Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware

Cagri Balkesen, Jens Teubner, Gustavo Alonso
Systems Group, ETH Zurich
Switzerland
{name.surname}@inf.ethz.ch

M. Tamer Özsu
University of Waterloo
Canada
tamer.ozsu@uwaterloo.ca

***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 architectural parameters (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 demonstrate through an extensive experimental analysis of different algorithms and architectures that hardware still matters. Join algorithms that take into consideration the architectural constraints perform far better than hardware-oblivious approaches. The analysis and comparisons in the paper show that many of the claims regarding the behavior of join algorithms that have appeared in literature are due to selection effects in choosing evaluation parameters (like the relative table sizes, the underlying architecture, or using sorted data) and are not supported by experiments run under different parameters settings. Through the analysis, we shed light on how modern hardware affects the implementation of data operators and provide the fastest implementation of radix join to date, reaching close to 200 million tuples per second.

I. 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 (SIMD; i.e., SSE, 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 main-memory join algorithms. However, the landscape that has emerged so far is confusing and highly contradictory.

One line of argument maintains that 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 [1]. These results also show that SIMD is still not good enough to tip the decision on join algorithm towards sort-merge join instead of the more commonly used hash join. In the future, however, as SIMD becomes wider, sort-merge join is likely to perform better.

Another line of argument suggests that join algorithms can be made efficient while remaining *hardware-oblivious* [2]. 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 in 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.

A third line of thought claims that sort-merge join is already better than hash join and can be efficiently implemented without using SIMD [3]. These results contradict the claims of Blanas et al. [2] because they are based on careful tuning to the hardware (in this case to its non-uniform memory access characteristics) and also the claims of Kim et al. [1] regarding the behavior of sort-merge vs. hashing when using SIMD.

For reasons of space and to scope the work to what can be presented in sufficient detail, in this paper we focus on the question of whether it is important to tune the main-memory hash join to the underlying hardware as claimed explicitly by [1] and implicitly by [3]. We also focus on radix join algorithms and leave the comparison with sort-merge joins for future work since such a comparison depends on different hardware features than the ones discussed here.

Answering the question of whether hardware still matters is a complex task because of the intricacies of modern hardware and the many possibilities available when implementing and tuning main-memory joins. To make matters worse, and as shown in the paper, there are many parameters that affect the behavior of the join operators: relative table sizes, use of SIMD, page sizes, TLB sizes, structure of the tables and organization, hardware architecture, tuning of the implementation, etc. Existing studies share very few points in common in terms of the space explored, making it difficult to compare their claims. As shown in the paper, many of these claims are specific to the choice of certain parameters and architectures and cannot be generalized across architectures or configurations.

The first contribution of the paper is algorithmic. We analyze the algorithms proposed in the literature and propose several important optimizations leading to new algorithms that are far more efficient and robust to parameter changes. In doing so, we provide important insights on the effects of multi-core hardware on algorithm design.

The second contribution is to put existing claims into context, showing what choice of parameters or hardware features cause the observed behaviors. These results shed light on what parameters play a role in multi-core systems, thereby establishing the basis for the choices a query optimizer for multi-core will need to make. The third and final contribution

is to settle the issue of whether tuning to the underlying hardware plays a role. The answer is a definitive yes, as it is only on a narrow combination of parameters and certain architectures where hardware-oblivious approaches have an acceptable performance.

To stimulate further empirical work in the field, we will make all code used to obtain our results publicly available.

II. Background: In-Memory Hash Joins

As a basis for our later evaluation, we briefly recap the state of the art regarding in-memory hash joins, including the motivation from the hardware side.

Existing algorithms can be classified in two camps. *Hardware-oblivious* hash join variants, represented here by the *no partitioning join* (Section II-B), do not depend on any hardware-specific parameters. Rather, they consider qualitative characteristics of modern hardware and are expected to achieve good performance on any technologically similar hardware. *Hardware-conscious* implementations, such as *(parallel) radix join* (Sections II-C and II-D), aim to maximally exploit a given piece of hardware by tuning algorithm parameters (*e.g.*, hash table sizes) to the particular hardware at hand.

The goal of our work is to assess whether hardware has now become good enough at hiding its own limitations—through pre-fetching or out-of-order execution—to make hardware-oblivious algorithms competitive, or whether explicit parameter tuning (usually by means of automated tools, such as *Calibrator* [4]) still yields enough performance advantages to warrant the effort.

A. Canonical Hash Join Algorithm

The basis behind any modern hash join implementation is the canonical hash join algorithm [5], [6], which operates in two phases as shown here in Figure 1. 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.

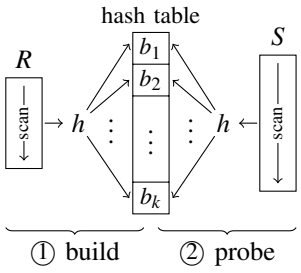


Fig. 1. Hash join.

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 $O(|R|+|S|)$.

B. No Partitioning Join

To benefit from modern parallel hardware, Blanas et al. [2] proposed a variant of the canonical algorithm that they termed *no partitioning join*, essentially a direct parallel version of the canonical hash join. It does not depend on any hardware-specific parameters and—unlike alternatives that we will discuss shortly—does not physically partition data. The argument is that the partitioning phase requires multiple passes over the data and can be omitted by relying on modern processor features such as simultaneous multi-threading (SMT) to hide cache latencies.

Both input relations are divided into equi-sized portions that are assigned to a number of worker threads. As shown in Figure 2, in the build phase, all worker threads populate a *shared* hash table that all worker threads can access.

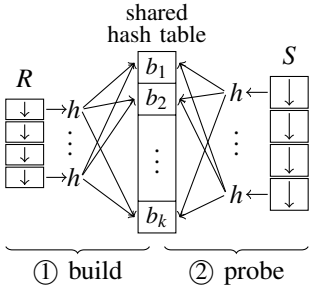


Fig. 2. *No partitioning* join.

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 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 $O(1/p(|R|+|S|))$.

C. Radix Join

Hardware-conscious, main-memory hash join implementations build upon the findings of Shatdal et al. [7] and Manegold et al. [4], [8]. 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. Thus, the main focus is on tuning main-memory access by using caches more efficiently, which has been shown to impact query performance [9]. Shatdal et al. [7] identify that when the hash table is larger than the cache size, almost every access to the hash table results in a cache miss. Consequently, partitioning the hash table into cache-sized chunks reduces cache misses and improves the join performance. Manegold et al. [4] 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*, which has become part of the standard *radix join* algorithm.

Partitioned Hash Join. The partitioning idea is illustrated in Figure 3. 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 3) on

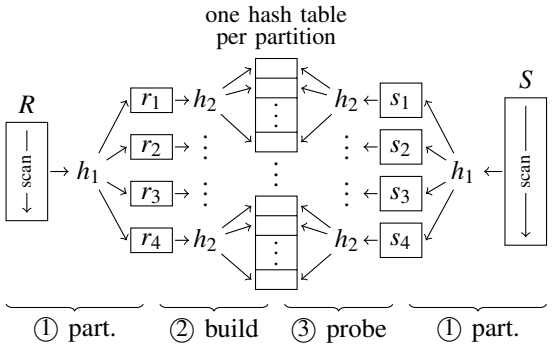


Fig. 3. Partitioned hash join (following Shatdal et al. [7]).

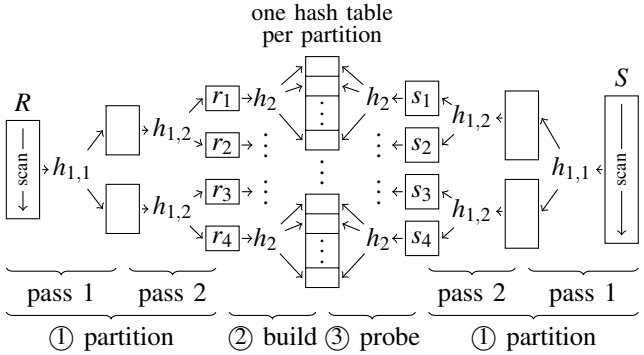


Fig. 4. *Radix join* (as proposed by Manegold et al. [4]).

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. [4] 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 this is called *radix partitioning*. For typical in-memory data sizes, two or three passes are sufficient to create cache-sized partitions, without suffering from TLB capacity limitations.

Radix Join. The complete *radix join* is illustrated in Figure 4. 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.

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 $O((|R|+|S|)\log |R|)$ for radix join.

Hardware Parameters. Radix join needs to be tuned to a particular piece of hardware essentially via two parameters: *(i)* the *maximum fanout* per radix pass is primarily limited by the number of TLB entries of the hardware; *(ii)* the resulting *partition size* should roughly be the size of the system's CPU cache. Both parameters can be obtained in a rather straightforward way, *e.g.*, with help of benchmark tools, such as *Calibrator* [4]. As we shall see later, *radix join* is not overly sensitive to a potential mis-configuration of either parameter.

D. Parallel Radix Join

Radix join can be parallelized by subdividing both input relations into sub-relations that are assigned to individual threads [1]. During the first pass, all threads create a *shared set of partitions*. As before, the number of partitions in this set is limited by hardware parameters and typically small (few tens of partitions). They are accessed by potentially many execution threads, creating a contention problem (the low-contention assumption of Section II-B no longer applies).

To avoid this contention, for each thread a dedicated range is reserved within each output partition. 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 4) that workers can perform work on their own. Load distribution among worker threads is typically implemented via task queueing (cf. [1]).

III. Experimental Setup

In this section we describe the experimental setup used for the evaluation of the algorithms.

A. Workload

For the comparison, we use machine and workload configurations that mimic scenarios where in-memory join processing is most relevant. In particular, all systems where the component truly matters assume a *column-oriented storage model*. We thus deliberately choose very narrow $\langle key, payload \rangle$ tuple configurations, where *key* and *payload* are four or eight bytes wide. As a side effect, narrow tuples better pronounce the effects that we are interested in, since they put more pressure on the system's caching system.[1]

[1]The effect of tuple widths was studied, *e.g.*, by Manegold et al. [10].

TABLE I
WORKLOAD CHARACTERISTICS

	A (from [2])	**B** (from [1])
size of *key*/*payload*	8/8 bytes	4/4 bytes
size of R	$16 \cdot 2^{20}$ tuples	$128 \cdot 10^6$ tuples
size of S	$256 \cdot 2^{20}$ tuples	$128 \cdot 10^6$ tuples
total size R	256 MiB	977 MiB
total size S	4096 MiB	977 MiB

TABLE II
HARDWARE PLATFORMS USED IN OUR EVALUATION

	Intel Nehalem	**Intel Sandy Bridge**	**AMD Bulldozer**	**Sun Niagara 2**
CPU	Xeon L5520 2.26 GHz	Xeon E5-2680 2.7 GHz	Opteron 6276 2.3 GHz	UltraSPARC T2 1.2 GHz
Cores/Threads	4/8	8/16	16/16	8/64
Cache sizes (L1/L2/L3)	32 KiB 256 KiB 8 MiB	32 KiB 256 KiB 20 MiB	16 KiB 2 MiB 16 MiB	8 KiB 4 MiB -
TLB (L1/L2)	64/512	64/512	64/1024	128/-
Memory	24 GiB DDR3 1066 MHz	32 GiB DDR3 1600 MHz	32 GiB DDR3 1333 MHz	16 GiB FBDIMM
VM Page size	4 KiB	4 KiB	4 KiB	8 KiB

We adopted the particular configuration of our workloads from existing work, which also eases the comparison of our results with those published in the past.

As illustrated in Table I, we adopted workloads from Blanas et al. [2] and Kim et al. [1] and refer to them as **A** and **B** here, respectively. All attributes are integers, and the keys of R and S follow a foreign key relationship. That is, every tuple in S is guaranteed to find exactly one join partner in R. Most of our experiments (unless noted otherwise) assume a uniform distribution of key values from R in S.

B. Hardware Platforms

We evaluated the algorithms on four different multi-core machines. Three are recent multi-core platforms, ranging from the older Intel Nehalem architecture to the newer Sandy Bridge architecture and including a recent AMD Bulldozer system (cf. Table II). Sun UltraSPARC T2 dates back to 2007 and provides eight thread contexts per core where eight threads share the L1 cache with a line size of 16 bytes. The two Intel machines support SMT with two thread contexts per core. Sun UltraSPARC T2 comes with two levels of cache, where cores share the L2 cache with line size of 64 bytes. On the Intel machines, cores use a shared L3 cache and a cache line size of 64 bytes. AMD machine has a different architecture than the others where two cores are packaged as single module and share some resources such as instruction fetch, decode, floating point unit and L2 cache. Accordingly, the effective L2 cache available per core is reduced to half, i.e., 1 MiB.

The Intel and AMD systems run stock Ubuntu Linux (kernel version 2.6.32) and Sun UltraSPARC T2 runs a Debian Linux (kernel version 3.2.0-3-sparc64-smp). For the results we report here, we used `gcc` 4.4.3 on Ubuntu and `gcc` 4.6.3 on Debian and the `-O3` and `-mtune=niagara2 -mcpu=ultrasparc` command line options to compile our code. Additional experiments using Intel's `icc` compiler did not show any notable differences, qualitatively or quantitatively. For the performance counter profiles that we report, we instrumented our code with the Intel Performance Counter Monitor [11].

TABLE III
EFFECT OF SORTED INPUT ON THE BUILD PHASE (CODE BY [2] VS. OUR OWN CODE; CPU PERFORMANCE COUNTERS REPORTED IN MILLIONS)

	Cycles	L3 miss	Instr.	TLB miss
Code of [2], sorted input	322	2	2215	1
Code of [2], unsorted input	1415	45.3	2263	52.7
Our code, unsorted input	966	25	572	56

IV. HARDWARE-OBLIVIOUS JOINS

In this section we first study and optimize the *no partitioning* strategy. To make our results comparable, we use similar hardware to that in earlier work, namely a Nehalem L5520 system (cf. Table II).

A. Build Cost

The overall cost of hardware-oblivious *no partitioning* join is given by

$$cost = \underbrace{c_{put} \cdot |R|}_{\text{build cost}} + \underbrace{c_{get} \cdot |S|}_{\text{probe cost}} ,$$

where c_{put} and c_{get} denote the (constant) cost of adding or reading an entry to/from the hash table (respectively). Writing to the hash table is generally more expensive, since it involves the acquisition of a bucket latch, hence, $c_{put} \gtrsim c_{get}$.

No partitioning was proposed and evaluated by Blanas et al. in [2]. Surprisingly, in their experiments—based on what we call Workload **A** in our work—the build phase accounts for only 2 % of the overall execution time. In this workload, $|R| = 1/16 \cdot |S|$, so we would expect the build phase to take at least $\approx 6\,\%$ of the overall cost.

The code used to obtain these results is publicly available [12]. Analysis of this code reveals that their results are based on experiments where R is *pre-sorted* and the hash function is a modulo function. As a result, as data items are hashed, they map to consecutive hash buckets, leading to strictly sequential memory accesses. The sorted input also removes any contention for the bucket latch.

Re-running the experiments with randomly permuted input (i.e., the general case) results in build costs of about 6 %, consistent with our assumption stated above. To confirm that our assessment is correct, we collected cache profile data. Table III illustrates how sorted input essentially eliminates all TLB and L3 cache misses. Otherwise, we could basically reproduce other performance results (cf. Figure 5, dark bars).

B. Cache Efficiency

The cache profile information in Table III also indicates hash table build-up incurs a very high number of cache and

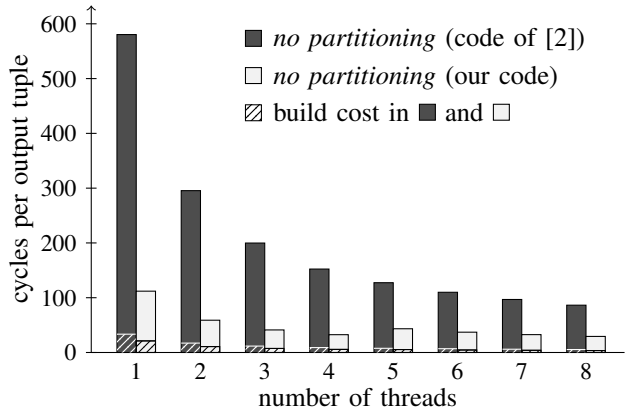


Fig. 5. Cycles-per-output-tuple for hardware-oblivious *no partitioning* strategy (Workload **A**; Intel Xeon L5520, 2.26 GHz).

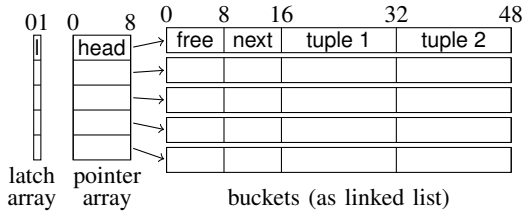


Fig. 6. Original hash table implementation.

TLB misses. Processing 16 million tuples results in 45.3/52.7 million L3/TLB misses, or about three misses per input tuple.

The reason for this inefficiency becomes clear as we look at the code of [2]. The hash table in this code is implemented as illustrated in Figure 6. That is, 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. In practice, each of these three steps likely results in a cache miss.

Optimized Hash Table Implementation. To improve the cache efficiency of *no partitioning*, in our re-implementation we directly combined locks and hash buckets to neighboring memory locations. More specifically, in our code we implemented the main hash table as a *contiguous array* of buckets, as shown in Figure 7. The hash function directly indexes into this array representation. For overflow buckets,

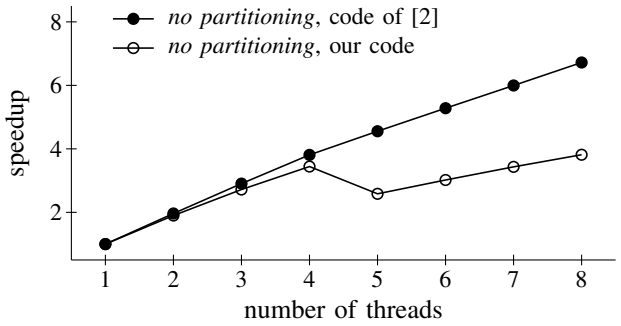


Fig. 8. Speedup of *no partitioning* algorithm on SMT hardware. First four threads are "native" threads; threads 5–8 are "hyper threads."

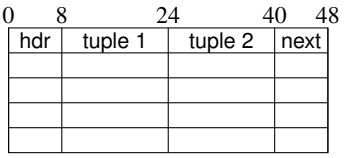


Fig. 7. Our hash table implementation.

we allocate additional bucket space outside the main hash table. Most importantly, the 1-byte synchronization latch is part of the 8-byte header that also contains a counter indicating the number of tuples currently in the bucket. In line with the original study [2], we configured our hash table to two 16-byte tuples per bucket, and an 8-byte next pointer chains hash buckets in the case of overflows.

The effect of this modified hash table representation is significant. As listed in Table III, it cuts by half the number of cache misses in the build phase (and also in the probe phase, though not shown in Table III) and speeds up join processing by a fair margin.

In terms of absolute join performance, our re-written code is roughly three times faster than the code of Blanas et al. [2], as shown in Figure 5. Yet, our code remains strictly hardware-oblivious: no hardware-specific parameters are needed to tune the code.

C. The Role of SMT Threads

Blanas et al. [2] argue that *no partitioning* draws its true benefit from its good interplay with simultaneous multi-threading (SMT) hardware. Simply speaking, 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. This gives the hardware the flexibility to perform useful work even when one of the threads is stalled, *e.g.*, because of a cache miss.

To study the interaction between *no partitioning* and SMT, we repeated the original SMT experiment [2] on comparable hardware. Our Nehalem system contains four cores with two hardware contexts each. As in the original study, we start by assigning threads to different physical cores. Once the phsysical cores are exhausted, we assign threads to the available hardware context in a round-robin fashion.

Figure 8 illustrates the performance of *no partitioning* relative to the performance of a single-threaded execution of the same algorithm ("speedup"). Our experiment indeed

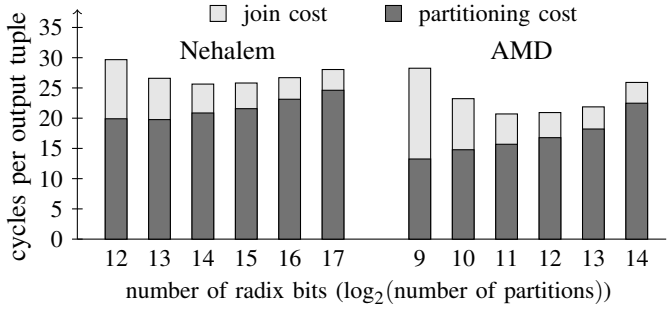


Fig. 9. Cost vs. radix bits (Workload **B**; Nehalem: 2-passes; AMD: 1-pass).

confirms the scalability with SMT threads on the un-optimized code of [2]. However, once we run the same experiment with our optimized code (with significantly better absolute performance, cf. Figure 5), SMT does not help the *no partitioning* strategy at all.

As the result shows, SMT can only remedy cache miss latencies if the respective code contains enough cache misses and enough additional work for the second thread while the first one is waiting. For code with less redundancy, SMT brings only negligible benefit. These results raise questions about a key hypothesis behind the hardware-oblivious *no partitioning* strategy.

V. Hardware-Conscious Joins

We perform a similar analysis for the *parallel radix join*. Blanas et al. [2] also provide an implementation for this hardware-conscious join execution strategy.

A. Configuration Parameters

The key configuration parameter of *radix join* is the number of *radix bits* for the partitioning phase ($2^{\#\text{radix bits}}$ partitions are created during that phase). Figure 9 illustrates the effect that this parameter has on the runtime of *radix join*.

The figure confirms the expected behavior that partitioning cost increases with the partition count, whereas the join phase becomes faster as partitions become smaller. Configurations with 14 and 11 radix bits are the best trade-offs between these opposing effects for the Nehalem and AMD architectures, respectively. But even more interestingly, the figure shows that *radix join* is fairly robust against a parameter misconfiguration: within a range of configurations, the performance of *radix join* degrades only marginally.

B. Hash Tables and Cache Efficiency

Following the partitioning of the input tables, hash tables are very small and always fully cache resident. Thus, our assessment about cache misses for hash table accesses in the previous section no longer holds for the hardware-conscious join execution strategy.

Various implementations have been proposed for radix join. Manegold et al. [4] use a rather classical bucket chaining mechanism, where individual tuples are chained to form a bucket. Following good design principles for efficient in-memory algorithms, all pointers are implemented as array position indexes (as opposed to actual memory pointers).

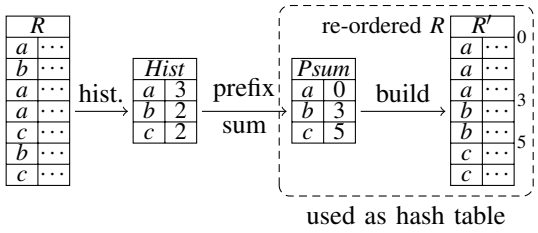


Fig. 10. Relation re-ordering and histogram-based hash table design.

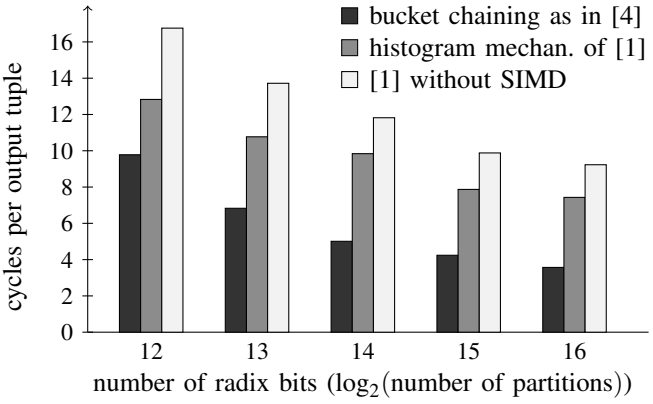


Fig. 11. Cost of join phase in *radix join* for three different hash table implementation techniques (Workload **B**; Intel Xeon L5520, 2.26 GHz; Using 8 threads and 2 pass partitioning).

Kim et al. [1] build their hash table analogously to the parallel partitioning stage. The input relation 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 the same 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 10.

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.

Evaluation. We evaluated the impact of different hash table implementation strategies on the join phase of *radix join*. Figure 11 shows the join phase cost in cycles per output tuple for three different strategies.

As can be seen, the Manegold et al. implementation [4] still has an edge over the more recent one by Kim et al. [1], in spite of the potential for SIMD optimization in the latter implementation. The graph also confirms that the join cost generally decreases as the input data is partitioned in a more fine-granular way. In practice, there is a sweet spot, because the partitioning cost (which has to be invested before joining) increases with the partition count.

TABLE IV
CPU PERFORMANCE COUNTER PROFILES FOR DIFFERENT RADIX JOIN IMPLEMENTATIONS (IN MILLIONS)

	code from [2]			our code		
	Part.	Build	Probe	Part.	Build	Probe
Cycles	9398	499	7204	5614	171	542
Instructions	33520	2000	30811	17506	249	5650
L2 misses	24	16	453	13	0.3	2
L3 misses	5	5	40	7	0.2	1
TLB misses	9	0.3	2	13	0.1	1

Since the Manegold et al. approach comes out best in this comparison, we will use it for all following experiments. We note that the choice we are making here does not depend on hardware parameters (this is a hardware-oblivious optimization). As we shall see in a moment, the impact of our choice is limited, however, since the cost of partitioning adds to either of those implementation techniques.

C. Overall Execution Time

The overall cost of join execution consists of the cost for data partitioning and the cost for computing the individual joins over partitions. To evaluate the overall cost of join execution (and to prepare for a comparison with the hardware-oblivious *no partitioning* algorithm), we measured our own, carefully tuned implementation, as well as those reported in earlier work.

We had two implementations of *radix join* available. For the code of Blanas et al. [12], we found one pass and 2048 partitions to be the optimal parameter configuration (which matches the configuration in their experiments [2]). Partitioning in that code turns out to be rather expensive. We attribute this to a coding style that leads to many function calls and pointer dereferences in critical code paths. Partitioning is much more efficient in our own code. This leads to a situation where two-pass partitioning with 16,384 partitions becomes most efficient. Table IV illustrates how the different implementations lead to significant differences in the executed instruction count. Our code performs two partitioning passes with 40 % fewer instructions than Blanas et al.'s code [2] that needs to perform only one pass.

Resulting overall execution times are reported (as cycles per output tuples) in Figure 12. This chart confirms that partitioning is rather expensive in the code of Blanas et al. Ultimately, this results in a situation where the resulting partition count is sub-optimal for the subsequent join phase, causing their join code to be also expensive. With optimized code, partitioning becomes the dominant cost, which is consistent with the findings of Kim et al. [1] that showed comparable cost at similar parameter settings. Overall, our code is about three times faster than the code of Blanas et al. for all shown configurations.

Performance Counters. We also instrumented the available *radix join* implementations to monitor CPU performance counters. Table IV lists cache and TLB miss counts for the three

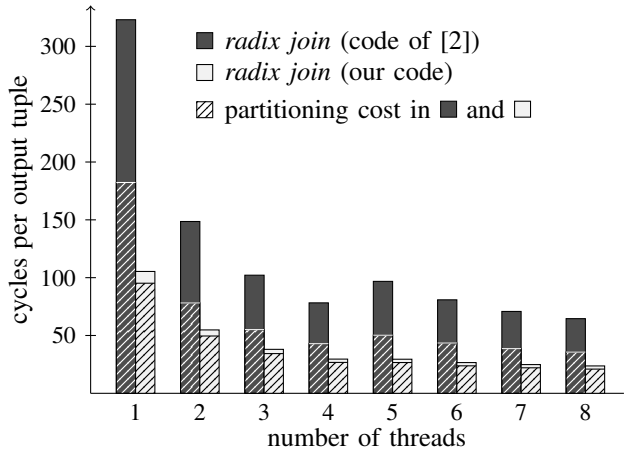


Fig. 12. Overall join execution cost (cycles per output tuple) for hardware-conscious *radix join* strategy (Workload **A**; Intel Xeon L5520, 2.26 GHz).

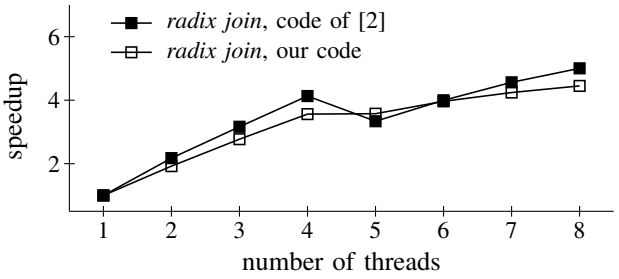


Fig. 13. Speedup of *radix* algorithm on SMT hardware. First four threads are "native" threads; threads 5–8 are "hyper threads."

tasks in *radix join*.

The table shows a significant difference in the number of cache and TLB misses between the implementation of Blanas et al. and ours. The idea behind *radix join* is that all partitions should be sufficiently small to fully fit into caches, so one should expect a very low number of misses, which is true for our implementation, but not for the one of Blanas et al.

The reason for the difference is an unfortunate execution order of hash building and probing in the latter code. Their code performs *radix join* strictly in three phases. After partitioning (first phase), hash tables are created for *all* partitions (second phase). Only then, in the third algorithm phase, are those hash tables probed to find join partners. Effectively, created hash tables will long be evicted from CPU caches, before their content is actually needed for probing. Our code avoids these unnecessary memory round-trips by running build and probe for each partition together.

D. Speedup from SMT Threads

Figure 13 shows that neither of the two *radix join* implementations that we evaluated can 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 (i.e., caches, TLBs) between threads. These results are also in line with the results of Blanas et al. [2]. 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 our code against that of Kim et al. [1]. With our optimized implementation, we achieve a speedup of 4.6, very close to the 4.4 factor reported by Kim et al. on a simliar Intel Nehalem processor (at comparable absolute performance).

VI. Hardware-Conscious or Not?

In this section we compare compare the algorithms above under a wide range of parameters and hardware platforms.

A. Effect of Workloads

The results of extended experiments over all workloads and hardware platforms are summarized in Figure 14. Figure 14(a) shows the performance of our own implementation using Workload **A** and the hardware platforms we had available (this workload was originally proposed by Blanas et al. [2] to support their hardware-oblivious *no partitioning* algorithm).

While Blanas et al. [2] reported only a marginal performance difference between *no partitioning* and *radix join* on x86 architectures, the hardware-conscious *radix join* is appreciably faster if both implementations are equally optimized. Only on our Sun Niagara system, the situation looks different. We will look into this architecture in more depth in a moment.

The results in Figure 14(a) may still be seen as a good argument for the hardware-oblivious approach to join execution. An approximate 25 % performance advantage, *e.g.*, on the two Intel platforms might not warrant the necessary parameter tuning for *radix join*.

Running the same experiments with our second workload, Workload **B** (Figure 14(b)), however, radically changes the picture. *Radix join* is approximately 3.5 times faster than *no partitioning* on Intel machines and 2.5 times faster on AMD and Sun machines. That is, *no partitioning* only has comparable performance to *radix join* when the relative relation sizes are very different, since that situation minimizes the cost of the build phase. As soon as the table sizes grow and become similar in size, the overhead of not being hardware-conscious becomes clearly visible (see the differences in the build phases for *no partitioning*).

B. Scalability

To study the scalability of our two join variants, we re-ran our experiments with a varying number of threads, up to the maximum number of hardware contexts available on each of our architectures. Figure 15 illustrates the results.

Besides the SMT issues that we already discussed in Sections IV-C and V-D, all platforms and both join implementations show good scalability behavior. Thanks to this scalability, our *radix join* implementation reaches a throughput of 196 million tuples per second. As far as we are aware, this is the highest throughput reached for in-memory hash joins so far.

On the AMD machine, *no partitioning* shows a clear bump around 8–10 threads. This is an artifact of the particular AMD architecture. Though the Opteron is marketed as a 16-core processor, the chip internally consists of two interconnected CPU dies [13]. It is likely that such an architecture requires a tailored design for the algorithms to perform well, removing an argument in favor of hardware-conscious algorithms as, even if it is parameter-free, some multi-core architectures may require specialized designs anyway as NUMA would create significant problems for the shared hash table used in *no partitioning* (let alone future designs where memory may not be coherent across the machine).

TABLE V
Latch cost per build tuple in different machines

	Nehalem	Sandy Bridge	Bulldozer	Niagara 2
Used instruction	`xchgb`	`xchgb`	`xchgb`	`ldstub`
Reported instruction latency in [14], [15]	∼20 cycles	∼25 cycles	∼50 cycles	3 cycles
Measured impact per build tuple	7-9 cycles	6-9 cycles	30-34 cycles	1-1.5 cycles

C. Sun UltraSPARC T2 "Niagara"

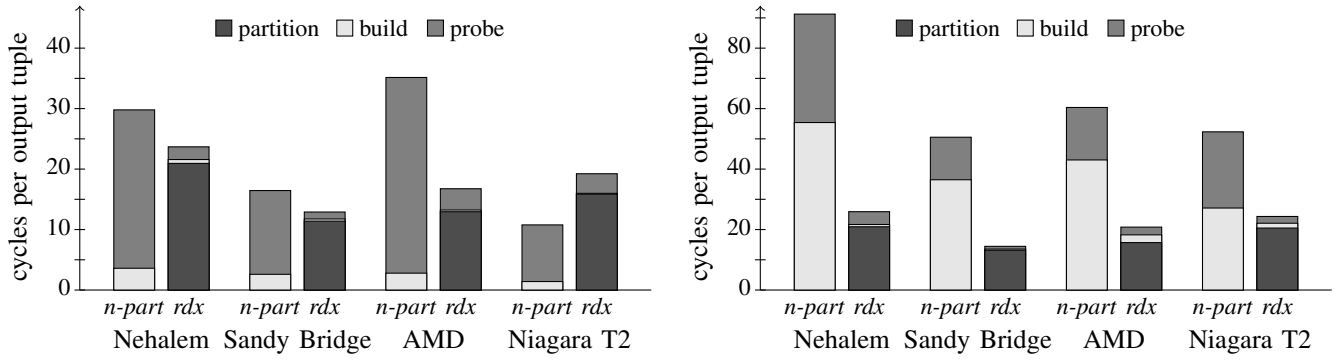
On the Sun UltraSPARC T2, with a totally different architecture than the x86 platforms, we see a similar result with Workload **B**. Hardware-conscious *radix join* achieves a throughput of 50 million tuples per second (cf. Figure 15(d)), whereas *no partitioning* achieves only 22 million tuples per second.

However, when looking to Workload **A**, *no partitioning* becomes almost two times faster than *radix join* on the Niagara 2 (shown in Figure 14(a)). One could attribute this effect to the highly effective on chip multi-threading functionality of the Niagara 2. However, there is more than that. First, the virtual memory page size on UltraSPARC T2 is 8 KiB. This in turn, reduces the number of virtual memory pages that the hash table spans in *no partitioning* join by half. As a result, the random access behavior and its impact in TLB misses is also reduced to a certain degree, compared to other architectures which use 4 KiB pages.

Second, the Niagara 2 architecture turns out to have extremely efficient thread synchronization mechanisms. To illustrate that, we deliberately disabled the latch code in the *no partitioning* join. We found out that the `ldstub` instruction which is used to implement the latch on UltraSPARC T2 is very efficient compared to other architectures as shown in Table V. These special characteristics of Sun UltraSPARC T2 also show the importance of architecture-sensitive decisions in algorithm implementations.

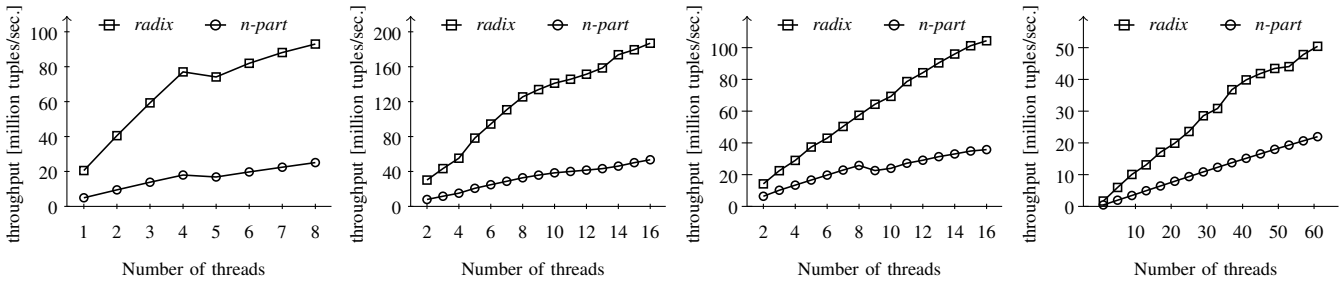
D. TLB and Virtual Memory Page Sizes

In-memory hash joins are known to be sensitive to the *virtual memory subsystem* of the underlying system, in particular to the caching of address translations via *translation look-aside buffers (TLBs)*. The virtual memory setup of modern systems is, to a small extent, configurable. By changing a system's *page size*, every address mapping (potentially cached in the TLB) covers a different amount of main memory, and with a



(a) Workload **A** (256 MiB ⋈ 4096 MiB) (b) Workload **B** (977 MiB ⋈ 977 MiB)

Fig. 14. Cycles per output tuple for hardware-oblivious *no partitioning* and hardware-conscious *radix join* algorithm, for different hardware architectures and workloads. Experiments based on our own, optimized code. Using 8 threads on Nehalem, 16 threads on Sandy Bridge and AMD, and 64 threads on Niagara.



(a) Intel Nehalem Xeon L5520 (b) Intel Sandy Bridge E5-2680 (c) AMD Bulldozer Opteron (d) Sun UltraSPARC T2

Fig. 15. Throughput comparison of algorithms on different machines using Workload **B**.

large page size, fewer TLB entries might be needed for the operations on a given memory region.

Intel Nehalem hardware can essentially be operated in either of two modes with the support of the OS [16]: *(i)* with a page size of 4 KiB (the default), the level 1 data TLB can hold up to 64 memory mappings; *(ii)* alternatively, when the page size is set to 2 MiB, only 32 mappings can be cached in TLB1. Here we study the effect of these two options on join performance.

No Partitioning Joins. During the hash table build and probe phases, the hardware-oblivious *no partitioning* join algorithm randomly accesses an element in the hash table that is created for the smaller join relation *R*. For our workload configuration **A**, this hash table is 384 MiB in size (tuples plus latches and bucket structure). Consequently, the chance to hit a memory page that is cached in TLB1 is $^{64}/_{98304}$ or $^{32}/_{192}$, depending on whether the system is configured for a 4 KiB or 2 MiB page size (respectively). The latter configuration might thus significantly reduce the number of TLB misses experienced and thus improve execution performance.

As listed in Table VI, we could indeed observe a performance improvement for *no partitioning* with larger pages. The dominating cost of *no partitioning* are actual data cache misses, however, which are unaffected by the page size configuration. This is why the performance improvement remains limited to about 15 % in our configuration. Projecting to larger *R* relations, the effect will be even less, since the TLB1 hit probability quickly decreases as the hash table size goes up.

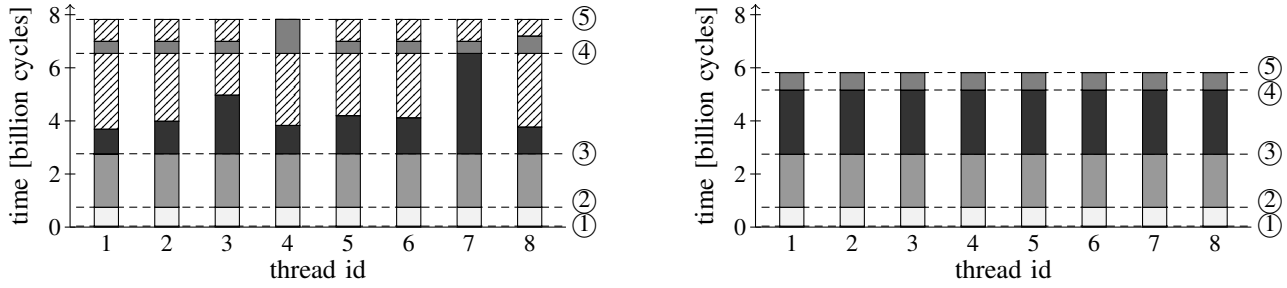
TABLE VI
PERFORMANCE OF NO PARTITIONING JOIN WHEN USING LARGE PAGES

No Partitioning Join	4 KiB pages	2 MiB huge pages
Build cycles per build tuple	57.92	49.74
Probe cycles per output tuple	26.10	22.88
Overall cycles per output tuple	29.72	25.99

Radix Join. Our hardware-conscious algorithm, *radix join*, is even more sensitive to TLB behavior. In fact, the TLB size is typically the limiting factor that determines the maximum number of partitions that can be created per partitioning pass. Since the 64-entry TLB1 of our system is assisted by a 512-entry shared TLB2, our Nehalem system actually achieved best join performance with two 128-way passes (cf. Section V-A).

Changing the system page size now may have opposing effects. On the one hand side, a 2 MiB page size reduces the number of available TLB entries (only 32 TLB1 entries). But on the other hand side, the in-memory page table structure of the system's virtual memory setup becomes smaller; fewer page tables have to be traversed for every TLB miss. In effect, the cost of a single TLB miss gets reduced.

Table VII illustrates what these opposing effects mean to the join performance of our Nehalem system. For the workload we used, changing the system page size to 2 MiB shifted the optimal *radix join* configuration to a single-pass 12-bit partitioning phase (with a throughput improvement of ≈ 15 %).



(a) Simple task queueing.

(b) Task decomposition for large partition/join tasks.

Fig. 16. Barrier synchronization cost in *radix join* (Workload **A**; foreign key distribution in S skewed with Zipf parameter $z = 1.5$; tasks that make progress are indicated using shades of gray; wait time for barrier synchronization indicated as ▨). Simple task queueing leaves many tasks under-utilized (leading to significant wait times ▨ in (a)). Fine-granular task decomposition in (b) (similar to of [1]) improves load distribution and increases join throughput by 25 %.

TABLE VII
PERFORMANCE OF RADIX JOIN WHEN USING LARGE PAGES

Radix Join	4 KiB pages (2 pass / 14 bits)	2 MiB huge pages (2 pass / 14 bits)	2 MiB huge pages (1 pass / 12 bits)
Partitioning cycles per input tuple	19.73	21.71	15.54
Join cycles per output tuple	2.77	2.75	3.64
Overall cycles per output tuple	23.74	25.81	20.15

Large Pages or Not? The above measurements indicate a performance advantage of systems that use a large page size configuration. But we note that this is a two-edged sword. Large pages generally increase the memory footprint of every process in the system, which in productive systems might be more problematic than in our micro-benchmarks.

In our benchmarks, both join strategies equally benefit from large pages, leaving the *"hardware-conscious or not?"* question unchanged. As already mentioned above, the small benefit from large pages might also disappear as input data sizes are scaled up in future systems.

E. Barrier Synchronization and Load Balancing

In the two join variants that we consider in this work, parallel execution threads synchronize in two ways: *(i)* write accesses to the shared hash table in *no partitioning* are protected by *latches*; *(ii)* both join variants operate in multiple phases which are separated by *barriers*. While barrier synchronization allows threads to perform a lot of work independently, there is a risk for wasted *idle time* when work is distributed unevenly over threads, so threads have to wait for each other.

Barrier synchronization is not a problem for the *no partitioning* join execution strategy since, by construction, tuples are distributed evenly across threads and per-tuple cost is basically independent of the tuple values.

Radix join, by contrast, is more vulnerable to penalties due to barrier synchronization whenever tasks are not scheduled properly over available worker threads (we discussed a related issue, the cache locality problem of the *radix join* implementation of [2], already in Section V-C). In total, *radix join* consists of five processing stages (assuming a two-pass partitioning scenario): ① compute local histogram for R; ② compute local histogram for S; ③ partitioning pass 1; ④ partitioning pass 2; ⑤ join phase (partition-wise build and probe). And while threads are guaranteed to receive an equal share of input data in the first three stages, partition sizes produced by stage ④ depend on the distribution of values in R and S.

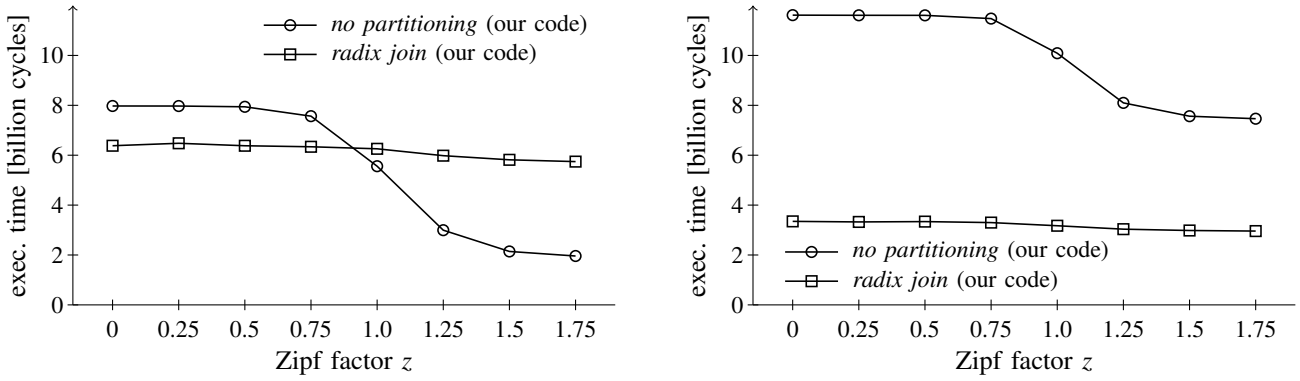
To study the potential load imbalance, we modified our data generator to produce a heavily skewed input data set. Foreign keys in S no longer reference keys in R with a uniform likeliness, but according to a Zipf distribution law with Zipf factor $z = 1.5$. Figure 16(a) illustrates, for each of the eight threads in our system (x axis), the type of work it is doing as time progresses (along the y axis).

As can be seen in the figure, all threads perform useful work near the beginning of each execution stage (indicated through different gray shades). But some finish their stage earlier than others, meaning that they have to *wait* until their last peer finishes the stage (threads 7 and 4 in the figure). The resulting *idle times*, indicated as ▨, waste CPU resources without any real thread progress.

Fine-Granular Task Decomposition. The barrier synchronization problem in Figure 16(a) is an outcome of the *task queueing* mechanism that we adopted from [2] to distribute load. This mechanism is insufficient to adapt to skewed input data.

To combat the problem, we modified our *radix join* implementation to perform *task decomposition*, similar to the strategy proposed by Kim et al. [1]. In a nutshell, whenever a partition after stage ③ significantly exceeds its expected size (as it would result from a uniform distribution), we break up the partition into smaller chunks that are handled by all threads in concert. This avoids such partitions that can "hog" one of the execution threads and affect overall throughput.

Figure 16(b) illustrates the effect on the execution progress. The modification successfully avoids load imbalances and speeds up join execution by about 25 %. Though the improved scheduling mechanism applies mainly to the *radix join* algorithm, we note that its realization is actually *parameter-free* (and not in itself a hardware-conscious optimization).



(a) Workload **A** (256 MiB ⋈ 4096 MiB)

(b) Workload **B** (977 MiB ⋈ 977 MiB)

Fig. 17. Join performance when foreign key references follow a Zipfian data distribution (Intel Xeon L5520, 2.26 GHz).

F. Skewed Data

In this section, we study the effects of skew following the same methodology of Blanas et al. [2]. More specifically, we populate the foreign key column (table *S*) of our data sets such that the probability of referencing individual key values (of *R*) follows a Zipf distribution law (we varied the Zipf factor between $z = 0$ and $z = 1.75$).

Figure 17 illustrates how *no partitioning* and *radix join* react to skew. The graphs confirm that skew helps the performance of the hardware-oblivious *no partitioning* join, which was observed already by Blanas et al. [2] and claimed "a big advancement over state-of-the-art" methods. Ultimately, *no partitioning* surpasses *radix join* in join throughput when using Workload **A**.

The observation does not come as a surprise, however, and only happens for data that is heavily skewed. For instance, in the "low skew" case of [2] ($z = 1.05$), the most frequent value in *S* occurs with a probability of 8.4 %; the chance to hit one of the 600 most frequent join keys (out of 16 million) already exceeds 50 %. For the "high skew" case of [2] ($z = 1.25$), more than 22 % of all *S* tuples carry the same value and the chance to hit one of the top-600 values is more than 83 %. Effectively, even a small L1 cache is sufficient to hold the small hot set of *R* that is relevant during the probe phase.

Our results indicate that the benchmark configuration of [2] (very high skew, suitable relation sizes) hits a sweet spot of the *no partitioning* algorithm. This can be seen also in Figure 17(b), where the same experiment with Workload **B** does not help *no partitioning* as much as the previous configuration did.

Performance improvement with increasing skew can be seen as an advantage of *no partitioning*. The effect also means, however, that the runtime characteristics of the algorithm becomes dependent on the input data distribution and thus difficult to predict (*e.g.*, by a cost-based query optimizer). *Radix join*, by contrast, offers predictable performance over a wide range of skew, a characteristic that is desirable in the context of *robust query processing*, an important and active criterion especially for productive query processors [17].

G. Effect of Relation Size Ratio

The experiments above show that relative sizes of the tables to join play a big role in the 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 18 shows the cycles per output for each phase as well as the entire run for different *R* sizes in a log-log plot.

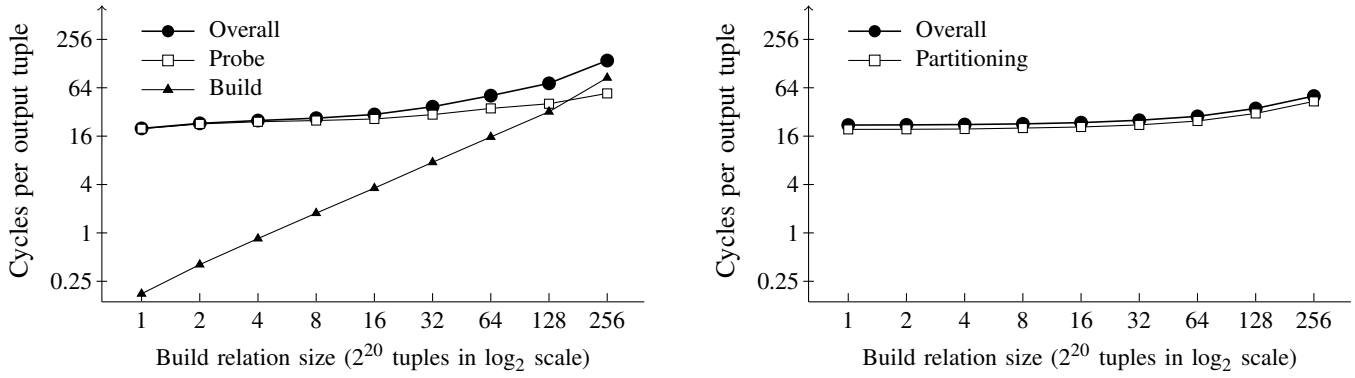
The results confirm the observation made so far and provide a clearer answer to the controversy between hardware-conscious and hardware-oblivious algorithms. *No partitioning* does very well when the build relation is very small compared to the large relation. Performance goes down as the size of *R* increases because of the cost of the build phase (Figure 18(a)). *Radix join* is much more robust to different table sizes and offers almost constant 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, *no partitioning* join is better that *radix join* only under skew and when the sizes of the tables being joined significantly differs. In all other cases, *radix join* is better (and significantly better in fact) in addition to also being more robust to different parameters like skew or relative table sizes.

VII. Related Work

After Manegold et al. [8] and Ailamaki et al. [9] both demonstrated the importance of memory and caching effects on modern computing hardware, soon new algorithm variants emerged to run classical database problems efficiently on modern hardware.

One of the design techniques to achieve this goal is the use of *partitioning*, which we discussed extensively also in this work. Besides a use for in-memory joins, partitioning is relevant also, *e.g.*, to perform *aggregation*, as investigated



(a) No partitioning join

(b) Radix Join

Fig. 18. Cycles per output tuple with varying build relation cardinalities in Workload **A** (Intel Xeon L5520, 2.26 GHz, Radix join was run with the best configuration in each experiment where radix bits varied from 13 to 15).

recently by Ye et al. [18]. And while the aggregation problem differs from join computation in many ways, the observations made by Ye et al. about different hardware architectures are very consistent with ours.

While here we mainly looked at local caching and memory latency effects, we earlier demonstrated how the *topology* of modern NUMA systems may add additional complexity to the join problem [19]. *Handshake join* is an evaluation strategy on top of existing join algorithms to make those algorithms topology-aware.

With a similar motivation, Albutiu et al. [3] proposed to use *sort-merge algorithms* to compute joins, leading to a hardware-friendly sequential memory access pattern. It remains unclear, however, whether the switch to a parallel merge-join is enough to adequately account for the topology of modern NUMA systems.

Similar in spirit to the *no partitioning* join is the recent GPU-based join implementation proposed by Kaldewey et al. [20]. Like in *no partitioning*, the idea is to leverage hardware SMT mechanisms to hide memory access latencies. In GPUs, this idea is pushed to an extreme, with many threads/warps sharing one physical GPU core.

VIII. Conclusion

The results above resolve the contradictions among existing results conclusively: hardware-oblivious algorithms only work well under a narrow parameter window (when the table sizes significantly differ) and on one particular hardware platform. Moreover, with the novel ideas introduced in the paper, hardware-conscious algorithms can be made significantly faster than what has been published so far and more robust to a wider set of parameters. These algorithms can also be easily tuned to the underlying hardware, as shown in the paper, significantly reducing the argument that they are more difficult to port than their hardware-oblivious counterparts.

References

[1] 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*, vol. 2, no. 2, pp. 1378–1389, 2009.

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

[3] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems," *PVLDB*, vol. 5, no. 10, pp. 1064–1075, 2012.

[4] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing Main-Memory Join on Modern Hardware," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 4, pp. 709–730, 2002.

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

[6] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of Hash to Data Base Machine and Its Architecture," *New Generation Comput.*, vol. 1, no. 1, pp. 63–74, 1983.

[7] A. Shatdal, C. Kant, and J. F. Naughton, "Cache Conscious Algorithms for Relational Query Processing," in *VLDB*, 1994, pp. 510–521.

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

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

[10] S. Manegold, P. Boncz, N. Nes, and M. Kersten, "Cache-conscious radix-decluster projections," in *VLDB*, Toronto, ON, Canada, Sep. 2004, pp. 684–695.

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

[12] 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.

[13] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[14] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," http://www.agner.org/optimize/instruction_tables.pdf, online, accessed July 2012.

[15] Sun, "UltraSPARC T2^{TM} Supplement to the UltraSPARC Architecture 2007," http://sosc-dr.sun.com/processors/UltraSPARC-T2/docs/UST2-UASuppl-HP-ext.pdf, online, accessed July 2012.

[16] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf, online, accessed July 2012.

[17] G. Graefe, "Robust query processing," in *ICDE*, 2011, p. 1361.

[18] Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable aggregation on multi-core processors," in *DaMoN*, 2011, pp. 1–9.

[19] J. Teubner and R. Müller, "How soccer players would do stream joins," in *SIGMOD Conference*, 2011, pp. 625–636.

[20] T. Kaldewey, G. M. Lohman, R. Müller, and P. B. Volk, "GPU join processing revisited," in *DaMoN*, 2012, pp. 55–62.