probe Relation in Machine 2
20	2621440	5242880	13107200	26214400
60	873814	1747628	4369067	8738134
100	524289	1048578	2621440	5242880
140	374491	748982	1872457	3744914

While for Machine 2 we use 250MByte build relation and 500MByte probe relation since we have larger main memory (4GByte). Our join key is 10Bytes, randomly generated such that each tuple in the build relation matches one tuple in the probe relation. The payload part of the tuple is of variable size. The number of tuples in each table (given the table’s constant size) depends on the tuple size. Table 2 shows the number of tuples used in each relation for different tuple sizes. We choose tuples of these sizes to study the cases where tuples are smaller than the L1 cache line (20Byte, 60Byte), between the L1 and the L2 cache line sizes (100Byte) and larger than the L2 cache line size (140Byte). In real DBMS the average tuple size is 120Byte [22]. Our naïve partitioning and probing algorithms are the same as those in [9]. Our hash function consists of XOR and shift operations [4] and generates 4Bytes hash codes. Once hash codes are computed at any stage, they are saved in temporary structures in memory to avoid recalculating them. Hash table buckets are calculated using the hash

code mod hash table size. Our hash tables are created such that the number of buckets equals the number of tuples in the corresponding R-cluster or the R relation in case partitioning is not used. We use the Intel® VTune™ Performance Analyzer for Linux 9.0 [12] to collect the hardware events from the hardware performance counters available in our machines. These events include L2 cache load misses, L1 data cache load misses, etc. Each run for VTune is repeated three times. Each time two runs are performed by VTune, the first is for calibration, which determines the frequency at which the event occurs. The second is for the actual event collection.

5. RESULTS FOR THE DUAL-THREADED HASH JOIN

We start by characterizing the main memory-hash join algorithm in Machines 2. Figure 9 shows that the level one (L1) data cache load miss rate (= L1 load cache misses/total loads) ranges from 4.7% to 5.3% while varying the tuple (record) size. Taking into account that the L1 miss latency does not exceed 10 cycles, we find that the L1 data cache does not affect the overall performance of the hash join. Next, we characterize the unified level two (L2) cache in Figure 10. The L2 cache load miss rate (= L2 load cache misses/L1 load cache misses) varies from 29% for tuple size = 140Bytes to 64% for tuple size = 20Bytes. As the L2 cache load miss latency is usually larger than 100 cycles, our results show that the L2 cache load miss rate is a critical factor in main-memory hash join performance. This agrees with previous research in [1]. Our measurements show that the maximum TC miss rate we get is very small and does not exceed 0.14%. In summary, the L2 cache miss rate has a major impact on the hash join performance. Therefore, reducing the L2 cache miss rate is vital to improve the hash join performance.

Partitioning vs. Non-Partitioning vs. Index Partitioning: Now, we study the effects of partitioning the build and probe relations on the execution time and memory usage of the hash join on Machine 1. As described in Figure 4, partitioning is the first step of the hash join algorithm. It creates small clusters (partitions) of the R and S-relations that fit in the cache. The goal is to divide the overall hash join into a set of smaller hash joins to work on data that fits in the cache. Recent papers ([4], [9]) copy the entire relations while partitioning. We implement three types of the hash join algorithms: partitioning (PT), non-partitioning (NPT), and index partitioning (Index PT): (1) PT uses the copy partitioning algorithm described in Figure 4. We use 1024 clusters for the R-relation which creates R-clusters of 50KByte each. Including the hash table size for each cluster, this fits easily in our 64KByte L1 cache. We find experimentally that using clusters of larger sizes will create cache thrashing, and smaller cluster sizes result in high partitioning overhead. We also use 1024 clusters for the S-relation. However, the S-relation is not as critical as the R-relation to have in the cache and is not partitioned to fit in the cache in techniques such as DC, described in Section 3. (2) NPT uses no partitioning, instead the full R and S-relations are hash joined. (3) Index PT. Instead of copying the actual tuples into the partition, pointers to the tuples are stored. We use 1024 clusters for both the R- and S-relations which allows our R-cluster (which includes pointer to the tuples only and not the full tuple) and its corresponding hash table to fit into our L1 cache. The PT and Index PT are two variants from main-memory Grace hash join. Our results in Figure 11 demonstrate that although the timing analysis for PT outperforms the NPT algorithm in tuple size = 20Byte, the overhead of the partitioning phase overcomes the

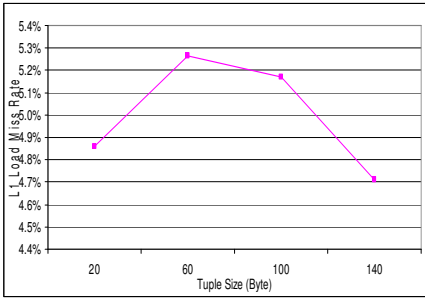


Figure 9: The L1 Data Cache Load Miss Rate for Hash Join

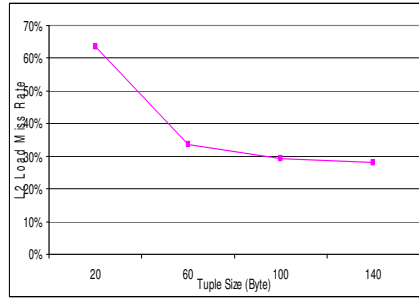


Figure 10: The L2 Cache Load Miss Rate for Hash Join

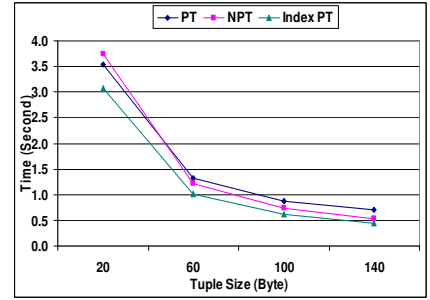


Figure 11: Timing for three Hash Join Partitioning Techniques

performance improvement due to partitioning in all other tuple sizes. This overhead is a result of the copying of large tuples from the source relation to the destination cluster. This overhead is eliminated by Index PT and therefore results in the performance improvement of Index PT over both NPT and PT in all tuple sizes. The longer execution time for smaller tuples is due to the larger number of tuples (as a result, more random accesses to caches for smaller tuples) in these cases as shown in Table 2.

Index PT by dividing the available S-clusters evenly between both threads to create SMT+PT and SMT+Index PT, respectively. While for NPT we split the probe relation between the two threads, such that each thread probes half of the large relation. Our results in Figure 12 show that Index PT (in the SMT+Index PT algorithm) continues to give the best performance. We calculate the speedups resulting from multithreading each of our three hash join algorithms to be; SMT+NPT is 39%, SMT+PT is 10% and SMT+Index PT is 15%, compared to NPT, PT and Index PT hash joins, respectively.

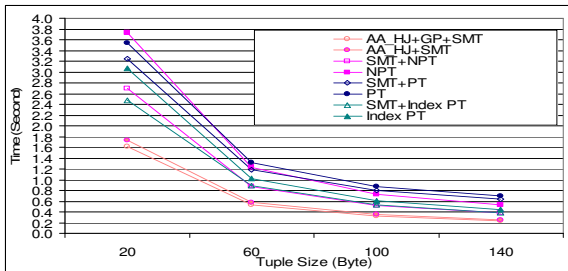


Figure 12: Timing Comparison of all Hash Join Algorithms

The memory usage of PT, NPT and Index PT: Since we are studying MMDB operations, our relations have to be main memory resident prior to any processing. Thus, the minimum memory space that any hash join requires will be equal to the total sizes of the two relations, which is 150MB, in addition to the memory needed to build the hash table. The size of the hash table(s) is proportional to the number of tuples involved in the table building. Our results show that PT requires almost two times the memory space required by NPT. This is because both relations are copied into the clusters in PT. While, Index PT memory requirements are in between PT and NPT, as each tuple in Index PT requires only 8Bytes in its cluster (4Bytes for tuple hash value and 4Bytes tuple pointer), regardless of the size of the tuple. Therefore, Index PT gives the best performance and has the intermediate memory usage. Speedups achieved from Index PT over NPT ranges from 18% to 21%.

Dual-threaded Hash Join: The probe phase is known to be the most time consuming phase in hash join due to its random access pattern to both the hash table and R-relation. Now, we study the performance of the straightforward parallelization of the probe phase on Machine 1. We develop dual-threaded versions of the three algorithms presented in the Section 3 on our SMT architecture. We refer to our algorithms as SMT+PT, SMT+NPT and SMT+Index PT. We parallelize the probe phase for PT and

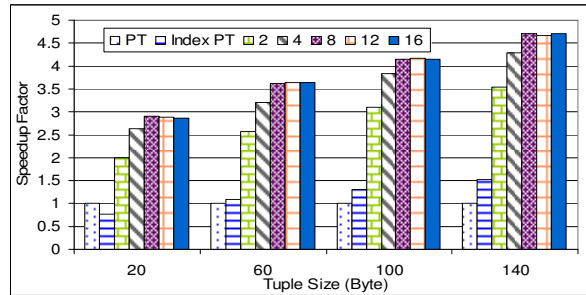


Figure 13: Speedups for the Multi-Threaded Architecture-Aware Hash Join

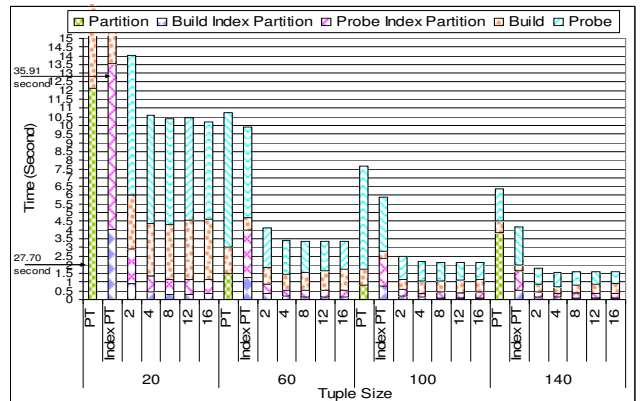


Figure 14: Time Breakdown Comparison for Hash Join Algorithms

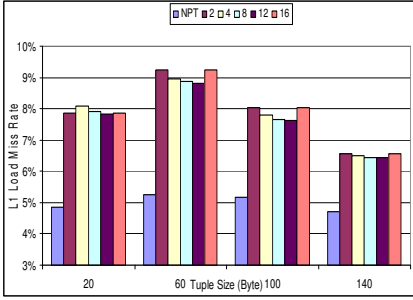


Figure 15: The L1 Data Cache Load Miss Rate for NPT and AA_HJ

The SMT+NPT has the highest speedups since it lacks the overhead of the sequential partitioning phases and its execution time is dominated by the probing phase. We use Index Partitioning in AA_HJ as it is the best performing partitioning algorithm. In contrast to the SMT+Index PT where two hash tables (one per thread) are used, AA_HJ forces the two threads to use the same hash table simultaneously. This reduces cache conflicts between the two hash tables in SMT+Index PT and allows accesses from one thread to prefetch parts of the table for the other thread. We refer to this version of our proposed algorithm as AA_HJ+SMT. Since our proposed technique is orthogonal to some of the previously proposed hash join enhancement techniques such as Group Prefetching (GP) [4], we further enhance our performance by adding GP to AA_HJ+SMT. GP prefetches the randomly accessed buckets of the hash tables, thus reducing our cold cache misses. We refer to this version of our proposed algorithm as AA_HJ+GP+SMT. Figure 12 shows that AA_HJ+SMT is able to increase the thread cooperation in the cache level for all tuple sizes and therefore considerably improve performance. AA_HJ+GP+SMT further enhances the performance. AA_HJ+SMT achieves a speedup ranging from 2.04 to 2.70 for tuple sizes 20Bytes to 140Bytes, respectively. Speedup for AA_HJ+GP+SMT ranges from 2.19 to 2.90 for tuple sizes 20Bytes to 140Bytes, respectively.

6. RESULTS FOR THE MULTI-THREADED ARCHITECTURE-AWARE HASH JOIN

In this section we present the results of our scalable AA_HJ algorithm. The workstation we conducted our experiments on is Machine 2, described in Section 4. The R-relation is 250MByte and the S-relation is 500MByte. We ran multithreaded AA_HJ with 2, 4, 8, 12 and 16 threads, each of which with tuple size = 20Byte, 60Byte, 100Byte and 140Bytes. To highlight the differences in performance with single-threaded AA_HJ, we also run NPT, PT and Index PT hash joins (for details about NPT and PT and Index PT refer to Section 5). Figure 13 shows the speedups of all multithreaded runs together with Index PT compared to PT hash join. We achieve speedups ranging from 2x for tuple size = 20Bytes with two threads, to 4.6x for tuple size = 140Bytes with 16 threads. The improvements in running time saturate while having eight threads for all tuple sizes. This is because number of clusters has proportional relation with number of threads. Therefore, the partitioning overhead together with the expensive off-chip communications will increase while having more working threads. AA_HJ takes advantage of sharing structures between each four threads in a dual-SMT core. Thus, enhancements in performance are large while having 2, 4 and 8

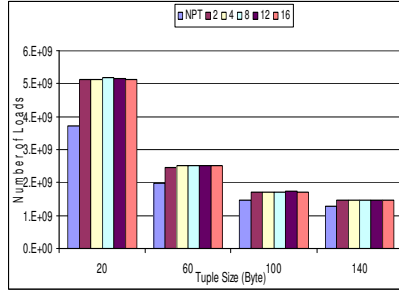


Figure 16: Number of Loads for NPT and AA_HJ

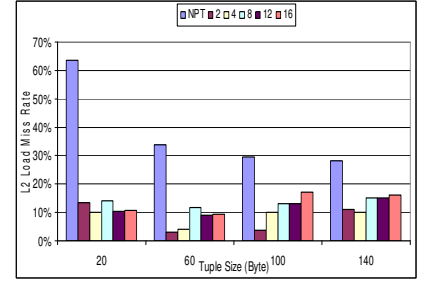


Figure 17: The L2 Cache Load Miss Rate for NPT and AA_HJ

threads. Despite the fact that PT accomplishes good execution time for tuple size = 20Bytes, its memory footprint is 3.4 times the relations sizes, which makes it impractical for machines with limited main memory environments. NPT hash join maintains its precedence over others in memory savings. While Index PT and all AA_HJ multithreaded hash joins are comparable in memory consumption. Figure 14 shows the time breakdown for all multithreaded AA_HJ, PT and Index PT for all tuple sizes. From Figure 14 we have the following observations: There are large improvements in probing and index-partitioning execution times for AA_HJ compared to PT and Index PT. Execution time decreases when using more threads for AA_HJ up to 8 threads, where it saturates. The R- and S-relations index partitioning phases saturate at eight-threaded AA_HJ. This is due to doubling number of clusters while adding each thread. The communication overhead for CPUs on the same chip is cheap (10-20 cycles) and is carried out through the L2 cache. While off-chip cores communicate through the main memory or a cache-coherence protocol which is very expensive (hundreds of cycles). In the probe phase clusters are collected from all cores to process a hash table, this generates large communication overhead that prevents further improvements.

In what follows, we use Intel VTune Performance Analyzer for Linux 9.0 to collect hardware events from the hardware counters available on Machine 2. First, we measure the L1 data cache load miss rate in Figure 15. NPT hash join is always generating low L1 data load miss rate, due to the low number of loads it executes (Figure 16). The relatively small number of loads is a direct effect for not using any intermediate structures but one hash table and to accessing both R and S-relations sequentially. The L1 data cache miss rate for multi-threaded AA_HJ decreases as we increase the tuple size except for tuple size = 20Bytes. Since number of tuples are smaller for larger tuple sizes Table 2. Therefore, hash joins with large tuple sizes process fewer movements while partitioning and probing. The L1 data cache load miss rate for NPT is about 5%, and for multi-threaded AA_HJ is from 6.5% to 9.1% showing an increase of 1.5% to 4%. This increase is very small and therefore has a minor affect on the overall performance. In Figure 17, we measure the L2 cache load miss rate. NPT has over a 60% L2 load miss rate at tuple size = 20Bytes. This is a result of the very large probe portion for the NPT for tuple size = 20Bytes, since it uses one hash table. All tuple sizes in Figure 17 are experiencing an improvement in L2 load miss rate, which is the dominating factor in the execution time and the main cause for the performance improvement. A noticeable decrease exists from NPT to two-threaded AA_HJ, due to the cache-sized index partitioning,

good load balance between both threads and constructive cache sharing.

7. CONCLUSIONS

In this paper we characterize the hash join operation on state_of_the_art multithreaded hardware that combines SMT, CMP and SMP. We find that the hash join is bound by the L2 miss rates, which range from 29% to 62%. We propose an Architecture-Aware Hash Join (AA_HJ) that relies on sharing critical structures between working threads at the cache level, benefiting from multithreaded architectural features. AA_HJ distributes the load evenly between threads while requiring almost the same memory space used by index-partitioning hash join. We study AA_HJ performance on two machines. The first is a two-threaded SMT processor where we achieve speedups ranging from 2.1 to 2.9 compared to the single-threaded hash join. The second is a quad dual-SMT-core server (with a total of 16 threads); we obtained speedups from 2 to 4.6 compared to the single-threaded hash join. We find that AA_HJ decreases the L2 cache miss rate from 62% to 11%, and from 29% to 15% for tuple size = 20Bytes and 140Bytes, respectively.

References

- [1] Ailamaki, A., DeWitt, D.J., Hill, M.D. and Wood, D.A. DBMSs on a Modern Processor: Where Does Time Go?. In Proceedings of the 25th International Conference on Very Large Data Bases (VLDB). Pages: 266-277, 1999.
- [2] Belzer, Jack. Very Large Data Base Systems to Zero-Memory and Markov Information Source. Encyclopedia of Computer Science and Technology, Volume 14.
- [3] Blleloch, G. and Gibbons, P. Effectively Sharing a Cache among Threads. Symposium on Parallelism in Algorithms and Architectures (SPAA). 2004.
- [4] Chen, S., Ailamaki, A., Gibbons, P. and Mowry, T. Improving Hash Join Performance through Prefetching. In IEEE International Conference on Data Engineering (ICDE). Page: 116-128, 2004.
- [5] Cieslewicz, J., Berry, J., Hendrickson, B. and Ross, K.A. Realizing Parallelism in Database Operations: Insights from a Massively Multithreaded Architecture. In Proceedings of the 2nd international workshop on Data Management on New Hardware (DAMON). Article No. 4, 2006.
- [6] Codd, E.F. A Relational Model of Data for Large Shared Data Banks. ACM, Vol. 13, No. 6, 1970.
- [7] Curtis-Maury, M., Ding, X., Antonopoulos, C. and Nikolopoulos, D. An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. In International Workshop on OpenMP (IWOMP). May, 2005.
- [8] Fushimi, S., Kitsuregawa, M. and Tanaka, H. An Overview of the System Software of a Parallel Relational Database Machine Grace. In Proceedings of International Conference on Very Large Data Bases (VLDB), 1986.
- [9] Garcia, P. and Korth, H. Database Hash-Join Algorithms on Multithreaded Computer Architectures. In Proceedings of Computing Frontiers (CF). Pages: 241 - 252, 2006.
- [10] Intel C++ Compiler for Linux. URL: <http://www.intel.com/cd/software/products/asm-na/eng/compilers/277618.htm>
- [11] Intel Hyper-Threading Technology. URL: http://www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf
- [12] Intel[®] VTune Performance Analyzer for Linux. URL: <http://www.intel.com/software/products/vtune/>.
- [13] Kim, W., Gajsk, D. and Kuck, J.D. A Parallel Pipelined Relational Query Processor. ACM Trans. On Data-Base Systems, 9 (2). Pages: 214-242, 1984.
- [14] Kitsuregawa, M., Tanaka, H. and Moto-Oka, T. Application of Hash to Data Base Machine and its Architecture. New Generation Computing, 1983.
- [15] Lo, J.L., Barroso, L.A., Eggers, S.J., Gharachorloo, K., Levy, H.M., and Parekh, S.S. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In Proceedings of International Symposium on Computer Architecture (ISCA) Conference, 1998.
- [16] Lu, H., Tan K. and Shan, M. Hash-Based Algorithms for Multiprocessor Computers with Shared Memory. In Proceedings of the 16th international conference on Very Large Data Bases (VLDB). Pages: 198-209, 1990.
- [17] Manegold, S., Boncz, P.A. and Kersten, M.L. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In Proceedings of International Conference on Very Large Data Bases (VLDB). Pages: 339 – 350, 2000.
- [18] McDowell, L., Eggers, S. and Gribble, S. D. Improving Server Software Support for Simultaneous Multithreaded Processors. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) and workshop on partial evaluation and semantics-based program manipulation. Pages: 37 – 48, 2003.
- [19] OpenMP[®]. URL: <http://www.openmp.org/>
- [20] Shatdal, A. Architectural Considerations for Parallel Query Evaluation Algorithms. PhD thesis, 1996.
- [21] Shatdal, A., Kant, C. and Naughton, J.F. Cache Conscious Algorithms for Relational Query Processing. In Proceedings of International Conference on Very Large Data Bases (VLDB). Pages: 510 – 521, 1994.
- [22] Shao, M., Ailamaki, A. and Falsafi, B. "DBmbench: Fast and Accurate Database Workload Representation on Modern Microarchitecture". In Proceedings of the Centre for Advanced Studies on Collaborative research conference. Pages: 254 – 267, 2005.
- [23] Tullsen, D., Eggers, S., Levy, H. Simultaneous Multithreading: Maximizing on-Chip Parallelism. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, (ISCA), 1995.
- [24] Zhou, J., Cieslewicz, J., Ross, K., and Shah, M. Improving Database Performance on Simultaneous Multithreading Processors. In Proceedings of International Conference on Very Large Data Bases (VLDB). Pages: 49 – 60, 2006.
- [25] Zukowski, M., Héman, S. and Boncz, P. Architecture-Conscious Hashing. In Proceedings of the 2nd international workshop on Data Management on New Hardware (DAMON). Article No. 6, 2006.
- [26] Jack Belzer. Encyclopedia of Computer Science and Technology - Volume 14. Very Large Data Base Systems to Zero-Memory and Markov Information Source. Marcel Dekker Inc., ISBN 0-8247-2214-0.
- [27] Hammond, L., Nayfeh, B. and Olukotun, K. A Single Chip Multiprocessor. IEEE Computer, 30(9). Pages: 79-85, 1997.
- [28] Liaskovitis, V. et al. Parallel Depth First vs. Work Stealing Schedulers on CMP Architectures. In Proceedings of the 18th Symposium on Parallelism in Algorithms and Architectures (SPAA). Pages: 330 – 330, 2007.
- [29] Colohan, C., Ailamaki, A., Steffan, J. and Mowry, T. Optimistic intra-transaction parallelism on chip multiprocessors. In Proceedings of international conference on Very Large Data Bases (VLDB), 2005.