MxTasks: How to Make Efficient Synchronization and Prefetching Easy

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

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

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 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.

KEYWORDS

Task-based parallelism; Synchronization; Prefetching

ACM Reference Format:
Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457268

1 INTRODUCTION

The basic architectures of 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 in a Non-Uniform Memory Access (NUMA) fashion. While the hardware keeps changing, the software has to adapt to benefit from the newly available resources.

Massive parallelism and heterogeneity offer immense opportunities to improve performance but also pose complex challenges. Synchronization of concurrency, utilization of available CPU resources, and integration of co-processors represent critical examples. Latches—as synchronization primitives—, for instance, reduce parallelism by serializing accesses and cause overhead by contention.

Parallelization and Synchronization. In this light, during the past years, researchers have invested great efforts to better leverage the available hardware parallelism, e.g., through very fine-grained latching mechanisms or by avoiding latches altogether [19, 28, 30, 31]. But despite the progress made, it remains difficult to design latch-free algorithms and data structures. Most of them are carefully tailored to very specific situations; it is not clear how they could be generalized to different problem settings. *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 was also shown how hard it is to outperform well-engineered "classical" code with HTM alternatives [29, 32].

Resource Management and Frameworks for Parallel Computing. To utilize the entire computational parallelism, operations have to be allocated carefully to available (CPU) resources. This requires a solid understanding of the particular physical system and application behavior. Dividing the work into small, closed units, called tasks, assists the developer in designing parallel software without having to worry about the underlying many-core hardware. Frameworks such as Intel® Threading Building Blocks (TBB) [40] and native support within Operating Systems (OSs) like fibers in Windows [2] as well as Apple Grand Central Dispatch in macOS [41] make use of this concept. 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 this way [11]. Not least because those frameworks have just sparse knowledge regarding the application and its intention.

In this paper, we present MxTasking[1], a task-based environment for today's and future many-core hardware. The principal abstraction in MxTasking is the MxTask. An MxTask is a short program sequence that performs a single, small unit of work, with the guarantee 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 the execution unit, for instance, *runtime characteristics* (such as expected resource needs); information about related *data objects* (including access intention such as read or write access); or desired *scheduling priorities*. MxTasking will then use such knowledge to optimize resource allocation, scheduling, and placement.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457268

[1] The source code of MxTasking is available at https://github.com/jmuehlig/mxtasking and http://dbis.cs.tu-dortmund.de

We will specifically report on two ways of using annotations that ease the development of parallel applications and improve performance at the same time. One is to enable *automatic data prefetching*. Existing work has shown that software-based prefetching can be very effective at reducing data access latencies [10, 22, 35, 39]. But the mechanism is also known to be cumbersome to use, its effectiveness hinges on the developer's hardware understanding baked into manually tuned code. With task annotations, `MxTasking` can instead infer relevant metrics and inject prefetch instructions automatically at runtime. In practice, this can reduce memory stalls by up to 50 % with negligible assistance from the application developer.

Synchronization annotations are the second powerful way of using annotations that we will discuss in this work. Rather than manually implementing and tuning intricate and error-prone synchronization mechanisms (spinlocks, reader/writer locks, version locks, ...), with `MxTasking`, developers may simply express their desired type of isolation as a task annotation. Our library will take care of the rest and inject the synchronization primitive that is appropriate for the current system and application state. Again, this not only eases the development of massively parallel applications but may also improve performance by several factors in contended scenarios.

The rest of this paper is organized as follows: Section 2 introduces task-based parallelism in general. Afterward, Sections 3 and 4 present details of the `MxTasking` runtime and annotation principles for memory-prefetching and synchronization. In Section 5, we provide practical insights into our tasking library. The first results of a key-value store, built with `MxTasks`, are demonstrated and discussed in Section 6. The paper concludes in Section 7.

2 TASK-BASED PARALLELISM

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.

2.1 Background and Related Work

The idea 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*. In general, lightweight threads (we refer to them as tasks; others may name them *fibers*) are scheduled and executed at the user-level. Some OSs provide native support for such lightweight threads (e.g., cooperative scheduled fibers in Windows [2] and tasks in macOS [41]).

With *Cilk*, Blumofe et al. published one of the first runtime systems for parallel programming that schedules tasks onto OS threads [8]. Aiming to simplify the engineers' work, Cilk focuses on the automatic load balancing of parallel applications and easy integration into existing software programs. To assist the synchronization of concurrent tasks, Cilk supports *lock*/*unlock* calls on a latch variable.

Intel® provides the TBB framework focussing on portability and robust performance [24, 40]. The latter aspect is accomplished primarily by using a work-stealing mechanism that balances the load over the worker threads. TBB provides several synchronization primitives such as 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 provides a graph-based programming interface. The *Wool* framework purses similar objectives through a comparable work-stealing strategy [13].

StarPU intends to provide fine-granular tasks for heterogeneous multicore platforms [3]. The authors argue that the modern hardware landscape features not only CPU-based parallelism but also uses special co-processors. StarPU offers a framework that supports both CPUs and co-processors such as GPUs. Like TBB, StarPU leaves the synchronization to the user.

Tasks—or similar concepts—have also been exploited in the context of DBMSs. Gasiunas et al. use fibers for realizing a DBMS underlying virtual network functions in a shared-nothing environment [15]. Tözün and Kotthaus provide a concept for scheduling database tasks to heterogeneous hardware and discuss the challenges of granularity and scheduling [43]. *TAMEX* translates logical query plans into task-graphs to benefit from the load-balancing of fine-grained work packages in parallel settings [48].

Specifically, in the HyPer engine, *morsels* resemble tasks that execute segments of a query [27]. The scheduler of the query execution framework takes care of NUMA-local execution. Morsels can be load-balanced at runtime. For example, when the workload changes.

DORA avoids disorganized data access across parallel transactions by transferring execution units to the data instead of vice versa [37]. To that end, DORA divides transactions into multiple (task-like) *actions* while the present data is partitioned logically to threads. For execution, DORA distributes actions among threads.

Bang et al. utilized tasks for various index structures like B-trees and hash-tables, as well as transactional workloads [4]. 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.

Based on *coroutines*, Psaropoulos et al. invented fine-granular tasks for index joins to hide memory latencies [38]. Every time a coroutine attempts to access not cached memory, it executes a `prefetch` instruction and yields the coroutine. This way, the CPU executes other coroutines while the memory subsystem loads the requested data into the cache instead of wasting cycles to wait for the load fulfilled. Likewise, Jonathan et al. exploited coroutines to hide memory latencies in terms of state-of-the-art index structures [20]. He et al. also demonstrated the performance-related benefits of software-based prefetching by interleaving coroutines to process transactions [18].

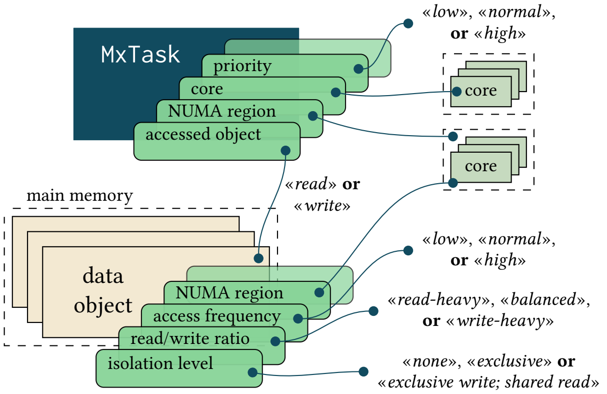


Figure 1: MxTasks provide annotations for accessed data objects, core, NUMA region, and priority. Data objects maintain metadata for isolation level and expected access frequency, as well as a read-write ratio.

2.2 MxTask Abstraction

A key-proposition of MxTasking is to replace the application-facing control flow abstraction by what we call MxTasks. 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 to optimize resource utilization. By using 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*. Annotating data objects with application-based knowledge, as well, offers the runtime a detailed understanding of the interaction of code and data. A complete, higher-level algorithm will be composed of a (possibly large) number of MxTasks 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 tree traversal. Each task will access a single tree node and spawn a new task (to process the next node) just before it finishes.

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).

Annotations. The engine provides mechanisms to attach annotations to tasks and data objects. Annotations can be used to transfer knowledge from the application to the execution layer, as demonstrated in Figure 1. Specifically, annotations enable the developer to specify the execution target of an MxTask, for example, a specific CPU core or a NUMA region to utilize locality.

In this paper, we will focus on an annotation that can be used to link a task to the data objects that it intends to access (including information whether that access will be read-only or writing). MxTasking will exploit such information to *(a)* minimize *memory access cost* and *(b)* perform scheduler-assisted *task synchronization.*

```
  /* create an annotated resource, that is read by tasks in parallel,
     read heavy, and accessed with high frequency                     */
1 tree->root =
   tasking::create_resource<TreeNode>(isolation::shared,
   rw_ratio::read_heavy, access_frequency::high)
  /* spawn a lookup task that starts traversal at the root node      */
2 lookup = tasking::create_task<LookupTask>(tree->root,key)
3 lookup->annotate(priority::high)                         // priority
4 lookup->annotate(access::readonly)                    // access mode
5 lookup->annotate(tree->root, tree->node_size()) // accessed data
   object and size
6 tasking::spawn(lookup)
```

Figure 2: Example-based usage of the MxTasking API to create a tree node and spawning a lookup-task accessing that node.

With respect to the former use of annotations, we will investigate *annotation-based memory prefetching* in Section 3. The latter use is on our agenda for Section 4, where we show how task-based programming can improve performance and ease developer efforts at the same time.

The use of annotations by the developer is optional (though more and better annotations may help MxTasking to improve performance). We envision that annotations may also be attached automatically, e.g., by a JIT-style code generator that knows about the characteristics of generated tasks or by an optimizing compiler.

2.3 MxTasking System Interface

MxTasking *Programming Interface.* Internally, task annotations are stored as part of the task object in a structured way. As such, annotated metadata is accessible for both the developer and the runtime. In Figure 2, we illustrate the MxTasking programmer interface. In the example, a data object is created where we allow shared access and assume a read-heavy workload and a high access frequency (line 1). In lines 2–5, a high-priority lookup task is created, starting in read-only mode at the root of a search tree. That task is finally spawned in line 6.

MxTasking *Library / Worker Threads.* MxTasking is a layer between task-based applications and the operating system. From the application's perspective, spawning a task adds it to the task pool of a *logical core.* This is a latch-free operation (realized using a single atomic xchg), 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

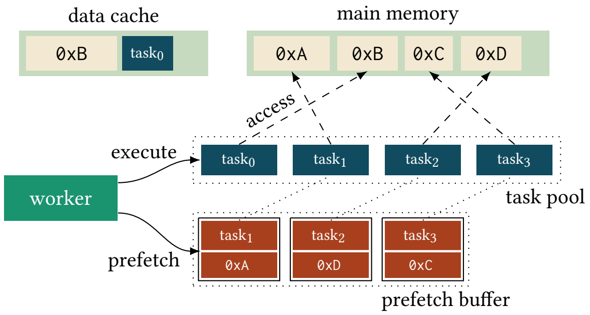


Figure 3: The worker knows tasks that will be executed soon and prefetches both tasks and data objects early enough to hide memory latencies.

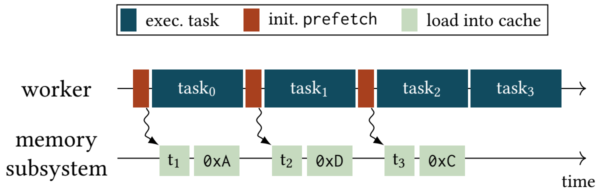


Figure 4: Timeline of prefetching and executing `MxTasks`.

further tasks. Such spawning will happen asynchronously and is lightweight.

3 ANNOTATION-BASED MEMORY PREFETCHING

Once annotations have been attached, `MxTasking` will use the annotated information to improve runtime performance and hardware efficiency. We will now specifically look into how `MxTasking` minimizes memory access cost based on annotations (no further action from the application side is required for this beyond creating the annotations).

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 bring the data into the cache; if the prefetch distance is too wide, data might already get evicted from the cache again before it is used.

Prefetch requests may be issued either by hard- or software. Efforts have been made to teach caching hardware the access patterns of database code [17, 23, 44, 47]. Yet, hardware prefetching remains unfeasible beyond stream-based look-aheads. Software-based prefetching was shown to be more effective (e.g., [10, 22, 35, 39]) but depends on substantial algorithm restructuring to work out.

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

Thereby, we hit 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 `MxTasks`. At the same time, 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. 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.

Prefetching Implementation. `MxTasking` keeps spawned tasks in a task pool. Thus, the runtime can already "see" tasks and their associated memory objects (together with the annotated size) ahead of time. Based on this information, the worker thread will inject prefetch instructions in-between task executions. `MxTasks` will thus see their data already cached in the CPU when they start.

Zooming into implementation details, a `tasking::spawn()` call will not only place the `MxTask` in the task pool but also add applicable prefetch information to a separate *prefetch buffer*.

Figure 3 illustrates this mechanism. Prefetch requests are placed in the prefetch buffer at a position such that prefetches belonging to a task are initiated sufficiently early. In the figure, we assume that prefetch requests are issued with a prefetch distance of 1, i.e., $task_1$ and its associated data are prefetched just before $task_0$ is executed, giving the hardware sufficient time to prefetch.

A corresponding timeline is illustrated in Figure 4, showing also how prefetch requests are processed asynchronously by the memory subsystem.

Prefetch Distance. The *prefetch distance* is a configuration parameter in our current prototype, which can be tuned to the characteristics of a particular piece of hardware. A natural extension of this mechanism would be to dynamically adjust the prefetch distance at runtime, e.g., based on performance monitoring [1, 25, 34]. In contrast to these existing proposals, however, `MxTasking` is not bound to profiling results from the past: annotations allow our scheduler to also glimpse into the future. This might, in fact, open up new opportunities to derive even better prefetch decisions at runtime.

4 ANNOTATION-BASED SYNCHRONIZATION

We will now turn our attention to a way of exploiting annotations that goes beyond "only" improving performance. Annotations can, in fact, be used to realize *synchronization* across parallel units in a simple, yet powerful and scalable way.

The classical mechanism to ensure correctness in parallel environments is to protect sensitive data structures using *latches*. The mechanism often quickly hits scalability limits unless enormous engineering efforts are put into tweaking the code toward very fine-grained latches. Optimistic synchronization techniques, such as *versioning*, may provide better scalability in the case of highly contended data objects. But they are error-prone to develop, and their benefits heavily depend on access characteristics and hardware parameters. More recently, *transactional memory* is touted as an alternative; but existing work shows that it is still hard to match the performance of well-engineered latch- or version-based code [29, 32]. To summarize, all these mechanisms require the developer to deeply intertwine application code with hardware characteristics, and often also with knowledge about workloads and/or data access patterns.

Instead, with `MxTasking`, we strictly separate concerns. From the developer, we only expect that she uses annotations to express *data dependencies* and/or *isolation properties* for data objects. Optionally, she may add hints about *access frequencies* or *read/write ratios*.

Based on the annotated information, `MxTasking` will infer and apply appropriate synchronization mechanisms—automatically, at runtime, and based on a view of the entire system state. With concerns on semantics and implementation separated in this way, the developer can focus her attention on the actual application logic, knowing that `MxTasking` will take care of the rest.

Before we delve into automatic mechanism selection and the inner workings of the `MxTasking` scheduler, we will first re-visit the synchronization mechanisms (or *primitives*) that `MxTasking` currently supports.

4.1 Integrated Synchronization Primitives

Our current prototype of `MxTasking` supports three basic synchronization primitives, which we will sketch 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 that `MxTasking` will choose (unless the task requests a particular primitive explicitly through annotations). `MxTasking` will select among its built-in primitives at runtime, depending on the current system state as well as on task annotations.

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. For mutual exclusion, a simple spinlock can serialize all accesses, whether tasks are read-only or not. Given an application that desires 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.

Optimistic Versioning. Latch-based protocols turn parallelism into concurrency. Especially for read-dominated workloads, this may limit the achievable parallelism and throughput unnecessarily.

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., using latches. Conflicts between read- and write accesses are allowed to occur; but they are *detected* with help of a *version counter*. More specifically, write accesses will increment the version counter after each modification; reading operations will check the version counter before and after they access the data object, and this way detect writes that happened in parallel. If a conflict is detected, the read operation is repeated.

Optimistic mechanisms were shown to achieve better throughput on hardware with high degrees of parallelism (e.g., [9, 28, 30]). In-memory index structures are a good example to illustrate the advantage of optimistic protocols. While all tree-operations will have to traverse the root node (thereby possibly leading to high contention), actual modifications of the root node are rare. Optimistic synchronization mechanisms can avoid high latching overhead; the penalty of a repeated read operation arises only rarely.

The positive effects of optimistic versioning are exacerbated on platforms (such as `MxTasking`) that are aware 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. Optimistic versioning, by contrast, enables the hardware to replicate and cache data structures for read-only accesses, thereby permitting true parallelism. Hardware cache coherence protocols will make sure that messages are sent between cores exactly and only when a true conflict arises.

Note that *writers* 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.

In `MxTasking`, optimistic versioning too is performed by the worker thread on behalf of the task that requested to be synchronized. If the worker thread detects a version mismatch, the task is reset and re-executed until the execution was valid.

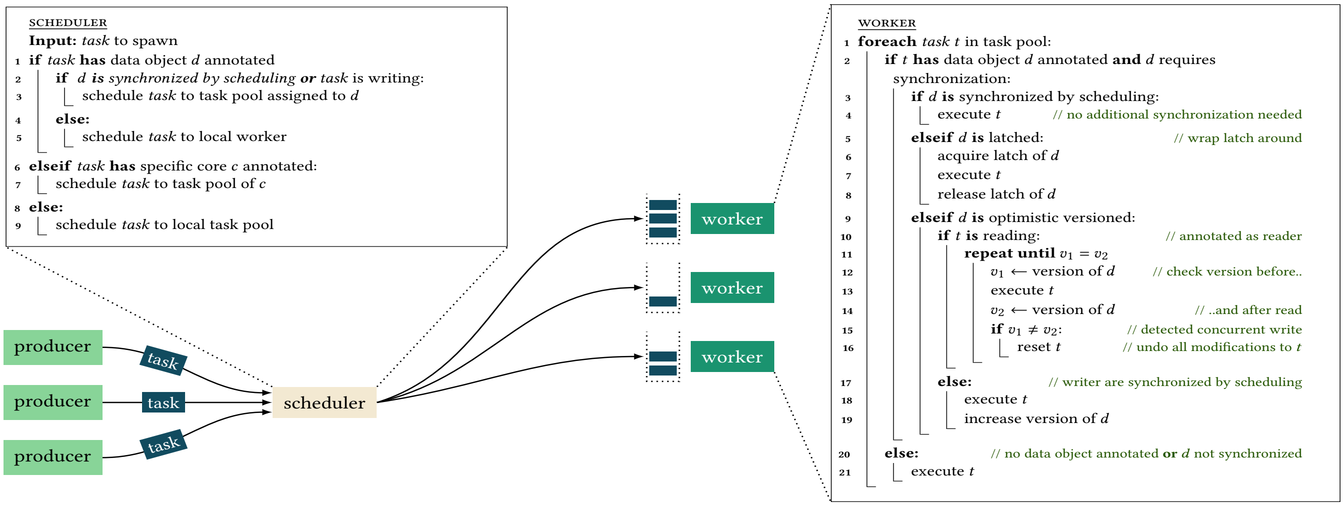
Synchronization through Scheduling. In addition to "classical" synchronization mechanisms such as latches or versioning, the task-based execution model of `MxTasking` enables another powerful synchronization mode: *scheduling-based synchronization.*

As the most simple form of scheduling-based synchronization, `MxTasking` can guarantee that tasks which access 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 that is associated with the pool (cf. Figure 5). Blocking or contention can fully be avoided this way.

Load balancing is achieved by instantiating several task pools per worker thread. If necessary, worker threads may also steal task pools (not tasks!) from each other to distribute the load.

Besides the avoidance of concurrency, scheduling-based synchronization can have an additional benefit, especially in NUMA environments. Rather than moving data objects between NUMA nodes, scheduling-based synchronization effectively moves *code to data.* Similar principles were shown to improve cache locality and transaction throughput, e.g., in systems like DORA [37] or H-Store [21].

A more generalized form of scheduling-based synchronization can be realized by annotating *dependencies* between tasks (in the spirit of [7]). To illustrate, in a task-based hash join implementation, the first probe task will not start before all build tasks have finished populating the in-memory hash table.



```
SCHEDULER
Input: task to spawn
1 if task has data object d annotated
2     if d is synchronized by scheduling or task is writing:
3         schedule task to task pool assigned to d
4     else:
5         schedule task to local worker
6 elseif task has specific core c annotated:
7     schedule task to task pool of c
8 else:
9     schedule task to local task pool
```

```
WORKER
1  foreach task t in task pool:
2      if t has data object d annotated and d requires
         synchronization:
3          if d is synchronized by scheduling:
4              execute t          // no additional synchronization needed
5          elseif d is latched:                          // wrap latch around
6              acquire latch of d
7              execute t
8              release latch of d
9          elseif d is optimistic versioned:
10             if t is reading:                         // annotated as reader
11                 repeat until $v_1 = v_2$
12                     $v_1 \leftarrow$ version of d     // check version before..
13                     execute t
14                     $v_2 \leftarrow$ version of d             // ..and after read
15                     if $v_1 \neq v_2$:               // detected concurrent write
16                         reset t                 // undo all modifications to t
17             else:              // writer are synchronized by scheduling
18                 execute t
19                 increase version of d
20     else:            // no data object annotated or d not synchronized
21         execute t
```

Figure 5: Interaction of *scheduler* and *worker thread*. The scheduler ensures the serialization of writing tasks by dispatching them to the identical task pool. The worker, however, injects synchronization of tasks in the execution of those.

4.2 Selecting Synchronization Primitives

For injecting synchronization, the runtime applies one of the described primitives to every newly spawned data object. This primitive can either specified by the developer using explicit annotations or selected by `MxTasking`. For choosing the synchronization mechanism automatically, the runtime uses a cost model considering given annotations that describe the access properties. Especially, the isolation level, the expected access frequency, and assumed read-write ratio are taken into account. For example, requesting *exclusive* access to the object will result in serializing by scheduling as it leverages better performance than spinlocks in our benchmarks. Using more relaxed isolation—which enables parallel reads—matches an optimistic synchronization strategy. Depending on the expected read-write ratio, `MxTasking` will use scheduling for writing tasks in case reading operations predominate. By specifying one task pool for write operations to an object, read-only tasks running at the same worker are guaranteed to execute successfully; version checks and saving the state of the task is dispensable. For written-heavy resources that are accessed moderate or sparse, scheduling overhead—mainly contention on a task pool—prevails latch-contention regarding the data object. Hence, `MxTasking` prefers optimistic latches for such resources.

As an illustration, the nodes within a task-based tree structure are synchronized using varying (optimistic) primitives: While the higher-level nodes are usually read, leaf-nodes are accessed less but may be modified more often. Thus, `MxTasking` determines optimistic scheduling for inner nodes and optimistic latches for leaf nodes—assuming appropriate annotations by the developer.

4.3 Scheduler/Worker/Task Interaction

Figure 5 illustrates how the synchronization of tasks bases its interaction with the scheduler and worker thread. The scheduler ensures to place `MxTasks` in the pool of the appropriate worker thread, depending on the synchronization mechanism and access type. The worker, in turn, applies synchronization primitives whenever needed.

The Scheduler Side. To serialize a set of tasks, the scheduler places them in the same task pool. This becomes necessary when (a) optimistic scheduling is applied, and the task modifies the annotated object *or* (b) all accesses to a data object are synchronized through scheduling. 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 overhead in the form of cache-coherence. Local, in this context, indicates the task pool of the worker thread producing the task (potentially while executing another task).

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

The Worker Side. Applying the synchronization primitive, whenever necessary, is performed by the worker thread, wrapped around the execution of a task. First, the worker evaluates the annotated data object (line 2). Supposing no synchronization is needed or scheduling already guarantees serial access, 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, the worker acquires the latch related to the data object before executing the task and releases it after execution (lines 5–8). Whenever possible, we acquire the latch in *shared mode* using reader/writer latches.

For data objects synchronized by optimistic versioning, the worker thread separates between reading and writing tasks. To verify a data object was not modified while performing a read-only operation, the worker checks the version before and after the execution (lines 10–16). Whenever the counter mismatches, the read access has to be retried. To ensure that the version changes at all, the worker increments it after executing a writing task (line 19).

4.4 Implementation Aspects

Comparable to other optimistic procedures, operations (or tasks in our context) that physically remove a shared object have to be treated with special care. Reference counting, hazard-pointers [36], and Epoch Based Memory Reclamation (EBMR) [14] are general approaches to protect read accesses while another thread attempts to free the memory. For performance reasons, MxTasking implements an EBMR, similar to Silo [45] and the decentral procedure realized by the open BwTree [46]. The memory of logically deleted objects is not released for reuse instantly but delayed until all potential read operations finish.

In general, time is separated into coarse-grained epochs using a *global* epoch counter that increases periodically (e.g., every 50 ms). To detect possible concurrent read- and delete operations, all utilized threads provide a local epoch wheres logically removed data objects are marked with the current global epoch when removing. The thread-local epochs, in turn, represent the relative progresses to the global epoch. On entering a critical section, the thread synchronizes the local- with the global epoch. When 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 beginning of a new epoch, a separated garbage collection routine determines the minimal progress made by emphasizing the lowest thread-local epoch. Data objects deleted logically within an even earlier epoch get safely reclaimed.

In widely used implementations, critical sections are defined 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 using tasks, however, logical operations are divided into multiple units of work, executed at different worker threads. Hence, it is ambiguous to determine the beginning and end of a logical operation processed by several tasks. A similar situation faces when using coroutines (which are asynchronous, too) [18]. Wrapping local epoch updates around every single MxTask-execution causes many 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 limit becomes a trade-off between maximum performance and delay of releasing unused 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.

5 MXTASKING IN ACTION

Utilizing tasks to design data structures and algorithms differs in general from well-understood thread-based programming. Morsel-driven parallelism [27] and DORA [37] have already demonstrated the advantages for analytical- and transactional processing in a task-like fashion. MxTasking advances the task-paradigm beyond the current standard by offering annotations for prefetching and implicit synchronization. This section reviews some practical aspects of using MxTasks for building parallel software.

We illustrate the simplicity of designing a latch-free, task-based data structure, using a B^{link}-tree as an example. Accordingly, we will examine the relevance of memory management regarding task-allocation and demonstrate the robustness of MxTasks concerning the granularity.

```
INSERT TASK
task input: node the task accesses, key and value to insert, callback
  to notify on finish
if node->high_key <= key:          // key is out of range of this node
    next = node->right_sibling
    task = tasking::create_task<InsertTask>(next,key,value)
    task->annotate(next,tree->node_size())
    tasking::spawn(task)
elseif node->type == inner:          // continue traversal to the leaf
    next = node->child(key)
    task = tasking::create_task<InsertTask>(next,key,value)
    task->annotate(next,tree->node_size())
    task->annotate(access::readonly)
    tasking::spawn(task)
elseif node->type == branch: // child is a leaf, next task will write
    next = node->child(key)
    task = tasking::create_task<InsertTask>(next,key,value)
    task->annotate(next,tree->node_size())
    task->annotate(access::write)    // change to writing mode
    tasking::spawn(task)
else:                               // found correct leaf, insert value
    node->insert(key,value)
    callback->insertion_finished(key,value)
```

Figure 6: Insert operation of an MxTask-based B^{link}-tree.

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

Since latching has become a bottleneck for in-memory data structures on modern hardware, past research investigated optimistic or fully latch-free procedures (e.g., [9, 28, 30, 31, 33, 46]). The B^{link}-tree [26], as a variant of the B-tree, focuses on reducing the number of simultaneously held latches at a time. To this end, newly inserted nodes are not connected to the parent instantly, eliminating the need for holding the parent's latch. Instead, a node split will create a link between the old and the new node. With that help, the recently inserted node will also be accessible for parallel traverse operations, even when there is no link from the parent to the new node. Consequently, every logical procedure becomes a concatenation of multiple tiny steps related to a single node.

Whereas thread-based implementations result in synchronous calls most of the time, MxTasks (and comparable task-based solutions, e.g., [3, 20, 27, 37, 40]) get executed asynchronously. For instance, instead of calling an *insert* method on a tree that returns after finishing, spawning an *insert task* that notifies the caller after inserting is the way to go.

Insert task. 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-value-pair as input parameters. The pseudocode in Figure 6 illustrates an example of implementing an insert task. On every step, it examines whether the accessed node is an inner- or a leaf node. If the node is of type inner, the task determines 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 task was spawned.

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 follows the right sibling pointer 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 spawns a new `MxTask`, annotated with the next node (e.g., lines 8–11). Labeling the new task as a reader (line 10) enables `MxTasking` to execute 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 inserts the item and notifies the caller (lines 19–20). To respond to a client's request in an end-to-end setting, we use a callback function. Another option would be to spawn a new follow-up task that handles the response.

Synchronization through Annotations. Noticeably, the demonstrated code comes without synchronization of competing tasks—rather, `MxTasking` injects synchronization at runtime. Instead of explicit synchronization, we annotate the insert task as `access::readonly` during the traversal and as `access::write` whenever it might attempt to modify a node. Annotating as a writer at the appropriate time, however, becomes a minor challenge: when annotating too soon, parallelism may decrease because of serializing accesses. Too late, contrarily, requires re-annotating and re-scheduling the task, which causes overhead. To inhibit, we need to know during the traversal whether the next is an inner or leaf node as modifications are related to leaf nodes. Since loading the targeted node's metadata causes additional cache-misses, we introduce a new node type: *branch* nodes represent inner nodes whose children are leaf nodes. Accordingly, we annotate the insert task preventively as writing when we reach a branch node during traversal (lines 13–17).

B^{link}-tree Node Splits. For simplicity, we ignored the case of node splitting in this example. Splitting a node is required when no capacity left for an additional record. Given that case, the insert task spawns a separated task that links the newly created node and the parent. Until the pointer is seated to the parent node, the new node can be reached by following the sibling pointer.

Beyond Insertion. Implementing the corresponding *update-* or *lookup-tasks* is straightforward. Instead of inserting the value into the leaf (line 19), the appropriate `MxTask` modifies or reads the requested record. Consequently, found values are passed to the callback. In particular, for lookups, we omit the writing-annotation since all steps during the lookup perform read-only accesses.

5.2 Task Allocation Cost

The B^{link}-tree-example indicates that `MxTask`s are created and deleted with a high frequency. 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.

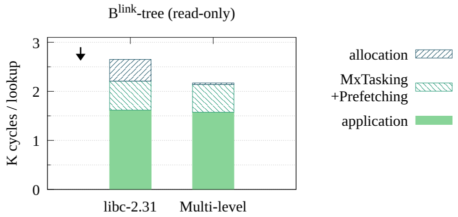


Figure 7: CPU cycles for a single lookup on a task-based tree, using different allocators for task allocation.

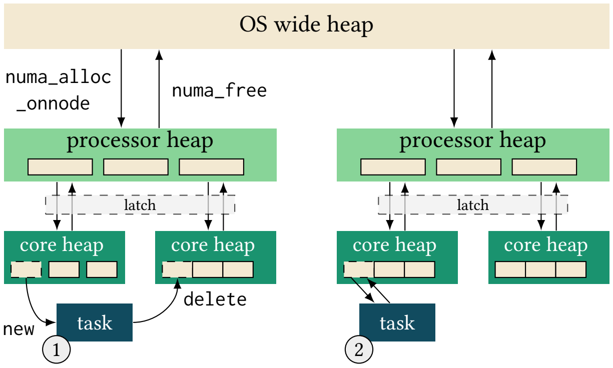


Figure 8: Reducing synchronization and enhance cache-awareness by using multiple levels for task-allocation.

Using the global heap may turn into a bottleneck when many cores request memory for new tasks concurrently.

Figure 7 shows the CPU cycles spent during a single lookup[2] on a task-based B^{link}-tree, including the traversal from the root to the leaf node. Allocating tasks using the system's `malloc` interface consumes 450 cycles per operation on a 48 core machine (∼ 16 % of total CPU cycles)[3].

To overcome this costly aspect, we designed a multi-level allocator, matching seamlessly into the tasking-runtime. The architecture is mainly inspired by *Hoard* [6]. Hoard focuses 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 worker thread becomes the point of contact for task allocation. Figure 8 outlines this approach. Whenever a task spawns a new one, it asks the local heap for memory. Allocating from this heap is very lightweight since the runtime guarantees `MxTask`s to run-to-completion, making synchronization

[2] We decided to use a read-only benchmark because nothing other than tasks are allocated.

[3] *Perf* was used to analyze and track consumed CPU cycles.

redundant. Free memory blocks 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, increasing the chance that the newly allocated task still rests in the CPU cache.

Reducing inter-processor communication and providing NUMA-aware allocation is a trade-off. Figure 8 shows a task (①) that is allocated on one core but deleted on a different one. The free block is pushed to the heap of the deleting core. In the worst case, where a task is allocated and deleted among workers located in different NUMA regions, memory blocks are shuffled across. However, we minimize synchronization and implicit communication costs between them.

When a core-heap runs out of memory, it will request a new memory block from the processor-heap. The processor-heap, in turn, 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 MxTasks requires only a single latch when allocating memory from the processor-heap, reinforcing scalability. However, by reusing deleted tasks, this issue occurs rarely. Compared to using malloc, Figure 7 demonstrates that our multi-level allocator has almost no overhead. Only 30 cycles are spent for task-allocation during a single tree-lookup. Likewise, we observe ∼ 7 % fewer cycles spent for prefetching as some tasks remain in the cache when reusing the memory.

5.3 Granularity of Tasks

While designing task-based applications or data structures, the granularity of a task may be an adjustable parameter. For some workloads, the task-granularity becomes implicit, given by the access-characteristics of the application. MxTasks accessing the B$^{\text{link}}$-tree described before, for instance, operate on a single node per MxTask. For various applications, the granularity is arbitrary.

To give an example, think of accessing and processing tuples of an in-memory DBMS for query execution. Spawning MxTasks causes additional overhead that could become a bottleneck when tasks are too short-lived. In particular, the exact costs for dispatching an MxTask depend on the targeted task pool: Transferring a task to any core located in a remote NUMA region is at more expense than attributing a consumer placed on the same socket.

For demonstration, we implement a parallel hash join on top of MxTasking, following the idea of morsels [27]. As shown in Figure 9, MxTasks are robust against performance penalties affected by granularities. Here, we join *consumer* and *order* tables from the TPC-H benchmark (with scale factor 100) using core-local hashtables. The join operator partitions the input and dispatches build- and probe-tasks, equipped with a certain number of records. Partitions are processed locally—exploiting NUMA-localities and without synchronization by taking advantage of the run-to-completion semantic.

The results verify a broad range of suitable task-granularities: Processing 2^7 up to 2^{16} records per task behaves approximately equivalent. Using only 16 tuples or less at a working unit causes scheduling-overhead to dominate the workload. Vice versa, too heavyweight (and consequently few) tasks cause imbalanced distribution.

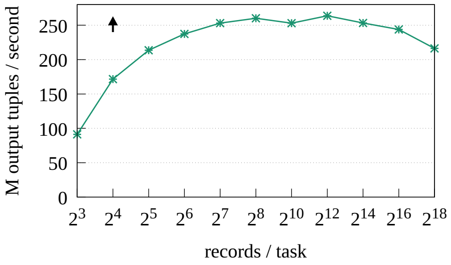


Figure 9: Hash join execution exploiting various task-granularities.

6 EXPERIMENTAL EVALUATION

To study the behavior and potential of MxTasking in real-world scenarios, we use an in-memory B$^{\text{link}}$-tree that is indicative of the behavior of modern in-memory database engines. Our implementation of the data structure follows state-of-the-art principles.

6.1 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 12 × 32 kB L1, 12 × 1 MB L2, and 1 × 19.25 MB L3 data caches. The logical 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. From then, hyperthreading cores are added step by step.

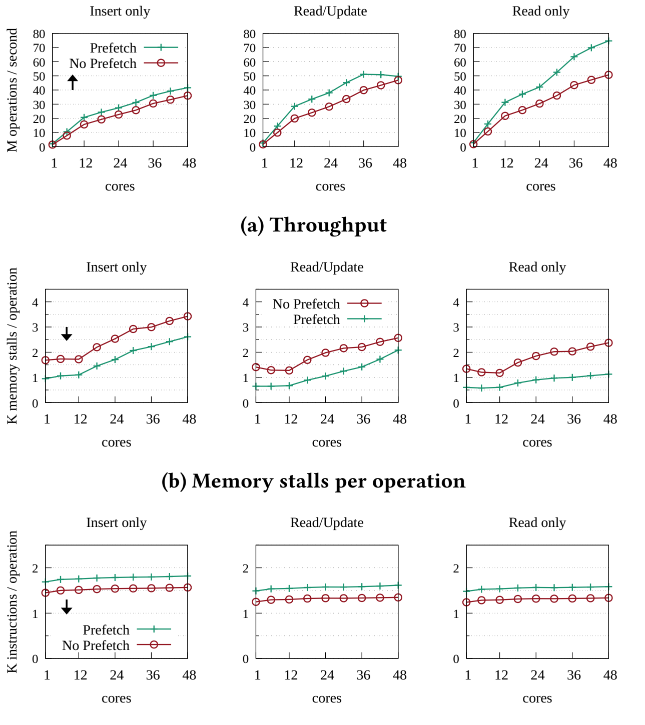
Following former work [46], we rely on the Yahoo! Cloud Serving Benchmark (YCSB) [12]. We use workloads **A** (*read*/*update*, 50/50) and **C** (*read-only*), both with Zipfian distribution and 100 million operations. Before running each workload, the tree is initialized with 100 million records. *Insert* results correlate to the initialization phase of workload **A**. The tree stores pairs of 64 b keys and 64 b payloads within 1 kB sized nodes.

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 p_thread 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 threads) when almost no other task is ready for execution.

Ubuntu 20.04 is used as OS, *clang* 10.0.0 as the compiler, configured to apply optimization level *-O*3. 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.

6.2 Annotation-based Prefetching

As discussed in Section 3, the fine granularity of tasks allows an exact prediction of which data is accessed by an MxTask. Figure 10 compares the B$^{\text{link}}$-tree build on MxTasking with and without annotation-based prefetching. Experiments regarding the prefetch distance indicated that the results behave as expected: If the interval



(a) Throughput

(b) Memory stalls per operation

(c) Executed instructions per operation

Figure 10: Impact of software-based prefetching for the Mx-Task-based B^{link}-tree.

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 four tasks apart), the advantage becomes smaller but still noticeable. For the measurements shown, we specified a distance of 2, which performed best on our experimental analysis of the prefetch distance.

Since the benchmark is *memory bound* for the most times, annotation-based prefetching of tree-nodes results in 21 % higher throughput rates on average for inserts and reads/updates, as well as 45 % for the read-only workload. We demonstrate the outcome in Figure 10a. Especially the tree-traversal that bases on binary search benefits from this mechanism. Because binary search creates a hard-to-predict access pattern for the CPU, hardware prefetching has a lesser effect.

The impact of software-based prefetching becomes visible when observing memory stalls, shown in Figure 10b. 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 workloads. Here, the number of memory stalls is reduced by 52 %. The insert and read/update workloads also benefit with 31 % and 41 % fewer stalls on average. Notably, the number of stalled cycles equalizes the read/update workload as more cores are applied. This is due to increasing latch-contention caused by updates.

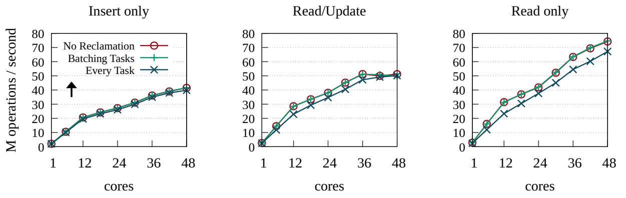


Figure 11: Scaling performance of EBMR in a task-based environment.

Preloading, however, requires prefetch instructions to be executed. Figure 10c demonstrates the number of performed instructions per operation. Prefetching utilizes about 245 additional instructions per operation—compared to the non-prefetching run. Nevertheless, these extra efforts reduce the memory stalled cycles to such an extent that prefetching still pays off.

6.3 Epoch Based Memory Reclamation

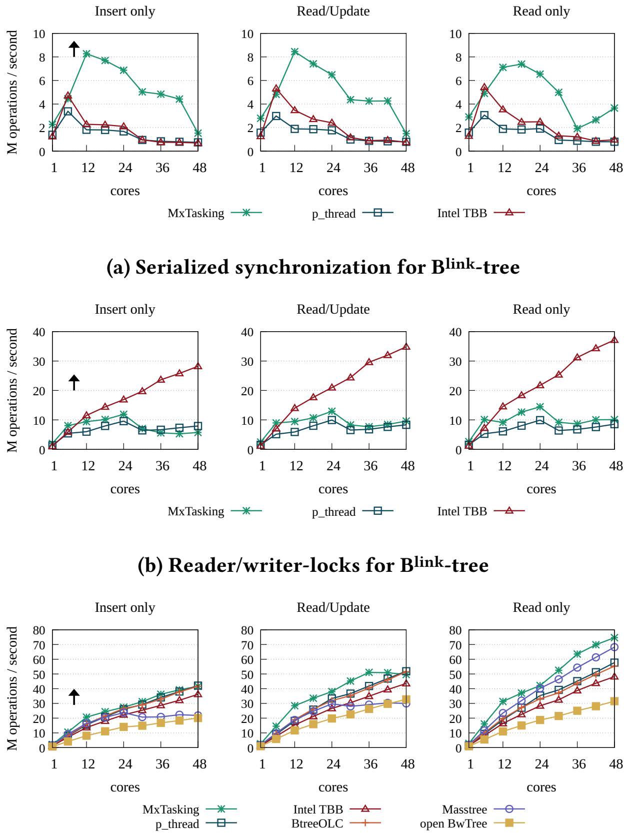
As discussed in Section 4.4, optimistic synchronization requires the coordination of concurrent physically removing and reading operations. Therefore, MxTasking adapts widely used EBMR (e.g., [33, 45, 46]) to a task-based environment. Instead of wrapping local epoch-updates around logical operations, a tree-insert including the traverse, for instance, we focus on individual tasks. MxTasking implements two different approaches: Synchronizing the local- and global epoch in advance of 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 in Figure 11 demonstrate that both mechanisms have little to no impact on performance, using omitting EBMR as a baseline. The most significant performance loss occurs during the execution of read-only workloads when enwrapping every Mx-Task with local epoch updates. Write-heavy workloads are almost not affected at all. Due to the slightly better performance, we will perform further measurements with batching-based EBMR.

6.4 Comparison of Tasks and Threads

We argue that MxTasks offer a superior abstraction level to build scalable software for modern and future many-core hardware easily—without decreasing performance. To study this hypothesis, we compare different programming models, libraries, and synchronization mechanisms: the B^{link}-tree on top of MxTasking, p_threads, and Intel's TBB tasking library. Additionally, we apply proven state-of-the-art index structures. Figure 12 shows the results.

Serialized Access. Spinlocks are widely used to serialize and, as a result, synchronize accesses to a specific resource. As discussed in Section 4, 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 12a compares the throughput of our B^{link}-tree implementation, using scheduling for MxTasking and spinlocks for TBB-tasks and threads. Applying TBB and threads, accesses to tree nodes are protected by spinlocks. MxTasks accessing the same tree node are, in



(a) Serialized synchronization for B^link-tree

(b) Reader/writer-locks for B^link-tree

(c) Optimistically synchronized B^link-tree and state-of-the-art

Figure 12: Throughput of different synchronization primitives and programming models of a B^link-tree and state-of-the-art index structures.

contrast, dispatched to the same task pool. Hence, tasks reading or writing the same tree node are serialized implicitly since MxTasks execute without interruption.

The results demonstrate that the scheduling-based synchronization offers a significantly better performance related to spinlocks until using logical cores (from 13) and the second NUMA region (from 25 cores). Although MxTasks avoids latching, we can observe two bottlenecks preventing this approach to scale. First of all, every operation starts by reading the root node of the tree. Regardless of subsequent steps operate 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, consequently. 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 simultaneously. 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 12b demonstrates the results using reader/writer-locks for synchronization. This way, MxTasks are primarily spawned at the local core, minimizing frequent accesses to a single task pool. Balancing the load in this fashion turned out to be a straightforward and effective strategy for the given workload. However, when using the second NUMA region, the throughput decreases, too. 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 45 % more lookups per second compared to threads. For both implementations, we borrowed the reader/writer-lock from *Facebook*'s *folly* library[4].

Contrarily, TBB provides different synchronization mechanisms, partially based on HTM. Applying the HTM-based reader/writer-lock to TBB, we notice less overhead due to latching and thus better performance: more than 2.6× compared to MxTasking and 3.7× to threads.

Optimistic Synchronization. MxTasking utilizes optimistic synchronization by providing versioned data objects and differentiating between reading and modifying tasks. Read-only annotated MxTasks perform optimistically while modifications, in contrast, are synchronized by the runtime. Most executed tasks in the B^link-tree benchmark are read-only, including the traverse needed by insert operations.

Using both 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* [46], *Masstree* [33], and *BtreeOLC* [28]. We compare our optimistic-synchronized B^link-tree implementations and the named data structures in Figure 12c. State-of-the-art implementations are borrowed from the *index-microbench* framework[5].

Given the insert-only workload, we observe comparable results for the B^link-tree based on threads and MxTasks, as well as for BtreeOLC. As far as using cores from a single NUMA region, Masstree achieves similar performance. The B^link-tree implementation based on TBB is comparable to the results of Masstree but also scales beyond the first NUMA region. Notably, BtreeOLC does not provide memory reclamation in contrast to all other evaluated data structures.

For the mixed read/update workload, MxTasking performs best until using logical cores located in the second NUMA region (from 37 cores). When applying all available cores, the thread-based B^link-tree and BtreeOLC achieve 4 % more read/update operations per second, compared to MxTasks. This is due to increased latch-contention on the side of MxTasks. 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. Here, MxTasks accomplish 74.6 million lookups, 9.3 % more compared to Masstree (68.2 M). Note that both implementations benefit from prefetching. The thread-based B^link-tree and BtreeOLC provide 57.7 and 55.3 million read-operations per second. All measurements show that TBB suffers from the additional effort caused by the runtime environment.

[4] https://github.com/facebook/folly

[5] https://github.com/wangziqi2016/index-microbench

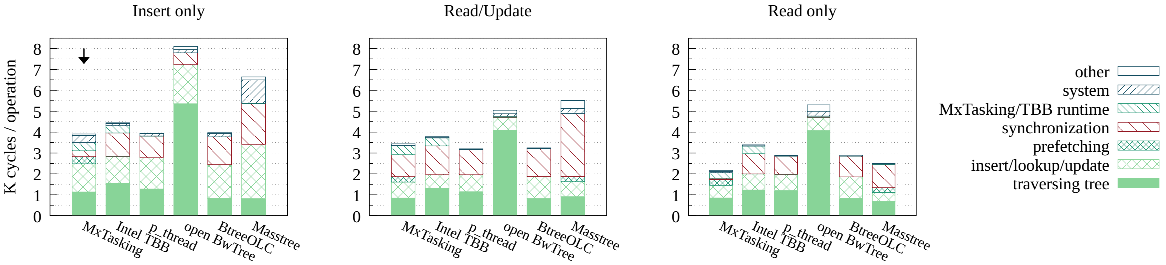


Figure 13: Detailed, cycle-based analysis of B$^{\text{link}}$-tree (using MxTasking, threads, and TBB) and state-of-the-art index structures.

Cycle-Based Analysis. Discovering the reasons for varying results, Figure 13 shows a cycle-accurate comparison between the task- and thread-based implementations. Note that MxTasking, Intel TBB, and p_thread are related to the B$^{\text{link}}$-tree. We distinguish between effort for traversing the tree, performing the insert/update/lookup operation, cycles spent in kernel mode (e.g., *syscalls*), as well as synchronization and memory reclamation. Additionally, we present cycles consumed by the runtimes of MxTasking and Intel TBB. For MxTasking and Masstree, we further show the complexity of initiating memory prefetches. We recorded those details using *perf*. However, the aggregated cycles only give an impression. Mapping spent cycles to function names might sometimes be ambiguous, e.g., as a result of inlining.

The results prove the effectiveness of the prefetching mechanism used by MxTasking: Traversing the tree requires fewer cycles when applying MxTasks, compared to threads and TBB tasks. It turns out that prefetching decreases synchronization costs by prefetching the header of tree-nodes containing the version counter. Accordingly, the additional effort for synchronization is lower for MxTasks than for TBB and threads. By this, MxTasking equalizes overhead coming along with tasks, mainly caused by task-spawning and -annotations. Comparable runtime-overhead is also observable for the TBB-scheduler. We are not able to break it down more precisely since perf does not provide revealing function names. We assume that this is an expense for load balancing, task-stealing, and scheduling.

The results confirm that the abstraction of tasks and simplification of 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 into threads and TBB manually requires considerably more effort: the application engineer is, possibly, forced to restructure the data structure.

In the end, there are only two extremes to choose from: leaving control entirely to the OS or taking control completely.

7 CONCLUSIONS

In this paper, we presented MxTasking, a task-based framework with run-to-completion semantic. The unique selling point of MxTasks is given by annotations, which offers the algorithm engineer to transfer knowledge from the 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 engineer's intervention. Furthermore, appropriately using fine-grained tasks completed by annotations opens up the possibility to design parallel data structures and algorithms, without exposed synchronization. Requirements of data object synchronization, e.g., type of isolation and the task's access intention, enable MxTasking to choose and inject the most efficient synchronization technique. As a result, MxTasking eases the development of latch-free data structures without performance degradation.

The first results of an MxTask-based key-value store promise great potential on modern and future many-core hardware. Using a B$^{\text{link}}$-tree as the foundation, MxTasks outperform classical threads by 29 % more lookups per second while using annotations reduces the implementation effort.

Looking ahead. MxTasking, as described in this paper, is part of our larger MxKernel[6] effort, where we try to resolve a dilemma in the design of hardware-conscious system stacks. Traditional operating systems essentially hide the inner characteristics of the underlying hardware, ruling out optimizations that could address just these characteristics. An often-seen circumvent is to overtake many operating system tasks—such as I/O, memory, or thread management—and perform them on the application side (through quirks such as thread or memory pinning). Conquering the entire system, however, is incompatible with efforts to consolidate services and/or share resources.

The vision of MxKernel is to make the interfaces between applications and the operating system underneath more transparent, in both directions. The annotation mechanisms of MxTasking are an important building block of that vision. Through annotations, applications can make their expectations and wishes explicit to the scheduling layer, obviating the need to conquer the entire system. In this sense, we see MxTasking also close to efforts on database/operating system co-design (e.g., *COD* [16]) or to facilities in modern operating system architectures (e.g., the *system knowledge base* of *Barrelfish* [5, 42]).

ACKNOWLEDGMENTS

This work was supported by DFG, Deutsche Forschungsgemeinschaft, grant number TE 1117/2-1. We would like to thank Maximilian Berens, Roland Kühn, Michael Müller, Olaf Spinczyk, and the anonymous reviewers for their helpful comments and suggestions.

[6] http://mxkernel.org

REFERENCES

[1] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreeniva Subramoney. 2004. Prefetch injection based on hardware monitoring and object metadata. *ACM SIGPLAN Notices* 39, 6 (2004), 267–276. https://doi.org/10.1145/996893.996873

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

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

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

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 29–44. https://doi.org/10.1145/1629575.1629579

[6] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices* 35, 11 (2000), 117–128. https://doi.org/10.1145/356989.357000

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

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *J. Parallel Distributed Comput.* 37, 1 (1996), 55–69. https://doi.org/10.1006/jpdc.1996.0107

[9] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, Vol. 1. 181–190.

[10] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 17–es. https://doi.org/10.1145/1272743.1272747

[11] Gilberto Contreras and Margaret Martonosi. 2008. Characterizing and improving the performance of intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 57–66. https://doi.org/10.1109/IISWC.2008.4636091

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

[13] Karl-Filip Faxén. 2008. Wool–a work stealing library. *ACM SIGARCH Computer Architecture News* 36, 5 (2008), 93–100. https://doi.org/10.1145/1556444.1556457

[14] Keir Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193

[15] Vaidas Gasiunas, David Dominguez-Sal, Ralph Acker, Aharon Avitzur, Ilan Bronshtein, Rushan Chen, Eli Ginot, Norbert Martínez-Bazan, Michael Müller, Alexander Nozdrin, Weijie Ou, Nir Pachter, Dima Sivov, and Eliezer Levy. 2017. Fiber-based architecture for NFV cloud databases. *Proc. VLDB Endow.* 10, 12 (2017), 1682–1693. https://doi.org/10.14778/3137765.3137774

[16] Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. 2013. COD: Database/Operating System Co-Design.. In *CIDR*.

[17] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proc. of the 36th annual international symposium on Computer Architecture*. ACM, 184–195. https://doi.org/10.1145/1555754.1555779

[18] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (2020). https://doi.org/10.14778/3430915.3430932

[19] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. 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*. ACM, 24–35. https://doi.org/10.1145/1516360.1516365

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

[21] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.

[22] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *Proc. VLDB Endow.* 9, 4 (2015), 252–263. https://doi.org/10.14778/2856318.2856321

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

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

[25] Jaekyu Lee, Hyesoon Kim, and Richard W. Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 1 (2012). https://doi.org/10.1145/2133382.2133384

[26] Philip L. Lehman and S. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670. https://doi.org/10.1145/319628.319663

[27] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. 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*. ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[28] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42.1 (2019), 73–84.

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

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

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

[32] Darko Makreshanski, Justin J. Levandoski, and Ryan Stutsman. 2015. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proc. VLDB Endow.* 8, 11 (2015), 1298–1309. https://doi.org/10.14778/2809974.2809990

[33] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*. ACM, 183–196. https://doi.org/10.1145/2168836.2168855

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

[35] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.* 11, 1 (2017), 1–13. https://doi.org/10.14778/3151113.3151114

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

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

[38] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins. *Proc. VLDB Endow.* 11, CONF (2017), 230–242. https://doi.org/10.14778/3149193.3149202

[39] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *The VLDB Journal* 28, 4 (2019), 451–471. https://doi.org/10.1007/s00778-018-0533-6

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

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

[42] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. 2008. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, Vol. 27.

[43] Pinar Tözün and Helena Kotthaus. 2019. Scheduling Data-Intensive Tasks on Heterogeneous Many Cores. *IEEE Data Eng. Bull.* 42.1 (2019), 61–72.

[44] John Tse and Alan Jay Smith. 1998. CPU cache prefetching: Timing evaluation of hardware implementations. *IEEE Trans. Comput.* 47, 5 (1998), 509–526. https://doi.org/10.1109/12.677225

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[46] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. 473–488. https://doi.org/10.1145/3183713.3196895

[47] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2010. Making address-correlated prefetching practical. *IEEE micro* 30, 1 (2010), 50–59. https://doi.org/10.1109/MM.2010.21

[48] Johannes Wust, Martin Grund, and Hasso Plattner. 2013. 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 2013, Koblenz, Germany, September 16-20, 2013 (LNI, Vol. P-220)*. GI, 487–501. https://dl.gi.de/20.500.12116/20773