XTasks: How to Make Efficient Synchronization and Prefetching Easy

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 `XTasking`, a task-based framework that assists the design of latch-free and parallel data structures. `XTasking` 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, `XTasking` also lays the foundation for our vision of a truly co-designed system of operating system and database management system.

ACM Reference Format:

. 2021. XTasks: How to Make Efficient Synchronization and Prefetching Easy. In *SIGMOD '21: International Conference on Management of Data, June 20–25, 2021, Xi'an, Shaanxi, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

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 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 sequencing accesses and cause overhead by contention.

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 [21, 28, 30, 31]. But despite the progress made, it remains difficult to design latch-free algorithms and data structures. Most of them are tailor-made—generalizing these specific approaches is ambitious and at significant expense. *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].

For utilizing the entire parallelism, present work needs to be carefully allocated 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) [39] and native support within OSs like fibers in Windows [4] 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 [13] this way. Not least because of those frameworks have just sparse knowledge regarding the application and its intention.

In this paper, we present `XTasking`, a task-based environment for today's and future many-core hardware. The elemental abstraction in `XTasking` is the `XTask`. An `XTask` 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 `XTasking` lies in the possibility to attach *annotations* to every `XTask`. With annotations, applications may convey characteristics of a task to `XTasking`, 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*. `XTasking` 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. `XTasking` 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 design of our tasking-approach enables us to take the framework one step further: `XTasking` can be extended into a *bare-metal runtime environment*. The use of specific knowledge about application behavior for scheduling or resource allocation has been

SIGMOD '21, June 20–25, 2021, Xi'an, Shaanxi, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

explored previously in the context of *database/operating system co-design* (albeit highly targeted at database applications only) [18]. With immediate control over underlying resources, such designs 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, `XTasking` provides the base for 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 introduces task-based parallelism in general. Afterward, Sections 3 and 4 present details of the `XTasking` runtime and annotation principles for memory-prefetching and synchronization. In Section 5, we provide practical insights into our tasking library. Our vision of the bare-metal runtime will be discussed in Section 6. The first results of a key-value store, built with `XTasks`, are demonstrated and discussed in Section 7. We conclude in Section 8.

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. Many 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 [4] 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 [10]. Targeting to simplify the engineers' work, Cilk focuses on the automatic load balancing of parallel applications and comfortable integration into existing software programs. For the synchronization of competing tasks, Cilk supports *lock*/*unlock* calls on a latch variable.

Inspired by Cilk, Intel® designed the TBB framework focusing on portability and robust performance [25, 39]. 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. 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 [15].

StarPU intends to provide fine-granular tasks for heterogeneous multicore platforms [5]. The authors argue that the modern hardware landscape features not only 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. Like TBB, StarPU leaves the synchronization to the user.

Tasks —or task-like fashions— 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 [17]. Tözün and Kotthaus provide a concept for scheduling database tasks to heterogeneous hardware and discuss the challenges of granularity and scheduling [44]. *TAMEX* translates logical query plans into task-graphs to gain more control and benefit from the load-balancing of fine-grained work packages in parallel settings [48].

Specifically, in the HyPer engine, *morsels* are kinds of 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.

In terms of transactional processing, *DORA* avoids disorganized data access across parallel transactions by transferring executing threads to the data instead of vice versa [36]. Comparable to tasks, DORA divides transactions into multiple (task-like) *actions* while the present data is partitioned logically to threads. Based on that, DORA distributes actions among threads for execution.

Bang et al. utilized tasks for various index structures like B-trees and hash-tables, as well as transactional workloads [6]. 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 [37]. Every time a coroutine attempts to access not cached memory, it executes a `prefetch` instruction and yields the coroutine. By that, 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.

2.2 `XTask` Abstraction

A key proposition of `XTasking` is to replace the application-facing control flow abstraction by what we call `XTasks`. An `XTask` 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 `XTasking`, such information can be attached to every `XTask` 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

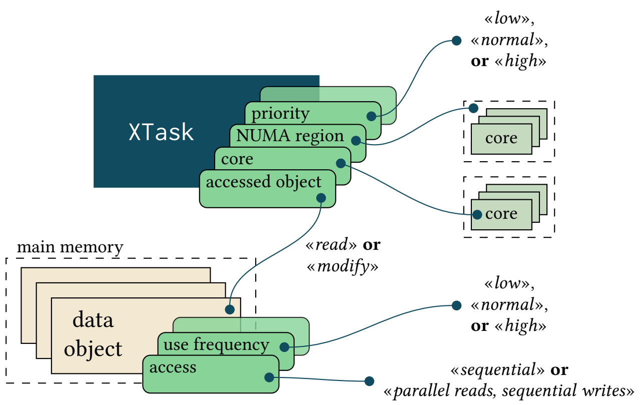


Figure 1: XTasks provide annotations for accessed data objects, core, NUMA region, and priority. Data objects, in turn, maintain metadata for access requirements and predicted usage.

and data. A complete, higher-level algorithm will be composed of a (possibly large) number of XTasks that jointly solve the given application problem.

To illustrate the tasking concept in this paper, we will use *tree navigation* as a running example. XTasking can be applied here by spawning a new task for every node visited during 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* [27] 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 XTasking runtime will select tasks for execution (possibly based on annotated information).

Figure 1 illustrates annotations for both tasks and data objects. For the moment, our annotation engine provides task-metadata for the priority to run with, the accessed data object, and the type of that access, whether reading or modifying. Alternatively, to accessed data objects, a task may be annotated with a specific core to run on or a particular NUMA region. Data objects, on the other hand, support annotations for synchronization requirements (e.g., parallel reads or sequential access), a NUMA region to be placed in, and the predicted usage frequency. The scheduler uses the latter for static load balancing. On-demand, annotations can be extended easily.

3 ANNOTATION-BASED MEMORY PREFETCHING

The optimization of memory access patterns is a good example to illustrate how annotated XTasks 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 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. But while efforts have been made to teach caching hardware the access patterns of database code [19, 24, 45, 47], hardware prefetching remains unfeasible beyond stream-based look-aheads. Software-based prefetching was shown to be more effective (e.g., [12, 23, 34, 38]), 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 XTasking 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—, XTasking will automatically inject software prefetching instructions on the application's behalf.

Thereby, we catch two birds with one stone. Prefetching becomes simple on the application end. In fact, the prefetching mechanism in XTasking is completely transparent to the application developer. All she needs to do is provide proper data object annotations to XTasks. At the same time, the prefetching mechanism is significantly more powerful than existing approaches. In contrast to hand-crafted solutions, XTasking 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.

XTasking 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 XTasking logical cores corresponds to a *worker thread* that will pick tasks from the pool and execute them. In this sense, XTasking 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 XTasking control over NUMA and locality effects.

Whenever the worker thread picks an XTask, 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 XTasking 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 when they start. To this end, the scheduler has to invoke *two* prefetches: first, the *task descriptor* has to

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

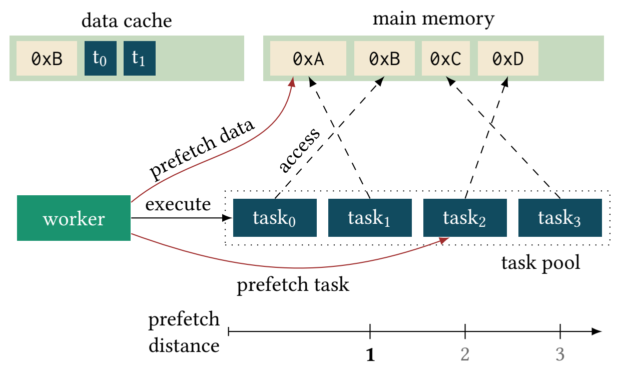


Figure 2: Execution and prefetching of XTasks. The worker knows tasks that will be executed soon and prefetches both tasks and data objects early enough to hide memory latencies.

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 2 illustrates this mechanism (assuming a prefetch distance of 1). Before executing *task*$_0$, the worker will issue a 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 the *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

So far, building massive parallel applications with concurrent control flows has become a challenge—efficient solutions often are tailored. Using latches for synchronization is a common and well-supported technique, but latches regularly suffer from overhead (and thus performance-decrease) and/or too coarse-grained utilization. Finding the perfect granularity for the scope of synchronization is not always trivial. Tasks, however, present an excellent base for this. They are fine-granular by design, accessing just one or a few data objects during execution. The use of XTasks almost endorses the developer to design the software fine-grained.

At the same time, this fine granularity assists the developer in transmitting execution and data dependencies to the runtime much better. By communicating those application characteristics transparently as annotations to XTasking, the execution engine is capable of synchronizing concurrent tasks—not only by latches. And even more, without the explicit usage of synchronization primitives by the developer's hand.

4.1 Integrated Synchronization Mechanisms

Instead, XTasking will select and inject an appropriate synchronization technique, based on the desired (and annotated) parallelism of the data object. Concurrency no longer becomes an enemy to be fought, but parallelism feels natural. As a result, the developer will drop synchronization at the application level—focussing on application logic only.

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. XTasking 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, XTasking 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 the 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. XTasks are governed by a run-to-completion semantic—all tasks scheduled to the same task pool are implicitly sequenced. Moreover, and this distinguishes XTasking from other known libraries, once dispatched, tasks are not stolen from other task pools.[2] Following this, we can avoid synchronization of concurrent accesses by scheduling all tasks accessing the same data object to the same task pool.

This approach evades 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 this end, 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 far 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 that are (by annotation) associated with heavily used data objects obtain a reduced total number of them. The task pool associated with the root node of a tree, for instance, is linked to only a few other nodes.

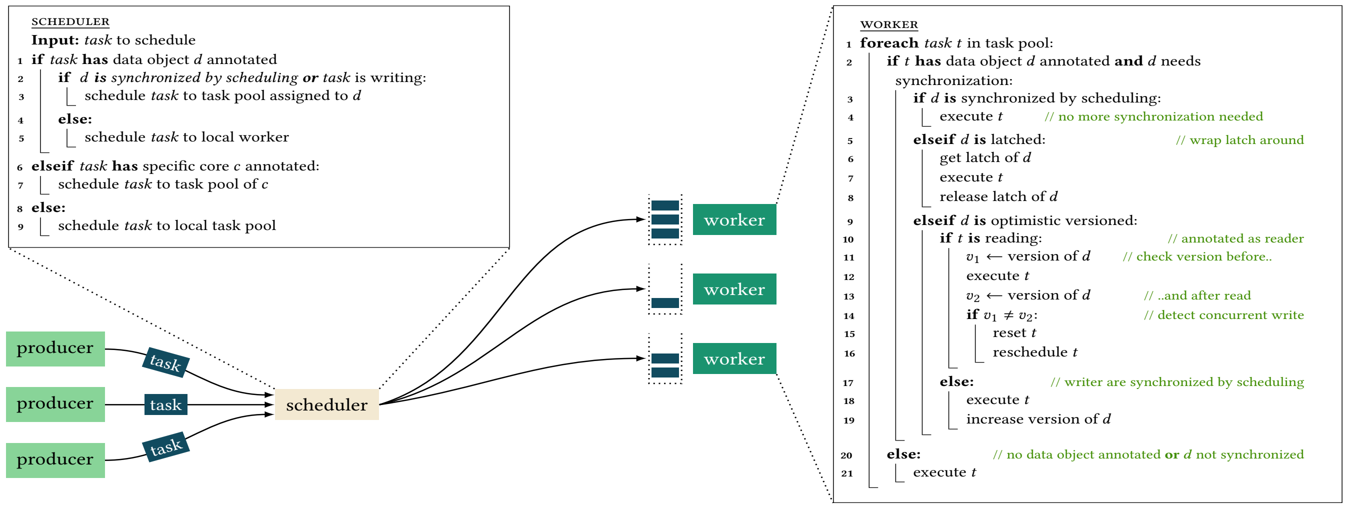
However, like basic spinlocks, sequencing the accesses to a data object by scheduling represents a pessimistic synchronization method, where XTasking does not distinguish between reading and modifying tasks.

This combination of scheduling and synchronization to avoid latches is unique and not known to us in any other task-based library. Instead, dependency graphs may be used to obtain a similar synchronization [9]. Although this is indeed possible on top of XTasking, using such a data structure implies additional runtime costs.

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

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



```
SCHEDULER
Input: task to schedule
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 needs
         synchronization:
3          if d is synchronized by scheduling:
4              execute t          // no more synchronization needed
5          elseif d is latched:                    // wrap latch around
6              get 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                 $v_1 \leftarrow$ version of d       // check version before..
12                 execute t
13                 $v_2 \leftarrow$ version of d          // ..and after read
14                 if $v_1 \neq v_2$:                 // detect concurrent write
15                     reset t
16                     reschedule 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 3: 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, ibbjects synchronization of tasks in the execution of those.

be read-only to locate the next node on traversing the tree. Optimistic approaches have proven their worth to benefit from parallel hardware [11, 28, 30]. 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. `XTasking` generalizes this concept as follows.

"Writing" annotated `XTasks` 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, `XTasking` checks the version counter of the data object. If it increased during the read operation, the `XTask` 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 `XTasking` takes care of version management.

Comparable to other optimistic procedures, operations (or tasks in our context) that delte a shared object must be treated with special care. Hazard pointers [35] and memory reclamation [20] are general approaches to protect read accesses while another thread frees the data at the same time. Even if currently not realized in `XTasking`, 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 Inject Synchronization

As illustrated in Figure 3, the synchronization of tasks bases on the interaction of scheduler and worker thread. The scheduler ensures to place `XTasks` 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 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 explicitly affect the placement of `XTasks`, 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. The synchronization itself, whenever necessary, is performed by the worker, wrapped around 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 already guarantees sequential 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, 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 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 verify a data object was not modified while performing a read operation, the worker checks the version before and after the execution (lines 10–16). Whenever the counter differs, the read access has to be retried at a later time. Therefore, the worker schedules the task

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

again (lines 14–16). To ensure that the version changes at all, it is incremented after the execution of a writing task (line 19).

5 XTASKING 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 [36] have already demonstrated the advantages for query- and transactional processing in a task-like fashion. `XTasking` advances the task-paradigm beyond the current standard by offering annotations for prefetching and implicit synchronization. This section reviews some practical aspects of using `XTasks` 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 show that `XTasks` are very robust concerning the granularity of a task.

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

Since latching has become a bottleneck for in-memory data structures on modern hardware, research investigated optimistic or fully latch-free procedures (e.g., [11, 28, 30, 31, 33, 46]). The B^{link}-tree [26], as a variant of the B-tree, focuses on reducing the number of simultaneously hold latches at a time. To this end, newly inserted nodes are not connected to the parent instantly, which eliminates 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 other traverse operations, even when there is no link from the parent to the new node. As a consequence, every operation is a concatenation of multiple tiny steps related to a single node.

Whereas thread-based implementations result in synchronous calls most of the time, `XTasks` (and comparable task-based solutions) get executed asynchronously. For instance, instead of calling a *lookup* method on a data structure that returns after the record is found, scheduling a *lookup task* that notifies the caller with the result is the way to go.

Tasks only have a limited view of the system. Every `XTask` solely performs on a single tree node, taking the appropriate node and the requested key as input parameters. The pseudocode in Figure 4 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 `XTasking` 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. Alternatively, we can schedule a follow-up task that handles the response.

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

Implementing the corresponding *insert task* is straightforward. Instead of reading the value from the leaf node (line 12), the task performs the insert operation. If this causes a node-split, the insert task 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 seated to the parent node, the new node can be reached by following the sibling pointer, as explained before. Even insert operations require no explicit synchronization since the engineer annotates the task as a writer when it arrives at the node to be modified. That hints `XTasking` to initiate the corresponding synchronization step, such as taking the writer latch or incrementing the version counter.

Annotating the modifying task as a writer at the appropriate time, however, becomes a minor challenge: When it comes too soon, parallelism may decrease because of sequencing writing accesses. Too late, contrarily, requires re-annotating and re-scheduling the task, which causes overhead. To inhibit, we need to know during traversal whether the following node is an inner or leaf node as

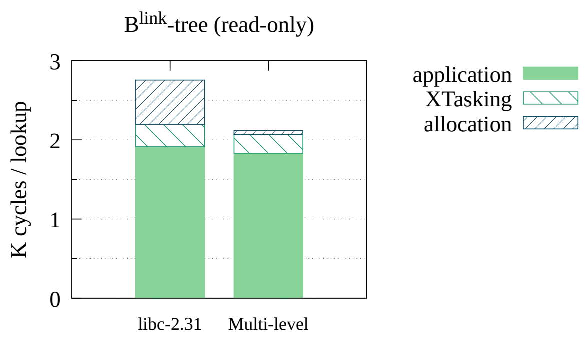


Figure 5: Aggregated CPU cycles for a single lookup on a task-based tree, using *GNU libc*'s `malloc` and our Multi-level allocator for task allocation.

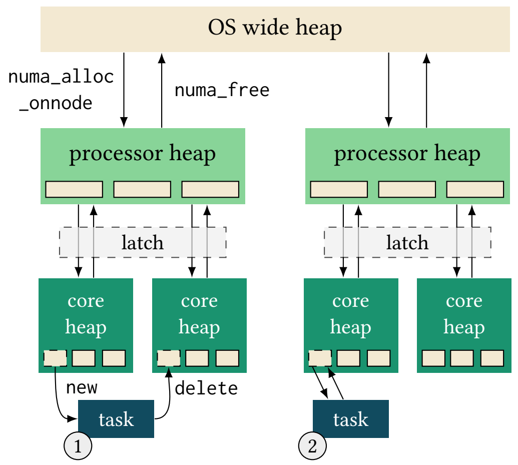


Figure 6: Reducing synchronization and enhance cache-awareness by using multiple levels for task-allocation.

modifications are related to leaf nodes. Since just loading the meta-data of the next node causes cache-misses, we introduce a new kind of node type: *branch* nodes represent inner nodes whose children are leaf nodes. With that help, we annotate a modifying task as a writer, whenever the current node is a branch.

5.2 Task Allocation

As indicated in the previous Section, `XTasks` 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 5 shows the CPU cycles spent 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 consumes 559 cycles per operation on a 48 core machine ($\sim 20\,\%$ of total CPU cycles)[3].

To overcome this costly aspect, we have designed a multi-level allocator, fitting seamlessly into the tasking-runtime. The architecture is 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 6 outlines this approach. Whenever a task needs to create a new one, it requests the local per-core heap for memory. Allocations on the per-core heap do not need any synchronization because `XTasks` are guaranteed to run-to-completion. 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 6 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 `XTasks` 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 5 demonstrates that our multi-level allocator has almost no overhead. Only 50 cycles are spent for task-allocation during a single tree-lookup.

5.3 Granularity of Tasks

While designing task-based applications or data structures, however, 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. Tasks accessing the B^{link}-tree described before, for instance, operate on a single node per `XTask`. For different applications, the granularity is arbitrary.

To give an example, think of accessing and processing tuples of an in-memory DBMS for query execution. Scheduling `XTasks` causes additional overhead that could become a bottleneck when tasks are too short-living. In particular, the exact costs for dispatching an `XTask` 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.

[3]Intel *VTune™* was used to analyze the *Microarchitecture Exploration.*

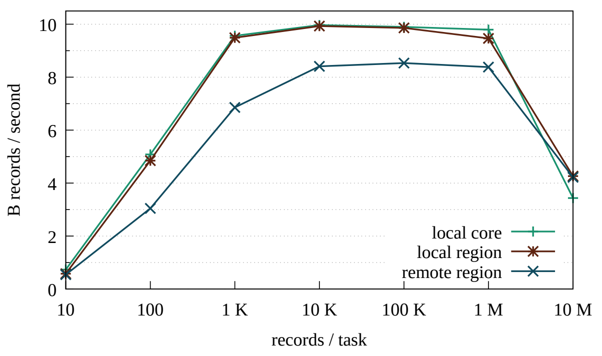


Figure 7: Execution of a simple aggregation query using different task sizes.

As shown in Figure 7, XTasks are very robust against performance penalties affected by granularities. For demonstration, we chose a workload with easy to change task-sizes and definable scheduling traits[4] (MIN-aggregation). The results verify a broad range of suitable task-granularities: Processing 1 000 up to 1 000 000 records per task behaves mostly equivalent. Aggregating only 100 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.

6 THE VISION OF XKERNEL

So far, we have considered XTasking primarily as an additional layer on top of the OS. We argue that XTasks 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 8 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 share this information with the OS. Using external libraries like libnuma [22] or extended interfaces for GPUs like OpenCL [43] allows applications 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.

[4]For *local region* and *remote region* runs, we configured the micro-benchmark to schedule every second task to a external task pool.

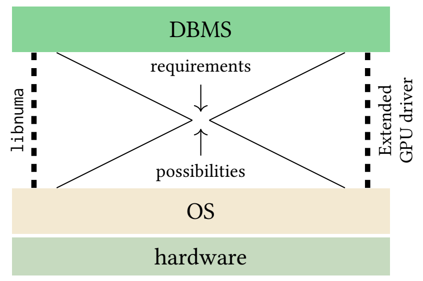


Figure 8: The interface between possibilities of the OS and requirements of the DBMS is narrow and often artificially supported by external libraries.

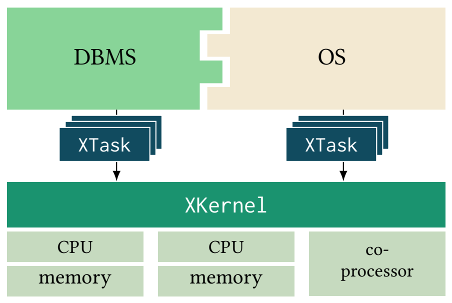


Figure 9: The architecture of XKernel: Both OS and DBMS run on top of the bare-metal runtime and use XTasks for control-flow abstraction.

With our bare-metal platform XKernel, we address the issue of insufficient interfaces between DBMS and OS. XKernel acts as a lightweight layer between the hardware and performance-critical applications, such as DBMSs. As shown in Figure 9, both OS and DBMS run as equal peers on top of XKernel. 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* [40]. Moreover, the DBMS can implement critical services on its 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 XKernel.

In addition to exploring the hardware components present in the system, XKernel primarily offers a lightweight abstraction of control flows. For this, the XTasking introduced in Section 2 is utilized with one worker instantiated on each logical CPU core. Applications running on top of XKernel employ XTasks 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 XTasks, 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 [16]. 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. `XTasks` exploit a way to use that heterogeneous hardware. Developers can provide implementations for various devices such as GPU and CPU for an `XTask`. Based on execution times, which can be annotated by the developer, and the load factors of possible processing units, `XKernel` will schedule the tasks to devices. For example, when an `XTask` has implementations for both CPU and GPU, where the GPU variant promises more performance. `XKernel` 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. *COD* [18] is a well-known and proven example, providing a richer interface between OS and DBMS through a central infrastructure. Based on the System Knowledge Base (SKB), applications and OS are enabled to exchange their knowledge about application requirements and OS state. The OS, on the one hand, can make use of this to schedule existing hardware resources—memory and computing resource for instance—more efficiently and with respect to the DBMS. Vice versa, the DBMS is enabled to to discover hardware-specifications of the underlying system through a service provided by the OS. The concept of the SKB was already introduced by the Barrelfish OS [7, 42], a multikernel that challenges problems associated with the growing number of cores and heterogeneity.

The equivalent in `XKernel` may the usage of annotated tasks. Those annotations hint the kernel with the application requirements that `XKernel` 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 `XKernel` that offers equal privileges to all.

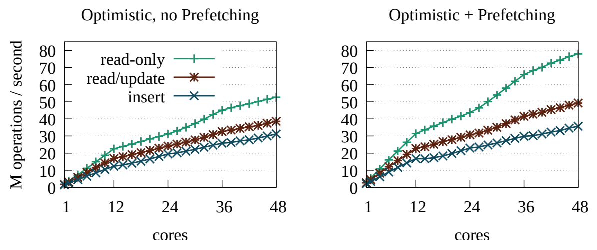
7 EXPERIMENTAL EVALUATION

To study the behavior and potential of `XTasking` in real-world scenarios, we use an in-memory B^{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.

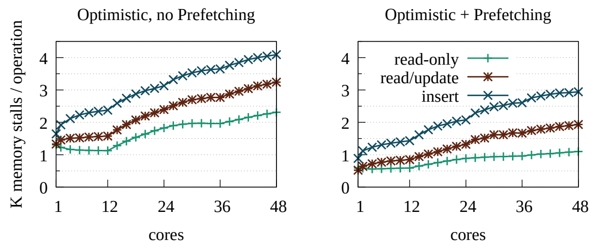
7.1 Environment

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 12 × 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. From then, hyperthreading cores are added step by step.

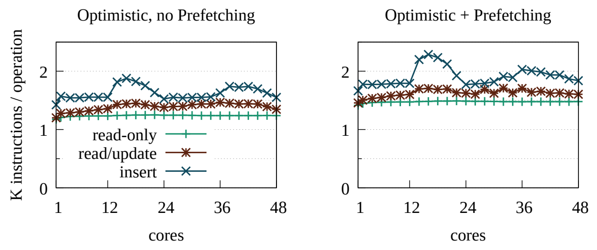
Following former work [46], we rely on the Yahoo! Cloud Serving Benchmark (YCSB) [14]. We use workloads **A** (*read/update*, 50/50) and **C** (*read-only*), both with Zipfian distribution and 100 million



(a) Throughput



(b) Memory stalls per operation



(c) Executed instructions per operation

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

operations. Before running each workload, the tree is initialized with 100 million records. *Insert* results are taken from the initialization phase of workload **A**. The tree stores pairs of 64 b keys and 64 b payloads within 1 kB sized nodes.

Ubuntu 20.04 was used as OS, *clang* 10.0.0 as the compiler, configured to apply 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 Annotation-based Prefetching

As discussed in Section 3, the fine granularity of tasks allows an exact prediction on which data an `XTask` will access. Figure 10 compares the B^{link}-tree build on `XTasking` with and without annotation-based prefetching. Experiments regarding the prefetch distance indicated that the results behave as expected: If the interval is too small (e.g., 1 for prefetching the next task ready), the workload will 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 3, 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 20 % higher insert, 30 % higher read/update, and 44 % higher read-only throughput rates on average. We demonstrate the outcome in Figure 10a. Especially the tree-traversal, which bases on binary search, benefits from this mechanism. Because binary 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 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 `XTasking` 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 up to more than a half. The insert and read/update workloads also benefit with 32 % and 44 % fewer stalls.

Preloading, however, requires `prefetch` instructions to be executed. Figure 10c demonstrates the number of performed instructions per operation. Prefetching results in ordinary about 239 additional instructions per lookup and 294 per insert, as well as 270 additive instructions per operation during the read/update workload—compared to the non-prefetching run. Nevertheless, these additional efforts reduce the memory stalled cycles to such an extent that prefetching still pays off.

7.3 Comparison of Tasks and Threads

We argue that `XTasks` 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 `XTasking`, we also implemented the B^{link}-tree on top of *p_threads* and the tasking library Intel TBB, which pursues similar goals as `XTasking`. Besides, we evaluate three different techniques for synchronizing parallel control flows: Sequencing of all accesses to a node, latches for parallel reads and exclusive writes, and an optimistic approach. Figure 11 shows the results for all programming models and synchronization procedures. We will first examine the various programming models individually.

`XTasking`. Figure 11a presents the results for filling and reading an `XTask`-based B^{link}-tree. We evaluated three different synchronization procedures, 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 `XTasks` 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 12 cores with 7.5 million insert- and read/write operations as well as 8.3 million lookups per second. As soon as the second NUMA region comes into play (26 cores and more), 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 11a (center). This way, tasks are primarily dispatched to the core they are produced by to minimizes overhead by scheduling. Balancing the load in this fashion turned out to be straightforward as well as an effective strategy for the given workload.

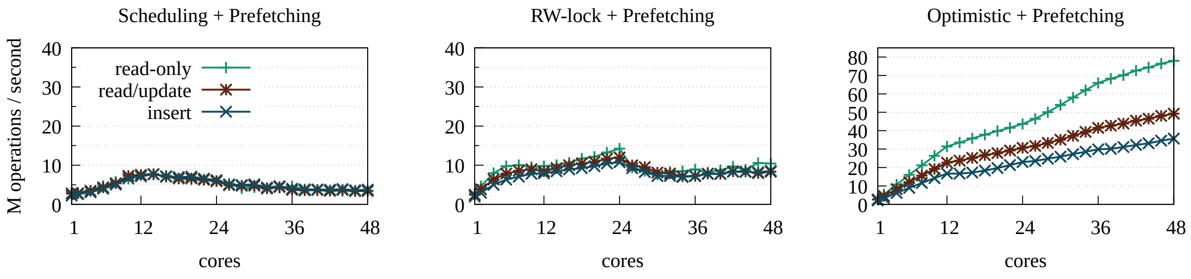
However, when using the second NUMA region, the throughput decreases again. In this case, the additional effort for keeping the latch variable coherent has a negative effect and causes communication costs across the sockets. The maximum throughput is reached at 10.5 million inserts, 11.4 million read/write operations, and 14 million lookups per second, utilizing 24 logical cores (in the same NUMA region).

Other works have already pointed out the advantages of optimistic synchronization [11, 30]. `XTasking` 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. Many tasks in the B^{link}-tree benchmark, even for insert operations, are read-only 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 achieve 36.9 million inserts, 50 million mixed (read/update) operations, and more than 77 million lookups per second. The tasks benefit, especially, from prefetching of data objects—they are already found in a local cache when accessed.

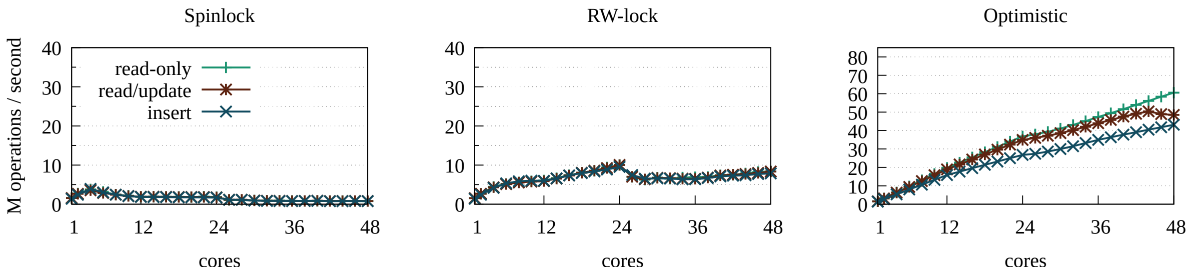
p_threads. We present the results for the B^{link}-tree-benchmark built on top of traditional threads in Figure 11b. Utilizing spinlocks for sequencing all accesses to one node ends up with a throughput of 3.5 – 3.9 million operations per second at four cores. After that, the throughput decreases steadily using more than four cores because of cache-coherence overhead. Using reader/writer-locks to enable parallel read operations to a node achieves 9.5 million inserts, as well as 10 million read/update and lookup operations per second with 24 cores and decreases afterward. We borrowed the reader/writer-lock implementation from *Facebook*'s *folly* library[5].

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 43 million inserts and 60.5 million lookups per second, using all available 48 logical cores. The highest throughput for the read/update workload is at 50.4 million operations per second at 44 cores. This result is comparable to the *BtreeOLC* [28], which ends

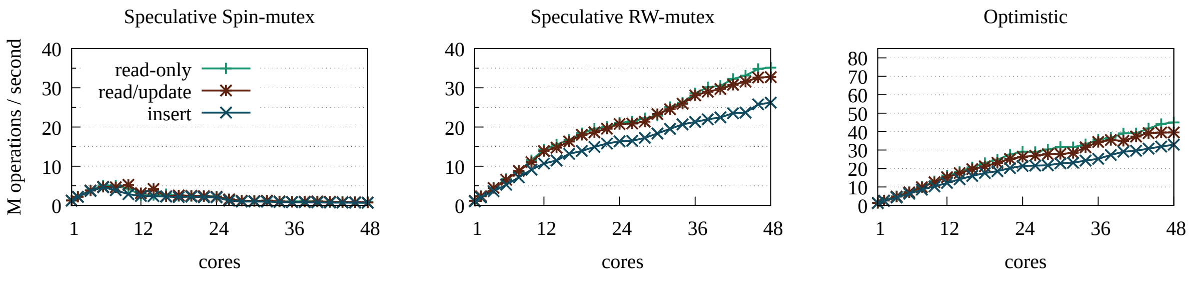
[5]https://github.com/facebook/folly



(a) `XTasking`



(b) p_thread



(c) Intel TBB

Figure 11: Comparing the throughput of different programming models and libraries for a B$^{\text{link}}$-tree-benchmark.

up with 41.2 million inserts, 50.4 million reads/updates, and 54.9 million lookups per second on our hardware.

Intel TBB. As a second task-based approach, we evaluated a B$^{\text{link}}$-tree based on Intel TBB using version 2020 Update 3. The tasks implementations of lookup-, update-, and insert- operations are built similar to `XTasking`, except synchronization, which is part of the application layer since TBB does not include automatic synchronization. Figure 11c presents the results. For synchronization, we use on-board synchronization: *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, compared to OS mutexes and standard spin- or reader/writer locks.

Using the *speculative_spin_mutex*, which is described as *scalable* by TBB documentation, the tree-benchmark gains the highest throughput with 4.6 – 5.2 million operations per second at ten 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 26.2 million insert-, 32.7 read/update-, and 35.1 million read-only 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 32.9 million inserts, 39.6 million reads/updates, and 45 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 11 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 `XTasks` avoids latching by scheduling, we can

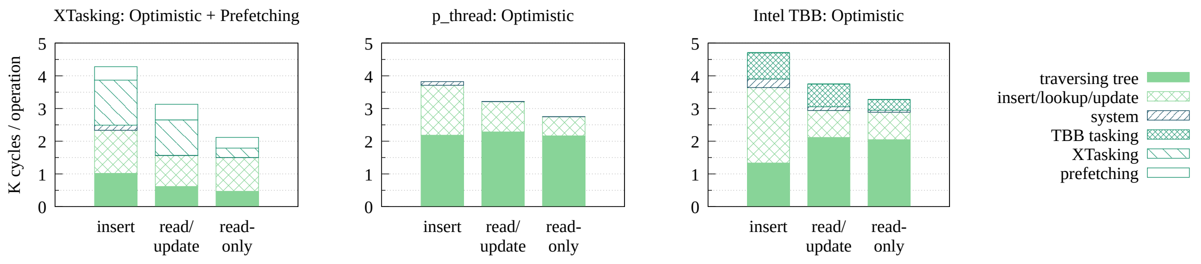


Figure 12: Detailed, cycle-based analysis of B^{link}-tree implementations using XTasking, threads, and TBB for control-flow abstraction.

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. That also applies to (speculative) spinlocks. Secondly, pushing tasks to task pools of other cores involves overhead due to cache-coherence, even if the operation itself is atomic. Especially the task pool associated with the root node is affected since many producers try to (atomically) insert tasks simultaneously. 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, the operations are, most notably, bound by latching overhead, which is due to cache-coherence. On the XTasking side, we can observe a benefit of up to 39 % more lookups per second compared to threads. Nevertheless, both programming models do not scale properly synchronizing by latches. Using HTM-based reader/writer-locks as done by TBB, we notice less overhead due to latching and thus better performance: more than 2.5× compared to XTasking and 3.5× to threads.

All three compared libraries and models perform best when using optimistic methods for synchronization. However, some differences are observable. While *p_thread*-based implementation provides the best results for insert operations with 43.1 million inserts per second, XTasking outperforms threads and TBB providing 77.5 million lookups per second. Under the mixed read/update workload, XTasking and the thread-based implementation supply similar results.

To discover the reasons for the differences, Figure 12 shows a cycle-accurate comparison between the task- and thread-based implementations. We distinguish between effort for traversing the tree, inserting/updating/searching, cycles spent in kernel mode (e.g., *syscalls*), and additional costs for the libraries XTasking and Intel TBB. For XTasking, we further differentiate between complexity for prefetching and other XTasking related work, most notably scheduling. We recorded those details using the Intel *VTune™* Profiler [3].

The results prove the effectiveness of the prefetching mechanism used by XTasking: Traversing the tree requires significantly fewer cycles when applying XTasks, 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 XTasking. That is primarily the effort for scheduling tasks that are moved from one core to another to sequence write operations. Contrarily, during the read-only workload, XTasks are dispatched solely to the local core. Also, the TBB scheduler involves some additional work, which we are not able to break down more precisely using VTune™. We assume that this is an expense for load balancing, task-stealing, and scheduling.

To summarize, we believe that the software-controlled prefetching of data objects, which is unique to the XTask 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 XTasking, a task-based framework with run-to-completion semantic. The unique selling point of XTasks is given by annotations, which offers the algorithm engineer to easily 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 intervention of the engineer. Furthermore, appropriately using fine-grained tasks in combination with task- and resource 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 XTasking to choose and inject the most efficient synchronization technique. As a result, XTasking eases the development of latch-free data structures.

Also, we presented our vision of XKernel, a bare-metal runtime for DBMS/OS Co-Design. XKernel 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, XTasks form the abstraction for work items.

The first results of an XTask-based key-value store promise great potential on modern and future many-core hardware. Using a B^{link}-tree as the foundation, XTasks outperform classical threads by 28 % more lookups per second, while the implementation effort gets reduced with the help of annotations.

REFERENCES

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

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

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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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.

[12] 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

[13] 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

[14] 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

[15] 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

[16] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1603–1618. https://doi.org/10.1145/3183713.3183734

[17] 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

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

[19] 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

[20] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel Distributed Comput.* 67, 12 (2007), 1270–1285. https://doi.org/10.1016/j.jpdc.2007.04.010

[21] 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

[22] Andi Kleen. 2005. A numa api for linux. *Novel Inc* (2005).

[23] 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

[24] 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.

[25] 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).

[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] 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

[35] 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

[36] 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

[37] 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

[38] 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

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

[40] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 9. https://doi.org/10.1145/2501620.2501623

[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] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 3 (2010), 66. https://doi.org/10.1109/MCSE.2010.69

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

[45] 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

[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