MxTasks: How to Make Efficient Concurrency Control and Prefetching Easy

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

Jens Teubner
TU Dortmund University, Germany
jens.teubner@cs.tu-dortmund.de

ABSTRACT

The hardware environment has changed rapidly in recent years: Many cores, multiple sockets, and large amounts of main memory have become a commodity. To benefit from these highly parallel systems, the software has to be adapted. Sophisticated latch-free data structures and algorithms are often meant to address the situation. But they are cumbersome to develop and may still not provide the desired scalability.

As a remedy, we present `MxTasking`, a task-based framework that assists the design of latch-free and parallel data structures. `MxTasking` also eases the information exchange between applications and the operating system, resulting in novel opportunities to manage resources in a truly hardware- and application-conscious way. As such, `MxTasking` also forms the basis for our vision of a truly co-designed system of operating system and database management system.

PVLDB Reference Format:
Jan Mühlig, Jens Teubner. A Sample Proceedings of the VLDB Endowment Paper in LaTeX Format. *PVLDB*, 14(xxx): xxxx-yyyy, 2021.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

The basic architectures of both Operating Systems (OSs) and Database Management Systems (DBMSs) in use today were designed decades ago. Since their inception, the hardware landscape has changed significantly: Today's servers have many cores distributed across multiple sockets, big caches, and large amounts of main memory, structured as Non-Uniform Memory Access (NUMA). While the hardware keeps changing, software has to adapt in order to benefit from the newly available resources.

In this light, during the past years, researchers have invested great efforts to increase parallelism, e.g. through very fine-grained latching mechanisms or by avoiding latches altogether [19, 29, 28, 26]. But in spite of the progress made, it remains difficult to design latch-free algorithms and data structures. *Transactional Memory*, e.g. in the form of Hardware Transactional Memory (HTM), promises to assist developers in the transformation of serial algorithms into parallel code. Again, progress has been made; but it also was shown how hard it is to outperform well-engineered "classical" code with HTM alternatives [30, 27].

Another approach is to divide the work into small, closed units of work, called tasks. Frameworks such as Intel® Threading Building Blocks (TBB) [35], Apple Grand Central Dispatch [37], or Wool [14] make usage of this concept. The idea of those frameworks is to enable developers to design parallel software without having to worry about the underlying many-core hardware. For this purpose, they offer sophisticated implementations for synchronization and automatic load balancing primitives. Yet, it remains the programmer's responsibility to apply them carefully; and experience shows that it is hard to exploit the full potential of parallel computing units [12] this way.

In this paper, we present `MxTasking`, a task-based environment for today's and future many-core hardware. The basic abstraction in `MxTasking` is the `MxTask`. An `MxTask` is a short program sequence that performs a single, small unit of work. `MxTask`s are guaranteed to run uninterruptedly to completion.

The true power of `MxTasking` lies in the possibility to attach *annotations* to every `MxTask`. With annotations, applications may convey characteristics of a task to `MxTasking`, for instance *runtime characteristics* (such as expected resource needs); information about related *data objects* (including access information such as read or write access); or desired *scheduling priorities*. `MxTasking` will then use such knowledge to optimize resource allocation, scheduling, and placement.

In this work, we will also report on a particularly powerful class of annotations: *synchronization annotations*. Rather than manually implementing and tuning intricate and error-prone synchronization mechanisms (spinlocks, reader/writer locks, version locks, ...), developers may simply express their desired type of isolation as a task annotation. `MxTasking` will take care of the rest and inject the synchronization primitive that works best for the current system and application state. This may significantly ease the development of massively parallel applications.

The use of specific knowledge about application behavior for scheduling or resource allocation has been explored previously in the context of *database/operating system co-design* (albeit highly targeted at database applications only) [16]. With immediate control over underlying resources, such de-

Proceedings of the VLDB Endowment, Vol. 14, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

signs can better address application specifics than commodity operating systems could with their narrow application/OS interfaces. And indeed, as we also show in this work, `MxTasking` can be extended into a *bare-metal runtime environment*, pushing the concept of task-based execution further and seamlessly leveraging, e.g., heterogeneous hardware such as CPUs, GPUs, or FPGAs.

The rest of this paper is organized as follows: Section 2 gives an overview of related work. Afterward, Section 3 presents details of the `MxTasking` runtime and annotation principles, before we discuss task-based synchronization in Section 4. In Section 5, we provide practical insights into our tasking library. Our vision of the `MxKernel` will be discussed in Section 6. First results of a key-value store built with `MxTask`s are demonstrated and discussed in Section 7. We conclude in Section 8.

2. BACKGROUND AND RELATED WORK

Our work directly relates to three areas of systems research: task-based parallelism, DB/OS Co-Design, and scalable index data structures (the latter will be the poster child to illustrate and evaluate the potential of `MxTasking`).

Task-based parallelism. The idea of asynchronous and fine-grained control flows has been discussed several times in the recent past. Many programming languages and environments implement this approach, for example, *NodeJS*, *C++*, and *Rust*. With *Cilk*, Blumofe et al. published one of the first runtime systems for parallel programming that schedules lightweight tasks onto OS threads [9]. Targeting to simplify the engineers' work, Cilk focuses on the automatic load balancing of parallel applications and comfortable integration into existing software programs.

Inspired by Cilk, Intel® designed the TBB framework focusing on portability and robust performance [35, 23]. The latter one is primarily done by using a work-stealing mechanism within the scheduler, balancing the load over the worker threads. TBB provides several synchronization primitives such as scalable (reader/writer-) latches, partially based on HTM. Similar objectives are pursued by the *Wool* framework through a comparable work-stealing strategy [14]. However, according to the authors, Wool is easier to embed into existing software.

StarPU intends to provide fine granular tasks for heterogeneous multicore platforms [4]. The authors argue that the modern hardware landscape not only features much parallelism based on CPU cores but also uses special co-processors. StarPU offers a framework that supports both CPU parallelism and co-processors such as GPUs.

Besides the already mentioned frameworks, *morsels* have been exploited in the context of DBMS, specifically in the HyPer engine–[25]. Morsels are kind of tasks, that execute segments of a query. The scheduler of the query execution framework takes care of NUMA-local execution and can be load-balanced at runtime, for example at time the workload changes.

Bang et al. utilized tasks for various index structures like B-trees and hash-tables, as well as transactional workloads [5]. The main idea is to divide a given multi-socket machine into several domains. By allocating data structures within a concrete domain, the accessing tasks are implicit processed NUMA-aware.

In the course of growing heterogeneity, Tözün and Kotthaus provide a concept for scheduling database tasks heterogeneous hardware [40]. Their work discusses the challenges about the granularity of tasks how to decide on which specific processing unit they should be scheduled.

Based on coroutines, Psaropoulos et al. invented fine granular tasks for index joins to minimize memory latencies [33]. Everytime a coroutine attempts to access not cached memory, a `prefetch` instruction is executed and the coroutine is yielded. By that, the CPU is enabled to execute another coroutine while the memory subsystem loads the requested data into the cache instead of wasting cycles to wait for the load fulfilled.

DBMS/OS Co-Design. Related to DBMS/OS Co-Design, Giceva et al. designed *COD*, a system that provides a richer interface between OS and DBMS [16]. With this, applications can transmit special requirements and knowledge about internal processes to the OS. Based on the System Knowledge Base (SKB), application and OS are enabled to exchange their knowledge about application requirements and OS state. The OS, on the one hand, can schedule the use of existing hardware resources more efficiently. The DBMS, on the other hand, has the possibility to discover the specific hardware configuration of the underlying system through a service provided by the OS. The concept of the SKB was already introduced by the Barrelfish OS [38, 6], a multikernel that challenges problems associated with the growing number of cores and heterogeneity. We will relate this to our approach later in Section 6.

Scalable index data structures.. Traditionally, B-trees are synchronized by latch-coupling, where a node's latch is released after the child's one is successfully acquired [7]. While latch-coupling enables holding at a max two latches at a time, the B^{link}-tree reduces this to only one [24].

Since latching has become a bottleneck for in-memory data structures on modern hardware, research investigated optimistic or fully latch-free procedures. Leis et al. presented *Optimistic Latch Coupling*, an optimization for latch-coupling which is simple to implement but offers scalability on less conflicted data structures [28, 26]. This concept permits parallel read and write operations, detects modifications by a version counter and repeats the operation if necessary. The *OLFIT* algorithm earlyer described a very similar approach for B+-trees and CSB+-trees [10].

The *Bw-tree*, introduced by Levandoski et al., represents a complete latch-free B-tree, focussing on many-core scalability and cache performance [29, 42]. This is mainly realized by applying modifications as delta records in a latch-free manner.

3. MXTASKING

With the shift of the hardware landscape toward massively parallel, heterogeneous architectures, the expectations toward software have become immense: software is supposed to leverage parallelism for scalability; exploit heterogeneous hardware for efficiency; use fine-grained synchronization for correctness; and tune cache and memory accesses for performance. And to make matters worse, most of these challenges are still each developer's responsibility, with only little assistance from the system software underneath.

We argue that this is also due to the prevalent control flow abstraction that essentially dates back to the 1960s: *threads*. Threads are essentially opaque about their runtime characteristics; schedulers—e.g., in operating systems—have to guess each program's intentions. Conversely, runtime systems tend to hide ("abstract") most hardware details away from application programs.

3.1 `MxTask` Abstraction

A key proposition of `MxTasking`, therefore, is to replace the application-facing control flow abstraction by what we call *tasks*. An `MxTask` corresponds to a small, closed unit of work, rather than to the sequence of straight-line code that a thread would correspond to. With tasks as an abstraction, it becomes surprisingly natural to convey precisely the information about application characteristics that the runtime system needs in order to optimize resource utilization. In `MxTasking`, such information can be attached to every `MxTask` in the form of *annotations*.

A task will typically process a single (or few) *data objects*. A complete, higher-level algorithm will be composed of a (possibly large) number of `MxTask`s that jointly solve the given application problem.

To illustrate the tasking concept in this paper, we will use *tree navigation* as a running example. `MxTasking` can be applied here by spawning a new task for every node visited during a tree traversal. That is, each task will visit a single tree node and spawn a new task (to process the next node) just before it finishes. Observe how execution strategies like *morsel-based parallelism* [25] naturally fit into this model.

Spawning a task is an extremely lightweight operation, implemented using efficient assembly atomic instructions. Spawned tasks will asynchronously be moved to a *task pool*, from where the `MxTasking` runtime will select tasks for execution (possibly based on annotated information).

3.2 Annotation-Based Memory Prefetching

The optimization of memory access patterns is a good example to illustrate how annotated `MxTask`s can significantly ease the development effort for modern hardware, while at the same time improving runtime efficiency.

Particularly for data processing systems, *memory access* has become the key factor when it comes to efficiency and performance. *Prefetching* memory contents into CPU caches can be an excellent means to hide memory access latency and improve performance. However, effective prefetching is intricate to achieve: if the prefetch request is issued too late, hardware will not have enough time to actually bring the data into memory; if the prefetch distance is too wide, data might already get evicted from the cache again before it is actually used.

Prefetch requests may be issued either by hard- or software. But while efforts have been made to teach caching hardware the access patterns of database code [22, 41, 43, 17], hardware prefetching remains unfeasible beyond stream-based look-aheads. Software-based prefetching was shown to be more effective (e.g., [11, 21, 31, 34]), but depends on substantial algorithm restructuring to work out.

In a task-based execution environment, efficient prefetching is surprisingly simple. When making scheduling decisions, the `MxTasking` scheduler will consult the task pool, giving it an understanding of the upcoming tasks for the near future. Whenever tasks are annotated with the data object that they access—which we assume they are, because this is a trivial annotation to make—, `MxTasking` will automatically inject software prefetching instructions on the application's behalf.

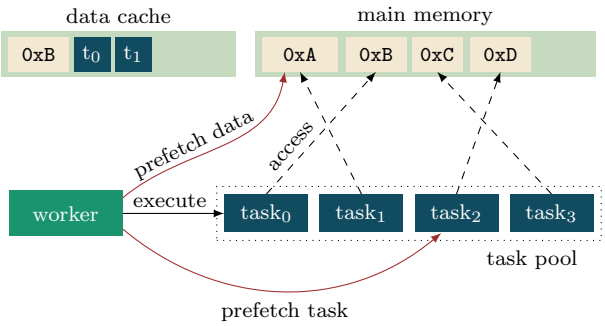


Figure 1: Execution and prefetching of `MxTask`s. The worker knows tasks that will be executed soon and prefetches both tasks and data objects early enough to hide memory latencies.

Thereby, we catch two birds with one stone. Prefetching becomes simple on the application end. In fact, the prefetching mechanism in `MxTasking` is completely transparent to the application developer. All she needs to do is provide proper data object annotations to `MxTask`s. At the same time, the prefetching mechanism is significantly more powerful than existing approaches. In contrast to hand-crafted solutions, `MxTasking` will automatically schedule prefetch instructions even across task executions from different applications. Plus, there is now only a single point in the system where details, such as the prefetch distance, can be configured. Though not realized in our current implementation, it is also conceivable to dynamically adapt prefetching, e.g., to data locality in NUMA environments.

3.3 Implementation

`MxTasking` is a layer between task-based applications and the operating system.[1] From the application's perspective, spawning a task adds it to the task pool of a *logical core*. This is a lock-free operation, making task spawns a very lightweight operation.

From the operating system's perspective, each of the `MxTasking` logical cores corresponds to a *worker thread* that will pick tasks from the pool and execute them. In this sense, `MxTasking` mediates between the task-based execution model and the thread model of the underlying operating system. In our implementation, we further pin all worker threads to a dedicated CPU core, which gives `MxTasking` control over NUMA and locality effects.

Whenever the worker thread picks an `MxTask`, that task will be executed uninterruptedly to completion. Often, tasks will spawn further tasks. Such spawning will happen asynchronously and lightweight.

The resulting task pool enables the `MxTasking` runtime to already "see" upcoming tasks as well as their associated memory objects. With this information, the scheduler can inject prefetch instructions in-between task executions, such that tasks will see their data already cached in the CPU

[1] We currently support Linux or `MxKernel` as the underlying operating system.

when they start. To this end, the scheduler has to invoke *two* prefetches: first, the *task descriptor* has to be brought into the CPU cache, since it includes the information about associated *data objects*; with the task descriptor in the cache, the scheduler read out that information and prefetch data objects as the second step.

Figure 1 illustrates this mechanism (assuming a prefetch distance of 1). Before executing $task_0$, the worker will issue prefetch request for the *descriptor* of $task_2$. This prepares the scheduler to prefetch data for $task_2$ in the next iteration. Already in the current iteration, the scheduler will prefetch *data object* associated with $task_1$, so that $task_1$ will find its data cached when it gets executed in the next iteration.

4. ANNOTATION-BASED SYNCHRONIZATION

Building parallel software has become a challenge since hardware offers massive numbers of cores. Finding the perfect granularity for the scope of latches is not always trivial. Tasks present an excellent base for this. They are fine-granular by design, accessing just one or a few data objects during execution. The developer is almost endorsed by the use of tasks to design the software fine-grained. At the same time, this assists the execution layer in understanding the developer's requirements much better. With the help of a handful of additional information, parallel tasks can be coordinated very comfortably, without the developer having to protect explicit accesses from each other (e.g., by using latches). Concurrency no longer becomes an enemy to be fought, but parallelism feels natural.

`MxTasking` uses annotations to pass information from the application to the execution layer. Annotating tasks with accessed data objects and type of access (reading or writing), as well as the synchronization requirements of data objects enables `MxTasking` to provide synchronization without further assistance of the developer.

4.1 Integrated synchronization mechanisms

`MxTasking` selects the appropriate one from various techniques, depending on the required parallelism of a data object, such as parallel read operations or complete mutual exclusion.

Latches. Spinlocks are known for their easy realization and simple usage. As in thread-based implementations, we can also apply spinlocks to synchronize concurrent tasks. `MxTasking` provides different spinlock variants. To supply mutual exclusion, a simple spinlock is used, which sequences all accesses, whether tasks are read-only or not. Given an application that enables parallel reads on a shared object, `MxTasking` chooses a reader/writer-lock instead. Acquiring and releasing the latch is done by the worker thread. Once the annotations of both the task next in line and corresponding data object have been evaluated, the related latch is acquired to guarantee safe access. After task execution, it will be released.

Sequencing by scheduling. `MxTask`s are governed by a run-to-completion semantic whereby all tasks scheduled to the same task pool are implicitly sequenced. Moreover, and this distinguishes `MxTasking` from other known libraries, once dispatched, tasks are not stolen from other task pools.[2] Following this, we can avoid latches for concurrent accesses by scheduling all tasks accessing the same data object to the same task pool.

This approach eliminates latch contention and time spinning on the latch until it is consumable. At the same time, we need to deal with an association between data objects and task pools to balance the load throughout available computing resources. To balance the load, data objects with foreseeable frequent access can be annotated accordingly. These are taken into account when associating data object and task pool, ensuring the load to distribute evenly.

To give an example, consider a tree-like structure. Inner nodes, particularly the root node, will be accessed for more often than leaf nodes. Hence, the mapping algorithm has to take care of access frequencies to provide load balancing over available computing resources. Task pools, which have (by annotation) frequently requested data objects assigned to them, obtain a reduced total number of objects. The task pool associated with the root node of a tree, for instance, is linked to only few other nodes.

However, like basic spinlocks, sequencing the accesses to a data object by scheduling represents a pessimistic synchronization method, where `MxTasking` does not distinguish between reading and modifying tasks.

This combination of scheduling and synchronization to avoid latches is unique and not known to us in any other task-based library. While load balancing across all resources becomes a challenge, it allows us to make more efficient use out of the data cache. Replication of data across multiple caches will be reduced since each data object is accessed by a distinct core.

Optimistic versioning. Avoiding latches might be a good idea, in particular for write-heavy workloads. However, read-operations dominate many database-related workloads. For instance, the root node of a tree-structure modifies over time, but most accesses will be read-only to locate the next node on traversing the tree. Optimistic approaches have proven their worth to benefit from parallel hardware [10, 28]. Read accesses are executed in parallel while concurrent write accesses are protected from each other by latches. To detect overlapping read and write operations, the resource is versioned by a counter which increments after each modification. Read-only procedures will check the counter before and after reading. When the version has changed during the access, it must be repeated. `MxTasking` generalizes this concept as follows.

"Writing" annotated `MxTask`s modifying the same resource are scheduled to the same task pool, avoiding latches due to sequencing. Reading tasks, however, are free to be executed by any worker thread. Before and after the execution of a read-only task, `MxTasking` checks the version counter of the data object. If it increased during the read operation, the task is reset, rescheduled, and executed again at a later time. Hence, write accesses continue to be sequenced on a specific worker thread, while read accesses act in parallel. The additional effort for the developer remains modest: She only has to annotate tasks as read-only or modifying, while `MxTasking` takes care of version management.

[2]Rather, `MxTasking` can steal entire task pools in order to improve load balancing while keeping synchronization lightweight.

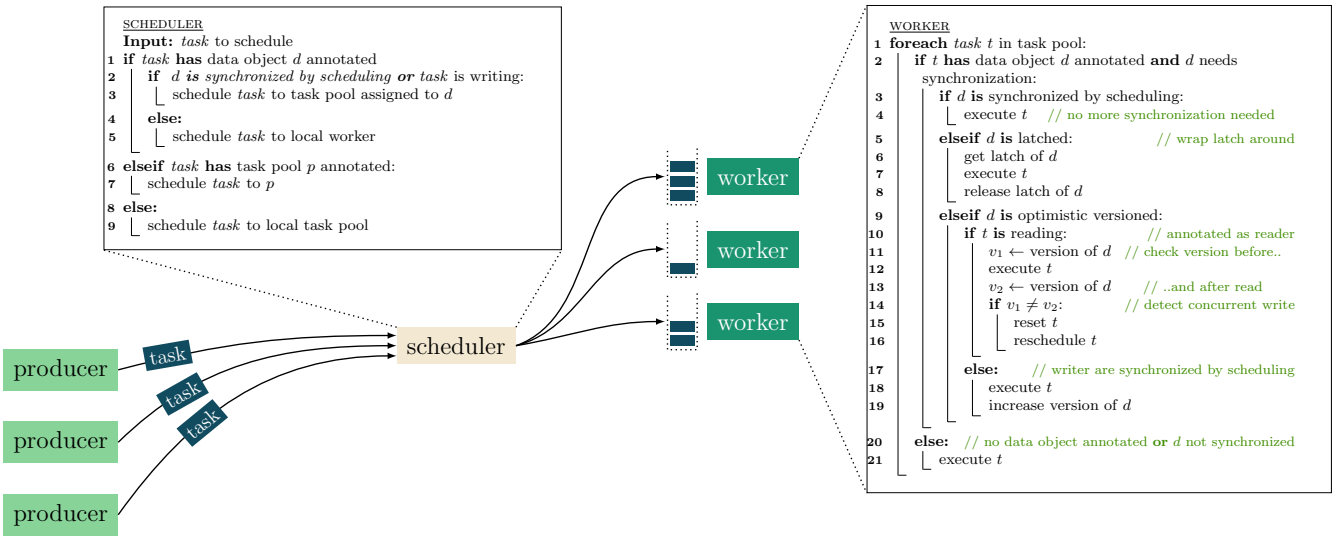


Figure 2: Interaction of *scheduler* and *worker thread*. The scheduler ensures the sequencing of writing tasks by scheduling them to the identical task pool. The worker, however, embeds synchronization of tasks in the execution of those.

Comparable to other optimistic procedures, operations (or tasks in our context) deleting a shared object must be treated with special care. Hazard pointers [32] and memory reclamation [18] are general approaches to protect read accesses while another thread frees the data at the same time. Even if currently not realized in `MxTasking`, it would be straightforward to keep track of which worker thread is reading which (to a task annotated) data object. With this knowledge, competing read and delete operations can be coordinated to avoid faults.

4.2 Embedded task synchronization

The fine granularity of tasks in combination with annotated metadata about the accessed data object, access pattern, and synchronization requirements provide adequate information for `MxTasking` to synchronize competing tasks. As a result, the application engineer has not to deal with concurrency and hands it over to `MxTasking`. Hence, the application logic and concurrency control are separated, which minimizes errors.

As illustrated in Figure 2, the synchronization of tasks is based on an interaction between scheduler and worker thread. The scheduler ensures to place tasks in the pool of the appropriate worker thread, depending on the synchronization mechanism and access type.

The scheduler side. To sequence a set of tasks, the scheduler places them in the same task pool. That is necessary when (a) optimistic versioning is applied and the task will write to the annotated object *or* (b) all accesses to a data object are synchronized by sequencing. For both cases, the scheduler selects the task pool associated with the annotated data object as a destination (lines 1–3). Otherwise, the scheduler prefers the local task pool to reduce scheduling overhead in the form of cache-coherence. Local, in this context, means the task pool of the worker thread that produced the task, potentially while executing another task.

Exceptions to this rule are annotations that affect the placement of tasks explicitly, for example, a required task pool (lines 6–7). It is also conceivable to annotate particular NUMA regions to support applications building NUMA aware software.

The worker side. The synchronization, whenever necessary, is performed by the worker, embedded in the execution of the tasks. First, the worker evaluates the annotated data object of the next task (line 2). Supposing no synchronization is needed or scheduling guarantees sequencing, the worker executes it directly (lines 21 and 3–4). Otherwise, we distinguish between the two additional mechanisms we discussed before. In case the accessed data object is synchronized using a latch, whether it is a pure spinlock or reader/writer-lock, the worker will acquire the latch related to the data object before executing the task and release it after execution (lines 5–8). Whenever possible, we will acquire the latch in *shared mode*.

For data objects that are synchronized by optimistic versioning, the worker thread separates between reading and writing tasks. To ensure a read data object was not modified, he checks the version before and after the execution (lines 10–16). If the counter differs, the read access has to be retried at a later time by scheduling the task again (lines 14–16). To make sure that the version changes at all, it is incremented after the execution of a writing task (line 19).

5. MXTASKING IN ACTION

Utilizing tasks to design data structures and algorithms differs in general from well-known thread-based programming. `MxTasking` advances this style beyond the current standard by offering annotations for prefetching and parallelism. This section reviews some practical aspects of using `MxTasks` for building parallel software. First of all, we illustrate the simplicity of designing a latch-free, task-based index structure, using a B^{link}-tree as an example. Accordingly, we will examine the relevance of memory management regarding task-allocation and show the benefits and limitations of `MxTasking`.

```
LOOKUP TASK
Task input: node the task accesses, key to lookup,
  callback to notify on finish
1  if highkey of node ≤ key:
       // key is out of range of this node
2      next ← right sibling of node
3      follow_up ← create a new task
4      annotate follow_up with next as reader
5      schedule follow_up
6  elseif node is of type inner:
       // continue traversal to the leaf
7      next ← child for key in node
8      follow_up ← create a new task
9      annotate follow_up with next as reader
10     schedule follow_up
11 else:
       // found correct leaf, read value
12     v ← get value of key in node
13     notify caller with v
```

Figure 3: Lookup operation in a task-based B^{link}-tree. Since `MxTasking` conducts synchronization, the application logic has not to deal with concurrent access.

5.1 Building a task-based B^{link}-tree

Whereas thread-based implementations result in synchronous method calls most of the time, `MxTask`s are asynchronous. For instance, instead of calling a *lookup* method on a data structure that returns after the wanted key-value pair is found, scheduling a *LookupTask* that notifies the caller with the result is the way to go.

Tasks only have a limited view of the system. Every `MxTask` solely performs on a single tree node, taking the appropriate node and the requested key as input parameters. The pseudocode in Figure 3 illustrates an example of implementing a lookup task. On every step, it examines whether the node it is working on is an inner or a leaf node. If the node is of type inner, the task has to determine the next node to traverse by applying a binary search (line 7). However, parallel insert operations may have modified the content of the node since the lookup task was scheduled. Sometimes, one of these insertions splits the node. At that point, a traversing task may have missed the direct pointer to the node containing the searched key for now. For that reason, every task checks the key-range of the given node and traverses to the right sibling when necessary (lines 2–5). That can also occur in traditional (thread) implementations and is—in general—part of the B^{link}-tree algorithm. Nevertheless, it is slightly more likely to happen using asynchronous models since the time between node accesses during a traversal may be increased.

For continuing the traversal, the task instantiates and schedules a follow-up task, annotated with the next node (lines 8–10). Labeling the follow-up task as a reader (line 9) enables `MxTasking` to execute it in parallel with other reading operations. In contrast, a thread-based implementation will call the `child` method in a loop until reaching a leaf node. Given a task executed on a leaf, it reads the value and notifies the caller (lines 11–13). A callback function, for instance, responds to a client's request in an end-to-end setting.

Figure 3 shows that an `MxTask` focusses on the application logic but contains no explicit mechanism for concurrency control, except annotations. The latter are used by `MxTasking` to realize the synchronization of competing tasks, as discussed in Section 4.

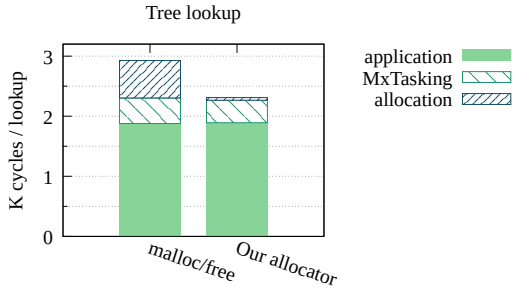


Figure 4: Aggregated CPU cycles for a lookup on a task-based tree, using `malloc` and our allocator for task allocation.

Implementing the corresponding *InsertTask* is straightforward. Instead of reading the value from the leaf node, it performs the insertion. If the operation causes a node-split, the *InsertTask* schedules a follow-up task that places the pointer to the newly created node in the parent node or produces a new tree root when the split node was the root. Until the pointer is attached to the parent node, the new node can be reached by following the sibling pointer, as explained before. Even the insert operation requires no explicit synchronization since the engineer annotates the task as a writer. That hints `MxTasking` to initiate the corresponding synchronization step, such as taking the writer latch or incrementing the version counter.

5.2 Task Allocation

As seen in the previous Section, `MxTask`s will be created and deleted frequently. Each operation on a tree structure, for example, corresponds to a separate task, which spawns several subsequent tasks. Hence, the allocation of those is a central component. Using the global heap may turn into a bottleneck because many cores will create new tasks at the same time.

Figure 4 demonstrates the CPU cycles spend during a single lookup on a task-based tree, including the traversal from the root to the leaf node. Allocating tasks using the system's `malloc` interface takes around 620 cycles per operation on a 48 core machine, which constitutes 21% of total consumed CPU cycles.

To overcome this costly aspect, we have designed a multi-level allocator, mainly inspired by *Hoard* [8]. Hoard focusses on fast, cache-aware, and scalable allocation. Dedicated memory heaps for every processor enable scalability. Threads allocate memory from their local processor heap instead of calling the system-wide `malloc` interface to request memory from the OS. Each processor heap holds a buffer of free memory and delegates it to threads that want to allocate memory. The processor heaps, in turn, demand memory from the OS when the local buffer becomes empty. That reduces synchronization costs between processors.

We extend this concept by supplying a third layer to the allocation stack: A separated heap per logical core that will be the point of contact for task allocation. Figure 5 outlines this approach. Whenever a task wants to create a new one,

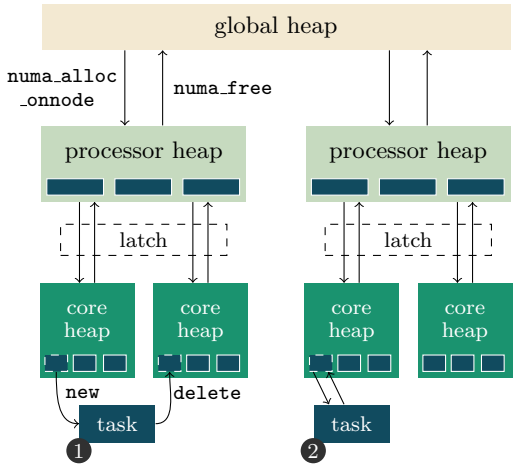


Figure 5: By using multiple levels for task allocation, synchronization is reduced and tasks are allocated cache-aware.

it requests the local per-core heap for memory. Allocations on the per-core heap do not need any synchronization because tasks will not be interrupted. Free memory blocks on this heap are stored within a LIFO list. Implicitly, the allocator places freed blocks at the top of the list (②). Thus, an allocation will use recently freed memory blocks, which increases the chance that the allocated task is still available in the CPU cache.

Reducing inter-processor communication and providing NUMA aware allocation is a trade-off. Figure 5 shows a task (①) that is allocated on one core but deleted on a different one. The free block will be pushed to the core-heap where the task is deleted. In the worst case, where a task is allocated and deleted among cores located in different NUMA regions, this shuffles memory blocks across them. However, we minimize synchronization and implicit communication costs between them.

At the time a core heap runs out of memory, it will request a new memory block from the processor heap. In turn, it will allocate memory from the global heap in a NUMA aware manner, when the processor heap has no memory in stock. As a result, memory management for tasks requires only a single latch in case of allocating memory from processor heap, reinforcing scalability. However, by reusing deleted tasks, this issue occurs rarely. Compared to using `malloc`, Figure 4 demonstrates that our multi-level allocator has almost no overhead. Only 45 cycles are spent in allocating tasks for a single tree-lookup.

5.3 Discussion

`MxTask`s provide an elegant way to design and build highly parallelized data structures and applications by offering implicit synchronization. In widespread paradigms, like threads or other task-based approaches, concurrency control is often error-prone and results in complex source code. Associating data objects and tasks facilitates `MxTasking` to manage concurrency control. That enables the developer to focus on application logic instead of fighting concurrency—almost getting parallelism as a bonus. With `MxTask`s, even the complex task of hiding memory latencies with software-based prefetching feels natural.

Still, the utilization of tasks-based programming is not as

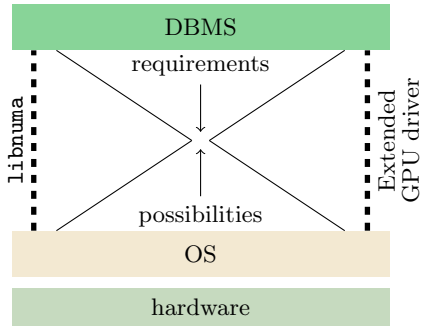


Figure 6: The interface between possibilities of the OS and requirements of the DBMS is narrow and often artificially supported by external libraries.

widespread as threads. The preferred (and most efficient) granularity of a task is not discussed in general by this work. However, the granularity has a notable impact on scheduling costs. As tasks become more fine-grained, they can be parallelized more accurately—but the scheduling costs also increase. Even since these are kept low by using a latch-free data structure for managing `MxTask`s, it is still an additional expense. Especially, when dispatching tasks from one NUMA region to another, costs in the form of cache-coherence and atomic instructions arise. These arrises, in particular, when we use the scheduling mechanism to synchronize competing tasks—but the avoidance of latches is an advantage (we will show in Section 7).

Nevertheless, the `MxTask`-based approach offers many possibilities that simplify the development of parallel data structures and applications. By clever use of annotations and implicit synchronization, as well as automatic prefetching of data objects, the mentioned disadvantages can be compensated.

6. THE VISION OF MXKERNEL

So far, we have considered `MxTasking` primarily as an additional layer on top of the OS. We argue that `MxTask`s can also be used as the elemental abstraction for control flows—replacing traditional threads altogether. In this Section, we present our vision of a DBMS/OS Co-Design.

To achieve optimal performance, high scalability, and robustness, the interaction of OS and DBMS is essential. General-purpose OSs try to provide a platform for a wide range of hardware, including various CPU architectures and different co-processors such as GPUs and FPGAs. To abstract these diversities, the OS unifies interfaces to the underlying system. As a side effect, it hides special features of individual devices and computing units for the user. Figure 6 illustrates how the database community posed this problem of interaction between DBMS and OS: OSs have all possibilities provided by the underlying hardware but do not know about the requirements of the applications running on top. Meanwhile, the DBMS knows all about the internal process and requirements. For example, the data distribution across the NUMA regions, which task accesses which data and the priority of those. Due to abstraction-related unified interfaces, the DBMS can not to share this information with the OS. Using external libraries like `libnuma` [20] or extended interfaces for GPUs like `OpenCL` [39] allows ap-

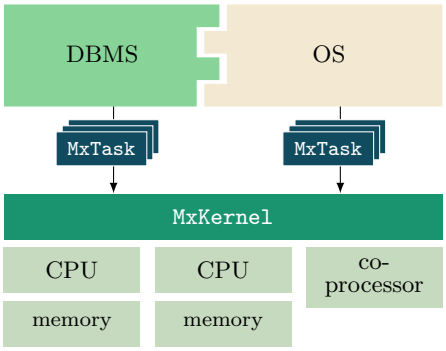


Figure 7: The architecture of `MxKernel`: Both OS and DBMS run on top of the bare-metal runtime and use `MxTasks` for control-flow abstraction.

plications to gain a closer view and more control over the underlying hardware components. The library `libnuma`, for example, enables NUMA aware memory allocation and thread scheduling. Nevertheless, the usage feels more like a crutch and does not solve the problem in general.

With our bare-metal platform `MxKernel`, we address the issue of insufficient interfaces between DBMS and OS. `MxKernel` acts as a lightweight layer between the hardware and performance-critical applications, such as DBMSs. As shown in Figure 7, both OS and DBMS run as equal peers on top of `MxKernel`. As a result, the DBMS can interact much more with the hardware and does not have to implement services based on OS components. In contrast to "traditional" environments, where the DBMS runs on top of the OS, both can share data structures and algorithms. That applies, for example, to index structures such as B-Trees, which are used not only for indexing data in DBMSs but also for file systems, e.g., *Btrfs* [36]. Moreover, the DBMS can implement critical services on its own responsibility, for example, memory and I/O management. Commercial systems such as DB2 [1] and Oracle [2] often re-implement services provided by the OS on top of them. That is not necessary anymore by using such a Co-Design offered by `MxKernel`.

In addition to exploring the hardware components present in the system, `MxKernel` primarily offers a lightweight abstraction of control flows. For this, the `MxTasking` introduced in Section 3 is utilized with one worker instantiated on each logical CPU core. Applications running on top of `MxKernel` employ `MxTasks` to accomplish their jobs. Due to the guaranteed ability to distribute tasks to a specific hardware resource like a particular CPU core, the applications have high control capabilities. Besides the already introduced advantages of `MxTasks`, these offer an elegant way to abstract heterogeneity.

Modern servers are often equipped with co-processors such as GPUs and FPGAs. Those processing units are used in the context of DBMS to solve specialized problems, e.g., query compilation on GPUs [15]. However, the software must be explicitly adapted to the use of these devices. Designing software in a way that all available computing resources can be used as efficiently as possible is complex and requires external libraries and frameworks. `MxTasks` exploit a way to use that heterogeneous hardware. Developers can provide implementations for various devices such as GPU and CPU for an `MxTask`. Based on execution times, which can be annotated by the developer, and the load factors of possible processing units, `MxKernel` will schedule the tasks to devices. For example, when an `MxTask` has implementations for both CPU and GPU, where the GPU variant promises more performance. `MxKernel` could execute the CPU variant of the task when other tasks use the GPU with high frequency.

Other works have already discussed and focused on closer cooperation between OS and DBMS (see related work in Section 2). COD [16] is a well-known and proven example. To offer a broader interface, COD provides a central infrastructure with detailed information about the hardware and the state of the OS. Applications running on top of the OS can use this central component to share requirements of the application-state and hints with the OS. This, in turn, can use this knowledge for resource allocation, e.g., memory and computing resources. The equivalent in `MxKernel` may the usage of annotated tasks. Those annotations hints the kernel with the application requirements that `MxKernel` also uses for resource allocation. As both, the application and the OS, run as equal peers on top of the runtime, they can benefit from each other's services and data structures. Since the application is close to the "bare iron", it has a detailed view of the hardware right from the very beginning. In contrast to COD, not the OS provides information about the system. Rather, it is `MxKernel` that offers equal privileges to all.

7. EXPERIMENTAL EVALUATION

In this section, we evaluate different aspects of `MxTasking` and examine the potential of tasks. To study the behavior and potential of the tasking approach in real-world scenarios, we use an in-memory B^{link}-tree implementation that is indicative of the behavior of modern in-memory database engines. Our implementation of the data structure follows state-of-the-art principles.

7.1 Environment

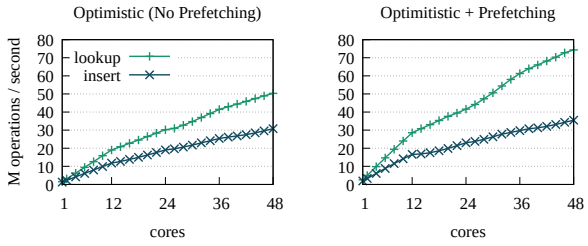
All benchmarks are evaluated on a two-socket Intel Xeon Gold 6226 machine, clocked at 2.7 GHz. Each of the two processors holds 12 cores, 24 hardware threads, and 1×32 kB L1, 12×1 MB L2, and 1×19.25 MB L3 data caches. The cores are ordered by NUMA regions, whereas the first 24 logical cores are located in the first region, the next 24 in the second. To be precise, the first 12 cores of each region are physical cores. Starting from then, hyperthreading cores are added step by step.

As a workload, we rely on the Yahoo! Cloud Serving Benchmark (YCSB) [13], using *Workload a* and two phases: Filling the tree with 100 million insert operations and performing 100 million lookups afterward. The tree stores pairs of 64b keys and 64b values within 1kB sized nodes.

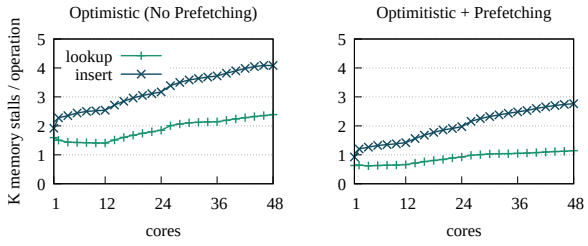
Ubuntu 20.04 was used as OS, *clang* 10.0.0 as compiler. All benchmarks are compiled with the highest optimization level. Because all threads are pinned to corresponding cores, we disabled the system's *NUMA balancing* option for all experiments. This way, the kernel will not migrate memory or threads between the regions.

7.2 Automatic prefetching

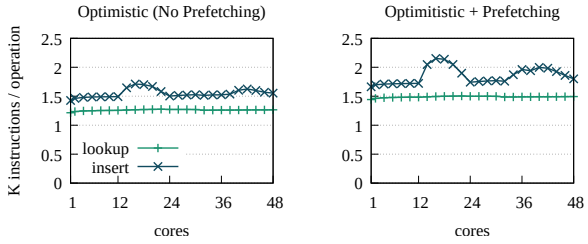
As discussed in Section 3.2, the fine granularity of tasks allows an exact prediction on which data an `MxTask` will access. The application engineer annotates tasks with accessed



(a) Throughput



(b) Memory stalls per operation



(c) Executed instructions per operation

Figure 8: Impact of software-based prefetching for the `MxTask`-based B^{link}-tree.

data objects. Besides, the scheduler has an ordered view of the next tasks in line.

This knowledge is used to pre-load tasks itself and annotated data from memory into the higher-level CPU cache, allowing to run worker threads right away, without waiting for memory. Figure 8 compares the B^{link}-tree build on `MxTasking` with and without the automatic prefetching mechanism. Since the benchmark is *memory bound* for the most times, automatically prefetching the nodes of the tree results in 48% more lookups and 15% more inserts per second, as demonstrated in Figure 8a. Especially the traversal of the tree, which bases on binary search, benefits from this mechanism. Because the search creates a hard-to-predict access pattern for the CPU, hardware prefetching has less effect.

The impact of software-based prefetching becomes particularly visible when observing memory stalls, shown in Figure 8b. Memory stalls are cycles in which the CPU actively waits for memory until it is available in the cache before continuing 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 benchmarks. Here, the number of memory stalls is reduced more than a half. The write-heavy workload also benefits with 30 % fewer stalls.

Preloading the memory into the cache reduces the number of stalls but requires `prefetch` instructions to be executed. Figure 8c demonstrates the number of performed instructions per operation. Prefetching results in about 228 additional instructions per lookup and 300 per insert operation, compared to the non-prefetching run. The difference results from the dynamics of the tree during the fill phase. Due to a node-split, a task may not be able to traverse directly but has to "climb" along a tree-level first. Nevertheless, these additional instructions reduce the memory stalled cycles to such an extent that prefetching still pays off.

7.3 Comparison of tasks and threads

We argue that `MxTask`s offer a superior abstraction level to build scalable software for modern and future many-core hardware. To study this hypothesis, we will compare different programming models, libraries, and synchronization mechanisms using the B^{link}-tree for a real-world scenario. In addition to `MxTasking`, we also implemented the B^{link}-tree on top of *p_threads* and the tasking library Intel TBB, which pursues similar goals as `MxTasking`. Besides, we evaluate three different techniques for synchronizing parallel control flows: Sequencing of all accesses to a node, pessimistic latches for parallel reads and exclusive writes, and an optimistic approach. Figure 9 shows the results for all programming models and synchronization procedures. We will first examine the various programming models individually.

`MxTasking`. Figure 9a presents the results for filling and reading an `MxTask`-based B^{link}-tree. We evaluated three different synchronization methods, described in Section 4. The approach called *Scheduling* (left) maps each data resource, e.g., a tree node, to a specific task pool. The scheduler dispatches all tasks accessing a node to the assigned one. Hence, tasks reading or writing the same tree node are implicitly sequenced, because `MxTask`s will not be interrupted during execution. That enables a parallelization level similar to spinlocks, where each node is accessed exclusively by one thread. However, in contrast to spinlocks, scheduling eliminates active waiting by avoiding latches due to implicit synchronization of concurrent control-flows. Furthermore, this mechanism exploits the CPU cache more efficiently, since a data object is not replicated in different caches, but is only ever occupied by the very same core. The results show that this synchronization method scales up to 10 cores with 8.2 million operations per second (both reads and writes) and decreases slightly afterward. As soon as the second NUMA region comes into play, the throughput decreases stronger. That is due to the high effort caused by the task-scheduling, which is realized by an atomic `exchange` instruction to push tasks to task-pools. In particular, the core owning the task pool associated with the root node gets highly frequented by all other. That causes overhead due to cache-coherence since the root is the starting point for all operations.

A similar factor is observable when using reader/writer-locks, shown in Figure 9a (center). This way, tasks are primarily dispatched to the core they are produced by to minimizes overhead by scheduling. Moreover, reader/writer-locks enable parallel reads that increases the throughput. However, when using the second NUMA region, the throughput decreases again. In this case, the additional effort for keeping the latch variable coherent has a negative effect. The maximum is reached at 12 million lookups and 9.3 million inserts per second, utilizing 22 logical cores.

Other works have already pointed out the advantages of

optimistic synchronization [10, 28]. `MxTasking` implements this approach by allowing any data objects to be versioned. Scheduling reading tasks to the local pool avoids additional overhead. To prevent concurrent write operations to the same data object, these are still synchronized by scheduling. However, many tasks in the B^{link}-tree benchmark, even for insert operations, are readers until they reach a leaf node to insert a value or an inner node to place a pointer to another node. With this synchronization strategy, we gain more than 74 million lookups and 35 million inserts per second. The tasks benefit, especially, from the possibility of prefetching data objects so that they are already found in a local cache when accessed.

p_threads. We present the results for the B^{link}-tree-benchmark built on top of traditional threads in Figure 9b. Utilizing spinlocks for sequencing all accesses to one node ends up with a throughput of 3.6 million lookup operations per second at four cores. After that, the throughput decreases steadily using more than four cores because of cache-coherence overhead. Using reader/writer-locks, to enable parallel read operations to a node, attain 9.3 million insert operations and 9.8 million lookup operations per second with 24 cores and decreases afterward. We borrowed the reader/writer-lock implementation from *Facebook*'s *folly* library[3].

According to expectation, optimistic synchronization accomplishes to the highest (thread related) throughput, where reading operations read the version of a node before and after read while writing procedures take a latch to be mutually exclusive. Here, we measure 42.7 million inserts and 59.3 lookups per second to the B^{link}-tree using all available 48 logical cores. This result is comparable to the *Btree-OLC* [26], which ends up with 39.8 million inserts and 54.7 million lookups per second on our hardware.

Intel TBB. As a second task-based approach, we evaluated a B^{link}-tree based on Intel TBB using version 2020 Update 2. The tasks implementing lookup and insert operations are build similar to `MxTasking`, except synchronization, which is part of the application layer since TBB does not include automatic synchronization. Figure 9c presents the results. For concurrency control, we use on-board synchronization primitives: *speculative_spin_mutex* for full mutual exclusion on node-level and *speculative_spin_rw_mutex* for enabling parallel read operations. Both primitives provide *speculative latching* based on HTM, which is supported by our hardware. We found that the speculative mutexes delivered by TBB produced the best performance.

Using the *speculative_spin_mutex*, which is described as *scalable* by TBB documentation, the tree-benchmark gains the highest throughput with 5.4 million lookups per second at six cores. However, when applying more computing resources, especially those from the second NUMA region, the throughput decreases.

In contrast, the *speculative_spin_rw_mutex* enables parallel reads. With this, the benchmark scales up to 25.8 million insert and 35 million lookup operations per second using all available 48 logical cores.

Utilizing the optimistic approach as we did before, using threads as abstraction level, the throughput increases to 32.2 million inserts and 42.3 million lookups per second. Again, we were required to build an optimistic synchronization on top of TBB tasks within the application layer.

Comparison. Putting it all together, Figure 9 illustrates the differences between the programming models and synchronization strategies. When enabling only sequential access to each node, both reading and writing do not scale on any of the evaluated platforms. That is not surprising since many other works already investigated into latches and approaches to read and write data structures in parallel. Although `MxTask`s avoids latching by scheduling, we can observe two bottlenecks preventing this approach to scale. First of all, every operation starts by reading the root node of the tree. Even when all subsequent steps run in parallel, by distributing tasks to the other nodes and implicit more CPU cores, the inherently sequential access to the root bounds the throughput. This also applies to (speculative) spinlocks. Secondly, pushing tasks to "external" task pools of other cores involves overhead due to cache-coherence, even if the operation itself is atomic. Especially the task pool, on which all tasks concerning the root are dispatched, is affected since many producers try to (atomically) dispatch tasks in parallel. The effect is similar to latches, where many threads (or TBB tasks) want to modify the same cache line to acquire the latch of the root.

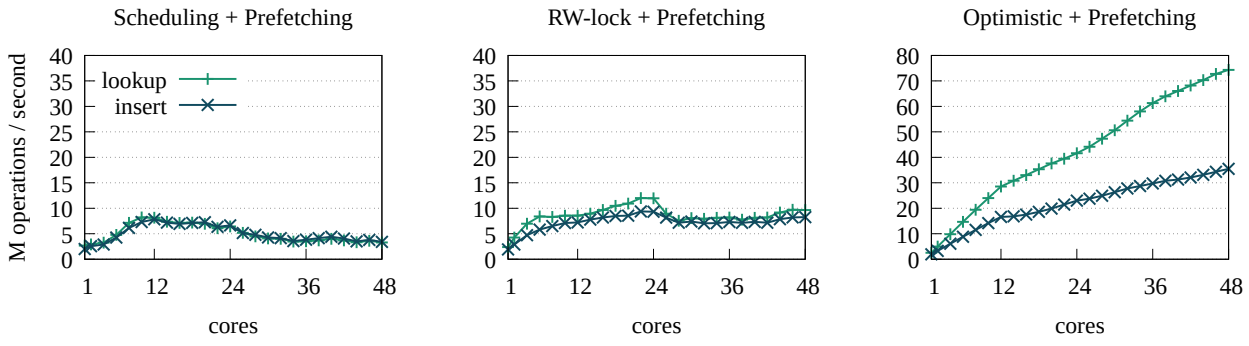
Using RW-locks as an alternative latching approach, for enabling parallel reads but separate writes to nodes, the operations are most notably bound by latching overhead, which is due to cache-coherence. On the `MxTasking` side, we can observe a benefit of 22% more lookups per second compared to threads. Nevertheless, both programming models do not scale properly. However, using speculative reader/writer-locks as done by TBB, we notice less overhead due to latching and thus better performance, up to 25.3 million inserts, and 35.4 million lookups per second. That is more than 2.9× compared to `MxTasking` and 3.6× to threads.

All three compared libraries and models perform best when using optimistic methods for synchronization. Nevertheless, some differences are observable in this context. While *p_thread*-based implementation provides the best results for insert operations with 42.7 million inserts per second, `MxTasking` outperforms threads and TBB providing 74 million lookups per second.

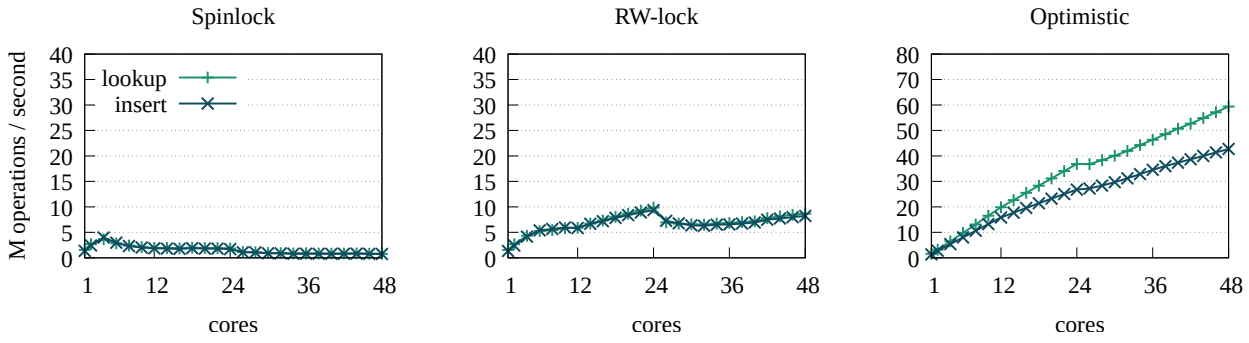
Figure 10 shows a cycle-accurate comparison between the approaches. We distinguish between effort for traversing the tree, inserting or searching for a value, cycles in kernel mode, e.g., syscalls, and additional effort for the libraries `MxTasking` and Intel TBB. For `MxTasking`, we further differentiate between complexity for prefetching and other `MxTasking` related work, most notably scheduling. We recorded those details using Intels *VTune™* Profiler [3].

The results prove the effectiveness of the prefetching mechanism used by `MxTasking`. Traversing the tree requires significantly fewer cycles when applying `MxTask`, compared to threads and TBB tasks. However, we even observe a noticeable amount of time spent on `prefetch` instructions, which, strictly speaking, should be added to the time of traversal. During the insert phase, we notice several cycles expended to `MxTasking`. That is primarily the effort for scheduling tasks that are moved from one core to another to sequence write operations. Also, the TBB scheduler involves some

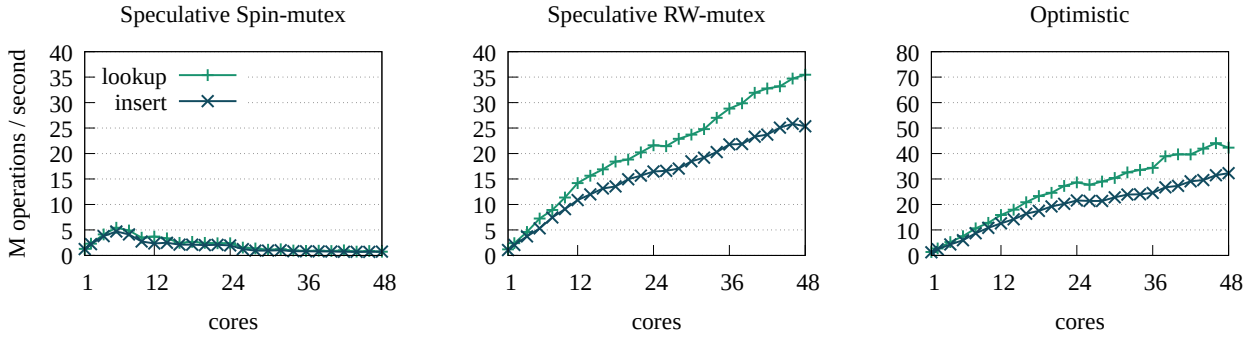
[3] `https://github.com/facebook/folly`



(a) `MxTasking`



(b) p_thread



(c) Intel TBB

Figure 9: Comparing the throughput of different programming models and libraries for a B^{link}-tree-benchmark.

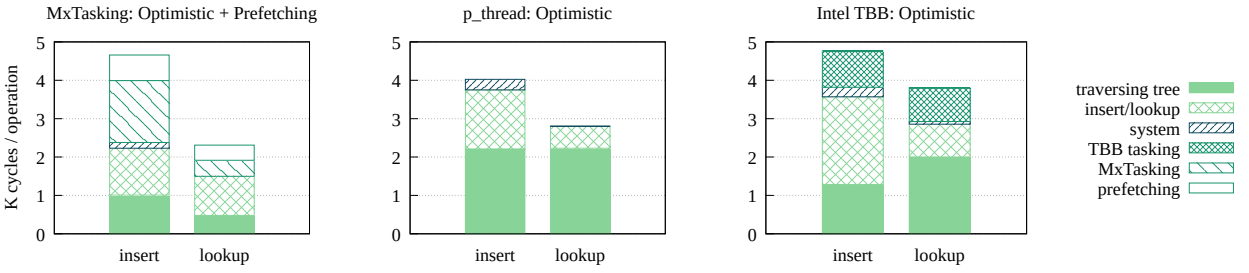


Figure 10: Detailed, cycle-based analysis of B^{link}-tree implementations using `MxTasking`, threads, and TBB for control-flow abstraction.

additional work, which we are not able to break down more precisely. We assume that this is an expense for load balancing and task stealing.

To summarize, we believe that the software-controlled prefetching of data objects, which is unique to the `MxTask` concept, offers considerable increases in throughput. Manually integrated prefetching into other approaches such as threads and TBB requires considerably more effort on the part of the application engineer and possibly restructuring of the data structures.

8. CONCLUSIONS

In this paper, we presented `MxTasking`, a task-based framework with run-to-completion semantic. The unique selling point of `MxTasking` is given by task-annotations, which offers the algorithm engineer to easily transfer knowledge from application level to the control-flow abstraction. Hence, the tasking runtime hides memory latencies by loading soon-to-be-accessed data into CPU caches, without the intervention of the engineer. Furthermore, appropriately using fine-grained tasks in combination with task- and data object annotations opens up the possibility to design parallel data structures and algorithms, without exposed synchronization. Requirements of data object synchronization, e.g., exclusive read/write, as well as the task's access intention, enable `MxTasking` to choose the most efficient synchronization technique. As a result, `MxTasking` eases the development of latch-free data structures.

Moreover, we presented our vision of `MxKernel`, a bare-metal runtime for DBMS/OS Co-Design. `MxKernel` provides a thin layer for both DBMS and OS running on top. This way, mutually needed data structures can be shared without any need for the DBMS to bypass the OS. Instead of traditional threads, `MxTask`s form the abstraction for work items.

The first results of an `MxTask`-based key-value store promise great potential on modern and future many-core hardware. Using a B^{link}-tree as the foundation, `MxTask`s outperform classical threads by 25% more lookups per second.

9. ACKNOWLEDGMENTS

This work was supported by DFG, Deutsche Forschungsgemeinschaft, grant number TE 1117/2-1.

10. REFERENCES

[1] The DB2 UDB memory model. `https://www.ibm.com/developerworks/data/library/techarticle/dm-0406qi/`, June 2004. Online available; accessed at 02/04/2020.

[2] Database Administrator's Guide. `https://docs.oracle.com/database/121/ADMIN/memory.htm#ADMIN00207`, July 2017. Online available; accessed at 02/04/2020.

[3] Intel® VTune™ Profiler. `https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html`, May 2020. Online available; accessed at 22/06/2020.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer, 2009.

[5] T. Bang, I. Oukid, N. May, I. Petrov, and C. Binnig. Robust Performance of Main Memory Data Structures by Configuration. In *Proceedings of the 2020 International Conference on Management of Data*, pages 1651–1666, 2020.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[7] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta informatica*, 9(1):1–21, 1977.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distributed Comput.*, 37(1):55–69, 1996.

[10] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, volume 1, pages 181–190, 2001.

[11] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17–es, 2007.

[12] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization*, pages 57–66. IEEE, 2008.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[14] K.-F. Faxén. Wool–a work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2008.

[15] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1603–1618. ACM, 2018.

[16] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. COD: Database/Operating System Co-Design. In *CIDR*, 2013.

[17] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proc. of the 36th annual international symposium on Computer Architecture*, pages 184–195. ACM, 2009.

[18] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distributed Comput.*, 67(12):1270–1285, 2007.

[19] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th*

International Conference on Extending Database Technology: Advances in Database Technology, pages 24–35. ACM, 2009.

[20] A. Kleen. A numa api for linux. *Novel Inc*, 2005.

[21] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *Proc. VLDB Endow.*, 9(4):252–263, 2015.

[22] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. of the 8th annual symposium on Computer Architecture*, pages 81–87. IEEE Computer Society Press, 1981.

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

[24] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[25] V. Leis, P. Boncz, A. Kemper, and T. 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*, pages 743–754. ACM, 2014.

[26] V. Leis, M. Haubenschild, and T. Neumann. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.*, 42.1:73–84, 2019.

[27] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*, pages 580–591. IEEE, 2014.

[28] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pages 1–8, 2016.

[29] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering*, pages 302–313. IEEE, 2013.

[30] D. Makreshanski, J. J. Levandoski, and R. 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.

[31] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, 2017.

[32] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[33] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with coroutines: a practical approach for robust index joins. *Proc. VLDB Endow.*, 11(CONF):230–242, 2017.

[34] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *The VLDB Journal*, 28(4):451–471, 2019.

[35] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[36] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[37] K. Sakamoto and T. Furumoto. Grand central dispatch. In *Pro Multithreading and Memory Management for iOS and OS X*, pages 139–145. Springer, 2012.

[38] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, volume 27, 2008.

[39] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66, 2010.

[40] P. Tözün and H. Kotthaus. Scheduling Data-Intensive Tasks on Heterogeneous Many Cores. *IEEE Data Eng. Bull.*, 42.1:61–72, 2019.

[41] J. Tse and A. J. Smith. CPU cache prefetching: Timing evaluation of hardware implementations. *IEEE Transactions on Computers*, 47(5):509–526, 1998.

[42] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488, 2018.

[43] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Making address-correlated prefetching practical. *IEEE micro*, 30(1):50–59, 2010.