Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs

Changkyu Kim[†] Eric Sedlar[⋆] Jatin Chhugani[†]
Tim Kaldewey[⋆] Anthony D. Nguyen[†] Andrea Di Blas[⋆]
Victor W. Lee[†] Nadathur Satish[†] Pradeep Dubey[†]

Contact: changkyu.kim@intel.com
[†]Throughput Computing Lab, Intel Corporation
[⋆]Special Projects Group, Oracle Corporation

ABSTRACT

Join is an important database operation. As computer architectures evolve, the best join algorithm may change hand. This paper re-examines two popular join algorithms – hash join and sort-merge join – to determine if the latest computer architecture trends shift the tide that has favored hash join for many years. For a fair comparison, we implemented the most optimized parallel version of both algorithms on the latest Intel Core i7 platform. Both implementations scale well with the number of cores in the system and take advantages of latest processor features for performance. Our hash-based implementation achieves more than 100M tuples per second which is 17X faster than the best reported performance on CPUs and 8X faster than that reported for GPUs. Moreover, the performance of our hash join implementation is consistent over a wide range of input data sizes from 64K to 128M tuples and is not affected by data skew. We compare this implementation to our highly optimized sort-based implementation that achieves 47M to 80M tuples per second. We developed analytical models to study how both algorithms would scale with upcoming processor architecture trends. Our analysis projects that current architectural trends of wider SIMD, more cores, and smaller memory bandwidth per core imply better scalability potential for sort-merge join. Consequently, sort-merge join is likely to outperform hash join on upcoming chip multiprocessors. In summary, we offer multicore implementations of hash join and sort-merge join which consistently outperform all previously reported results. We further conclude that the tide that favors the hash join algorithm has not changed yet, but the change is just around the corner.

1. INTRODUCTION

Join is a key operation in relational databases that facilitates the combination of two relations based on a common key. Join is an expensive operation and an efficient implementation will improve the performance of many database queries. There are two common join algorithms: sort-merge join and hash join. The debate over which is the best join algorithm has been going on for decades. Currently, for in-memory database operations, the hash-join algorithm has been shown to outperform sort-merge join in many cases.

Today's commodity hardware already provides large degrees of parallelism, with multiple cores on a single chip, multiple hardware threads on each core (SMT) and vector instructions (SIMD) operating on 128-bit vectors whose capability will increase in the near future. Coupled with growing compute density of chip multiprocessors (CMP) and memory bandwidth challenges, it is not clear if hash join will continue to ourperform sort-merge join. In this paper, we re-examine both algorithms under the context of CMP that offers thread-level parallelism (TLP), data-level parallelism (DLP), large on-die caches, and high memory bandwidth.

For a fair comparison, we optimize the implementations of both algorithms for the latest multi-core platform. Our hash-based implemenation can join 100 million tuples per second on a 3.2GHz quad-core Intel Core i7 965 platform which is faster than any reported CPU implemenation thus far. We implemented sort-merge join algorithm by exploiting all salient features of CMP such as TLP, DLP, blocking for on-die caches, and utilizing high memory bandwidth judiciously. Moreover, our implementations are tolerant of data skew without sacrificing performance.

In our study, we observe a number of interesting features of join implementations. First, both join algorithms benefit greatly from multi-threading. Second, sort-merge join benefits greatly by exploiting SIMD architecture offered by today's processor. Its performance will continue to improve with the trend of wider SIMD [21, 34]. Based on our analytical model, we project that sort-merge join will surpass hash join with 512-bit SIMD. For hash join to make use of SIMD execution, we believe that hardware support for efficient scatter and atomic vector operations are necessary (Section 7). Finally, by efficiently managing memory bandwidth usage, we show that both hash join and sort-merge join are compute bounded on today's CMP system. However, our analytical model shows that hash join demands at a minimum 1.5X more memory bandwidth than sort merge join. If the gap between compute and bandwidth continues to grow for the future computer systems, the sort-merge join will be more efficient than hash join.

Our contributions include: first, we implement the most efficient hash join and sort-merge join on the latest computer platform. The performance of hash join is 17X faster than the best published CPU numbers and 8X faster than that reported for GPUs [18]. Second, our join performance is constant for a wide range of input sizes and data skews. Third, we compare the performance of sort-merge and hash join for current architectures, and conclude that hash-join is superior in performance. Fourth, by constructing an analytical model for both hash join and sort-merge join, we conclude that future architectural trends towards increasing SIMD width and limited per-core memory bandwidth will favor sort-merge join versus hash join.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 examines modern computer architectures and their implications on the join performance. Section 4 presents our hash join algorithm and the considerations for parallelization. Section 5 describes our sort-merge join implementation and the considerations for parallelization. Section 6 presents the results on the two join implementations. Section 7 discusses architecture improvements that are beneficial to both join algorithms and discusses future architecture trends that would influence the performance of hash join and sort-merge join. Section 8 concludes.

2. RELATED WORK

Over the past few decades, significant efforts have been made to develop efficient join algorithms. Among the algorithms developed, sort-merge join and hash join algorithms are two most popular algorithms for computing the equi-join of two relations. The sort-merge join algorithm [3] was dominantly used in early relational database systems. Later, the hash join algorithm became popular and was shown to outperform sort merge join in many situations. The Grace hash join [23] and the hybrid hash join [38] algorithms were first proposed to overcome the disk I/O overhead of general hash-based join algorithms. As the capacity of main memories increased over the years, researchers have focused on main-memory join operations [35, 5, 27]. Shatdal et al. [35] propose a cache partitioning algorithm, where they partition the hash table to fit in cache memory so as to reduce the latency of access to the hash table. Manegold et al. [27, 28] observe that the partitioning stage itself incurs a lot of TLB and cache misses and becomes the performance bottleneck when the size of relations is too large and there are too many partitions required for each to fit in cache. They propose a radix-clustering algorithm that fixes the number of partitions based on the number of TLB entries and performs a partial radix sorting. When the total number of partitions is greater than some fixed number, they perform multi-pass partitioning. Our implementation also relies on a similar multi-pass partitioning scheme. In contrast to algorithms based on cache partitioning, Chen et al. [6] argue that the trend towards concurrent databases will create more cache conflicts, thus reducing the effectiveness of caches. Instead of exploiting caches, they propose to use software prefetch schemes to hide the long latency of accessing hash tables. However, as memory bandwidth becomes an important consideration for performance, partitioning-based schemes that attempt to maintain the working set in cache will still retain an advantage. In this work, we use a partitioning-based scheme to reduce memory bandwidth use.

While the above join implementations are based on sequential algorithms, there has been considerable research on parallel partitioning [9] and join algorithms [10, 26]. One key issue in parallel joins is to achieve good load-balancing, especially when data are skewed. Different schemes to handle data skew in parallel joins have been proposed for both sort-merge join and hash join algorithms [20, 37, 36]. Unlike prior algorithms, our implementation does not require an extra tuning and scheduling phase to address the problem of data skew.

Recently, researchers have explored new architectures to improve join performance. He at al. [18] present GPU-based implementations of both sort-merge join and hash-join. Gedik et al. [12] optimize join code for the Cell processors. Both papers try to exploit the parallel nature of these devices with associated high compute density and bandwidth and show significant performance benefits (from 2-7X on the GPU and 8X on Cell) over optimized CPU-based counterparts.

Current trends in general purpose CPUs have also been in the direction of increasing parallelism, both in terms of the number of cores on a chip and with respect to the SIMD width on each core. Chip multiprocessors are different from conventional multiprocessor systems in that inter-thread communication is much faster with shared on-chip caches [17] and the cost for thread synchronization and atomic operations is much lower. Cieslewicz et al. [8] examine aggregation operations on 8-core chip multiprocessors and exploit thread-level parallelism (TLP) and the shared on-chip caches for high performance aggregation. Zhou et al. [39] implement various database operations to exploit data-level parallelism (DLP) with SIMD instructions. In this work, we show that by efficiently exploiting the capabilities of modern CPUs, we can obtain significant performance advantages over previous join implementations.

Sort-merge join is highly dependent on a good sort implementation. Quicksort is one of the fastest algorithms in practice, but it is unclear whether it can be mapped efficiently to the SIMD architecture. In contrast, bitonic sort [2] uses a sorting network that predefines all comparisons without unpredictable branches and permits multiple comparisons in the same cycle. These characteristics make it well suited for SIMD processors. Bitonic sort has also been implemented on GPUs [16, 13, 32] since it mapped well to the high compute density and bandwidth of GPUs. Chhugani et al. [7] show an efficient implementation of a merge sort algorithm by exploiting both TLP and DLP on recent multi-core processors. Our paper adopts the fastest CPU sorting implementation by Chhugani et al. [7] and extends it to sort tuples of (`key`, `rid`).

With regards to the choice of the join algorithm, Graefe et al. [14] compare sort-merge join and hash-join and recommend that both algorithms be included in a DBMS and be chosen by a query optimizer based on the input data. The hash join algorithm is a natural choice when the size of two relations differ markedly. They also show that data skew hurts the performance of hash join; they thus recommend sort-merge join in the presence of significant skew in the input data. Our paper revisits this comparison of both algorithms focusing on in-memory join operations. Our hash join implementation is not affected by data skew and is optimized for the modern multi-core CPUs.

3. JOIN FOR MODERN PROCESSORS

The performance of computer systems has improved steadily over the years, mainly from advances in semiconductor manufacturing and processor architecture. In this section, we will examine how the architectural improvements have impacted the performance of the join operation.

3.1 Main Memory Databases

With the increase in capacity of main memory, a large number of database tables reside completely in main memory. Typical databases consist of tables with numerous columns, with each column having different width (in bytes). User queries performing join on more than two such tables are decomposed into pairwise table join operations. Performing join on the original tables is not an efficient utilization of the memory bandwidth and computation, and therefore the tables are stored using 2 columns, `key` and `rid`, with `key` being the join key, and `rid` storing the address of the tuple [5].

For main-memory databases, the number of entries in a table is typically less than the range of 32-bit numbers (2^{32}), and hence `rid` can be represented using 32 bits. Although the `key` can be of any variable width, since the number of records is less than 2^{32}, the number of **distinct keys** cannot be more than 2^{32}, and should also be represented using 32 bits. Of course, representing a variable length `key` using 32 bits and without changing the information content is computationally hard; schemes like key-prefix [31] and 32-bit XOR and shift hash function [6] have been used to represent keys using 32 bits.

Therefore, we focus, analyze and provide results for tables consisting of **32-bit `key` and 32-bit** `rid`. We propose our join computation pipeline to include a prologue phase that converts the inputs to the aforementioned representation, and an epilogue that operates on the generated output, and removes the false positive results while gathering the actual keys.

3.2 Optimizing for Memory Subsystem

Join is a memory intensive operation and consequently is directly affected by the performance of memory subsystem. As compute performance improves at a much faster rate than memory subsystem performance, memory access latency continues to worsen. To address this performance gap, a number of architectural features have been devised to improve average memory access latency.

Cache: A cache is a module of small but fast SRAM cells that provides low latency accesses to recently used data. It is used to bridge the performance gap between the processor and the main memory. Not only caches can reduce memory access latency, they serve as memory bandwidth amplifiers by filtering out external memory requests. Bandwidth between processor cores and caches is orders of magnitude higher than external memory bandwidth. Thus, blocking data into caches is critical for data intensive operations such as join.

TLB: Virtual memory is developed to alleviate programmers from having to deal with memory management. It allows a program to use memory that is larger than the amount of physical memory in the system. However, every memory access must go through a virtual-to-physical address translation that often is in the critical path of memory access. To improve translation speed, a translation look aside buffer (TLB) is used to cache virtual-to-physical translation of most frequently accessed pages. Ideally, the number of TLB entries should match the number of pages touched by an application. With memory size in the Gigabyte range, the number of TLB entries would be in the thousands. However, to make translation fast, TLB is typically designed as either a fully associative or highly associative cache. A TLB size greater than a certain size (e.g., 64) is very complex and consumes a lot of power. Recent processors use multiple levels of TLBs with the lowest level caching the most frequent use pages.

Prefetch: Another mechanism that has been employed to reduce the memory access latency is prefetches. Modern processors often include hardware prefetchers that track memory access patterns and automatically prefetch data into caches [11]. However, prefetchers only work well when there is a regular access pattern to begin with. For the join operation, the memory access pattern is fairly random that reduces much of the benefit of the hardware prefetcher.

Processor-Memory Bandwidth: Besides memory access latency, memory bandwidth is another critical component in the memory subsystem. Over the years, improvements to increase bandwdith include faster data transfer rate and wider interfaces. Despite these improvements, memory bandwidth still grows at a much lower rate than transistor count [33]. Chip-multiprocessors exacerbate the bandwidth problem as compute grows faster than bandwidth.

3.3 Optimizing for TLP

The number of cores and thread contexts will continue to grow in future processors. To obtain performance from such architecture, applications should be threaded to exploit thread-level parallelism (TLP). For best performance, data accessed by threads must be partitioned to minimize concurrent updates to shared data structures.

3.4 Optimizing for DLP

Single-Instruction-Multiple-Data (SIMD) execution is an effective way to increase compute density by performing the same operation on multiple data simultaneously. A 128-bit wide SIMD (e.g. SSE) is common in processors today. Future processors will have 256-bit or wider SIMD supports [21, 34]. SIMD execution requires contiguous data in registers or in memory. If data accesses are not contiguous, gather and scatter overheads are incurred. Another issue with SIMD execution is the requirement of fixed width data structures. In today's databases, tuples are often compressed with light compression schemes such as prefix compression, resulting in variable length tuples. Use of SIMD instructions causes the size of the data to increase by a factor of 2x to 10x due to decompression.

4. HASH JOIN

As explained in Section 3.1, we focus on equi-join queries on two tables with each tuple consisting of two fields (`key`, `rid`), each being a 32-bit number.

```
Q: SELECT ...  FROM R, S WHERE R.key = S.key
```

In addition, the tuples completely reside in main memory. For the remainder of the paper, we use the following notation:
$\mathcal{N}_{\mathtt{R}}$: Number of tuples in outer relation (`R`).
$\mathcal{N}_{\mathtt{S}}$: Number of tuples in inner relation (`S`).
$\mathcal{T}$: Number of hardware threads (including SMT).
$\mathcal{K}$: SIMD width.
$\mathcal{C}$: Cache size (L2) in bytes.
$\mathcal{P}$: Size of 1st level TLB (in number of entries).

The basic idea behind a hash join implementation is to create a hash table with keys of the inner relation (`S`), and reorder its tuples. This partitioning phase is followed by the actual join phase, by iterating through the tuples in `R`, and for each key – searching for matching keys in the hash table, and appending the matching tuples from `S` to the output table. The expected O(1) search cost of hash tables makes this an attractive option for database implementations. However, with increasing number of entries in the tables, this approach suffers from following performance limiters on current CPU architectures:

Size of hash table: In order to avoid wasteful comparisons during join, it is imperative to avoid collisions during hash lookups. Theoretically, this requires a hash table with size around *two* times larger than the number of input elements, or the cardinality of the input keys (if known a priori). We also need a hash function belonging to the class of strongly 2-universal functions [30].

A fixed (small) number of TLB entries: With large table sizes, it is possible to access regions of memory whose page entries are not cached in the TLB – thereby incurring TLB misses, and a large increase in latency. Therefore, it is critical to perform memory accesses in an order that avoids significant TLB misses – even for large input sizes.

Duplicate `key`'s in `S` : Each duplicate key necessarily leads to a collision in the hash table. Both direct chaining and open addressing methods [25] lead to poor cache behavior leading to increased

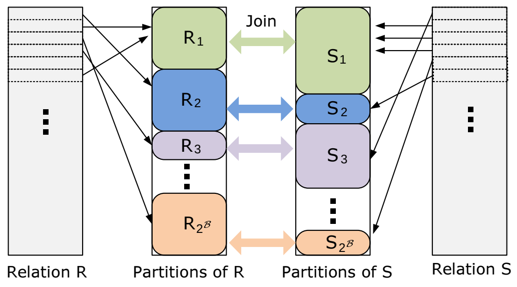

Figure 1: Partitioning relations R **and** S **to speedup the hash join operation**

memory latency. Array Hashes [1] lead to increased memory consumption and do not scale with multiple cores.

Having described the potential performance bottlenecks, we now describe our hash join implementation that addresses these issues, along with a corresponding analytical performance model. This is followed by the detailed algorithm description, and its extension to exploit the multiple cores and the 128-bit SIMD (SSE).

4.1 Algorithm Overview

To overcome the dependence on memory latency and memory bandwidth, we need to *partition* the input table[1] into smaller disjoint tables (referred to as sub-tables), such that each sub-table can reside in cache. This is followed by the actual join between the corresponding sub-tables in the two relations. We now describe the two phases in detail:

4.1.1 Partitioning Phase

Our partitioning phase is based on the radix-cluster algorithm proposed by Manegold et al. [28]. We partition the data based on the rightmost $\mathcal{B}$ bits of the two input tables to obtain $2^{\mathcal{B}}$ sub-tables, denoted by $\mathtt{R}_1$, .., $\mathtt{R}_{2^{\mathcal{B}}}$ and $\mathtt{S}_1$, .., $\mathtt{S}_{2^{\mathcal{B}}}$ (Figure 1). Note that we now need to perform $2^{\mathcal{B}}$ independent join operations – between $\mathtt{R}_i$ and $\mathtt{S}_i$ $\forall\, i$ 1..$2^{\mathcal{B}}$. The parameter $\mathcal{B}$ is chosen in a way that the average size of the resultant sub-tables in the inner relation ($\mathcal{N}_{\mathtt{S}}/2^{\mathcal{B}}$) fits in the L2 cache. Furthermore, in order to avoid TLB misses, we do not want to have more than $\mathcal{P}$ open pages at any point of time, where $\mathcal{P}$ is the size of the 1st level TLB. In practice, $2\mathcal{P}$ pages seem to work well when backed up by the next level TLB. However, having more than $2\mathcal{P}$ pages seems to expose the TLB miss latency, affecting the run-time performance on CPUs. Hence, the *maximum* number of sub-tables that can be generated at any one point of time is fixed to be $\mathcal{P}'$ (= $2\mathcal{P}$).

Since we wish to partition the table into $2^{\mathcal{B}}$ sub-tables, we perform a *hierarchical* partitioning, with each level subdividing a given table into $\mathcal{P}'$ disjoint sub-tables. We start with the input table, and subdivide it into $\mathcal{P}'$ sub-tables. Each of these sub-tables is further subdivided into $\mathcal{P}'$ sub-tables to obtain a total of $\mathcal{P}'^2$ sub-tables. This operation is carried out until the total number of sub-tables equals $2^{\mathcal{B}}$. A total of $\lceil \mathcal{B} / \log(\mathcal{P}') \rceil$ levels are required.[2]

For level(l) $\leftarrow$ 1 ... $\lceil \mathcal{B} / \log(\mathcal{P}') \rceil$:

Step P1: Iterate over the tuples of the table and build a histogram (`Hist`), with the $\mathtt{j}^{th}$ entry storing the number of input keys that hash to index `j`. Note that the hash function used simply considers log($\mathcal{P}'$) bits of the key, i.e., bit positions [l*log($\mathcal{P}'$) .. l*log($\mathcal{P}'$) + log($\mathcal{P}'/2$)] from the right to compute the hash index.

Step P2: Perform the prefix sum of the histogram (`Hist`) to compute the starting addresses of the elements mapping to the respective indices of the histogram. For example, after computing the prefix sum, `Hist[j]` stores the address where the first tuple from the table whose key maps to index `j` needs to be scattered to.

Step P3: Reorder the tuples of the table by iterating over them, and scattering a tuple to the address stored at the corresponding hash location. The address at the corresponding location is incremented by the size of the tuple to correctly reflect the scatter address for the next tuple that maps to that index.

We perform the above three steps for each level of subdivision for each of the sub-tables. Note that the final size of each sub-table depends on the distribution of the `key`'s in the relation. Furthermore, we read each tuple twice (once during Step P1 and later during step P3). This helps in computing the starting address for each sub-table within a pre-allocated memory chunk and avoids allocating separate buffers for each sub-table and maintaining them. We later show that the partitioning phase is compute bound on current CPUs, and not affected by the two trips to main memory.

We now describe the analytical model — 1) the amount of data that needs to be accessed from/to the main memory and 2) the number of operations that need to be executed — for the partitioning phase. We assume that the input table is too big to fit into the cache. Also, since the histogram fits in the cache, the reads/writes from/to the histogram are not dependent on the memory bandwidth. For each tuple, Step P1 reads 8 bytes and Step P3 reads 8 bytes and writes 8 bytes (scattered tuple). Note that the scattered write in Step P3 causes the cache subsystem to first bring in the cache line of the destination into the cache before the write is performed. So, this scattered write indirectly reads 8 bytes into the cache and then overwrite that location without using it. Hence, a total of 16 bytes are read, and 8 bytes are written in Step P3. In short, a total of **24 bytes are read** and **8 bytes are written** *per tuple* during the partitioning phase. Note that both the reads and writes are performed in a sequential fashion, hence the bandwidth is effectively utilized.

To compute the number of operations, let cost$_{hash}$ denote the cost of hash computation on the key as well as loading input data. During Step P1, we compute the hash value and also increment the histogram index (effectively loading, adding one, and storing back). Let cost$_{incr}$ denote the cost of incrementing the histogram index. Furthermore, the counter is incremented and compared to check for the end of the table. Let cost$_{epil}$ denote the cost of this epilogue operation. We denote cost$_{P1}$ as the number of operations executed *per tuple* during the execution of P1. Hence,

cost$_{P1}$ = cost$_{hash}$ + cost$_{incr}$ + cost$_{epil}$ (*per tuple*).

Step P2 operates on the histogram, and for each entry, reads it, modifies it and writes it back. The cost is the same as cost$_{incr}$. This step has the same epilogue operations to obtain a resultant of cost$_{P2}$ operations *per hash entry*.

cost$_{P2}$ = cost$_{incr}$ + cost$_{epil}$ (*per hash entry*).

Step P3 again computes the hash index, increments it, and scatters the input tuple to its permuted location. Let the cost of writing be cost$_{write}$ *per tuple*. Hence,

cost$_{P3}$ = cost$_{hash}$ + cost$_{incr}$ + cost$_{write}$ + cost$_{epil}$ (*per tuple*).

We denote the cost of partitioning for each tuple (for every level)

[1]The terms table and relation are used interchangeably throughout the paper.

[2]Unless otherwise stated, log refers to logarithm with base 2 ($\log_2$).

as $\text{cost}_{Partition}$. Note that we partition both R and S tables into the same number of sub-tables using the above algorithm before moving to the join phase, described next.

4.1.2 Join Phase

The join phase performs $2^{\mathcal{B}}$ independent joins – one for each partition generated in the partitioning phase. Let R_i and S_i denote the two relations that are being joined in one such partition, and $\mathcal{N}_{R_i}$ and $\mathcal{N}_{S_i}$ be the number of tuples in the two relations respectively. We again **build** a histogram using a hash fuction for S_i and reorder the tuples to obtain S_i'. The histogram together with S_i' comprises the hash table. The size of the histogram is chosen differently, and we will derive it at the end of the subsection. In the meanwhile, we refer to the size of the histogram as $\mathcal{N}_{\texttt{Hist}}$. We perform the 3 steps (P1, P2 and P3) described above, with a different hash function. Note that after reordering the tuples in S_i, all the tuples with keys that map to the same hash index are stored contiguously in S_i'.

The build phase is followed by the **probe** phase, where we iterate over the tuples of R_i, and hash each key, and go over the corresponding tuples in S_i' to find matching keys and output the result. Note that for each tuple in R_i, all the potential matches in S_i' are stored in consecutive locations. Hence to reduce the dependency on memory latency, we issue software prefetches by computing the starting address of the matching tuples in S_i' for keys at a certain distance from the current tuple being considered in R_i. We now describe the **build** and **probe** phases in detail.

Step J1: Similar to Step P1, iterate over the tuples of S_i and build a histogram, Hist. The hash function uses $\log(\mathcal{N}_{\texttt{Hist}})$ bits, i.e., bit positions $[(\mathcal{B}+1) .. (\mathcal{B}) + \log(\mathcal{N}_{\texttt{Hist}})]$ from the right to compute the hash index.

Step J2: Similar to Step P2, compute the prefix sum of Hist.

Step J3: Similar to Step P3, permute the tuples in S_i using Hist to obtain S_i'.

Having obtained S_i', we now perform the **probe** phase – Step J4.

Step J4: In order to issue prefetches, we implement the **probe** phase as follows. We keep a buffer (Buffer) of small number (say b) of elements. We iterate over tuples in R_i in batches of b tuples. For each tuple (say with key R_i[k].key), we store the tuple and its computed hash index (j_k) in the Buffer. Furthermore, we also issue a prefetch with the appropriate address of the first potentially matching tuple in S_i'. By construction, the offset of this element is Hist[j_k] within S_i'.

After filling up Buffer with b elements, we iterate over the tuples stored in Buffer, and now for each key Buffer[k′].key, and the corresponding hash value j′, compare all tuples in S_i' between indices Hist[j′] and Hist[j′+1]. Since we had already issued a pre-fetch for the first element at address Hist[j′], we expect the first few tuples to be in the L1 cache. Furthermore, as we sequentially search for the matching keys, the hardware prefetcher would fetch the subsequent cache lines into the L1 cache. This reduces the latency requirements of our **probe** phase. Note however, that we still incur branch misprediction (while comparing the keys in S_i'), and our performance should improve with the support of simultaneous multi-threading (SMT) on one-core.

We now derive the value of $\mathcal{N}_{\texttt{Hist}}$. Since we have already considered $\mathcal{B}$ bits during the partitioning phase, the maximum number of unique elements cannot exceed $2^{32-\mathcal{B}}$. In addition, as described in Section 4, we need a hash table of size around *two* times the number of elements (or cardinality) for reducing collisions. Hence, $\mathcal{N}_{\texttt{Hist}}$ is chosen to be $\min(2^{32-\mathcal{B}}, 2\mathcal{N}_{S_i})$.

Finally, we derive how to choose $\mathcal{B}$ for a given cache size $\mathcal{C}$. During the join phase, the original table (S_i), the permuted table (S_i') and the Histogram Hist need to be cache resident together. Hence for a table with $\mathcal{N}_{S_i}$ entries, $(8+8+4)\mathcal{N}_{S_i}$ bytes of cache are required. Thus $\mathcal{N}_{S_i}$ is around $\lfloor(\mathcal{C}/20)\rfloor$. Thus, we need to create around $\mathcal{N}_S/\lfloor(\mathcal{C}/20)\rfloor$ partitions. Therefore, $\mathcal{B}$ equals $\lceil\log(\mathcal{N}_S/\lfloor(\mathcal{C}/20)\rfloor)\rceil$.

We now derive a cost model for the amount of data that needs to be read from the main memory and the number of operations that need to be executed and for the Join phase. Although we set our partition parameters such that S_i, S_i' and Hist can reside together in the cache in the average case, it is indeed possible to obtain some partitions where this is not true. Therefore, to compute the memory requirements, we distinguish between the cases where above three entities fit together, and when they do not. For the former case, Step J1 reads 8 bytes and Step J4 reads 8 bytes. Hence a total of **16 bytes are read** *per tuple*.

If the three entities do not fit into the cache, 8 bytes are read during J1 and 8 bytes are read and 8 bytes are written during J3. Note that our partitioning scheme ensures that Hist always fits in the cache, and hence does not stress the memory bandwidth. The scattered write (during J3) also reads 8 bytes to fetch data for write, and hence a total of 24 bytes are read and 8 bytes of written. For Step J4, 8 bytes of R_i are read, and the probe phase now may need to bring in a complete cache line (64 bytes) of S_i' in the worst case to compare the first 8 bytes. Hence in the *worst case*, a total of **96 bytes are read and 8 bytes are written**. In the *best case*, each cache line that is read is completely utilized by different inputs, thus requiring only 8 bytes being read in the probe phase, for a total of **40 bytes read and 8 bytes written** during the whole Join phase. Of course, for each output tuple, 12 bytes are written (and correspondingly 12 bytes read).

As far as the number of operations are concerned, we borrow the expressions for steps J1, J2 and J3 from Section 4.1.1.

$\text{cost}_{J1} = \text{cost}_{hash} + \text{cost}_{incr} + \text{cost}_{epil}$ (*per tuple*).

$\text{cost}_{J2} = \text{cost}_{incr} + \text{cost}_{epil}$ (*per hash entry*).

$\text{cost}_{J3} = \text{cost}_{hash} + \text{cost}_{incr} + \text{cost}_{write} + \text{cost}_{epil}$ (*per tuple*).

Step J4 computes the hash index, and stores locally the tuple and the computed hash index, followed by issuing the prefetch instruction. As far as the probe phase is concerned, it reads the two consecutive addresses stored in the Histogram, and compares the tuples in that range in S_i'. We represent the cost of locally storing and issuing the prefetch as cost_{pref}. Furthermore, let h denote the average number of tuples used for comparison from S_i' per tuple in R_i and let cost_{comp} denote the cost of one comparison.

$\text{cost}_{J4} = \text{cost}_{hash} + \text{cost}_{pref} + h\text{cost}_{comp} + \text{cost}_{epil}$ (*per tuple in R_i*).

We denote the cost of join as cost_{Join}, which is the sum of the above *four* expressions.

4.2 Exploiting Thread-Level Parallelism

In order to exploit the multiple cores, and simultaneous multithreading within one core, we need to parallelize both the partitioning and the join phases.

4.2.1 Parallelized Partition Phase

During the first level of partitioning (for R or S), all the $\mathcal{T}$ threads need to simultaneously perform Steps P1, P2 and P3. In addition, there needs to be an explicit barrier at the end of each step.

After the first level of partitioning, there are enough partitions and each thread can operate on a single partition without any explicit communication with the remaining threads. This is especially true on current multi-core architectures, with $\mathcal{T}$ being small ($\leq$16). We now describe in detail the algorithm for parallelizing the first level of partitioning and the issues that impact scalability. We refer to parallelized Steps P1, P2 and P3 as Steps P1_p, P2_p and P3_p respectively. To reduce the impact of load imbalance, we use a scheme based on dynamic partitioning of the tuples. Specifically, we use the Task Queueing [29] model, and decompose the execution into parallel *tasks*, each executing a fraction of the total work (described below). This allows the runtime system to schedule the tasks on different hardware threads. For the discussion below, we assume $\mathcal{T}'$ ($\geq \mathcal{T}$) tasks, and later explain the relation between $\mathcal{T}'$ and $\mathcal{T}$.

Step P1_p: Equally divide the input tuples amongst the $\mathcal{T}'$ tasks. Each task $\mathcal{T}_i'$ maintains its local histogram ($\texttt{Hist}_i$) and updates it by iterating over its share of tuples. The hash function used is same as the one used in serial P1 step.

Step P2_p: Having computed their local histograms, the tasks compute the prefix sum in a parallel fashion. Consider the $\texttt{j}^{th}$ index. At the end of Step P1_p, each task stores the number of tuples whose keys map to $\texttt{j}^{th}$ index, and hence the total number of tuples mapping to $\texttt{j}^{th}$ index is $\sum\texttt{Hist}_i[\texttt{j}]$. For the $\texttt{i}^{th}$ task, the starting address of each index j can be computed by adding up the histogram values of all indices less than j (for all the tasks), and $\texttt{j}^{th}$ index for all tasks chronologically before the $\texttt{i}^{th}$ task. This is same as the prefix sum operation, and we use the algorithm by Hillis et al. [19] to parallelize it.

Step P3_p: Each task again iterates over its share of tuples and uses its local histogram to scatter the tuple to its final location (similar to P3).

In practice, we set $\mathcal{T}' = 4\mathcal{T}$. As a result, the dynamic load balancing using task queue's improved the scaling by 5% – 10% over a static partitioning of tasks. This may be attributed to the reduction in the latency of the writes, since different tasks are at different execution stages during the task and are not simultaneously writing to the main memory.

4.2.2 *Parallelized Join Phase*

The partitioning phase creates $2^{\mathcal{B}}$ partitions. Statically dividing the sub-tables amongst the $\mathcal{T}$ threads may lead to severe load imbalance amongst the threads since the partitions are not guaranteed to be equi-sized. In addition, for skewed distributions, it is possible to have some partitions that have a large percentage of the input elements, and hence only a few threads will be effectively utilized. We devised a *three* phase parallelization scheme that accounts for all kinds of data skewness and efficiently utilizes all the computing resources. As in the partitioning phase, we use the task queueing model. Note that we maintain $\mathcal{T}$ output tables, that are merged at the end of the complete join algorithm.

Phase-I: We create $\mathcal{T}'$ tasks, and evenly distribute the sub-tables amongst the tasks. If the size of both the inner and outer sub-table is less than a pre-defined threshold ($\texttt{Thresh}_1$), it performs the join operation (steps J1 .. J4) and appends the output to the relevant output table. Note that there is no contention for writing between threads. In case the size of any of the sub-tables is greater than $\texttt{Thresh}_1$, the task simply appends that sub-table's pair id to a list of pairs to be addressed in the next phase.

In addition, the use of task queues ensures that we can achieve a better load balancing between threads and address the variability between individual task execution times. There is an explicit barrier at the end of Phase-I, and now we describe Phase-II, where all the threads work simultaneously to join a pair of sub-tables. All the sub-tables that were not joined during Phase-I now go through through Phase-II.

Phase-II: Let $\texttt{R}_i$ and $\texttt{S}_i$ denote the two relations to be joined by multiple threads. As in the partitioning phase, the first three steps (J1, J2 and J3) are parallelized in a similar fashion to steps P1, P2 and P3 respectively. After the *build* phase, we now execute the *probe* phase (J4_p) as described below.

Step J4_p: Evenly divide the input tuples in $\texttt{R}_i$ amongst the $\mathcal{T}'$ tasks. Each task maintains a separate Buffer ($\texttt{Buffer}_i$) and operates in batches of b tuples. While searching for potential matches for any key in $\texttt{S}_i'$, it is possible to find a lot ($\geq \texttt{Thresh}_2$, a pre-defined threshold) of potential matches (for skewed distributions like Zipf [15]). To avoid load imbalance in such cases, the task does not perform the search and appends that key to the list of unprobed keys, and also stores the starting and ending probing address.

At the end of the above phase, we have a list of unprobed keys. We now consider each of the unprobed keys, and perform the search in a parallel fashion.

Phase-III: All the threads work simultaneously for each of the probes. We evenly divide the search range amongst the keys and each thread searches for matching keys and appends the relevant tuples to their respective output tables.

The above *three* phase parallelization scheme incurs low overhead (for large input size), and aims at efficiently utilizing the memory bandwidth, and computation cores. The thresholds used above, $\texttt{Thresh}_1$ and $\texttt{Thresh}_2$ are set to $\mathcal{T}\mathcal{C}_1$ and $\mathcal{T}^2\mathcal{C}_1$ respectively, where $\mathcal{C}_1$ is the number of tuples that fit in the L1 cache. These cutoffs are chosen to reduce the overhead of parallelization.

Prior attempts have been made to solve the load imbalance problem in parallelizing the join phase in the presence of data skew [20, 36]. However, these were in the context of parallelizing across clusters of computers and not chip multiprocessors. Consequently, network communication latency and synchronization overhead did not allow for such fine-grained task-level parallelization. The scheme described in this section creates fine-grained tasks (phases-II and III) that can divide up work evenly across threads. Any remaining imbalance due to arbitration and latency effects is handled through a task queue mechanism that performs work stealing to balance the load across threads. We shall see in Section 6 that our scheme results in scalable performance even for heavily skewed data.

4.3 Exploiting Data-Level Parallelism

There is inherently a lot of data-level parallelism in the hash join implementation. We describe below a data parallel algorithm assuming a $\mathcal{K}$ element wide SIMD, with $\mathcal{K}$ being equal to 4 for the current SSE architecture.

During the partitioning phase, we operate on each tuple by computing the hash index, followed by updating the histogram followed by the scatter operation. The corresponding steps P1_{dp}, and P3_{dp} are as follows:

Step P1_{dp}: Iterate over the input tuples by operating on $\mathcal{K}$ tuples simultaneously. We need to extract the $\mathcal{K}$ keys and compute the hash function. Since the same hash function is applied on all the keys, this maps to SIMD in a straightforward way. This would be followed by updating the $\mathcal{K}$ histogram bins simultaneously.

Step P2$_{dp}$: As described in Section 4.1.1, Step P2 involves a prefix sum over a histogram table. Data parallel prefix-sum algorithms have been proposed in the literature [4]. However, such algorithms do not seem to give much benefit at $\mathcal{K} = 4$.

Step P3$_{dp}$: Operating on $\mathcal{K}$ tuples simultaneously, and computing the hash index, followed by gathering the scatter address from the histogram bins, and scattering the $\mathcal{K}$ tuples at their respective permuted locations.

For the join phase, steps J1$_{dp}$ through J3$_{dp}$ can exploit data level parallelism in a similar fashion to steps P1$_{dp}$ through P3$_{dp}$ respectively. Step J4$_{dp}$ is modified as follows:

Step J4$_{dp}$: Operating on $\mathcal{K}$ tuples simultaneously, and performing the search by comparing one element for each of the tuples.

In order to achieve SIMD scaling, we need efficient hardware implementation of the following features:

- *Data Gather:* In Step J4$_{dp}$, we need to pack together the elements from distinct search locations in a SIMD register to perform comparisons with the respective keys.
- *Data Scatter:* In Step P3$_{dp}$, we need to write consecutive elements in the SIMD register to non-contiguous locations.
- *SIMD update collision:* In P1$_{dp}$ and J1$_{dp}$, the simultaneous update of the histogram bins needs to be handled correctly in SIMD. In case the $\mathcal{K}$ bins are all distinct, it reduces to a gather operation, followed by an increment, followed by a scatter back. However, if two or more bins are the same, the update for all the distinct bins needs to be done in the first pass, followed by the next set of distinct bins. Implementing it in software is prohibitively expensive, and we need hardware support for such atomic vector updates. The benefit of such hardware support can be more significant especially when there are few conflicts expected within SIMD lanes (e.g., Step J1$_{dp}$).

However, the current CPU SSE architecture lacks support for efficient implementation of all the above features. Hence, we do not see any appreciable speedup in the data-level parallel implementation of the hash join algorithm.

5. SORT-MERGE JOIN

Sort-merge join sorts rows in both input tables by the join key and then merges these tables. The most expensive part ($\geq$98%) of sort-merge join is the sorting of those two tables. Therefore, it is essential to use the most efficient sorting implementation to achieve the best sort-merge join performance.

5.1 Scalar Implementation

For a scalar sort implementation, we adopt an efficient implementation of merge sort by Chhugani et al. [7]. Merge sort essentially merges two lists of length L to produce a list of length $2L$. In the next step, it merges two lists of length $2L$ to produce one of length $4L$, and so on until there is a single sorted list. Chhugani et al. optimized their implementation by (1) replacing branches with conditional moves that do not suffer branch misprediction, (2) blocking for cache to make efficient use of memory bandwidth, and (3) using multi-way merging to merge cache-size blocks to produce a single sorted list. However, their implementation sorts only keys. For this work, we extend their implementation to sort tuples of (`key`, `rid`).

There are two ways to sort the tuples. One way is to treat (`key`, `rid`) as a single entity. For example, if both key and rid are both 32-bit values, they can be treated as a single 64-bit entity. The benefit of this approach is that no extra instructions are needed on 64-bit architecture. A comparison can be performed only on the 32-bit keys but the actual sort moves the entire 64-bit entity that includes both `key` and `rid`.

The snippet of x86 assembly instructions below depicts the innermost loop that merges two lists, A and B. It loads pointers to A and B (lines 1-2), assumes that B is less than A and loads B's content into register `rdx` (line 3), and speculatively advances both A and B pointers (lines 4-5). A comparison of A's and B's keys (line 6) sets a conditional flag. Conditional move instructions (cmov) use this conditional flag to fix `rdx` (line 7) and roll back B's pointer (line 8) if A is less than B. If A is greater than B, A's pointer is rolled back to the old value (line 9). Finally, the content of `rdx` (whether containing A or B) is stored into a destination (line 10). Although this code executes more instructions than a simple if-then-else block, it eliminates the branch based on the comparison of A and B, and improves the runtime performance.

Note that the number of instructions below is the same as the number of instructions for sorting 32-key only, except that the `quadword` keyword is used to actually move 8 bytes to and from memory.

```
1.  mov    rsi, rax               ; save old ptr_A
2.  mov    rdi, rbx               ; save old ptr_B
3.  mov    rdx, qword ptr [rbx] ; load B's key&rid
4.  add    rax, 8                 ; ptr_A+=2
5.  add    rbx, 8                 ; ptr_B+=2
6.  cmp    dword ptr [rsi], edx ; compare keys only
7.  cmovc  rdx, [rsi]             ; A<B, load A's key&rid
8.  cmovc  rbx, rdi               ; A<B, roll back ptr_B
9.  cmovnc rax, rsi               ; A>=B, roll back ptr_A
10. mov    qword ptr [rcx], rdx ; store both key&rid
```

The second way is to treat them separately, therefore incurring extra instructions to move the associated `rid` as well. This results in slowdown but is more general, as it does not assume that the `key` and `rid` are kept together.

When the size of a key or rid is greater than 32 bits, a (`key`, `rid`) tuple cannot co-locate in a single 64-bit scalar register. In this case, extra instructions are needed to sort them. The code above needs three new instructions to explicitly move rid with the key: a load (load B's rid), a conditional move (load A's rid), and a store (store rid).

5.2 Exploiting Data-Level Parallelism

We use a bitonic merge network [7] to exploit data-level parallelism. Figure 2 shows a 4x4 bitonic merge network that merges two sorted sequences of length 4 and produces a single sorted sequence of length 8. A 4x4 merge network has three levels, each of which comprises of comparisons of four pairs of elements on four lanes (e.g., four boxes at each level). Within a lane `i`, it assigns to L_i the smaller element and H_i the larger element. Between each level is a shuffle network that routes the L's and H's to the desired lanes for the next level.

Initially, sequences A and B are sorted in the same ascending order. Bitonic merge needs one sequence to be sorted in ascending order (A), and the other in descending order (B). In the figure, A_0, A_1, A_2, A_3 are four contiguous elements of A that are already loaded in a SIMD register. B is shown after loaded into a SIMD register and permuted into descending order (B_3, B_2, B_1, B_0), called B$'$. At level 1, a SIMD comparison on A and B$'$ assigns the smaller values of the pairs (A_0, B_3), (A_1, B_2), (A_2, B_1), (A_3, B_0) to one SIMD register containing L's ($L_0L_1L_2L_3$) and larger values to another SIMD register containing H's ($H_0H_1H_2H_3$). For level 2, these L's and H's need to be routed the desired lanes for another comparison. If the

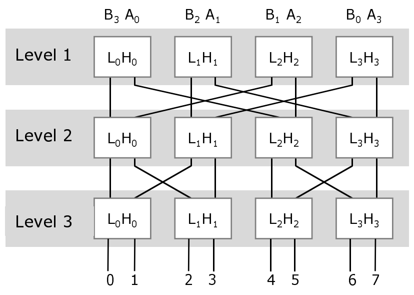

Figure 2: A bitonic merge network that merges two sequences of 4 elements each (A and B) to produce a single sorted sequence of 8 elements.

L's (or H's) within a SIMD register need to be routed in different directions, then shuffles are need. In practice, one shuffle is needed for each direction. As the bitonic merge network becomes larger, the top levels do not need shuffles because all L's or H's within a SIMD register move in the same direction. The same operation is repeated for level 3. At the end of level 3, two resulting SIMD registers containing $L_0L_1L_2L_3$ and $H_0H_1H_2H_3$ are interleaved (via a pair of shuffle instructions) to get a sorted sequence of L_0, H_0, L_1, H_1, L_2, H_2, L_3, and H_3.

Mapping this bitonic merge network to SSE4 [22] produces the sequence of instructions below. The instructions in black text (lines 1-2, 5-12) are for sorting keys only, as four keys can fit into a single SIMD register and can be processed by a SSE instruction concurrently. Ignore the instructions in blue text for now (lines 3-4, 13-20) as they are for (`key`, `rid`) tuples (describe below).

```
// xmm2 and xmm3 in descending order
// level 1
1.  xmm4 = sse_min(xmm0,xmm3);
2.  xmm5 = sse_max(xmm0,xmm3);
3.  xmm6 = sse_min(xmm1,xmm2);
4.  xmm7 = sse_max(xmm1,xmm2);
 // two shuffles for keys only
// no shuffle for (key, rid)
// 1st 2x2 network begins
// level 2
5.  xmm8 = sse_min(xmm4,xmm6);
6.  xmm9 = sse_max(xmm4,xmm6);
// shuffle
7.  xmm12 = sse_shuffle(xmm8, xmm9, dir1);
8.  xmm13 = sse_shuffle(xmm8, xmm9, dir2);
// level 3
9.  xmm16 = sse_min(xmm12,xmm13);
10.  xmm17 = sse_max(xmm12,xmm13);
// interleave result
11.  xmm0 = sse_shuffle(xmm16, xmm17, dir3);
12.  xmm1 = sse_shuffle(xmm16, xmm17, dir4);
// 2nd 2x2 network begins
// level 2
13.  xmm10 = sse_min(xmm5,xmm7);
14.  xmm11 = sse_max(xmm5,xmm7);
 // shuffle
15.  xmm14 = sse_shuffle(xmm10, xmm11, dir1);
16.  xmm15 = sse_shuffle(xmm10, xmm11, dir2);
 // level 3
17.  xmm18 = sse_min(xmm14,xmm15);
18.  xmm19 = sse_max(xmm14,xmm15);
 // interleave four results
19.  xmm2 = sse_shuffle(xmm18, xmm19, dir3);
20.  xmm3 = sse_shuffle(xmm18, xmm19, dir4);
```

Each level contains a SIMD min and a SIMD max instruction. We use generic names such as sse_min and sse_max to simplify discussion as SSE has a variety of min and max for different data types and sizes. Moreover, SSE uses different SIMD instructions to shuffle elements, depending on the shuffle patterns. Here we use a generic sse_shuffle(A,B,direction) to represent all these instructions, where `direction` tells the shuffle instructions how to route the elements of A and B.

Note that for sorting keys only, after min/max instructions at lines 3-4, a pair of shuffle instructions is needed because H_0, H_1 need to go to the lanes 3 and 4 while L_2, L_3 need to go to lanes 0 and 1, respectively. Due to the way the shuffle is implemented in SSE, the pair of shuffles at lines 7-8 actually require three SSE instructions to route 2 4-wide SIMD registers. This peculiarity disappears when routing 2-wide SIMD registers. Thus, the total number of SIMD instructions for sorting keys only is 13.

The same 4x4 network can be applied to tuples of a 32-bit key and a 32-bit rid, treating a (`key`, `rid`) tuple as a single 64-bit entity. Since SSE4 operates on two 64-bit values at a time, the number of comparisons at each level doubles (i.e., 2 SIMD min and 2 SIMD max instructions). In the code above, all lines (1-20) belong to the merge network. The number of shuffle instructions at level 2 and level 3 doubles. However, no shuffle is needed at level 1 because the entire SIMD registers (xmm4, xmm5, xmm6, xmm7) remain intact going to the next level. The number of instructions is 20, and increase of 1.54X over 32-bit keys only (13 instructions). The performance is expected to be **less than 2X slower** than keys only.

5.3 Exploiting Thread-Level Parallelism

Intel Core i7 provides two aspects of TLP: two hardware contexts on each core (SMT) and four cores on the same CPU package. Merge sort can take advantage of SMT by running two merging threads on the same core to hide instruction and memory latency. Without SMT, a wider network should be used to increase parallelism and overlap SIMD instruction latency with computation. Consider the merge network in Figure 2, a 4x4 network is composed of two independent 2x2 networks at level 2 and level 3. However, a 4x4 network has one extra level (level 1 in Figure 2) that includes extra min/max (two for keys only, four for key-rid) and two shuffles (keys only). Going to a 8x8 network gives two independent 4x4 networks but requires yet another level (four extra min/max). In short, as the network becomes wider, more levels and thus instructions are required. Note that when the network is wider than SIMD lanes, no shuffles are needed at the upper levels.

SMT obviates the need to go to wider networks by overlapping instruction and memory latency with instructions from the other thread. Using smaller network results in fewer instructions and as long as all pipeline stalls are overlapped with useful work, it will result in shorter execution time.

The second aspect of TLP is parallel merge. Tuples are first partitioned among $\mathcal{T}$ threads, which sort their own partitions. Then they cooperate in merging $\mathcal{T}$ sorted list into a single sorted list. When intermediate lists are larger than caches, merging may become bandwidth bound, as these lists streams from/to memory multiple times. We address the bandwidth issue in the next section.

5.4 Bandwidth-Oblivious Sort

Multiway merging [7] is used to address the bandwidth bottleneck by forming a tree of threads that incrementally merge the heads of partially sorted lists simultaneously. As tuples are merged, they are pushed to a "parent" thread up the tree. The parent thread

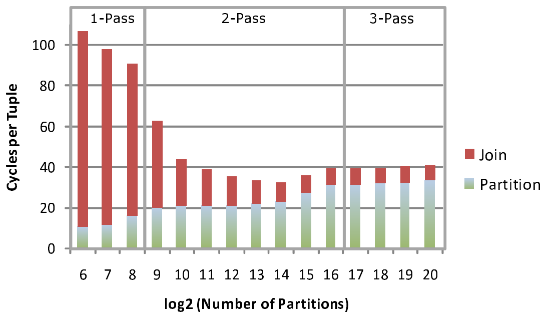

Figure 3: Time spent in partitioning and join phases with varying number of partitions for 128M tuples with uniformly distributed keys.

merges these tuples with tuples from other child threads and pushes the merged tuples up to its own parent. This goes on until tuples reach the root thread, which merges and stores them to memory. By merging all lists in parallel in this fashion and limiting the number of "in-flight" tuples within the last level cache, multiway merging reads each tuple once from main memory and writes it once to main memory. In short, multiway merging turns a bandwidth-bound merge sort into a compute-bound merge sort by juduciously managing its usage of memory bandwidth.

6. PERFORMANCE EVALUATION

In this section, we show the performance of both hash-based join implementation and sort-based join implementation. We run our experiments on a single socket Intel Core i7 965 system with 6GB of DDR3-1333 memory. The processor runs at 3.2GHz and has four out-of-order superscalar processor cores that support simultaneous multi-threading (SMT) with two hardware threads per core. Each core has a 32KB L1 instruction cache, a 32KB L1 data cache and a 256KB combined L2 instruction and data cache. The four cores share a 6MB L3 cache. For fast virtual-to-physical address translations, each core maintains a 64-entry fully-associative L1 TLB and a 512-entry four-way set associative L2 TLB.

6.1 Hash Join Performance

We evaluate the performance of the hash join in three aspects: 1) the benefit of partitioning; 2) the handling of input variations such as the table size, the join selectivity and the number of distinct keys; 3) the handling of heavily skewed data such as the Zipf distribution [15].

6.1.1 Partitioned Hash Join

To study the benefit of partitioning for hash join algorithm, we join two tables of 128 million tuples with uniformly distributed keys [9, 18, 28]. Figure 3 shows the time (in cycles) spent to process each tuple on the quad-core processor when the number of partitions varies from 64 to 1M[3].

To study the performance trade offs, we separate the time spent in the partitioning phase and the join phase. When the number of partitions is small (64, 128 and 256 partitions), the partition size is too big to fit in the caches. The join phase is memory bounded and it becomes the performance bottleneck. As the number of partitions increases, the partition size reduces and it eventually fits into the caches. When the number of partitions is greater than 4K, the time required for the join phase stabilizes. Further increase in the number of partitions would not improve performance any more because the time spent in the partitioning phase is dominating.

[3]1M refers to 1 million.

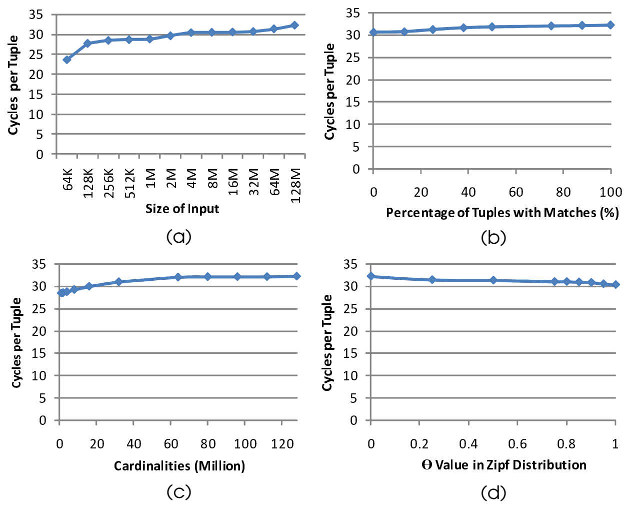

Figure 4: Computation time measured in cycles per tuple with various inputs: (a) varying the number of tuples from 64K to 128M (b) varying the join selectivity (c) varying the input cardinalities (d) varying the θ value of Zipf distributions.

Experimentally, we found the optimal number of partitions is 16K when each partition just fits in the on-die caches. Multi-pass partitioning is necessary when the number of partitions per pass is greater than 128, which is twice the size of L1 TLB.

6.1.2 Uniform Distribution

Next, we study how the hash join algorithm handles input variations. In this study, we do not include the time to generate the output, which is less than 3 cycles per output tuple. Even when the output size is similar to the input size, this adds less than 10% to the actual runtime.

Figure 4(a) shows the effect of changing the input data size from 64K to 128M tuples. The time per tuple varies from 25 cycles to 32 cycles as the input data size increases. However, the variation is small and very stable. This corresponds to 100M to 128M tuples per second. This result is better than any published performance in the literature. For example, He et al. [18] report a runtime of 2.5 seconds on a 2.4GHz Intel Core2 quad-core processor with tables of 16M tuples that have uniformly distributed 32-bit keys. In comparison, our Core i7 performance is around 0.15 seconds ($30cpe * 16M/3.2G$) which is **16.6X** faster. We also measured our performance on the same 2.4GHz Core2 quad-core processor used by He et al. and found our implementation to be **6.5X** faster. This illustrates the efficiency of our implementation. Furthermore, contrary to their claim that the hash join implementation on a Core2 quad-core CPU is 1.9X slower than a GPU (Nvidia 8800GTX), our Core2 implementation is in fact 3.4X faster than the same GPU platform.

Figure 4(b) shows the effect of the join selectivity by changing the percentage of matching tuples for the table size of 128M tuples. 0% means that there is no matching tuples, which results in no output data. We notice that the overall join time improves slightly as there are less tuples with matches[4]. This is because the branch prediction in the probing phase improves as the prediction accuracy increases by always predicting a non-match.

Figure 4(c) shows the effect of varying the number of distinct keys from 1M to 128M for the table size of 128M tuples. In general, data with low cardinality shows less branch mispredictions in

[4]the graph does not include the time for writing the output.

the probe phase, thus achieving better performance by around 5%-10% as compared to the higher cardinality data. Note that we do not exploit the cardinality information during execution of our algorithm. In case the cardinality is low and known a priori, we can reduce the number of partitioning phases or completely eliminate it, thereby further speeding up the runtime.

TLP Scaling: The performance results reported earlier in this section correspond to the parallel implementation of the hash join algorithm as described in Section 4.2. The partitioning step is parallelized by dividing the input tables evenly among the threads. In the join phase, each thread is responsible for joining one or more independent partitions.

For the uniform distribution, both the partitioning and the join phases scale very well with respect to the number of cores. There is no load imbalance among the different threads executing in parallel. Load imbalance usually arises in the join phase due to the variation of the partition sizes when input keys are skewed. For uniform distributed keys, partition sizes do not vary a lot and this is not an issue. Consequently, our parallelization of the join step only needs to go through Phase-I of Section 4.2. We see a scaling of **4.4X** over scalar code using four cores. The scaling is over 4X because SMT threads hide memory latency and improves the core efficiency.

6.1.3 Handling Skewed Data

While the uniform distribution offers the best case for parallel scalability, skewed input distribution such as the Zipf [15] distributions would test the ability of the parallel hash join implementation to handle load imbalance. Under this circumstance, our *3-phase* parallelization algorithm of Section 4.2 will be exercised and we will discuss the results in this section. The serial performance of our hash-join algorithm for skewed data is comparable to the serial performance for uniform data. This is in accordance with the results of our analysis of Section 4.1.

TLP Scaling: When all partitions are not uniformly sized, the amount of time to join each partition varies. In the extreme case when all the tuples fall into a single partition, we will see no parallel scalability with a naive implementation. The Zipf distribution is one such skewed distribution [15]. Consequently, we only see a 2.8X scaling on the 4-core processor when only the phase-I of our parallel join scheme is employed.

We address the load imbalance problem through our **3-phase** join parallelization algorithm. In Phase-II, threads cooperatively work on joining large partitions by dividing up the tuples among themselves. For the Zipf distribution with $\theta = 100$ (which is heavily skewed), 30-40% of the inputs are greater than 32K tuples, which was selected as the threshold. However, there may still be load imbalance in the **probe** phase when certain tuples have a large set of potential matches. To handle this, we separate out such probes in phase-III and cooperatively probe each such tuple using all threads. For the Zipf distribution with $\theta = 100$, less than 0.1% of the tuples went through this phase. Using these optimizations, we **improved our parallel scalability of the join phase from 2.8X to 3.9X**.

Figure 4 (d) shows the stability of our algorithm with skewed data. We control data skew by changing θ value from 0 to 100. With our load-balancing optimization, the figure shows that our hash join implementation is **stable across different degrees of skewed data**.

6.1.4 Analytical Model

Table 1 shows the breakdown of the time spent in each step as described in Section 4. The numbers are reported for joining two 128M relations with uniformly distributed keys. Since the cache size (L2) is 256 KB, the number of partition bits ($\mathcal{B}$) should equal $\lceil \log(128M/12.8K) \rceil$ (= 14). This is validated by Figure 3, where the join time is minimized with 14 bits of partitioning. We now compare the runtimes with our derived analytical model for the current platform. The cost symbols below are defined in Sections 4.1.1 and 4.1.2. The primitive costs were determined by counting instructions in the binary. All cycle and bandwidth numbers, unless stated otherwise, are given per tuple of data operated on.

Time	Partitioning			Join		
	P1	P3	Total	Build	Probe	Total
SMT OFF	1.2	4.4	5.6	4.7	5.7	10.4
SMT ON	1.2	6.0	7.2	4.7	4.7	9.4

Table 1: Computation time (in cycles per tuple in the inner relation) for each step in a hash join implementation and the effect of SMT support. For the partitioning phase, we report the time taken for every pass.

Step P1 (Section 4.1.1) reads 8 bytes of data. The peak bandwidth for our platform is around 7.2 bytes per cycle[5], and hence step P1 is bandwidth bound and should take around 1.1 cycles, which is *close* to the actual performance (1.2 cycles). Since the performance is limited by memory bandwidth, SMT does not improve the runtime any further. Step P2 has negligible runtime (less than 0.01) and is not reported. On our current system, $\text{cost}_{hash} = 4$ ops[6] , $\text{cost}_{incr} = 3$ ops, $\text{cost}_{write} = 5$ ops and $\text{cost}_{epil} = 3$ ops. Hence step P3 should take around 15 ops of computation on a single core. The total memory bandwidth requirement is around 24 bytes. Assuming a throughput of 1 op per cycle, step P3 should be compute bound, and take around 3.75 cycles on our system (with linear scalability). This is within 20% of the actual measured time (4.4 cycles). Note that SMT threads degrades the performance since we incur TLB misses (2 threads sharing the same TLB) which incur the additional latency.

For the build phase during join (Section 4.1.2), the total cost = $\text{cost}_{J1} + \text{cost}_{J2} + \text{cost}_{J3} = 27$ ops. With a throughput of 1 op/cycle and linear scaling, this amounts to 6.75 cycles. Since our current system can issue multiple instructions in one cycle, it increases the throughput for steps J1 and J2 and we measure a runtime of 4.7 cycles per tuple. Note that this does not benefit the partitioning phase, since step P1 has an explicit barrier at the end of its execution, and it is bandwidth bound. However, during the building phase, there is no barrier after individual steps, and the entire phase is compute bound. As expected, SMT does not provide any further benefits.

For the probing phase during join (Section 4.1.2), $\text{cost}_{pref} = 7$ ops and $\text{cost}_{comp} = 10$ ops (accounting for average case of branch misprediction). Thus, the total time evaluates to 24 cycles on one core, and around 6 cycles on 4 cores, which is within 6% of the actual measured data. SMT further improves the performance since the stalls during branch misprediction can be overlapped with potentially other computation being performed by the other executing thread on the core. Hence the effective value of cost_{comp} should further reduce to around 3 ops (assuming complete overlap), to obtain the required speedup. Thus, SMT substantially benefits the join phase of the hash join algorithm. To summarize, our analytical

[5]measured using an in-house bandwidth calibrator.
[6]1 op implies 1 operation or 1 executed instruction.

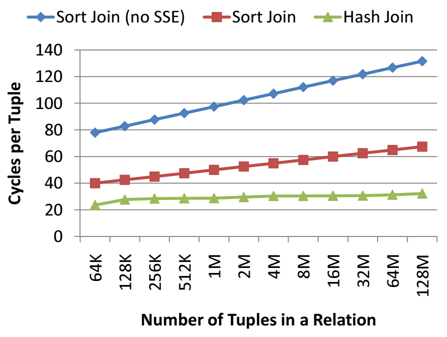

Figure 5: Comparison between sort-merge join and hash join performance with varying number of tuples in the inner and outer relations.

model predicts runtimes within 6% - 20% of the observed times for most of the cases. However, the model cannot evaluate the effect of multiple instruction issues, and hence is an upper bound for such phases (like building phase in the join operation).

6.2 Sort-merge Join Performance

For the sort-merge join implementation, most of execution time is spent in sorting two tables. For the scalar version, sorting 32-bit or 64-bit keys takes 11 clock cycles per element per iteration (cepi) on our system. The execution time is cepi*N*logN cycles. Sorting two 128M-key tables takes 79 billion cycles (24.9 seconds). Sorting two tables, each with 128M 64-bit tuples of (`key`, `rid`) takes 11.4 cycles per element per iteration (25.8 seconds), a negligible increase in cycles over sorting only keys. The small increase in clock cycles for sorting (key, rid) is likely due to moving more data (both key and rid) from memory. Another factor that impacts sorting performance is the size of keys and rids. When a (`key`, `rid`) tuple cannot fit in a single 64-bit scalar register, extra instructions are needed to sort them. On our test system, it takes 14.2 cycles per tuple per iteration to sort 128-bit (`key`, `rid`) tuples. That translates to 32 seconds for two 128M-tuple tables.

For the SIMD implementation, sorting keys only takes 3 cepi (6.8 seconds for two 128M-key tables) while sorting 64-bit (`key`, `rid`) tuples takes 4.5 cepi (10.2 seconds). These numbers matches the analytical model proposed by Chhugani et al. [7]. The slowdown of sorting (`key`, `rid`) tuples over keys only is 1.5X, which matches the increase in the number of instructions over keys only (1.53X). Parallel scaling of the SIMD implementation is nearly linear, 3.6X on four cores. The cepi for keys only is 0.83 (1.8 seconds for 2 tables of 128M keys) and for (`key`, `rid`) tuples is 1.25 (2.8 seconds for two tables of 128M tuples).

6.3 Comparison between Hash Join and Sort-merge Join

Figure 5 shows computation time of hash join and sort-merge join with varying number of tuples in both relations. For sort-merge join, we show both non-SSE and SSE implementation numbers. The SSE implementation of sort-merge join improves performance by 1.9X over non-SSE implementation. The theoretical maximum improvement with 128-bit SSE is 2X because each tuple consists of a 32-bit (`key`, `rid`) pair, and therefore we can accommodate two tuples in one 128-bit word. With 128 million tuples, our hash join implementation is 2X faster than even this optimized SSE sort-merge join implementation. Sort-merge join becomes faster with smaller tuples because the number of sort levels decreases proportional to $\log N$ (N: number of tuples). The gap between hash join and sort-merge join decreases to 1.6X with 64K elements.

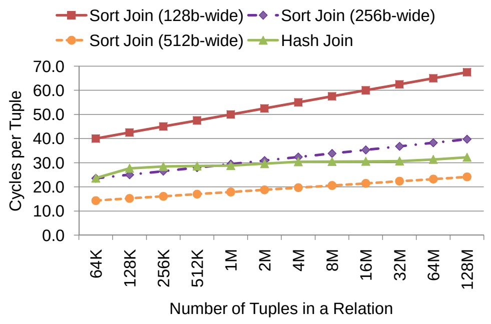

Figure 6: Comparison between sort-merge join and hash join. For sort-merge join, we add projecting performance with 256 bit-wide and 512 bit-wide SIMD.

7. FUTURE ARCHITECTURE TRENDS

In this section, we discuss future architectural trends from both the near term and longer term perspective, and how these trends affect the join algorithm choices.

Wider SIMD Execution: In Section 5.2, we show that sort-merge join can fully exploit DLP using SIMD execution. In sort-merge join, the efficiency of SIMD execution is affected only by the size of the (`key`, `rid`) tuple. For example, with a 32-bit (`key`, `rid`) pair, each tuple is already 64 bits and a 128-bit wide SIMD implementation (such as SSE) can only operate on two tuples simultaneously. In the near term, future processors will adopt wider SIMD execution (such as 256-bit for AVX [21] and 512-bit for Larrabee [34]). These wider SIMD support would strongly benefit sort-merge join.

Figure 6 shows the effect of wider SIMD execution on sort-merge join and hash join. We project the performance of sort-merge join with 256-bit and 512-bit SIMD based on the work by Chhugani et al. [7]. With 256-bit SIMD, sort-merge join starts performing better than hash join for small number tuples, and 512-bit SIMD execution of **sort-merge join is projected to be 1.35X – 1.65X faster than hash join**.

For hash join, the scatter update to the partitions (Step P1) or the hashed bucket (Step J1) is the primary limiter in exploiting DLP. In order to exploit DLP in this step, efficient hardware scatter support is necessary. An efficient scatter operation will write multiple elements to different memory locations in the most bandwidth efficient manner with the minimal latency.

More importantly, further performance benefit can be achieved with atomic vector support. In Steps P1 and J1, multiple tuples can potentially hash into the same partition. These steps require the targeted hash entry to be updated accordingly. When performing SIMD execution, multiple elements will update the same memory location. Current SIMD architectures cannot handle this collision case and would require reverting back to the scalar implementation. As a result, the SIMD execution of step P1 and J1 is slower than serial execution due to instruction overhead of conflict detection. Efficient support for atomic vector operations such as that proposed by Kumar et al. [24] would be beneficial.

Limited Per-Core Bandwidth: As described in Section 3.2, external memory bandwidth is becoming a scarce resource with the advent of many-core processors. Once the memory bandwidth requirement reaches the peak external bandwidth, integrating additional processor cores would not provide any performance benefit and would increase the power consumption. Therefore, any parallel algorithms need to reuse the data in the cache as many times as possible before data are written back to main memory.

As far as sort-merge join is concerned, we only need to access data from/to the main memory **two times** (Section 5). On the other hand, for a hash-join, we need to partition the data followed by the actual (cache-friendly) join phase. As we argue in Section 4, the restricted size of TLB forces multiple levels of partitioning for efficient runtime – at least two levels for large database sizes. In addition, the actual join requires one more main memory read/write for a total of **three** trips to the main memory. Therefore, as compared to sort-merge join, hash join would require *1.5X more bandwidth*. Therefore, for future scenarios with limited per-core bandwidth, the join runtime would be proportional to the number of memory external reads/writes of the data and **hash join is projected to be 1.5X slower than sort-merge join** for large datasets with high or unknown cardinality.

8. CONCLUSIONS

In this paper, we re-examined the two popular join algorithms – hash join and sort-merge join – and provided efficient implementations along with a detailed analysis and analytical model of the runtime performance. Our join implementations efficiently utilize the modern processor features by cache blocking to minimize access latency, vectorizing for SIMD to increase compute density, and balancing the load amongst cores, even for heavily skewed input datasets. Our hash-based implementation achieves more than 100M tuples per second on the latest quad-core processor which is 17X faster than the best reported numbers on quad-core processors and 8X faster than the best reported GPUs. Furthermore, our sort-merge join algorithm achieves more than 50M tuples per second – an order of magnitude faster than the best reported numbers.

We developed analytical models to project the performance of the two algorithms with future architectural trends towards *increasing SIMD width* and *limited per-core memory bandwidth*. The lack of appropriate hardware features to exploit SIMD limit the scalability of hash join algorithms, while sort-based join algorithms scale near-linearly with SIMD and are projected to be faster with a SIMD width of 512-bits or higher. In addition, the higher inherent memory bandwidth requirements of the hash join algorithm as compared to sort merge further point towards sort-merge join executing faster than hash join.

9. REFERENCES

[1] N. Askitis and J. Zobel. Cache-conscious collision resolution in string hash tables. In *in Proc. String Processing and Information Retrieval Symposium (SPIRE*, pages 92–104, 2005.

[2] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, pages 307–314, 1968.

[3] M. W. Blasgen and K. P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):362–377, 1977.

[4] G. E. Blelloch. *Synthesis of Parallel Algorithms*, chapter Prefix sums and their applications, pages 35–60. Morgan Kaufmann, 1993.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[6] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116–127, 2004.

[7] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *VLDB*, pages 1313–1324, 2008.

[8] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[9] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, pages 25–34, 2008.

[10] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB*, pages 151–164, 1985.

[11] J. Doweck. Inside Intel core microarchitecture and smart memory access. *White Paper*, Intel Corporation Jul 2006.

[12] B. Gedik, P. S. Yu, and R. Bordawekar. Executing stream joins on the cell processor. In *VLDB*, pages 363–374, 2007.

[13] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the ACM SIGMOD Conference*, pages 325–336, 2006.

[14] G. Graefe, A. Linville, and L. D. Shapiro. Sort versus hash revisited. *IEEE Trans. Knowl. Data Eng.*, 6(6):934–944, 1994.

[15] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.

[16] A. Greß and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, page 45, Apr. 2006.

[17] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

[18] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *SIGMOD Conference*, pages 511–524, 2008.

[19] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[20] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535, 1991.

[21] Intel Advanced Vector Extensions Programming Reference. 2008, http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf.

[22] Intel SSE4 programming reference. 2007, http://www.intel.com/design/processor/manuals/253667.pdf.

[23] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1), 1983.

[24] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. Hughes, C. Kim, V. Lee, and A. Nguyen. Atomic vector operations on chip multiprocessors. *In 35th International Symposium on Computer Architecture*, pages 441–452, June 2008.

[25] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, 1986.

[26] H. Lu, K.-L. Tan, and M.-C. Shan. Hash-based join algorithms for multiprocessor computers. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *VLDB*, pages 198–209, 1990.

[27] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? dissecting cpu and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[28] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[29] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2:185–197, 1991.

[30] R. Motwani and P. Raghvan. *Randomized Algorithms*. Cambridge University Press, 1995.

[31] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: a risc machine sort. *SIGMOD Rec.*, 23(2):233–242, 1994.

[32] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Graphics Hardware 2003*, pages 41–50, July 2003.

[33] M. Reilly. When multicore isn't enough: Trends and the future for multi-multicore systems. In *HPEC*, 2008.

[34] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *Proceedings of SIGGRAPH*, 27(3), 2008.

[35] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[36] J. L. Wolf, D. M. Dias, and P. S. Yu. A parallel sort merge join algorithm for managing data skew. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):70–86, 1993.

[37] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A parallel hash join algorithm for managing data skew. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1355–1371, 1993.

[38] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In *VLDB*, pages 186–197, 1990.

[39] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD Conference*, pages 145–156, 2002.