XTasks: How to Make Efficient Synchronization and Prefetching Easy

Anonymous Author
anonymous@anon.edu
Anonymous 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 XTasking, a task-based framework that assists the design of latch-free and parallel data structures. XTasking 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.

ACM Reference Format:

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 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 [18, 27, 29, 30]. 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 [28, 31].

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) [39] and native support within OSs like fibers in Windows [2] as well as Apple Grand Central Dispatch in macOS [40] 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 [11] this way. Not least because 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 principal 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.

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, 21, 34, 38]. 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, XTasking 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 XTasking, 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 XTasking 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 latch variable. The second powerful way of using annotations is to replace the application-facing synchronization mechanisms with annotations that hide the details of the synchronization mechanisms. Instead of manually implementing synchronization mechanisms, developers may simply express their desired type of isolation as an 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 XTasking 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 latch variable, built with XTasks, 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. 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 [2] and tasks in macOS [40]).

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.

Inspired by Cilk, Intel® designed the TBB framework, though with a focus on portability and robust performance [23, 39]. The latter aspect is primarily achieved 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 [13].

StarPU intends to provide fine-granular tasks for heterogeneous multicores platforms [3]. 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 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]. Törmä and Kotthaus provide a concept for scheduling database tasks to heterogeneous hardware and discuss the challenges of granularity and scheduling [42]. 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 [47].

Specifically, in the HyPer engine, morsels resemble tasks that execute segments of a query [26]. 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 [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 coronet, 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 coronettes while the memory subsystem loads the requested data into the cache instead of wasting cycles to wait for the load fulfilled. In the same manner, Jonathan et al. exploited coronettes to hide memory latencies in terms of state-of-the-art index structures [19].

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 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 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. Each task will visit 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 XTasking 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 XTask, 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). XTasking will exploit such information to (a) minimize memory access cost and (b) perform scheduler-assisted task synchronization.

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 XTasking 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 XTasking System Interface

XTasking 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 XTasking 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.

XTasking Library / Worker Threads. XTasking 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 lock-free operation (realized using a single atomic xchg), 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 is lightweight.

3 ANNOTATION-BASED MEMORY PREFETCHING

Once annotations have been attached, XTasking will use the annotated information to improve runtime performance and hardware
All she needs to do is provide proper data object annotations to will automatically inject software prefetching instructions. Tasking environments.

to dynamically adapt prefetching, e.g., to data locality in NUMA applications. Plus, there is now only a single point in the system where prefetch instructions even across task executions from different applications, but depends on substantial algorithm restructuring to work out. 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, 22, 43, 46]. Yet, hardware prefetching remains unfeasible beyond stream-based look-aheads. Software-based prefetching was shown to be more effective (e.g., [10, 21, 34, 38]) 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 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 the existing approaches. In contrast to hard-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.

Efficiency. We will now specifically look into how XTasking 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. Though, prefetching was shown to be more effective (e.g., [10, 21, 34, 38]) 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 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.

Prefetching Implementation. XTasking 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. XTasks will thus see their data already cached in the CPU when they start.

Zooming in to implementation details, a tasking:: spawn () call will not only place the XTask 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., task1 and its associated data are prefetched just before task0 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, 24, 33]. In contrast to these existing proposals, however, XTasking 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 [28, 31]. To summarize, all these mechanisms require the
Instead, with XTasking, 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, XTasking 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 XTasking will take care of the rest.

Before we delve into automatic mechanism selection and the inner workings of the XTasking scheduler, we will first revisit the synchronization mechanisms (or primitives) that XTasking currently supports.

### 4.1 Integrated Synchronization Primitives

Our current prototype of XTasking 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 XTasking will choose (unless the task requests a particular primitive explicitly through annotations). XTasking 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. XTasking 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, XTasking chooses a reader/writer-lock instead.

In XTasking, 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, 27, 29]). 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 XTasking) 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, XTasking will combine optimistic versioning with either a latch-based or a scheduling-based (see below) synchronization primitive.

In XTasking, 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 XTasking enables another powerful synchronization mode: scheduling-based synchronization.

As the most simple form of scheduling-based synchronization, XTasking can guarantee that tasks which access the same data object are executed sequentially. Such a guarantee is easy to make: XTasking 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 [36] or H-Store [20].

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.

### 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 be specified by the developer using explicit annotations or selected by XTasking. For choosing the synchronization mechanism automatically, the runtime uses a cost model considering given annotations that describe access properties. Especially the isolation
level, 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, XTasking 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, XTasking 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, XTasking 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 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 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 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. Applying the synchronization primitive, whenever necessary, is performed by the worker thread, wrapped around the execution of tasks. First, the worker evaluates the annotated data object of the subsequent XTask (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 [35],
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, XTasking implements an EBMR, similar to Silo [44] and the decentral procedure realized by the open BwTree [45]. 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 whereas 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 its local epoch with the value of the global one. 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. In XTasking, however, logical operations are divided into multiple XTasks, executed at different worker threads. Hence, it is ambiguous to determine the beginning and end of a logical operation processed by several tasks. Wrapping local epoch updates around every single XTASK-execution causes many fenced memory loads and stores. To avoid this potential inefficiency, XTasking updates the local to the global epoch after a limited number of executed tasks (and if the task pool is empty, 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, XTasking schedules corresponding tasks at the beginning of a new epoch.

5 XTASKING IN ACTION

Utilizing tasks to design data structures and algorithms differs in general from well-understood thread-based programming. Morsel-driven parallelism [26] and DORA [36] have already demonstrated the advantages for analytical- 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 Bink-tree as an example. Accordingly, we will examine the relevance of memory management regarding task-allocation and demonstrate the robustness of XTasks concerning the granularity.

5.1 Building a Task-based Bink-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, 27, 29, 30, 32, 45]). The Bink-tree [25], 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, 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 other 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, XTasks (and comparable task-based solutions) get executed asynchronously. For instance, instead of calling an insert method on a tree that returns on 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 XTASK 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 node it needs to hold 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.

Figure 6: Insert operation in a task-based Bink-tree. Since XTasking conducts synchronization, the application logic does not have to deal with concurrent access.

```c
// found correct leaf, insert value
node->insert(key,value)

// child is a leaf, next task will write
next = node->child(key)

// change to writing mode
ans = tasking::create_task<InsertTask>(next,key,value)

// child is a leaf, next task will write
next = node->child(key)

// change to writing mode
ans = tasking::create_task<InsertTask>(next,key,value)

// change to writing mode
ans = tasking::create_task<InsertTask>(next,key,value)

// change to writing mode
ans = tasking::create_task<InsertTask>(next,key,value)

// child is a leaf, next task will write
next = node->child(key)

// change to writing mode
ans = tasking::create_task<InsertTask>(next,key,value)

// found correct leaf, insert value
node->insert(key,value)

// child is a leaf, next task will write
next = node->child(key)

// change to writing mode
ans = tasking::create_task<InsertTask>(next,key,value)

// found correct leaf, insert value
node->insert(key,value)
```
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\textsuperscript{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 XTask, annotated with the next node (e.g., lines 8–11). Labeling the new task as a reader (line 10) enables XTasking 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. Notably, the demonstrated code comes without synchronization of competing tasks—rather, XTasking 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\textsuperscript{link}-tree node splits. For simplicity, we ignored the case of node splitting in this example. Splitting a node is required when no capacity is 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 XTask 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\textsuperscript{link}-tree-example indicates that XTasks are 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 concurrently.

Figure 7 shows the CPU cycles spent during a single lookup\textsuperscript{1} on a task-based B\textsuperscript{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 (\(\sim 16 \%\) of total CPU cycles)\textsuperscript{2}.

To overcome this costly aspect, we designed a multi-level allocator, matching seamlessly into the tasking-runtime. The architecture is mainly inspired by Hoard\textsuperscript{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.

\textsuperscript{1}We decided to use a read-only benchmark because nothing other than tasks are allocated.

\textsuperscript{2}Perf was used to analyze and track consumed CPU cycles.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{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.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8.png}
\caption{Reducing synchronization and enhance cache-awareness by using multiple levels for task-allocation.}
\end{figure}
Whenever a task needs spawns a new one, it requests the local heap for memory. Allocating from this heap is very lightweight since the runtime guarantees XTasks to run-to-completion, making synchronization redundant. Free memory blocks are stored within a LIFO list. Implicitly, the allocator places freed blocks at the top of the list (1). 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, this shuffles memory blocks 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. 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 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. XTasks accessing the Bink-tree described before, for instance, operate on a single node per XTask. 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 XTasks causes additional overhead that could become a bottleneck when tasks are too short-lived. In particular, the exact costs for dispatching an XTask depend on the targeted task pool: Transferring a task to any XTask through reusing the memory.

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.

6 EXPERIMENTAL EVALUATION
To study the behavior and potential of XTask in real-world scenarios, we use an in-memory Bink-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 [45], 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 -O3. Because all threads are pinned to corresponding cores, we disabled the system’s NUMA balancing option for all experiments. This way, the kernel will not migrate memory or threads between the regions.
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\textsuperscript{link}-tree build on XTask\textsuperscript{ing} 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 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 XTask\textsuperscript{ing} 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.

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 physically removing and reading operations. Therefore, XTask\textsuperscript{ing} adapts widely used EBMR (e.g., [32, 44, 45]) to a task-based environment. Instead of wrapping local epoch-updates around logical operations, a tree-insert with traversing, for instance, we focus on individual tasks. We implemented two schemes: 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 X-Task with local epoch updates. Modifying 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 XTasks 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\textsuperscript{link}-tree on top of XTask\textsuperscript{ing}, 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, XTask\textsuperscript{ing} supports synchronization by scheduling for exclusive accesses. Although it is well-known that serialization does not perform best for tree-like data structures, we want to
discuss some insights into the comparable scheduling-based synchronization. Figure 12a compares the throughput of our B\textsuperscript{link}-tree implementation, using scheduling for XTasking and spinlocks for TBB-tasks and threads. Applying TBB and threads, accesses to tree nodes are protected by spinlocks. XTasks accessing the same tree node are, in contrast, dispatched to the same task pool. Hence, tasks reading or writing the same tree node are serialized implicitly since XTasks 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 XTasks 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. Even when all subsequent steps run in parallel, by distributing tasks to further nodes and implicit more CPU cores, the inherently sequential access to the root bounds the throughput. This also applies to spinlocks. Secondly, moving tasks to task pools of other cores involves overhead. Even if the operation is atomic, the expense of cache-coherence can degrade performance. Especially the task pool associated with the root node is affected. Many producers try to (atomically) dispatch tasks simultaneously. That effect is similar to latches: many threads (or TBB tasks) want to modify the same cache line to acquire the root’s latch.

Reader–writer latches. A similar factor is observable when using reader–writer-locks, shown in Figure 12b. This way, XTasks are primarily dispatched to the local core to minimize overhead by task-spawning. 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 again. 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 XTasking, 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\textsuperscript{5}.

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

Optimistic synchronization. Other works have already validated the advantages of optimistic synchronization [9, 29]. XTasking implements this approach by enabling versioned data objects and differentiates between reading and modifying tasks. Read-only annotated XTasks will perform read-operations opportunistically while updates, in contrast, are synchronized by the runtime. Many tasks in the B\textsuperscript{link}-tree benchmark, even for insert operations, are read-only until reaching a leaf node to insert a value or an inner node to place a pointer to another node.

Using threads or TBB as a fundament, optimistic synchronization requires careful implementation on top of the application. Widely used state-of-the-art data structures like the open BwTree [45], Masstree [32], and BtreeOLC [27] also use optimistic approaches\textsuperscript{4}.

We compare our optimistic-synchronized B\textsuperscript{link}-tree implementations and the named data structures in Figure 12c.

Given the insert-only workload, we observe comparable results for the B\textsuperscript{link}-tree based on threads and XTasks, as well as for Btree-OLC. As far as using cores from a single NUMA region, Masstree achieves similar performance. Our B\textsuperscript{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, XTasking performs best until using logical cores located in the second NUMA region (from 37 cores). When applying all available cores, the thread-based B\textsuperscript{link}-tree and BtreeOLC achieve 4 % more read/update operations per second, compared to XTasks. This is due to increased latch-contention on the side of XTasks. As observed within the insert-only workload, Masstree scales until using the second NUMA region.

\textsuperscript{4}The implementations are borrowed from https://github.com/wangziq/2016/index-microbench
\textsuperscript{5}https://github.com/facebook/folly
The most significant differences are noticeable in the read-only workload. Here, XTasks accomplish 74.6 million lookups, 9.3% more compared to Masstree (68.2 M). Both implementations benefit from prefetching. The thread-based B\textsuperscript{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.

Cycle-based analysis. Discovering the reasons for varying results, Figure 13 shows a cycle-accurate comparison between the task- and thread-based implementations. XTasking, Intel TBB, and p_thread are related to the B\textsuperscript{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 caused by the runtimes of XTasking and Intel TBB. For XTasking 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 and might not always be assigned unambiguous, e.g., by inlined functions.

The results prove the effectiveness of the prefetching mechanism used by XTasking: Traversing the tree requires fewer cycles when applying XTasks, 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 XTasks than for TBB and threads. By this, XTasking equalizes overhead coming along with tasks, mainly caused by task-spawning and -annotations. Comparable 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 XTasking do not cause major 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 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 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 XTasking to choose and inject the most efficient synchronization technique. As a result, XTasking eases the development of latch-free data structures without performance degradation.

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

Looking ahead. XTasking, as described in this paper, is part of our larger XKernel 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 circumstance 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 design of XKernel is to make the interfaces between applications and the operating system underneath more transparent, in both directions. The annotation mechanisms of XTasking 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 XTasking 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 Barrellfish [5, 41]).
