Main-Memory Hash Joins
on Modern Processor Architectures

Cagri Balkesen, *Member, IEEE,* Jens Teubner, *Member, IEEE,* Gustavo Alonso, *Fellow, IEEE,*
and M. Tamer Özsu, *Fellow, IEEE*

Abstract—Existing main-memory hash join algorithms for multi-core can be classified into two camps. *Hardware-oblivious* hash join variants do not depend on hardware-specific parameters. Rather, they consider qualitative characteristics of modern hardware and are expected to achieve good performance on any technologically similar platform. The assumption behind these algorithms is that hardware is now good enough at hiding its own limitations—through automatic hardware prefetching, out-of-order execution, or simultaneous multi-threading (SMT)—to make hardware-oblivious algorithms competitive without the overhead of carefully tuning to the underlying hardware. *Hardware-conscious* implementations, such as *(parallel) radix join*, aim to maximally exploit a given architecture by tuning the algorithm parameters (*e.g.*, hash table sizes) to the particular features of the architecture. The assumption here is that explicit parameter tuning yields enough performance advantages to warrant the effort required.

This paper compares the two approaches under a wide range of workloads (relative table sizes, tuple sizes, effects of sorted data, etc.) and configuration parameters (VM page sizes, number of threads, number of cores, SMT, SIMD, prefetching, etc.). The results show that *hardware-conscious* algorithms generally outperform *hardware-oblivious* ones. However, on specific workloads and special architectures with aggressive simultaneous multi-threading, *hardware-oblivious* algorithms are competitive. The main conclusion of the paper is that, in existing multi-core architectures, it is still important to carefully tailor algorithms to the underlying hardware to get the necessary performance. But processor developments may require to revisit this conclusion in the future.

Index Terms—Databases, Query Processing, Modern Hardware, Multi-Core and Parallelism

✦

1 INTRODUCTION

The radical changes and advances in processor architecture caused by multi-core have triggered a revision of the algorithms used to implement main-memory hash joins. The results available so far present a very confusing and contradictory landscape.

On the one hand, some authors claim that the best performance is to be achieved by fine tuning to the underlying architecture. For instance, Kim et al. [1] look at the effects of caches and TLBs (translation lookaside buffers) on main-memory parallel hash joins and show how careful partitioning according to the cache and TLB sizes leads to improved performance. Along the same lines, Lang et al. [2] have shown how tuning to the non-uniform memory access (NUMA) characteristics also leads to improved performance of parallel hash joins. We will refer to the algorithms that take hardware characteristics into consideration as *hardware-conscious*. Some of this work further emphasizes the impact of hardware by suggesting that, in the future, as SIMD becomes wider, sort-merge join is likely to perform better than hash joins [1].

On the other hand, other authors argue that parallel hash 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 in the multi-layer memory hierarchy. In addition, so the argument goes, the fine tuning of the algorithms to specific hardware makes them less portable and less robust to, e.g., data skew.

Further contradictory results exist around the performance of sort-merge joins. For instance, Albutiu et al. [4] claim that sort-merge join is already better than hash join and can be efficiently implemented without using SIMD [4]. These results contradict the claims of both Blanas et al. [3] and Kim et al. [1] (the former argues that hardware consciousness is not necessary to achieve high performance; the latter concludes that sort-merge will only be competitive on wide SIMD architectures). We recently looked into this question in more detail, too [5] and found that hardware-conscious (radix-)hash join algorithms still maintain an edge over sort-merge joins despite the advances on SIMD.

In this paper we focus on main-memory, parallel hash join algorithms and address the question of whether it is important to tune the algorithms to the underlying hardware as claimed by Kim et al. [1] and by Albutiu et al. [4] or whether *hardware-oblivious* approaches provide sufficient performance to make the overhead of being hardware-conscious unnecessary.

Answering this question is a non-trivial task because

Balkesen and Alonso are with the Systems Group, Department of Computer Science, ETH Zurich, Switzerland.
E-mail: firstname.lastname@inf.ethz.ch.
Teubner is with the DBIS Group, Department of Computer Science, TU Dortmund University, Germany.
E-mail: jens.teubner@cs.tu-dortmund.de.
Özsu is with the Database Research Group, Cheriton School of Computer Science, University of Waterloo, Canada.
E-mail: tamer.ozsu@uwaterloo.ca.

of the intricacies of modern hardware and the many possibilities available for tuning. To make matters worse, many parameters affect the behavior of 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 common points in terms of the space explored, making it difficult to compare their results.

The first contribution of the paper is algorithmic. We analyze the algorithms proposed in the literature and introduce several important optimizations, which leads to algorithms that are 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 perspective, 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 affirmative in most situations. However, for certain combinations of parameters and architectures, hardware-oblivious approaches have an advantage. As our results show, architectural features such as effective and aggressive on-chip multi-threading on Sparc RISC architecture CPUs favor the hardware-oblivious no partitioning idea. Therefore, heavy SMT might change the picture in the future.

In this paper we extend and expand on the results by Balkesen et al. [6]. Through additional experiments on more recent Sparc RISC processors, we have determined that machines with aggressive SMT benefit *hardware-oblivious* algorithms. This advantage does not happen in all architectures but it is the result of a combination of features (low synchronization cost, fully associative TLBs, aggressive on-chip multi-threading, etc.). These results open up an interesting research direction for future processor architectures that could be tailored to more efficient data processing operators. All previous work, including [6], explored only deployments on single sockets. In this paper, we look into deployments on multi-sockets using the latest x86 multi-core processors (Intel Sandy Bridge). Multi-socket deployment turns out to be a key factor in boosting the performance of join algorithms leading to results that are the fastests published to date. The paper also contains a new analysis of the impact of cache hierarchies and large SMT architectures on the performance of hash joins and why these hardware features do not benefit *hardware-conscious* algorithms. In response to feedback since the publication of [6], we include as well a discussion on how to turn hardware-oblivious algorithms into hardware-conscious ones by using software prefetching and cache alignment.
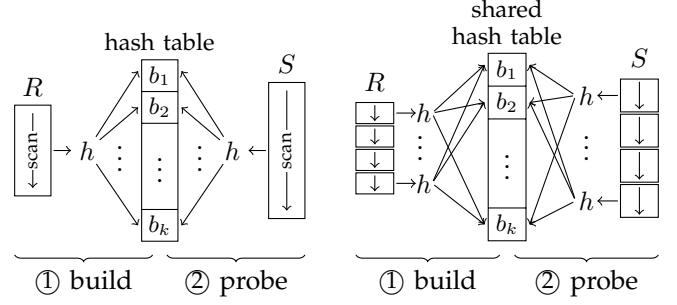

Fig. 1. Canonical hash join.   Fig. 2. *No partitioning* join.

Finally, the paper also explores new optimizations like the use of software-managed buffers.

The code for all the algorithms discussed in this paper plus the data generators used in the experiments are available at http://www.systems.ethz.ch/projects/paralleljoins.

2 BACKGROUND: IN-MEMORY HASH JOINS

2.1 Canonical Hash Join Algorithm

The basis behind any modern hash join implementation is the canonical hash join algorithm [7], [8]. It operates in two phases, as shown in Figure 1. In the *build phase*, the smaller of the two input relations, R, is scanned to populate a hash table with all R tuples. Then the *probe phase* scans the second input relation, 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 is $O(|R| + |S|)$.

2.2 No Partitioning Join

To benefit from modern parallel hardware, Blanas et al. [3] proposed a variant of the canonical algorithm, *no partitioning join*, which is essentially a direct parallel version of the canonical hash join. It does not depend on any hardware-specific parameters and—unlike alternatives that will be discussed 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 accessible to all worker threads.

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
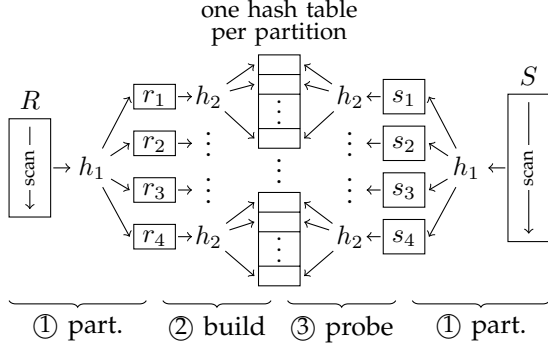
Fig. 3. Partitioned hash join (following Shatdal et al. [9]).

Fig. 4. *Radix join* (as proposed by Manegold et al. [10]).

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 $O\left(\frac{1}{p}(|R| + |S|)\right)$.

2.3 Radix Join

Hardware-conscious, main-memory hash join implementations build upon the findings of Shatdal et al. [9] and Manegold et al. [10], [11]. While the principle of hashing—direct positional access based on a key's hash value—is appealing, the resulting *random access* to memory can lead to 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 [12]. Shatdal et al. [9] 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 blocks reduces cache misses and improves performance. Manegold et al. [10] refined this idea by considering as well the effects of *translation look-aside buffers (TLBs)* during the partitioning phase. This led to *multi-pass partitioning*, now a standard component of the *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 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.
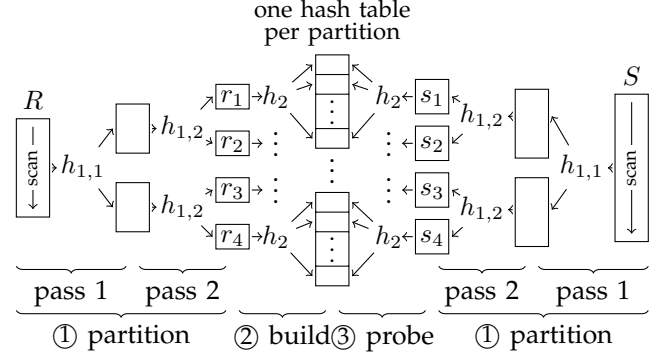
While avoiding cache misses during the build and probe phases, partitioning the input data may cause a different type of cache problem. The partitions will typically reside on different memory pages with a separate entry for *virtual memory mapping* required for each partition. This mapping is cached by TLBs in modern processors. As Manegold et al. [10] 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 efficiently created or accessed *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 in 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 $O\left((|R| + |S|)\log |R|\right)$ for radix join.

Hardware Parameters. Radix join needs to be tuned to a particular architecture 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

TABLE 1
Experimental setup

(a) Workload characteristics

	A (from [3])	**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

(b) Hardware platforms used in our evaluation

	Intel Nehalem	Intel Sandy Bridge	AMD Bulldozer	Sun Niagara 2	Intel Sandy Bridge	Oracle Sparc T4
CPU	Xeon L5520 2.26 GHz	Xeon E5-2680 2.7 GHz	Opteron 6276 2.3 GHz	UltraSPARC T2 1.2 GHz	Xeon E5-4640 2.4 GHz	SPARC T4 3.0 GHz
Cores/Threads	4/8	8/16	16/16	8/64	32/64	32/256
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 -	32 KiB 256 KiB 20 MiB	16 KiB 128 KiB 4 MiB
TLB (L1/L2)	64/512	64/512	32/1024	128/-	64/512	128/-
Memory	24 GiB DDR3 1066 MHz	32 GiB DDR3 1600 MHz	32 GiB DDR3 1333 MHz	16 GiB FBDIMM	512 GiB DDR3 1600 MHz	1 TiB -
VM Page size	4 KiB	4 KiB	4 KiB	8 KiB	2 MiB	4 MiB

CPU cache. Both parameters can be obtained in a rather straightforward way, *e.g.*, with help of benchmark tools such as *Calibrator* [10]. As we shall see later, *radix join* is not overly sensitive to minor mis-configurations of either parameter.

2.4 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 is limited by hardware parameters and is typically small (a few tens of partitions). They are accessed by potentially many execution threads, creating a contention problem (the low-contention assumption of Section 2.2 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]). Overall, we expect a runtime of $O(1/p(|R| + |S|) \log |R|)$ for a parallel radix join on p cores.

3 EXPERIMENTAL SETUP

3.1 Workload

For the comparison, we use machine and workload configurations corresponding to modern database architectures. Thus, we assume a *column-oriented storage model*. We 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

emphasize the effects that we are interested in since they put more pressure on the caching system.[1]

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 1(a), we adopted workloads from Blanas et al. [3] and Kim et al. [1] and refer to them as **A** and **B**, 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.

3.2 Hardware Platforms

We evaluated the algorithms on six different multi-core machines shown in Table 1(b). The Sun UltraSPARC T2 provides eight thread contexts per core where eight threads share the L1 cache with a line size of 16 bytes. All the Intel machines support SMT with two thread contexts per core. The 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. The AMD machine has a different architecture than the others: 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.

Among those platforms, Oracle Sparc T4 [14] and Intel E5-4640 are recent high-end multi-core servers in a four-socket configuration. In the Oracle Sparc T4, each of the sockets provides eight processor cores and each core has hardware support for eight threads. Therefore the machine provides a total of 256 hardware threads. The T4 has 16 KiB L1 and 128 KiB L2 per core with 32-byte cache lines, and a shared 4 MiB L3 with 64-byte cache lines. The memory management unit has a fully associative, 128-entry DTLB and supports 8 KiB, 64 KiB, 4 MiB, 256 MiB and 2 GiB page sizes with a hardware table-walk engine. In comparison to the previous generations of Sparc

---

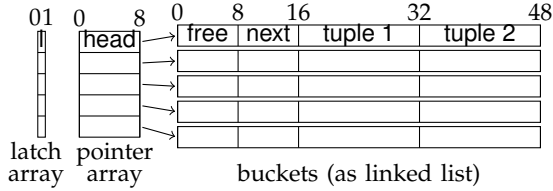1. The effect of tuple widths was studied, *e.g.*, by Manegold et al. [13].

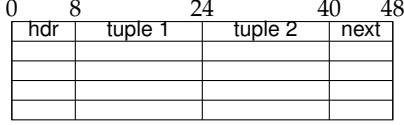Fig. 5. Original hash table implementation in [3].

Fig. 6. Our hash table implementation.

Fig. 7. Cycles per output tuple for hardware-oblivious *no partitioning* strategy (Workload **A**; Intel Xeon L5520, 2.26 GHz).

(such as T2), T4 provides many improvements such as an aggressive clock frequency of 3.0 GHz, advanced branch prediction, out-of-order execution, and a shared L3 cache. Our system runs Solaris 11.1 (SunOS 5.11). The Intel E5-4640 has a total memory of 512 GiB and a clock frequency of 2.4 GHz; it runs Debian Linux 7.0, kernel version 3.4.4-U5 and uses 2 MiB VM pages for memory allocations transparently.

The Intel L5520, E5-2680 and AMD systems run Ubuntu Linux (kernel version 2.6.32); the Sun Ultra-SPARC T2 runs 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, neither qualitatively nor quantitatively. For the performance counter profiles that we report, we instrumented our code with the Intel Performance Counter Monitor [15].

4 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 employed in earlier work, namely a Nehalem L5520 (cf. Table 1(b)).

4.1 Build Cost

Sorted input might have a positive impact on the build cost. Using a hash function such as modulo together with a sorted input, one can observe strictly sequential memory accesses, in addition to the reduced contention for the bucket latches. For the impact of using sorted input on the build cost, we refer the reader to [6]. In the rest of the paper, we assume a randomly permuted input.

4.2 Cache Efficiency

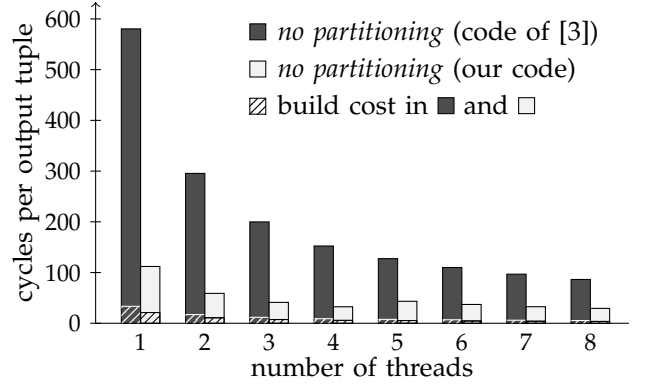When running the *no partitioning* code of [3], the cache profile information indicates hash table build-up incurs a very high number of cache and TLB misses. Processing 16 million tuples results in 45.3/52.7 million L3/TLB misses, or about three misses per input tuple.

The reason for this inefficiency becomes clear as we look at the code of Blanas et al. [3]. The hash table in this code is implemented as illustrated in Figure 5. The hash table 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 which is 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 is likely to result in a cache miss.

Optimized Hash Table Implementation. To improve the cache efficiency of *no partitioning*, 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 6. The hash function directly indexes into this array representation. For overflow buckets, 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 [3], for Workload **A**, we configured our hash table to two 16-byte tuples per bucket. An 8-byte next pointer is used to chain hash buckets in case of overflows.

In terms of absolute join performance, our re-written code is roughly three times faster than the code of Blanas et al. [3], as shown in Figure 7. Yet, our code remains
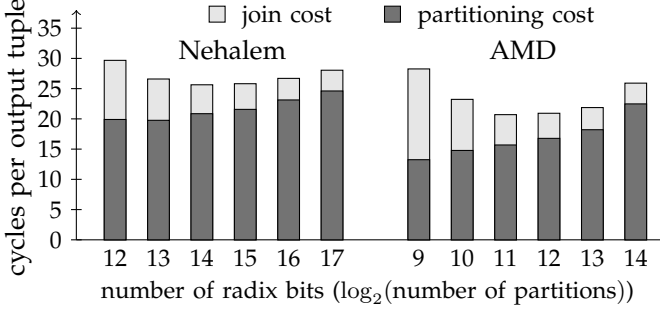
Fig. 8. Cost vs. radix bits (Workload **B**; Nehalem: 2-passes; AMD: 1-pass).

strictly hardware-oblivious: no hardware-specific parameters are needed to tune the code.

5 HARDWARE-CONSCIOUS JOINS

5.1 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 8 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 parameter mis-configuration: within a relatively broad range of configurations, the performance of *radix join* degrades only marginally.

5.2 Hash Tables in Radix Join

Various hash table implementations have been proposed for radix join. Manegold et al. [10] use a 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). Based on our analysis, *bucket chaining* mechanism turns out to be superior to other approaches, even those utilizing SIMD [6]. Hence, we use it for all the following experiments.

5.3 Overall Execution Time

The overall cost of join execution consists of the cost for data partitioning and the cost of 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.

TABLE 2
CPU performance counter profiles for different radix join implementations (in millions); Workload **A**

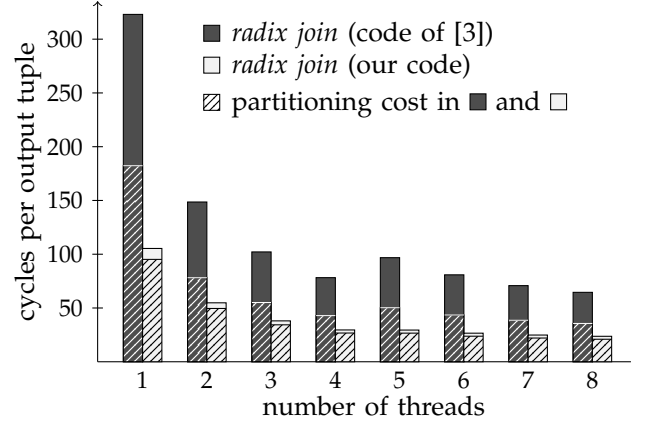|	code from [3]			our code		
	Part.	Build	Probe	Part.	Build	Probe
Cycles	9398	499	7204	5614	171	542
Instructions	33520	2000	30811	17506	249	5650
L2 misses	24	16	453	13	0.3	2
L3 misses	5	5	40	7	0.2	1
TLB load misses	9	0.3	2	13	0.1	1
TLB store misses	325	0	0	170	0	0

Fig. 9. Overall join execution cost (cycles per output tuple) for hardware-conscious *radix join* strategy (Workload **A**; Intel Xeon L5520, 2.26 GHz).

We had two implementations of *radix join* available. For the code of Blanas et al. [16], we found one pass and 2,048 partitions to be the optimal parameter configuration (matching the configuration in their experiments [3]). 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 the most efficient configuration. Table 2 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 the Blanas et al.'s code [3] needs to perform one pass.

The resulting overall execution times are reported (as cycles per output tuples) in Figure 9. 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 the optimized code, partitioning becomes the dominant cost, which is consistent with the findings of Kim et al. [1], which showed comparable cost at similar parameter settings. Overall, our code is about three times faster than the code of Blanas et al. for all tested configurations.

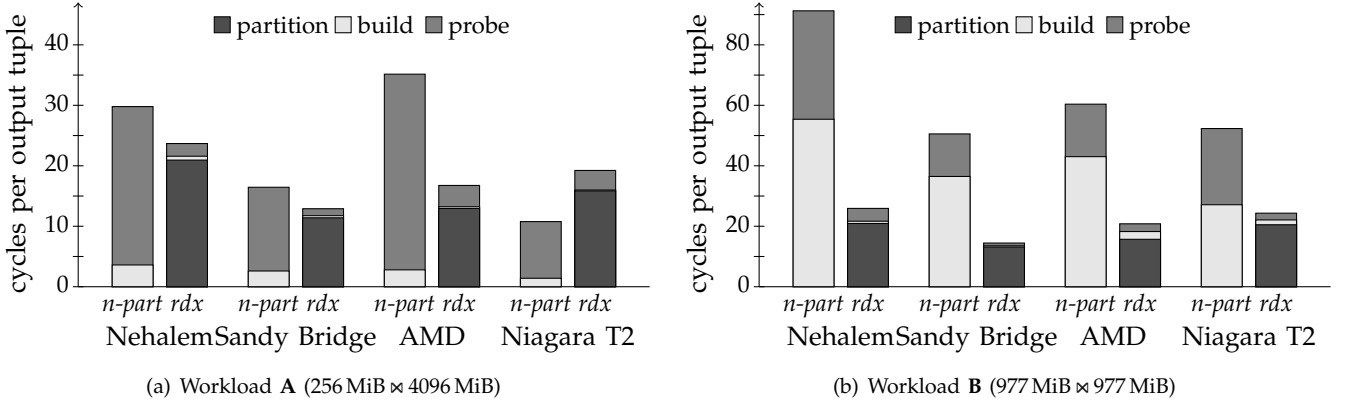(a) Workload **A** (256 MiB ⋈ 4096 MiB) (b) Workload **B** (977 MiB ⋈ 977 MiB)

Fig. 10. 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.

Performance Counters. We have also instrumented the available *radix join* implementations to monitor CPU performance counters. Table 2 lists cache and TLB miss counts for the three tasks in *radix join*. The table shows a significant difference in the number of cache and TLB misses between the implementations. The idea behind *radix join* is that all partitions should be sufficiently small to fully fit into caches. One should expect a very low number of misses. This is true for our implementation but not for the one of Blanas et al.

The reason for the difference is the 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, the created hash tables will long be evicted from CPU caches before their content is actually needed for probing. Our code avoids these memory round-trips by running the build and probe phases for each partition together.

6 HARDWARE-CONSCIOUS OR NOT?

6.1 Effect of Workloads

Figure 10 summarizes the performance results we observed for both workloads on our four single-socket architectures.

Focusing first on Workload **A**, Figure 10(a) shows the performance of our own implementation. For the same workload, Blanas et al. [3] reported only a marginal performance difference between *no partitioning* and *radix join* on x86 architectures. In our results the hardware-conscious *radix join* is appreciably faster when both implementations are equally optimized. Only on the Sun Niagara the situation looks different. We will look into this architecture later in the paper.

The results in Figure 10(a) may still be seen as a good argument for the hardware-oblivious approach. An approximate 25 % performance advantage, *e.g.*, on the

two Intel platforms, might not justify the effort needed for parameter tuning in *radix join*.

However, running the same experiments with our second workload, Workload **B** (Figure 10(b)), 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* has comparable performance to *radix join* only when the relative relation sizes are very different. This is because in such a situation, the cost of the build phase is minimized. As soon as table sizes grow and become similar, the overhead of not being hardware-conscious becomes visible (see the differences in the build phases for *no partitioning*).

6.2 Scalability

To study the scalability of the two join variants, we re-ran the experiments with a varying number of threads, up to the maximum number of hardware contexts available on each of the architectures. Figure 11 illustrates the results.

All platforms and both join implementations show good scalability. Thanks to this scalability, the optimized *radix join* implementation reaches a throughput of 196 million tuples per second. As far as we are aware, this is the highest throughput for in-memory hash joins on a single CPU socket 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 [17]. 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. Note that NUMA would create significant problems for the shared hash table used in *no partitioning* (let alone in future designs where memory may not be coherent across the machine).
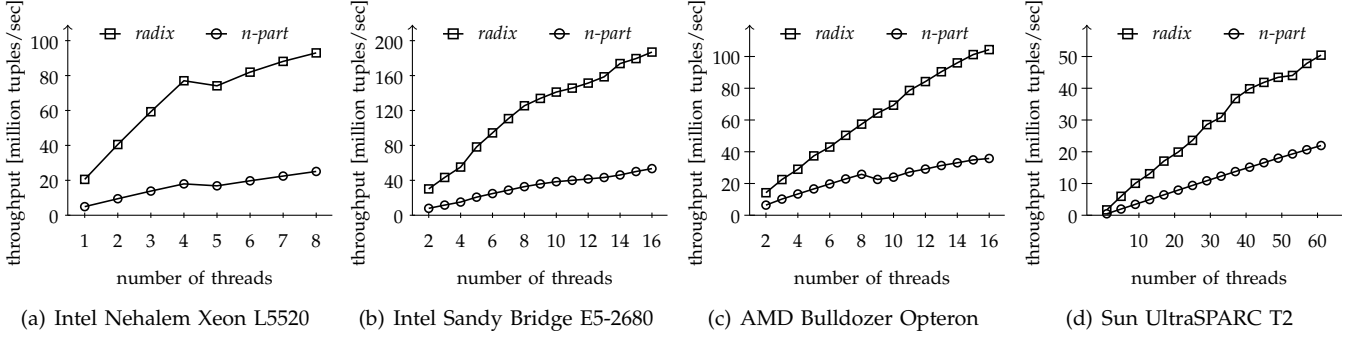
Fig. 11. Throughput comparison of algorithms on different machines using Workload **B**. Computed as input-size/execution-time where input-size $= |R| = |S|$.

TABLE 3
Latch cost per build tuple in different machines

	Nehalem	Sandy Bridge	Bulldozer	Niagara 2
Used instruction	xchgb	xchgb	xchgb	ldstub
Reported instruction latency in [18], [19]	∼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

A similar bump also exists in the Intel machines which is mainly due to the SMT effects on the Intel architectures. We will discuss that later in the paper

6.3 Sun UltraSPARC T2 "Niagara"

On the Sun UltraSPARC T2, a totally different architecture than the x86 platforms, we see a result similar to the other architectures with Workload **B**. Hardware-conscious *radix join* achieves a throughput of 50 million tuples per second (cf. Figure 11(d)), whereas *no partitioning* achieves only 22 million tuples per second.

When looking to Workload **A**, *no partitioning* becomes faster than *radix join* on the Niagara 2 (shown in Figure 10(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 and the TLB is fully associative, which are significant differences to other architectures.

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 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 3. These special characteristics also show the importance of architecture-sensitive decisions in algorithm implementations.

6.4 TLB, VM Page Size and Load Balancing Effects

The virtual memory setup of modern systems is, to a small extent, configurable. By using a larger system *page size*, fewer TLB entries might be needed for the operations on a given memory region. In general both join strategies can equally benefit from large pages, leaving the *"hardware-conscious or not?"* question unchanged. Large pages may have a more significant effect on the performance of *no partitioning*–but only if their use is combined with hardware-conscious optimizations such as explicit data prefetching (cf. Section 6.8). For reasons of space, we omit the detailed experiments and analysis here and refer the reader to a technical report with more experiments [20].

In contrast to *no partitioning* join, *radix* join is vulnerable to load-imbalance if tasks are not properly scheduled over worker threads. However, *radix* join can be made robust to load-imbalance of the barrier synchronization by using the *fine-granular task decomposition* strategy that we described earlier [6].

Additionally, higher skew on the foreign key relation might improve the performance of the *no partitioning* join significantly due to increasing cache hits. However, this makes *no partitioning* faster than *radix* only under special cases. The details and the experiments covering load balance issues and data skew are included in [6].

### 6.5 Effect of Relation Size Ratio

The above experiments show that the relative sizes of the joined tables 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 fix the number of threads at 8. We vary 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 change the size of R, we also adjust the distribution of values in S accordingly.

Figure 12 shows the cycles per output tuple for each phase as well as the entire run for different sizes of R in a log-log plot.

The results confirm the observations made so far and provide a clearer answer to the controversy between hardware-conscious and hardware-oblivious algorithms. *No partitioning* does well when the build relation is
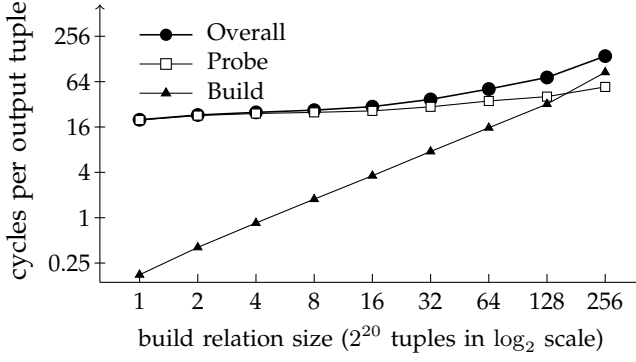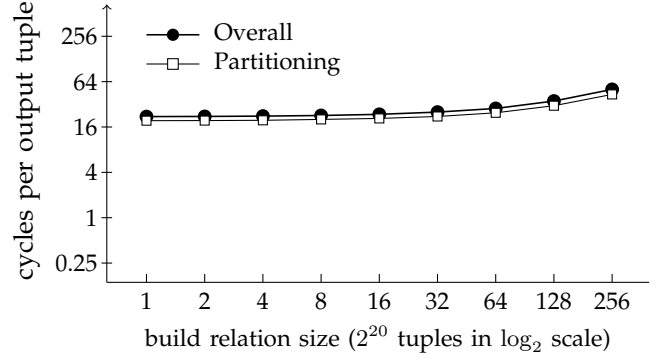
(a) *No partitioning* join   (b) *Radix join*

Fig. 12. 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).
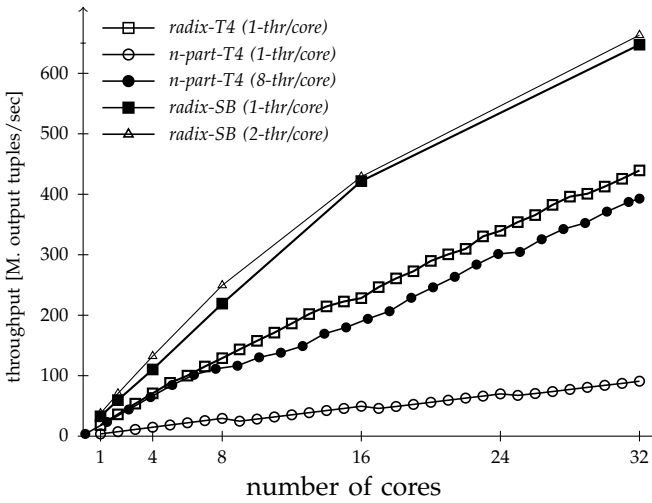

Fig. 13. Performance on recent multi-core servers, Sparc T4 and Sandy Bridge (SB) using Workload **B**. Throughput is in output tuples per second, *i.e.* $|S|/\text{execution time}$.

very small compared to the large relation. Performance deteriorates as the size of R increases because of the cost of the build phase (Figure 12(a)). *Radix join* is 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 slightly better than *radix join* only under skew and when the sizes of the tables being joined significantly differ. In all other cases, *radix join* is better (and significantly better in fact) in addition to also being more robust to parameters like skew or relative table sizes.

6.6 Evaluation on Recent Multi-Core Servers

Figure 13 shows the performance of different algorithms on Sparc T4 and Sandy Bridge. First, the *no partitioning* algorithm draws its true benefit from effective on-chip

multi-threading on T4. When eight threads are executed on a single physical core, the underlying system takes care of problems such as cache and TLB misses. The performance comparison of running a single thread per core (cf. -○-) vs. eight threads per core (cf. -●-) clearly demonstrates this phenomenon with a speedup factor of more than four. On the other hand, *radix* join also scales linearly while using all the physical cores. However, it does not benefit from hardware multi-threading. We will look into the issue of why hardware-conscious algorithms are not well suited to this architecture in detail in the next sub-section. Overall, *no partitioning* clearly becomes a competitive approach to hardware-conscious *radix* on this platform.

Is on-chip multi-threading or SMT always beneficial?

Simultaneous multi-threading (SMT) hardware 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.

The results on T4 in Figure 13 highly suggest that hardware-oblivious algorithms benefit from SMT. However, the question is whether this is a particular feature of T4 (such as aggressive SMT) or this can be generalized to other hardware. In order to answer this question, we conducted experiments on the Nehalem system which contains four cores with two hardware contexts each (experiments with other Intel SMT hardware produced similar results). In the experiment, we start by assigning threads to different physical cores. Once the physical cores are exhausted, we assign threads to the available hardware context in a round-robin fashion.

Figure 14 illustrates the performance of *no partitioning* relative to the performance of a single-threaded execution of the same algorithm ("speedup"). Our experiment indeed confirms the scalability with SMT threads on the un-optimized code of [3]. However, once we run the same experiment with our optimized code (with significantly better absolute performance, cf. Figure 7),
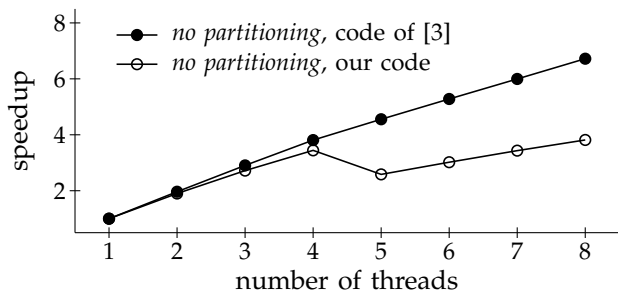
Fig. 14. Speedup of *no partitioning* algorithm on Intel SMT hardware. First four threads are "native"; threads 5–8 are "hyper" threads (Workload **A**; Xeon L5520).

Fig. 15. Speedup of *radix* algorithm on Intel SMT hardware. First four threads are "native"; threads 5–8 are "hyper" threads (Workload **A**; Xeon L5520).

SMT does not help the *no partitioning* strategy at all.

As the result shows, in this particular Intel hardware 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 a negligible benefit.

Similarly, Figure 15 shows that neither of the two *radix join* implementations can significantly 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. [3]. 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.

Overall, results in Figure 13, 14 and 15 indicate that aggressive SMT can make hardware-oblivious algorithms competitive but this only occurs in particular hardware and cannot be generalized.

Impact of architecture aware compilation

Another interesting fact on the T4 is the compiler optimizations. Using Sun Studio compiler 12.3 with compilation options "`-xO4 -xipo -fast -xtarget=T4 -Bdynamic`" resulted in ≈ 30 % performance improvement over `gcc` 4.7.1. with options "`-mtune=niagara4 -mcpu=niagara4 -O3`". This fact highlights the benefit of architecture aware compilation optimizations.

Multi-socket deployment and scalability

Looking at the performance of *radix* join on the Intel Sandy Bridge machine in Figure 13 (cf. ■), we observe that x86 retains a significant performance advantage over the Sparc architecture despite the radical architectural changes and improved single thread performance on T4. The performance gain of *radix* from two-way SMT on Intel remains limited (cf. ▲). Meanwhile, the performance of *radix* is still superior to that of *no partitioning* on Intel almost by a factor of five, showing the importance of hardware-consciousness on this architecture (not shown in the graph). Finally, this experiment also demonstrates
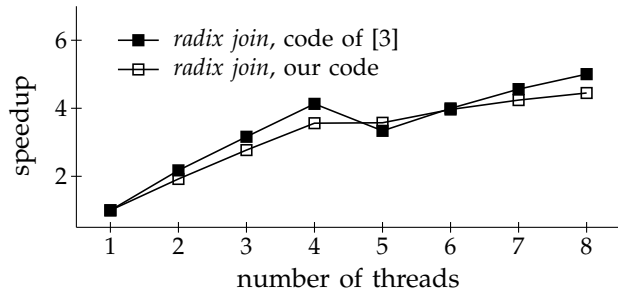
the scalability of hash join algorithms beyond a single CPU socket and exhibits the fastest performance of a radix hash join implementation to date (≈ 400-650 M/s).

6.7 When Does Hardware-Conscious Not Work Well?

Some of the architectural features have an important impact on the performance of the algorithms. In this section, we shed a light on such features and highlight what makes a hardware-conscious algorithm to misbehave.

Less is more for hardware-conscious algorithms

In dynamically multi-threaded architectures, most of the physical core resources are shared among all active threads. The Sparc T4 architecture takes this to an extreme where resources such as caches, load/store buffer entries, DTLB, hardware table-walk engine, and branch prediction structures are dynamically shared between all the eight threads running on a core. This usually works well for applications with a mix of high and low IPC threads running concurrently and provides a better utilization of system resources. However, in case of hardware-conscious algorithms such as *radix* join, this introduces detrimental effects to performance.

The performance of *radix* join with different number of threads is shown on the left side of Figure 16. In the case with 32 threads, each thread runs on an individual physical core and it does not contend with other threads for resources. In case of 256 threads, each physical core runs eight threads concurrently where all the core resources are shared. Surprisingly, using more threads ruins the hardware-consciousness of *radix* join making it slower than the 32-thread case. The performance problem can be clearly seen in the actual join phase of the algorithm, which is the cache-sensitive, high IPC phase of the algorithm. The eight threads contend for the same L1/L2 space and due to excessive cache misses this phase slows down. Essentially, the cache-conscious nature of *radix* join disappears.

Incongruent cache architecture

The cache architecture plays an important role for hardware-conscious algorithms. In the Sparc T4, the
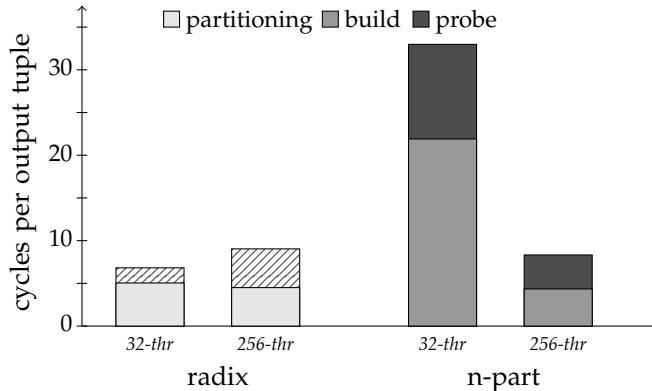
Fig. 16. Impact of number of threads on Sparc T4 on overall performance for different algorithms. Using Workload **B** (977 MiB ⋈ 977 MiB); Oracle Sparc T4.

16 KiB L1 cache is relatively small in comparison to x86 architectures. First, the small cache size requires a higher partitioning fan-out which makes the partitioning costlier in this architecture. Second, the maximum partitioning fan-out is also limited by the L1 size. Due to the use of a fully-associative 128-way DTLB and large pages, TLB misses are no longer the main bottleneck in partitioning. However, the L1 cache can hold 512 cache lines and puts a hard limit at around 512 for efficient partitioning fan-out per pass. Moreover, the 4-way associativity of the L1 cache does not match the number of eight hardware threads. Therefore, accesses from different threads can potentially cause unnecessary conflict misses. In addition, the 32 bytes L1/L2 cache line size reduces the potential benefit from sequential writes during partitioning compared to a cache line size of 64 bytes in x86 architecture. Last but not least, the relatively small shared L3 brings almost no benefit to a hardware-conscious algorithm such as *radix* join as the effective L3 share per thread ($\approx$ 64 KiB) is less than the size of the L2 cache. Overall, all these characteristics of the cache architecture in Sparc T4 makes the *radix* join ill-behaved in this architecture.

6.8 Making No Partitioning Hardware-Conscious

In this section, we propose hardware-conscious optimizations such as a combination of *cache alignment*, *software-prefetching*, and use of *large pages* to improve the performance of *no partitioning* join. These techniques abandon the strictly *hardware-oblivious* nature of the original algorithm but require little parameter tuning.

Cache Alignment

As discussed in Section 4.2, the *no partitioning* implementation of Blanas et al. [3] uses a hash table design where up to three accesses to different memory locations are needed to access a single hash bucket (latch array, pointer array, and actual data buckets; cf. Figure 5).

TABLE 4
No partitioning join; cache misses per tuple (original code of Blanas et al. [3] vs. our own implementation).

	Code of [3]		Our code		Our code (cache-aligned)	
	Build	Probe	Build	Probe	Build	Probe
L2 misses	2.97	2.94	1.56	1.39	1.01	1.00
L3 misses	2.72	2.65	1.56	1.36	1.00	0.99

To avoid this potential memory access bottleneck, in our own code we wrapped the necessary latches into the bucket data structure and removed the indirection caused by the pointer array of Blanas et al. In effect, only a single record needs to be accessed per data tuple. Only true hash collisions will require extra bucket fetches.

We measured the number of cache misses required per build/probe tuple in both implementations (cf. Table 4). Somewhat counter-intuitively, the number of misses per tuple is considerably higher, however. This is most noticeable during the build phase of our own implementation, where we see more than 1.5 misses/tuple even though only a single hash bucket must be accessed per tuple.

The reasons for this is the lack of *cache alignment* of both hash table implementations. As illustrated in Figures 5 and 6, both hash table implementations use a bucket size of 48 bytes. If such buckets are packed one after another, a single bucket access may span *two* cache lines and thus cause more than a single cache miss on access. Specifically, four 48-byte buckets will occupy three successive cache lines. On average, each bucket intersects with 1.5 cache lines, which coincides with the cache miss numbers shown in Table 4.

Hash buckets can be forced to stay within a cache line by *aligning* them to 64-byte boundaries. As the last two columns in Table 4 show, changing *no partitioning* in this way reduces the cache miss rate to the expected one miss per tuple.

Software Prefetching

Another way to avoid cache misses is the use of *prefetching*. Chen et al. [21], for instance, described how hash table accesses can be accelerated by issuing *software prefetch instructions*. If those instructions are issued early enough, the CPU can overlap memory accesses with instruction execution and thus hide memory access latencies.

We applied the prefetching mechanisms of Chen et al. to our *no partitioning* implementation. The proper *prefetch distance* is a hardware-specific parameter, which we manually tuned to the behavior of our machine. The effect of this optimization is illustrated in Figure 17 for Workloads **A** and **B** (bars labeled "w/ prefetch").

Figure 17 also illustrates the effect of cache alignment (bars labeled "aligned"). Interestingly, cache alignment alone does not significantly improve the performance of *no partitioning*. Both optimizations together, however,
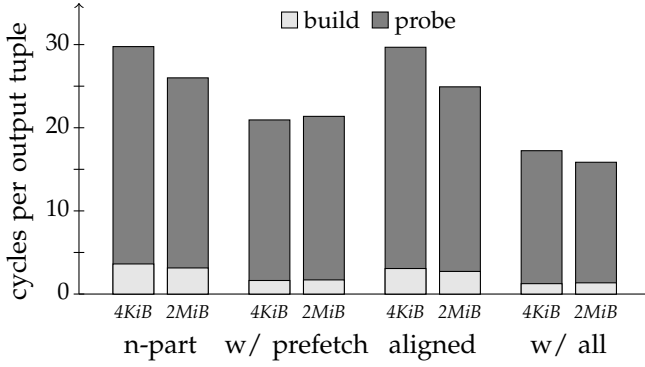
Fig. 17. Impact of different optimizations on cycles per output tuple for *no partitioning* using Workload **A** (256 MiB ⋈ 4096 MiB); 8 threads, Intel Nehalem L5520.

Fig. 18. Impact of different optimizations on cycles per output tuple for *no partitioning* using Workload **A** (256 MiB ⋈ 4096 MiB); 16 threads, AMD Bulldozer Opteron 6276.

can improve the throughput of *no partitioning* by more than 40 %. To achieve this improvement, however, we had to give up the strictly hardware-oblivious nature of *no partitioning* and introduce tuning parameters such as prefetch distance and cache line size.

Figure 18, in fact, illustrates how sensitive our code changes are to the underlying hardware platform. When running the same experiment on the AMD Opteron machine, aligning hash buckets to the cache line size has a significant impact on overall throughput. Software prefetching can improve only little over that. Together, both optimizations again yield a performance gain of $\approx 40\%$ over the baseline implementation.

6.9 Improving Radix with Software-Managed Buffers

Conceptually, each partitioning phase of *radix join* takes all input tuples one-by-one and writes them to their corresponding destination partition (pos[·] keeps track of the current write location within each partition):

```
1 foreach input tuple t do
2     k ← hash(t);
3     p[k][pos[k]] = t;        // copy t to target partition k
4     pos[k]++;
```

Generally, partitions are far apart and on separate VM pages. Thus, if the *fanout* of a partitioning stage is larger than the number of TLB entries in the system, copying each input tuple will cause another TLB miss. Typically, the number of TLB entries is considered an upper bound on the partitioning fanout that can be efficiently realized.

This TLB miss count can be reduced, however, when writes are first *buffered* inside the cache. The idea is to allocate a set of buffers, one for each output partition and each with room for up to N input tuples. Buffers are copied to their final destination only when they are full:
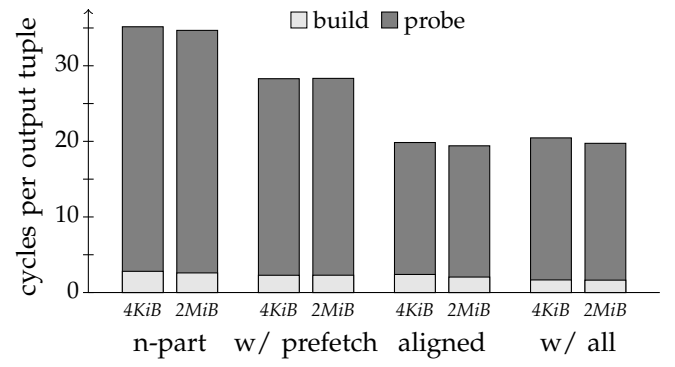
```
1 foreach input tuple t do
2     k ← hash(t);
3     buf[k][pos[k] mod N] = t;     // copy t to buffer
4     pos[k]++;
5     if pos[k] mod N = 0 then
6         copy buf[k] to p[k]; // copy buffer to partition k
```

Obviously, buffering leads to additional copy overhead. However, for sufficiently small N, all buffers will fit into a single memory page. Thus, a single TLB entry will suffice unless a buffer becomes full and the code enters the copying routine in line 6. Beyond the TLB entry for the buffer page, an address translation is required only for every Nth input tuple, significantly reducing the pressure on the TLB system. As soon as TLB misses become infrequent, it is likely the CPU can hide their latency with its usual out-of-order execution mechanisms.

The buffering strategy mentioned above follows the idea of Satish et al. [22], which employed the same technique to reduce the TLB pressure of *radix sort*.

We added an implementation of such software-managed buffers to our *radix join* code and configured N such that one buffer will exactly fill one cache line (64 bytes); *i.e.*, $N = 4$ for Workload **A** and $N = 8$ for Workload **B**. Configuring the buffer size in this manner allows for another low-level optimization. Since we are now always writing a full cache line at once to global memory, the CPU can take advantage of its *write combining* facilities, thus avoiding to read the cache line before writing it back.

Figure 19 illustrates the effect of software-managed buffers on the performance of partitioning. In both figures, we partition a 128 million-tuple data set with 8 bytes per tuple (Workload **B**) and measure the achievable throughput for single-pass radix partitioning with and without software-managed buffers.

As can be seen in the figure, software-managed buffers indeed cause some copying overhead. But the investment clearly pays off once the available TLB entries are exhausted. At about 8 radix bits (Figure 19(a)) the
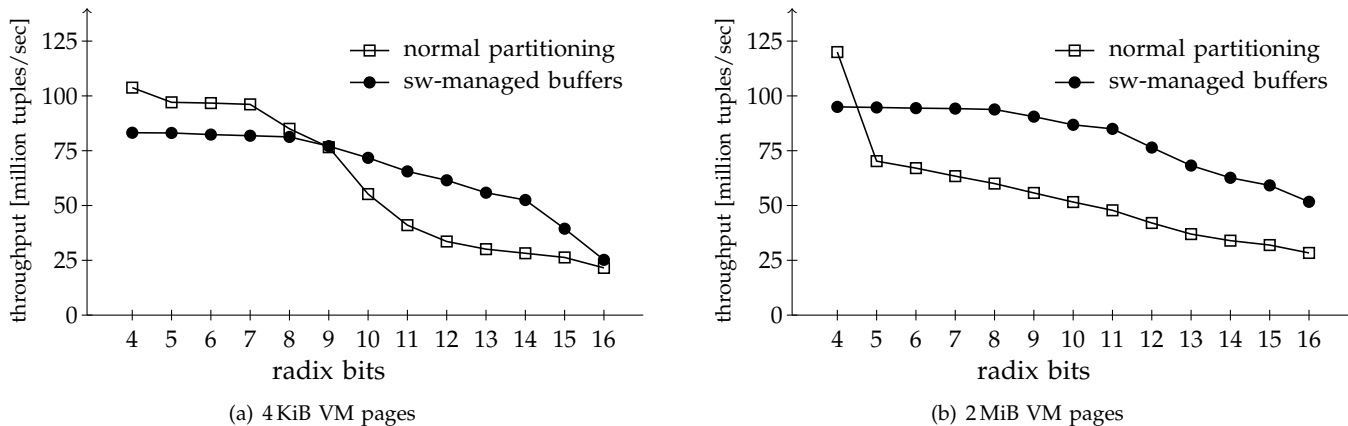
(a) 4 KiB VM pages

(b) 2 MiB VM pages

Fig. 19. Partitioning performance comparison when using 4 KiB and 2 MiB pages (Using a single core on Intel Xeon L5520, 2.26 GHz).

performance of the naïve strategy begins to suffer from the growing TLB miss cost,[2] whereas the implementation with software-managed buffers handles the growing fanout much more gracefully. Essentially, software-managed buffers shift the TLB exhaustion problem to the configurations beyond 14 radix bits, where TLB entries are not even sufficient to hold the "cache-local" buffer.

The effect is even more pronounced when we configure our system to use a 2 MiB page size (cf. Figure 19(b)). With now only 32 TLB entries available, conventional radix partitioning seriously suffers from TLB misses even for five radix bits (*e.g.*, 32 partitions), while software-managed buffers can keep partitioning speed almost constant even for very large fanouts.

In practice, the advantage of software-managed buffers is two-fold: *(i)* for many situations, software-managed buffers offer better absolute performance, since fewer passes can usually achieve the same overall fanout; *(ii)* the optimization is very robust toward the configured number of radix bits, hence, it reduces the potential damage of ill-chosen algorithm parameters.

7 RELATED WORK

Following the insights on the importance of memory and caching effects on modern computing hardware by Manegold et al. [11] and Ailamaki et al. [12], new algorithm variants have emerged to run database operators efficiently on modern hardware.

One of the design techniques to achieve this goal is the use of *partitioning*, which we discuss extensively in this paper. Besides its use for in-memory joins, partitioning is also relevant for *aggregation*, as recently investigated by Ye et al. [23]. While the aggregation problem differs from a join in many ways, the observations made by Ye et al. about different hardware architectures are consistent with ours.

While we mainly looked at local caching and memory latency effects, we earlier demonstrated how the

2. Note that the 64-entry TLB1 is assisted by a 512-entry TLB2.

topology of modern NUMA systems may add additional complexity to the join problem [24]. *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. [4] 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. [25]. 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.

He and Luo [26] addressed the "hardware-conscious or not?" question on a slightly higher system level. They evaluated a full query engine (EaseDB), which was built on *cache-oblivious* design principles [27]. Their conclusion was that cache-oblivious strategies can achieve comparable performance, but only when those algorithms are very carefully and sophisticatedly designed.

8 CONCLUSION

In this paper we revisit the existing results in the literature regarding the design of main-memory, hash join algorithms on multi-core architectures. Through an extensive experimental evaluation, we trace back the observed differences in performance of existing algorithms to a wide range of causes: workload, caching effects, and multi-threading. In doing so, we provide optimizations to all existing algorithms that significantly increase their performance. The results in the latest hardware available to us indicate that *hardware-conscious* algorithms maintain an edge over *hardware-oblivious* in most systems and configurations. However, there are specific cases where features of modern hardware such aggressive on-chip multi-threading make *hardware-oblivious* algorithms competitive, thereby establishing an interesting architectural option for future systems.

ACKNOWLEDGMENTS

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] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper, "Massively parallel numa-aware hash joins," in *IMDM*, 2013.

[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*, 2011, pp. 37–48.

[4] 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.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *PVLDB*, 2013.

[6] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *ICDE*, 2013.

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

[8] 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.

[9] A. Shatdal, C. Kant, and J. F. Naughton, "Cache conscious algorithms for relational query processing," in *VLDB*, 1994, pp. 510–521.

[10] 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.

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

[12] 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.

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

[14] M. Shah, R. T. Golla, G. Grohoski, P. J. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. A. Ziaja, "Sparc T4: A Dynamically Threaded Server-on-a-Chip," *IEEE Micro*, 2012.

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

[16] 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.

[17] 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, Mar. 2010.

[18] 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.

[19] Sun, "UltraSPARC T2™ supplement to the UltraSPARC architecture 2007," http://sosc-dr.sun.com/processors/UltraSPARC-T2/docs/UST2-UASuppl-HP-ext.pdf, online, accessed July 2012.

[20] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," ETH Zurich, Systems Group, Tech. Rep., Nov. 2012.

[21] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving hash join performance through prefetching," *ACM Trans. Database Syst.*, vol. 32, no. 3, Aug. 2007.

[22] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort," in *SIGMOD*, 2010, pp. 351–362.

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

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

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

[26] B. He and Q. Luo, "Cache-oblivious databases: Limitations and opportunities," *ACM TODS*, vol. 33, no. 2, Jun. 2008.

[27] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS*, New York, NY, USA, Oct. 1999, pp. 285–298.

Çağrı Balkesen is a fourth-year PhD student in the Systems Group at ETH Zurich. His research topics are in-memory database processing on modern computing architectures as well as data stream processing. He holds a MSc in Computer Science of ETH Zurich and a BSc in Computer Engineering of the Middle East Technical University (METU) in Turkey.
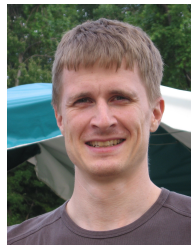
Jens Teubner is leading the Databases and Information Systems Group at TU Dortmund in Germany. His main research interest is data processing on modern hardware platforms, including FPGAs, multi-core processors, and hardware-accelerated networks. Previously, Jens Teubner was a postdoctoral researcher at ETH Zurich (2008–2013) and IBM Research (2007–2008). He holds a PhD in Computer Science from TU München (Munich, Germany) and an M.S. degree in Physics from the University of Konstanz in Germany.

Gustavo Alonso is a professor at the Department of Computer Science of ETH Zurich. At ETH, he is a member of the Systems Group (www.systems.ethz.ch). His areas of interest include distributed systems, data management, enterprise architecture, middleware, and system design. He is a fellow of the ACM and IEEE. For more information, see: www.inf.ethz.ch/~alonso.

M. Tamer Özsu is a professor of computer science and Associate Dean (Research) of Faculty of Mathematics at the University of Waterloo, Cheriton School of Computer Science. His current research focuses on large scale data distribution and management of unconventional data. He is a fellow of the ACM and IEEE, an elected member of Turkish Academy of Science, and a member of Sigma Xi.