Datenbank Spektrum (2020) 20:223–230
https://doi.org/10.1007/s13222-020-00357-5

SCHWERPUNKTBEITRAG

mxkernel: A Novel System Software Stack for Data Processing on Modern Hardware

Jan Mühlig[1] · Michael Müller[2] · Olaf Spinczyk[2] · Jens Teubner[1]

Received: 29 May 2020 / Accepted: 14 September 2020 / Published online: 6 October 2020

Abstract
Emerging hardware platforms are characterized by large degrees of parallelism, complex memory hierarchies, and increasing hardware heterogeneity. Their theoretical peak data processing performance can only be unleashed if the different pieces of systems software collaborate much more closely and if their traditional dependencies and interfaces are redesigned.
We have developed the key concepts and a prototype implementation of a novel system software stack named MXKERNEL. For MxKernel, efficient large scale data processing capabilities are a primary design goal. To achieve this, heterogeneity and parallelism become first-class citizens and deep memory hierarchies are considered from the very beginning. Instead of a classical "thread" model, MXKERNEL provides a simpler control flow abstraction: MXTASKS model closed units of work, for which MXKERNEL will guarantee the required execution semantics, such exclusive access to a specific object in memory. They can be a very elegant abstraction also for heterogeneity and resource sharing. Furthermore, MXTASKS are annotated with metadata, such as code variants (to support heterogeneity), memory access behavior (to improve cache efficiency and support memory hierarchies), or dependencies between MXTASKS (to improve scheduling and avoid synchronization cost). With precisely the required metadata available, MXKERNEL can provide a lightweight, yet highly efficient form of resource management, even across applications, operating system, and database.
Based on the MXKERNEL prototype we present preliminary results from this ambitious undertaking. We argue that threads are an ill-suited control flow abstraction for our modern computer architectures and that a task-based execution model is to be favored.

1 Introduction

It is quite remarkable how *Moore's Law* still prevails after more than half a century. Its consequences, however, have become very intricate. Rather than ever-increasing single-thread performance, today's hardware provides performance improvements in the form of high degrees of *parallelism*, increasingly paired with growing *heterogeneity*. Beyond the need to express algorithms in a parallel and, ideally, re-targetable way, the hardware trends shift performance bottlenecks toward communication and synchronization across a pool of diverse resources. Hard- and software must tightly cooperate to achieve performance in this new world [4].

With MXKERNEL, we set out to re-think the interplay of hardware, system software, and applications in the light of the shifting hardware landscape and the tremendously growing demands on data processing capabilities.

Classical system designs build on rigid interfaces that strictly separate concerns, e.g., between the "operating system"—in charge of managing resources—, the DBMS—responsible for managing data—, and applications, which are supposed to implement logic in a resource-oblivious way. Such a separation can hardly address the challenges that come with modern hardware. Applications have to jump through hoops to leverage modern hardware features. Classical system software stacks, on the other end,

✉ Jan Mühlig
jan.muehlig@tu-dortmund.de

Michael Müller
michael.mueller@uos.de

Olaf Spinczyk
olaf.spinczyk@uos.de

Jens Teubner
jens.teubner@udo.edu

[1] TU Dortmund University, Otto-Hahn-Straße 14, 44227 Dortmund, Germany

[2] Osnabrück University, Postfach 4469, 49079 Osnabrück, Germany

know very little about the actual characteristics and needs of individual applications. Under these premises, resource management essentially becomes a blind flight.

A key design goal of MXKERNEL, therefore, is to exchange such knowledge much better between the resource manager and the code running on top. We argue that threads—one of the most fundamental building blocks in today's application/OS interfaces—are a poor basis to express the relevant knowledge. On commodity hardware, they are too course-grained; resource access patterns often change considerably over the lifetime of a thread. In a heterogeneous environment, featuring, e.g., FPGAs and/or GPUs, "threads" might not even have a sensible meaning in some of these hardware components [5].

In MXKERNEL, the principal unit of reasoning are tasks—or MXTASKS—instead. Tasks represent a unit of work (rather than a straight-line sequence of code) to the system. Code-wise, they tend to be much smaller than classical threads; the equivalent of a single classical thread may fire a number of MXTASKS in MXKERNEL. Since they relate to a very specific unit of work, tasks are a good abstraction for metadata that describes the characteristics of the unit [24]. Thus, tasks can be annotated with metadata that provides hints on future software behavior to the system software. These hints can be used to optimize scheduling decisions and synchronization of concurrent tasks.

In the course of the paper we will focus on the task-based programming environment and how task annotations can be used to improve the performance and simplify the development of data processing applications. For details on how our system deals with heterogeneity, the reader may refer to our work on He..roDB [25]. Since the MXKERNEL Project is still in an early stage, not all details of the system have been fleshed out yet. So we are still exploring the set of useful annotations and their applications.

First we will give a short overview of previous work related to MXKERNEL. Sect. 3 describes the current state of our task runtime and the annotations that are currently available. Sects. 4 and 5 present examples how annotations can be used to ease development of parallel data processing applications and first results we obtained so far. Finally, Sect. 6 will give a short summary and conclusion.

2 Related Work

Control flow models with short tasks have become popular in several frameworks for parallel programming in user space. Apple's Grand Central Dispatch, the Cilk runtime system, and Intel's Threading Building Blocks follow this approach [27]. The maintainers of the mentioned frameworks argue, that tasks provide a suitable granularity for data parallelism on today's multi-core systems.

Making tasks a unit for scheduling decisions has been explored very recently also by Giceva et al. [10], who extended the Barrelfish operating system [2] by a support for *run-to-completion tasks*. Their main argument for doing so are the potentially better cache locality and the avoidance of ill synchronization decisions (such as de-scheduling a lock holder).

Server-based architectures such as *microkernel* operating systems partly solve the problem as well. If state is managed by single-threaded servers, the input message queues of the servers effectively serialize the requests and thus concurrent access is avoided by design. As queues can be synchronized in a lock-free manner, scalability tends to be good on manycore systems. The fos microkernel follows this idea in a more flexible manner: servers are "elastic'' and can have more than a single thread [33].

When designing scalable system software, one can also learn from other domains. For instance, embedded operating systems have a much stricter execution plan for control flows then general purpose server operating systems. An extreme case of that is *offline scheduling* of short tasks with a run-to-completion semantic – an option of the AUTOSAR OS specification [9]. As concurrency is planned ahead of runtime, the offline scheduler can make sure that race conditions are avoided even without synchronization at runtime [13]. Though, this approach works only for static task schedules in contrast to the dynamic task scheduling and synchronization of MXKERNEL.

Scalability and performance can also be improved by avoiding concurrency situations with an innovative system design. For example DORA [28], does not use one thread per transaction but instead one thread per data object, e.g. a part of a table. Transactions are then split into smaller actions which are executed by the thread that manages the data the action is going to access. So DORA avoids expensive data movements between caches and also reduces contention between threads, since actions that access the same data are executed by the same thread and thus never run in parallel. The ERIS system [16] extends the concepts of DORA by NUMA-awareness and a load balancing mechanism. Since DORA focuses on transactional workloads, ERIS reduces skew for analytical workloads.

MXKERNEL borrows the fundamental concept of a data-oriented architecture, but provides a general abstraction for data-driven applications. By supplying a lightweight layer between a full-fledgded DBMS on top and the hardare below, MXKERNEL intends to offer a programming model to ease the construction of scalable data-processing applications.

Psaroudakis et al. explored the benefits and the right granularity of tasks for highly concurrent analytical and transactional main-memory workloads in SAP HANA [30].

Our poster child for MXKERNEL will be a B^{link}-tree [19]. B-trees (and their derivations) are widely used index structures. However, much research has been investigated into parallel and latch-free tree structures. Bragisnky and Petrank built a latch-free B+-tree using atomic CPU instructions resulting in less contention and higher scalability compared to using latches for synchronization [7]. The Bw-tree [21, 32] focusses on scalability and cache-performance. Delta changes are applied in a latch-free manner. As a result, the Bw-tree needs no latches at all.

With Optimistic Lock Coupling, Leis et al. introduced an optimistic synchronization technique for B-tree, which is not latch-free but focusses on parallel reads [20]. As a result, they achieved high scalibility, especially for read operations. This approach is similar to OLFIT [8], where reads are applied optimistically without using a latch. Instead, they check whether concurrent writes have happened with help of using a version counter.

3 Task Model: mxtasks

For lack of suitable means to communicate application behavior to the resource management component, due to the rather generic and rigid system interfaces of common systems, applications worked around the limitations, imposed by these interfaces, in often rather crude ways. The common approach is to let the application code take over all resources from the operating system, then use low-level mechanisms such as *thread or NUMA pinning*, deeply building on the assumption that no resources have to be shared with other applications. But often this assumption proves to be false, as there are always OS services running (e.g. kernel threads) and often applications are run concurrently, e.g. a key-value store and a web-server. So these work-arounds often quickly lead to severe interference of applications in a system, competing for CPU cores, cache lines and memory. Often enough, deteriorating the performance of the system instead of improving it.

Hence, in MXKERNEL, metadata that provides hints on future software behavior can be provided to the resource manager through *annotations* to MXTASKS. Poster child examples where the resource manager could leverage such annotations could be information about *data location*—so the task can be scheduled close to the object that it operates on—or *communication patterns* with related tasks—so related tasks could be co-located on the same CPU core. *Data processing and database algorithms* fit particularly well into this pattern. Their behavior is often highly predictable; and there are many examples that show how the awareness of resources and their uses in database code can significantly improve performance [1, 14, 15, 18, 26, 31, 34].

Fig. 1 shows a simplified excerpt of MXKERNEL's task model. Every task can have multiple implementations. For example, one implementation that can run on a CPU core, one implementation that can run on a GPGPU, and finally an implementation for reconfigurable logic. For each task the task scheduler will make the decision on which CPU core or accelerator it shall be executed. In the remaining sections of this paper we will not address exploitation of accelerators again.

Tasks can have relations to each other, for example, caused by data dependencies. Furthermore, tasks have relations to data objects in memory or other resources, modeled here as `SharedResource` objects. Tasks can access data objects in `read` or `write` `AccessMode`. With this meta-

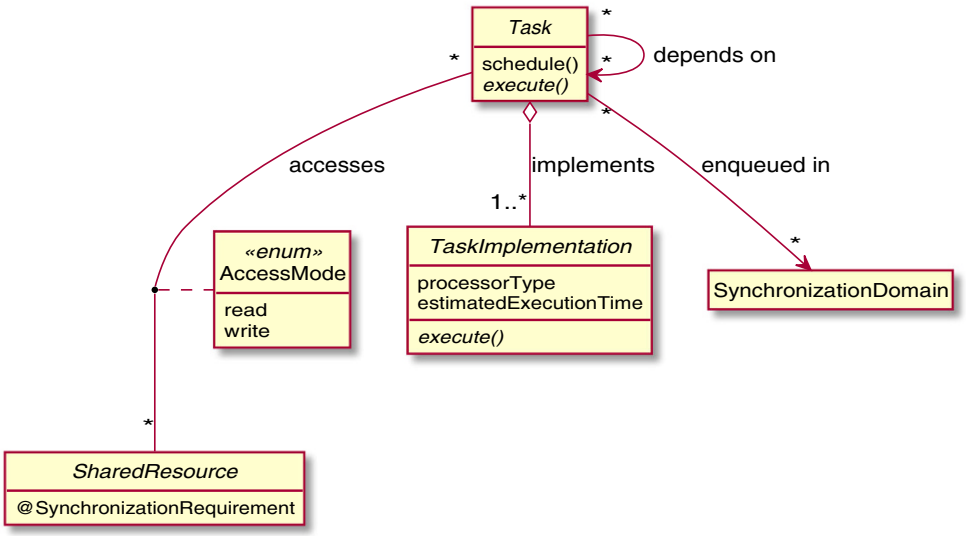

Fig. 1 Task model in MXKERNEL

data on tasks the resource manager of MXKERNEL, i.e. the task scheduler, has a lot of information on synchronization requirements and access pattern for the tasks that shall be run in the *near future*. It can exploit this information for an optimized task and data mapping, scheduling, and synchronization within the same component. Examples will be given in the following two sections.

Each example was implemented on a prototype of the MXKERNEL that can run natively on bare-metal and as a multi-threaded application under Linux. The examined application scenario was an *in-memory key/value store*. Its heart deliberately consists of an optimized implementation of a B^{link}-tree, a variant of the most-widely used data structure in databases. B^{link}-trees are a good device to study the benefits of locality and NUMA. Traversing them yields non-trivial data access patterns, and the tree shape results in non-uniform load on the individual nodes.

In experiments, we demonstrated the advantages of scheduling tasks on a core local to the accessed data, which can be decided based on task annotation [11, 12, 17, 23]. In our prototype, the access to each B^{link}-tree node is handled by a dedicated *mxtask*; that is, a B^{link}-tree search will spawn a new task for every tree level of the root-to-leaf traversal. Each of these MXTASKS is annotated by the programmer with a reference to the tree node that it will access, together with the desired mode of access. Thus, the scheduler of MXKERNEL knows for each enqueued task object which tree node it will access in advance.

To obtain the evaluation results, presented in the following two sections, we ran our prototype on top of Ubuntu Linux 20.04 LTS, as it provided easier means for measuring and debugging, and made the comparison with existing approaches more fair. For each logical CPU core, we instantiated one worker thread. Our evaluation platform consisted of two Intel(R) Xeon(R) CPU E5-2690 processors running Linux kernel 5.4.0 at 2.90 GHz. Both processors provide 16 hardware threads, which makes a total of 32 hardware threads on our platform and two NUMA regions. Hyper-threading has been activated for both processors during the benchmarks. We used the Yahoo Cloud Service Benchmark with 50 million inserts and 50 million lookups to evaluate the scalability of our prototype. Each worker thread has been pinned to a dedicated CPU and it has been ensured that there were no other applications sharing resources with our benchmark. Furthermore, we employed libnuma for numa-aware allocation of task queues, task objects and tree nodes.

4 Model Exploitation: Automatic Prefetching

One example that shows the benefits of using tasks, is that it is easy to use automated prefetching of tree nodes. As

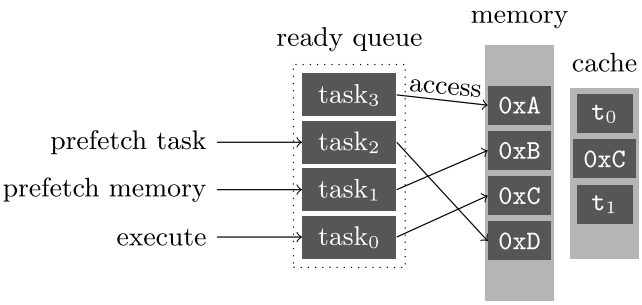

Fig. 2 MXKERNEL prefetches tasks and accessed data automatically with help of annotations

shown in Fig. 2, the scheduler performs prefetching of task metadata and accessed data fully transparent for the application. Everytime when a task is dequeued from the ready queue, the scheduler prefetches the data, accessed by the following task, and the metadata of its successor. So while one task is executed by the CPU, the data for the next task in line is already transfered to the cache. Since, the scheduler provides automatic prefetching of data, the programmer does not have to reason about how to implement prefetching for his specific application, instead she just needs to annote her tasks with the referenced data.

Fig. 3 shows the effect our automatic prefetching mechanism has for resource efficiency on a single core.

As can be seen in the figure, pre-loading significantly reduces the number of CPU cycles that B^{link}-tree traversal will stall and wait for memory (Fig. 3a), with immediate effect on the overall B^{link}-tree throughput (Fig. 3b).

In addition, we evaluated the prefetching mechanism using lookups on a B^{link}-tree with diverging cores. Fig. 4 shows the comparison of tasks with and without prefetching. As a result, pre-loading the tasks annotated data structures results in 20% more lookups per second on average. This is mainly caused by less CPU cycles wasted during memory stalls.

5 Scheduling Challenges: Course-Grained Work Stealing

As by Amdahl's Law, scalability and synchronization are tightly coupled, and have to be regarded together. Not doing so, inevitably leads to not only reducing scalability, but even more reversing it. So the method for synchronization and the granularity of critical sections has to be chosen carefully [6]. Even though fine-granular latches for synchronization are the common method and can improve scalability, their usage is error-prone and can easily lead to synchronization errors [22].

Therefore, tasks in MXKERNEL are run with *run-to-completion* semantics, i.e., they are not preempted by the sched-

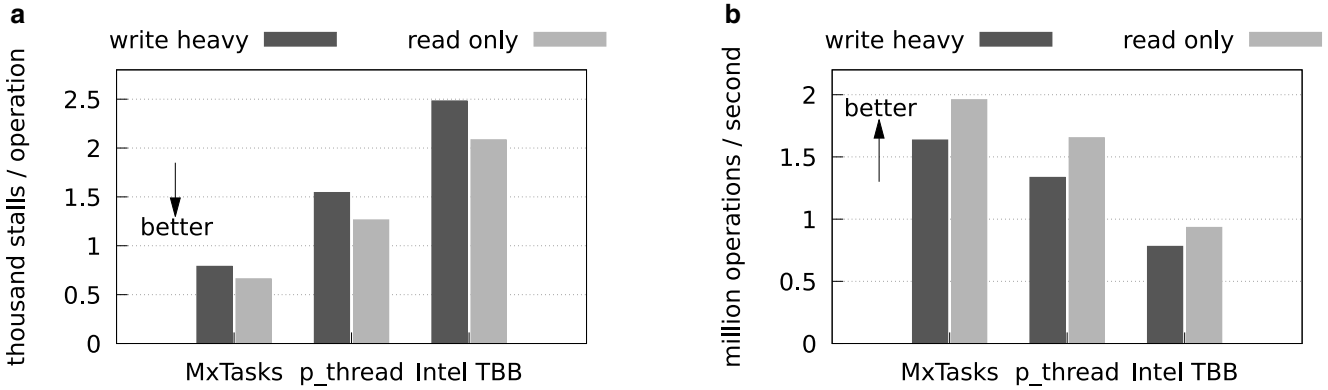

Fig. 3 Comparison of throughput and memory stalls using metadata for prefetching data objects, accessed by MXTASKS. **a** Memory Stalls, **b** Throughput

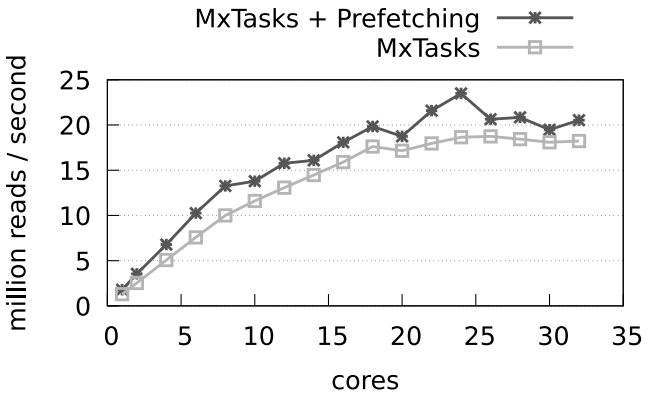

Fig. 4 B^{link}-tree lookups with and without automatic prefetching

uler. This guarantee can significantly ease software development, while at the same time avoid expensive latches. To illustrate, tasks that execute on the same CPU core cannot interfere with one another, because of the *implicit serialization*.

Building on that observation, it is sufficient to enqueue tasks, competing for a shared resource, in a shared waiting queue, serializing them that way. This queue is then processed by a CPU, running each task in the queue one after another. As waiting queues can be implemented latch-free, there is no need for any latches for this synchronization method to work.

Hence, the MXKERNEL provides the possibility to create waiting queues for shared resources. It is not necessary to pay the overhead of thousands of queues for all data objects, e.g. one queue per tree node. In contrast, mulitple resources can be joined in more course-grained *synchronization domains* as long as the sequential execution of tasks that access the same resource is guaranteed. A synchronization domain, thus, is a kind of monitor, managing the access of activities to shared resources. Moreover, synchronization domains are also used as ready-queues for tasks.

The MXKERNEL employs a two-level scheduling approach for scheduling tasks and synchronizing them. On the first level, which is entered upon spawning a new task, it is determined into which synchronization domain a task will be enqueued. To make this decision the scheduler uses the annotations of the task regarding its access patterns. Knowing which data object a task will access, the scheduler can find the appropriate synchronization domain by looking it up in its mapping of data objects to synchronization domains. On the second level, synchronization domains will be scheduled onto CPUs. The exact strategies for scheduling synchronization domains can be configured by the programmer to fit the needs of the application.

When mapping resources to synchronization domains, the scheduler has to pay attention to not map to many resources to a single synchronization domain, since only one CPU can process it, and the more resources are mapped to a single synchronization domain, the more the load of the processing CPU increases. Hence, to avoid overloading single CPUs, the scheduler periodically redistributes the shared resources onto the synchronization domains, such that the load on each synchronization domain, and thus CPU, is balanced.

Based on that, the in-memory B^{link}-tree that we implemented does *not* require latches for synchronization at all. Each tree node is statically assigned to a synchronization domain, and tasks that access a node will always be scheduled on the assigned synchronization domain.[1] Each synchronization domain can be mapped to a CPU core for execution by various means in the MXKERNEL. the simplest mapping being a one-to-one mapping of synchronization domains to CPU cores.

Fig. 5a, shows the throughput of our B-link tree, using a static one-to-one mapping of synchronization domains to

[1] The underlying per-synchronization domain task queues are synchronized algorithmically in a wait-free manner.

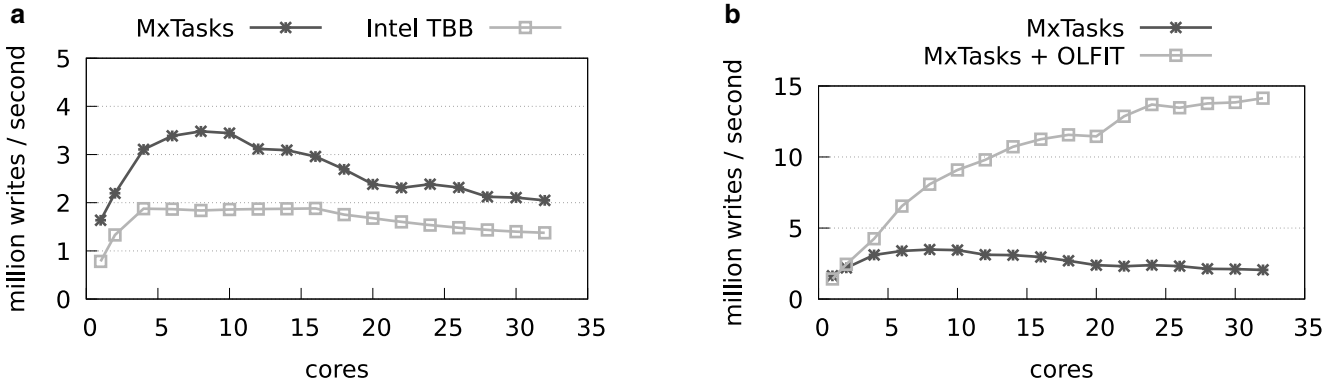

Fig. 5 Throughput of B^{link}-tree-demonstrator. **a** MXTASKS vs. Intel TBB, **b** MXTASKS vs. MXTASKS with implicit optimistic locking (OLFIT)

CPU cores, with increasing core count. We implemented our B-link tree using MxTasks and Intel TBB for comparison, where Intel TBB utilizes reader-writer latches. As can be seen, the reduced synchronization overhead achieved by this simple strategy can already have *performance advantages* compared to traditional reader-writer latches.

Yet, it may introduce a new bottleneck that thread-based execution environments may not suffer from in the same way: Some tree nodes—the root node in particular—will be accessed more often than others, causing *load imbalances* in the system. B^{link}-tree operations are a good example to demonstrate skew effects that will arise in many environments.

To mitigate the problem, we implemented a more sophisticated strategy to map synchronization domains to CPU cores for execution. Knowing that most synchronization domains will only have a few tasks enqueued to them as they will likely manage the access to a rarely accessed leaf node, we allow a CPU core to execute tasks from more than one synchronization domain. Thus, we instantiate more synchronization domains than there are physical cores. The actual mapping of synchronization domains to a CPU core is then decided by a work-stealing algorithm, similiar to that of Cilk, first described by Blumofe and Leierson [3]. The difference here to Cilk's scheduler is that we do not steal tasks from other worker's queues, but instead our algorithm, that we call *coarse-grained work stealing* steals a whole synchronization domain, and thus a whole queue of tasks, from another CPU. This way, the MXKERNEL can still provide implicit serialization by qeueing and benefit from the load balancing achieved via work stealing. The effect of this mechanism is illustrated in Fig. 6.

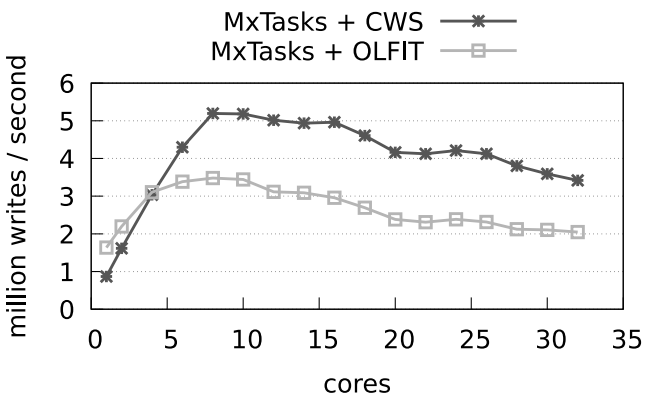

Fig. 6 MXTASKS vs. MXTASKS + work stealing

With or without course-grained work stealing, the question which tree nodes to co-assign to the same core (or domain) may affect performance, particularly in NUMA environments. Intuitively, it may be desirable to assign entire sub-trees to the same CPU core, so as to avoid the cost of invoking tasks across cores. In experiments [29] we showed that deliberately migrating to a new core for each level of a B^{link}-tree traversal may be a better choice. Task invocation happens asynchronously in MXKERNEL and is thus cheap, whereas distributing the tree along its levels results in better load balancing across cores. These results confirm that there are trade-offs between synchronization and data locality, which we plan to investigate further.

For larger core counts, exclusive data access—as provided by implicit serialization—will still limit the achievable scalability. High-performance B^{link}-tree implementations therefore adapt *object versioning* as a lock-less concurrency protocol that permits parallel reads (e.g., OLFIT [8] or Bw-trees [21]). Therefore, we adopted object versioning with optimistic reader execution as additional synchronization method in MXKERNEL.

To allow the easy usage of this method in an application, MXKERNEL provides an abstract object type with version numbering as a foundation for user-defined data types and tasks that can be annotated as optimistic.

The scheduler then implements the OLFIT protocol in the following way. Each time an optimistic task is scheduled, the scheduler runs it in parallel to other tasks, accessing the same resource, if it is a reader. While enqueing it in the corresponding synchronization domain and increasing the resource's version number after execution, if it is a writer.

Since the protocol for object versioning is directly implemented in the scheduler, the programmer has not to deal with the intricacies of implementing such a protocol for her data structures and algorithms. All she needs to do is deriving her data structures needing synchronization from the abstract object type, provided by the MXKERNEL, and annotating each task that access them as optimistic.

In early experiments, we could show that our task-based scheduling can benefit from this optimistic approach (see Fig. 5b). Current (B^{link}-tree) implementations, however, still put the responsibility of such protocols on the developer of the data structure and on the application developer, who has to choose the right abstraction for the particular use case without much awareness of the underlying hardware characteristics.

6 Conclusions

The experiments, which we conducted with the MXKERNEL prototype, have shown that there are benchmarks and sample applications for which we can improve the state-of-the-art in two ways: First, we gain *better performance* by exploiting the task-based execution model, which gives us much information about memory access patterns, synchronization constraints, and inherent parallelism of data management operations *in advance*, i.e. without the need for vague predictions. Second, the design and implementation of application, database, and operating system components is *simplified*, because the underlying MXKERNEL already provides the necessary abstractions and strategies to fully make use of a modern, i.e. parallel and heterogeneous, computing platform. We thus believe that we are on the right track. The proposed techniques actually work and further exciting discoveries are to be expected. Based on these insights and the MXKERNEL prototype, which is a solid platform for further experiments, we can now dig deeper and prove our claims for other and especially for more complex application scenarios.

Acknowledgements The work on this paper has been supported by Deutsche Forschungsgemeinschaft (DFG) under grant no. SP 968/9-1 and TE 1117/2-1.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Balkesen C, Teubner J, Alonso G, Özsu MT (2015) Main-memory hash joins on modern processor architectures. IEEE Trans Knowl Data Eng 27(7):1754–1766
2. Baumann A, Barham P, Dagand PE, Harris T, Isaacs R, Peter S, Roscoe T, Schüpbach A, Singhania A (2009) The multikernel: A new os architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. SOSP, vol 09. ACM, New York, NY, USA, pp 29–44 https://doi.org/10.1145/1629575.1629579
3. Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. J ACM 46(5):720–748. https://doi.org/10.1145/324133.324234
4. Borghorst H, Spinczyk O (2019) Operating systems for many-core systems, Institution of Engineering and Technology. IET Professional Applications of Computing. https://digital-library.theiet.org/content/books/pc/pbpc022e. Accessed 08/14/2020
5. Borghorst H, Müller M, Spinczyk O (2019) More or less? A discussion about the abstraction level of future operating systems. In: Proceedings of the 1st International Workshop on Next-Generation Operating Systems for Cyber-Physical Systems. NGOSCPS, vol 2019
6. Boyd-Wickizer S, Kaashoek MF, Morris R, Zeldovich N (2012) Non-scalable locks are dangerous. In: Proceedings of the Linux Symposium, pp 119–130
7. Braginsky A, Petrank E (2012) A lock-free b+tree. In: Proceedings of the 24th ACM Symposium on Parallelism in algorithms and datastructures, vol 12. Association for Computing Machinery, New York, NY, USA, pp 58–67 https://doi.org/10.1145/2312005.2312016
8. Cha SK, Hwang S, Kim K, Kwon K (2001) Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In: Proceedings of the 27th International Conference on Very Large Databases (VLDB). Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, pp 181–190
9. development cooperation A (2015) AUTOSAR 4.2.2 – 043 – general requirements on basic software modules. AUTOSAR development cooperation, Munich, Germany
10. Giceva J, Zellweger G, Alonso G, Rosco T (2016) Customized os support for data-processing. In: Proceedings of the 12th International Workshop on Data Management on New Hardware. DaMoN, vol 16. ACM, New York, NY, USA, pp 1–2 https://doi.org/10.1145/2933349.2933351
11. Golbert S (2019) Lockless parallel skiplists on the mxkernel runtime system. Master thesis, Technische Universität Dortmund
12. Hilkens K (2017) Lightweight Object Threads with Meta Data. Masterarbeit, Technische Universität Dortmund
13. Höttger R, Igel B, Spinczyk O (2017) On reducing busy waiting in AUTOSAR via task-release-delta-based runnable reordering. In: Proceedings of the 2017 Conference on Design, Automation & Test in Europe. DATE, vol 17. IEEE, Lausanne, Switzerland
14. Jiang P, Agrawal G (2017) Efficient simd and mimd parallelization of hash-based aggregation by conflict mitigation. In: Proceedings of the International Conference on Supercomputing, pp 1–11. ACM, Chicago, IL, USA
15. Kim C, Sedlar E, Chhugani J, Kaldewey T, Nguyen AD, Blas AD, Lee VW, Satish N, Dubey P (2009) Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. Proceedings VLDB Endowment 2(2):1378–1389

16. Kissinger T, Kiefer T, Schlegel B, Habich D, Molka D, Lehner W (2014) ERIS: A numa-aware in-memory storage engine for analytical workload. In: Bordawekar R, Lahiri T, Gedik B, Lang CA (eds) International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, vol 2014, pp 74–85 (http://www.adms-conf.org/2014/adms14_kissinger.pdf). IEEE, Hangzhou, China
17. Kühn R (2019) Aggregation auf Manycore-Architekturen. Masterarbeit, Technische Universität Dortmund
18. Lang H, Passing L, Kipf A, Boncz P, Neumann T, Kemper A (2019) Make the most out of your simd investments: counter control flow divergence in compiled query pipelines. Vldb J :1–18. https://doi.org/10.1007/s00778-019-00547-y
19. Lehman PL, Yao SB (1981) Efficient locking for concurrent operations on b-trees. ACM Trans Database Syst 6(4):650–670. https://doi.org/10.1145/319628.319663
20. Leis V, Haubenschild M, Neumann T (2019) Optimistic lock coupling: a scalable and efficient general-purpose synchronization method. Bull IEEE Comput Soc Tech Comm Data Eng 42:73–84
21. Levandoski JJ, Lomet DB, Sengupta S (2013) The bw-tree: A b-tree for new hardware platforms. In: Proceedings of the 29th International Conference on Data Engineering(ICDE). IEEE, Brisbane, Australia, pp 302–313
22. Lochmann A, Schirmeier H, Borghorst H, Spinczyk O (2019) LockDoc: trace-based analysis of locking in the Linux kernel. In: Proceedings of the 14th ACM SIGOPS/EuroSys European Conference on Computer Systems. EuroSys, vol 19. ACM Press, New York, NY, USA https://doi.org/10.1145/3302424.3303948
23. Mühlig J (2018) B-link-trees for DB/OS Co-Design. In: Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, vol 18, pp 59–64. CEUR-WS.org. Wuppertal, Germany
24. Müller M, Spinczyk O (2019) Mxkernel: rethinking operating system architecture for many-core hardware. In: 9th Workshop on Systems for Multi-core and Heterogenous Architectures. Dresden, Germany
25. Müller M, Leich T, Pionteck T, Saake G, Teubner J, Spinczyk O (2020) He..ro DB: a concept for parallel data processing on heterogeneous hardware. In: Proceedings of the 33th International Conference on Architecture of Computing Systems. ARCS, vol 20. Springer
26. Noll S, Teubner J, May N, Böhm A (2018) Accelerating concurrent workloads with cpu cache partitioning. In: Proceedings of the 34th International Conference on Data Engineering (ICDE). IEEE, Paris, France, pp 437–448
27. Oracle Corporation (2016) Developing parallel programs – a discussion of popular models. White Paper
28. Pandis I, Johnson R, Hardavellas N, Ailamaki A (2010) Data-oriented transaction execution. Proceedings VLDB Endowment 3(1–2):928–939. https://doi.org/10.14778/1920841.1920959
29. Picker AS (2019) Placement-optimierung in task-basierten umgebungen. Bachelor thesis, Technische Universität Dortmund
30. Psaroudakis I, Scheuer T, May N, Ailamaki A (2013) Task scheduling for highly concurrent analytical and transactional main-memory workloads. In: International workshop on accelerating data management systems using modern processor and storage architectures, vol 2013, pp 36–45 (http://www.adms-conf.org/2013/psaroudakis_adms13.pdf). IEEE, Riva del Garda, Trento, Italy
31. Teubner J, Mueller R (2011) How soccer players would do stream joins. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, vol 11. ACM, Athens, Greece, pp 625–636
32. Wang Z, Pavlo A, Lim H, Leis V, Zhang H, Kaminsky M, Andersen DG (2018) Building a bw-tree takes more than just buzz words. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, vol 18. Association for Computing Machinery, New York, NY, USA, pp 473–488 https://doi.org/10.1145/3183713.3196895
33. Wentzlaff D, Agarwal A (2009) Factored operating systems (fos): The case for a scalable operating system for multicores. SIGOPS Oper Syst Rev 43(2):76–85. https://doi.org/10.1145/1531793.1531805
34. Ye Y, Ross KA, Vesdapunt N (2011) Scalable aggregation on multicore processors. In: Proceedings of the 7th International Workshop on Data Management on New Hardware. DAMON, vol 11, ACM, Athens, Greece, pp 1–9