Micro Partitioning: Friendly to the Hardware *and* the Developer

Jan Mühlig
jan.muehlig@tu-dortmund.de
DBIS Group, TU Dortmund University

Jens Teubner
jens.teubner@cs.tu-dortmund.de
DBIS Group, TU Dortmund University
Lamarr Institute for Machine Learning and Artificial Intelligence

ABSTRACT

Modern hardware's complexity has made studying hardware-conscious algorithms a relevant topic for many years. Partitioning algorithms, for instance, break data into bits that fit into fast CPU caches. Unfortunately, they are often challenging to design, develop, and maintain. While hardware-oblivious algorithms are easier to build, they may perform poorly when hardware or data deviate from expectations.

In this paper, we introduce micro partitioning, which enhances the partitioning problem in a way that outperforms state-of-the-art solutions while being hardware-agnostic. By storing tuples in a tight address space, micro partitioning creates an access pattern that is friendly to both caches and translation lookaside buffers. We also show how micro partitioning interacts with task-based execution strategies in a symbiotic way, making micro partitioning intuitive to express for developers.

ACM Reference Format:
Jan Mühlig and Jens Teubner. 2023. Micro Partitioning: Friendly to the Hardware *and* the Developer. In *19th International Workshop on Data Management on New Hardware (DaMoN '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3592980.3595310

1 INTRODUCTION

The complexity of modern computing hardware has lead to the design of *hardware-conscious algorithms* many years ago. A prime example are *partitioning strategies*, which break up data into pieces small enough so they can fit into fast CPU caches (e.g., to prepare for a *partitioned hash join*). The downside of such algorithms is that they are often intricate to design, develop, and maintain. This has resulted in a literature debate as to whether *hardware-oblivious algorithms* may be the better choice: They are easier to design, do not depend on well-set parameters, and may be more robust when data or hardware behave differently than expected [3, 5, 11].

In this work, we aim to eat the cake and have it, too. Following up on existing work, we first devise *micro partitioning* to solve the partitioning problem in a way that outperforms state-of-the-art solutions such as *radix partitioning* [3, 9, 15, 21, 24, 27]. While those existing solutions are limited either by the available number of entries in the system's Translation Lookaside Buffers (TLBs) or by redundant data copying, micro partitioning ensures a data access pattern that is friendly to both caches and TLBs.

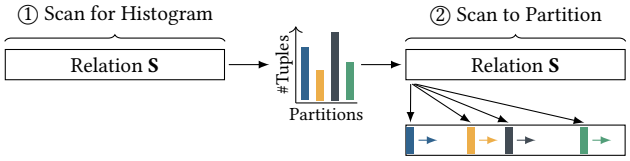


Figure 1: Data partitioning using a histogram.

Micro partitioning is also developer-friendly and agnostic to the hardware substrate. As a second contribution of this work, we show how the strategy interacts with *task-based execution strategies* [8, 13, 16, 20, 26] in a symbiotic way. We illustrate this by the example of our `MxTasking` framework [16]. Its *annotation* mechanism makes micro partitioning intuitive (and hardware-oblivious) to express for developers. At the same time, it enables the framework to run the code in a hardware-conscious way. In fact, `MxTasking` is able to even dynamically adapt to the system state and to resource availability at execution time.

We present micro partitioning as follows. Section 2 recaps the partitioning problem and introduces micro partitioning. Section 3 shows how micro partitioning works hand-in-hand with task-based execution. We evaluate both aspects experimentally as we go in Sections 2.3 and 4 before we summarize in Section 5.

2 DATA PARTITIONING

With memory accesses becoming the bottleneck of data-intensive systems, applications must use cached data to provide optimal performance. This is particularly challenging for hash tables because of their random and hard-to-predict access patterns.

2.1 Hash-Based Partitioning

One way to make hash tables cache-aware is to divide the data into smaller partitions such that each partition's hash table fits into a core's private cache [3, 7, 9, 15, 25]. A common (and simple) way to implement such hash-based partitioning is to scan the data twice: Once to establish a histogram and set up a contiguous memory region for each of the partitions and a second time to move data to their proper partition. Figure 1 illustrates this approach.

However, the appropriate number of partitions must be chosen carefully. On the one hand, the data must be divided into a sufficient number of partitions such that each partition results in a cache-sized hash table. On the other hand, too many partitions lead to several logical memory addresses that exceed the TLB's capacity—which defines an upper limit on the number of partitions that can be written simultaneously [3, 7].

DaMoN '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0191-7/23/06.
https://doi.org/10.1145/3592980.3595310

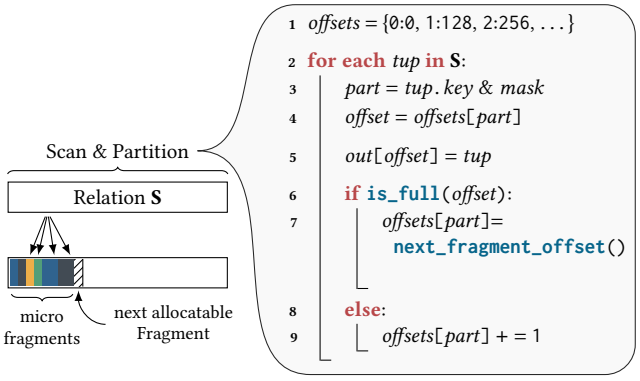


Figure 2: Materializing data into micro fragments of limited capacity.

As a remedy, more complex partitioning strategies have been developed in the literature. *Radix partitioning* performs the task in multiple rounds, in each round keeping the partitioning fan-out below the limits defined by TLBs. *Software-managed buffers* [4, 23, 24] buffer up several tuples (usually one cache line in size) for each partition in the CPU cache; once a buffer is at capacity, its tuples are copied to their in-memory location at a stretch. In effect, TLB misses arise only once per buffer flush but no longer for every single input tuple.

2.2 Micro Partitioning

Software-managed buffers are a two-edged sword. While they may indeed reduce the number of incurred TLB misses by several factors, the overhead of the extra memory stores becomes prohibitive when the partitioning fan-out stays small. We argue that additional copying from the buffer to memory is unnecessary. We introduce *micro partitioning* as an alternative to reduce TLB thrashing during partitioning.

Micro partitioning follows the idea of buffers concerning memory density. Like software-managed buffers, which reserve one cache line for each partition as a buffer, we reserve small chunks of memory that we dub *micro fragments* (or simply *fragments*). Micro fragments receive a limited number of *tuples* during the partition phase. However, unlike software-managed buffers, micro fragments go beyond the size of a cache line and are not temporary buffers. Instead, they directly act as a (tiny) subset of the overall partitioned data set. In this light, fragments resemble *morsels* [6, 8, 13] as used in the engines of *HyPer* [17] and *Umbra* [18, 26].

For partitioning a relation into micro fragments, we first allocate a contiguous memory block from the OS, large enough to accommodate all tuples. Like hash-based partitioning, we maintain a map that associates each partition with its corresponding offset within the partitioned data set for inserting the next tuple. However, in contrast to hash-based partitioning, these slots do not need to be calculated prior to partitioning. Instead, we "allocate" a micro fragment from the partitioned data (which is not an allocation in the sense of traditional memory allocation; instead, we reserve a fixed number of offsets for each micro fragment). As soon as one fragment is at capacity, we allocate the next from the tuple-receiving data set, which relates to incrementing an overall micro fragment

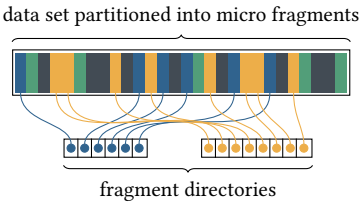


Figure 3: Bookkeeping of micro fragments and their partitions. For each logical partition, a *fragment directory* connects the fragments that form the *partition*.

counter multiplied by the fragment capacity to derive its offset. We illustrate our approach in Figure 2 in combination with pseudocode. In that sense, our approach levitates between "classical" hash-based partitioning (with substantial and variable-sized partitions) and software-managed buffers (which are limited to very few tuples).

The partitions' physical layout becomes a fundamental difference to hash-based partitioning: While hash-based partitioning stores a partition's tuples in a continuous line, micro partitioning splits the data over several locations. Additional bookkeeping is needed to combine the fragments into *logical partitions* processed at a stretch in the subsequent phase (e.g., building the hash table). Figure 3 shows an example using queues of pointers as fragment directories to represent logical partitions.

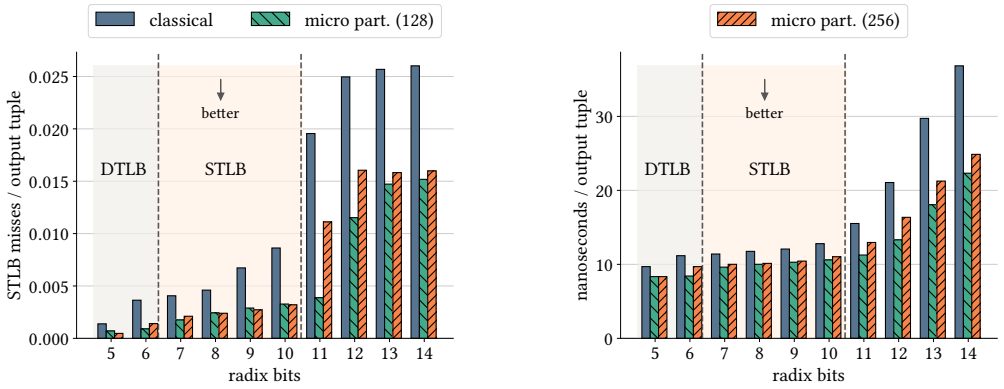
As a bonus, micro partitioning eliminates the need for a histogram: Since the tuples are consistently partitioned into fixed-sized subsets, there is no need to determine boundaries in advance (or realign partitioned tuples during partitioning when desisting from histograms). Instead, the boundaries of the individual micro fragments are statically determined by their capacity. Consequently, additional space is allocated since not every partition is guaranteed to be sized by exactly a multiple of the capacity. However, this is negligible considering the large amounts of main memory in today's servers.

2.3 Micro Partitioning in Action

The true strength lies in the tightly spaced physical layout. Given the small size of each micro fragment, multiple of them will fit into a single memory page. This increases the chances that fewer virtual page addresses must be translated into physical page addresses even with various partitions—hence fewer requests miss the TLB.

Based on this, capacity is a determining factor in the design of micro fragments. Two considerations are necessary: If the capacity is too small, the phase that follows partitioning (such as join or grouped aggregation) must reassemble many fragments. We will discuss this fact later in Section 3. In contrast, when fragments are too coarse-grained, the address space for the partitions written simultaneously will span over multiple memory pages, which may reduce the TLB-friendly advantage.

To understand the performance possibilities, we compare two micro fragment capacities with state-of-the-art ("classical") radix partitioning in a micro-benchmark fashion. For that purpose, we adopt the workload used by Balkesen et al. [3], which partitions 16 byte-sized tuples. So far, we will only focus on the partitioning phase and use a 4 GB relation as a workload (the "full" radix join



(a) STLB misses **(b) Execution time**

Figure 4: Micro benchmark comparing state-of-the-art ("classical") and micro partitioning with capacities of 128 and 256 tuples.

will be addressed in Section 3). Accordingly, the influence of the number of partitions is "only" limited to the TLB and does not affect hash tables in the cache. However, we want to spot the potential and the limits for different numbers of partitions. For capacity, we choose 128 tuples and 256 tuples. With 16-byte wide tuples in this benchmark, the resulting micro partitions are 2 kB (128 tuples per micro fragment) and 4 kB in size, yielding two and one fragment on a 4 kB memory page, respectively. We run the benchmark single-threaded on a machine with 64 DTLB and 1 536 STLB entries (more hardware details are given later in Section 4).

Figure 4 demonstrates the results. While classical partitioning leads to a comparatively large number of STLB misses with only a few partitions (Figure 4a), our approach with 128-tuple-wide micro fragments benefits from the tight partition layout in memory. This becomes notably evident when moving from 1 024 (10 radix bits) to 2 048 partitions: As fragments with 128 tuples still fit into the second-level TLB (one appropriate micro fragment only inhabits half of a memory page), the number of simultaneously written memory pages for classical partitioning and coarser fragments exceeds the TLB's capacity.

To our surprise, the benchmark shows that even 256-tuple-sized micro fragments result in considerably fewer STLB misses at a high fan-out (e.g., 14 radix bits), which also reflects in the execution time (see Figure 4b). This effect is also observable with broader micro fragments (e.g., 512 tuples). Experiments on additional hardware (with considerably smaller caches and less TLB capacities) have led to very similar results.

Effect of Software-managed Buffers. Micro partitioning is related to software-managed buffers to a certain extent: Tuples are written in condensed space to reduce TLB misses, while micro partitioning avoids copies from a temporary buffer. As depicted in Figure 4, however, a large number of partitions increases the number of simultaneously written memory pages to such an extent that also micro partitioning drives the TLB to its limit and beyond. Software-managed buffers can also extend micro partitions in such scenarios, just like "classical" hash-based partitioning. Figure 5 compares

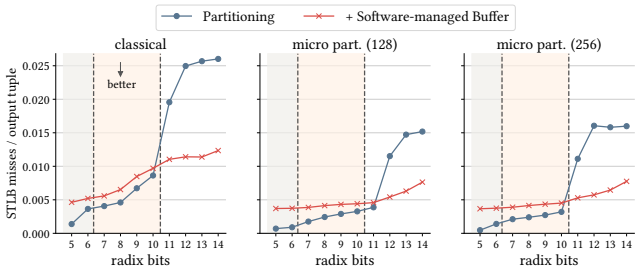


Figure 5: Comparing STLB misses during partitioning with and without software-managed buffers.

partitioning with and without buffers for hash-based and micro partitioning, using the STLB misses as a metric. The results indicate that software-managed buffers have indeed a positive effect on micro partitions. Besides reducing TLB misses at high partitioning fan-outs, utilizing software-managed buffers on top of micro partitions shows fewer TLB misses than on top of classical partitioning (and implicitly better performance).

3 TASK-BASED PARTITIONING

We saw how micro partitioning can provide a practical benefit. Let us now turn our attention to assembling micro fragments for the subsequent step, such as building or probing a hash table. By their design, classical partitioning algorithms facilitate linear scanning of each partition. Our approach, contrarily, spreads a partition over several dispersed locations, which must be glued together for further processing.

3.1 `MxTask` Abstraction

In recent years, task-based processing models have gained attention in database research (e.g., [6, 8, 13, 19, 20, 26]) and beyond (e.g., [1, 2, 12, 22]). In previous work [16], we presented `MxTasks`, which extend the conventional task interface by the capability to *annotate* tasks with application-specific knowledge. Annotations

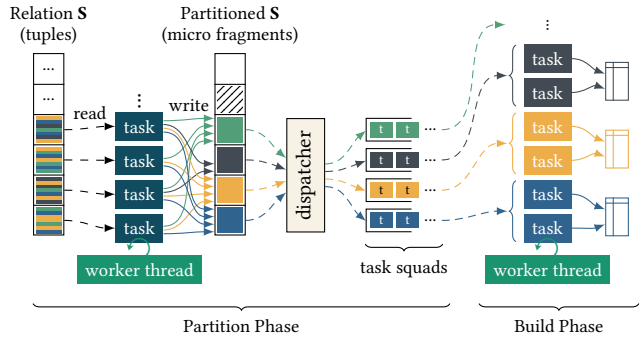


Figure 6: Illustration of the partition- and build phases. During the partition phase, tasks materialize tuples into micro fragments and spawn tasks if a fragment is at capacity. Annotating appropriate tasks with task squads helps the dispatcher to execute tasks partition-wise.

can include synchronization requirements for data objects or indications to the MxTasking framework about the in-memory data a task will read during execution. This knowledge sharing enables the runtime system to synchronize concurrently executed tasks and automatically *prefetch* data from memory into higher-level caches to hide latencies behind the execution of other tasks. Furthermore, each task is guaranteed to execute without interruption. Using a B^{link}-tree as a poster child, we demonstrated that MxTasks could handle even fine-grained data structures.

3.2 Dispatching Micro Fragments

Section 2.2 mentioned that fragments of a logical partition must be accounted for to be processed at a stretch. A straightforward way would be to take Figure 3 literally and maintain an explicit fragment directory that associates a partition with all its fragments. However, this strategy has two drawbacks: Firstly, the resultant code grows more complicated and must be tailor-made. Secondly, the scan of a partition for the subsequent phase is scattered and thus nearly unpredictable for the hardware prefetcher. Notably, hardware-based prefetching is known to assist linear scans on adequate volumes of data and implicitly simple-to-identify access patterns.

Annotation-driven Task Dispatching. To reduce the implementation effort of micro partitioning-based algorithms, we take advantage of the annotation mechanism of the MxTasking framework. We augment each task with a label that we refer to as its *task squad*. The task squad annotation logically connects (and makes this connection explicit to the MxTasking framework) MxTasks that process fragments from the same logical partition.

When multiple tasks should access the same data structure in succession, the application developer requests one or more task squads from the MxTasking runtime. In the light of data partitioning, each partition relates to a unique task squad. Internally, MxTasking pairs each squad with an individual task queue that receives only suitably annotated tasks. Whenever the dispatcher finds a task annotated

```
   // Create 1024 hash tables and one task squad for each partition.
1  tables = new HashTable[1024]
2  partitions = mxtasking::new_squads(1024)
3  offsets = {0:0, 1:128, 2:256, ..., 1023:130944}

   // The following loop is executed by multiple PartitionTasks.
   // Note, that data is a subset of the to-partitioned relation.
4  for each tup in data:
5      part = tup.key & mask
6      offset = offsets[part]
7      out[offset] = tup                  // Write the tuple to the partition
8      if is_full(offset):
9          start = offset - 127
           // Create a task that probes the HT and annotate..
10         task = mxtasking::new_task<ProbeHT>(tables[part],
             &out[start], 128)
11         task->annotate(partitions[part])    // ...the partition for
             dispatching
12         task->annotate(&out[start], 1kB)    // ...data to prefetch
13         mxtasking::spawn(task)              // Commit the task
           // Allocate the next micro fragment.
14         offsets[part] = next_fragment_offset()
15     else:
16         offsets[part] += 1

   // Push all tasks accessing the micro fragments to the task-engine.
17 mxtasking::spawn(partitions)
```

Figure 7: Pseudocode for partitioning the data using micro partitioning. When a micro fragment is at capacity, we spawn a new task to process the partitioned data and annotate it with the partition. Note that we chose a capacity of 128 tuples per fragment in this example.

with a squad, the task will be sent to the squad's queue instead of a worker's[1] default ready list.

After requesting a bunch of task squads, the application enters a phase during which it spawns all (or at least a series of) tasks required to access a data object (e.g., a hash table) in bulk. We illustrate this procedure in Figure 6. In the partitioning phase, the application spawns tasks that scan the input relation and materialize tuples into micro fragments. To that end, we also segment the relation into morsel-like *relation fragments* (statically in advance) of the same size we use for micro fragments. Each task reads a relation fragment in a one-to-one relationship.

When a micro fragment reaches its maximum capacity, we spawn a new task to scan and process it during the subsequent phase. Before sending the newly created task to the MxTasking runtime, we annotate the partition the fragment belongs to as a task squad. Figure 7 depicts the pseudocode for the task that partitions the data. We request (line 2) and annotate the task with information about the task's logical partition (line 11). Further, we annotate the task with the data that it will scan (line 12). The latter will

[1] *Worker* (threads) are responsible for executing tasks in MxTasking. To that end, each worker is linked to their task queue, which is populated by the *dispatcher*. Each worker is pinned to a dedicated logical CPU core.

empower MxTasking's built-in prefetching mechanism [16].[2] In line with our earlier work [16], we observed how the software-based prefetching mechanism built into MxTasking complements hardware prefetching in a very effective way, giving the latter sufficient time to recognize the access pattern.

The annotation mechanism of MxTasking allows us to kill two birds with one stone: We delegate the bookkeeping and composition of micro fragments to the underlying task execution engine, eliminating the need for the developer to manage it manually. Plus, we take advantage of the built-in software-prefetching mechanism to scan dispersed fragments while reducing memory latencies. Most importantly, however, all hardware-specific aspects are handled where they should be: in the MxTasking scheduler/dispatcher. All developer-written code stays oblivious to the underlying hardware.

Finalizing the Partitioning Phase. After partitioning the input relations, the operator must move on to the next phase. Up to this point, all tasks have found their way into squad-associated queues but must still be published to the worker's ready lists to get executed. Generally, this challenge can be addressed in several ways, e.g., by periodically transferring the tasks or by the developer "spawning" the squads at the end of the partition phase, which hints at the runtime to publish the tasks for execution. We found that the latter is effective and straightforward to use (line 17 in Figure 7). However, any improvement (to MxTasking in general) will further optimize task-based partitioning.

Parallel Partitioning. We still have to address the implementation of task-based micro partitioning in a parallelized setting. In fact, this requires minimal effort. Since we have implemented the entire set of operators around tasks, it does not matter (from the implementation point of view) how many cores or worker threads execute tasks in parallel. Solely the allocation of the fragments must be implemented atomically. As this only involves incrementing an integer, *atomic* compiler built-ins (or atomic C++ types like `std::atomic`) allow for a lightweight solution.

We chose to allocate a distinct partition block to each logical CPU core. This enables the tuple's materialization respecting NUMA domains while the partitioning remains unchanged from an implementation aspect. Nevertheless, this coin has two sides: Partitions may be processed by one region even if another materializes them. Due to the higher write latencies, we decided to write the partitions NUMA-local instead of optimizing for local read accesses.

4 EXPERIMENTAL EVALUATION

For the experimental evaluation, we use a two-socket Intel Xeon Gold 6226 machine clocked at 2.7 GHz. Each of the two processors holds 12 cores, 24 hardware threads, and 12 × 32 kB L1D, 12 × 1 MB L2, and 1 × 19.25 MB L3 data caches. Each (physical) core has a data TLB of 64 entries and a second-level TLB of 1536 entries for 4 kB pages.

We implemented a parallel task-based radix join to evaluate micro partitioning in a database-related context. For the benchmark, we follow former work [3], joining two relations with 16 B-sized tuples (8 B key and payload each). With 4 GB, we maintain the probe relation steadily throughout the benchmark. The build relation, however, varies with the number of partitions so that every partition allocates the entire L2 cache. This is equivalent to the original *Workload A* of [3] at 10 radix bits, joining relations of 4 GB and 256 MB. To classify our task-based join implementation, we choose the hardware-conscious radix join of Balkesen et al. [3] for comparison. Each of the following benchmarks utilizes all 48 available logical cores.

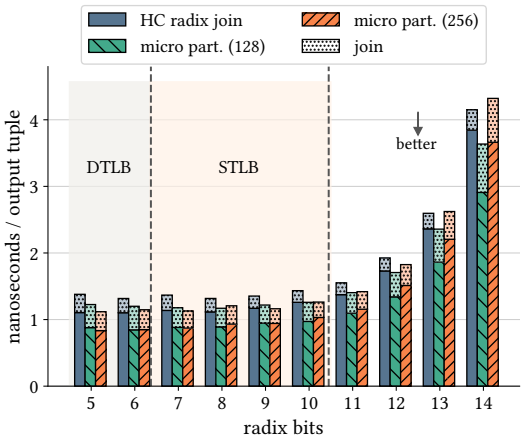


Figure 8: Comparison of the hardware-conscious radix join [3] using state-of-the-art partitioning and our task-based implementation using micro partitioning.

4.1 Comparison with State-of-the-Art

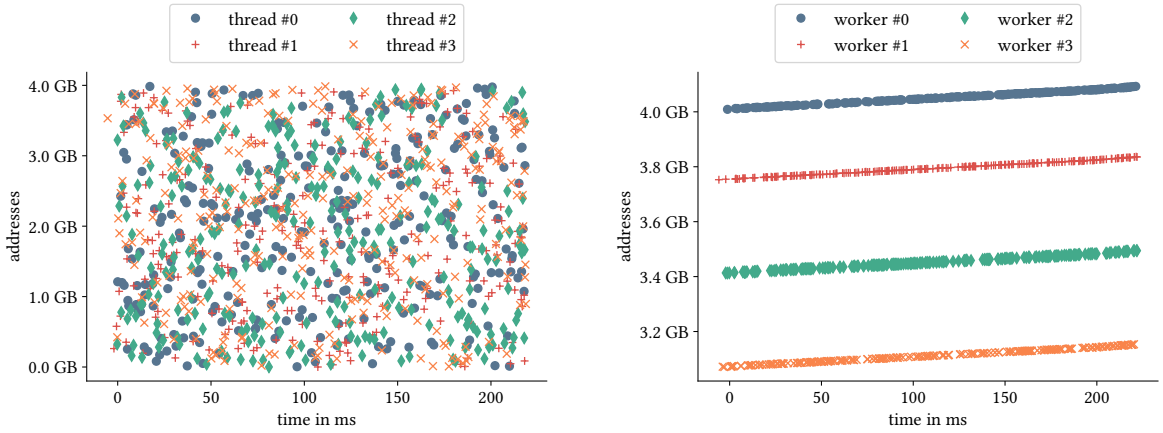
First, we compare the state-of-the-art radix-join implementation with our micro partitioning- and task-driven join. We demonstrate the results in Figure 8, analyzing partition granularities of 128 and 256 tuples. Similar to our previous analysis of the partitioning stage (from Section 2.3), the storage layout of micro fragments is advantageous for the partitioning phase: Partitioning improves by up to 25 %. On average, micro partitioning demonstrates a performance boost for the partitioning phase of 21 % when using 128-sized fragments. Coarser-grained fragments (256 tuples in this case) lose some advantage due to the less compact representation. However, the average performance benefit is still 17 %.

Even comparing end-to-end runtimes, micro partitioning substantially improves performance. Using 128-wide fragments leads to an 11 % improvement, whereas 256 tuples per fragment result in a 10 % improvement. Both values are averaged over all configurations.

In contrast to partitioning, join times have increased significantly. On average, the build and probe stages exhibit a performance degradation of 66 % (128 tuples per fragment) and 44 % (256 tuples) compared to the thread-based implementation. In the most extreme scenario (with 14 radix bits), the join becomes twice as costly, while this is still limited when having fewer partitions.

We found two reasons for this. Firstly, the micro partitioning-based join had to be adapted: In the original implementation, only indexes of the materialized tuples rather than the tuples data are stored in the table. The partitioned arrays are accessed during the

[2] Stating a prefetch *size* in MxTasking is optional. To keep matters simple, we statically set the prefetch size to 1 kB in this work.



(a) Partitiong phase of [3] **(b) Partitioning phase using micro partitioning**

Figure 9: Comparing memory stores to the partitioned data array of the state-of-the-art implementation and micro partitioning. For illustrative reasons, the plot shows only the partitioning phase of the probe relation on four randomly selected threads (worker threads in the context of `MxTasking`).

probe phase to check the tuple's keys for matches. The bucket-chaining mechanism is also built around array indexes, storing the (possible) chained bucket for each index of the build relation's tuples. However, micro fragments do not produce a comparable (coherent) memory chunk. Thus, we immediately store the keys and chained bucket references in the hash table, resulting in a more extensive data structure.

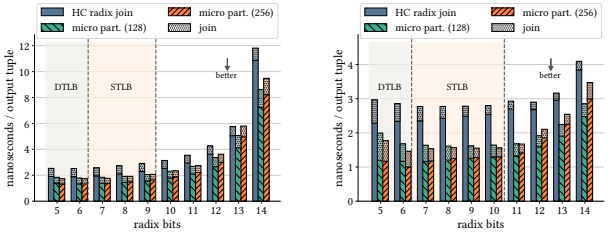
Secondly, each executed task scans a small subset of tuples for further processing (corresponding to the capacity of fragments). Using profiling tools (Intel *VTune* [10] and *perf* [14]), we discovered that the hardware prefetcher performs better on extended linear scans, which are given for coherent partitions.

Analyzing additional comparative measures (e.g., performance counters as in Section 2.3) is problematic. We found that individual worker threads occasionally stay idle between the partitioning and join stages until all partitions have been realized. This leads to single workers frequently querying their task queues, executing many instructions, and producing numerous TLB misses. Although this has a minimal effect on the workload, it has a considerable impact on the measures—because this occurs when there is temporarily no work. However, by reading performance counters, we can not discern this. Implicitly, this shows optimization potential: With a tuned, e.g., more dynamic, allocation of partitions to worker threads (such as task squad-level work stealing), idle times may be reduced and the throughput can be increased.

We executed the benchmark on additional hardware to conduct a more comprehensive evaluation of micro partitioning's hardware awareness:

- Intel Xeon E5-2690 with 16/32 logical/physical cores, 64/512 dTLB/STLB entries, and 256 kB L2 cache per physical core
- AMD EPYC 7501 with 64/128 logical/physical cores, 64/1, 024 dTLB/STLB entries, and 512 kB L2 cache per physical core.

We illustrate the outcomes obtained from that hardware in Figure 10. The contrast between "classical" partitioning and micro partitioning



(a) Intel Xeon E5-2690 **(b) AMD EPYC 7501**

Figure 10: Radix Join Benchmark (same as in Figure 8) executed on additional hardware.

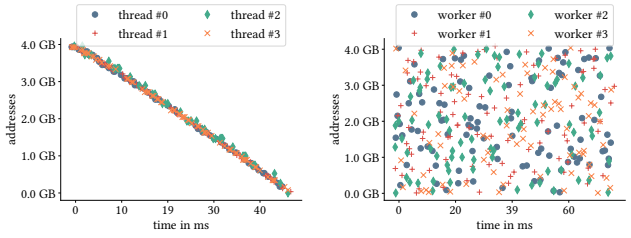
is notably greater on both machines, in relative terms (compared to the hardware mentioned above). Micro partitioning significantly improves partitioning performance, particularly in cases where the partition fan-out is low, indicating that our approach is not limited to a specific hardware configuration.

4.2 Memory Access Patterns

We will now demonstrate how micro fragments affect memory access patterns. To that end, we evaluate *memory stores* and *loads* that hit the partitioned data during the partition phase (when the data is written) and the join phases (which read the data).

We focus only on accesses to the partition of the probe relation. *Perf* was used to collect samples of memory operations. As the setup, we have chosen 10 radix bits for partitioning and 128-sized fragments to join relations of 4 GB and 256 MB.

Stores. Figure 9 depicts the chronological order of memory addresses written during the partitioning of the probe relation. We illustrate only 4 of the 48 threads as examples for visual clarity. The results support our argument that micro partitioning provides a more



(a) Partitiong phase of [3] **(b) Using micro partitioning**

Figure 11: Comparing memory loads during the probe phase.

TLB-friendly write pattern. The "classical" technique of radix partitioning materializes tuples by writing them extensively throughout the partition array (Figure 9a)—accessing a broad range of different memory addresses (and thus pages) simultaneously. Accordingly, the figure demonstrates a complicated writing pattern.

In contrast, when employing micro fragments, the (worker-local) partitions are written to ascending memory addresses (Figure 9b). The write operations to memory regions near one another result in fewer simultaneously accessed memory pages, making better use of the TLB.

Loads. When utilizing micro partitioning, a partition comprises many fragments that are spread across the memory and assembled during the join phase. This results in a random read pattern, as seen in Figure 11b, similar to the write pattern of radix partitioning. However, annotated tasks make the random read pattern predictable since the runtime knows upcoming tasks and accessed partition fragments. Task-assisted prefetching becomes a natural optimization that requires no effort for the developer.

In contrast, radix partitioning enables linear scanning of a contiguous memory chunk per partition during the join phases (Figure 11a). Finally, it is necessary to pick a battle: Random accesses during partitioning or random accesses at the join phase, with the latter allowing for more optimization potential—specifically in the dominating partition phase.

4.3 Task-driven Micro Partitioning in Detail

Utilizing micro partitioning with tiny fragments requires many tasks to process. This raises the question of task-based runtime overhead. We will address this question by breaking down the task-based radix join's CPU time into several individual parts.

Figure 12 shows the CPU cycles per output tuple. *Kernel*, *partition*, and *join* refer to radix join charges, while *runtime* and *tasking* are `MxTasking`-related. The number of cycles represents all logical core cycles. We used Intel *VTune* to record these samples.

Kernel. The measurement reveals that the OS kernel consumes several cycles. Most of these relate to mapping virtual to physical memory pages during partitioning. The findings are not surprising as we did not map the partitions ahead of the benchmark to provide comparability with the thread-based implementation of [3].

Runtime. We can also observe a variable quantity of cycles spent in the `MxTasking` *runtime*. These are associated with the "usual" handling of tasks (e.g., when fetching tasks from the queue) and—more

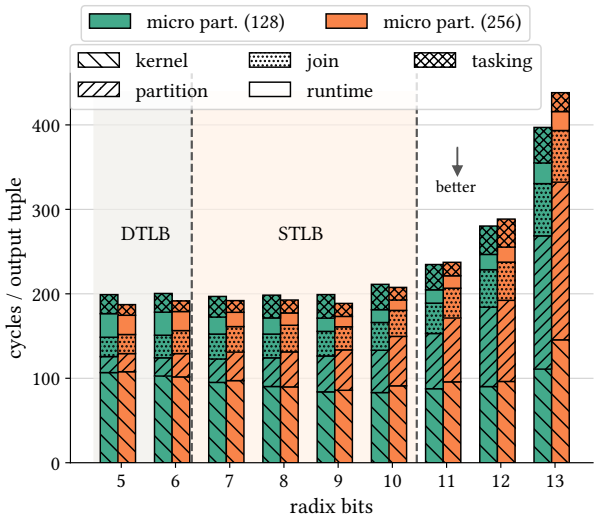


Figure 12: Cycle-based comparison of different micro fragment-granularities.

frequently—worker idle times. When a worker thread finds no tasks ready for execution, it aggressively pulls for new tasks. For instance, before the partitions can be joined, it requires the partitioning to complete by all worker threads, leaving some workers temporarily without work.

Tasking. As Figure 13 shows, *tasking* includes costs for *allocating* and *spawning* tasks. Plus, for each task, we instruct the `MxTasking` runtime to *prefetch* the task's processed data which consumes additional instruction bandwidth. The tasking costs are proportional to the capacity of micro fragments: Given more fine-granular fragments, more tasks must be processed (and more data has to be prefetched). We observed this pattern also for coarser and finer fragments.

During extended scans of the partitioned data chunks, fragments with a granularity of 256 tuples benefit more from the hardware prefetcher (what we compensate with task-assisted software prefetching for shorter scans).

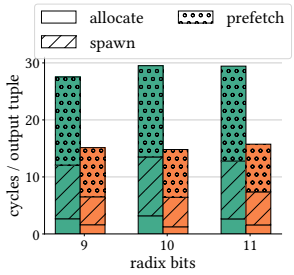


Figure 13: Breaking down *tasking* cycles in Figure 12.

Despite the minor overhead of tasking, `MxTasking` manages to keep it at a low level. Future optimizations promise to improve the join performance, as enhancements to the framework will have a direct positive impact.

5 SUMMARY

In this paper, we presented *micro partitioning* as a unique technique for partitioning that is TLB- and cache-friendly. Micro partitioning

tightens the simultaneously accessed address space during tuple materialization by separating the entire partition into tiny fixed-capacity fragments.

Combined with `MxTasks`, this approach also eases the implementation of partitioning: Spawning a new task for each fragment, annotated with the proper partition, is all the developer has to do. The task dispatcher takes over the assembling of fragments: With the help of the annotation, all tasks that belong to the same partition execute in bulk—aiming to reuse the CPU cache for partitioned data structures. However, micro partitioning is not limited to `MxTasks` and can also be adapted to, e.g., morsel-driven execution models and traditional threads.

The findings demonstrate that micro partitioning outperforms state-of-the-art radix partitioning by 21 % in our benchmarks while boosting the end-to-end radix join by 11 %.

ACKNOWLEDGMENTS

This work was supported by DFG, Deutsche Forschungsgemeinschaft, grant number TE 1117/2-1, and has partly been funded by the Federal Ministry of Education and Research of Germany and the state of North-Rhine Westphalia as a part of the Lamarr-Institute for Machine Learning and Artificial Intelligence. We would like to thank the anonymous reviewers, Maximilian Berens, Roland Kühn, and Lea Schönberger for their helpful comments and suggestions. We also would like to thank the ESS group from Osnabrück University for providing their hardware for benchmarking.

REFERENCES

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. 2002. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, Carla Schlatter Ellis (Ed.). USENIX, 289–302. http://www.usenix.org/publications/library/proceedings/usenix02/adyahowell.html

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*. Springer, 863–874. https://doi.org/10.1007/978-3-642-03869-3_80

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1754–1766. https://doi.org/10.1109/TKDE.2014.2313874

[5] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 168–180. https://doi.org/10.1145/3448016.3452831

[6] Alexander Baumstark, Philipp Götze, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2021. Instant Graph Query Recovery on Persistent Memory. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, Danica Porobic and Spyros Blanas (Eds.). ACM, 10:1–10:4. https://doi.org/10.1145/3465998.3466011

[7] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 54–65. http://www.vldb.org/conf/1999/P5.pdf

[8] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *Proc. VLDB Endow.* 12, 12 (2019), 2218–2229. https://doi.org/10.14778/3352063.3352137

[9] Dateng Hao and Li Sun. 2013. DPAgg: A Dynamic Partition Aggregation on Multicore Processor in Main-Memory Database. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*. IEEE, 1769–1777. https://doi.org/10.1109/HPCC.and.EUC.2013.253

[10] Intel. 2023. VTune Profiler. https://software.intel.com/vtune/. [Online; accessed Februrary 2023].

[11] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.* 8, 6 (2015), 642–653. https://doi.org/10.14778/2735703.2735704

[12] Alexey Kukanov and Michael J. Voss. 2007. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal* 11, 4 (2007).

[13] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[14] Linux. 2023. perf. https://perf.wiki.kernel.org/. [Online; accessed Februrary 2023].

[15] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[16] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1331–1344. https://doi.org/10.1145/3448016.3457268

[17] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[18] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[19] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1 (2010), 928–939. https://doi.org/10.14778/1920841.1920959

[20] Ippokratis Pandis, Pinar Tözün, Miguel Branco, Dimitris Karampinas, Danica Porobic, Ryan Johnson, and Anastasia Ailamaki. 2011. A data-oriented transaction execution engine and supporting tools. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 1237–1240. https://doi.org/10.1145/1989323.1989463

[21] Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 755–766. https://doi.org/10.1145/2588555.2610522

[22] Kazuki Sakamoto and Tomohiko Furumoto. 2012. Grand central dispatch. In *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 139–145.

[23] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 351–362. https://doi.org/10.1145/1807167.1807207

[24] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *Proc. VLDB Endow.* 8, 9 (2015), 934–937. https://doi.org/10.14778/2777598.2777602

[25] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 510–521. http://www.vldb.org/conf/1994/P510.PDF

[26] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1879–1891. https://doi.org/10.1145/3448016.3457260

[27] Jan Wassenberg and Peter Sanders. 2011. Engineering a Multi-core Radix Sort. In *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 6853)*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Springer, 160–169. https://doi.org/10.1007/978-3-642-23397-5_16