B-link-trees for DB/OS Co-Design

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

ABSTRACT

With modern and future hardware, which is characterized by many compute units, large main memories, and heterogeneity, we have enough computational power, but software has to adapt to these hardware changes. In this paper, we present some aspects of the `MxKernel`, which is a bare metal runtime focused on mentioned hardware and Database / Operating System Co-Design. As a key function, the `MxKernel` does not use threads but a smaller unit of work as abstraction model for parallel control flows. To figure out the behavior, we implemented a prototypical data structure for indexing, whose results will be presented and discussed.

Keywords

Databases on Modern Hardware, Database/Operating System Co-Design, Database Performance

1. INTRODUCTION

While hardware changes in the order of many processor cores, large amounts of main memory and complex memory architectures, software must also adopt these changes. The purpose of increasing the number of processor units is a linear speedup improvement, which is hard to reach. One reason therefor is given by parts of a software, that cannot be parallelized e.g. because of concurrent accesses to the same shared resource, like shared memory, must be synchronized [2, 14]. Otherwise, the software could end up in undefined behavior such as system crashes or incorrect results [25]. In conclusion, this means that synchronization like locking has to be avoided both for Operating Systems (OSs) and applications running on top, including Database Management Systems (DBMSs).

In this work, we will introduce interesting aspects of the project "`MxKernel`: A Bare-Metal Runtime System for Database Operations in Heterogenous Many-Core Hardware" which is a joint project of the Embedded System Software Group and the Databases and Information Systems Group at TU Dortmund. While common OSs allow the database to run as an application on top, the `MxKernel` is a minimal layer which can be used by both OS and DBMS as an abstraction layer for the underlying hardware. This software architecture enables the reuse of data structures and algorithms e.g. indexes that are used by OSs for file system implementation [11, 18] and databases to manage the data. Furthermore, all applications running on top of the `MxKernel` will have a less abstract interface to the hardware than the most operating systems offer so far. Additional to the software architecture, one main aspect of the `MxKernel` is a control flow abstraction, that is not built on the common thread model and pledges for a more straightforward way to avoid locks.

This paper is organized as follows. In Section 2 we will describe modern hardware in more detail and discuss problematic aspects. More precise information about the concept of the `MxKernel` will be introduced in Section 3. As a first data structure using the `MxKernel`, we built a B-link-tree, which is presented and discussed in Section 4, before we summarize in Section 5. Finally, the next steps of the project respecting databases will be outlined in Section 6.

2. MODERN HARDWARE

There are several properties of current and future hardware, that we need to follow from the software's view. This Section will discuss some of these characteristics.

2.1 Growing core count

Because it is no longer feasible to increase the clock frequencies of processing units, more cores are installed on a chip instead in order to enable more power through parallelism [3]. This growing number of processor cores affects servers as well as end-user systems, such as desktop computer and mobile devices. In fact, today we already have manycore processors and in the future, these will continue to increase [8]. Because not every part of an application can be parallelized, at that point the concurrent control flows have to be to synchronized, for example updating a shared memory location or writing to the command line. While some techniques like mutexes and spinlocks ensure only one control flow to pass that critical section, smarter algorithms use atomic load, store and compare operations [7] to implement wait- or lock-free synchronization [21]. The development of these algorithms is difficult and depends on the underlying data structure.

2.2 Non-uniform Memory Access and cache coherence

With the ongoing increase of processor units, symmetric

30th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 22.05.2018 - 25.05.2018, Wuppertal, Germany.

memory architectures stopped scaling with modern manycore systems [14]. Instead, more complex memory architectures like Non-uniform memory access (NUMA) exists in modern hardware systems. A NUMA system is divided into several nodes on which the CPUs are attached directly to the node's local memory. The memory is still organized in a coherent memory space where every core can access every memory address, but the latency differs on local and remote accesses and may be higher in the latter case. Thus, the memory accesses matters and can be a reason for bad performance [6], when obtaining data from a remote NUMA node. To prevent the latency that occurs on remote memory access, the OS and other applications should prefer local data. Experiments in context of DBMSs have shown that NUMA awareness can improve performance, but this requires knowledge about the system and interfaces to allocate local memory and place threads in defined regions [16]. Furthermore, NUMA aware join algorithms like MPSM [1] and Handshake join [23] have confirmed this.

In terms of distributed memory, parallel computing and scalability, caches respectively cache coherence are also significant. While caches speed up repeated access to data, redundant data must be synchronized through complex cache hierarchies. Since locks that are touched by many cores are often implemented by using shared memory, cache coherence quickly results in solutions that are not scalable [22].

2.3 Heterogeneity

In addition to the growing number of computing units, we also see an increase in heterogeneous hardware, like different cores with specialized functions in a single machine [4]. The cores may differ in regard to energy consumption, performance characteristics and also the instruction set architecture [15]. Even peripheral devices are used more often, such as special cores based on Field-programmable gate arrays (FPGAs) and Graphics processing units (GPUs), which are for example exploited on database query processing [9].

3. MXKERNEL

The goal of the `MxKernel` is to improve the performance of manycore systems in regard to the increasing amount of data. In order to achieve this, the `MxKernel` forms the basis for applications like DBMSs and operating system services as well. Additional small units of work, we call them `MxTasks`, are used rather than threads for control flow abstraction.

This section will give an overview of the software architecture of the kernel, the `MxTasks` and how to synchronize them.

3.1 Architecture

In most cases, operating systems are the basis for applications. The OS manages and abstracts the underlying hardware, which makes it simple to deploy applications on a wide set of different hardware. For this purpose, OSs provide interfaces for the hardware and a set of services to applications running on top of it. However, this endeavor also has disadvantages, for example, the OS is the only one who knows much about running applications.

Particularly, DBMSs does need only a few of those provided services and implements his own ones. This concludes that "a small efficient operating system with only desired services" [26] would be preferred by database people. Furthermore, it is also important to consider the overall status of

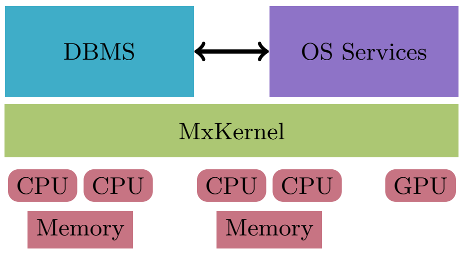


Figure 1: Software architecture of the `MxKernel`.

the system if the database is to perform well in the presence of other applications [12], but in most cases, only the OS has knowledge about the status. While existing OSs in the industry are being adapted, like Linux for Oracles Exadata Database Machine [27], some research operating systems allow specialization for applications [28, 4].

In contrast to this, the `MxKernel`, whose software architecture is shown in Figure 1, provides a platform for both OS and DBMS. This makes sense for various reasons. When the operating system is the base layer for applications, the hardware will be abstracted to minimize any effort in regard to different hardware architectures. On the other hand, the abstraction may become prevalent and applications have fewer possibilities to control hardware resources precise. While hardware changed after OSs such as Linux were published, it has become tedious to adopt these changes. For example, the entire Linux kernel was temporarily locked by a single lock to prevent multiple threads passing the kernel in parallel [5]. Another example is `libnuma` [17], a library that provides an interface for NUMA architectures but does not fit seamlessly into existing interfaces.

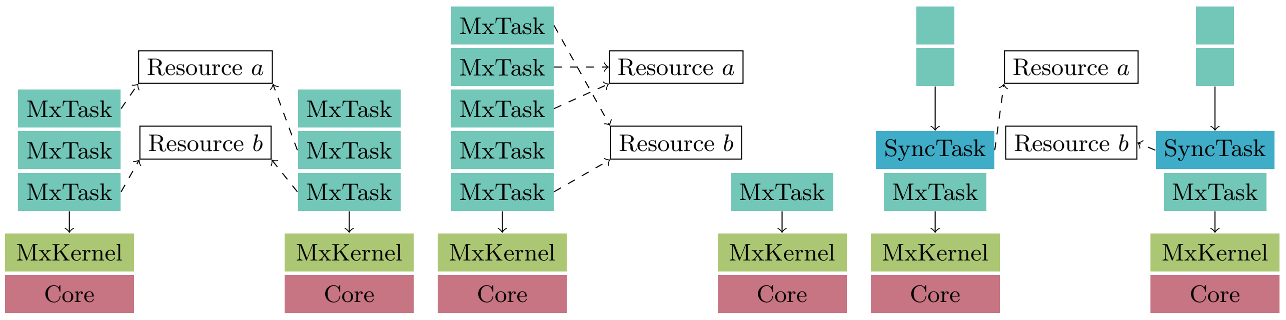
By providing a minimal layer for OSs and other applications like DBMSs, which normally run on top with a need for less abstracting interfaces of the underlying hardware, we promise advantages for both sides. For example, the DBMS could make a better placement of data on the hard disk, improve scheduling of control flows and take more care of NUMA awareness.

Further, there are components that are introduced by both OSs and DBMSs. Indexing techniques, for example, are used in databases to efficiently locate tuples. Even file systems like BeOS [11] implemented similar data structures and algorithms to access files in a quick way. Both use the same approach, but they can not share concrete implementations because of the structure, where the database is built upon the OS or vice versa. As a result, those functionalities will be held redundant, where the `MxKernel`'s software architecture allows sharing and reduces those redundancies.

3.2 Control flow abstraction

Threads are a well-known and heavily used method to abstract concurrent control flows. Many OSs, as well as programming languages, implement threads, which also have some disadvantages. When several threads access a data structure at the same time and at least one of them updates the data, the accesses must be atomic or synchronized, e.g. by mutexes or spinlocks. This could impair the scalability of the system because in case of synchronization only a single thread could pass the guarded section.

Another costly aspect is scheduling and the included context switches of threads. Since there are mostly more threads



(a) Unsynchronized access. (b) Synchronized by core mapping. (c) Synchronized by Synchronize-Task.

Figure 2: Concept of Synchronize-Task.

than processing units on a system, the operating system has to schedule them periodically. When a thread is suppressed by the OS for the benefit of another, the context of the replaced thread has to be saved and the context of the restored thread has to be recovered. On a Linux based system, a context switch takes micro- up to milliseconds [20].

An alternative approach to threads, which represents a large sequence of instructions, is to split the work into smaller units, named tasks. Several libraries and operating systems make use of this concept, e.g. Intel's Threading Building Blocks [19] and the AUTOSAR OS [10], which provides an option for offline scheduled tasks.

Also, the `MxKernel` uses the concept of tasks, namely `MxTasks`, to manage compute resources. The `MxTasks` are characterized by a run-to-completion semantic, which means that running tasks will never be suppressed by the kernel. Thus, a task does not need his own stack, instead, all tasks of one core can share the same stack, which minimizes the costs of a context switch between two tasks. Further, the kernel guarantees the execution of a task per core to be atomically, whereas a common thread could be interrupted at any time. As a consequence, all tasks scheduled to the same core are synchronized by definition and do not need any lock. This makes it easier to synchronize conflicting tasks and simplifies the development of lock-free data structures and algorithms.

In regard to modern hardware described in Section 2, we see another benefit that tasks can profit in contrast to heavyweight threads. Due to the usually longer life and execution time of threads, it is difficult to predict memory accesses and migrate threads to the suitable NUMA region. `MxTasks`, on the other hand, have a short duration of execution and in this way lesser memory accesses. This allows finer scheduling with respect to local memory requests.

3.3 Synchronization

Nevertheless, the access to one resource from different `MxTasks` has to be synchronized. Otherwise different tasks could update one resource on different cores at the same time like shown in Figure 2a, which causes undefined behavior. For this purpose, we present two methods.

CPU core based synchronization.

Based on the run-to-completion semantic of tasks, we can ensure that all tasks executed on the same CPU core are serialized. By implication, this means when all accesses on one resource are done by tasks assigned to the same core, the resource does not need to be protected by locks or mutual exclusion. Following this, we can synchronize multiple tasks accessing the same resource by scheduling them to the same core without any overhead. This is shown in Figure 2b, where both Resource a and b are mapped to the first core in the system.

Synchronize-Task.

Using the core based synchronization with a fixed resource to core mapping may result in an unbalanced load of the system, where some cores may have a lot of work and others not. In order to avoid that balancing problem and to get rid of the static task-to-core-assignment, we implemented a special task for synchronization, called Synchronize-Task. Within a system where concurrent tasks access a shared resource like shown in Figure 2a, every Synchronize-Task represents such a shared resource e.g. a monitor. Every task, that wants to use the shared resource, for example, to print some text on the monitor, needs to enqueue to the waiting list of the Synchronize-Task, shown in Figure 2c. After that, the Synchronize-Task will register itself as ready to run to the `MxKernel`. By the time the `MxKernel` executes the Synchronize-Task as a normal `MxTask`, a set of tasks waiting in the ready list of the Synchronize-Task will be executed directly. To avoid too long execution times, the set of running tasks within a Synchronize-Task will be restricted and the Synchronize-Task will be marked as ready again if not all (sub-) tasks were executed. In this way, the Synchronize-Tasks can move around between cores to balance the load and take care of NUMA aware execution. Moreover, when multiple tasks accessing the same data are executed consecutive, their behavior will be more cache-friendly because already cached data could be used for several tasks in a direct way.

4. MICRO-BENCHMARK

The task model introduced in Section 3 differs from programming with well-known threads and looks more like an event-based and asynchronous development. To get started, we opted for a B-link-tree-based index structure, with which we have already gained some experience in our group [24].

4.1 B-link-tree

B-trees and their variations like B$^+$-trees and B-link-trees [13] are key-value stores and approved data structures for

```
Data: tree, key, value
1  $node \longleftarrow$ Root($tree$)
2  while $node$ is not leaf do
3      TakeLatch($node$)
4      $n \longleftarrow$ FindChild($node, key$)
5      ReleaseLatch($node$)
6      $node \longleftarrow n$
7  end
8  TakeLatch($node$)
9  if $node$ is not full then
10     Insert($node, key, value$)
11     ReleaseLatch($node$)
12 else
       // recursive split and insert
13 end
```

Algorithm 1: Thread based insert operation on a B-link-tree.

```
Data: node, key, value
1 if $node$ is not leaf then
2     $node \longleftarrow$ FindChild($node, key$)
3     EnqueueTask($node$, InsertTask($key, value$))
4 else
5     if $node$ is not full then
6         Insert($node, key, value$)
7     else
          // split, insert and enqueue
             propagation task to parent
8     end
9 end
```

Algorithm 2: Insertion using task abstraction.

Processor	2× Intel® Xeon® CPU E5-2690
Base clock	2.90GHz
CPU Cache	L1: 32KB / L2: 256KB / L3: 20MB
NUMA Nodes	2 (CPUs 0 – 7 / CPUs 8 – 15)

Table 1: Hardware setup for experiments.

indexing data in databases or file systems. In contrast to original B-trees, B-link-trees store values in leaf nodes only, inner nodes point the way down to child nodes using keys as fences. Additionally, and contrary to B+-trees, every node in a B-link-tree contains a high key, which indicates the highest key that node will hold, and a link to the right sibling. The latter allows sequential processing of the inner and leaf nodes. Moreover, the B-link-tree allows the split operation in consequence of a node overflow to be executed in two single steps [13], whereby only the modified node must be protected by a latch. This condition allows the B-link-tree to be implemented with the task model by considering each node of the tree as a shared resource, which can be synchronized by the methods presented above in Section 3.3.

As an example for developing with the task model in the context of the MxKernel, we will take a look at the insert operation to store a key-value pair as a record into the B-link-tree. Algorithm 1 shows the pseudo code of a simplified insert operation using threads. First, we will find a leaf in a B-link-tree by traversing through different levels of the tree, shown in lines 2 to 7. As we found the correct leaf, we insert the record composed of a key and the corresponding value, represented by lines 8 to 11. In case that the leaf node is full, it is split into two nodes, both filled with half of the key-value pairs and the separator to the new node has to be propagated up to the parent node. Because this is not important for the comparison between the two abstraction models, we neglect this step.

While navigating down to the leaf the record will be stored in, we need to take a latch on every inner node to prevent other threads from updating that node. Even for the leaf node, we have to take a latch during insertion. As a result, when frequently used nodes, such as the root node, will be accessed by multiple threads, the latch data structure will be touched by all of them. The lock contention will be a big part of the execution time [28]. Another aspect is the memory access in a NUMA system. When multiple threads on different NUMA regions touch the same node, the data will be shipped across. This can be associated with high latency. By using the task model, we want to bring the code to the data instead of moving the data to the code.

The concept of MxTasks and the way they are synchronized with each other gives us the possibility to avoid the lock contention. In opposition to threads, where one thread will traverse through the tree to find a leaf, multiple tasks are used to process node by node, shown in Algorithm 2. Every insert task will start his work on the root node of the tree. Assuming that the root node is not a leaf, the task will look up for the next child node and create a follow-up task for that node (lines 1-3). On the located child node, this steps will be repeated, until we found a leaf. When the task located the wanted leaf, the key-value pair will be inserted (line 6) and the task is done. Assuming that the node has to be split, an insert task for the pointer to the new node will be spawned at the parent node. Deviating from thread model, where the propagation of the new node would be done bottom-up recursive, the MxKernel uses tasks to propagate the key-pointer pair for the new node up to the parent.

In regard to different synchronization techniques described in Section 3.3, the EnqueueTask method (line 3) could have various implementations. When using the core, based synchronization, every node is mapped to a core based on its memory address. Enqueue in this context means to mark the task as ready for the run on the mapped core so that only this core will process all tasks accessing the node. Otherwise, when the Synchronization-Task is used, every node is seen as a shared resource and will be a MxTask which can be processed by the MxKernel. In this way, every node will execute the tasks that are accessing it, to avoid concurrent accesses to one node.

4.2 Results

All measurements are carried out on a machine using the hardware presented in Table 1. While the MxKernel is running directly on the hardware, we used an Ubuntu 17.10 with a Linux Kernel 4.13.0 − 36 to measure the thread based variant of our benchmark.

As a workload, we insert 5, 000, 000 random generated 32-bit key-value pairs into the B-link-tree using a global set of predefined values, which are “stolen” by threads and tasks. The results of our experiments can be seen in Figure 3. A first finding is that all measurements show a loss of performance when using more than eight cores. At this point, the

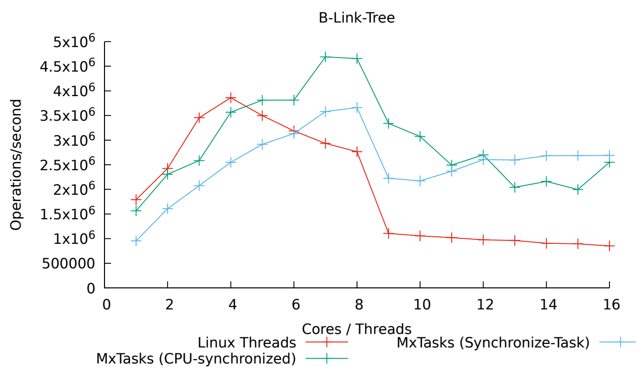


Figure 3: Results of the B-link-tree.

first processor with eight cores installed is not enough and some tasks and threads are scheduled to the second processor, which is also a separate NUMA region. While the thread benchmark (red) is fastest at four cores and slows down when additional cores are added, all variants of the benchmarks using `MxTask` gain more throughput until using the ninth core. At the thread variant, we have no influence on the scheduling of threads and left it to Linux. Therefore, it is possible that two threads on the same core compete for computing time, which may slow down the benchmark.

We also see some differences in regard to throughput within the two different techniques of task synchronization, described in Section 3.3. While synchronization by Synchronize-Task (blue) seems to be slower on the usage of the first eleven cores, the core based synchronization (green) variant loses more speed when using $12-15$ cores. In the end, when using all 16 processing units, their throughput is approximately equivalent.

By profiling the given benchmark and both techniques for task synchronization we have uncovered problems concerning the task queue data structure, which is used by both Synchronize-Task and the task management of the `MxKernel`. In order to achieve a wait-free behavior of the queue, an atomic variable is shared between consumer and producer when just one or none item is remaining in the queue. In the case of the synchronize task, where each node is represented by such a task, we get a lot of contention on that shared variable, which slows down the whole application. Unbalanced workloads in which some cores have little work end up behaving the same way. The extra effort to keep the cache coherent seems to get more expensive when more than one NUMA region is involved which ends in poor performance on two nodes.

Nonetheless, `MxTasks` seems like a promising approach to make better use of modern hardware than usual threads does. Although we have not made any optimizations regarding the NUMA architecture and do not yet schedule tasks in the best possible way, the `MxKernel` achieved a higher throughput on the B-link-tree.

5. SUMMARY

In this paper, we presented the `MxKernel`, which is a bare-metal runtime system for Database/Operating System Co-Design. With the project, we focus on modern hardware that is characterized by many cores, complex memory architectures, and heterogeneity. Small units of work are used to abstract control flows rather than threads.

As a first data structure, we implemented a B-link-tree on top of the `MxKernel` and `MxTasks`. Experiments have revealed promising results and have shown that tasks sometimes scale better than threads, even there is still room for optimizations. With the software architecture outlined above, we will obtain a better interface between software and hardware to enable optimization in regard to modern hardware.

6. NEXT STEPS

As the experiments show, we need to find a more efficient data structure for task management in order to reduce the contention of shared variables. Considering the intention to create a full runtime environment for databases and operating systems, we must first solve basic problems such as efficient memory allocation. With `MxTasks` we have a data structure that is allocated and destroyed at high frequencies, our current usage of a global heap may be a bottleneck for scalability. Similarly, there is still no clarity as to how tasks could be ideally scheduled to available cores. Therefore, we want to model dependencies among `MxTasks` and their access patterns as metadata, connected directly to the tasks. Also, offline scheduling may be an opinion.

In regard to databases, more data structures like hash tables have to be implemented in order to further explore the behavior and programming of `MxTasks`. Furthermore, we will add a transactional interface to the B-link-tree.

7. ACKNOWLEDGMENTS

This research was supported by the Deutsche Forschungsgemeinschaft, DFG, project number TE 111/2-1.

8. REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] A. Barbalace, B. Ravindran, and D. Katz. Popcorn: a replicated-kernel os based on linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

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

[5] S. P. Bhattacharya and V. Apte. A measurement study of the linux tcp/ip stack performance and scalability on smp systems. In *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, pages 1–10. IEEE, 2006.

[6] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel*

architectures and compilation techniques, pages 557–558. ACM, 2010.

[7] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[8] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[9] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1891–1906. ACM, 2016.

[10] K. Devika and R. Syama. An overview of autosar multicore operating system implementation. *International Journal of Innovative Research in Science, Engineering and Technology*, 2:3162–3169, 2013.

[11] D. Giampaolo and D. Giampaolo. *Practical File System Design*. Morgan Kaufmann Publishers, 1998.

[12] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. Cod: Database/operating system co-design. In *CIDR*, 2013.

[13] G. Graefe et al. Modern b-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011.

[14] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[15] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):186–197, 2007.

[16] T. Kiefer, B. Schlegel, and W. Lehner. Experimental evaluation of numa effects on database management systems. In *BTW*, volume 13, pages 185–204, 2013.

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

[18] P. Koruga and M. Bača. Analysis of b-tree data structure and its usage in computer forensics. In *Central European Conference on Information and Intelligent Systems*, 2010.

[19] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[20] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 2007.

[21] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30. ACM, 2002.

[22] S. Ramos and T. Hoefler. Cache line aware optimizations for ccnuma systems. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 85–88. ACM, 2015.

[23] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *Proceedings of the VLDB Endowment*, 7(9):709–720, 2014.

[24] M. Schröder. Using Modern Synchronization Mechanisms in Databases. Master's thesis, TU Dortmund, Dortmund, Germany, 2016.

[25] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[26] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[27] R. Weiss. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[28] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.