MxTasks: A Novel Processing Model to Support Data Processing on Modern Hardware

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Jan Mühlig

Dortmund

2023

Tag der mündlichen Prüfung:	12.10.2023
Dekan*in:	Prof. Dr.-Ing. Gernot Fink
Gutachter*innen:	Prof. Dr. Jens Teubner
	Prof. Dr. Viktor Leis

Abstract

The hardware landscape has changed rapidly in recent years. Modern hardware in today's servers is characterized by many CPU cores, multiple sockets, and vast amounts of main memory structured in NUMA hierarchies. In order to benefit from these highly parallel systems, the software has to adapt and actively engage with newly available features. However, the processing models forming the foundation of many performance-oriented applications have remained essentially unchanged. Threads, which serve as the central processing abstractions, can be considered a "black box" that hardly allows any transparency between the application and the system underneath.

On the one hand, applications are aware of the knowledge that could assist the system in optimizing the execution, such as accessed data objects and access patterns. On the other hand, the limited opportunities for information exchange cause operating systems to make assumptions about the applications' intentions to optimize their execution, e.g., for local data access. Applications, on the contrary, implement optimizations tailored to specific situations, such as sophisticated synchronization mechanisms and hardware-conscious data structures.

This work presents `MxTasking`, a task-based runtime environment that assists the design of data structures and applications for contemporary hardware. `MxTasking` rethinks the interfaces between performance-oriented applications and the execution substrate, streamlining the information exchange between both layers. By breaking patterns of processing models designed with past generations of hardware in mind, `MxTasking` creates novel opportunities to manage resources in a hardware- and application-conscious way. Accordingly, we question the granularity of "conventional" threads and show that fine-granular `MxTask`s are a viable abstraction unit for characterizing and optimizing the execution in a general way. Using various demonstrators in the context of database management systems, we illustrate the practical benefits and explore how challenges like memory access latencies and error-prone synchronization of concurrency can be addressed straightforwardly and effectively.

Contents

1 Introduction

The fundamental structures of data processing models used today were designed decades ago. At their inception, hardware was characterized by single-core CPUs clocked at modest speeds and limited RAM capacities from today's point of view. Since then, the hardware landscape has changed significantly. Modern servers are equipped with many CPU cores spread across multiple sockets, big caches, and vast amounts of main memory, organized in a non-uniform memory access (NUMA) fashion. Historically, the software experienced performance improvements due to hardware advancements, such as increased CPU clock frequencies, with minimal engagement required. However, we have reached a point where relying only on faster hardware to achieve straightforward performance boosts has ended. In order to leverage newly available resources, software must adapt and explicitly exploit provided hardware features, e.g., massive parallelism, multi-level cache hierarchies, and SIMD instructions. And the trend continues. Heterogeneous and specialized CPU cores and various types of memory, such as persistent and high-bandwidth memory, can already be found in increasingly complex systems today [118]. In the future, the hardware is expected to become even more sophisticated. Although modern hardware can accelerate software, it can also impede its efficiency. Massive parallelism and heterogeneity have great potential for improving performance but pose complex challenges, such as the efficient utilization of CPU resources, synchronization of concurrency, and incorporation of coprocessors.

Owing to the increasingly affordable cost of DRAM, it has become common practice for database management systems (DBMSs) to store data exclusively in the main memory. Consequently, DBMSs are now combating memory access *latencies* and *bandwidth* as a formidable foe [6, 24], particularly when the hardware is confronted with unpredictable access patterns. In light of this, hardware and software engineers have invested substantial efforts toward

optimizing data access. Hardware employs fast CPU caches to accelerate recurring data access and attempts to bring data into caches proactively before it is needed. Software developers, in turn, tackle this challenge by designing data structures and algorithms *hardware-consciously* to respect the characteristics of modern hardware. For example, they prioritize local over remote data access in NUMA environments [80, 92], implement sequential instead of random access patterns [101], and reorganize data accesses to align with the properties of caches [100, 13]. Additionally, we can witness initiatives focused on bringing data into caches before its actual demand to reduce memory access latencies [34, 105, 124].

The rapid growth of parallel hardware has further led to an increased need for optimizing the *synchronization* of concurrent control flows. Latches, as mutual excluding synchronization primitives, for instance, reduce parallelism by serializing accesses instead of optimally utilizing parallel resources. Consequently, more intricate mechanisms have been proposed in the literature, such as fine-grained latches [72], differentiation between reading and writing operations to enable parallel reads, and optimistic variants speculatively performing read operations [102, 94, 95]. Specific endeavors propose the implementation of data structures that carefully avoid latches altogether [96, 147]. Nevertheless, most of these techniques are tailored to particular situations and are cumbersome to implement and apply universally.

1.1 Motivation

To fully develop its potential, the hardware requires explicit support from the software. The abovementioned challenges—reducing memory access costs and efficiently utilizing parallel resources—illustrate particular examples. The efforts discussed also demonstrate that many of these problems are tackled at the application level. DBMSs, for instance, tend to choose this road to squeeze the last bit of performance out of the system. In fact, specific applications have taken the step of re-implementing services traditionally provided by the operating system (OS), such as thread scheduling and memory management, to tailor them to their needs. In doing so, they *bypass* the OS and manage resources in-house. *SAP Hana* serves as an illustrative example [117]. Other work addresses the challenges within the OS to devise optimizations beyond the boundaries of a single application. *Linux*, for instance, attempts to improve data locality by migrating memory pages between NUMA nodes based on the historical access pattern of the applications [142, 103, 43].

These two opposed methodologies need to be revised to address the problem fundamentally. To accomplish this, the execution layer, e.g., the OS, needs a comprehensive understanding of the application's behavior and intention. However, such information is hard to exchange as current interfaces lack the possibilities to do so. *Threads*, the standard control flow abstraction, can be

considered a "black box" that hardly provides any communication between both layers. One of the reasons can be attributed to the coarse granularity: During its execution, a thread typically performs numerous data accesses and computations, making an accurate specification of a thread's intention challenging despite the knowledge available in the application.

In order to address this dilemma, specific frameworks interweave knowledge from the application—specifically DBMSs—with the control flow. The basic idea is to break computational work into fine granular packages, called *tasks*, which are distributed to specialized worker threads with the assistance of application knowledge. Which specific knowledge is leveraged to allocate tasks to worker threads depends on the particular system. For example, *DORA* uses its internal understanding of transactions to divide them into small execution units scheduled with accessed data in mind [119]. As a result, the code is migrated to the data instead of vice versa, avoiding disorganized access across parallel transactions and diminishing contention on central components, e.g., the lock manager. The engines of *HyPer* and *Umbra* break down data into small fragments ("morsels") that are scheduled alongside the code for query execution in a NUMA-aware manner [92, 115]. Decomposing work into tasks enables the developer to design parallel software without worrying about the underlying many-core hardware. HyPer and Umbra take advantage of task-based parallelism by dynamically balancing the workload at runtime. Concluding from these observations, we argue that tasks, as small and encapsulated work units, might provide an appropriate level of granularity for expressing the application's characteristics and enabling optimizations in a general way.

1.2 Open Challenges

Expanding the concept of tasks as a central control flow abstraction presents various unique challenges. In order to evaluate the viability of the methodology, it is essential to address the following questions.

Exchange between Applications and the Execution Layer. Assuming that tasks illustrate an appropriate unit to describe the execution behavior, applications and the execution layer need transparent interfaces in both directions to specify and communicate this information. Examining *which* knowledge is shared and *how* it is propagated becomes necessary. This may require a fundamental redesign of the interfaces between these two tiers. However, it offers the potential to introduce new ways for optimizations: With information about accessed data and access patterns, for example, the execution layer can base optimizations on knowledge instead of relying on predictive models informed by historical patterns. Moreover, the ramifications of such optimizations beyond enhancing data locality in NUMA-based environments are uncertain. Plus, utilizing finely-grained work packages introduces additional efforts, such as

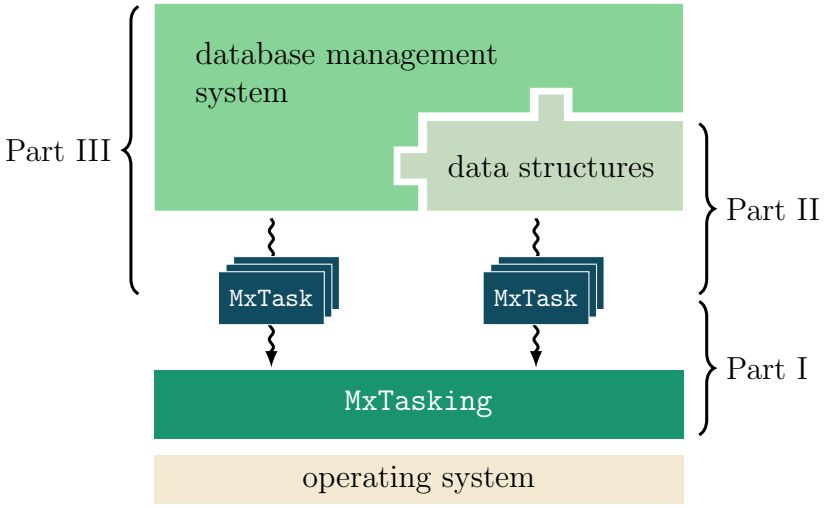


Figure 1.1: `MxTasking` software architecture and overview of this thesis.

dispatching, which must be minimized to ensure that the optimizations can pay off.

Leveraging Tasks on the Application Level. The consequent transition from a thread-based to a task-based processing model affects not only the execution substrate but will also lead to changes in the application's execution methodology to benefit from the newly available optimization possibilities. First, the granularity will shift toward well-defined and closed processing units instead of long code sequences. This transformation can interfere with the synchronization of concurrent control flows and require restructuring and redesigning algorithms and data structures. Second, the application must collect appropriate information to enable optimizations through an exchange. The methods for collecting such information and the level of detail need to be investigated.

1.3 Contributions and Outline

This thesis addresses the abovementioned challenges to optimize and ease the implementation of performance-driven systems on today's and future many-core hardware. Throughout this work, we present `MxTasking`, a task-oriented framework that facilitates the developer to convey the characteristics of applications to the execution unit. The thesis is divided into three parts. Part I delves into the details of `MxTasking`. Parts II and III demonstrate how the task-based abstraction leverages efficiency and implementation, using demonstrators sourced from various layers of a DBMS software stack. We illustrate the system stack as an overview of this thesis in Figure 1.1.

The thesis introduces the `MxTask` model in Chapter 2, exploring how applications can effectively communicate their intention with the `MxTasking` runtime. We present several building blocks that contribute to the performance of `MxTasking`. In Chapters 3 and 4, we will report on two ways of using the application knowledge to enhance the execution: bringing data into caches before it is needed and streamlining the synchronization of concurrent tasks. In Chapter 5, we show that these concepts can indeed optimize real-world use cases by utilizing a task-based B^{link}-tree as a demonstrator. We conclude the chapter with a comprehensive experimental evaluation that sheds light on various aspects of `MxTasking`. `MxTasking` and the task-based B^{link}-tree were introduced in

[109] Jan Mühlig and Jens Teubner. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In Proceedings of the 2021 International Conference on Management of Data, SIGMOD, pages 1331–1344. 2021.

Chapter 6 shifts the focus to the query execution engine. We examine how tasks can be used to process substantial data volumes in cache-aware segments while simplifying the implementation. Through an experimental evaluation, we prove with the example of joins that tasks have the potential to enhance the memory access pattern and require very little overhead. The technique was introduced in

[110] Jan Mühlig and Jens Teubner. Micro Partitioning: Friendly to the Hardware *and* the Developer. In 19th International Workshop on Data Management on New Hardware, DaMoN, pages 27–34. 2023.

Chapter 7 takes up these concepts and reflects the engineering of an `MxTask`-based DBMS. We show the applicability of `MxTasking` in the implementation of complex control flows, specifically those necessary for query execution. This is achieved by translating queries into a series of tasks that work collaboratively to answer the request. Additionally, we discuss multiple techniques for optimizing query compilation based on *FlounderIR*, which we use in our demonstrator to generate code for tasks. FlounderIR is a low-level and assembly-near intermediate representation that was initially introduced in

[52] Henning Funke, Jan Mühlig, and Jens Teubner. Efficient generation of machine code for query compilers. In 16th International Workshop on Data Management on New Hardware, DaMoN, pages 6:1–6:7. 2020

and extended in

[53] Henning Funke, Jan Mühlig, and Jens Teubner. Low-latency query compilation. VLDB J., 31(6):1171–1184, 2022.

Finally, Chapter 8 wraps up this work. Throughout the thesis, we emphasize the efficiency of various techniques used to build `MxTasking` and optimize the execution with interim evaluations.

The Bigger Picture. `MxTasking`, as described in this thesis, is one building block of the overarching `MxKernel`[1] vision, where we aim to address a dilemma in the design of hardware-conscious system stacks. Conventional OSs conceal the inner characteristics of the underlying hardware, thereby precluding optimizations that could address just these characteristics. `MxKernel` envisions enhancing the transparency of the interfaces between applications and the OS underneath in both directions. The `MxKernel` effort was initiated by Olaf Spinczyk and Jens Teubner, who contributed fundamental ideas and discussions in this context. Initial results of the `MxTasking` runtime as the central control flow abstraction of the `MxKernel` system were presented in:

[117] Stefan Noll, Norman May, Alexander Böhm, Jan Mühlig, and Jens Teubner. From the application to the CPU: holistic resource management for modern database management systems. IEEE Data Eng. Bull., 42(1): 10–21, 2019.

[111] Jan Mühlig, Michael Müller, Olaf Spinczyk, and Jens Teubner. mxkernel: A novel system software stack for data processing on modern hardware. Datenbank-Spektrum, 20(3):223–230, 2020.

The Author's Contributions. According to § 10 (2) of the doctoral regulations of the computer science department at TU Dortmund University from August 29, 2011, the author should indicate their own contributions to the results of collaborations that are used. The author is the principal author of the articles [109, 110] and of all contents from the articles that are used in chapters of this thesis. He is responsible for the concepts, the implementations, the presentation, and the analyses. Furthermore, the author of this thesis co-authored [52], where he contributed parts of the implementation and [53], where he contributed parts of the implementation and parts of Section 5. In addition, he contributed results and parts of Section 4 in [117]. Finally, the author of this thesis co-authored [111], where all authors have contributed equally.

[1] `http://mxkernel.org`

Part I

MxTasking Layer

2 MxTasking Building Blocks

*Parts of this chapter have been published in [**109**].*

Contemporary software faces an ambitious mission: In order to meet the demands for efficiency, applications must exploit highly parallel and heterogeneous systems with sophisticated memory structures. The responsibility for this undertaking lies almost exclusively in the hands of an application. As the complexity of hardware increases, the application mostly remains unaware of its characteristics—or is compelled to rely on the assistance of external libraries, e.g., *libnuma* [81]. And the situation is not different the other way around: The execution runtime, such as an OS, has to guess the intentions of the application running on top.

2.1 Background

We argue that the central problem stems from the characteristics of the prevailing control flow abstraction, which can be traced back to the 1960s: *threads*. Threads serve as a crucial interface between the OS and applications, intending to abstract computing resources and enable multiple applications to share these resources without mutual knowledge and fundamental understanding of the hardware. However, threads pose a considerable challenge: Applications cannot communicate their expertise to the OS. This results in a substantial untapped potential for optimizing the application's execution based on its unique requirements and characteristics. Memory accesses serve as a prime illustration. In increasingly complex memory hierarchies, applications tend to store their data across multiple NUMA regions, which exhibit different access latencies based on the accessing thread's execution location [20, 108]. As a result of the insufficient understanding, the OS is compelled to speculate on

which CPU core a thread must be scheduled to access data locally. OSs try to tackle this problem in various ways, for instance, by migrating memory pages between NUMA regions to minimize remote accesses based on the access pattern of threads [142, 103, 43]. In doing so, the OS bases its decisions on the past, as the future is inherently unpredictable.

How applications synchronize concurrent accesses is another consequence of the absence of information exchange between the application and the OS: The application bears the responsibility of synchronization. This approach is error-prone and cumbersome to implement as applications often use tailor-made synchronization primitives built without knowledge of the actual hardware. The primitive preference may vary based on the hardware characteristics, such as the presence of *hardware transactional memory* (HTM). On the contrary, the system knows about the hardware properties and the execution of other applications at runtime—and could generalize synchronization if it knew about the intentions of the applications. One reason for the missing exchange is the longevity of threads: It is not apparent which data objects a thread will access throughout its entire execution, making this information challenging to specify.

2.2 MxTask Abstraction

As a remedy, we present `MxTasking`, a task-based framework that assists the design of latch-free and parallel data structures. One of the fundamental tenets of `MxTasking` is to replace the "traditional" thread-based control flow abstraction with what we call `MxTask`s. An `MxTask` is a small, closed unit of work rather than a sequence of straight-line code to which a thread corresponds. From the application's point of view, `MxTasking` aligns with an event-based design: When the application aims to perform computational work, it *spawns* `MxTask`s that are received and processed by `MxTasking` asynchronously. Once a task has been picked up for execution, it completes atomically and non-preemptively, enabling tasks to share the stack and reducing the overhead of context switches.

By definition, an `MxTask` accesses only one data object (or at least a few) during execution. This characteristic renders tasks to be more fine-grained than threads. The equivalent of using a thread is to spawn multiple tasks one after the other, finishing a higher-level work package through collaborative efforts. We will demonstrate the concept of tasking using the traversal of a tree-like data structure as a running example. Conventional thread-based implementations of tree traversals iteratively visit one node after the other, following the nodes' child pointers from the root down to a leaf node. This pattern results in a lack of transparency for the thread and the underlying system. It is almost impossible to predict which nodes a thread will access during execution. The thread knows about the next node to visit only in the short term. By employing the `MxTask`s methodology, we break down the

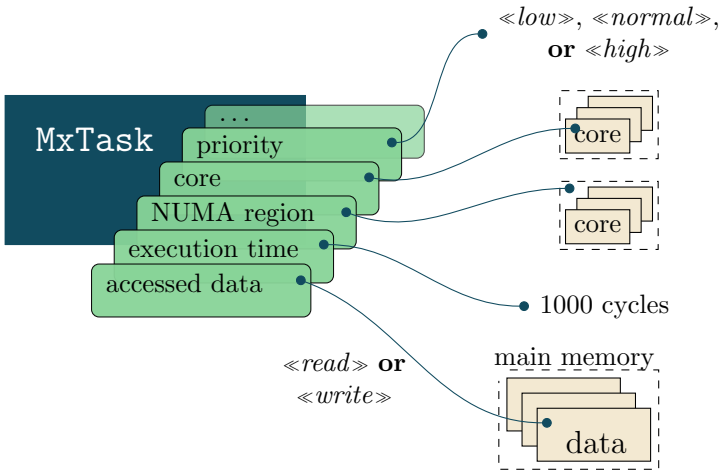


Figure 2.1: `MxTasking` provides annotations to share application knowledge with the underlying execution layer. Tasks can be annotated with its accessed data objects, a specific core or NUMA region, and priority.

traversal into a series of tasks: Every task visits only a single node and spawns a follow-up task for the next node to traverse. Consequently, it becomes concrete to the developer which node an individual processing unit will access during its execution when spawning that task. To make this knowledge also accessible to the system, the developer can attach this expertise to every `MxTask` in the form of *annotations*.

Annotations serve as an interface between the application and the underlying `MxTasking`. The fundamental concept aims to enable the sharing of application-based knowledge about runtime characteristics and effectively communicate a task's requests and requirements to the execution layer. As illustrated in Figure 2.1, the application can, for example, request the execution of a task at a specific CPU core or within a particular NUMA domain. However, the true power of annotations lies in the possibility of annotating the data objects a task will access. When spawning a task visiting a tree node during traversal, the developer can share their knowledge with the execution layer, providing the runtime with a detailed understanding of the interplay between code and data. Internally, task annotations are stored as a part of the task object in a structured manner. As such, annotated metadata are accessible to both the developer and the runtime.

`MxTasking` exploits annotations in several ways. The association between tasks and accessed data objects enables the runtime to load the data into the cache before it is accessed by the application, aiming to hide memory latencies behind the execution of further tasks. We will delve into details in Chapter 3. Plus, `MxTasking` leverages the provided knowledge to coordinate the tasks that concurrently access the same data object. Instead of bothering with

```
   // allocate an annotated tree root, that is accessed by tasks in parallel,
   // read heavy, and highly frequented
1  tree->root =
   mxtasking::new_resource<TreeNode>(mxtasking::isolation::shared,
   mxtasking::rw_ratio::read_heavy,
   mxtasking::access_frequency::high)

   // spawn a lookup task that starts traversal at the root node
2  task = mxtasking::new_task<LookupTask>(tree->root, key)
3  task->annotate(mxtasking::priority::high)
4  task->annotate(mxtasking::readonly)
5  task->annotate(tree->root, tree->node_size())

6  mxtasking::spawn(task)
```

Figure 2.2: Example-based usage of the `MxTasking` API to create a tree node and spawn a lookup task to start the traversal.

implementing various synchronization mechanisms (such as latches or optimistic techniques), the developer can entrust this responsibility to the execution layer, which will wrap synchronization around the task execution. All the developer has to do is annotate tasks with the accessed data object and—to optimize synchronization—indicate whether the task will read from or write to the data. Furthermore, `MxTasking` can harmonize the designated synchronization with the hardware substrate. The intricacies of this mechanism are discussed in detail in Chapter 4. Note that the use of annotations by the developer is optional, although more and better annotations may help `MxTasking` to improve performance.

For the final execution of tasks, `MxTasking` spawns a group of *worker threads*, each pinned to an individual logical CPU core. The primary mission of each worker is to pick up and execute tasks assigned to its *task pool*. In light of this, `MxTasking` mediates between the task-based execution model and the thread model of the underlying OS.

2.3 System Interface and Dispatching

In Figure 2.2, we illustrate the `MxTasking` programmer interface. In the example, we create a data object allowing shared access and assuming a read-heavy workload and a high access frequency (line 1). In lines 2–5, we create a high-priority lookup task, starting in read-only mode at the root of a search tree. The task is finally spawned in line 6. Subsequently, the *dispatcher* will select a specific worker as the recipient of that task and appends it to the worker's task pool. Notably, the dispatcher does not operate as a dedicated service (e.g., within a separate thread that runs alongside the application).

Instead, the application triggers dispatching by invoking a function on top.

Annotations primarily drive the allocation of tasks to the worker threads. In the most basic scenario, the application requests a particular worker or NUMA domain as an execution target—giving the application complete and lightweight control over the dispatching process. If the application annotates a data object, the selection of an appropriate worker depends predominantly on whether the access requires synchronization. We will discuss additional synchronization-dependent annotations and dispatching in Chapter 4. Otherwise, the dispatcher spawns the task locally within the task pool of the worker that invokes the dispatch.

To keep things simple, `MxTasking` does not track the load of workers and desists from implementing sophisticated and load-balanced scheduling. Therefore, we denote this process as dispatching instead of scheduling. Widely used task-based frameworks, such as Intel *Threading Building Blocks* (TBB) and *OpenMP*, incorporate load balancing by implementing task-stealing and calculating the percentage of load to balance the system [38, 88].

2.4 Task Pool

From the application's perspective, spawning a task and declaring it "ready for execution" corresponds to pushing it into a worker's task pool. From the system's point of view, the task is transferred to a queue-like data structure from which a worker thread continuously picks tasks for execution. Consequently, the task-receiving worker thread and multiple senders (primarily tasks executed on several worker threads) access the task pool concurrently, necessitating synchronization. This can result in increased overhead, such as cache coherence. We will now present different task pool strategies utilized for task management and discuss their strengths and weaknesses.

2.4.1 Single Queue

The task pool's most straightforward implementation is using a single queue per worker. Thanks to modern processors, which facilitate *atomic* instructions to read and modify a value without needing the assistance of a higher-level latch, the implementation of a single queue-based task pool becomes lightweight. Compared to latches (e.g., spinlocks or mutexes), atomic instructions reduce the synchronization overhead to a minimum, eliminating the need for senders to wait for another [107]. TBB, as a prominent task-based example, employs this strategy for internal task management [79]. However, intricate memory hierarchies can negatively affect performance. When tasks are spawned at high frequencies and directed toward a single task pool, multiple senders endeavor to concurrently modify the same cache line to append their task to the queue's tail—which ends up in high contention. In our experiments, we observed

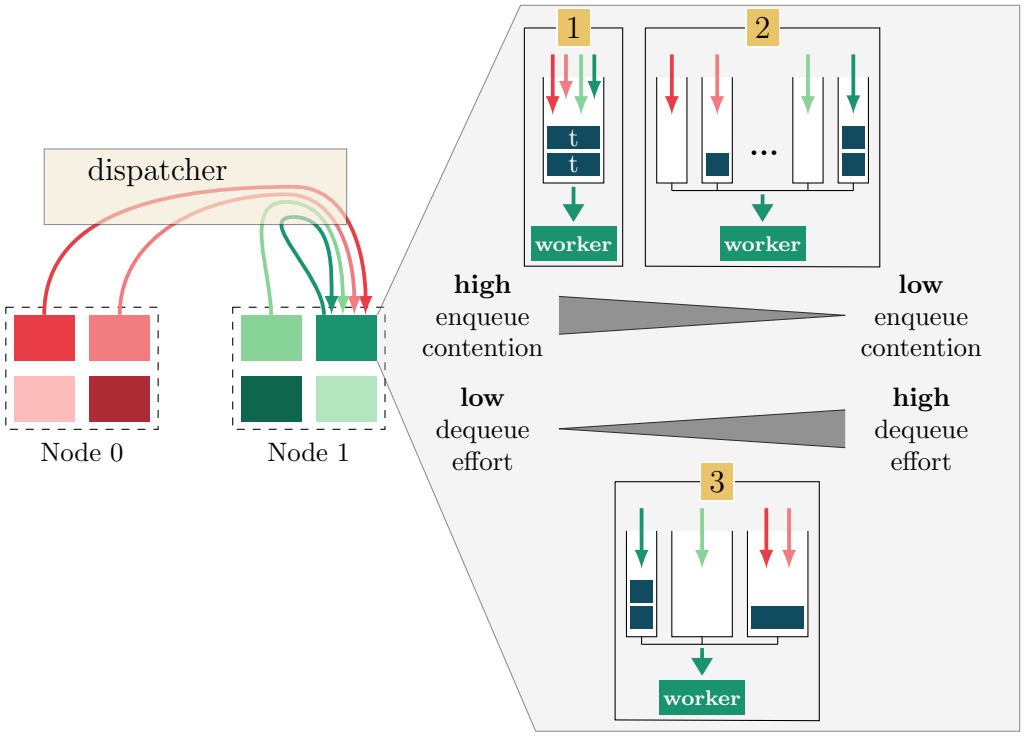


Figure 2.3: Illustration of spawning tasks from different worker threads to the identical task pool. The task pool can manage tasks within a single queue (1), a direct connection between each pair of workers (2), or using one queue per NUMA region to minimize contention (3).

that a single queue could become a bottleneck due to the high overhead for cache coherency, particularly when tasks are sent from multiple NUMA regions simultaneously to the same task pool. Figure 2.3 illustrates task dispatching through multiple worker threads that send tasks to a single task pool using a solitary queue (marked with 1).

2.4.2 Worker-to-Worker Queues

A potential solution that minimizes contention by reducing synchronization requirements among transmitting worker threads is to provide each worker a distinct location for appending tasks to the task pool of any other worker thread. We illustrate this strategy in Figure 2.3 (highlighted with 2). For implementation, each task pool comprises a fixed collection of queues corresponding to the number of worker threads alive within the system. Implicitly, each worker has a slot to place tasks into another worker's task pool. This approach guarantees that two senders will not append their tasks to the identical queue, thereby preventing potential communication among multiple senders. Consequently, synchronization is streamlined as it solely entails the sender and the receiver.

The limitation arises from the worker's perspective: Every individual worker thread is required to exert notable computational effort to retrieve all tasks from the task pool. Given that modern hardware systems offer hundreds of cores (and `MxTasking` uses one worker thread per logical core), the number of queues required to pick tasks from scales accordingly. Since the equitable distribution of tasks among all workers is an unrealistic scenario, some (or many) queues will remain empty but must still be polled. This allocation of substantial computational resources for performing lookups across several queues diminishes the task execution throughput of a worker thread.

2.4.3 NUMA-local Queues

The two presented methods exhibit opposite ends of a spectrum regarding contention overhead and dequeue effort, as indicated in Figure 2.3. A feasible approach to balance these conflicting considerations is to use a limited number of queues as an intermediary between the principles (marked as 3). Mitigating contention among multiple NUMA domains is one critical factor in optimizing the performance of shared data structures [152], such as the task pool. For that reason, `MxTasking` utilizes one separate queue per NUMA region within each task pool. In this manner, the number of queues to pick tasks from is kept low, and the acceptance of contention is limited and almost exclusively emerges between worker threads that belong to the same region. The only exception of contention across NUMA borders occurs when a worker from one region transfers a task into the pool from another region while the receiving worker tries to pick that task at the same time. However, this is observed solely when the queue comprises a small number of tasks or none at all, thereby signifying a minimal workload that contention could affect.

According to our observations, various workloads spawn a more extensive set of tasks within the local environment, targeting the task pool belonging to the sender. The run-to-completion paradigm of `MxTask`s ensures no overlap on a single worker when inserting and receiving tasks from the local task pool. Thus, `MxTasking` uses an additional lightweight and non-synchronized queue for locally spawned tasks.

Intermediate Evaluation

To understand the impact of the different task pool strategies, let us look at the performance of these concepts using tree inserts as an illustrative example. Note that we utilized all available cores on a two-socket machine; further details on the hardware configuration and workload will be given in Chapter 5. Within this particular workload, each node is linked to a specific worker thread; writing tasks are dispatched to that worker while reading tasks are processed locally (we will discuss further mechanisms for

synchronization later in Chapter 4). Figure 2.4 depicts the measurement results, breaking down the CPU cycles required for the insert operation and efforts for dispatching and dequeuing tasks.

The findings confirm that tasks can be efficiently picked from a single queue while dispatching causes noticeable overhead due to contention, negatively impacting performance. Worker-to-worker queues considerably minimize that contention among worker threads and implicitly accelerate dispatching by 50 %; however, the computational effort for receiving tasks triples compared to a single queue. Adopting NUMA-local queues for each worker is beneficial for achieving a balance between contention and computational effort. Compared to the worker-to-worker model, utilizing NUMA-local queues requires equivalent computational resources for task spawning. On the other hand, receiving tasks from queues increases by only 20 % compared to a single queue per worker. By combining the best of both worlds, NUMA-local queues experience 11 % and 20 % better performance for this benchmark than single and worker-to-worker queues, respectively.

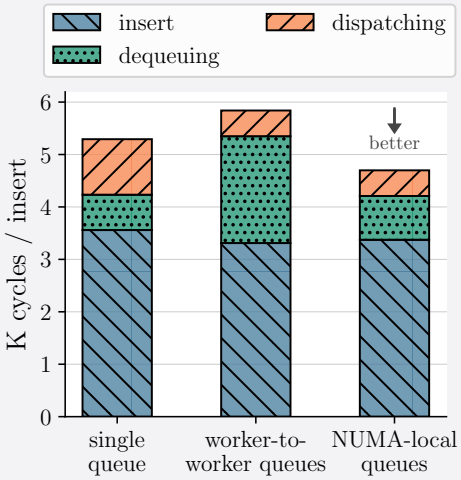


Figure 2.4: Comparison of different task pool concepts used to exchange tasks between workers.

2.5 Descriptor Allocation

Each `MxTask` corresponds to a code segment and a *descriptor* that includes annotations and execution arguments, such as the key to lookup and the tree node to process. Upon task spawning, the descriptor is initially allocated in memory. A task descriptor is typically sized to one or at least a few cache lines, encompassing arguments for execution. As a single task is responsible for processing only a fraction of a higher-level operation, the associated descriptors are subject to frequent creation and deletion. Hence, allocating the task descriptors is pivotal in achieving satisfactory performance. Using a global heap to allocate memory, akin to the system's native `malloc` call, will quickly become a bottleneck, particularly when multiple worker threads simultaneously request memory for new tasks.

The challenge of memory allocation is not limited to `MxTask`s. Various memory allocation libraries, including *Hoard* [19], *jemalloc* [46], and *TCMalloc* [54], manage allocation within the user space to achieve superior performance. These

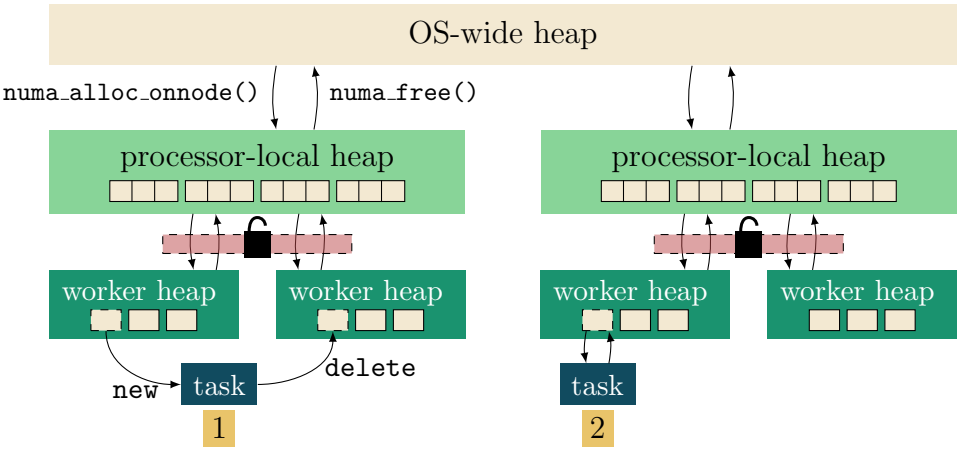


Figure 2.5: `MxTask`s are allocated within a multi-level allocator to reduce synchronization costs and enhance cache awareness.

allocators optimize memory allocation for scalability, heap fragmentation, and caching behavior while reducing the number of system calls.

2.5.1 Scalability

Specific allocators like *Hoard* facilitate scalability using dedicated memory heaps for each processor. Threads allocate memory from their local processor heap instead of invoking the system-wide `malloc` interface to request memory from the OS. Each processor-local heap holds a buffer of free memory and delegates it to threads seeking to allocate memory. In turn, the processor heap demands memory from the OS when the local buffer becomes empty, implicitly reducing synchronization costs between the processors.

`MxTasking` extends this concept by providing an additional layer to the allocation stack: A separated heap per worker thread becomes the point of contact for descriptor allocation. Figure 2.5 outlines this approach. When a task spawns a new one, it requests memory from the local heap. The allocation process from this heap is characterized by its lightweight nature, as the runtime guarantees that `MxTask`s execute until completion, thereby making synchronization superfluous. When a worker-local heap runs out of memory, it requests a new memory block from the processor heap with the capacity for a series of descriptors. Since the worker threads of a socket operate on a shared processor heap, it is imperative to ensure synchronization. The processor heap, in turn, will allocate memory from the global heap in a NUMA-aware manner when it has no memory in stock. As a result, memory management for `MxTask`s requires only a single latch[1] when allocating memory from the processor heap, thereby enhancing scalability.

[1]Apart from latches used by the OS to allocate memory for specific NUMA regions.

2.5.2 Cache Efficiency

After execution, the majority of the tasks are redundant for the application. By then, their descriptors can be made available for subsequent allocations. Inspired by `TCMalloc`, which keeps released memory under the ownership of the previous thread, `MxTasking` returns the free descriptor to the worker-local heap that executed the task (cf. 2 in Figure 2.5). The worker-local heap maintains memory in a last-in, first-out list to enhance subsequent allocations' chances of finding the descriptor in the CPU cache. Implicitly, the allocator places returned descriptors at the beginning of the list, and the following allocation will pick the memory block that has been recently released.

Reducing inter-processor communication and providing NUMA-aware allocation is a trade-off. Figure 2.5 illustrates a descriptor that is allocated by one worker but released by a different one (1). The free block is pushed to the heap of the releasing worker. In the worst case, this pattern affects multiple processors, where a task is allocated within one and deleted in a different NUMA region, shuffling the memory across NUMA borders. However, we accept mixing the memory to diminish synchronization efforts by minimizing communication.

Intermediate Evaluation

To understand the performance impact of descriptor allocation for task-based applications, we examine the efficacy of different allocators: glibc as the default backend for `malloc`, jemalloc, TCMalloc, and `MxTasking`'s multi-level allocator. As a representative workload, we employ parallel tree lookups which tend to stress the descriptor allocator due to their short execution duration. Figure 2.6 depicts the results, breaking down the consumed CPU cycles of a single lookup into costs for the runtime, lookup, and descriptor allocation. By using glibc as an example, the findings indicate that the allocation overhead can become significant, mainly attributable to the expenses associated with system calls and synchronization. While TCMalloc and jemalloc minimize costs substantially with comparable performance, `MxTasking`'s multi-level allocator

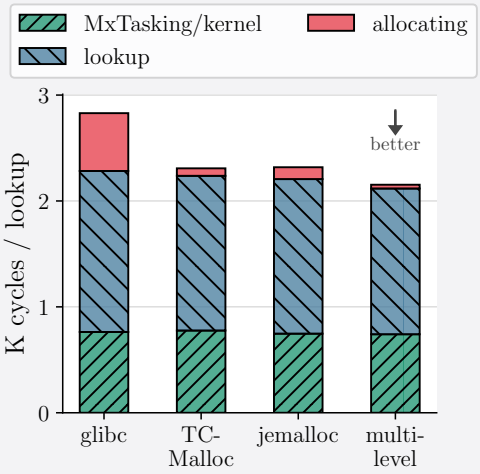


Figure 2.6: Comparison of different allocators used to allocate memory for the task descriptors.

improves the performance by reusing already cached descriptors for the following task allocation. As a result, we observe a reduction of about 9.5 % in the number of cycles consumed by the operation. Furthermore, only 34 cycles are sampled for descriptor allocation during a tree lookup.

2.6 Related Work

The concept of asynchronous, fine-grained control flows has been discussed several times in the recent past. Popular programming languages and environments implement this approach, for example, *NodeJS*, *C++*, and *Rust*. Tasks (others refer to them as lightweight threads or fibers) are typically scheduled and executed at the user level. Some OSs provide native support for lightweight threads, e.g., cooperatively scheduled fibers in Windows [5] and tasks in macOS [130].

With *Cilk*, Blumofe et al. [23] published one of the first runtime systems for parallel programming that schedules tasks onto OS threads. Aiming to simplify the engineers' work, Cilk focuses on the automatic load balancing of parallel applications and easy integration into existing software programs. Cilk supports *lock/unlock* calls on a latch variable to assist in synchronizing concurrent tasks. Since then, various task frameworks have been developed (e.g., [48, 11, 74, 137, 65]). Intel provides the TBB framework [88, 128, 149], focusing on portability and robust performance. TBB provides several synchronization primitives, including scalable (reader/writer-) latches, partially based on HTM. It is up to the developer to use them accordingly. For higher-level (and typically stream-based) data flow processing, TBB supports a graph-based programming interface and parallel container implementations, such as hash maps and vectors. *OpenMP* [12] forms another example of a task-based framework widely used in parallel software today. OpenMP enables developers to parallelize loops and regions of code by using compiler directives, which require only minimal modifications to the original code—making it easy to add parallelism to existing programs. Internally, the runtime can create independent units of work dynamically assigned to threads, allowing OpenMP to adapt to changing workloads.

Tasks—and similar concepts—have also been exploited in the context of DBMSs. Beyond HyPer, morsel-driven parallelism [92] has found its way into several applications and query execution engines. For instance, Umbra's scheduler can adapt the size of tasks (i.e., the number of processed morsels) to tune, e.g., for lower query latencies [144]. *SiliconDB* [44] targets heterogeneous platforms based on a morsel-driven query engine. Similarly, Baumstark et al. [17] exploited morsels for graph databases. *TAMEX* [151] translates logical query plans into task graphs to benefit from the load-balancing and prioritization possibilities of fine-granular work packages in parallel settings. Bang et al. [16] utilize tasks for various index structures like B-trees and hash-tables,

as well as transactional workloads. The primary concept entails dividing a given multi-socket machine into multiple domains. The accessing tasks are implicitly processed NUMA-aware by allocating data structures within a specific domain.

3 Annotation-based Memory Prefetching

*Parts of this chapter have been published in [**109**].*

Annotations provide the execution layer with knowledge from the application. This understanding enables `MxTasking` to tackle a fundamental problem of today's software: Data processing systems suffer from memory access costs, particularly when the data is not cached. The delay caused by the transfer of data from memory to registers substantially reduces computational efficiency. In order to reduce such waiting times, data can be brought close to the CPU before it is needed, accelerating subsequent accesses. We will now explore a first use case for utilizing annotations that assist task-based applications in this process: *prefetching* annotated data objects. Once annotations have been attached, `MxTasking` will improve runtime performance and hardware efficiency by prefetching data into fast CPU caches before executing the accessing tasks.

3.1 Memory Prefetching

Memory prefetching plays a crucial role in improving the performance of modern computer systems by hiding memory access latencies—aiming to bridge the gap between the CPU's processing speeds and slower main memory access times. Prefetching can be initiated by either hardware or software.

3.1.1 Hardware-based

To bring data into the cache ahead of access, the CPU attempts to predict the data that an application will likely access soon. The *hardware prefetcher* bases

its prediction on identifiable access patterns in the data access stream [40, 139]. One basic form is *adjacent prefetching*, where the hardware prefetcher speculatively fetches the neighboring cache line in addition to a requested one. A common and more complex example is *sequential access* patterns, where the hardware prefetcher identifies linear scans through a contiguous memory block and loads potentially required cache lines into the cache. However, the hardware prefetcher is limited to simple-to-detect access patterns. As soon as the application—from the CPU's point of view—randomly "jumps" through the memory, it becomes impractical to identify associated patterns.

3.1.2 Software-based

The application often has a better perspective regarding the data that will be needed subsequently. *Software-based* prefetching facilitates the program to hint to the hardware about upcoming data access by executing specific prefetch instructions. This gives the hardware the ability to asynchronously load data into the cache that is unpredictable by a stream-based look-ahead. Operations on tree-like data structures serve as excellent illustrations of *random access* patterns that challenge prediction by only examining data streams. While navigating through a tree and jumping from one node to the next, it is not feasible for hardware to predict upcoming node accesses. Binary search, typically utilized for node lookups, further complicates recognizing recurring patterns.

While prior research has demonstrated that software-based prefetching can indeed hide memory latencies (e.g., [34, 84, 105, 124]), it is also known to be cumbersome to use. Its effectiveness relies on the developer's comprehension of the hardware, typically baked into manually tuned code. Plus, temporal considerations pose significant challenges. In order to hide memory latencies, the hardware requires a *temporal gap* between initiating prefetches and accessing the data. If the software prefetches the memory too late, the hardware has too little time to bring the data into the cache, which almost eliminates the benefit of prefetching. If the prefetch occurs too early, the data may be already evicted from the cache when accessing and has to be brought back. In both scenarios, the performance is negatively affected by the penalty for the additional costs of executing the prefetch instruction and potentially through the increased use of the memory bandwidth.

And in practice, things are even more complex. The application usually acquires knowledge of imminently accessed data only after the prefetching opportunity has elapsed. Traversing a tree is an excellent example: Inner nodes are browsed to determine the subsequent node to traverse. Upon identification, the corresponding node is accessed promptly and scanned to continue the navigation. As only little computation occurs between identification and access, the temporal gap is insufficient for prefetching the node.

In order to tackle this challenge, various approaches interleave a set of

requests. With *group prefetching*, Chen et al. introduced a technique to leverage inter-tuple parallelism within hash joins [34]. This approach hides cache misses related to one tuple behind the computational effort for another. Group prefetching combines multiple iterations into a group and transforms the probe operation. Each stage (e.g., calculating the hash value) is performed for all grouped tuples before moving to the next stage (e.g., accessing the bucket). Immediately after computing a stage for a tuple, the following memory reference is prefetched. The reorganization of the code introduces a temporal gap between the identification and access of data, during which the hardware can move the data to the cache.

The idea of interleaving for various index joins is generalized by Psaropoulos et al. [124]. Their approach builds on *coroutines*, corresponding to stackless functions that can pause and continue execution with the assistance of the compiler. Whenever attempting to access data (e.g., a tree node or hash bucket), the coroutine executes a prefetch instruction and suspends. With a series of coroutines waiting, the thread continues execution by transitioning to the subsequent coroutine. The temporal window between the suspension and resume of the coroutine enables the memory subsystem to load the requested data into the cache. As a result, the actual load (initiated by the prefetch instruction) is hidden behind the execution of other coroutines. Similarly, Jonathan et al. exploit coroutines to hide memory latencies for state-of-the-art index structures [73]. He et al. demonstrate the performance-related benefits of software-based prefetching by interleaving coroutines to process transactions [63].

3.2 Prefetching Annotated Data Objects

Using coroutines in combination with software-based prefetching demonstrates the latter's effectiveness. However, this approach has two drawbacks: It requires extensive rebuilding of data structures and algorithms and provides only a limited ability to control the timing of the prefetches. In contrast, annotated tasks enable temporally separating the determination of accessed data objects and their actual usage. This facilitates `MxTasking` to hide access latencies behind the computational work of other tasks. Once dispatched, workers "see" tasks and associated data annotations within their task pools. Based on this information, the worker will inject prefetch instructions in-between task executions. Consequently, tasks will find the annotated data in the cache at the time of execution.

From the developer's perspective, prefetching becomes surprisingly simple. In fact, the prefetching mechanism in `MxTasking` is entirely transparent to the application developers. All they need is to annotate the data objects that a task will access and an optional hint, helping `MxTasking` to understand which data segments to preload. As a consequence, applications built on `MxTask`s require

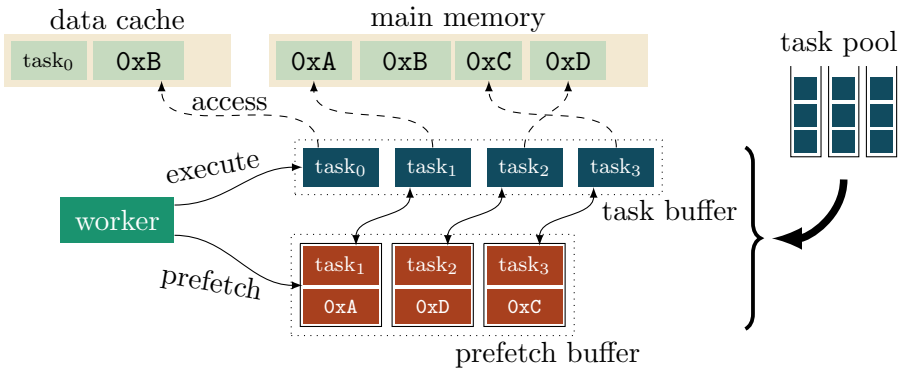


Figure 3.1: To get an explicit view of the next tasks to execute and data objects to prefetch, `MxTasking` extends the task pool by fixed-sized buffers. This also reduces pointer-chasing.

no additional adaption to enable prefetching. In addition, the prefetching mechanism is significantly more powerful than the existing approaches. In contrast to hand-crafted solutions, `MxTasking` will automatically schedule prefetch instructions even across task executions from different applications operating on the same tasking instance.

Implementation Details. To adequately perform data prefetching, the worker must "see" upcoming tasks and their annotations. Due to the temporal demands of the memory subsystem, it may be insufficient to only prefetch the subsequent task's data object, necessitating examining more than one task in line. The task pool, however, includes a set of queues that manage tasks like a linked list, chaining tasks through pointers. Consequently, to inspect the waiting tasks for their annotations, the worker must iterate over the chained items, which results in expensive *pointer chasing* and implicit cache misses for the task descriptors. To minimize implicit cache misses and streamline the access to a specific imminent task (e.g., the third-closest task), we extend the task pool by two additional buffers: one for a limited set of upcoming tasks and one for associated prefetches. Figure 3.1 outlines this approach. The worker contacts the task pool to refill the buffers whenever it runs out of tasks. This kills two birds with one stone: The worker must interact with each task descriptor only once when refilling the buffer instead of frequently traversing the queue to find the next data object to prefetch, minimizing pointer chasing. Plus, periodically picking tasks from the shared task pool, instead of after every task execution, reduces contention between former senders and the worker.

During the transfer of tasks from the pool to the buffer, `MxTasking` strategically positions the relevant prefetch information to ensure that the task's prefetches are initiated with an adequate lead time. In Figure 3.1, we assume a *prefetch distance* of one. Before executing $task_0$, the worker performs instruc-

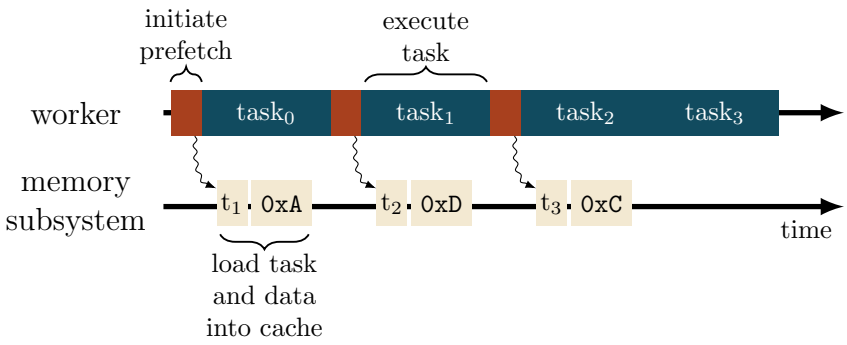


Figure 3.2: Timeline of interleaved prefetching and task execution.

tions to prefetch the descriptor and the accessed data object of $task_1$. While executing $task_0$, the hardware can move the requested data asynchronously from memory into the cache, hiding the latencies behind computations. Figure 3.2 depicts a corresponding timeline, showing how prefetches initiate the asynchronous transfer via the memory subsystem.

3.3 Static Prefetching

The prefetch distance for an adequate time gap depends on several factors. These range from hardware-specific parameters, e.g., the time required to bring data from memory into caches, to software characteristics, such as the execution time of tasks, while the memory subsystem brings data into the caches. Furthermore, online parameters such as current memory utilization can impact the ideal prefetch distance.

`MxTasking` provides a central parameter for configuring the prefetch distance in task units. Upon initiating the runtime, the developer can instruct `MxTasking` always to prefetch a task's data object the specified number of tasks in advance. This way, the developer can tune the execution for the unique attributes of a particular hardware system and the anticipated workload. However, a single parameter can be inadequate, especially when multiple applications simultaneously operate on top of one `MxTasking` instance. In practical scenarios, a single application will spawn tasks with differing execution durations, which may not align with a single predetermined prefetch distance.

3.4 Dynamic Prefetching

The natural progression of this mechanism is the dynamic adjustment of the prefetch distance during workload execution. This ensures that sufficient time is allocated between prefetching and execution for each task and its corresponding

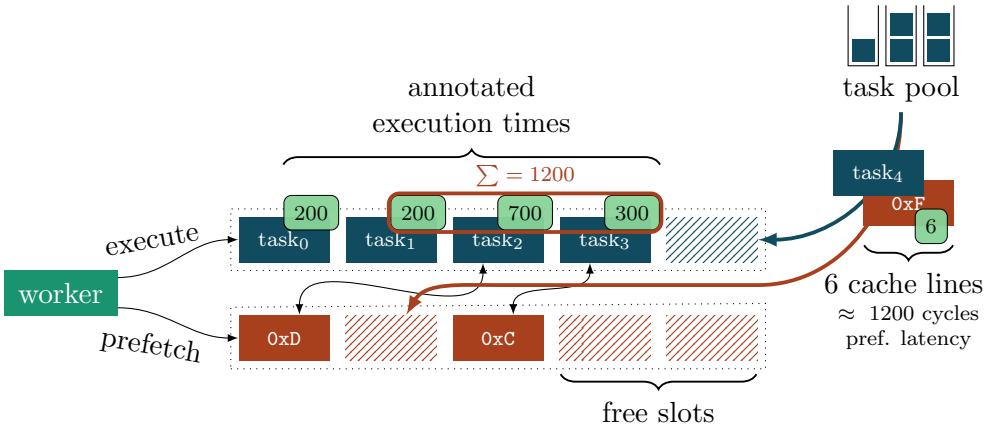


Figure 3.3: To schedule prefetches for annotated data objects, the worker calculates the prefetch latency and finds a slot in the prefetch buffer to hide the transfer from memory into the cache behind the execution of tasks.

accessed data object in applications with heterogeneous task durations. The challenge of scheduling software-based prefetches is not limited to `MxTasks`: Compilers and similar prefetching algorithms (e.g., [29, 90, 104, 8, 70]) face a similar dilemma when optimizing accesses to pointer-based data structures and loops by injecting prefetches into the generated code [126, 8]. To this end, it is necessary to strategically position the prefetch instruction with a suitable number of instructions prior to data access [90]. However, not the exact number of instructions is the determining factor, but their execution time, which can be approximated either with the help of cost models (e.g., [30, 150]) or profiling (e.g., [33, 78, 70]).

3.4.1 Scheduling Prefetch Instructions

In the `MxTask`-based world, the prefetch distance is established at the task level rather than the instruction level, as the runtime inserts prefetch operations only between the execution of two tasks. Upon appending a new task to the task buffer, `MxTasking` calculates an appropriate slot within the prefetch buffer. The challenge lies in finding a slot where enough tasks are executed between the prefetch and the newly placed task to cover the prefetch latency. This primarily depends on two key variables: (a) The prefetch latency, specified as the product of the number of prefetched cache lines and the memory latency for a single cache line, and (b) the execution time of tasks executed ahead of the dispatched task.

Figure 3.3 illustrates dequeuing a task from the pool, which is annotated with a data object requested to be positioned in the prefetch buffer. Upon appending the task to the task buffer, the runtime aggregates the approximate number of execution cycles for tasks scheduled in advance. The illustrated

example involves prefetching annotated data with six cache lines[1]. Given an exemplary latency of 200 cycles for prefetching a cache line, this sums up to 1200 cycles needed to prefetch the data object. The runtime compares the necessary cycles with the tasks currently awaiting execution in the buffer and subsequently computes an appropriate slot within the prefetch buffer.

In this illustration, we have taken two variables as given: The latency per cache line and the execution time of tasks. The prefetch latency is contingent upon the underlying hardware. It can be determined using third-party tools, e.g., Intel's *Memory Latency Checker* [1], which provides accurate results according to our observation. The information about task execution times can be made available to `MxTasking` in two different ways. First, the developer can provide annotations to the tasks, specifying the anticipated duration of their execution. Since this can be challenging, profiling tools (e.g., *VTune* [69] and *perf* [97]) and static analysis tools can be of assistance (as commonly used in embedded systems where satisfying given worst-case execution time bounds is an essential factor for correct behavior, e.g., [125, 121, 45, 41]).

3.4.2 Monitoring Execution Times

Second, `MxTasking` can generate corresponding annotations automatically by *monitoring* the tasks' execution durations throughout the workload. This also addresses the issue of offline estimations deviating from actual execution times, which can stem from various factors, e.g., the characteristics of the underlying hardware, the composition of the processed data, and the data placement with associated access latencies. To keep the monitoring overhead low and minimize the impact on the tasks' execution, we sample tasks in batches and aggregate the time for each task type (e.g., traversal or insert task). Although this is only an approximation, the values converge toward the actual execution times as time progresses. However, in contrast to annotating, monitoring always provides insights into the past and does not need to align with the temporal duration of forthcoming tasks.

Intermediate Evaluation

We compare static and dynamic prefetching with and without task monitoring to understand how dynamic prefetching can enhance `MxTasking`'s built-in prefetching. As a workload, we use tree inserts in a parallel setting, which include heterogeneous task execution times, such as short-running traversals and more time-consuming insert operations. Figure 3.4 illustrates the results.

The allocation of CPU cycles per insert operation is classified into different categories: the insert, including the traversal, executing prefetch

[1]The prefetching size is a part of the annotation.

instructions, monitoring task execution times, estimating the prefetch distance, and other tasking and system activity (e.g., dispatching). The results indicate that dynamic prefetching slightly improves efficacy compared to utilizing a static prefetch distance. Plus, in contrast to the latter, dynamic prefetching does not require any fine-tuning by the developer to adjust the central prefetch distance parameter. Approximately 44 cycles are spent on determining the prefetch distance for a single insert operation, which includes the execution of four traversal tasks and one insert task.

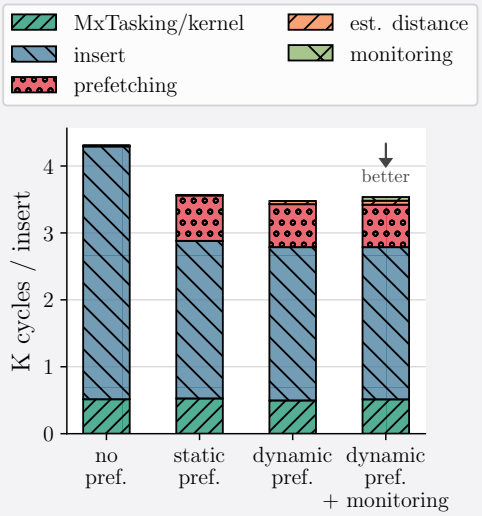


Figure 3.4: Comparing static and dynamic prefetching.

Monitoring the task execution times at runtime requires additional computational effort (66 cycles per operation), which stems from instructions to insert and look up execution times in a worker-local hash table. Plus, we utilized memory barriers to prevent out-of-order execution before and after the measurement of task execution times, which has an indirect impact, too. For measuring without monitoring, we annotated the tasks with observed execution times from an earlier iteration.

Despite the computational burden of monitoring and calculating the prefetch distance, we observe that the insert operation exhibits a 4 % increase in performance when employing dynamic prefetching as opposed to static prefetching. In contrast, static prefetching is more effective for lookup operations due to the workload comprising more homogeneous tasks. However, any improvement to this mechanism will directly enhance the performance of `MxTask`-based applications. Detailed comparisons with additional metrics (e.g., memory stalls) will be presented later in Chapter 5. Note that we used a static prefetch distance of two tasks for comparison, which performed best in this benchmark.

3.5 Extended Annotations

The quantity of prefetched data directly impacts the costs of prefetching. Software-based prefetching trades reduced memory access latencies for higher capacity utilization of instruction and memory bandwidth. Besides, CPUs may have only a limited buffer for outstanding data loads, including prefetches [2]. Hence, the application should carefully avoid prefetching too many or too

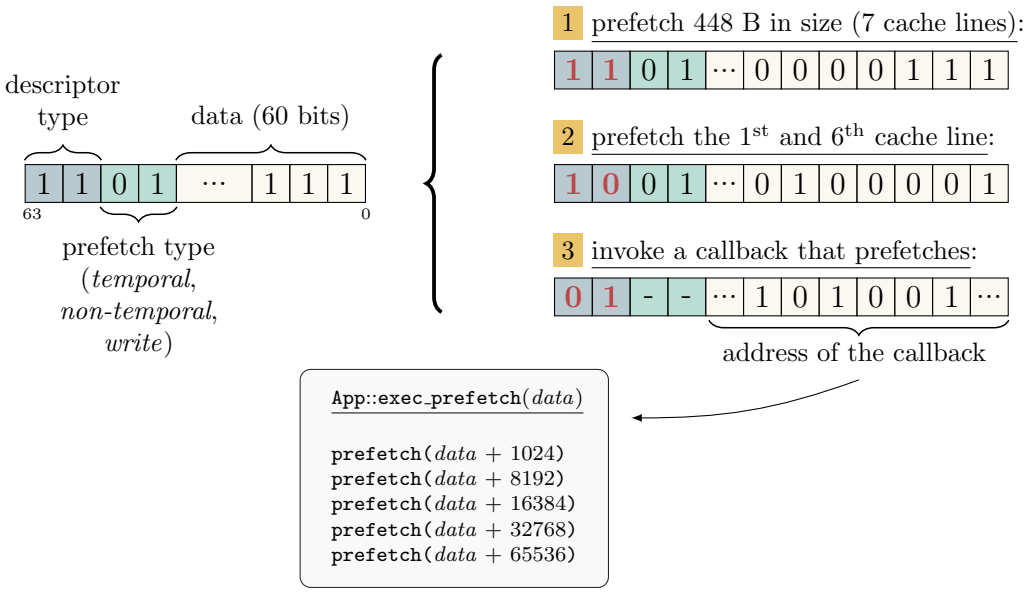


Figure 3.5: In addition to the data a task accesses, `MxTasking` accepts hints about which data segments should be prefetched. Such hints can be provided in the form of a size to be prefetched (1), specific cache lines (2), or a callback implemented by the application (3).

large data volumes. During a tree lookup, for example, only a single pointer or value is needed after identifying the index of the payload through a key lookup. Assuming that the node is split into halves, one for keys and one for child pointers or values, it may be supersized to prefetch the node entirely. Achieving an optimal balance between preloading specific node segments and tolerating cache misses is imperative, as accurately predicting the regions that will be accessed is challenging. `MxTasking` can only perform this application fine-tuning with the developer's engagement. To assist the developer in this process, `MxTasking` provides extended annotations to specify which segments of a data object require prefetching. A corresponding hint's most basic form is a parameter specifying the amount to prefetch. To give the developer even more control, `MxTasking` facilitates two additional types of hints: The specification of particular cache lines and a callback[2] that hands over the control of prefetching to the application.

Zooming into the implementation details, the annotation utilizes a 64-wide bitset[3]. `MxTasking` reads and interprets this hint at runtime and translates it into specific prefetch instructions. Figure 3.5 illustrates the bitset. The

[2]Special thanks to Olaf Spinczyk for bringing up this idea.

[3]As annotations are stored as part of the task descriptor, their memory footprint should be kept small.

two most significant bits are used to indicate the particular denotation for interpretation. The third and fourth most significant bits denote the nature of the prefetched data, specifically whether it is temporal, non-temporal, or intended for writing. Depending on the hardware, this affects how the CPU manages the prefetched data within the cache. While temporal refers to data that will be accessed repeatedly, non-temporal prefetches are intended for data that is only accessed once. Therefore, to maximize cache utilization, hardware systems tend to prioritize caching of temporal data, as non-temporal data can pollute the cache [2]. Sharing application-based knowledge with the hardware helps to minimize cache pollution. For example, we observed that seldomly accessed nodes in a tree exhibit the best performance when prefetched in a non-temporal manner. Conversely, prefetches signaling imminent write access will invalidate copies in other caches [2]. The remaining 60 bits express what exactly `MxTasking` should prefetch, depending on the annotation type, sketched in the following.

Prefetch Size. The runtime can be directed to prefetch a contiguous data segment by specifying a data object's (prefetch-) size (cf. 1 in Figure 3.5). This provides adequacy for a wide range of applications and has the potential to be transformed into prefetch instructions in a computationally efficient manner, as the data can be interpreted as an integer.

Specific Cache Lines. Alternatively, specifying a mask of accurate cache lines (cf. 2 in Figure 3.5) can reduce the amount of unnecessary data. One application is selective prefetching of a node during a tree lookup, where only the node's header (located at the beginning) and a specific cache line that contains the intended value are needed.

Prefetching by the Application. Both forms of representation encounter limitations when dealing with large data objects desired to prefetch in a scattered manner. While the size-based representation only enables prefetching data at a stretch, the fixed number of bits confines the mask-based representation to a maximum of 60 cache lines (3.75 kB for 64-byte wide cache lines). To enable prefetching of extensive and dispersed data, the developer can annotate a callback that grants the application control over prefetching. Internally, `MxTasking` interprets the 60 bits allocated for the data as a function pointer invoked with the data object to be prefetched as an argument (cf. 3 in Figure 3.5). As a result, the application can handle complex prefetching patterns for extensive data by executing software prefetches; scheduled by `MxTasking`. For example, tasks within a task-based query engine may only prefetch specific columns or rows of table fragments.

4

Synchronization of Concurrent Tasks

Parts of this chapter have been published in [109].

We will now focus on exploiting annotations beyond "only" enhancing performance: `MxTasking` employs annotations to *synchronize* concurrent tasks accessing the same data object. On parallel platforms, synchronization can quickly become a limiting factor of applications. For the developer, optimizing the program for scalability becomes challenging while balancing the application's needs and the underlying hardware's characteristics. The choice of an appropriate synchronization mechanism often depends on the access pattern of the application, the degree of parallelism, and the features of the underlying hardware. In practice, at the time of implementation, the developer typically lacks knowledge of the specific hardware on which the application will run.

The conventional (and uncomplicated) way to synchronize concurrent control flows is to use *latches*. Conversely, one of the most performant strategies is building data structures that avoid latches (e.g., the Bw-Tree [96, 147])—but this is notoriously difficult and error-prone. Somewhere in-between, *optimistic* primitives have shown superior performance for read-heavy workloads [31, 94, 95]. Nevertheless, the challenges are centered on the implementation and the customization to align with an individual application. In recent years, hardware has also been developed to provide synchronization support. HTM [64] appears to have the potential to facilitate synchronization straightforwardly and effectively [42, 93].

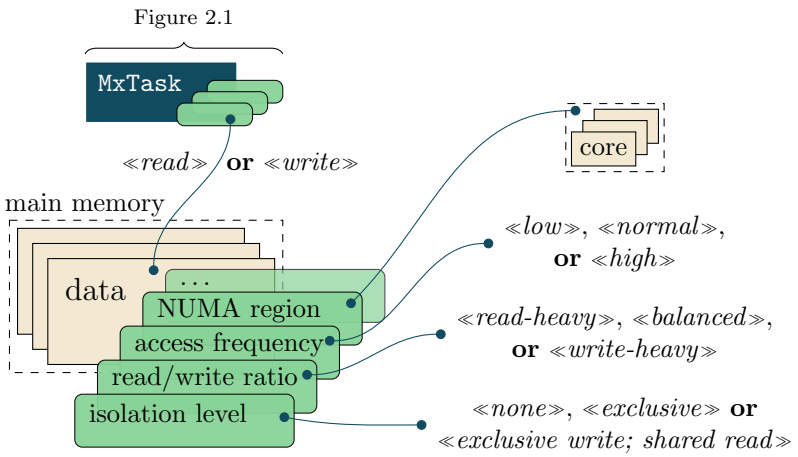


Figure 4.1: `MxTasking` provides annotations for data objects, enabling the application to share the needs and characteristics with the underlying execution layer. With the given knowledge, `MxTasking` will select a specific synchronization primitive that is wrapped around the execution of concurrent tasks.

4.1 Annotation-based Synchronization

The attractiveness of HTM lies in its user-friendliness, as the developer merely marks critical sections for synchronization purposes. By then, the hardware will ensure the correctness of concurrently accessed data structures. Although HTM has been deactivated on several available Intel processors and is excluded from further development [3], there are still efforts for different architectures to implement HTM, e.g., *ARMv9* [4]. `MxTasking` provides a likely comfortable and powerful way to apply synchronization to concurrent control flows—independently of the underlying hardware platform. Instead of using (hardware-) specific instructions, the developer can express the application's synchronization requirements through the annotation interface. All the developer needs is to annotate tasks and data objects. To facilitate synchronization, `MxTasking` requires information about the accessed data during a task's execution and its access characteristics, i.e., either reading or writing.

Annotating also data objects with application-based knowledge offers the runtime a comprehensive understanding of the interaction of code and data. Figure 4.1 illustrates such annotations. Based on information like the desired *isolation level*, the expected *access frequency*, and the predicted *read-to-write access ratio*, `MxTasking` will choose a matching synchronization mechanism for the data object. For instance, utilizing an optimistic primitive, as opposed to an exclusive latch, is a more convenient approach for synchronizing the *heavy-read* root node of a tree. Section 4.3 will provide a thorough discussion on choosing an appropriate mechanism. Upon selecting a synchronization

primitive, MxTasking guarantees to apply it to concurrent tasks. With the tasks on the one hand and the annotated data objects on the other, the worker can establish synchronization during the execution process.

4.2 Integrated Synchronization Primitives

MxTasking provides four basic synchronization primitives, sketched in the following. Some of them may be tunable, e.g., to distinguish between reading and writing accesses or not. A task itself is unaware of the primitive MxTasking will choose, i.e., the synchronization is decoupled from the task's implementation. However, the developer can request a particular primitive explicitly through annotations. If not forced in such a way, MxTasking will select among its built-in primitives at runtime, depending on the current system state and annotations.

4.2.1 Latches

Spinlocks are known for their straightforward realization and simple usage. Similar to the utilization in thread-based implementations, spinlocks can be applied to synchronize concurrent tasks. MxTasking provides different spinlock variants. For mutual exclusion, a simple spinlock can serialize all accesses, whether tasks are read-only or not. Given an application that requests parallel reads on a shared object, MxTasking chooses a reader/writer-lock instead.

In MxTasking, tasks are executed by a worker thread of the runtime system. The corresponding worker thread will acquire and release latches automatically on behalf of the executed task.

4.2.2 Optimistic Versioning

Latch-based protocols turn parallelism into concurrency. Especially for read-dominated workloads, this may unnecessarily limit the achievable parallelism and throughput. Here is where optimistic alternatives excel. The idea is to let read operations run in parallel and without synchronization. Only concurrent write accesses are protected from each other, e.g., by using latches. Conflicts between read and write accesses are allowed to occur. However, they are detected with the help of a *version counter*. More precisely, write accesses will increment the version counter after each modification. Reading operations, in turn, will check the version counter before and after they access the data object. By comparing both versions, MxTasking can identify concurrent writes and repeat the read operation until it is valid.

Optimistic mechanisms were shown to achieve better throughput on hardware with high degrees of parallelism (e.g., [102, 31, 94, 95]). In-memory index structures are an excellent example to illustrate the advantage of optimistic

protocols. Although all tree-related operations need to visit the root node, which may lead to high contention, modifications to the root node are rare. Optimistic synchronization mechanisms can mitigate the negative impact of latching overhead. Hence, the penalty of a repeated read operation arises only rarely. The positive effects of optimistic versioning are amplified on platforms (such as `MxTasking`) that possess knowledge of the underlying hardware cache locality. Pessimistic synchronization strategies inevitably depend on data, latches, or code to be exchanged between parallel units upon every access. In contrast, optimistic versioning enables the hardware to replicate and cache data structures for read-only accesses, thereby facilitating true parallelism. Hardware cache coherence protocols ensure that messages are sent between cores exactly and only when a true conflict arises. Note that write operations still have to be synchronized in optimistic schemes. In this sense, `MxTasking` will combine optimistic versioning with either a latch-based or a scheduling-based (see below) synchronization primitive. The worker thread handles optimistic versioning on behalf of the task that has requested synchronization. If the worker detects a version mismatch, the task is reset and re-executed until the execution is valid.

In this context, resetting a task involves only restoring the task's descriptor to its initial state before starting the read operation, which includes returning parameters and annotations to their previous values. However, the reset cannot revert changes made during the execution that affect the system, such as tasks spawned. Thus, tasks should spawn follow-up tasks (or apply other system-wide alterations) only when operating as a writer. In order to enable `MxTask`s to spawn other tasks, the interface allows returning follow-up tasks after executing. `MxTasking` will dispatch corresponding tasks only if the read operation is successful.

Implementation Details. Similar to other optimistic procedures, operations (or `MxTask`s in our context) that physically remove a shared object must be treated with special care. Due to parallelism, one task may read a data object while another frees the data physically. Consequently, the reading task will operate on garbaged data without realizing it, potentially leading to errors.

Reference counting, hazard pointers [106], and *epoch-based memory reclamation* (EBMR) [49] are general approaches for safeguarding read accesses in the presence of concurrent memory deallocation attempts. For performance reasons, `MxTasking` implements an EBMR similar to *Silo* [141] and the decentral procedure realized by the open BwTree [147]. When the application removes objects logically, the release of physical memory is deferred until all potential read operations have been completed. Time is generally separated into coarse-grained epochs using a universal epoch counter that increases at intervals (e.g., every 50 ms). All utilized threads provide a local epoch for detecting possible conflicts of concurrent read and delete operations. When

data objects are removed logically, they are marked as such and tagged with the current global epoch. The thread-local epochs, in turn, represent the relative progress to the global epoch. On entering a critical section, the thread synchronizes the local with the global epoch. After leaving the critical path, the worker resets its local value to infinity, indicating that the thread is not in a crucial execution state.

At the onset of a new epoch, a separated garbage collection procedure determines the minimal progress made by emphasizing the epoch with the lowest thread-local value. Data entities deleted logically during an even earlier epoch get safely reclaimed. Widely used implementations define *critical sections* as logical operations that include optimistic reads (e.g., a tree insert including the traversal). Consequently, the local epoch is updated at the beginning of such an operation and reset afterward. By employing fine-grained tasks, logical operations are fragmented into several work units executed by several worker threads. Hence, it is ambiguous to determine the beginning and end of a logical operation processed by several tasks. A similar scenario occurs when using coroutines [63], which also possess asynchronous properties. Wrapping local epoch updates around the execution of every `MxTask` leads to a considerable number of fenced memory loads and stores. To avoid this potential inefficiency, `MxTasking` updates the local to the global epoch after a limited number of executed tasks (and also when idling to guarantee progress). The chosen threshold becomes a trade-off between the maximum performance and the delay in releasing logically freed memory. In our implementation, we keep the number as small as possible without suffering from performance losses (e.g., 50). For garbage collection of finally unused memory, `MxTasking` spawns corresponding tasks at the beginning of a new epoch.

4.2.3 Hardware Transactional Memory

Although HTM appears user-friendly at first glance, the practical application is considerably more complex. As transactions might fail, the developer has to specify a fallback path using further synchronization (such as latches) to guarantee that (HTM-) synchronized code makes progress [93, 62]. In addition, HTM support is restricted to a particular range of hardware.

To streamline access to hardware-assisted synchronization, `MxTasking` integrates HTM as a potential primitive to synchronize concurrent tasks automatically. On the application's behalf, the worker wraps hardware transactions around task execution using appropriate CPU instructions. If a transaction repeatedly fails, the worker moves to a fallback mode utilizing a traditional spinlock. Consequently, the burden of handling implementation details falls on the runtime instead of the application developers; they only need to annotate the task appropriately. Plus, `MxTasking` will employ an alternative synchronization primitive as a fallback if the underlying hardware lacks support for HTM.

4.2.4 Synchronization through Scheduling

In addition to "classical" synchronization mechanisms such as latches or versioning, the task-based execution model of `MxTasking` enables another powerful synchronization mode: scheduling-based synchronization. As the simplest form of scheduling-based synchronization, `MxTasking` can guarantee that tasks accessing the same data object are executed sequentially. Such a guarantee is easy to make: `MxTasking` will schedule tasks that access the same data object to the same task pool. Tasks within one pool are executed in order by a worker thread linked with that pool. Active waiting for resources to become obtainable and contention can be avoided. In addition to mitigating issues related to concurrency, scheduling-based synchronization can offer advantages, particularly in NUMA settings. Instead of moving data objects between NUMA nodes, scheduling-based synchronization effectively moves code to data. Systems like DORA [119] and H-Store [75] have demonstrated that similar principles can enhance cache locality and transaction throughput.

To that end, `MxTasking` maintains a record of allocated data objects and links each data object to a specific worker. Whenever the application spawns a task scheduled by synchronization, the dispatcher transfers the task to the designated worker's pool. Our prototype implements a round-robin scheduling algorithm to allocate data objects to worker threads, achieving a balanced load distribution within the system. However, data objects may exhibit varying levels of access frequency. For instance, the root of a tree is typically visited more frequently than a leaf node. To ensure equitable workload distribution among the workers, the dispatcher also considers the *access frequency* annotation of a given data object at its creation. Accordingly, workers once associated with frequently accessed data are considered less often for subsequent assignments.

Scheduling-based synchronization can be made more general by annotating dependencies between tasks (in the spirit of [22]). To illustrate, in a task-based hash join implementation, the probe tasks must start after all build tasks have completed populating the in-memory hash table. We will discuss this in more detail later in Chapter 7.

4.3 Selecting Synchronization Primitives

For injecting synchronization, the runtime applies one of the described primitives to every newly created data object that requests isolated access. The developer can specify this primitive through explicit annotations or leave this to `MxTasking`. In the latter case, the runtime uses a cost model considering specified annotations detailing the access properties. The factors considered include the isolation level, expected access frequency, and assumed read-write ratio.

When requesting exclusive access to an object, serialization is guaranteed

by using HTM, if supported by the underlying hardware, or through scheduling. We observed in our benchmarks that scheduling exhibits superior performance compared to spinlocks. Employing a less stringent form of isolation that allows for parallel reads aligns with an optimistic synchronization strategy. Based on the (by the developer) anticipated and annotated read-write ratio, `MxTasking` uses scheduling techniques to prioritize writing tasks in scenarios where reading operations predominate. Linking a specific task pool to data objects for writing operations ensures that read-only tasks executed by the same worker will succeed without the need for version checks or saving the task's state. Given written-heavy resources that are accessed infrequently or in moderation, the primary challenge lies in managing scheduling overhead, primarily caused by competition for access to a shared task pool. Since latch-contention on infrequently accessed data rarely arises, but dispatching overhead is invariable, the latter tends to be a more notable issue. Hence, `MxTasking` prefers optimistic latches for such resources.

As an illustration, the synchronization of nodes in a task-based tree structure is achieved through various optimistic primitives. In general, operations typically access nodes at higher levels in a read-only manner; leaf nodes, in turn, are visited less often but are subject to more frequent modifications. Thus, `MxTasking` determines optimistic scheduling for inner nodes and optimistic latches for leaf nodes, assuming appropriate annotations by the developer.

4.4 Dispatcher/Worker Interaction

Figure 4.2 illustrates how the synchronization of tasks is based on the interaction of the dispatcher and worker thread. The dispatcher guarantees the allocation of `MxTask`s to the corresponding worker thread's pool based on the access type and synchronization mechanism. The worker, in turn, applies synchronization primitives whenever needed.

4.4.1 The Dispatcher Side

The dispatcher assists the synchronization process by placing a set of tasks to be serialized in the same task pool. This becomes necessary (a) when optimistic scheduling is applied and the task modifies the annotated object or (b) when all interactions with a data object are synchronized through scheduling. In both instances, the dispatcher chooses the task pool linked to the annotated data object as a destination (lines 1–3). The task pool associated with the object is encoded within the object and specified upon the object's instantiation.

Utilizing a global data structure, such as a hash table, for association suffers from cache misses, as the structure can expand significantly. To avoid cache misses when querying the data object for its associated task pool, `MxTasking` utilizes *pointer tagging* (in the spirit of [115] and [21]). The task pool's identifier

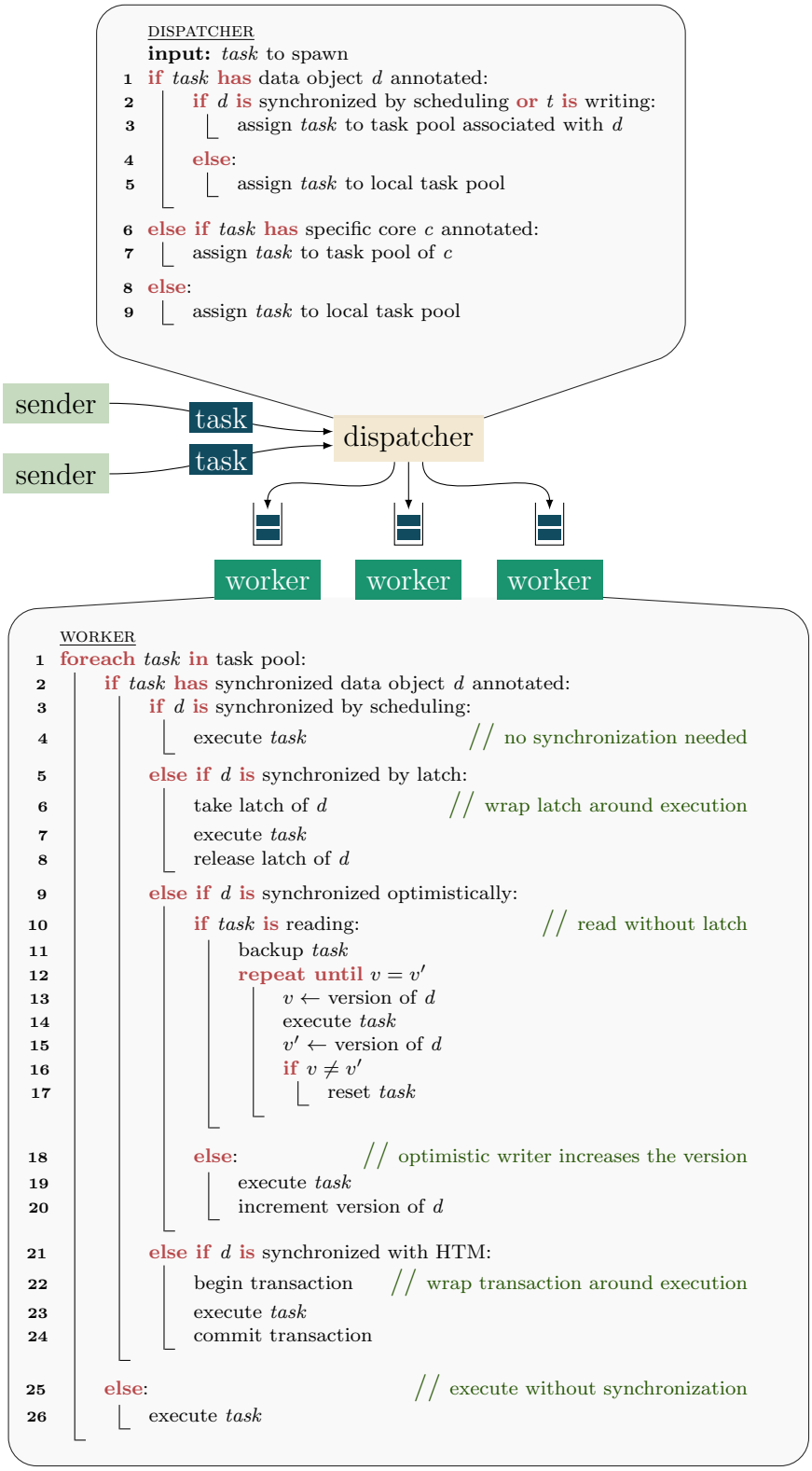


Figure 4.2: Interaction between the dispatcher and the worker to realize synchronization of tasks. The dispatcher places tasks in a specific task pool, depending on the synchronization primitive. The worker applies synchronization around the task execution.

is inscribed within the object's pointer yielded by the memory allocator. By that, the dispatcher can identify the associated pool using lightweight *shift* instructions.

Suppose the data object is not synchronized by scheduling, and the task will not write to the data. In that case, the dispatcher prioritizes the local task pool to minimize dispatching overhead. In this context, local refers to the task pool of the worker thread spawning the task (potentially while executing another task). Exceptions of this rule are annotations that explicitly request the placement of `MxTask`s, for example, on a specific core (lines 6–7). Annotating particular NUMA regions to facilitate the development of NUMA-aware software is also conceivable.

4.4.2 The Worker Side

The worker thread is responsible for applying the synchronization primitive as needed, which is enveloped around the execution of a given task. Initially, the worker assesses the annotated data object and its synchronization requirements (line 2). Assuming synchronization is unnecessary, or scheduling already ensures sequential and exclusive access, the worker executes the task straightforwardly (lines 3–4 and 26). Otherwise, we distinguish between the three additional mechanisms we discussed before. If the accessed object is synchronized through a latch, the worker acquires the corresponding latch associated with the data object before executing the task. Subsequently, the worker releases it (lines 5–8). We acquire the latch in shared mode using reader/writer latches whenever possible.

In the light of optimistic versioning, the worker distinguishes between reading and writing tasks. To verify the integrity of a data object during a read-only operation, the worker conducts the version check before and after the execution (lines 10–17). If the versions mismatch, it is necessary to retry the read access. This also requires resetting tasks to their original state, as parameters may have been changed during execution (lines 16 and 17). Therefore, the worker saves the state before each optimistic read (line 11). To ensure that the version increases, the worker increments it after completing a writing task (line 20). In this illustration, we have considered the optimistic approach under the condition that writing tasks are serialized by scheduling. `MxTasking` implements yet another optimistic procedure in which write accesses are synchronized by classical latches. For writing tasks, the worker will acquire the latch. This makes scheduling appropriate tasks into a specific task pool unnecessary. Reading operations proceed as described, but additionally, check whether a writing task occupies the latch. If this is true, the read operation is aborted prematurely and subsequently repeated.

Upon selecting transactional memory as the synchronization mechanism, the worker initiates a new transaction before executing the task and commits it afterward (lines 21–24). Although we simplified the representation in this

illustration, in practice, multiple attempts are typically made to execute the transaction in case of failure. One potential instance involves concurrent accesses to the data object, with at least one being a write operation. If the hardware aborts the transaction after repeated tries, the worker will employ a spinlock to guarantee the task's execution. After successfully starting the transaction, the worker must ensure that the spinlock is available and not held by another worker thread. If the data object is locked, the transaction is aborted and repeated.

Part II

Leveraging Tasks for Data Structures

5 MxTasking in Action

*Parts of this chapter have been published in [**109**].*

Utilizing tasks to design data structures and algorithms generally differs from well-understood thread-based programming. Morsel-driven parallelism [92] and DORA [119] have already demonstrated the advantages of analytical- and transactional processing in a task-like fashion. `MxTasking` surpasses the existing task paradigm beyond the current standard by offering annotations to optimize and simplify the execution, e.g., by prefetching data and taking care of synchronization. This chapter provides pragmatic considerations for employing `MxTask`s to build parallel software. We illustrate the simplicity of designing latch-free, task-based data structures using a B^{link}-tree as a demonstrator.

5.1 Background

B-trees [18, 58] play a fundamental role as index structures for file systems (e.g., *BTRFS* [129]) and DBMSs. As such, B-trees have been extensively researched and refined with a focus on their caching behavior (e.g., [127, 60, 131]) and concurrency synchronization, leading to several B-tree variations. For instance, the B^+-tree [35] extends the B-tree by storing the payload exclusively within the leaf node layer to enable more efficient range queries. Inner nodes "only" guide a query for a specific key from the root toward the proper leaf node. The B^+-tree serves as the foundation for various synchronization-optimized variants, such as the B^{link}-tree [91], *Masstree* [102], *Bw-Tree* [96, 147], and *BtreeOLC* [95]. The B^{link}-tree focuses on reducing the number of simultaneously held latches. To this end, newly inserted nodes are not immediately linked to the parent, eliminating the necessity of retaining the parent's latch. Instead,

each node includes a pointer to its right sibling. A dedicated operation will place a link within the parent node. Until then, the node is accessible for parallel traverse operations through its left sibling. As a result, each logical operation is transformed into a series of numerous small steps, each associated with a single node. This pattern makes the B^{link}-tree match the task model, executing every step as a task.

5.2 Operations

Classical thread-based implementations employ iterative procedures to navigate from the root to a specific leaf node, sequentially accessing node by node in succession. In task-based systems, every logical procedure becomes a concatenation of multiple tiny steps, each of them related to a single node. Contrarily to threads, `MxTask`s (and similar methodologies, e.g., [128, 124, 63, 73]) are executed asynchronously. For instance, instead of invoking an insert method that returns upon completion, spawning an *insert task* that notifies the caller after successfully inserting is the way to go.

5.2.1 Insert Task

The pseudocode depicted in Figure 5.1 provides an example of an *insert task* implementation. The code implements the traversal of the tree (lines 1–15) and the insertion (line 17) within a single task segment, taking the processed node and the requested key-value pair as input parameters. During the tree traversal, each step involves an examination of the accessed node to determine whether it is an inner or a leaf node. If the node is of type inner, the task identifies the subsequent node to visit using binary search (line 7). However, parallel insert operations may have modified the node's content during the period between the spawning and execution of the task.

Sometimes, one of these insertions splits the node. At that point, a traversal operation may have failed to locate the direct pointer to the node containing the desired key. For that reason, each task verifies the key range of the accessed node and follows to the right sibling as needed (lines 2–5). This may also emerge in traditional (thread-based) implementations and is generally part of the B^{link}-tree algorithm. However, the probability of occurrence is higher in asynchronous models due to the extended time gap between node visits during a singular traversal, depending on the current set of tasks awaiting execution.

In order to proceed with the traversal, the task instantiates a new `MxTask` and spawns it, annotated with the subsequent node (e.g., lines 8–10). Labeling the new task as a reader (line 10) enables `MxTasking` to execute the task in parallel with other reading operations. In contrast, a thread-based implementation will invoke the child lookup iteratively in a loop until reaching a leaf node. Given that the task executes on a leaf, it inserts the item and notifies

INSERT TASK
input: *node* the task accesses, (*key*, *value*) to insert, *callback* to notify

```
1  if node->high_key() ≤ key:                    // key is out of range of this node
2      next = node->right_sibling()
3      task = mxtasking::new_task<InsertTask>(next, key, value)
4      task->annotate(next, mxtasking::readonly)
5      mxtasking::spawn(task)
6  else if node->type() == inner:                // continue traversal to the leaf
7      next = node->child(key)
8      task = mxtasking::new_task<InsertTask>(next, key, value)
9      task->annotate(next, mxtasking::readonly)
10     mxtasking::spawn(task)
11 else if node->type() == branch:    // child is a leaf; next task will insert
12     next = node->child(key)
13     task = mxtasking::new_task<InsertTask>(next, key, value)
14     task->annotate(next, mxtasking::write)
15     mxtasking::spawn(task)
16 else:                              // found correct leaf, insert value, and notify callback
17     node->insert(key, value)
18     callback->insertion_finished(key, value)
```

Figure 5.1: Pseudocode of an `MxTask`-based insert operation, inserting a key-value pair into a B^{link}-tree. Each task operates on a single node and spawns a new task for the next one.

the caller (lines 17 and 18). We use a callback function to respond to a client's request in an end-to-end setting. Another option would be to spawn a new follow-up task that handles the response.

5.2.2 Node Splits

In this instance, we ignored the case of node splitting for the sake of simplicity. Splitting becomes imperative if no capacity is available to insert an additional record. Given that case, the insert task creates a new node and moves half of the records. Subsequently, it spawns a follow-up task that places a reference to the newly created node in its parent. The node can be accessed until the connection is established by following the left neighbor's sibling pointer.

5.2.3 Beyond Insertion

Implementing the corresponding *update* and *lookup* tasks is straightforward. Instead of inserting the value into the leaf (line 17), the appropriate `MxTask`

modifies or reads the requested record. Consequently, found values are passed to the callback. In particular, we omit the "is writing"-annotation for lookups since all steps during the lookup perform read-only accesses.

However, both tasks access a specific node segment of a leaf to modify or read the value—contrarily to insertion tasks. If the node is not fully present in the cache, accessing the value results in a cache miss (we will see why in Section 5.4). Therefore, we split the update and lookup task into two steps. The first uses the binary search to identify the value's index, analogous to performing a lookup for the next child node to traverse. The second entails the lookup or update of the value to finalize the operation. Each step is performed as a separate task. This separation enables annotating the exact accessed segment of the node for prefetching, i.e., the cache line containing the value.

5.3 Annotation-based Synchronization

The code for the insert operation in Figure 5.1 comes without logic for synchronization. Instead, `MxTasking` injects synchronization at runtime based on annotations provided by the application. To make things work, we annotate the insert task as *read-only* during the traversal (lines 4 and 9) and as *write* whenever it might attempt to modify a node (line 14). Consequently, `MxTasking` will wrap a pre-selected synchronization primitive around the task execution based on the characteristics of the tree node.

To assist `MxTasking` in choosing a matching primitive, the B^{link}-tree passes two attributes to the runtime whenever a node is created due to a node split: The predicted access frequency and the expected read/write ratio. Both properties are mainly related to the type of the newly created node. We anticipate the root node to experience high access frequencies, as each request will access the root node to start its traversal. For inner- and leaf nodes, we expect medium to few accesses. And we expect the read/write ratio to be inverted: The frequency of writing operations is higher for leaf nodes, while inner nodes are primarily accessed in read-only mode during traversal and are modified only after node splits. By communicating this knowledge, `MxTasking` will prioritize optimistic synchronization mechanisms for nodes (logically) close to the root node.

However, annotating a task as a writer at the appropriate time presents a minor challenge. When labeling a task as writing too early, parallelism may decrease due to the serialization of accesses. Doing this too late, contrarily, requires re-annotating and re-scheduling the task, resulting in additional costs. In order to inhibit, it is necessary to ascertain whether the subsequent node is an inner or leaf node during the traversal, as modifications are mainly related to leaf nodes, aside from inserting references after node splits. The most straightforward way would be to query the type of the next node by accessing its header. However, this causes additional cache misses. Note that these cache

misses will not emerge during the traversal since `MxTasking` will prefetch the node a task accesses (see below). To avoid this costly aspect, we propose a new node type: *Branch* nodes represent inner nodes whose children are leaf nodes. Accordingly, we annotate the insert task preemptively as writing when the task reaches a branch node during traversal (lines 13–17). Tasks spawned to insert a new child pointer after a node split are always labeled as writers.

5.4 Annotation-based Prefetching

So far, annotating tree nodes has been discussed solely within the context of synchronization. In addition, `MxTasking` will exploit that annotation to bring the node into the cache prior to its access—reducing memory stalls while traversing the tree. However, software-based prefetching is a double-edged sword: memory access latencies can be reduced while increasing memory bandwidth utilization and imposing additional pressure on the CPU. Therefore, it is imperative to exercise caution when annotating the data to be prefetched.

In tree-based data structures, only specific node segments are accessed during execution. While a node's header, which includes metadata like the size, the type, and the latch variable, is accessed either way, not all keys and values are of interest in practice. Since we use binary search to locate a specific key—starting with the mid-key—it is not predictable which half of the keys will be examined further. Consequently, the application faces the dilemma of either prefetching the complete array of keys, despite the knowledge that at least one-half of it will remain unused, or speculating and tolerating cache misses. This becomes even more specific when accessing a value after locating the index. The application can hint `MxTasking` to preload all values, although only one will be accessed. Alternatively, it can tolerate a cache miss, given the unpredictability of the accessed value at the time of annotating.

Intermediate Evaluation

We will now take a concrete look at the impact of prefetching specific node segments on the overall performance of a lookup operation. To that end, we vary the prefetch annotation so that the entire node, only the keys, only a subset of the keys, or only the node header is prefetched. Figure 5.2 compares the CPU cycles for a single lookup within the B^{link}-tree. As a baseline, we use a lookup with disabled prefetching. The lookup operation (including the traversal) consumes many cycles, mainly due to memory stalls. When prefetching the entire node with its header, keys, and values, the cycles spent for the lookup are noticeably lower: only 540 cycles are required for the operation, compared to 2400 cycles without prefetching. This illustrates the efficacy of software prefetching in general.

However, the cycles spent executing prefetch instructions exceed the benefit notably, including effort for checking annotations and interpreting prefetch hints. As a result, prefetching all data of the node is perceptibly slower than accepting the cache misses.

Prefetching only the node's header already reduces the cycles spent for the traversal by 39 %, compared to not prefetching. This way, the lookup "only" suffers from cache misses for accessing the keys and the payload. The node's metadata, in turn, will be found in the cache. It is worth noting that prefetching also includes the descriptors of `MxTask`s. The sweet spot, however, lies in prefetching half of the keys. Cache misses are accepted when the searched key is located in the second half to keep the number of prefetch instructions reasonable.

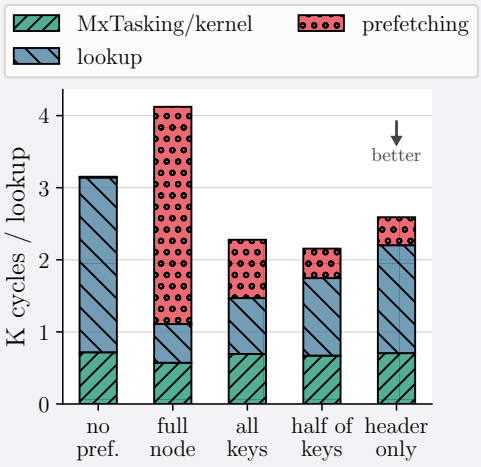


Figure 5.2: Prefetching different parts of a tree node during traversal and lookup.

The amount to prefetch per node also depends on the operation, e.g., insert or lookup. While we saw that, for lookups, prefetching the first half of the keys has the most benefit, this is not true for insert operations. Due to frequent data moves during (sorted) insertion, more extensive parts of the node are affected, making it advantageous to prefetch half of the node (which is equal to prefetching all keys).

5.5 Experimental Evaluation

To study the behavior, potential, and limits of `MxTasking` in real-world scenarios, we use an in-memory B^{link}-tree that characterizes the behavior of modern in-memory database engines. The implementation of the data structure follows state-of-the-art principles. We will use both read-heavy and write-heavy workloads and consider different metrics such as throughput, instructions, and stalled cycles to analyze the effect of annotation-driven prefetching, EBMR, and `MxTask`-performance in total.

5.5.1 Environment

System. All benchmarks are evaluated on 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 L1, 12 × 1 MB L2, and 1 × 19.25 MB L3 data caches. For all benchmarks showing an ascending number of CPU cores, the logical cores are ordered by NUMA regions, whereas the first 24 logical cores are located in the first region and the next 24 in the second. To be precise, the first 12 cores of each region are physical cores only. From then, hyperthreads are added step by step. Throughout the presentation of results, we will hyperthreads as "SMT" regions. In addition, we will emphasize NUMA borders with a dashed line.

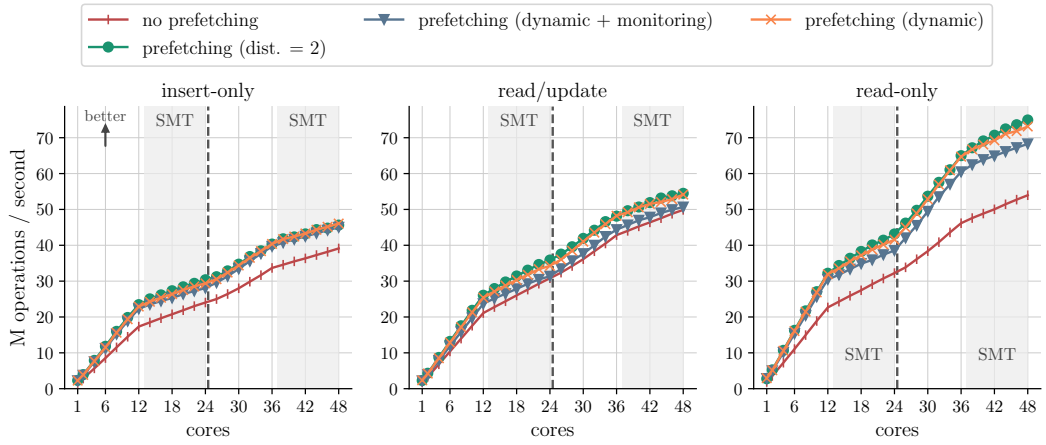
Ubuntu 20.04 is used as OS, *clang* 13.0.1 as the compiler, configured to apply optimization level *-O3*. Because all threads are pinned to corresponding cores, we disabled the system's NUMA balancing option for all experiments. This way, the kernel will not migrate memory or threads between the regions.

Workload. Following former work [147], we rely on the *Yahoo! Cloud Serving Benchmark* (YCSB) [36]. We use workloads *A* (read/update, 50/50) and *C* (read-only), with *Zipfian* distribution and 100 million operations. Before running both workloads, we initialize the tree with 100 million records and refer to that workload as „insert only" in the benchmarks. The tree stores pairs of 64 b keys and 64 b payloads within 1 kB-sized nodes, correlating to a fan-out of 59 children per inner node. For all implementations, we distribute the workload operations in batches of 500 requests at a time to (worker) threads. Whenever a thread finishes its assigned work, it picks the next batch. Our *pthread*-based implementations use an atomic integer to acquire work packages from a global list. Within task-based environments, we spawn one low-prioritized task per core that takes the next batch (like the thread-based implementations do) when almost no other task is ready for execution.

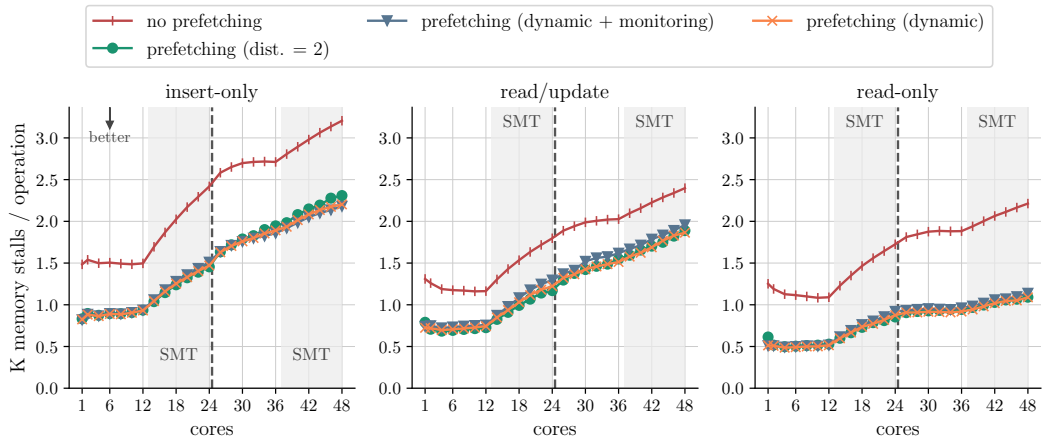
5.5.2 Annotation-based Prefetching

As discussed in Chapter 3, the fine granularity of tasks allows an accurate annotation of the data an `MxTask` will access. Sharing this knowledge enables `MxTasking` to prefetch that data from memory into CPU caches before the task executes. Figure 5.3 compares the B^{link}-tree built on top of `MxTasking` with and without annotation-based prefetching. We distinguish between two mechanisms: Prefetching with a static distance and dynamically injecting the prefetch distance, with and without monitoring.

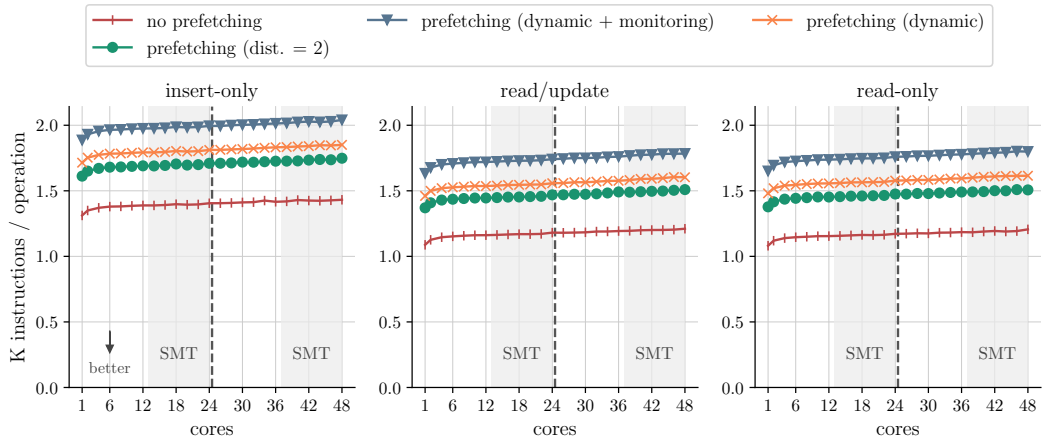
Static Prefetching. Experiments regarding the static prefetch distance indicated that the results behave as expected: If the interval is too small (e.g., 1 for prefetching the next task ready), the workload does not benefit from prefetching. Similarly, if the prefetch comes too late (more than 4 tasks apart), the advantage becomes smaller but is still noticeable. For the measurements



(a) Throughput



(b) Memory stalls per operation



(c) Retired instructions per operation

Figure 5.3: Effect of software-based prefetching for an `MxTask`-based B^{link}-tree, comparing static, dynamic, and no prefetching.

shown with a static prefetch distance, we specified a distance of 2, which performed best on our experimental analysis of the prefetch distance.

Since the benchmark is mostly bound by memory latency, annotation-based prefetching of tree nodes experiences a significant improvement. The throughput increases by 24 % on average for inserts, 16 % for mixed reads/updates, and 40 % for the read-only workload. We demonstrate the outcome in Figure 5.3a. Especially the traversal, implemented using binary search, is leveraged by software-based prefetching. As binary search creates a hard-to-predict access pattern for the CPU, hardware prefetching has a less beneficial impact. In update-heavy workloads, we observe that memory prefetching has a less noticeable effect. The reason is an increased latch contention caused by multiple tasks trying to acquire the latch for a leaf node concurrently.

The impact of software-based prefetching becomes evident when observing memory stalls, demonstrated in Figure 5.3b. Memory stalls refer to cycles where the CPU actively waits to complete data transfers from memory (or cache) into the register before it can continue execution. The prefetching mechanism of `MxTasking` reduces the number of those stalled cycles, resulting in increased throughput. This effect is, in particular, observable in read-only workloads. Here, the number of memory stalls is reduced by up to 51 %. The insert and read/update workloads also benefit with an average of 34 % and 40 % fewer memory stalls.

Figure 5.3c demonstrates the number of performed instructions per operation. Static prefetching utilizes approximately 300 additional instructions per operation compared to the non-prefetching execution. This comprises specific instructions necessary to initiate prefetching and instructions to schedule the prefetches, even if the latter is straightforward, as prefetches are placed two tasks ahead in the prefetch buffer. Nevertheless, these extra efforts reduce the memory stalled cycles to such an extent that prefetching still pays off.

Dynamic Prefetching. The static prefetch distance refers to a central configuration parameter tunable by the developer for all tasks simultaneously. Consequently, the effectiveness of prefetching is limited to tasks having a homogeneous runtime. In practice, however, this is rare, especially when several applications are running on top of the same `MxTasking` instance. While traversal and lookup tasks for the B^{link}-tree are approximately homogeneous, the execution times of insert operations differ. The task that finally inserts a key-value pair in the leaf node consumes significantly more CPU cycles than the traversal (according to the monitoring results, more than 2 $\times$). Moving data within a node or creating a new one is much more complex than loading a value for a lookup. As the measured B^{link}-tree has a depth of five, every fifth task has a significantly longer runtime in the insert-only workload.

To automatically inject the adjusted prefetch distance, `MxTasking` can monitor the tasks' execution times—or the developer can annotate the expected

execution time to tasks at compile time. In addition, instead of "mindlessly" inserting the prefetches at a statically determined offset in the prefetch buffer, the offset is calculated by exploiting the (monitored) execution times of preceding tasks in the task buffer. Both recording execution times and calculating the prefetch distance require additional effort, observable in the form of instructions. Figure 5.3c demonstrates the impact. On average, monitoring execution times and dynamically injecting the prefetch distance invokes 581 additional instructions per operation, compared to no prefetching. Utilizing annotations to hint to `MxTasking` about the execution time of tasks instead of monitoring[1] requires "only" 400 additional instructions.

Particularly for memory stalls, dynamic prefetching is more effective than static prefetching: Through the insert-heavy workload, dynamic prefetching causes 5 % fewer memory stalls (cf. Figure 5.3b). For mixed and read-heavy workloads, we can barely observe significant differences. This is to be expected since the workloads mostly show homogeneous execution times and static prefetching performs properly. The effect is also observable with regard to the throughput. However, the increased number of executed instructions, particularly for monitoring purposes, diminishes the throughput. A natural improvement would be to utilize the compiler (e.g., by implementing an appropriate plugin) and annotate statically predicted task execution times while generating the code.

Additionally, we used the average prefetch latency across both NUMA regions for prefetch scheduling using Intel MLC [1]. Distinguishing between local and remote latencies could improve accuracy. Due to the slightly better performance, we will perform further measurements with a static prefetch distance of 2.

5.5.3 Epoch-based Memory Reclamation

As discussed in Section 4.2, optimistic synchronization requires the coordination of concurrent physical removing and reading operations. To that end, `MxTasking` adapts widely used EBMR (e.g., [102, 141, 147]) to a task-based environment. Instead of wrapping local epoch updates around logical operations, an insert including the traverse, for instance, we focus on individual tasks. `MxTasking` implements two approaches: Synchronizing the local and global epoch before every task execution (resetting the local epoch afterward) and batching a limited number of tasks before aligning the local to the global counter. The results illustrated in Figure 5.4 prove that both mechanisms have little to no impact on performance, using no EBMR as a baseline. The most significant performance loss occurs during the execution of read-only workloads when wrapping local epoch updates around every `MxTask` execution. Write-

[1] We annotated the execution times we observed in a previous iteration with monitoring enabled.

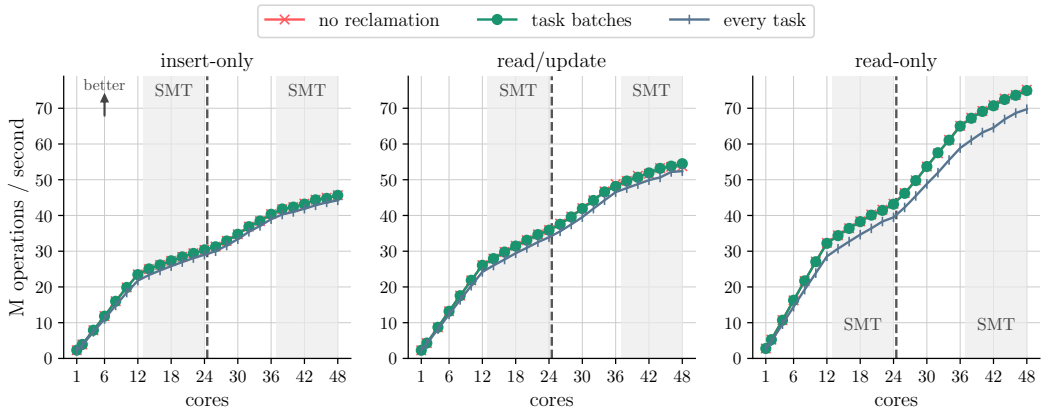


Figure 5.4: Scaling performance of EBMR in a task-based environment.

heavy workloads stay almost unaffected. Due to the slightly better performance, we will perform further measurements with batching-based EBMR.

5.5.4 Comparison of Tasks and Threads

We argue that `MxTask`s offer a superior abstraction level to easily build scalable software for modern and future many-core hardware—without decreasing performance. To study this hypothesis, we compare different programming models, libraries, and synchronization mechanisms: the B^{link}-tree on top of `MxTasking`, common threads, and Intel's TBB tasking library. Additionally, we apply proven state-of-the-art index structures, including latch-free ones. For the following benchmarks, we forced `MxTasking` to apply specific synchronization primitives using appropriate annotations.

Serialized Access. Spinlocks are widely used to serialize and, as a result, synchronize accesses to a specific resource. As discussed in Section 4.2, `MxTasking` supports synchronization by scheduling for exclusive accesses. Although it is well-known that serialization does not perform best for tree-like data structures, we discuss some insights into comparable scheduling-based synchronization. Figure 5.5 depicts the results, comparing the throughput of our B^{link}-tree implementation, using scheduling for `MxTask`s and spinlocks for TBB-tasks and threads. When utilizing TBB and threads, accesses to tree nodes are protected by spinlocks. In contrast, `MxTask`s accessing the same tree node are dispatched to the same task pool. Hence, tasks reading or writing the same tree node are implicitly serialized since `MxTask`s execute in a run-to-completion semantic.

The results demonstrate that the scheduling-based synchronization offers a significantly better performance related to spinlocks until utilizing multiple hardware threads per physical core (from 13 logical cores) and the second

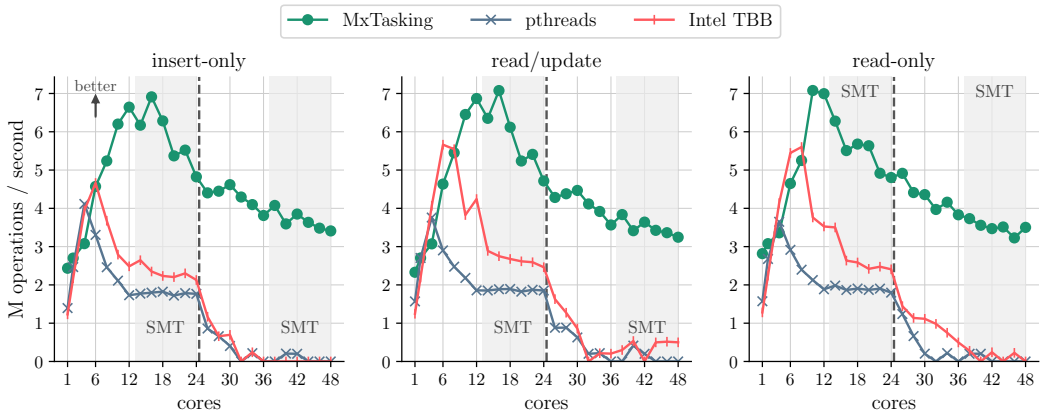


Figure 5.5: Throughput when utilizing serialized synchronization for the B^{link}-tree. We use spinlocks for threads and TBB and scheduling for `MxTask`s.

NUMA region (from 25 logical cores). Although `MxTask`s avoids latching, we can observe two bottlenecks preventing this approach from scaling. First, every operation starts by reading the tree's root node. Regardless of subsequent steps operating in parallel by distributing tasks to further tree nodes and implicit additional CPU cores, the inherently sequential access to the root limits parallelism and throughput. This also applies to spinlocks. Secondly, moving tasks to task pools of other cores involves overhead. Even if the operation is atomic, the expense of cache-coherence can degrade performance. This affects, in particular, the task pool associated with the root node since many producers try to (atomically) dispatch tasks concurrently. We observe a similar effect using latches when many threads (or TBB tasks) access the same cache line to acquire the root's latch.

Reader/Writer-locks. Figure 5.6 demonstrates the results using reader/-writer-locks as the synchronization primitive. This way, `MxTask`s are primarily dispatched to the local worker, minimizing frequent spawn requests hitting a single task pool. And, enabling parallel read operations, the root node stops constituting the bottleneck.

Balancing the load in this fashion turned out to be a straightforward and effective strategy for the given workload. However, as soon as cores in both NUMA regions are utilized, the throughput also decreases. In this case, the additional effort for keeping the latch variable coherent has a negative effect and causes communication costs across the sockets. We obtain similar results when using threads. Due to the built-in prefetching mechanism of `MxTasking`, we can observe a benefit of up to 36 % more lookups per second compared to threads. For both implementations, we borrowed the reader/writer-lock from Facebook's open-source *folly* library [47]. Contrarily, TBB provides different synchronization mechanisms, partially based on HTM. Applying the HTM-

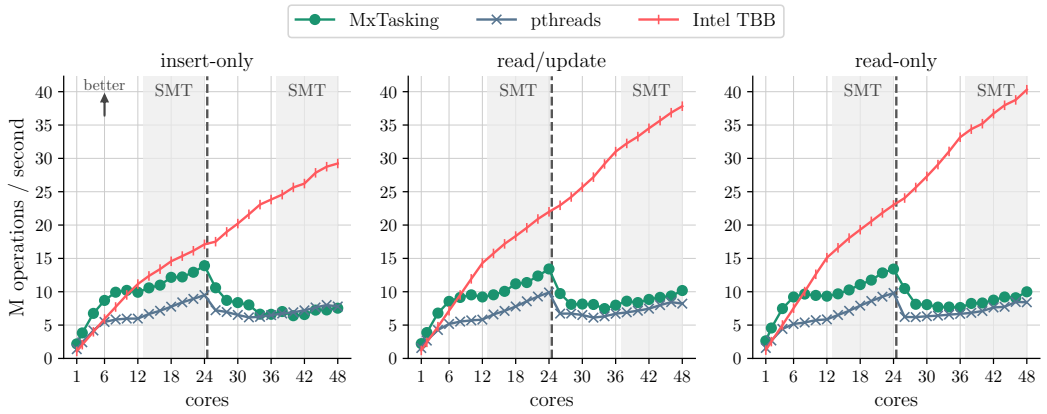


Figure 5.6: Utilizing reader/writer-locks for synchronizing operations accessing a B^{link}-tree.

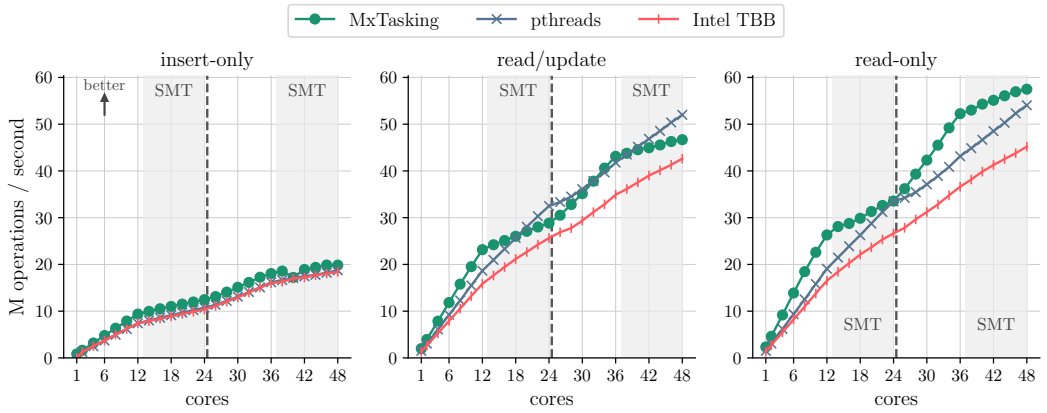


Figure 5.7: Utilizing HTM as a synchronization primitive for operations accessing a B^{link}-tree.

based reader/writer-lock to TBB, we notice less overhead due to latching and implicit better performance: more than 2.5 × compared to `MxTasking` and 3.5 × to threads.

Hardware Transactional Memory. Using HTM to synchronize concurrent tasks delegates synchronization to the underlying hardware. Previous work (e.g., [42, 93, 99]) has found that the usage of HTM in tree-based (and thread-based) data structures and beyond is straightforward and efficient. `MxTasking` encapsulates the execution of tasks within a transaction. Similar to reader/writer locks, tasks are mostly dispatched locally, which lowers worker communication but may increase write contention (and, in the light of HTM, transaction aborts).

Figure 5.7 depicts the results, comparing `MxTasking`-, TBB-, and thread-based B^{link}-tree implementations synchronized with HTM. While recurring

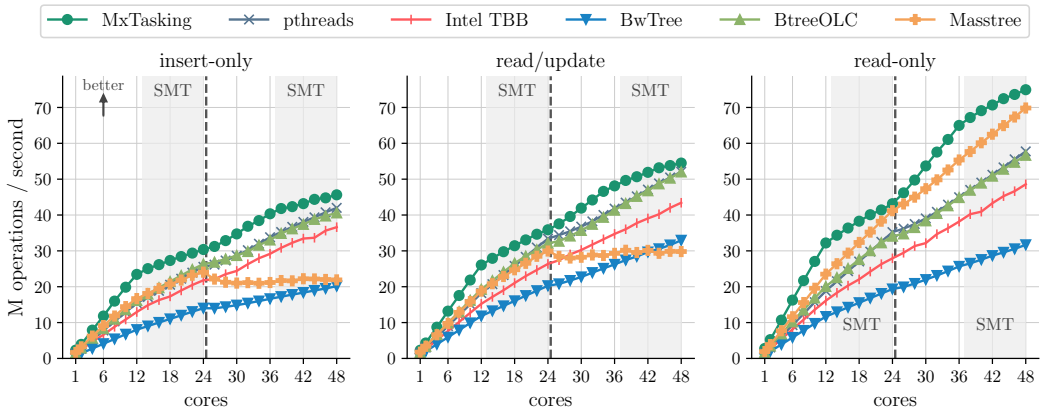


Figure 5.8: Optimistically synchronized B^{link}-tree and state-of-the-art index structures.

transaction aborts lead to performance penalties in write-heavy workloads, `MxTask`s benefit from automated prefetching in read-heavy workloads. During the read-only workload, it becomes evident that the hardware "forgives" frequent cache misses thanks to hyperthreading: With 12 cores, `MxTasking` achieves almost 1.4 × more lookup throughput than threads and TBB. The difference decreases as the number of logical cores increases, up to the NUMA boundary, from where a similar pattern is repeated.

Optimistic Synchronization. `MxTasking` utilizes optimistic synchronization by providing versioned data objects and differentiating between reading and modifying tasks. Read-only annotated `MxTask`s perform optimistically and ensure the validity of the operation afterward. Modifications, in contrast, are synchronized by the runtime[2]. Most executed tasks in the B^{link}-tree benchmark are read-only, including the traverse needed by insert and update operations to navigate to a leaf node.

Using threads and TBB as a fundament, optimistic synchronization requires careful implementation on top of the application. This also applies to state-of-the-art data structures like the open BwTree [147], Masstree [102], and BtreeOLC [95]. Figure 5.8 compares our optimistic-synchronized B^{link}-tree implementations and the named data structures. The state-of-the-art implementations are borrowed from the *index-microbench* framework [146]. The `MxTask`s-based B^{link}-tree achieves the highest throughput throughout the insert-only workload, 8.5 % more than its thread-based implementation and BtreeOLC. The B^{link}-tree implementation based on TBB is comparable to the results of Masstree but also scales beyond the first NUMA region. It is worth

[2]In practice, `MxTasking` mixes two variants of optimistic synchronization that differ in synchronization of writing tasks: writer-scheduling for read-heavy data objects, e.g., inner nodes, and applying latches for write-heavy objects, e.g., leaf nodes (cf. Section 4.4).

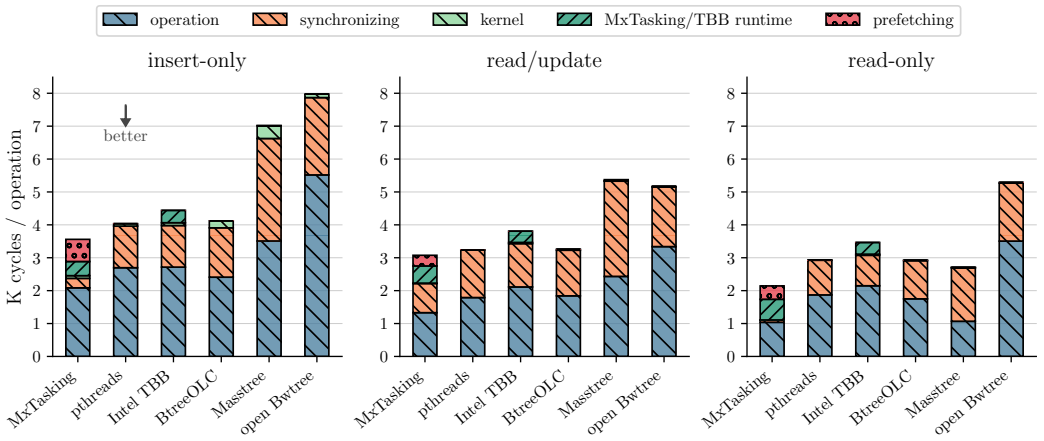


Figure 5.9: Detailed, cycle-based analysis of the B^{link}-tree (using `MxTasking`, threads, and TBB) and state-of-the-art index structures.

noting that BtreeOLC does not provide memory reclamation contrarily to all other evaluated data structures.

Also, for the mixed read/update workload, `MxTasking` performs best. However, with adding more logical cores to the benchmark, the throughput does not increase at a uniform rate. This is due to the higher contention on latch variables, particularly when modifying a node's values. On average, the `MxTask`-based B^{link}-tree executes 15 % more operations than the thread-based variant but only 4.7 % when utilizing all 48 cores. As observed within the insert-only workload, Masstree scales until using the second NUMA region.

The most significant differences are noticeable in the read-only workload. `MxTask`s accomplish 75 million lookups per second, 7 % more than Masstree (69.8 M), utilizing all available cores. Note that both implementations benefit from prefetching. The thread-based B^{link}-tree and BtreeOLC provide a similar throughput (roughly 57 million read operations per second), proving that our B^{link}-tree implementation meets the state-of-the-art. All measurements show that TBB suffers from the additional effort caused by the runtime environment.

Cycle-based Comparison. Discovering the reasons for varying results, Figure 5.9 shows a cycle-accurate comparison between the task- and thread-based implementations. `MxTasking`, Intel TBB, and pthread are related to the B^{link}-tree. We distinguish between effort for performing the insert/update/lookup operation, including the traversal, cycles spent in kernel mode (e.g., syscalls), and synchronization, including memory reclamation. Additionally, we present cycles consumed by the runtimes of `MxTasking` and Intel TBB. For `MxTasking`, we further show the complexity of initiating memory prefetches. We recorded those details using Intel VTune [69] and perf [97] for analyzing the results. To be precise, the aggregated cycles only give an impression. Mapping

spent cycles to function names is sometimes ambiguous, e.g., due to inlining. One specific example is the prefetching mechanism of Masstree: Although we know from the source code that prefetching instructions are executed and the measurement results also suggest this, we could not always observe the corresponding instructions through profiling tools.

The results prove the effectiveness of the prefetching mechanism used by `MxTasking`: Traversing the tree and performing the operation requires fewer cycles when applying `MxTask`s, compared to threads and TBB tasks. This is especially true for lookups, which we can also observe in the measurements of Masstree. Notably, prefetching during the insert-only benchmark consumes significantly more cycles than during the mixed and read-only phases. Since it is likely for insert operations to access the entire key set (e.g., to provide space for a new entry through moving keys), we hinted `MxTasking` to preload all keys. Consequently, more prefetch instructions are executed for insertion tasks.

In comparison to other data structures, `MxTask`s spent fewer cycles for synchronization, particularly during the read-only workload. Prefetching helps to reduce these costs by loading the header of tree nodes that include the version counter and latch variable. However, during the mixed read/update phase, there is an additional overhead due to contention on the latch. As multiple worker threads concurrently modify the latch's cache line (and the cache coherency protocol invalidates obsolete copies), prefetching cannot prevent these memory stalls.

By prefetching, `MxTasking` pays off the overhead coming along with tasks, mainly caused by task-spawning and managing annotations. Comparable runtime overhead is also observable for the TBB scheduler. We cannot break down these cycles more precisely since the profiler and/or the library do not provide revealing function names. However, we assume this is an expense for load balancing, task-stealing, and scheduling.

The results confirm that the abstraction of tasks and simplifying synchronization and prefetching implemented in `MxTasking` do not cause substantial performance degradation. In particular, the software-controlled prefetching of data objects offers considerable increases in throughput. The findings of Masstree strongly support this assumption. Integrating prefetching manually into threads and TBB requires considerably more effort: The application engineer is likely forced to reorganize the data structure.

5.5.5 Summary

In the experimental evaluation, we examined the performance of `MxTasking` using a task-based B^{link}-tree as a demonstrator. The analysis explored various aspects of EBMR, software-based prefetching, and concurrency synchronization. By conducting a comparative analysis with Intel TBB as an alternative task implementation and state-of-the-art data structures, we observed that `MxTask`s

noticeably enhances the performance of our B^{link}-tree implementation, particularly for read-only workloads. The central aspect of performance optimization is prefetching, which reduces the number of cycles in which the CPU waits until data is loaded from memory into the registers. Through annotations, `MxTasking` can preload accessed nodes so that tasks will find them (partially) cached when executing. However, the mechanism must be used carefully to ensure that the additional pressure in the form of instructions does not take away its benefits. The experiments indicate that utilizing a static prefetch distance, which determines the number of tasks to initiate prefetching in advance, is satisfactory for homogeneous workloads. In the case of the slightly heterogeneous insert-only workload, where specific tasks exhibit a longer execution time, dynamic prefetching leads to a modest improvement despite the extra effort needed to adjust the prefetch distance. We conclude that more efficient prefetch instructions by the hardware could improve software-based prefetching. The `AVX512`-based prefetch instructions are a noteworthy illustration, as they can prefetch multiple offsets starting from a base address through a single instruction. Nevertheless, these instructions belong to the `AVX512PF` instruction set, implemented exclusively on Intel Xeon Phi processors [2].

Annotations also help to decouple the synchronization from the application logic. This results in a barely noticeable overhead and allows synchronization mechanisms to be tailored to the underlying hardware substrate. Consequently, synchronization needs not to be implemented by an `MxTask`-based application. Instead, the developer can delegate it to the execution unit as annotated synchronization requests, simplifying the implementation of concurrent applications. In summary, we found that `MxTasking` can leverage write-heavy workloads by up to 8.5 % and read-only workloads by up to 30 % compared to our thread-based implementation.

Part III

Exploiting Tasks at the System Layer

6 Micro Partitioning

*Parts of this chapter have been published in [**110**].*

Following our previous observation that annotations assist the application in bringing data proactively into caches, this chapter delves into how annotations leverage the design of cache-aware algorithms and data structures. With memory accesses becoming the bottleneck of data-intensive systems, applications must explicitly make use of caches to provide optimal performance. This is particularly challenging for hash tables because of their random and hard-to-predict access patterns, and led to the design of *hardware-conscious* algorithms. *Partitioning strategies* break data into pieces small enough to fit into fast CPU caches (e.g., to prepare for a *partitioned hash join*). The drawback of these algorithms is their complexity in terms of design, development, and maintenance. This has resulted in a literature debate on whether *hardware-oblivious* algorithms may be the better choice: They are easier to design, do not require an understanding of the underlying hardware characteristics, and may be more robust when data or hardware behave differently than expected [13, 71, 15].

6.1 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 [136, 24, 100, 13, 61]. 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 6.1 illustrates this approach.

However, the appropriate number of partitions must be chosen carefully.

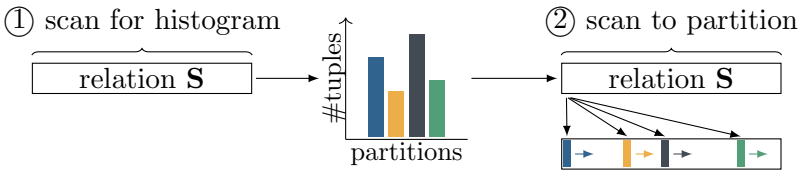


Figure 6.1: Data partitioning using a histogram.

On the one hand, the data must be divided into sufficiently small enough portions so that each partition leads to a cache-sized hash table. On the other hand, too many partitions lead to several logical memory addresses that exceed the capacity of the *translation lookaside buffer* (TLB)—which defines an upper limit on the number of efficiently writable partitions that can be handled simultaneously [24, 13].

More complex partitioning strategies have been developed in the literature as a remedy. *Radix partitioning* [100, 148, 13, 61, 123, 133] performs the task in multiple rounds, in each round keeping the partitioning fan-out below the limits defined by TLBs. *Software-managed buffers* [132, 14, 133] 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.

6.2 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* [92] as used in the engines of HyPer [114] and Umbra [115, 144].

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

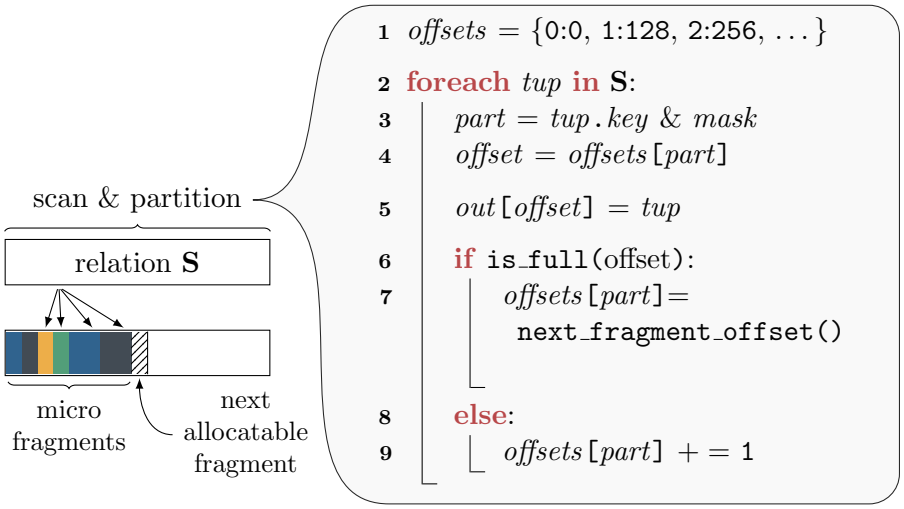


Figure 6.2: Materializing data into micro fragments of limited capacity.

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 counter multiplied by the fragment capacity to derive its offset. We illustrate our approach in Figure 6.2 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 6.3 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 contemporary servers' large amounts of main memory.

The true strength of micro partitioning lies in the tightly spaced physical

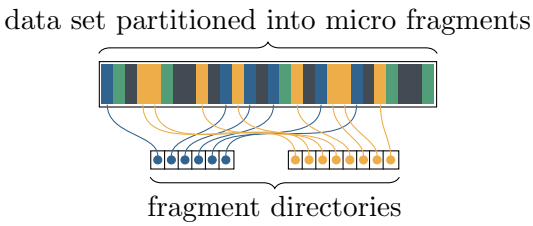


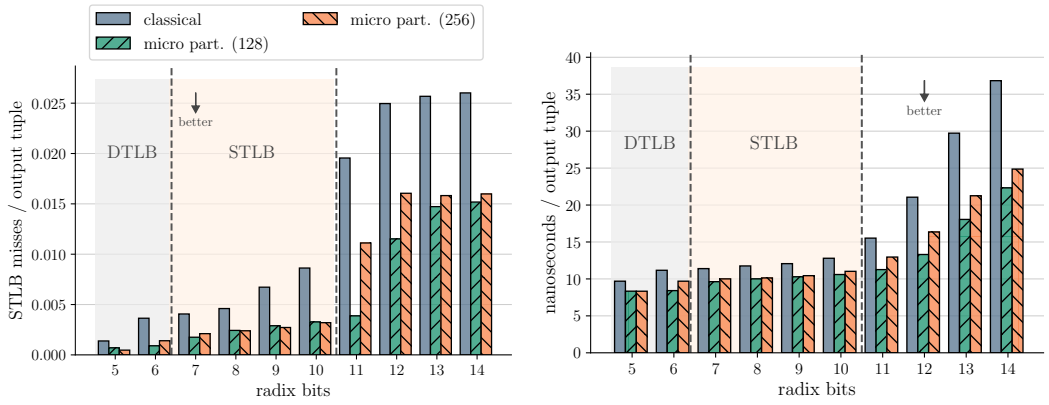
Figure 6.3: Bookkeeping of micro fragments and their partitions. For each logical partition, a *fragment directory* connects the fragments that form the *partition*.

layout. Given the small size of each micro fragment, multiple fragments 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 6.4. 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.

6.3 Micro Partitioning in Action

To understand the performance possibilities of micro partitioning, we compare two 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. [13], 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 "entire" radix join will be addressed in Section 6.4). 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 fragments 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 provided in Section 5.5).

Figure 6.4 demonstrates the results. While classical partitioning leads to a comparatively large number of STLB misses with only a few partitions (Figure 6.4a), our approach with 128-tuple-wide micro fragments benefits from the tight partition layout in memory. This becomes notably evident when moving



(a) STLB misses (b) Execution time

Figure 6.4: Micro benchmark comparing state-of-the-art ("classical") and micro partitioning with capacities of 128 and 256 tuples.

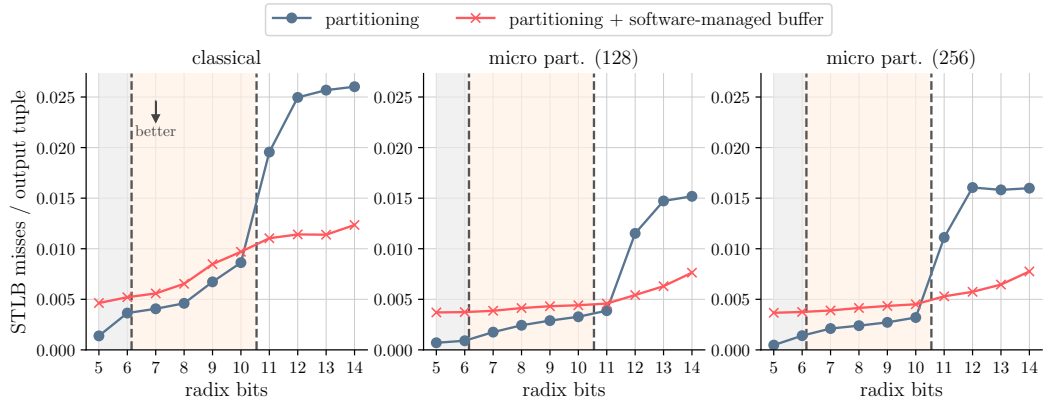


Figure 6.5: Comparing STLB misses during partitioning with and without software-managed buffers.

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. 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 is also reflected in the execution time (see Figure 6.4b). 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) yielded 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 6.4, however, a large number of partitions increases the amount 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 6.5 compares 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).

6.4 Dispatching Micro Fragments

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 logical partition over several dispersed locations, which must be glued together for further processing.

Section 6.2 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 6.3 literally and maintain an explicit fragment directory that associates a partition with all its fragments. However, this strategy has two drawbacks: First, the resultant code grows more complicated and must be tailor-made. Second, 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.

6.4.1 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` runtime) `MxTask`s that process fragments from the same logical partition.

When multiple tasks should access the same data structure in succession, the application developer requests a task squad from `MxTasking`. 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

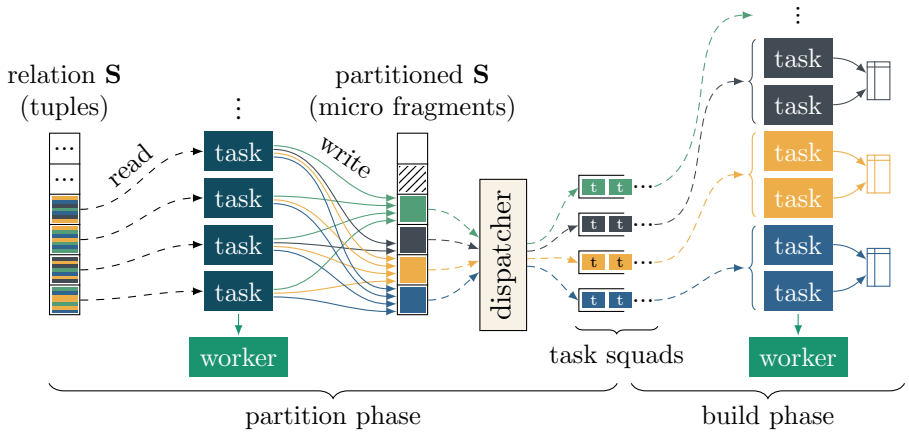


Figure 6.6: Illustration of the partition and build phases. During the partition phase, tasks materialize tuples into micro fragments and spawn tasks if a fragment is at capacity. Annotating appropriate tasks with task squads helps the dispatcher to execute tasks partition-wise.

suitably annotated tasks. Whenever the dispatcher finds a task annotated with a squad, the task will be sent to the squad's queue instead of a worker's task pool. 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 6.6. 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 6.7 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 empower `MxTasking`'s built-in prefetching mechanism. To keep matters simple, we statically set the prefetch size to 1 kB. While annotating is optional, we observed that the software-based prefetching mechanism built into `MxTasking` complements hardware prefetching effectively, giving the latter sufficient time to recognize the access pattern.

The annotation mechanism of `MxTasking` enables us to achieve two objectives simultaneously: 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

```
   // 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  foreach 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 6.7: 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.

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.

6.4.2 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 task pool 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 6.7).

6.4.3 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.

6.5 Experimental Evaluation

We perform the experimental evaluation using the machine described in Section 5.5: A two-socket machine with 24 physical and 48 logical cores in total. The data TLB has a capacity of 64 and the STLB of 1 536 entries.

We implemented a parallel task-based radix join to evaluate micro partitioning in a database-related context. For the benchmark, we follow former work [13], joining two relations with 16 B-sized tuples (8 B key and payload each). With 4 GB, we maintain the probe relation steady throughout the benchmark. The build relation, however, varies with the number of partitions

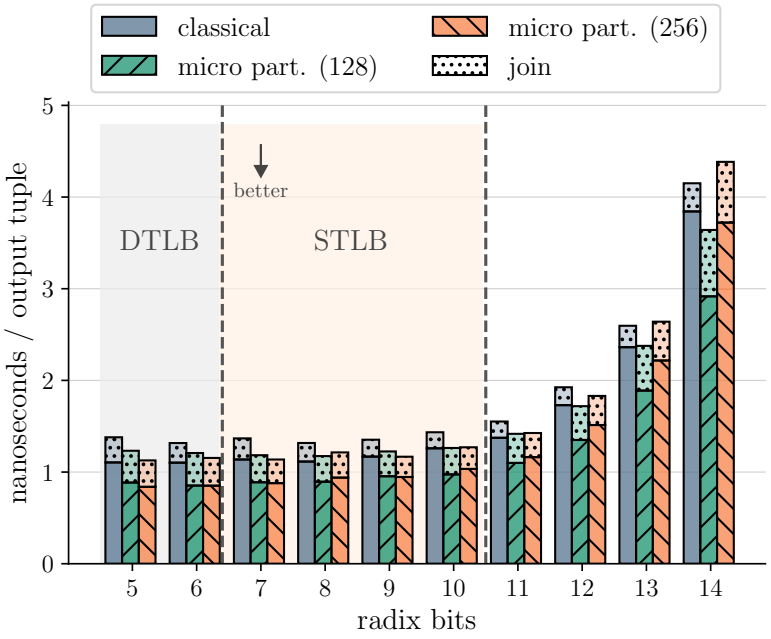


Figure 6.8: Comparison of the hardware-conscious radix join [13] using state-of-the-art partitioning and our task-based implementation using micro partitioning.

so that every partition allocates the entire L2 cache. This is equivalent to the original *Workload A* of [13] 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. [13] for comparison. Each of the following benchmarks utilizes all 48 available logical cores.

6.5.1 Comparison with the State of the Art

First, we compare the state-of-the-art radix-join implementation [13] with our micro partitioning- and task-driven join. We demonstrate the results in Figure 6.8, analyzing partition granularities of 128 and 256 tuples. Similar to our previous analysis of the partitioning stage (from Section 6.3), 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. First, the micro partitioning-based join had to be adapted: In the original implementation, only indexes of the materialized tuples instead of the tuples' data are stored in the table. The partitioned arrays are accessed during the 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.

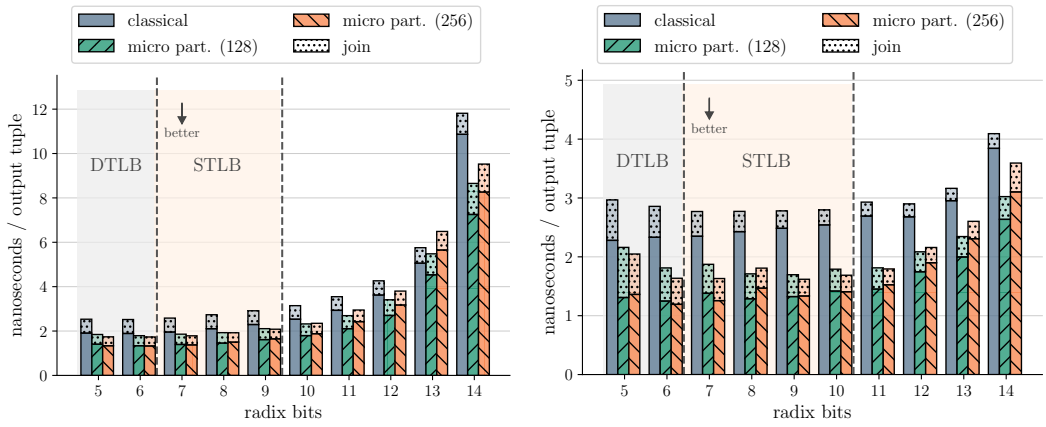
Second, each executed task scans a small subset of tuples for further processing (corresponding to the capacity of fragments). Using profiling tools (Intel VTune [69] and perf [97]), we discovered that the hardware prefetcher performs better on extended linear scans, which are given for coherent partitions.

Analyzing additional comparative measures (e.g., performance counters as in Section 6.3) 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 considerably impacts the measurements—because this occurs when there is temporarily no work. However, while reading performance counters, we cannot separate counted events that emerged during idling and those that affect the join execution. 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 dTLB and 512 STLB entries, and 256 kB L2 cache per physical core
- AMD EPYC 7501 with 64/128 logical/physical cores, 64 dTLB and 1 024 STLB entries, and 512 kB L2 cache per physical core.

We illustrate the outcomes obtained from that hardware in Figure 6.9. The contrast between "classical" partitioning and micro partitioning 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.



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

Figure 6.9: Radix Join Benchmark (same as in Figure 6.8) executed on additional hardware.

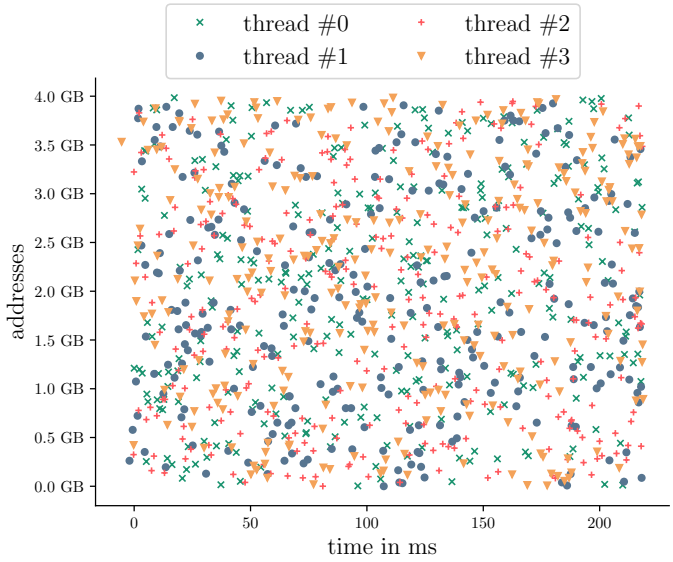
6.5.2 Memory Access Patterns

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.

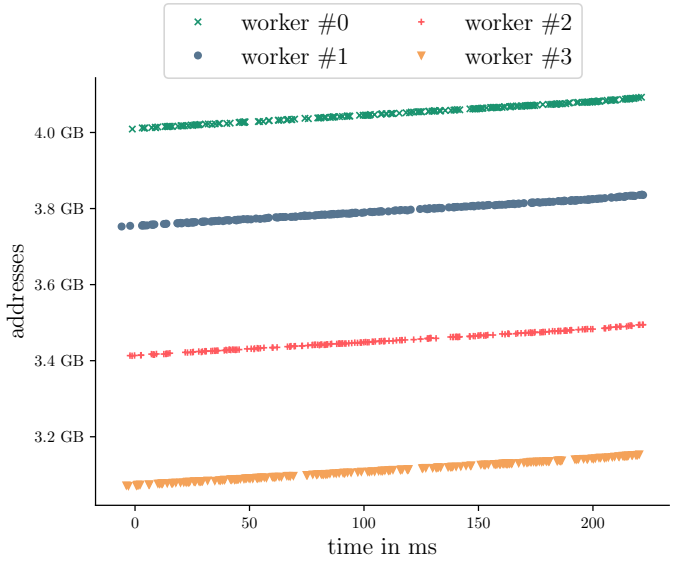
Memory Stores. Figure 6.10 depicts the chronological order of memory addresses written during the partitioning of the probe relation. We illustrate only 4 of the 48 threads as examples for visual clarity. The results support our argument that micro partitioning provides a more TLB-friendly write pattern. The "classical" technique of radix partitioning materializes tuples by writing them extensively throughout the partition array (Figure 6.10a)—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 6.10b). The write operations to memory regions near one another result in fewer simultaneously accessed memory pages, making better use of the TLB.

Memory Loads. When utilizing micro partitioning, a logical partition comprises many fragments that spread across the memory and are assembled during the join phase. This results in a random read pattern, as seen in Figure 6.11b, similar to the write pattern of radix partitioning. However, annotated

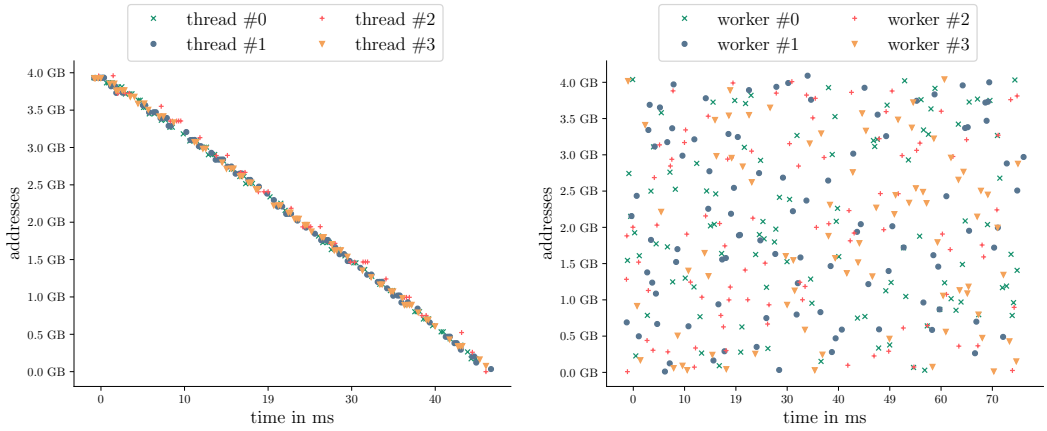


(a) Partitiong phase of [13]



(b) Partitioning phase using micro partitioning

Figure 6.10: 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`).



(a) Partitiong phase of [13] (b) Using micro partitioning

Figure 6.11: Comparing memory loads during the probe phase.

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 from the developer.

In contrast, radix partitioning enables linear scanning of a contiguous memory chunk per partition during the join phases (Figure 6.11a). 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 partitioning phase.

6.5.3 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 6.12 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 [13].

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 frequently—worker idle

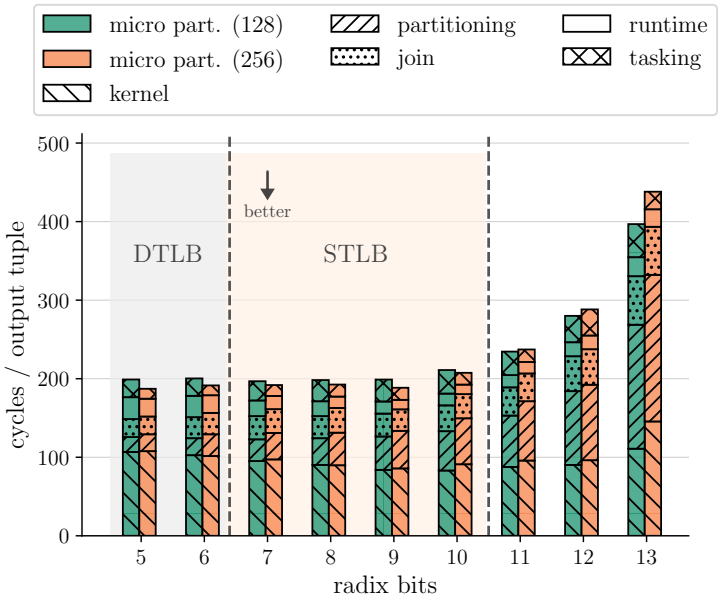


Figure 6.12: Cycle-based comparison of different micro fragment-granularities.

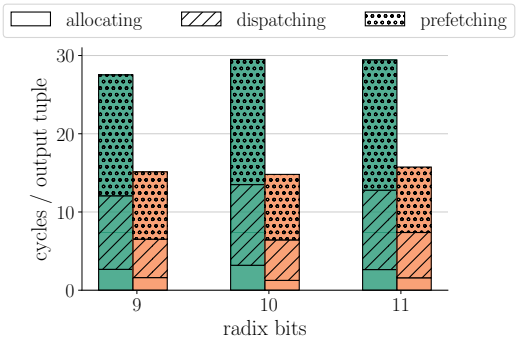


Figure 6.13: Breaking down *tasking* cycles in Figure 6.12.

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 6.13 shows, *tasking* includes costs for *allocating* and *dispatching* 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 implicitly, more data has to be prefetched by the software). 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—which we compensate with task-assisted software prefetching for shorter scans. 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.

6.5.4 Summary

In the experimental evaluation, we studied the performance of 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 `MxTask`s, 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 `MxTask`s 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 %.

7 Engineering a Task-based DBMS

So far, our exploration of the task-based processing model has primarily focused on its application in data structures and algorithms. The question of how `MxTask`s will integrate into a more intricate system remained open. In this chapter, we will address this challenge and delve into practical implications and benefits through an in-depth examination of our `MxTask`-based DBMS demonstrator, *TunaDB*.

Figure 7.1 provides an overview of TunaDB's operating principle. TunaDB accepts SQL queries and translates them into a sequence of tasks executed by `MxTasking` that work collaboratively to generate the desired query results. The query engine employs a sophisticated control flow abstraction to dynamically chain task-based operators, which will be discussed in Section 7.1. The tasks' code is compiled at runtime, leveraging *FlounderIR* [52, 53] to generate data-centric code. In Section 7.2, we illustrate query compilation and discuss optimizations related to FlounderIR.

Along with micro partitioning for (bloom-filtered [15, 89]) radix joins and grouped aggregations, TunaDB utilizes task annotations in two additional

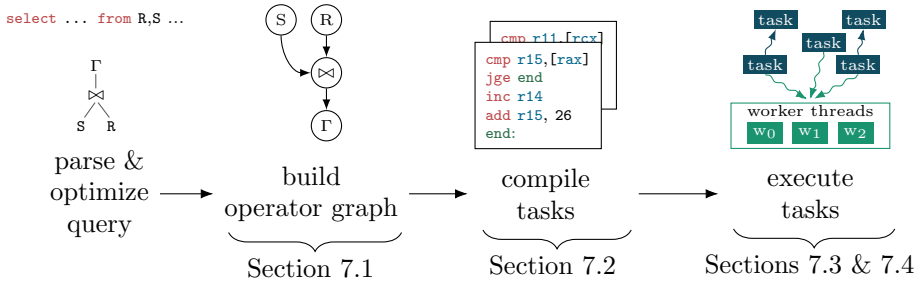


Figure 7.1: Overview of TunaDB.

ways. First, tasks are dispatched NUMA-consciously to access data locally whenever possible, particularly in the case of relation fragments. Second, TunaDB employs `MxTasking`'s prefetch mechanism so that tasks will find their accessed data fragments right in the cache, which we discuss in Section 7.3.

7.1 Control Flow Abstraction

The basic control flows, as implemented for the task-based B^{link}-tree (cf. Chapter 5) and micro partitioning (cf. Chapter 6), are not sufficient to utilize tasks for a more intricate query engine. In the B^{link}-tree, for instance, each task spawns a specific follow-up to continue the traversal or perform a specific operation, e.g., a lookup. A straightforward case distinction drives the decision on which task type to spawn as a follow-up; based on the type of the next node to be visited. However, in a query engine, control flow chains must be assembled at runtime as the operator sequence is unknown at compile time.

7.1.1 Control Flow of Query Engines

The *pull-based* iterator model [98] is a well-known approach for chaining and pipelining operators, implemented in various DBMSs, which became popular in the *Volcano* system [55, 57]. The control flow is transferred from operators to their children by invoking a `next()` function, allowing them to request ("pull") the next tuple for processing. Upon receiving a tuple, an operator will execute its code and return the result to its parent. This *tuple-at-a-time* model, however, leads to many function calls that the compiler cannot inline since the operator chain is first known at the query's planning time. By passing multiple tuples at a time through a Volcano-styled interface, *MonetDB/X100* [25, 153] reduces the number of function calls and allows vectorizing the operators' code.

Push-based query engines operate reversely in terms of their control flow. The evaluation of a query starts at the source operators of a query plan and propagates data toward the root operator. Specific operators (e.g., hash joins and aggregations) must materialize the data on the way up and have to be declared as *pipeline breakers*. The push-based paradigm was popularized by HyPer [114, 116] and has found significant application in other query compilation engines (e.g., *LegoBase* [82], *DBLAP* [135], and *ReSQL* [53]). However, the concept is not limited to compiling engines [140] and was found to perform similarly to the pull-based approach [134, 39].

7.1.2 MxTask-based Control and Data Flow

The push-based methodology aligns with the asynchronous nature of the task-based paradigm, making it a suitable choice for `MxTask`-based control and data flows. This methodology has also been employed by other task graph computing

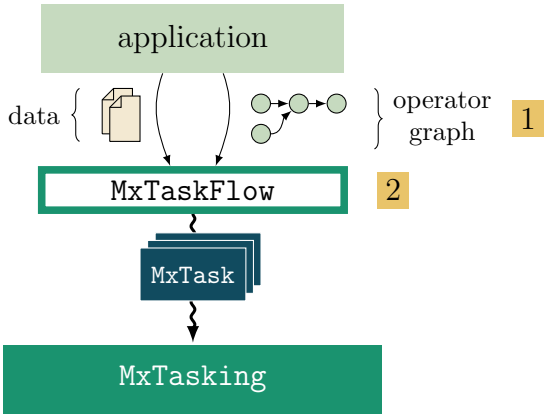


Figure 7.2: Interaction of the application and `MxTasking` using `MxTaskFlow`.

systems (e.g., [88, 9, 11, 74, 66]). In the context of a query engine, a task processes a data fragment and generates a follow-up task for further processing. However, translating a chain of operations into a set of tasks entails a couple of challenges. First, tasks that execute operators must know the type of follow-up tasks to spawn. Second, not all tasks are allowed to run simultaneously due to interdependencies among operators. For instance, all tasks of a hash join probe must wait for the build side to finish before proceeding. To ease the implementation of complex and task-based control flows, `MxTasking` provides an additional component that manages the execution of tasks for control flows specified at runtime.

An application expresses its control and data flow through an *operator graph* that is transmitted to `MxTaskFlow`, along with initial data to process. Figure 7.2 illustrates the interaction of the application and `MxTasking`. The operator graph (1) represents the relationship between operators, describing their chaining and interdependencies. Each node in the graph corresponds to an operator, i.e., a piece of code executed by tasks. The input data for each operator is either the output produced by its predecessor or a data fragment provided by the application for the initial operators, e.g., tuples structured in tables. When the application has specified its desired control flow and data to process, `MxTaskFlow` (2 in Figure 7.2) translates the graph into tasks. To start, `MxTaskFlow` spawns tasks executing the initial operators of the graph with the data provided by the application. The result of a task execution is passed to the subsequent operator for further processing by spawning an appropriate task. By monitoring the states of tasks and operators, `MxTaskFlow` manages dependencies, ensuring that no tasks are spawned that execute operators still awaiting the completion of another one.

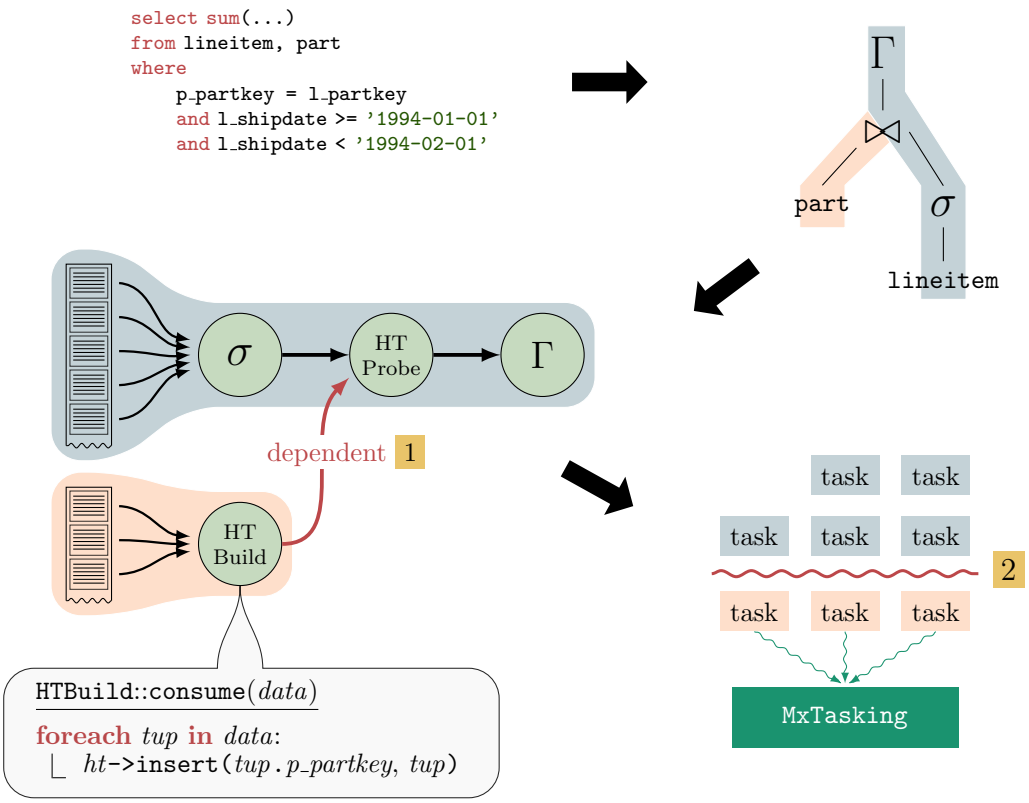


Figure 7.3: Illustration of the entire pipeline from receiving an SQL query to the execution of tasks.

7.1.3 Task-driven Query Processing

In this way, `MxTasking` decouples data and control flow from the operators' implementation. An operator task must only return the data to process for the subsequent operator. We will use relational query execution as an illustrative instance for `MxTaskFlow`; however, the control flow abstraction is not limited to this context and can also assist, for example, stream processing. Upon receiving a query, the DBMS translates it into an (optimized) logical query plan. To further transform the plan into an executable set of tasks, TunaDB maps the logical plan to a physical operator graph. The process is demonstrated in Figure 7.3, utilizing a hash join as an example. In the following, we will sketch various features of `MxTaskFlow` that assist query execution.

Dependency Management. The hash join, as an example, constrains the engine first to process all tasks of the build pipeline before tasks probing the hash table are allowed to execute. In order to propagate these relationships for `MxTaskFlow`, the application marks the corresponding dependencies between

operators in the graph. Beneath the surface, `MxTaskFlow` organizes operators into pipelines and establishes a dependency graph representing the constraints of pipelines as specified by the application. To address inter-pipeline dependencies, `MxTaskFlow` only spawns tasks executing operators whose dependencies are already fulfilled (or do not have any). In the example illustrated in Figure 7.3, only tasks associated with the `HTBuild` operator are spawned when starting the query execution (1). Once all tasks within a given pipeline have been processed, `MxTaskFlow` updates the dependency graph and propagates tasks linked to (now) independent pipelines to the `MxTasking` runtime.

As a bonus, this allows executing multiple, dependency-free pipelines in parallel, e.g., those building separate hash tables. However, from our observations, this only improves performance if the pipelines are small, each handled by a few workers. It is imperative to have sufficient pipelines ready for execution to keep workers (and consequently logical cores) busy, which limits the benefit to rare scenarios. This observation aligns with findings made in HyPer [92].

Task Tracking. Once all of a pipeline's tasks have been executed, `MxTaskFlow` can start the dependent pipelines (or send the query result to the requesting user). Nonetheless, determining the exact moment when all tasks within a pipeline have been completed is not readily discernible. Since tasks are dispatched to the system in a manner commonly referred to as "fire and forget," they are executed asynchronously at an undefined point in time. The straightforward way to determine would be to count and compare the number of both tasks spawned and executed. For example, Intel TBB uses atomic counters to track the execution status of child tasks [68]. However, this results in high pressure on the counter, frequently modified by multiple cores from multiple NUMA regions concurrently.

To address this issue, we leverage that tasks pushed to task pools are executed run-to-completion and in a first-in, first-out manner. By implication, at a task's execution time, all previously spawned tasks (dispatched to the same worker) have been executed. We exploit this by spawning one additional *barrier task* per operator and worker after all tasks of an operator have been published. By incrementing an atomic counter once per worker (and comparing it with the total number), the barrier task identifies the moment at which all workers have completed all tasks of an operator. This notably reduces the pressure on the counter, compared to incrementing per task. We highlight an appropriate barrier in Figure 7.3 (2).

Parallelism. Vertical parallelism, which refers to executing different operators simultaneously, becomes straightforward in an `MxTask`-based query engine; without the need for additional exchange operators used in the Volcano model [55, 56]. Once a task has been completed, the result can be streamed into the next operator. If there are no dependencies to other pipelines, a

corresponding task can be spawned directly, which will continue processing the data. A worker can pick up that task for execution while others run side-by-side with preceding operators. Also, intra-operator ("horizontal") parallelism feels natural by partitioning the data into small fragments: Tasks accessing different fragments can execute in parallel on several worker threads. Notably, this allows accessing fragments in a NUMA-aware manner since `MxTaskFlow` dispatches tasks accordingly, similar to the morsel-driven framework [92]. However, intra-operator parallelism requires operators to be implemented in a parallelism-aware way, at least those at the pipeline's endpoint, such as, for instance, hash table build and aggregation operators.

Execution Phases. So far, we have only addressed the execution phase, where operators consume the result of the predecessor and produce new data for the successor. The *selection* operator is a prime example for which this is sufficient: Tuples passing the filter are streamed to the next operator. However, certain operators, such as *aggregation* and *sort*, must consume all data before producing a result that can be pushed to the next operator. For this reason, `MxTaskFlow` distinguishes between two different execution phases: The *consume* phase, in which data flows from one operator to the next (or is only consumed), and the *finalize* phase, in which corresponding operators can pass their collective results to the next node. Operators that group and aggregate the results will consume all tuples (e.g., storing them in a hash table) before materializing and streaming the data in the finalize phase.

After all tasks of an operator have been executed, and its predecessor has committed that it will not spawn further tasks, `MxTaskFlow` spawns specific tasks that invoke the `finalize()` function of an operator. However, different operators may finalize in different ways. The partition operator, for instance, will spawn worker-local partitioned tasks (cf. Chapter 6). In contrast, aggregation operators may merge multiple, locally aggregated hash tables into a single one. `MxTaskFlow` will either spawn a single task or multiple tasks for worker-local finalization or initiate a more complex *merge* procedure on an operator's behalf, which can be expressed through annotations similar to task annotations. For the merge routine, `MxTaskFlow` spawns a set of tasks, each passing two locally computed results as arguments to the finalization code until the result is finally merged. The query engine only needs to implement the `finalize()` interface and annotate the graph nodes accordingly.

7.2 Compiling Tasks

Let us redirect our attention from the control flow abstraction toward particular implementation intricacies of TunaDB. Compiling the query execution plan into native code can yield significant performance benefits over interpretation (which is only comparable when using vectorization [76]). The advantage is

mainly driven by minimizing the overhead due to function calls and virtual dispatching and by the ability to maintain a tuple in registers rather than in the cache throughout the execution of pipelined operators [114].

7.2.1 Data-centric Code-Generation

How a query is transformed into machine code is crucial for performance. Specifically, two factors are decisive: The time of compiling the query plan and the quality of the generated code. For instance, *HIQUE* [87] emits high-level C code, which is subsequently compiled and linked to the system. Although the execution of compiled code demonstrates noteworthy efficiency, the compilation process incurs costs ranging from several hundred milliseconds to a few seconds.

By directly generating *intermediate representation* (IR) code, systems can eliminate the costly step of parsing and translating a higher-level language. HyPer introduced data-centric code generation using *LLVM IR* [114, 116]. Since then, various systems have also adopted this approach (e.g., [50, 145, 86, 120, 53, 59]). For both C/C++ and LLVM IR, the compiler allows specifying the level of optimization, enabling the adjustment of the balance between better code quality and longer compilation times or lower code quality and shorter compilation times. However, compilation times ranging from tens to a few hundred milliseconds can significantly impact query performance when dealing with small data volumes. HyPer and Umbra dynamically switch between interpretation and compiling engines to address this challenge, starting with interpreting bytecode and transitioning to optimized compiled code when deemed advantageous [85, 77]. FlounderIR [52, 53] tackles high compilation times by providing an IR similar to `x86_64` assembly. By employing lightweight abstractions such as virtual registers with explicit lifetime annotations and C++ function calls, FlounderIR achieves both efficient compilation and user-friendly operation.

7.2.2 Tuning the Code Quality of FlounderIR

The translation process of FlounderIR is straightforward: The *Register Allocator* converts virtual registers into machine registers while spilling values to memory if necessary [52]. Subsequently, the instructions are translated into assembly code (with nearly one-to-one correspondence) and compiled to native code utilizing either *Nasm* or *AsmJit* [83] as a backend. In between, almost no optimizations are made in favor of code quality, which keeps compilation time low but also accepts low code quality as a tradeoff. However, some lightweight and low-level optimizations can enhance the code quality while keeping compilation times short. In the following, we will examine multiple examples. Note that all presented optimizations relate to the `x86_64` architecture and, more precisely, to FlounderIR but can also be adapted for similar IRs.

```
1 cmp counter, fragment_size
2 jge end_scan_loop
3 scan_loop:
4 mov l_discount,
    [fragment+counter*8+8192]
5 mov l_tax,
    [fragment+counter*8+10240]
6 mov l_returnflag,
    [fragment+counter+12288]
```

(a) FlounderIR

```
1  mov rdx, [rsp+24]     ; spill
2  mov rax, [rsp+8]      ; spill
3  cmp rdx, rax
4  jge end_scan_loop
5  scan_loop:
6  mov rdx, [rsp+16]     ; spill
7  mov rax, [rsp+24]     ; spill
8  mov r9, [rdx+rax*8+8192]
9  mov rdx, [rsp+16]     ; spill
10 mov rax, [rsp+24]     ; spill
11 mov r8, [rdx+rax*8+10240]
12 mov rdx, [rsp+16]     ; spill
13 mov rax, [rsp+24]     ; spill
14 rex mov dil,[rdx+rax+12288]
```

(b) Assembly

Figure 7.4: Illustration of FlounderIR and the corresponding assembly, including spill code.

Reducing Spill Instructions. The allocation of virtual registers, or variables in a general sense, to machine registers is a critical factor that notably impacts the performance of a generated program [122]. Register allocation algorithms try to find a mapping from virtual to machine registers, aiming to keep as many values as possible in machine registers. However, due to the limited number of machine registers, it is often unavoidable to *spill* values to memory and load them as needed. Graph coloring [32] leads to favorable allocations, but its computational complexity renders it unsuitable for compiling short-running queries. Instead, TunaDB uses an improved linear scan algorithm [122] that balances efficient register allocation and acceptable performance.

When virtual registers are spilled, the translation algorithm incorporates `mov` instructions to facilitate loading spilled values from memory into temporary registers. Subsequently, the values are written back as required, particularly when instructions modify the value. Figure 7.4a shows a snippet of emitted FlounderIR taken from a scan loop which is responsible for loading multiple attributes from the scanned fragment into virtual registers. The assembly code depicted in Figure 7.4b results from Flounder's register allocation and translation process. The present instance showcases the register allocation process opting to spill the loop variables (`counter` and `fragment_size`) along with the pointer to the fragment that holds the data intended for scanning. Consequently, it is necessary to load the pointer and the loop counter from the stack into temporary registers to access the tuple's attributes. We highlight the relevant spill instructions in the figure.

Given that, in this illustrative snippet, the temporary machine registers

```
1 mov rdx, [rsp+24]          ; spill
2 cmp rdx, [rsp+8]
3 jge end_scan_loop
4 scan_loop:
5 mov rdx, [rsp+16]          ; spill
6 mov rax, [rsp+24]          ; spill

7 mov r9, [rdx+rax*8+8192]
8 mov r8, [rdx+rax*8+10240]
9 rex mov dil,[rdx+rax+12288]
```

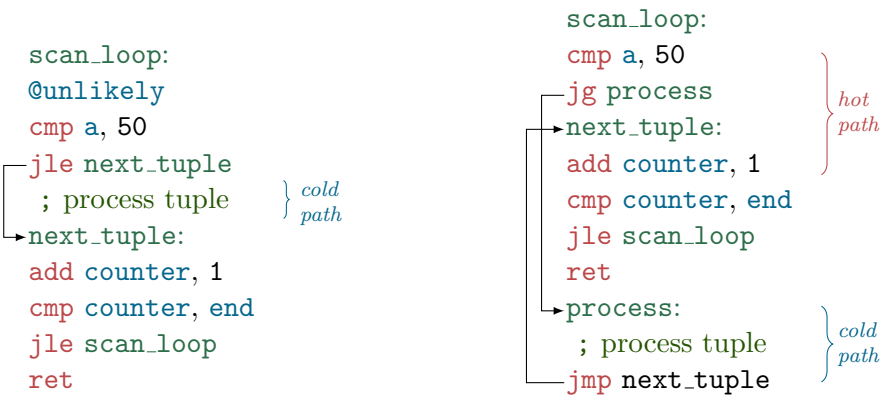
Figure 7.5: Assembly code after reducing spill instructions.

are not utilized to load other values, the repeated spill loads can be considered unnecessary. An optimizer can remove the redundant `mov` directives following the initial loading. However, this is only true inside a single basic block, which relates to a sequence of instructions without any control flow transfer instructions (such as branches or jumps). Hence, the optimizer must take care of such block borders, as the control flow can jump from another program segment. In this example, we cannot assume that the values are present in the corresponding temporary registers when entering the scan loop, which starts at the `scan_loop` label.

Eliminating superfluous spill instructions can be implemented as a single optimizer pass by tracking spill loads and perceiving their allocated values. When identifying spill-code, the optimizer checks if the requested value is already assigned to a temporary machine register. If so, the optimizer can remove the spill-load and use that machine register as the operand within the instruction. Figure 7.5 illustrates the code from Figure 7.4b after applying the optimization. Primarily, the repeated spill loads within the loop are optimized away. Umbra addresses this challenge by attempting to allocate a machine register, in conjunction with a spill slot, to keep a spilled value in a register above the scope of a single instruction [77].

In addition to eliminating repeated `mov` instructions, the optimizer can use the memory location of the spilled value as an operand instead of moving the value into a register. Figure 7.5 demonstrates an example by the `cmp` instruction in line 2: The memory address pointing to the value of the virtual register `fragment_size` is used as a second operand instead of loading it into a temporary register. Nevertheless, at least one of the operands must be a machine register.

Reordering Branches. Another optimization concerns the placement of branches in the code layout. Modern pipelined microprocessors divide the process of executing instructions into several stages. This enables the CPU to begin processing forthcoming instructions before the present one is completed.



(a) FlounderIR (b) Optimized FlounderIR

Figure 7.6: Optimizing the *hot path* in a scan loop by moving less frequented branches out of the loop.

In this process, CPUs can act speculatively, fetching and decoding the following instructions despite the ambiguity surrounding their execution [2]. Conditional jumps, used to follow a branch after testing a condition, can affect the speculative execution when the CPU's branch predictor is wrong [67]. We observed that taking a branch by performing a jump operation is more expensive than executing the branch without a jump operation. Hence, it may be beneficial to organize conditional branches so that the branch executed most frequently is placed sequentially after the condition, whereas the less frequently executed branch is reached through a jump.

Here is where statistics such as data distributions, often maintained by DBMSs to improve query plans, come into play. These statistics can provide indications to the query compiler engine regarding frequently accessed branches. By leveraging this knowledge, the FlounderIR optimizer can arrange compiled predicates to coincide with modern CPU characteristics. Higher-level languages enable the enrichment of conditional branches with metadata, e.g., built-ins like `__builtin_expect` to hint to the compiler for a probably taken branch. Inspired by this idea, we supplement FlounderIR with a similar concept: Annotations allow the code-generating query engine to pass internal knowledge to the IR optimizer. Annotations are implemented as pseudo-instructions and not translated to "real" assembly instructions but assist the optimizer, similar to FlounderIR's built-in lifetime annotations for virtual registers. For example, let us examine a query that applies a selective predicate `a > 50`. The natural approach employed by FlounderIR for constructing a predicate is to produce code that tests the condition and executes a jump operation to the end of the loop if the tuple fails to satisfy the predicate. Figure 7.6a depicts an instance of FlounderIR generated by TunaDB for the illustrated predicate. The *hot path* in this example will follow the conditional jump, as only a small number

of tuples satisfy the filter.

In addition, the example illustrates an annotation to hint to the FlounderIR optimizer: It is *unlikely* that the tuple will pass the filter[1]. The annotation facilitates the FlounderIR optimizer reorganizing the code layout so that the frequently executed branch (skipping the tuple) is positioned immediately after the condition evaluation (`cmp` *a*, `50`). As we show in Figure 7.6b, the optimizer moves the branch taken less frequently outside the scan loop. Consequently, the hot path does not follow a jump. The cost incurred by the optimizer is relatively low, as it merely requires a single scan of the code to identify specific annotations, invert the predicate, and move instructions. Our findings indicate that end-to-end performance increases by up to 9 % for queries with highly selective predicates and tight scan loops.

Strength Reduction. With the help of strength reduction [10, 37], code optimizers replace expensive with cheaper instructions. A simple form is transforming time-consuming divisions and multiplications into weaker algorithmic instructions, e.g., using shift operations if the divisor or multiplicand is constant and of a power of two. In practice, compilers spend much effort replacing multiplications and divisions with weaker (algorithmic) instructions, for example, by using the address generation unit. However, it turned out that complex reductions can rapidly increase the optimization time to such an extent that it is not cost-effective for short-running queries. We found that strength reduction for divisions and multiplications by shifting leads to a minor but notable improvement in compute-bound segments.

We observed a more significant improvement in reducing branches for range predicates. A smaller number of branches lead to fewer branch penalties. A range predicate like `a` ≥ 10 `and` `a` ≤ 100 would naturally be translated into two separate branches:

```
cmp a, 10
jl next_tuple
cmp a, 100
jg next_tuple
```

that are tested one after the other. By subtracting the lower bound from both, the actual value to test and the upper bound, we can translate the predicate to a single comparison:

```
sub a, 10
cmp a, 90
ja next_tuple
```

In doing so, we exploit that the `ja` (*jump above*) instruction interprets a comparison as unsigned, treating negative numbers as large positive ones that

[1] In TunaDB, we annotate predicates with a selectivity of less than 20 % as unlikely.

fall out of the range. We have observed this type of strength reduction also in widely used compilers like *Clang* and *GCC*.

Intermediate Evaluation

We will now examine in which way the optimizations improve the end-to-end performance of queries compiled with FlounderIR. Figure 7.7 depicts the relative execution time (including parsing, planning, and compiling) of optimized FlounderIR compared to the unoptimized iteration for a subset of TPC-H queries[a].

Reducing spill accesses demonstrates an improvement, especially in queries 1 (12 %) and 3 (7 %). Queries 5, 10, 12, and 14 benefit from both spill access optimization and branch reordering: execution time is reduced by 7 % to 13 %. The advantages of optimizing branches by reordering the code layout and reducing between predicates to a single comparison are especially evident in the context of query 6, reducing the execution time by 11 %. The measurements suggest that accepting a modest increase in compilation times for the sake of optimizations in lightweight IRs is a worthwhile trade-off, as it leads to a significant improvement in execution time.

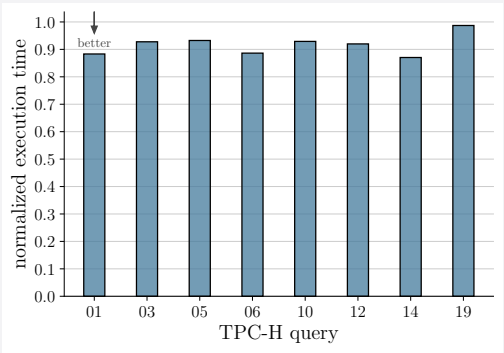


Figure 7.7: Normalized execution time of optimized FlounderIR, compared to the unoptimized iteration.

[a]TunaDB supports TPC-H queries without subqueries.

7.3 Prefetching Materialized Data

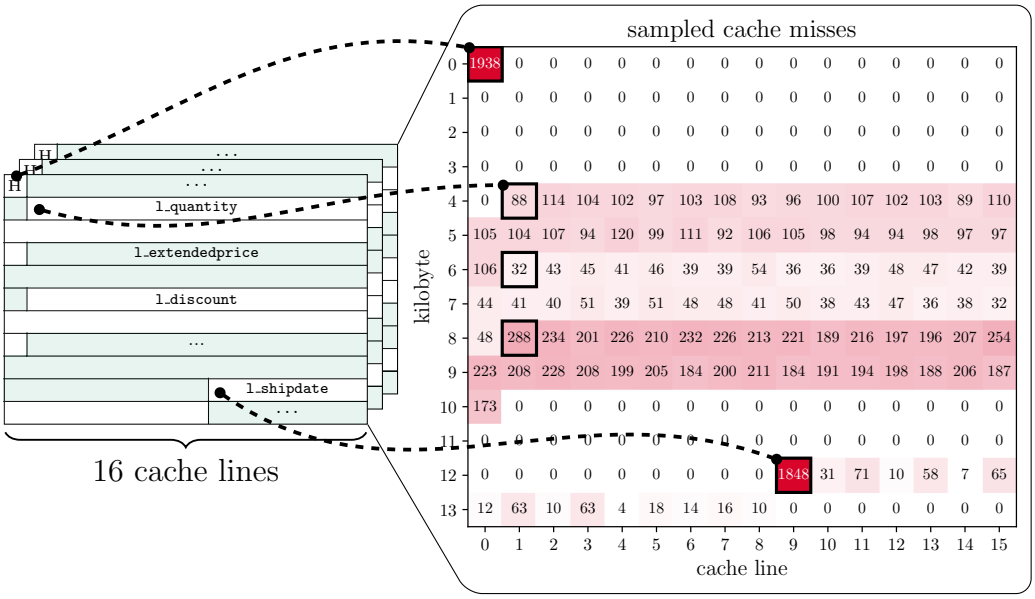
We will now focus again on the execution of tasks. Each `MxTask` accesses and scans a discrete data fragment containing a few hundred tuples. The hardware cannot discern in advance which memory a given task will access. Consequently, the scan operation executed by a task experiences cache misses, at least during the initial stages of execution, until the hardware prefetcher can identify a sequential access pattern and starts proactively loading data into the cache.

7.3.1 Analysis

Let us analyze the access pattern and occurring cache misses that arise from executing the query in Figure 7.8a: a simple scan that selects and aggregates a few columns from the `lineitem` relation. For execution, the runtime initiates

```
select sum(l_extendedprice * l_discount)
from lineitem
where
    l_shipdate <= '1993-08-01' and l_shipdate > '1994-01-01'
    and l_discount between 0.08 and 0.1
    and l_quantity < 24
```

(a) Example query



(b) PAX structure of table fragments (left) and heatmap of cache misses per cache line for all fragments (right)

Figure 7.8: Example query and sampled cache misses during query execution.

a sequence of tasks, each linked to its fragment. TunaDB stores tuples within fragments in a *Partition Attributes Across* (PAX) layout [7], enabling tasks to access only portions of the fragments. The layout of a fragment is presented on the left-hand side of Figure 7.8b. Aside from the tuples, every fragment stores metadata, e.g., the number of tuples, within the header at the initial cache line.

In order to analyze, we sampled the last-level cache misses during the execution of the example query. The heatmap depicted on the right-hand side of Figure 7.8b presents an aggregation of cache misses for each cache line across all relation fragments. Note that these numbers are only *samples* and not definitive quantities. Since the `l_shipdate` predicate has the lowest selectivity of all given predicates, the plan optimizer prioritizes its evaluation for every tuple. Consequently, the corresponding cache lines are accessed most frequently besides the header. However, we can observe that only the first cache line of the `l_shipdate` column exhibits a significant amount of cache misses. During execution, the hardware prefetcher identifies the sequential

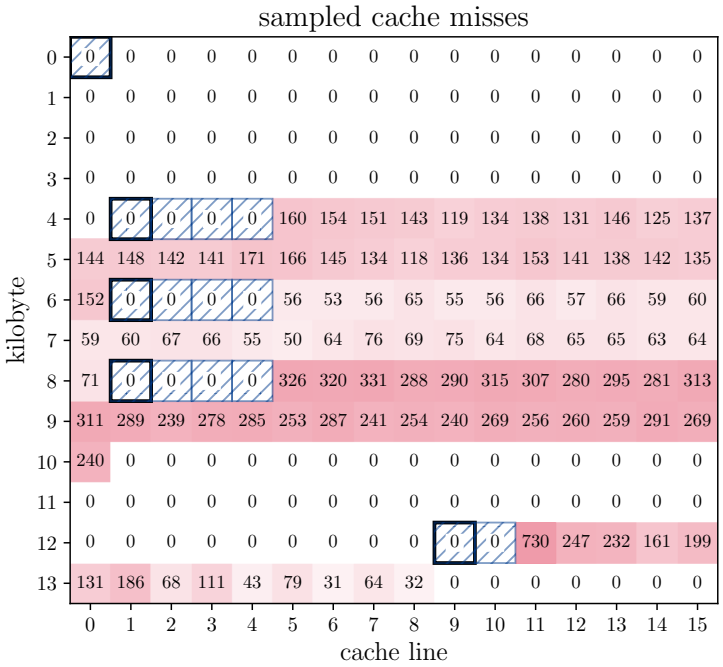
access pattern and proactively fetches the subsequent cache lines. Once a tuple satisfies the first predicate, the subsequent predicates are evaluated sequentially. Here is where things get more complicated for the hardware prefetcher: As only a few tuples match the first predicate, the access pattern for the other columns becomes ambiguous; not every loop iteration will access the columns besides `l_shipdate`. This poses a challenge for the hardware prefetcher to detect a pattern. As a result, cache lines beyond those associated with `l_shipdate` experience notably more cache misses.

7.3.2 Annotation-based Prefetching

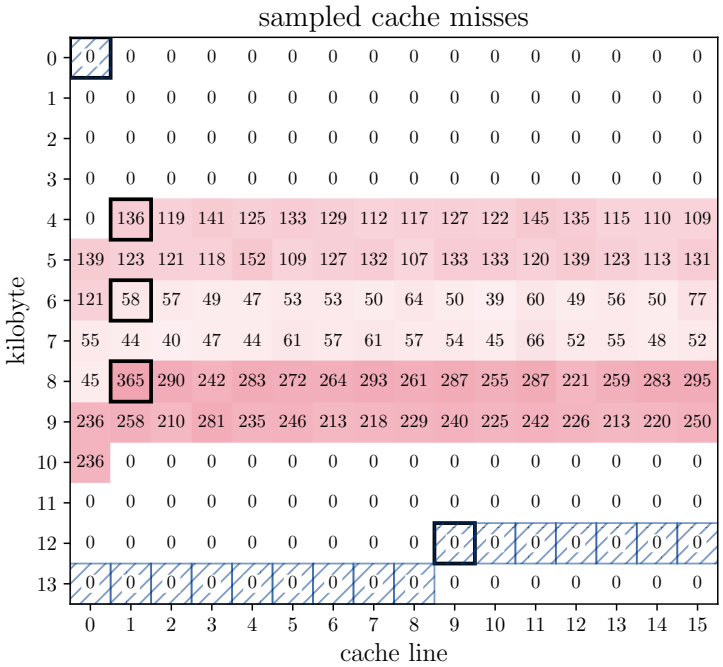
Task annotations can assist in reducing these cache misses during task-based query execution. By annotating the data object the task will access, `MxTasking` can bring the data into the cache prior to the task's execution (cf. Chapter 3). And in the context of `MxTaskFlow`, they are easy to use: The runtime automatically annotates tasks with accessed fragments when spawning to execute the operator graph. However, to optimize prefetching, `MxTaskFlow` must be made aware of which segments of the fragment are accessed and should be prefetched accordingly. In the best case, a task finds all accessed data already cached when executing.

Depending on the size and number of accessed columns, this may require prefetching several kilobytes, which induces pressure on the CPU due to the execution of prefetch instructions and may pollute the cache. Furthermore, the CPU's *Line Fill Buffer* (LFB) capacity is limited to a few cache lines [2], which is responsible for holding unfulfilled data loads, including those initiated by software-based prefetch requests. Consequently, it becomes a tradeoff between prefetching insufficient data and risking cache misses, and prefetching excessive data, which results in an increased number of executed instructions and floods the LFB and the cache. In light of this, two options exist: prefetching the first cache lines of all accessed columns or prefetching some chosen columns entirely.

Prefetching the first cache lines of each column reduces cache misses in the first iterations of the scan loop. But, software and hardware prefetching interfere with each other. Cache *misses* train the hardware prefetcher; software prefetching, in turn, prevents cache misses and, consequently, the hardware prefetcher from recognizing a sequential access pattern [90]. As a result, the cache misses only occur as the scan loop progresses. However, they cannot be prevented entirely as the hardware prefetcher takes action only after detecting some cache misses. We observed that this prefetching strategy improves query performance if all columns are touched approximately equally, i.e., a query segment has no or only lowly selective predicates. Nevertheless, in the case of the example query, prefetching only the first cache lines results in slightly worse performance, as the accesses are focused on `l_shipdate`. We show the partly prefetched heatmap in Figure 7.9a; prefetched cache lines are marked with blue fill patterns.



(a) Prefetching the first cache lines of multiple columns



(b) Prefetching `l_shipdate` entirely

Figure 7.9: Heatmap of sampled cache misses of the lineitem fragment when prefetching the fragment's header and (a) multiple columns partly or (b) the entire `l_shipdate` column. Prefetched cache lines are marked with fill patterns. We used the example query from Figure 7.8a as a workload.

Columns accessed to a high degree exclusively and relatively small in size are good candidates for our second strategy: prefetching specific columns entirely. For instance, tuples in the example query in Figure 7.8a are filtered by a highly selective predicate on the `l_shipdate` column. Only about 3.5 % of the tuples satisfy that predicate. Consequently, most iterations in the scan loop will access `l_shipdate` exclusively and discard the tuple afterward. In contrast, the columns `l_extendedprice` and `l_discount` are mainly accessed in conjunction due to their shared usage for aggregation. Prefetching only one of the two columns is less advantageous as the CPU must await both values to be loaded from memory (which may happen simultaneously, thanks to out-of-order execution implemented in most contemporary processors).

Figure 7.9b illustrates the heatmap of cache misses while executing the example query and prefetching `l_shipdate` entirely due to its dominating access pattern. This way, that column's cache misses are reduced significantly, consequently increasing performance. We observe an improvement of 10 %, although cache misses may still occur when accessing non-prefetched columns. In practice, both techniques can be combined flexibly for different query pipelines, depending on the accessed columns and their interaction. TunaDB uses a simple heuristic and primarily considers the selectivities of the predicates to select one of the two variants: Predicates on a single column with a selectivity of less than 10 % are considered dominant; this column is prefetched entirely. However, this heuristic holds potential for further enhancements. Another optimization is to start prefetching (either some columns entirely or all partly) based on task annotations and continue by generating corresponding prefetch instructions within the scan loop, eliminating the reliance on the hardware prefetcher altogether.

7.3.3 Generating Prefetch Instructions

The optimal prefetching strategy may deviate across different pipelines within a single query. TunaDB executes every pipeline (separated by pipeline breakers) within a compiled query as a particular task type. As a result, different pipelines consume and produce differently structured fragments that exhibit varying access patterns. The radix join, discussed in Chapter 6, is an appropriate example: During the probing phase, one pipeline will consume relation fragments (1) and produce partitioned micro fragments, which the probing pipeline will consume (2); both pipelines may need unique prefetching patterns. We illustrate this example in Figure 7.10.

To ensure that the data for each pipeline can be brought to the cache in advance, TunaDB utilizes FlounderIR to generate a separate *prefetch callback* for each pipeline. The callback considers the access pattern of the pipeline, such as attribute conjunctions, and the physical data layout of the accessed data fragment, which can be either the result of a previous pipeline or a relation in the case of scanning. During execution, `MxTaskFlow` annotates the appropriate

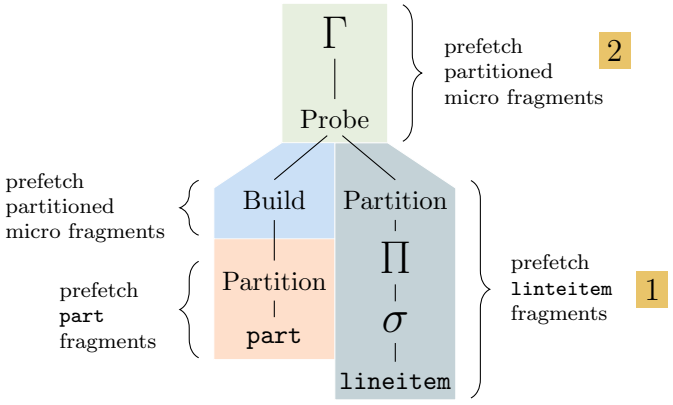


Figure 7.10: Illustration of query segments and appertaining prefetches.

callback pointer along with the accessed fragments to spawned tasks to hint at the underlying runtime for prefetching through task annotations. `MxTasking`, in turn, will schedule the callback and invoke it in-between task execution (cf. Chapter 3).

7.4 Experimental Evaluation

We will now evaluate TunaDB to study the `MxTask`-abstraction in a more complex environment, such as the granularity of tasks and annotation-driven prefetching. As a workload, we rely on the TPC-H [26] benchmark suite, using a scale factor of 50 and those queries that do not include subqueries. Additionally, we have not implemented a sort operator in TunaDB and refrain from sorting in all benchmarks. All workloads are executed on a two-socket machine with 24 physical and 48 logical cores in total (further details are already given in Section 5.5).

7.4.1 Prefetching

First, we evaluate the annotation-driven prefetching mechanism. Since the different query pipelines (executed as tasks) have varying execution times, we utilize `MxTasking`'s dynamic prefetching mechanism (as described in Section 3.4). Figure 7.11 shows the relative execution performance, comparing a non-prefetched with a prefetched run. To classify the results, we examine the number of executed instructions, the number of memory stalls, and the execution time for each query. We have recorded the details using the Linux performance counter interface. To comprehend the interplay between software prefetching and *simultaneous multithreading* (SMT), we employ the logical cores as follows: Initially, we utilize all physical cores from both NUMA regions. When all cores are in use, we gradually include hyperthreads until all logical

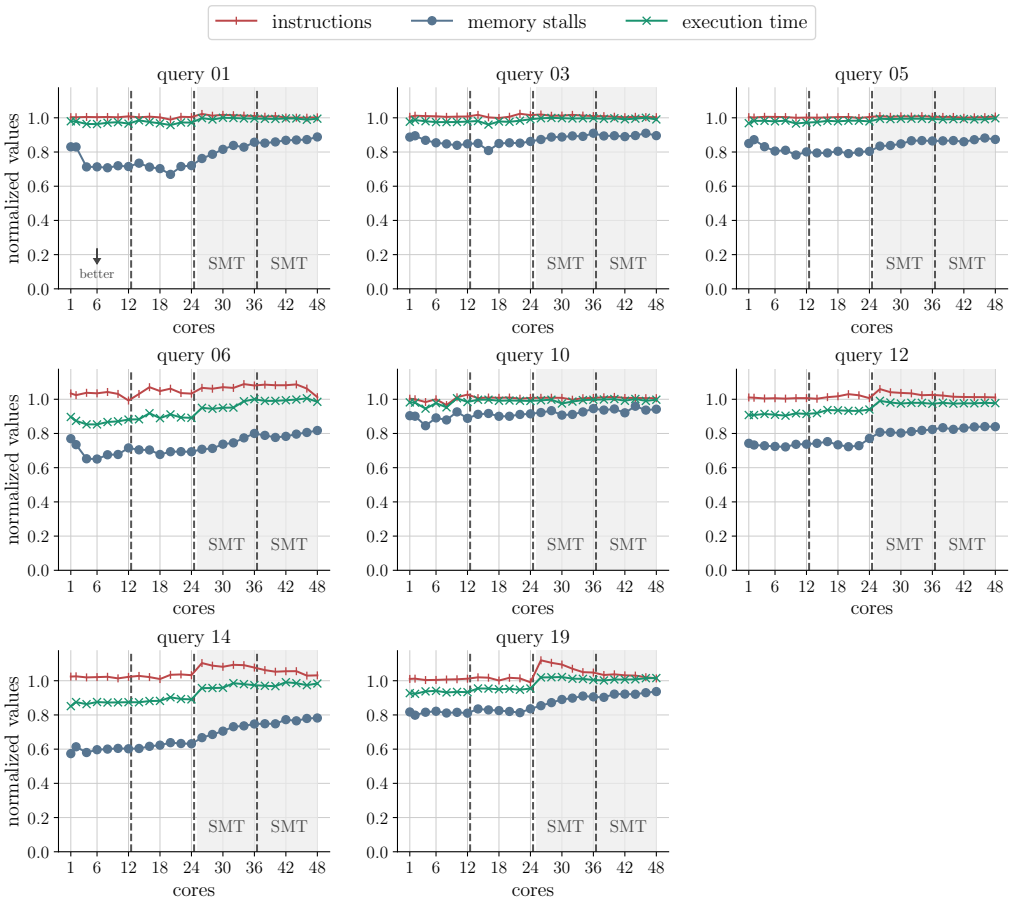


Figure 7.11: Effect of prefetching for a subset of TPC-H queries, comparing a prefetched execution with a non-prefetched one.

cores are effectively engaged. Note that we emphasized NUMA borders with a dashed line.

The obtained measurements reveal two insights: The efficacy of prefetching varies considerably across the queries, with specific queries exhibiting up to a 15 % increase in execution speed. In contrast, other queries experience occasional decreases of up to 2 % compared to non-prefetched execution. The impact of prefetching on memory stalls varies across different queries. In some queries, prefetching can reduce the number of stalled cycles by up to 42 %. However, for other queries, the reduction in stalled cycles is more limited, with improvements of about 7 %. Moreover, the number of memory stalls tends to increase when using multiple hardware threads per physical core (irrespective of whether the execution is prefetched). We will now examine the different queries before looking at the interference of prefetching and SMT.

Varying Efficiency in Queries. As discussed in Section 7.3, the accessed columns play an important role in prefetching. Since not all accessed columns can be prefetched, deciding whether all accessed columns should be prefetched partially or some columns should be brought into the cache entirely is necessary. Query 14 is an excellent example of the latter case: a join with a highly selective predicate on the build side, which can be prefetched entirely, and a projection on the altogether small probe relation. The materialized columns needed by the query at an early stage (e.g., the join predicate for the probe relation, which is hashed and partitioned) are so small that they can be prefetched entirely. This reduces memory stalls between 42 % and 22 %, depending on whether multiple hardware threads are used per physical core. Query 10 represents an example of a less prefetch-friendly query: Many columns, including some larger text-containing ones, are accessed in conjunction, without one attribute being accessed exclusively and therefore being a good target for prefetching completely. As a result, all columns are prefetched partly, leading to a lower prefetching efficiency since software-based prefetching does not train the hardware prefetcher, and cache misses still occur as the scan loops make progress.

Furthermore, we can observe that the coherence between reduced memory stalls and reduced execution time is asymmetric across the queries. Throughout the execution of query 1, for instance, prefetching reduces memory stalls by up to 23 % (at 20 cores) while the execution time improves by up to 4 %. We can observe a similar pattern for queries 3 and 5. For query 19, in contrast, memory stalls are reduced by up to 20 % while the execution time improves by up to 8 %. As of the example of query 1, we can determine with the assistance of additional profiling tools (Intel VTune [69] and perf [97]) that prefetching shifts the central bottleneck: from latency bound during the scan and loading of the attributes to core bound during the computation of the hash and the aggregation of grouped values in the hash table. Since we generate the code for the hash table inserts and lookups using FlounderIR, we attribute this "new bottleneck" to the less efficient code quality. Using more complex instructions (e.g., *SIMD*) and a more mature hash table implementation may improve performance noticeably as task execution is less affected by fetching data from memory using prefetching. For queries utilizing bloom-filtered radix joins (e.g., queries 3, 5, and 10), we observe that the execution time is dominated by lookups to the bloom filter on the probe side, where every lookup leads to a cache miss. Because the corresponding filter address for each tuple depends on the join predicate and is calculated during the scan, the filter's memory location cannot be predicted and annotated and implicitly not be prefetched in advance by `MxTasking`.

Interference with SMT. The second insight we derive from the measurements is that prefetching becomes less effective when we utilize two hardware

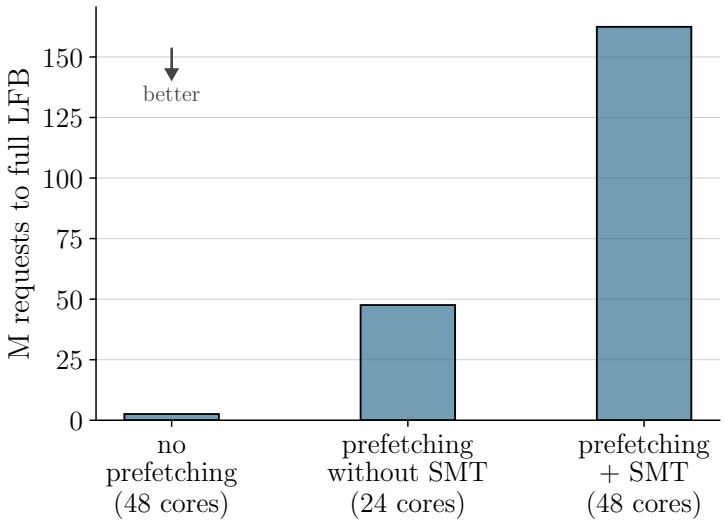
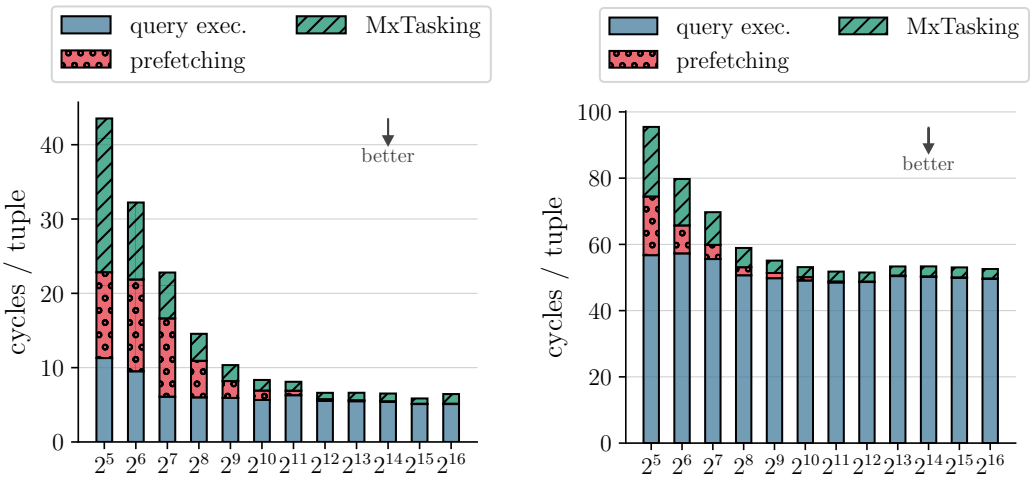


Figure 7.12: Quantity of requests to the LFB without capacity available, using query 6 as an example. We illustrate the measured counter values for a not-prefetched run, a run with prefetching but only physical cores, and with prefetching and all logical cores.

threads per physical core. To a certain degree, this phenomenon is inherent, as SMT is known to hide memory stalls, particularly when the logical cores are executing a similar instruction stream. Initial findings on the B^{link}-tree (cf. Section 5.5) already suggest that the efficiency of prefetching is not uniform when utilizing SMT compared to its efficiency when using only a single hardware thread per core. Our observations indicate another reason for this is the limited capacity of the LFB, which manages the pending data accesses, including software prefetch requests. While prefetching without SMT already floods the LFB with requests, the effect becomes amplified with utilizing all logical cores: the number of requests to insert an entry in the buffer but no capacity left more than triples. We show the number of requests that hit a full LFB in Figure 7.12 using TPC-H query 6 as an illustrative example. Based on that observation, we assume that both hardware threads have to share one LFB on a physical core, significantly reducing the improvement of software-based prefetching using SMT.

7.4.2 Task Granularity

While designing task-oriented applications or data structures, the granularity of a task may be an adjustable parameter. In certain workloads, the access characteristics of the application imply task granularities. For example, `MxTask`s traversing a B^{link}-tree, as described in Chapter 5, operate on a single node per task. In contrast, the task-based radix join's granularity is driven by hardware characteristics. More precisely, the TLB capacity restricts the number of



(a) Single aggregation (b) Grouped aggregation

Figure 7.13: Effect of varying task granularities for different compute-intensive queries.

tuples to prevent the partitioning phase from exceeding the limits while writing to micro fragments (cf. Chapter 6). For various applications, however, the granularity is arbitrary. Spawning `MxTask`s causes additional overhead that could become a bottleneck when tasks are too short-lived.

In order to explore potential variations in task granularities, we execute two queries that exhibit contrasting levels of computational intensity. The first query aggregates only a single column (`l_quantity` from `lineitem`), which exerts considerable pressure on `MxTasking` components, such as the dispatcher and the worker queues, owing to its low CPU utilization. The second query projects and aggregates on five distinct columns and groups the result by two columns (de facto query 1 of the TPC-H benchmark, but without selection), which generates noticeably more CPU utilization by calculating the result. Figure 7.13 depicts the results, grouping the consumed CPU cycles by efforts for `MxTasking`, prefetching, and query execution. The results were recorded with Intel VTune using all 48 logical cores. We vary the number of tuples per task from 32 to 65 536.

We observe that the utilization of fine granularities results in noticeable overhead for tasking-related components compared to the query execution efforts for both queries. With a limit of 32 tuples per task, dispatching and receiving tasks incur a comparable number of cycles to those required for computing the results of the first query (cf. Figure 7.13a). As the granularity increases, the overhead of tasking decreases. We can observe the same for prefetching overhead, mainly driven by executing appropriate instructions. Moreover, fine granularities cause additional overhead for each task. For instance, task-local aggregation results are merged into a worker-local result

after task execution, which appears more frequently with smaller tasks. In conjunction with function call overhead (every task invokes the generated code), this extra work overwhelms the advantage of prefetching on smaller columns. Upon a granularity of 1 024 tuples per task, the task overhead becomes negligible for both measured queries. We examine that the best-performing granularity depends on the query intensity: The local minimum is achieved at 32 768 tuples per task for the simple aggregation (Figure 7.13a) and at 2 048 tuples per task for the more compute-intensive grouped aggregation (Figure 7.13b).

7.4.3 Summary

In this experimental evaluation, we analyzed different aspects of `MxTasking` in the context of query execution: prefetching and task granularities. As tasks must comprise a certain amount of tuples to minimize tasking overhead, entirely prefetching a fragment's data has proven to be impractical in some cases. Specifically, on the Intel systems we used for the evaluation, the CPU cannot buffer all software prefetching requests. Combined with efforts for initiating prefetches, this limits the benefits of prefetching to specific parts of a data fragment. The utilization of multiple hardware threads per physical core further amplifies this effect. However, with certain strategies, specific columns of a fragment can be prefetched advantageously, for example, by loading highly selective columns in their entirety.

8 Summary and Future Directions

This thesis investigated the implications of a profound transition from a thread-oriented to a task-based processing paradigm. In doing so, we addressed a fundamental problem of the former: The abstraction of control flows by "conventional" threads seems inadequate for exchanging relevant information between the application and the execution substrate. The lack of effective communication led to both tiers working around each other, missing out on optimizations that leverage the resources available in today's hardware landscapes. As a remedy, we presented `MxTasking`, a task-based framework that breaks with patterns of processing models designed for earlier generations of hardware.

8.1 Summary

The unique selling point of `MxTask`s is to provide *annotations* that enable algorithm engineers to transfer knowledge from the application level to the control-flow abstraction. Consequently, the execution layer no longer has to guess the applications' intentions but can base optimizations on actual knowledge the application provides. In Part I ("`MxTasking` Layer"), we discussed two ways of leveraging this information by the `MxTasking` runtime. With the knowledge of the data objects a task will access, the runtime can enhance performance by injecting *prefetch* instructions that bring the relevant data into fast CPU caches before execution. This way, expensive access latencies are hidden behind the computational work of other tasks, reducing the time the CPU has to wait for data to be transferred from memory into registers. The additional benefit of data object annotations is the *synchronization* of concurrent accesses. By understanding the interplay of code and data, `MxTasking` can take over the synchronization of tasks—decoupling the error-prone

utilization of synchronization mechanisms from the application logic. All the developer has to do is annotate tasks with data objects accordingly. In order to optimize synchronization, the annotation can be supplemented by the access pattern, explicitly indicating whether the task will read or write the data object. `MxTasking`, in exchange, picks the most appropriate synchronization mechanism and harmonizes it with the underlying hardware.

Part II ("Leveraging Tasks for Data Structures") studied the `MxTask`-abstraction using a B^{link}-tree as a demonstrator that represents the behavior of modern in-memory database engines. We have found that the prefetching mechanism is especially beneficial for data accesses that are particularly challenging for the hardware to predict. Additionally, we demonstrated that sophisticated and robust concurrency synchronization can be integrated seamlessly into the data structure with minimal effort from the developer. Our evaluation reveals that the utilization of `MxTask`s for a B^{link}-tree leads to a significant enhancement in performance by up to 30 % compared to existing processing models. Furthermore, the experiments indicate that this approach outperforms state-of-the-art tree-like data structures.

In Part III ("Exploiting Tasks at the System Layer"), we focused on the query execution engine in the light of database systems. First, we discussed how task-based *micro partitioning* streamlines the design of hardware-conscious algorithms, using radix joins as a showcase. The developer can request cache-aware execution by annotating tasks accessing the same data structure. `MxTasking` dispatches tasks appropriately to keep the data structure available in the cache. Second, we reviewed how tasks drive the development of a push-based query engine and saw how `MxTasking` decouples the data and control flow abstraction from the operator implementation. Using our demonstrator

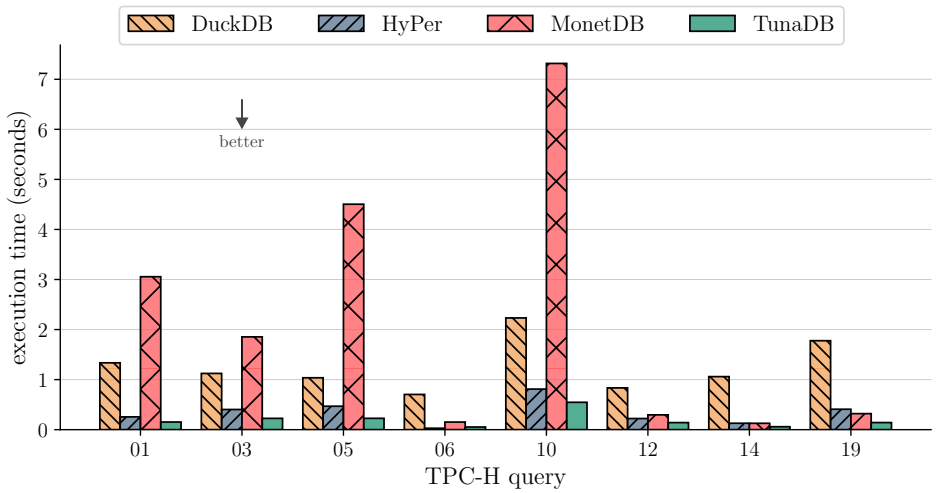


Figure 8.1: Comparison of our `MxTask`-based DBMS TunaDB with state-of-the-art DBMSs, using TPC-H with a scale factor of 50 on a two-socket machine.

TunaDB, we studied the access patterns of tasks inside the query engine and presented different strategies for prefetching data fragments. Throughout the experimental evaluation, we found that prefetching can reduce cache misses and memory stalls, in some cases significantly, and improve performance. Plus, the modest overhead of `MxTasking` facilitates a broad range of task granularities.

Finally, we conclude this summary by briefly comparing TunaDB and state-of-the-art DBMSs. Although it is crucial to exercise caution when interpreting the findings[1], the results presented in Figure 8.1 indicate the competitive performance of our `MxTask`-based demonstrator.

8.2 Future Research Directions

The concepts presented in this work provide the potential for further promising research directions, which we will discuss in the following. `MxTasking`, as described in this thesis, constitutes a fundamental component of the `MxKernel` initiative. `MxKernel` envisions enhancing the interfaces and integration of the operating system and performance-oriented applications, such as DBMSs, and enabling a seamless exchange in both directions to optimize execution on sophisticated, modern hardware. In this light, our published `MxTasking` implementation[2] serves as the primary control flow abstraction in the `MxKernel` prototype[3]. Although such an ambitious project requires substantial efforts, it can resolve the ongoing conflict between the OS and DBMS for control over the system. We expect to improve the efficiency of task-oriented applications, both in the standalone `MxTasking` framework and within the `MxKernel` system.

Task variants tailored for heterogeneous landscapes, such as GPUs, FPGAs, many-core co-processors, and network-based processing, could attractively complement the `MxTasking` runtime environment. Query processing on heterogeneous hardware has been discussed intensively in recent years (e.g., [138, 113, 27, 28, 51]). Embedded in `MxTasking`, the scheduler could make optimized and annotation-supported decisions about allocating task variants on specific devices. In the context of `MxKernel`, appropriate task variants can lead to an efficient and fair distribution of heterogeneous resources, particularly across the boundaries of a single application. Preliminary ideas were already presented by Müller et al. [112].

Throughout this thesis, we saw that annotation-based software prefetching improves performance notably. Utilizing (query) compilers to estimate task execution times and generate appropriate annotations could enhance this mechanism's robustness and prepare it for adoption across various applications. Latency-conscious augmentations (e.g., for different NUMA regions) and the

[1] For example, TunaDB does not implement transaction handling and `NULL` values. Furthermore, the logical query plans of the examined DBMSs differ.

[2] `https://github.com/jmuehlig/mxtasking`

[3] `https://github.com/mmueller41/genode`

integration of diverse memory types, such as high-bandwidth and persistent memory, enable further optimizations and improve performance. Plus, task-based prefetching can extend beyond main memory: Prefetching of disk-based data and, in general, asynchronous task-driven I/O would also be conceivable (for example, in the sense of [143]).

Finally, the primary focus of TunaDB was on analytical query processing. However, we have also found that tasks, particularly in conjunction with annotations, can simplify synchronization for concurrent control flows. A promising avenue for further exploration would be incorporating transaction processing into the task-based abstraction model (in the spirit of DORA [119]). This would enable the seamless integration of transaction handling into DBMSs and applications beyond, harnessing the power of task annotations.

Acknowledgements

Completing this work has been a challenging journey, and I could not have done it without the support and assistance of many individuals. I would like to express my deepest gratitude to everyone who has supported and encouraged me, both personally and professionally. In the ensuing paragraphs, I express my heartfelt appreciation to selected individuals; however, this is by no means exhaustive.

First and foremost, I would like to express my sincere appreciation to *Jens Teubner* for giving me the opportunity to realize this work. His guidance and advice, the knowledge he taught me, and his encouragement have made it possible for me to attain this goal.

Furthermore, I would like to express my gratitude to *Olaf Spinczyk* for introducing me to the `MxKernel` project and consistently supporting me with valuable discussions and feedback throughout.

Next, I would like to thank the members of the PhD thesis committee for taking the time and effort. Many thanks to *Viktor Leis* for reviewing this thesis and to *Ben Hermann* and *Mario Botsch* for participating in the defense. I also want to thank *Peter Ulbrich* for his mentoring and insightful advice.

During my time in the DBIS group, I had the privilege of interacting with many individuals. In particular, I would like to extend my gratitude to *Maximilian Berens*, *Henning Funke*, *Roland Kühn*, *Alexander Lochmann*, and *Lea Schönberger* for their collaborative efforts, engaging discussions, invaluable feedback, and proofreading. Their unwavering support and guidance played a crucial role in keeping me on track. I would especially like to thank *Lea* for our cooperation throughout the writing process, which not only made the work easier but also helped me to stay focused and determined. Many thanks also to *Florian Grieskamp*, *Michael Kussmann*, *Thomas Lindemann*, and *Jannik Wolff* for their support and contributions. Over the years, there were collaborations with *Henning Funke*, *Stefan Noll*, *Alexander Böhm*, *Norman May*, and *Michael Müller*. These partnerships are something I value greatly; they have been both enjoyable and educational for me.

This thesis would not have been possible without funding from the Deutsche Forschungsgemeinschaft (DFG) within the SPP 2037 "Scalable Data Management for Future Hardware." Many thanks to all who organized and participated in this program, which led to exciting and educational discussions, instructive

collaborations, and enjoyable events.

Last but not least, I wish to express my heartfelt appreciation to my beloved family and friends. A profound thank you goes to my parents, *Gerrit* and *Sabine*, and my brother *Tim*, whose unwavering support has been a constant pillar throughout my entire journey. Additionally, I extend my gratitude to many friends who have offered support and a helping hand whenever I needed it. Many thanks, in particular, to *Tim* and *Diana* for their time and effort in proofreading my work, a gesture I genuinely appreciate. While I would further like to mention *Henning*, *Mario*, and *Vanessa* in particular, I am grateful to all my friends for their invaluable presence.

Bibliography

[1] Intel® Memory Latency Checker v3.10. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2021. [Online; accessed February 2023].

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200, 2022. [Online; accessed October 2022].

[3] Intel® Transactional Synchronization Extension (Intel® TSX) Disable Update for Selected Processors. https://cdrdv2.intel.com/v1/dl/getContent/643557, 2022. [Online; accessed November 2022].

[4] Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile. https://developer.arm.com/documentation/ddi0608/latest/, 2023. [Online; accessed May 2023].

[5] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 289–302. USENIX, 2002. URL http://www.usenix.org/publications/library/proceedings/usenix02/adyahowell.html.

[6] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 266–277. Morgan Kaufmann, 1999. URL http://www.vldb.org/conf/1999/P28.pdf.

[7] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 169–180. Morgan Kaufmann, 2001. URL http://www.vldb.org/conf/2001/P169.pdf.

[8] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO*, pages 305–317. ACM, 2017. URL `http://dl.acm.org/citation.cfm?id=3049865`.

[9] Marco Aldinucci, Massimo Torquati, and Massimiliano Meneghin. FastFlow: Efficient Parallel Streaming Applications on Multi-core. *CoRR*, abs/0909.1187, 2009. URL `http://arxiv.org/abs/0909.1187`.

[10] Frances E Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. *Program Flow Analysis*, pages 79–101, 1981.

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009 Parallel Processing, 15th International*, volume 5704, pages 863–874. Springer, 2009. doi: 10.1007/978-3-642-03869-3_80.

[12] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay P. Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distributed Syst.*, 20(3):404–418, 2009. doi: 10.1109/TPDS.2008.105. URL `https://doi.org/10.1109/TPDS.2008.105`.

[13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE*, pages 362–373. IEEE Computer Society, 2013. doi: 10.1109/ICDE.2013.6544839.

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

[15] Maximilian Bandle and Thomas Neumann. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD*, pages 168–180. ACM, 2021. doi: 10.1145/3448016.3452831.

[16] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. Robust Performance of Main Memory Data Structures by Configuration. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD*, pages 1651–1666. ACM, 2020. doi: 10.1145/3318464.3389725.

[17] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Adaptive Query Compilation in Graph Databases. In *37th IEEE International Conference on Data Engineering Workshops, ICDE*, pages 112–119. IEEE, 2021. doi: 10.1109/ICDEW53142.2021.00027. URL `https://doi.org/10.1109/ICDEW53142.2021.00027`.

[18] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141. ACM, 1970. doi: 10.1145/1734663.1734671. URL `https://doi.org/10.1145/1734663.1734671`.

[19] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128. ACM Press, 2000. doi: 10.1145/378993.379232.

[20] Lars Bergstrom. Measuring NUMA effects with the STREAM benchmark. *CoRR*, abs/1103.3225, 2011. URL `http://arxiv.org/abs/1103.3225`.

[21] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davi. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *7th IEEE European Symposium on Security and Privacy, EuroS&P*, pages 502–519. IEEE, 2022. doi: 10.1109/EuroSP53844.2022.00038.

[22] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 640–652. ACM, 2011. doi: 10.1145/1993498.1993573. URL `https://doi.org/10.1145/1993498.1993573`.

[23] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 207–216. ACM, 1995. doi: 10.1145/209936.209958.

[24] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 54–65. Morgan Kaufmann, 1999. URL `http://www.vldb.org/conf/1999/P5.pdf`.

[25] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR*, pages 225–237. www.cidrdb.org, 2005. URL http://cidrdb.org/cidr2005/papers/P19.pdf.

[26] Peter A. Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC*, volume 8391 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2013. doi: 10.1007/978-3-319-04936-6_5. URL https://doi.org/10.1007/978-3-319-04936-6_5.

[27] Sebastian Breß, Max Heimel, Michael Saecker, Bastian Köcher, Volker Markl, and Gunter Saake. Ocelot/HyPE: Optimized Data Processing on Heterogeneous Hardware. *Proc. VLDB Endow.*, 7(13):1609–1612, 2014. doi: 10.14778/2733004.2733042. URL http://www.vldb.org/pvldb/vol7/p1609-bress.pdf.

[28] Sebastian Breß, Henning Funke, and Jens Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*, pages 1891–1906. ACM, 2016. doi: 10.1145/2882903.2882936. URL https://doi.org/10.1145/2882903.2882936.

[29] George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-Aware Compiler Prefetching for Many-Cores. In *Ninth International Symposium on Parallel and Distributed Computing, ISPDC*, pages 133–140. IEEE Computer Society, 2010. doi: 10.1109/ISPDC.2010.16. URL https://doi.org/10.1109/ISPDC.2010.16.

[30] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler Optimizations for Improving Data Locality. In *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262. ACM Press, 1994. doi: 10.1145/195473.195557. URL https://doi.org/10.1145/195473.195557.

[31] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 181–190, 2001. URL http://www.vldb.org/conf/2001/P181.pdf.

[32] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation Via

Coloring. *Comput. Lang.*, 6(1):47–57, 1981. doi: 10.1016/0096-0551(81)90048-5. URL https://doi.org/10.1016/0096-0551(81)90048-5.

[33] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO*, pages 12–23. ACM, 2016. doi: 10.1145/2854038.2854044. URL https://doi.org/10.1145/2854038.2854044.

[34] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering, ICDE*, pages 116–127, 2004. doi: 10.1109/ICDE.2004.1319989.

[35] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979. doi: 10.1145/356770.356776. URL https://doi.org/10.1145/356770.356776.

[36] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC*, pages 143–154. ACM, 2010. doi: 10.1145/1807128.1807152.

[37] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, 2001. doi: 10.1145/504709.504710. URL https://doi.org/10.1145/504709.504710.

[38] Julita Corbalán, Alejandro Duran, and Jesús Labarta. Dynamic Load Balancing of MPI+OpenMP Applications. In *33rd International Conference on Parallel Processing (ICPP)*, pages 195–202. IEEE Computer Society, 2004. doi: 10.1109/ICPP.2004.1327921. URL https://doi.org/10.1109/ICPP.2004.1327921.

[39] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*, pages 215–226. ACM, 2016. doi: 10.1145/2882903.2903741. URL https://doi.org/10.1145/2882903.2903741.

[40] Fredrik Dahlgren and Per Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distributed Syst.*, 7(4):385–398, 1996. doi: 10.1109/71.494633.

[41] Laurent David and Isabelle Puaut. Static Determination of Probabilistic Execution Times. In *16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 223–230. IEEE Computer Society, 2004. doi: 10.1109/ECRTS.2004.34. URL https://doi.ieeecomputersociety.org/10.1109/ECRTS.2004.34.

[42] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168. ACM, 2009. doi: 10.1145/1508244.1508263.

[43] Matthias Diener, Eduardo Henrique Molina da Cruz, and Philippe Olivier Alexandre Navaux. Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP*, pages 9–16. IEEE Computer Society, 2015. doi: 10.1109/PDP.2015.11.

[44] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *Proc. VLDB Endow.*, 12(12):2218–2229, 2019. doi: 10.14778/3352063.3352137. URL http://www.vldb.org/pvldb/vol12/p2218-dursun.pdf.

[45] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *Int. J. Softw. Tools Technol. Transf.*, 4(4):437–455, 2003. doi: 10.1007/s100090100054. URL https://doi.org/10.1007/s100090100054.

[46] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference*, 2006. URL https://papers.freebsd.org/2006/bsdcan/evans-jemalloc.files/evans-jemalloc-paper.pdf.

[47] Facebook. Folly: Facebook Open-source Library. https://github.com/facebook/folly, 2023. [Online; accessed April 2023].

[48] Karl-Filip Faxén. Wool-A work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, 2008. doi: 10.1145/1556444.1556457.

[49] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, UK, 2004. URL https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193.

[50] Craig Freedman, Erik Ismert, and Per-Åke Larson. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.*, 37(1):

22–30, 2014. URL http://sites.computer.org/debull/A14mar/p22.pdf.

[51] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD*, pages 1603–1618. ACM, 2018. doi: 10.1145/3183713.3183734. URL https://doi.org/10.1145/3183713.3183734.

[52] Henning Funke, Jan Mühlig, and Jens Teubner. Efficient generation of machine code for query compilers. In *16th International Workshop on Data Management on New Hardware, DaMoN*, pages 6:1–6:7. ACM, 2020. doi: 10.1145/3399666.3399925. URL https://doi.org/10.1145/3399666.3399925.

[53] Henning Funke, Jan Mühlig, and Jens Teubner. Low-latency query compilation. *VLDB J.*, 31(6):1171–1184, 2022. doi: 10.1007/s00778-022-00741-5. URL https://doi.org/10.1007/s00778-022-00741-5.

[54] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009. [Online; accessed October 2022].

[55] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 International Conference on Management of Data, SIGMOD*, pages 102–111. ACM Press, 1990. doi: 10.1145/93597.98720. URL https://doi.org/10.1145/93597.98720.

[56] Goetz Graefe. Parallelizing the Volcano database query processor. In *Intellectual Leverage: Thirty-Fifth IEEE Computer Society International Conference*, pages 490–493. IEEE Computer Society, 1990. doi: 10.1109/CMPCON.1990.63729. URL https://doi.org/10.1109/CMPCON.1990.63729.

[57] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994. doi: 10.1109/69.273032. URL https://doi.org/10.1109/69.273032.

[58] Goetz Graefe. Modern B-Tree Techniques. *Found. Trends Databases*, 3 (4):203–402, 2011. doi: 10.1561/1900000028. URL https://doi.org/10.1561/1900000028.

[59] Immanuel Haffner and Jens Dittrich. mutable: A Modern DBMS for Research and Fast Prototyping. In *13th Conference on Innovative Data Systems Research, CIDR*, 2023. URL https://www.cidrdb.org/cidr2023/papers/p41-haffner.pdf.

[60] Richard A. Hankins and Jignesh M. Patel. Effect of node size on the performance of cache-conscious B^+-trees. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS*, pages 283–294. ACM, 2003. doi: 10.1145/781027.781063. URL https://doi.org/10.1145/781027.781063.

[61] Dateng Hao and Li Sun. 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*, pages 1769–1777. IEEE, 2013. doi: 10.1109/HPCC.and.EUC.2013.253. URL https://doi.org/10.1109/HPCC.and.EUC.2013.253.

[62] William Hasenplaugh, Andrew Nguyen, and Nir Shavit. Quantifying the Capacity Limitations of Hardware Transactional Memory. *WTTM'15: 7th Workshop on the Theory of Transactional Memory*, 2015. URL https://www.dpss.inesc-id.pt/~salaa/wttm2015/html/abstracts/Hasenplaugh.pdf.

[63] Yongjun He, iacheng Lu, and Tianzheng Wang. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.*, 14(3): 431–444, 2020. doi: 10.5555/3430915.3442440.

[64] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. ACM, 1993. doi: 10.1145/165123.165164.

[65] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 40(8):1687–1700, 2021. doi: 10.1109/TCAD.2020.3025075. URL https://doi.org/10.1109/TCAD.2020.3025075.

[66] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Trans. Parallel Distributed Syst.*, 33(6):1303–1320, 2022. doi: 10.1109/TPDS.2021.3104255. URL https://doi.org/10.1109/TPDS.2021.3104255.

[67] Wen-mei W. Hwu, Thomas M. Conte, and Pohua P. Chang. Comparing Software and Hardware Schemes For Reducing the Cost of Branches. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 224–233. ACM, 1989. doi: 10.1145/74925.74951. URL https://doi.org/10.1145/74925.74951.

[68] Intel. Intel oneTBB. `https://github.com/oneapi-src/oneTBB`, 2023. [Online; accessed March 2023].

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

[70] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. APT-GET: profile-guided *timely* software prefetching. In *Proceedings of the Seventeenth EuroSys Conference*, pages 747–764. ACM, 2022. doi: 10.1145/3492321.3519583. URL `https://doi.org/10.1145/3492321.3519583`.

[71] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.*, 8(6):642–653, 2015. doi: 10.14778/2735703.2735704. URL `http://www.vldb.org/pvldb/vol8/p642-Jha.pdf`.

[72] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT 2009, 12th International Conference on Extending Database Technology*, volume 360 of *ACM International Conference Proceeding Series*, pages 24–35. ACM, 2009. doi: 10.1145/1516360.1516365. URL `https://doi.org/10.1145/1516360.1516365`.

[73] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.*, 11(11):1702–1714, 2018. doi: 10.14778/3236187.3236216.

[74] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS*, pages 6:1–6:11. ACM, 2014. doi: 10.1145/2676870.2676883. URL `https://doi.org/10.1145/2676870.2676883`.

[75] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008. doi: 10.14778/1454159.1454211.

[76] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything You Always Wanted to Know

About Compiled and Vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018. doi: 10.14778/3275366.3275370. URL http://www.vldb.org/pvldb/vol11/p2209-kersten.pdf.

[77] Timo Kersten, Viktor Leis, and Thomas Neumann. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in umbra. *VLDB J.*, 30(5):883–905, 2021. doi: 10.1007/s00778-020-00643-4. URL https://doi.org/10.1007/s00778-020-00643-4.

[78] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 163–181. USENIX Association, 2021. URL https://www.usenix.org/conference/osdi21/presentation/khan.

[79] Wooyoung Kim and Michael J. Voss. Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Softw.*, 28(1):23–31, 2011. doi: 10.1109/MS.2011.12. URL https://doi.org/10.1109/MS.2011.12.

[80] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A numa-aware in-memory storage engine for analytical workload. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS*, pages 74–85, 2014. URL http://www.adms-conf.org/2014/adms14_kissinger.pdf.

[81] Andi Kleen. A NUMA API for Linux. http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf, 2005. [Technical Linux Whitepaper; Online; accessed October 2022].

[82] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building Efficient Query Engines in a High-Level Language. *Proc. VLDB Endow.*, 7(10):853–864, 2014. doi: 10.14778/2732951.2732959. URL http://www.vldb.org/pvldb/vol7/p853-klonatos.pdf.

[83] Petr Kobalicek. asmjit: Low-latency machine code generation. https://github.com/asmjit/asmjit, 2014. [Online; accessed April 2023].

[84] Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.*, 9(4):252–263, 2015. doi: 10.14778/2856318.2856321.

[85] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive Execution of Compiled Queries. In *34th IEEE International Conference on Data Engineering, ICDE*, pages 197–208. IEEE Computer Society, 2018. doi: 10.

1109/ICDE.2018.00027. URL https://doi.org/10.1109/ICDE.2018.00027.

[86] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR*, 2015. URL http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf.

[87] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering, ICDE*, pages 613–624. IEEE Computer Society, 2010. doi: 10.1109/ICDE.2010.5447892. URL https://doi.org/10.1109/ICDE.2010.5447892.

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

[89] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proc. VLDB Endow.*, 12(5):502–515, 2019. doi: 10.14778/3303753.3303757. URL http://www.vldb.org/pvldb/vol12/p502-lang.pdf.

[90] Jaekyu Lee, Hyesoon Kim, and Richard W. Vuduc. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.*, 9:2:1–2:29, 2012. doi: 10.1145/2133382.2133384.

[91] Philip L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981. doi: 10.1145/319628.319663.

[92] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 International Conference on Management of Data, SIGMOD*, pages 743–754. ACM, 2014. doi: 10.1145/2588555.2610507.

[93] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *IEEE 30th International Conference on Data Engineering*, pages 580–591. IEEE Computer Society, 2014. doi: 10.1109/ICDE.2014.6816683.

[94] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN*, pages 3:1–3:8. ACM, 2016. doi: 10.1145/2933349.2933352.

[95] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019. URL `http://sites.computer.org/debull/A19mar/p73.pdf`.

[96] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE*, pages 302–313. IEEE Computer Society, 2013. doi: 10.1109/ICDE.2013.6544834.

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

[98] Raymond A. Lorie. XRM - an extended (n-ary) relational memory. *Research Report / G / IBM / Cambridge Scientific Center*, G320-2096, 1974.

[99] Darko Makreshanski, Justin J. Levandoski, and Ryan Stutsman. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *Proc. VLDB Endow.*, 8(11):1298–1309, 2015. doi: 10.14778/2809974.2809990. URL `http://www.vldb.org/pvldb/vol8/p1298-makreshanski.pdf`.

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

[101] Stefan Manegold, Peter A. Boncz, and Niels Nes. Cache-Conscious Radix-Decluster Projections. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 684–695. Morgan Kaufmann, 2004. doi: 10.1016/B978-012088469-8.50061-9. URL `http://www.vldb.org/conf/2004/RS18P3.PDF`.

[102] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the Seventh EuroSys Conference*, pages 183–196. ACM, 2012. doi: 10.1145/2168836.2168855.

[103] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *J. Parallel Distributed Comput.*, 70(12):1204–1219, 2010. doi: 10.1016/j.jpdc.2010.08.015. URL `https://doi.org/10.1016/j.jpdc.2010.08.015`.

[104] Sanyam Mehta, Zhenman Fang, Antonia Zhai, and Pen-Chung Yew. Multi-stage coordinated prefetching for present-day processors. In *International Conference on Supercomputing*, pages 73–82. ACM, 2014. doi: 10.1145/2597652.2597660.

[105] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.*, 11(1): 1–13, 2017. doi: 10.14778/3151113.3151114.

[106] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distributed Syst.*, 15(6):491–504, 2004. doi: 10.1109/TPDS.2004.8.

[107] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996. doi: 10.1145/248052.248106.

[108] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *44th International Conference on Parallel Processing (ICPP)*, pages 739–748. IEEE Computer Society, 2015. doi: 10.1109/ICPP.2015.83.

[109] Jan Mühlig and Jens Teubner. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD*, pages 1331–1344. ACM, 2021. doi: 10.1145/3448016.3457268.

[110] Jan Mühlig and Jens Teubner. Micro Partitioning: Friendly to the Hardware and the Developer. In *19th International Workshop on Data Management on New Hardware, DaMoN*, pages 27–34. ACM, 2023. doi: 10.1145/3592980.3595310. URL `https://doi.org/10.1145/3592980.3595310`.

[111] Jan Mühlig, Michael Müller, Olaf Spinczyk, and Jens Teubner. mxkernel: A Novel System Software Stack for Data Processing on Modern hardware. *Datenbank-Spektrum*, 20(3):223–230, 2020. doi: 10.1007/s13222-020-00357-5. URL `https://doi.org/10.1007/s13222-020-00357-5`.

[112] Michael Müller, Thomas Leich, Thilo Pionteck, Gunter Saake, Jens Teubner, and Olaf Spinczyk. He..ro DB: A Concept for Parallel Data Processing on Heterogeneous Hardware. In *Architecture of Computing Systems - ARCS*, volume 12155 of *Lecture Notes in Computer Science*,

pages 82–96. Springer, 2020. doi: 10.1007/978-3-030-52794-5_7. URL https://doi.org/10.1007/978-3-030-52794-5_7.

[113] René Müller, Jens Teubner, and Gustavo Alonso. Streams on Wires - A Query Compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, 2009. doi: 10.14778/1687627.1687654. URL http://www.vldb.org/pvldb/vol2/vldb09-622.pdf.

[114] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011. doi: 10.14778/2002938.2002940. URL http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[115] Thomas Neumann and Michael J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR*. www.cidrdb.org, 2020. URL http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[116] Thomas Neumann and Viktor Leis. Compiling Database Queries into Machine Code. *IEEE Data Eng. Bull.*, 37(1):3–11, 2014. URL http://sites.computer.org/debull/A14mar/p3.pdf.

[117] Stefan Noll, Norman May, Alexander Böhm, Jan Mühlig, and Jens Teubner. From the Application to the CPU: Holistic Resource Management for modern database management systems. *IEEE Data Eng. Bull.*, 42(1):10–21, 2019. URL http://sites.computer.org/debull/A19mar/p10.pdf.

[118] Wei Pan, Zhanhuai Li, Yansong Zhang, and Chuliang Weng. The New Hardware Development Trend and the Challenges in Data Management and analysis. *Data Sci. Eng.*, 3(3):263–276, 2018. doi: 10.1007/s41019-018-0072-6. URL https://doi.org/10.1007/s41019-018-0072-6.

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

[120] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research, CIDR*. www.cidrdb.org, 2017. URL http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf.

[121] Stefan M. Petters. Bounding the execution time of real-time tasks on modern processors. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA)*, pages 498–502. IEEE Computer Society, 2000. doi: 10.1109/RTCSA.2000.896433. URL https://doi.org/10.1109/RTCSA.2000.896433.

[122] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999. doi: 10.1145/330249.330250. URL https://doi.org/10.1145/330249.330250.

[123] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the 2014 International Conference on Management of Data, SIGMOD*, pages 755–766. ACM, 2014. doi: 10.1145/2588555.2610522. URL https://doi.org/10.1145/2588555.2610522.

[124] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *Proc. VLDB Endow.*, 11(2):230–242, 2017. doi: 10.14778/3149193.3149202.

[125] Peter P. Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real Time Syst.*, 1(2):159–176, 1989. doi: 10.1007/BF00571421. URL https://doi.org/10.1007/BF00571421.

[126] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler orchestrated prefetching via speculation and predication. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 189–198. ACM, 2004. doi: 10.1145/1024393.1024416. URL https://doi.org/10.1145/1024393.1024416.

[127] Jun Rao and Kenneth A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 International Conference on Management of Data, SIGMOD*, pages 475–486. ACM, 2000. doi: 10.1145/342009.335449. URL https://doi.org/10.1145/342009.335449.

[128] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. ISBN 978-0-596-51480-8. URL http://www.oreilly.com/catalog/9780596514808/index.html.

[129] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: the linux b-tree filesystem. *ACM Trans. Storage*, 9(3):9, 2013. doi: 10.1145/2501620.2501623. URL https://doi.org/10.1145/2501620.2501623.

[130] Kazuki Sakamoto and Tomohiko Furumoto. *Grand Central Dispatch"*, pages 139–145. 2012. ISBN 978-1-4302-4117-1. doi: 10.1007/978-1-4302-4117-1_6.

[131] Michael L. Samuel, Anders Uhl Pedersen, and Philippe Bonnet. Making CSB+-Tree Processor Conscious. In *Workshop on Data Management on New Hardware, DaMoN*, 2005. URL http://www-2.cs.cmu.edu/%7Edamon2005/damonpdf/2%20making%20csb+%20trees%20processor%20conscious.pdf.

[132] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD*, pages 351–362. ACM, 2010. doi: 10.1145/1807167.1807207. URL https://doi.org/10.1145/1807167.1807207.

[133] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *Proc. VLDB Endow.*, 8(9):934–937, 2015. doi: 10.14778/2777598.2777602. URL http://www.vldb.org/pvldb/vol8/p934-schuhknecht.pdf.

[134] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push vs. Pull-Based Loop Fusion in Query Engines. *CoRR*, abs/1610.09166, 2016. URL http://arxiv.org/abs/1610.09166.

[135] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*, pages 1907–1922. ACM, 2016. doi: 10.1145/2882903.2915244. URL https://doi.org/10.1145/2882903.2915244.

[136] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 510–521. Morgan Kaufmann, 1994. URL http://www.vldb.org/conf/1994/P510.PDF.

[137] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 81:1–81:12. ACM, 2015. doi: 10.1145/2807591.2807629. URL https://doi.org/10.1145/2807591.2807629.

[138] Hayden Kwok-Hay So and Robert W. Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable

computers using BORPH. *ACM Trans. Embed. Comput. Syst.*, 7 (2):14:1–14:28, 2008. doi: 10.1145/1331331.1331338. URL `https://doi.org/10.1145/1331331.1331338`.

[139] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *13st International Conference on High-Performance Computer Architecture*, pages 63–74. IEEE Computer Society, 2007. doi: 10.1109/HPCA.2007.346185.

[140] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD*, pages 307–322. ACM, 2018. doi: 10.1145/3183713.3196893. URL `https://doi.org/10.1145/3183713.3196893`.

[141] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP*, pages 18–32. ACM, 2013. doi: 10.1145/2517349.2522713.

[142] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289. ACM Press, 1996. doi: 10.1145/237090.237205. URL `https://doi.org/10.1145/237090.237205`.

[143] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth! In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*, 2022. URL `https://db.in.tum.de/~fent/papers/coroutines.pdf`.

[144] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-Tuning Query Scheduling for Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD*, pages 1879–1891. ACM, 2021. doi: 10.1145/3448016.3457260.

[145] Skye Wanderman-Milne and Nong Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, 37(1):31–37, 2014. URL `http://sites.computer.org/debull/A14mar/p31.pdf`.

[146] Ziqi Wang. index-microbench. `https://github.com/wangziqi2016/index-microbench`, 2018. [Online; accessed April 2023].

[147] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD*, pages 473–488. ACM, 2018. doi: 10.1145/3183713.3196895.

[148] Jan Wassenberg and Peter Sanders. Engineering a Multi-core Radix Sort. In *Euro-Par 2011 Parallel Processing - 17th International Conference*, volume 6853 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2011. doi: 10.1007/978-3-642-23397-5_16. URL `https://doi.org/10.1007/978-3-642-23397-5_16`.

[149] Thomas Willhalm and Nicolae Popovici. Putting intel® threading building blocks to work. In *Proceedings of the 1st International Workshop on Multicore Software Engineering, IWMSE*, pages 3–4. ACM, 2008. doi: 10.1145/1370082.1370085.

[150] Peng Wu, Arun Kejariwal, and Calin Cascaval. Compiler-Driven Dependence Profiling to Guide Program Parallelization. In *Languages and Compilers for Parallel Computing, 21th International Workshop, LCPC*, volume 5335 of *Lecture Notes in Computer Science*, pages 232–248. Springer, 2008. doi: 10.1007/978-3-540-89740-8_16. URL `https://doi.org/10.1007/978-3-540-89740-8_16`.

[151] Johannes Wust, Martin Grund, and Hasso Plattner. TAMEX: a task-based query execution framework for mixed enterprise workloads on in-memory databases. In *43. Jahrestagung der Gesellschaft für Informatik, Informatik angepasst an Mensch, Organisation und Umwelt, INFORMATIK*, volume P-220 of *LNI*, pages 487–501. GI, 2013. URL `https://dl.gi.de/20.500.12116/20773`.

[152] Xin Zhao, Jin Zhou, Hui Guan, Wei Wang, Xu Liu, and Tongping Liu. NumaPerf: predictive NUMA profiling. In *International Conference on Supercomputing*, pages 52–62. ACM, 2021. doi: 10.1145/3447818.3460361.

[153] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28 (2):17–22, 2005. URL `https://ir.cwi.nl/pub/11098/11098B.pdf`.

List of Figures