Efficient Main-Memory Hash Joins on Multi-Core CPUs: Does Hardware Still Matter?

C. Balkesen, J. Teubner, G. Alonso
ETH Zurich

M. T. Özsu
University of Waterloo

ABSTRACT

The architectural changes introduced with multi-core CPUs have triggered a redesign of main-memory join algorithms. In the last few years, two diverging views have appeared. One approach advocates careful tailoring of the algorithm to the parameters of the architecture (cache sizes, TLB, and memory bandwidth). The other approach argues that modern hardware is good enough at hiding cache and TLB miss latencies and, consequently, the careful tailoring can be omitted without sacrificing performance.

In this paper we resolve this conflict through an extensive experimental analysis of the design space using different algorithms and architectures. We have taken the algorithms presented in previous work and reimplemented them so that their results are comparable. We have run these optimized algorithms on the same datasets and compared their performance results. Our results are conclusive: hardware still matters. Join algorithms that take into consideration the architectural constraints perform far better than counterparts which rely on the hardware to hide the overhead of cache misses. In the paper, we explore a broader set of cardinalities and data distributions than in the original papers and show that these parameters also play a significant role in the overall performance.

1. INTRODUCTION

Modern processors provide parallelism at various levels: instruction parallelism via super scalar execution; data-level parallelism by extended support for single instruction over multiple data (i.e. SIMD, 128-bits; AVX, 256-bits); and thread-level parallelism through multiple cores and simultaneous multi-threading (SMT). Such changes are triggering a profound redesign of the algorithms used for performing joins in main memory. From these efforts, two opposing views have emerged regarding how to deal with the changing hardware landscape.

On the one hand, it has been argued that efficient, main memory parallel joins should be *hardware-conscious*: the best performance can only be achieved by fine tuning the algorithm to the specifics of the underlying architecture. In itself not a new claim [15, 12, 7], the argument is that in the context of multi-core architectures tuning is even more necessary [10]. Based on this premise, Kim et al. [10] have presented an algorithm that can process 100 million tuples per second.

On the other hand, some recent results have shown that join algorithms can be made efficient while remaining *hardware-oblivious* [3]. That is, there is no need for tuning—particularly of the partition phase of a join where data is carefully arranged to fit into the corresponding caches—because modern hardware hides the performance loss inherent to the multi-layer memory hierarchy. In addition, so the argument goes, fine tuning of the algorithms to specific hardware makes them less portable and less robust to, e.g., data skew. On this premise, Blanas et al. [3] have proposed a simple *no partitioning* join algorithm with performance comparable or even superior to that of the carefully tuned algorithm of Kim et al. [10].

Such an obvious contradiction needs to be resolved because it has far reaching consequences for data operator implementations over multi-core hardware. In this paper, we, therefore, address the question of whether hardware consciousness still matters when implementing parallel, main-memory hash joins on multi-core machines. Resolving the conflict between these results is a complex task because the experimental settings used in the original papers are not compatible: the algorithms use different implementations; the underlying hardware varies; the data used in the experiments differs in structure, size, and organization; and the relative and absolute sizes of the tables being joined also vary substantially. These differences have also been pointed out by Blanas et al. [3], who have published an addendum to their work arguing the validity of the arguments in favor of hardware-oblivious algorithms in spite of these differences [5].

The methodological approach followed in this paper is as follows. First, we analyze the hardware-conscious and hardware-oblivious algorithms proposed in the literature and describe several important optimizations leading to new algorithms that implement the same ideas but are more efficient (in some cases, significantly more efficient — an important aspect since they are later used as baselines). Second, we analyze each class of algorithms and put the existing results in context, validating the new algorithms to ensure they remain faithful to the original idea, and providing insights on what are the relevant design parameters. Finally,

using the optimized versions of the algorithms, we compare the performance of the hardware-conscious and hardware-oblivious approaches under a wide range of settings (hardware, data sizes, table sizes, skew).

The results demonstrate that hardware still matters and carefully tuning of the algorithm parameters to the underlying hardware characteristics leads to far better performance than relying on the hardware to hide cache misses. The results also explain the source of the conflicting conclusions in the current literature, which are more due to differences in data layout, format, table sizes, and degrees of optimization than to intrinsic properties of the algorithms. In doing this, the paper resolves an important question affecting the implementation of high performance parallel data operators and, through the optimized algorithms[1], provides key insights on the relevant design parameters determining the performance of main memory joins over multi-core hardware.

2. BACKGROUND

Hardware-conscious, main-memory hash join implementations build upon the findings of Shatdal et al. [15] and Manegold et al. [12, 7]. The focus on this work was to tune main-memory access by using caches more efficiently, which had been shown to impact query performance [2]. The key idea in this early work is adding a partitioning phase to the canonical hash join algorithm, which would normally consist of hash table build and probe phases. Shatdal et al. [15] identify that when the hash table is larger than the cache size, every access to the hash table results in a cache miss. Consequently, they partition the hash table into cache-sized chunks to reduce cache misses and improve the join performance.

Manegold et al. [12] refined this idea to consider also the effects of *translation look-aside buffers (TLBs)* during the partitioning phase. This lead to the idea of *multi-pass partitioning*, an approach that has become known in conjunction with *radix join*.

The advent of the modern multi-core CPUs renewed interest in parallel implementation of hash join algorithms particularly addressing this architecture and tuned for memory access. Under this constraint, Kim et al. [10] proposed a parallel version of the radix-join algorithm which is further optimized for the SIMD instructions and prefetching provided by modern processors. The algorithm applies the multi-pass partitioning idea of radix join but using multiple threads.

The hardware-oblivious camp argues differently. The argument is that the partitioning phase, which requires multiple passes over the data, can be omitted when the algorithm relies on modern processor features to hide cache latencies. In particular, simultaneous multi-threading (SMT) can hide cache miss latencies by running other threads while the first is waiting for memory.

Blanas et al. [3] follow this idea and introduce a very simple and parameter-free hash join algorithm without the partitioning phase. In the build phase of the multi-threaded implementation, every thread works on a region of the relation and populates a shared hash table where accesses are protected via latches. Similarly, in the probe phase, all the threads work on regions of the relation and probe the same hash table. By eliminating the costly partitioning phase, the hash join becomes simpler and parameter-free, yet achieves a performance comparable or better to that of cache-conscious alternatives.

The points of disagreement between the two camps that we address on in this paper can be summarized as follows:

Cache-Consciousness. Various studies have demonstrated the impact of caches on query performance [2, 7, 12]. Hardware-conscious algorithms, therefore, focus on optimizing their caching behavior. In particular, partitioning-based join algorithms focus on creating partitions that would fit into the caches to minimize cache misses and the resulting latencies. Hardware-oblivious proponents, on the other hand, argue that modern CPUs are very effective in hiding cache miss latencies via SMT. Consequently, they eliminate the partitioning step, and show that comparable or even better performance can be achieved.

Synchronization. Hardware-conscious main-memory join algorithms, based on partitioning, tend to be more complex and run in several phases where threads must synchronize between phases using barrier synchronization primitives. Proponents of hardware-oblivious algorithms claim that reducing (or eliminating) the cost of synchronization is more important than reducing (or eliminating) cache misses. Furthermore, they claim that algorithms that do not require synchronization are simpler, and potentially faster. While simplicity of algorithms is obvious, the efficiency argument is not entirely intuitive, since hardware-oblivious algorithms use a shared hash table that is accessed by all threads concurrently, requiring some synchronization with every access.

Data Distributions. Skewed data distributions are common and the effect of skew on join performance has long been recognized [8]. One study [3] argues that a hardware-oblivious hash join algorithm can handle skew gracefully whereas hardware-conscious algorithms suffer from the effects of data skew. As with cache misses, the argument is that modern hardware is intrinsically successful in handling skewed access patterns to memory without any hardware-specific tuning.

These are important points of disagreement, and we study them in this paper by implementing the algorithms highlighted in this section and running extensive experiments on them.

3. MAIN-MEMORY HASH JOINS

In this paper, we study six algorithms that are shown in Table 1. In the remainder of this section we describe each one of them in detail.

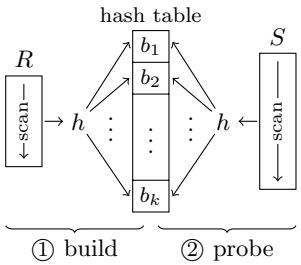
Algorithm	Description
NPB	No partitioning basic [3] (Section 3.3)
NPO	No partitioning optimized (Section 3.4)
RJ	Radix [12] (Section 3.5)
PR	Parallel radix [10] (Section 3.6)
PRB	Parallel radix basic [3] (Section 3.6)
PRO	Parallel radix optimized (Section 3.7)

Table 1: Algorithms under study.

[1]Upon publication, all algorithms and experimental data will be made publicly available.

3.1 Canonical Hash Join Algorithm

The canonical hash join algorithm [14, 11] operates in two phases as shown here on the left. In the first *build phase*, the smaller of the two input relations, say R, is scanned to populate a hash table with all R tuples. The *probe phase* then scans the second input relation, say S, and probes the hash table for each S tuple to find matching R tuples.



Both input relations are scanned once and, with an assumed constant-time cost for hash table accesses, the expected complexity for the canonical hash join algorithm is $O(|R| + |S|)$.

3.2 No Partitioning Join

The *no partitioning* join [3] is a direct parallel version of the canonical hash join. It does not depend on any hardware-specific parameters and, it does not partition the data.

Both input relations are divided into equi-sized portions that are assigned to a number of worker threads. As shown in Figure **??**, in the build phase, all worker threads populate a *shared* hash table that all worker threads can access.

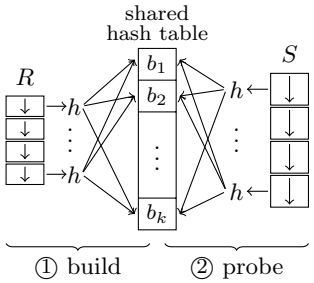


Figure 1: No Partitioning Join.

After synchronization via a *barrier*, all worker threads enter the probe phase and concurrently find matching join partners for their assigned S portions.

An important characteristic of *no partitioning* is that the hash table is *shared* among all participating threads. This means that concurrent insertions into the hash table must be *synchronized*. To this end, each bucket is protected via a *latch* that a thread must obtain before it can insert a tuple. The potential *latch contention* is expected to remain low, because the number of hash buckets is typically large (in the millions). The probe phase accesses the hash table in a read-only mode. Thus, no latches have to be acquired in that second phase.

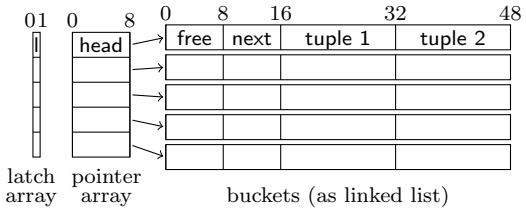
On a system with p cores, the expected complexity of this parallel version of hash join is $O\left(1/p(|R| + |S|)\right)$.

3.3 No Partitioning Join Basic

As we shall see also in the evaluation part of this work, the *no partitioning* idea leads to potentially different implementations. In this section we highlight the design and implementation choices that were made in the original implementation [3] (which we denote as **NPB**). This code is available on the authors' web site [4].

Input Representation. The representation of input relations in NPB is inspired by a page-oriented storage model. Tuples are placed in buffers of fixed length, and multiple buffers are chained as a linked list to host a full input relation. Accessing tuples in the buffers may, therefore, require following these links.

Hash Table. The hash table representation in NPB consists of three parts:



The hash table itself is an array of head pointers, each of which points to the head of a linked bucket chain.

Each bucket is implemented as a 48-byte record. A free pointer points to the next available tuple space inside the current bucket. A next pointer leads to the next overflow bucket, and each bucket can hold two 16-byte input tuples.

Since the hash table is shared among worker threads, latches are necessary for synchronization. As illustrated above, they are implemented as a *separate* latch array position-aligned with the head pointer array.

In this table, a new entry can be inserted in three steps (ignoring overflow situations due to hash collisions): (1) the latch must be obtained in the latch array; (2) the head pointer must be read from the hash table; (3) the head pointer must be dereferenced to find the hash bucket where the tuple can be inserted.

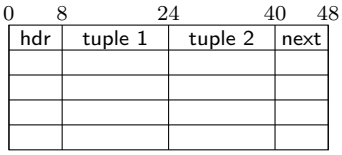
3.4 No Partitioning Join Optimized

In our study, we consider an alternative implementation of the no partitioning algorithm. We will hereafter call this implementation *no-partitioning-optimized* and denote it as **NPO**. Here we sketch the design choices that we made in NPO.

Input Representation. We organize input relations as contiguous in-memory arrays of tuples rather than as a linked list of pages. Our data is fixed-width, so any tuple can be accessed by position. When we assign sub-relations to threads, we use a $\langle \text{position}, \text{count} \rangle$ pair to characterize the sub-relation. This representation is consistent with modern main-memory execution engines, such as that used in MonetDB [6].

Hash Table. The hash table must be capable of holding a very large number of small entries. We consider this by representing the main hash table as a *contiguous array* of buckets of the same width. The hash function (we use the modulo operation for simplicity) directly indexes into this array representation. For overflow buckets, we allocate additional bucket space outside the main hash table.

The array of buckets is structured as shown on the right. An important design decision is to include the 1-byte synchronization latch as part of



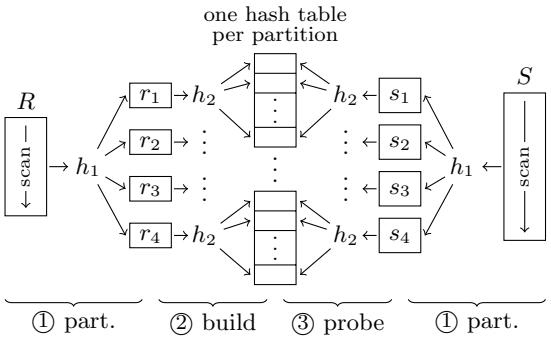


Figure 2: Partitioned Hash Join.

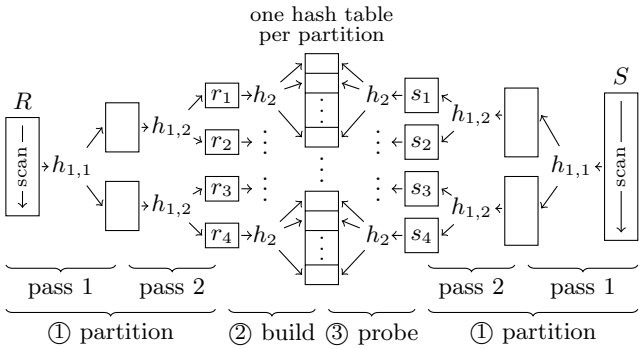


Figure 3: Radix Join.

the 8-byte header that also contains a counter indicating the number of tuples currently in the bucket. Each bucket can hold two 16-byte tuples, and an 8-byte next pointer chains hash buckets in the case of overflows. Each hash bucket is 48 bytes long, which easily fits into one 64-byte cache line. Having the latch as part of the header avoids a separate access to the latch table.

3.5 Radix Join

While the principle of hashing—direct positional access based on a key's hash value—is appealing, the resulting *random access* pattern to memory bears a high risk of expensive cache misses. In practice, as soon as the size of the hash table exceeds the system's cache capacity, virtually any hash lookup leads to a cache miss.

One of the first to identify this problem were Shatdal et al. [15] who proposed to *partition* both input relations first, then create hash tables only over partitions of the input tables. If partitions are sufficiently small, then each individual hash table will fit into the CPU cache and no cache misses will occur.

Partitioned Hash Join. The idea of Shatdal et al. is illustrated in Figure 2. In the first phase of the algorithm the two input relations R and S are divided into partitions r_i and s_j, respectively. During the build phase, a separate hash table is created for each r_i partition (assuming R is the smaller input relation). Each of these hash tables now fits into the CPU cache. During the final probe phase, s_j partitions are scanned and the respective hash table is probed for matching tuples.

During the partitioning phase, input tuples are divided up using *hash partitioning* (via hash function h_1 in Figure 2) on their key values (thus, $r_i \bowtie s_j = \varnothing$ for $i \neq j$) and another hash function h_2 is used to populate the hash tables.

While avoiding cache misses during the build and probe phases, partitioning the input data may cause a different type of cache problem. The partitions will typically reside on different memory pages with a separate entry for *virtual memory mapping* required for each partition. This mapping is cached by TLBs in modern processors. As Manegold et al. [12] point out, the partitioning phase may cause TLB misses if the number of created partitions is too large.

Essentially, the number of available TLB entries defines an upper bound on the number of partitions that can be created or accessed efficiently *at the same time*.

Radix Partitioning. Excessive TLB misses can be avoided by partitioning the input data in *multiple passes*. In each pass j, all partitions produced by the preceding pass $j-1$ are refined, such that the partitioning fan-out never exceeds the hardware limit given by the number of TLB entries. In practice, each pass looks at a different set of bits from the hash function h_1, which is why Manegold et al. [12] termed their idea *radix partitioning*. For typical in-memory data sizes, two or three passes are sufficient to create cache-sized partitions, yet not suffer from TLB capacity limitations.

Radix Join. The complete *radix join* is illustrated in Figure 3. ① Both inputs are partitioned using two-pass radix partitioning (two TLB entries would be sufficient to support this toy example). ② Hash tables are then built over each r_i partition of input table R. ③ Finally, all s_i partitions are scanned and the respective r_i partitions probed for join matches. Hereafter we will refer to our implementation of the algorithm described here as **RJ**.

In radix join, multiple passes have to be done over both input relations. Since the maximum "fanout" per pass is fixed by hardware parameters, $\log |R|$ passes are necessary, where R again is the smaller input relation. Thus, we expect a runtime complexity of $O\left((|R| + |S|) \log |R|\right)$ for radix join.

3.6 Parallel Radix Join

Radix join can be parallelized by subdividing both input relations into sub-relations that are assigned to individual threads. During the first pass, all threads create a shared set of partitions. These few tens of partitions are accessed by potentially many execution threads, creating a contention problem (the low-contention assumption of Section 3.2 no longer apply).

To overcome this contention, Kim et al. [10] reserve, for each thread, a dedicated range within each output partition to avoid contention. To this end, both input relations are scanned twice. The first scan computes a set of histograms over the input data, so the exact output size is known for each thread and each partition. Next, a contiguous memory space is allocated for the output, and, by computing a prefix-sum over the histogram, each thread pre-computes the exclusive location where it writes its output. Finally, all threads perform their partitioning without any need to synchronize.

After the first partitioning pass, there is typically enough independent work in the system (cf. Figure 3), so workers can perform work on their own. Task queuing is an effective mechanism to distribute the work among worker threads.

Hash Table Implementation. Manegold et al. [12] suggested to use *bucket chaining* to deal with hash collisions in the created hash tables. The main idea is to keep tuples with the same hash value chained together by their offset in the input relation array.

Kim et al. [10] build their hash table analogously to the parallel partitioning stage. The input relation R is first scanned to obtain a histogram over hash values. Then, a prefix sum is used to help re-order relation R (to obtain R'), such that tuples with equal hash value appear contiguously in R'. The prefix sum table and the re-ordered relation now together serve as a hash table as shown in Figure 4.

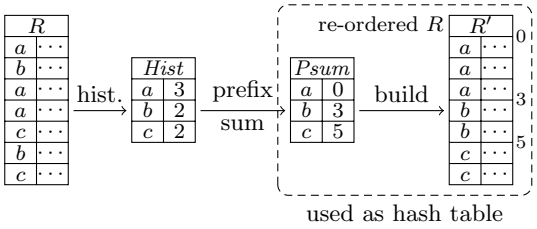


Figure 4: Histogram-based hash table build phase.

The advantage of this strategy is that contiguous tuples can now be compared using the SIMD instructions. In addition, software prefetching mechanisms can be applied to bring potential matches to the L1 cache before comparisons by storing the probe items in a small buffer.

We refer to the algorithm by Kim et al. [10] as **PR**. The implementation by Blanas et al. [3], which does not include the SIMD and prefetching optimizations, is denoted as **PRB**.

3.7 Parallel Radix Join Optimized

We have implemented several versions of the original *parallel radix join* [10] trying different optimizations. In one of the versions, we used *bucket chaining* for the build phase instead of the histogram-based relation re-ordering (denoted as **PRO**). As we shall see in Section 6.3, this implementation performs better than the version using SIMD and prefetching.

Apart from the differences in the build phase, there are also several minor differences of our implementations to that of Kim et al. [10]. First, during the join phase, Kim et al. [10] propose a 3-step fine granularity processing for handling very high skew. We have only implemented the task queue-based load balancing scheme inspired by [8] which is also the case for the implementation of Blanas et al. [3]. Second, the prefix-sum computation is not done in parallel as we have found out that this step takes insignificant amount of time.

When experimenting with our code, we found that the performance often degrades when partition sizes are a perfect power of 2. This is because partitions then contend for the same *cache set* within the set-associative cache of our system. Due to the low cache associativity, spurious cache misses occur without fully utilizing all cache lines. To avoid this problem, our code inserts padding between partitions when necessary.

4. EXPERIMENTAL SETUP

In the remainder of the paper, unless otherwise noted, all experiments are based on the following setup.

4.1 Data Sets

The two main data sets used in our evaluations are those used in earlier studies [3, 10]. The main differences among the data sets are the relative size of the tables and the data layout.

Non-Equal Data Set: This data set reflects the case where the join is performed between the primary key of the inner relation R and the foreign key of the outer relation S. The size of R is fixed at $16 \cdot 2^{20}$ and size of S is fixed at $256 \cdot 2^{20}$. The ratio of the inner relation to the outer relation is 1 : 16, which is claimed to mimic the relative sizes in decision support settings. In this data set, tuples are represented as $\langle \text{key}, \text{payload} \rangle$ pairs of 8 bytes each, summing up to 16 bytes per tuple. We populated the key column relation R with a random permutation of the values from 1 to the given relation cardinality.

Relation S consists of the same data as R repeated a number of times depending on the ratio between the two relations. This means that the foreign key relation contains 16 copies of the primary key relation concatenated one after another where each copy is permuted differently. This is the data set used by Blanas et al. [3].

Equal Data Set: In this data set, the inner relation R and outer relation S have $128 \cdot 10^6$ tuples each. The tuples are 8 bytes long, consisting of 4-byte (or 32-bit) integers and a 4-byte payload. Although this setup only allows relation cardinalities up to 2^{32}, it is argued that this range is typically larger than the number of entries in a common main-memory database [10]. As for the data distribution, if not explicitly specified, we use relations with randomly shuffled unique keys ranging from 1 to $128 \cdot 10^6$. This is the data set used by Kim et al. [10].

4.2 Hardware Platforms

The machines used in the experiments are summarized in Table 2. SMT functionality is only supported by Intel machines in this study and each core can support two thread contexts. In all the machines, L3 cache is shared and cache line size is 64 bytes. On the AMD Bulldozer machine, we used only 8 cores during the experiments to isolate from non-uniform memory access effects due to the special architecture of this CPU.

	Intel Nehalem	**Intel Sandy Bridge**	**AMD Bulldozer**
CPU	Xeon L5520 2.26 Ghz	Xeon E5-2680 2.7 Ghz	Opteron 6276 2.3 Ghz
Cores/Threads	4/8	8/16	16/16
Cache (L1/L2/L3)	32KB 256KB 8MB	32KB 256KB 20MB	48KB 1MB 16MB
Memory	12GB DDR3 1066 MHz	32GB DDR3 1600 MHz	32GB DDR3 1333 MHz

Table 2: Specifications of hardware platforms.

The experiments in Section 5 and 6 are conducted on an Intel Xeon L5520 2.26 Ghz with 2 sockets and 4 cores each with simultaneous multi-threading enabled. As in Blanas et al. [3], we use only one socket of the CPU where we first assign threads to 4 hardware cores and then assign to 4 SMT threads. All the hardware performance counter profiling experiments were conducted by integrating the Intel

Performance Counter Monitor [1] with the code, which enables very fine-grained and detailed instrumentation.

5. ANALYSIS OF NO PARTITIONING JOIN

In what follows, we experimentally evaluate and compare the original implementation of no-partitioning join (NPB) with our optimized version (NPO) using the non-equal data set. We developed NPO to improve on several performance issues that we observed in NPB. An answer to the hardware-oblivious vs. hardware-conscious controversy should not depend on particular implementation details (which might affect absolute performance, however). Thus, here we verify that NPO and NPB show similar trends and characteristics. When later we explore the main question of this paper, we will base our arguments on our optimized NPO implementation rather than on NPB.

5.1 Execution Time

Compared to NPB, NPO—together with a more suitable data representation for the tables—leads to significant performance gains. Figure 5 shows how the cycles-per-output-tuple (the ratio of total execution cycles to the total number of output tuples) vary with the number of parallel threads for both NPB and NPO. The cycles count is divided into the build and probe phases to show their relative contribution to the overall cost. In all cases, NPO is at least 3 times faster than NPB. This holds for both the probe and the build phases. In the build phase, NPO is at least 1.5 times faster. In the probe phase, NPO is at least 3 times faster.

Taken at face value, one could argue that the substantial performance improvements of NPO strengthen the case for hardware-oblivious algorithms. However, as we will show later, PRO offers similar performance improvements over PRB, so the fact that we can improve on NPB does not lead in itself to any clear conclusion.

5.2 Build Phase - No partitioning

In Blanas et al. [3], the build phase accounts for only 2% of the overall execution time of NPB. In our experiments, the ratio is much higher, from 6 to 12% for both NPO and NPB. Note that one would expect the lower bound of the cost of the build phase to be proportional to the relative table sizes—for the unequal data set about 6%. An overhead of 2% indicates that scanning the smaller table is somehow much faster than scanning the larger table. Furthermore, the build phase needs synchronization during accesses to the shared hash table, suggesting a cost higher than just 2%.

The code and scripts to generate the base data provided by Blanas et al. indicate that the results are based on experiments where the smaller table is pre-sorted. As a result, as data items are hashed, they map to consecutive hash table buckets and memory access is through sequential addresses. The sorted input also removes any contention for the bucket latch.

The results in Figure 5 are for unsorted data. To measure the effects of using a sorted input relation, we repeated the experiment with data sorted as in [3] (Table 3). These results confirm that the low contribution of the build phase in [3] is due to the use of a sorted input and not a feature of NPB or NPO.

5.3 Cache Efficiency

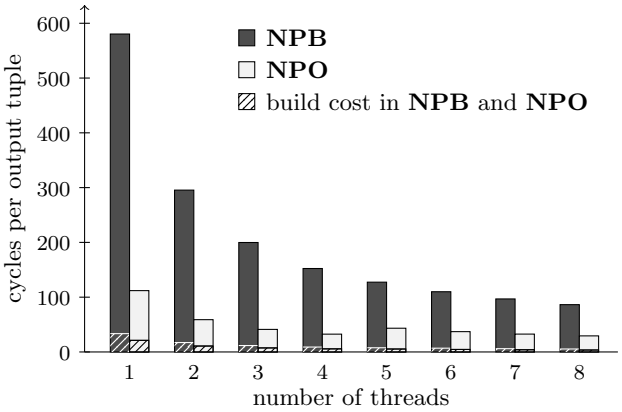


Figure 5: Cycles-per-output-tuple for NPB and NPO. Uniform non-equal data set.

	Cycles	L3 miss	Instr.	TLB miss
Build - sorted	322	2	2215	1
Build - unsorted	1415	45.3	2263	52.7

Table 3: Effect of sorted input on the build phase of NPB.

When using unsorted data as input, several performance limitations of the setup in [3] become visible: a high number of cache misses occur due to the implementation of latches as a separate data structure. Additional cache inefficiencies are caused by implementing the tables as linked lists instead of cache-aligned records as it is common in column store engines. However, none of these problems are essential to the basic ideas behind non-partitioning; they just need to be avoided as NPO does.

NPB uses a latch table separated from the hash table. During the build phase this leads to three potential cache misses per access to the hash table: one for acquiring the latch, one for reading the bucket pointer, and one for getting the actual bucket. Typical CPU caches today will not be able to hold the entire latch array ($1\,^{\text{byte}}/\text{bucket} * 8$ million buckets $= 8$ MB) and bucket pointer array ($8\,^{\text{bytes}}/\text{bucket} * 8$ million buckets $= 64$ MB). Likely, each of the three memory references will thus results in a cache miss.

Unfortunately, this problem will not even go away with larger caches. To acquire the latch during a parallel execution, a core will have to take ownership of the respective cache line and migrate it to its own local cache. The resulting cache line shipping can be expensive. Exclusive access to a remote cache line can incur a latency of up to 83 CPU cycles on current Intel Nehalem systems [13].

NPO needs 48 bytes for each bucket, and 4 consecutive buckets span three cache lines. As a result, a bucket spans 1.5 cache lines on average and we expect 1.5 cache misses per tuple during the build phase instead of three. This analysis is confirmed by the data obtained from the performance monitoring tool (Table 4).

Similar calculations can be done for the probe phase although the problem with NPB now is not the separate latches but the data layout. In NPB, the latch array is not used during the probe phase so the expected amount of cache

	NPB		NPO	
	Build	Probe	Build	Probe
L2 misses	3.06	3.25	1.56	1.39
L3 misses	2.88	3.19	1.56	1.36

Table 4: Cache misses per tuple.

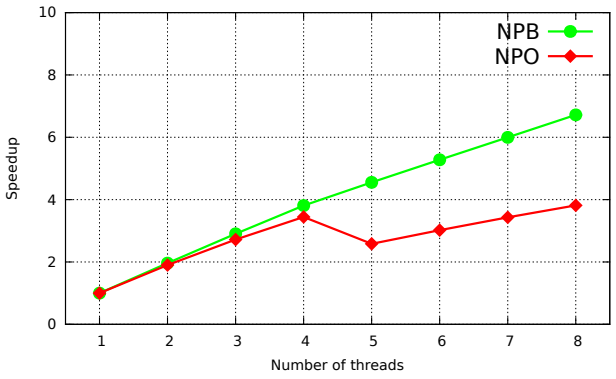


Figure 6: Speedup with SMT threads.

misses should be about two per tuple. However, the measured L2/L3 cache misses in Table 4 yield more than three misses per tuple. The additional misses are due to the buckets not being cache line aligned. Although each bucket is 48 bytes and can fit in a single cache line, the lack of cache alignment leads to almost all of the buckets spanning two cache lines, which explains the extra cache miss observed. This result is confirmed by experiments of NPB over cache-aligned buckets, which lead to 2.25/2.21 L2/L3 misses per tuple, much closer to the analysis.

5.4 The Role of SMT Threads

Given that NPO is faster than NPB, the question arises whether the NPO benefits from simultaneous multi-threading (SMT) as much as NPB does. SMT provides the illusion of an extra CPU by running two threads on the same CPU and cleverly switching between them at the hardware level when one of them stalls waiting for some operation to conclude. SMT works very well when the threads have to wait periodically. NPB has many more cache misses and, thus, waits more than NPO does. As the analysis above indicates, it is understandable why NPB works well with SMT.

In order to find out how SMT impacts NPO, we repeated the SMT experiment of Blanas et al. [3] on an Intel Xeon L5520 2.26 Ghz with 4 cores and 2 hardware thread contexts per core (the original study used an Intel Xeon X5650 2.67 Ghz with 6 cores and 2 hardware thread contexts per core). Similar to the original paper, we start by assigning each thread to a different core. Once the available physical cores are exceeded, we assign threads to the available hardware contexts in round-robin fashion.

The results are shown in Figure 6. Clearly, SMT does not have the same impact on NPO whereas the positive impact for NPB in our experiment confirm what was reported in the original study [3]. NPO is much faster than NPB so the fact that SMT does not work with NPO actually raises some questions about the case for hardware-oblivious algorithms. One of the arguments of Blanas et al. [3] is that no partitioning works because modern hardware can hide cache and TLB misses behind, among other things, SMT. Unfortunately, this holds only if there are enough cache misses and enough additional work for the second thread to make progress while the first one is waiting. The results for NPO indicate that an optimized implementation does not scale linearly with SMT threads, potentially removing one argument in favor of cache-oblivious algorithms on top of SMT multi-cores.

In summary, the behavior of NPO implementation is consistent with existing work (the work of Blanas et al. in particular). The absolute performance of NPO implementation is higher than that of NPB, making it a good base line for a meaningful evaluation of the hardware-conscious vs. hardware-oblivious controversy.

6. ANALYSIS OF RADIX JOIN

We perform a similar analysis for the parallel radix join. We study the implementation of Blanas et al. [3] (PRB) and our optimized version (PRO) using the non-equal data set. We also compare the behavior and performance of PRO with that of the algorithms presented in [10]. As with no-partitioning, PRO is faster than PRB and exhibits a performance comparable to that of the algorithm in [10]. Thus, it is a good baseline for the comparison with no-partitioning join.

Since hardware-conscious algorithms need optimal parameters (i.e., determining the number of optimal passes and partitions) for the machine on which the experiments are run, we pick the best configurations of each implementation. For PRB, the best configuration is 1 pass and 2048 partitions as in [3] since our experimental machine has the same L1/L2 cache sizes but different L3 cache size (12 MB vs. 8 MB). The optimal number of passes and partitions for PRO is 2 and 4096, respectively, which is same as in [10] since our experimental machine has the same L1/L2 cache sizes but different L3 cache size (6 MB vs. 8 MB). Finally, for the single threaded implementation RJ, the optimal number of passes is 2 and the number of partitions is 4096.

6.1 Execution Time

Figure 7 shows the execution cycles per output tuple for different numbers of threads for PRB and PRO. PRO is approximately 3 times faster than PRB in all cases. For reference and to control the actual behavior of the algorithms, we also compared PRB and PRO with the basic, single-threaded radix join (RJ) [12] and parallel radix join (PR) [10]. RJ is our own implementation along the lines of [12]; for PR we report in Figure 7 the numbers published in the original paper [10].

The single-thread performance of PRO is comparable to that of RJ which achieves 128 cycles per output tuple (not shown in Figure 7). Similarly, the 8-thread performance of PRO is slightly faster than the results reported in [10] as 30–32 cycles per output tuple (also not shown in the figure).

Again, the fact that PRO is faster is not an argument in favor of cache conscious algorithms. On the one hand, the results are very similar to those of [10] so PRO is not a significant variation over existing results. On the other hand, the performance improvements of PRO over PRB are comparable to the performance improvements of NPO over

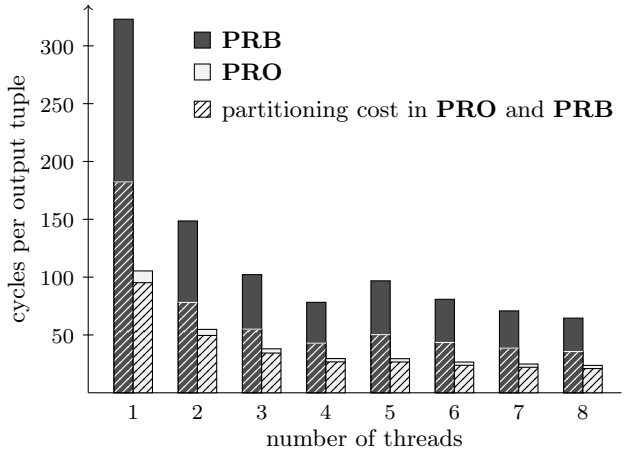


Figure 7: Cycles per output tuple for PRB and PRO. Uniform non-equal data set.

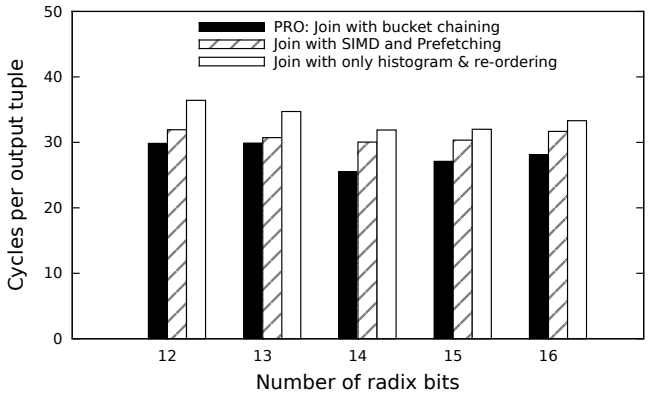


Figure 8: Comparison of join algorithms and effect of optimizations.

NPB. Thus, the enhancements give us just a more optimized baseline rather than advancing the case for hardware awareness.

6.2 Partitioning Phase - Parallel Radix

An interesting aspect of PRO's advantage over PRB is the contribution of each phase of the algorithm. Figure 7 shows the breakdown of the cycles per output tuple into two parts: partitioning time and actual join time (build and probe). In PRO, the partitioning time dominates the execution time (88% of the total time) due to two reasons. First, this implementation uses 2-pass partitioning resulting in several trips to the main memory. Second, as the partitions are fine-grained and can fit into L2 or L1 caches, the join phase is significantly faster and caches are fully utilized. In contrast, PRB spends between 52% and 56% of the overall execution time on the partitioning phase.

In spite of the difference in overall contribution, PRO is faster in both parts of the algorithm: PRO is $1.6-1.9$ times faster than PRB for partitioning, and almost an order of magnitude faster for the join. Table 5 shows the cycle and instruction count numbers measured in the experiments. The results for PRB conform to the results of Blanas et al. [3].

	PRB			PRO		
	Part.	Build	Probe	Part.	Build	Probe
Cycles	9398	499	7204	5614	171	542
Instructions	33520	2000	30811	17506	249	5650

Table 5: Cycle and instruction counts for PRB and PRO (in millions).

Based on this results, PRO (and PR) seems to capture the essence of hardware-conscious algorithms much better than PRB. In spite of adding a partitioning phase, PRB still spends about half the time doing the join. In contrast, PRO spends only about a tenth of the overall time doing the join, indicating that the extra partitioning phase clearly manages to speed up the join phase by a significant factor. One argument against this kind of algorithms is that they do not minimize processor synchronization costs as the partitioning phase requires much synchronization. Given the speed up achieved in the join phase of PRO through the partition phase, the high synchronization costs need to be seen in the context of the later gains and should not be considered in isolation. We will revisit the issue when comparing NPO and PRO.

6.3 Join Phase - Parallel Radix

Although PRO has a performance comparable to that of PR, it uses different mechanisms in the join phase. These mechanisms are an important part of the process of tuning the algorithm to the underlying hardware so it is worth looking in more detail at their impact on performance. For this purpose we have implemented different versions of the join phase of PRO: one using bucket chaining (the version studied above), one based on a histogram-based relation reordering algorithm using SIMD instructions and software prefetching mechanisms [10], and one based on a histogram-based relation re-ordering algorithm implemented without SIMD instructions and software prefetching. Results are summarized in Figure 8. These results conform to those reported by Kim et al. [10] where the optimal performance is achieved with 14 radix bits using a 2-pass partitioning (as in our case).

The interesting aspect of these experiments is that they show that SIMD and prefetching optimizations are not an important part of the potential advantage of parallel radix join. They improve performance but only by a small factor. It also turns out that the bucket chaining idea already proposed by Manegold et al. is still the best approach in multicore as it avoids repeated reading and writing of the relation for histogram computation and re-ordering, in addition to using less auxiliary memory. Accordingly, in the rest of the paper, we use the bucket chaining version of PRO.

In terms of the comparison with hardware-oblivious algorithms, these results are important because they show that the biggest part of the potential gain of hardware-conscious algorithms is indeed the careful tuning of the partitioning phase and not any fundamental reimplementation of the join phase.

6.4 Speedup from SMT Threads

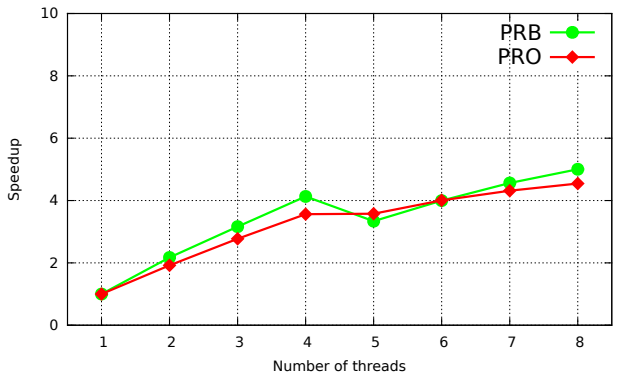


Figure 9: Speedup with SMT threads

Figure 9 shows that neither PRO and PRB benefit from SMT threads. Up to the number of physical cores, both implementations scale linearly, and in the SMT threads region both suffer from the sharing of hardware resources between threads. These results are also in line with the results of Blanas et al. [3]. As pointed out before, cache efficient algorithms cannot benefit from SMT threads to the same extent since there are not many cache misses to be hidden by the hardware. The results are also useful in validating the implementation of PRO against that of Kim et al. [10] as PRO achieves a speedup of 4.6, very close to the 4.4 factor reported by Kim et al.

6.5 Cache Misses

The issue of what is the cost of synchronization in parallel radix algorithms can be explored in more detail using low level instrumentation data. Table 6 shows the cache misses for PRO and PRB. Obviously PRO utilizes caches very well resulting in very low L2 and L3 cache misses. This explains the better performance of PRO in the build and probe phases.

Surprisingly, PRB has high L2 and L3 misses in the build and probe phases in spite of being a cache efficient algorithm. The explanation for this lies in the underlying synchronization used in PRB which, indeed, is a problem. In PRB, the probe phase only begins after all hash tables have been built by all threads introducing a very strong synchronization point. The problem is that some of the hash tables might be swapped out of the cache before probing begins which would explain the instrumentation numbers we measure for PRB.

In PRO, the hash table must is probed as soon as it is built so it is never swapped out of the cache. In PRO, each thread picks the build and probe relations together and does the probe immediately after building. In other words, it is possible to build a cache conscious algorithm with a very efficient partitioning phase where the synchronization costs are minimized. This removes an argument in favor of hardware-oblivious algorithms although it does not answer the question of which one does better.

PRO performs worse only for TLB misses in the partitioning phase due to having to read and write the data 2 times. The code path length also gives a good insight for the observed performance differences. The code path of PRB is longer by 1.9, 8 and 5.5 times, respectively in partitioning, build and probe phases.

	PRB			PRO		
	Part.	Build	Probe	Part.	Build	Probe
L2 misses	24	16	453	13	0.3	2
L3 misses	5	5	40	7	0.2	1
TLB misses	9	0.3	2	20	0.1	1
Bytes read	22845	266	4320	24856	211	4148
Bytes written	14166	0.7	7	10818	0.7	1.2

Table 6: Cache misses for different radix join implementations (in millions).

7. DOES HARDWARE MATTER?

In this section, we compare NPO and PRO over a wide range of settings such as different data sets, hardware, skew, and relative tables sizes.

7.1 Effect of Data Set

We have run PRO and NPO on different machines and using the two different data sets to evaluate the impact of the data set on the results (Figure 11).

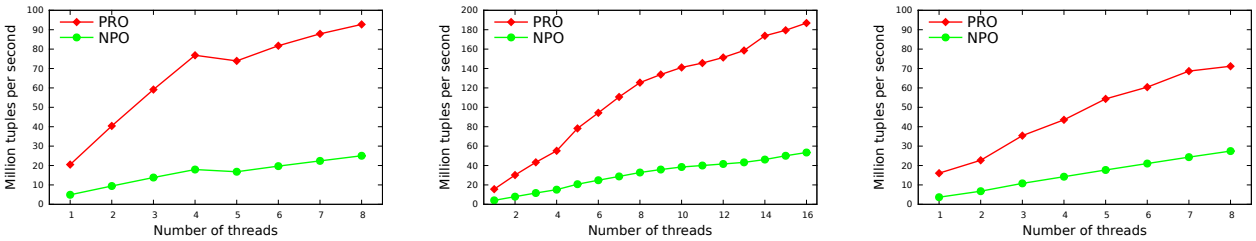
Figure 11(a) shows results for the uniform, non-equal data set. Since the data is not skewed, the results support the claim by Blanas et al. that hardware-conscious algorithms perform better only when highly optimized to the architecture and that hardware-oblivious algorithms can offer comparable performance. Although PRO is faster, one can argue that the simpler design and parameter-free nature of NPO makes it a better choice -which is the main argument in favor of hardware-oblivious approaches.

Unfortunately , when running the same algorithms on the same machines with the uniform, equal data set, the picture changes radically (Figure 11(b)). PRO is approximately 3.5 times faster than NPO on Intel machines and 2.5X faster on the AMD machine. That is, NPO has only comparable performance to PRO when the relative relation sizes are very different as this minimizes the cost of the build phase. As soon as the table sizes grow and become similar, the overhead of not being hardware-conscious becomes clearly visible (see the differences in the build phases for NPO).

7.2 Effect of Different Machines

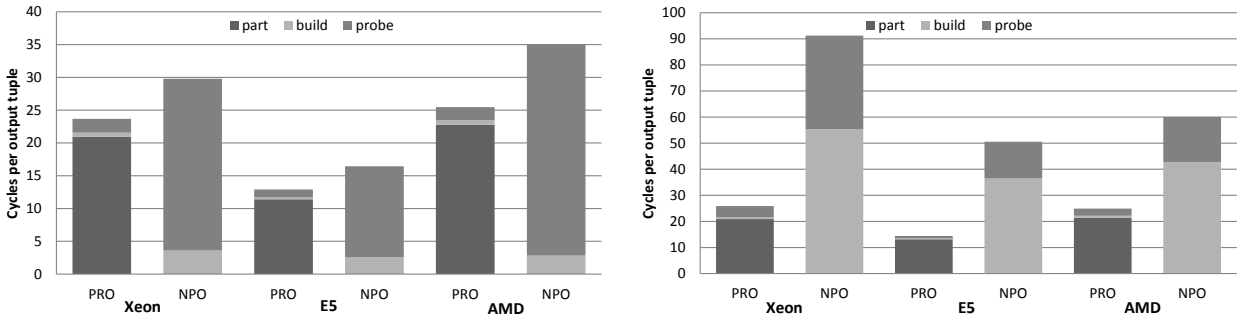
The actual underlying hardware is also an important factor in how NPO and PRO behave. On Intel Sandy Bridge E5, a very recent machine, both NPO and PRO show the best performance. On the AMD machine, performance is not as good as even the Intel Xeon L5520 machine using only 8 cores of the available 16 cores. This is due to the special topology and non-uniform memory access effects of the AMD processor, where using all the 16 cores did not show any significant benefit and we do not report it here. It is likely that such an architecture requires a tailored design for th algorithms to perform well, removing an argument in favor of NPO as, even if it is parameter free, some multicore architectures may require specialized designs anyway as NUMA would crate significant problems for the shared hash table used in NPO (let alone future designs where memory may not be coherent across the machine).

Figure 10 shows the total throughput of NPO and PRO on the three platforms using the uniform, equal data set. On Intel Xeon, PRO reaches up to 95M tuples per second, whereas NPO's performance only reaches 26M tuples per



(a) Intel Nehalem Xeon L5520 (b) Intel Sandy Bridge E5-2680 (c) AMD Opteron Interlagos

Figure 10: Throughput comparison of algorithms on different machines with the uniform equal data set.



(a) Non-equal data set (b) Equal data set

Figure 11: Cycles per output tuple comparison with different data sets and machines.

second as shown in Figure 10(a). On the Intel Sandy Bridge E5 machine, of PRO reaches 196M tuples per second showing that a hardware-conscious design can make better use of the underlying hardware: as far as we are aware, this is the highest throughput reached for a main memory hash join algorithm so far.

7.3 Effect of Skew

In this section, we study the effects of skew following the same methodology of Blanas et al. [3]. The equal data set is extended by introducing a skewed join key distribution in the foreign key relation while leaving the primary key relation as it is. In our experiments, we test both data sets and a skew ranging from 0 (no skew) to 1.75 (very high skew). The experiments are run on the Intel Xeon L5220 machine.

The results are shown in Figure 12 and Figure 13.

The claim that NPO is better than PRO in the presence of skew is confirmed in Figure 12(a). As the skew increases, NPO gets better and better, significantly outperforming PRO. Skews beyond 1.25 does not improve the performance further. The reason is that beyond such a skew, almost all of the matching tuples in the build hash table fit into the lowest-level cache. This is also confirmed by the results in Figure 13(a), where the build time stay constant and the probe time decreases dramatically with the skew. Although we have not implemented the 3-phase parallelized join scheme to handle skewness suggested by Kim et al. [10] (also the case in the paper of Blanas et al. [3]), the results seem to validate the claim in favor of hardware-oblivious algorithms.

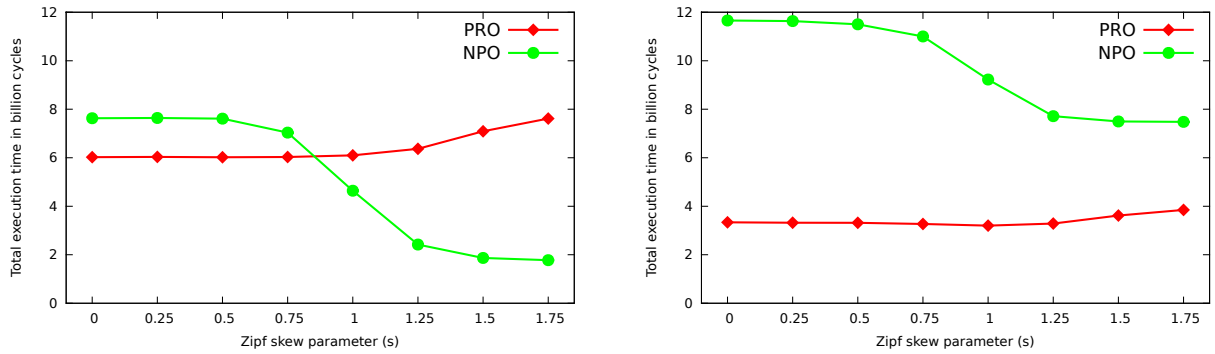
However, when using the equal data set (Figure 12(b)) the situation is reversed. NPO becomes slower as a result of the increasing cost of the build phase. As shown in Figure 13(b), the build time dominates the overall execution as it stays constant with increasing skew and only probe time decreases. Due to the same fact, the performance improvement in NPO with skew is not as high as in the non-equal data set, which is now around 1.6 times. PRO becomes faster with the equal data set. Although around the same number of tuples are partitioned ($128 \cdot 10^6 + 128 \cdot 10^6$ vs. $16 \cdot 2^{20} + 256 \cdot 2^{20}$), because of the decrease in tuple sizes, less memory transfers are done in the partitioning phase. Overall, performance of NPO still improves with skew but the implementation PRO is still almost 2 times faster than NPO with the equal data set.

The findings in Figure 12 show the importance of the data sets when discussing the relative performances of different join algorithms. They also show that, in spite of the careful tuning, parallel radix join is much less sensitive to skew in general. In actual implementations, *robustness* and *predictability* [9] tend to be much more critical than raw performance, which would favor hardware-conscious algorithms such as PRO.

7.4 Effect of Relation Size Ratio

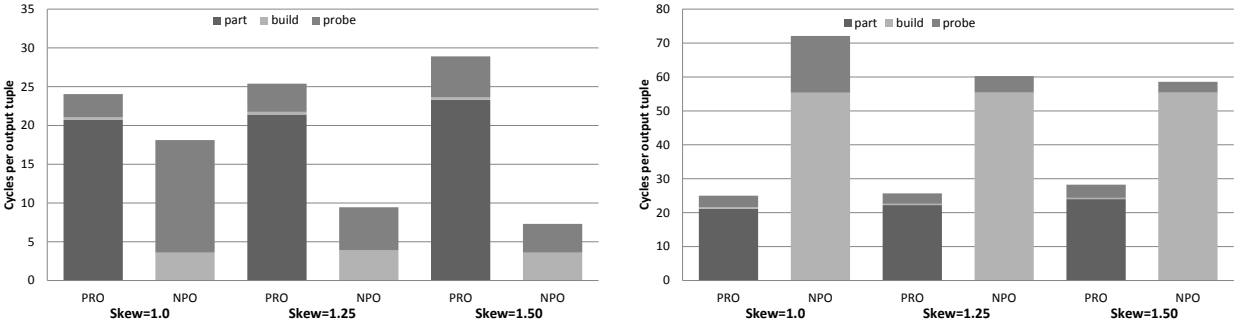
The experiments above show that actual relative sizes of the tables to join play a big role in the actual behavior of the algorithms. In the following set of experiments, we explore the effect of varying relation cardinalities on join performance. For these experiments, we use the Intel Xeon L5520 and fixed the number of threads at 8. We varied the size of the primary key build relation R in the non-equal data set from $1 \cdot 2^{20}$ to $256 \cdot 2^{20}$ tuples. The size of the foreign key relation S is fixed at $256 \cdot 2^{20}$. However, as we changed the size of R, we have also adjusted the distribution of values in S accordingly.

Figure 14 shows the cycles per output for each phase as well as the entire run for different R sizes in a log-log plot.



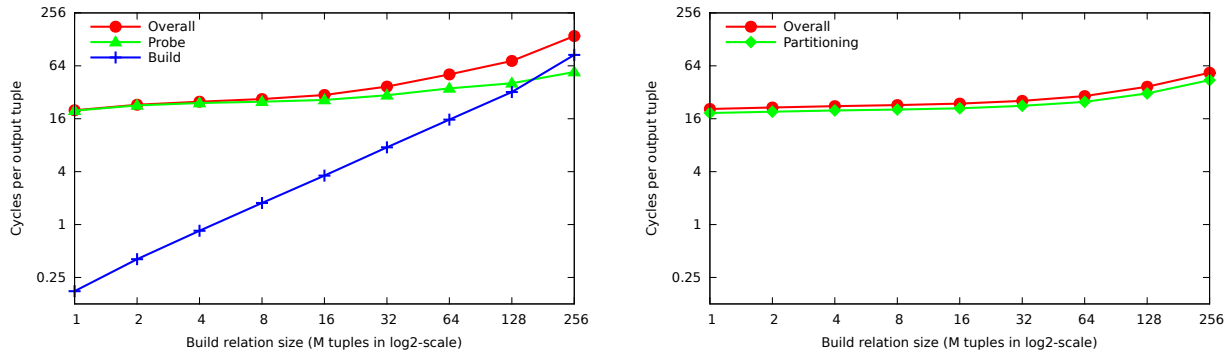
(a) Non-equal data set (b) Equal data set

Figure 12: Experiments with varying skew over different data sets.



(a) Non-equal data set (b) Equal data set

Figure 13: Cycles per output tuple comparison with different data sets and skew.



(a) NPO (b) PRO

Figure 14: Cycles per output tuple with varying build relation cardinalities in non-equal data set.

The results confirm the observation made so far and provide a more clear answer to the controversy between hardware-conscious and hardware-oblivious algorithms. NPO does very well when the small table is very small compared to the large table. Performance decreases linearly as the site of R increases solely because of the cost of the build phase (Figure 14(a)). PRO proves to be very robust to different table sizes and exhibits very much the same performance across all sizes of R. More importantly, the contribution of the partitioning phase is the same across the entire range, indicating that the partitioning phase does its job regardless of table sizes.

In other words, NPO is better that PRO only under skew and when the sizes of the tables being joined significantly differs. In all other cases, PRO is better (and significantly better in fact) in addition to also being more robust to different parameters like skew or relative table sizes.

8. CONCLUSION

In this paper we have addressed the controversy regarding the need for hardware-conscious algorithms to make the most of modern multicore architectures when implementing main memory joins. The results are conclusive, based on the algorithms proposed in the literature, hardware-oblivious algorithms only work well under skew and when the table

sizes significantly differ. In light of all the additional experimental results presented in the paper, the conclusion is that hardware-conscious algorithms perform better under a wide range of parameters and are more robust than their hardware-oblivious counterparts.

9. REFERENCES

[1] Intel Performance Counter Monitor. `http://software.intel.com/en-us/articles/intel-performance-counter-monitor/`. Online, accessed April 2012.

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

[3] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*, pages 37–48, 2011.

[4] S. Blanas and J. M. Patel. Source code of main-memory hash join algorithms for multi-core CPUs. `http://pages.cs.wisc.edu/~sblanas/files/multijoin.tar.bz2`. Online, accessed April 2012.

[5] S. Blanas and J. M. Patel. Addendum on radix join efficiency: How efficient is our radix join implementation? `http://pages.cs.wisc.edu/~sblanas/files/comparison.pdf`, August 2011. Online, accessed April 2012.

[6] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, pages 54–65, 1999.

[8] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, pages 27–40, 1992.

[9] G. Graefe. Robust query processing. In *ICDE*, page 1361, 2011.

[10] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

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

[12] 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.

[13] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proc. PACT*, pages 261–270, 2009.

[14] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, 3rd edition*. Springer, 2012.

[15] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*, pages 510–521, 1994.