Inter-Query Parallelism on Heterogeneous Multi-Core CPUs

Experience Report

Felix Schuhknecht[1], Tamjidul Islam[2]

Abstract:

Traditional multi-core CPU architectures integrate a set of homogeneous cores, where all cores are of exactly the same type. With the release of Intel's 12th generation Core x86_64 processors, this setup has finally changed in the realm of commodity hardware: Apart from so-called performance cores, which provide a high clock frequency, hyper-threading, and large caches, the architecture also integrates so-called efficient cores, which are less performant but rather energy efficient. Obviously, such a performance-heterogeneous architecture complicates task-to-resource scheduling and should be actively considered by the application that schedules the tasks. In this experience report, we discuss our first steps with this new architecture in the context of parallel query processing. We focus on inter-query-parallelism, where whole transactions/queries are the unit of schedule, and investigate which type of core fits to which type of workload best. To do so, we first perform a set of micro-benchmarks on the cores to analyze their different performance characteristics. Based on that, we propose two scheduling strategies that actively schedule tasks to different core types, depending on their characteristics. Our initial findings suggest that the awareness of heterogeneous CPU architectures must indeed be actively incorporated by the task scheduler within a DBMS to efficiently utilize this new type of hardware.

Keywords: Query Processing; Parallelism; CPU Architectures; Heterogeneous Cores

1 Introduction

For many years, multi-core architectures used to be homogeneous in that they consist of a set of compute cores that all have exactly the same characteristics. This simplified the scheduling problem, as the choice to schedule a task to a specific core was solely determined by the current load and the locality of work to the core.

This situation drastically changed with the advent of heterogeneous multi-core architectures, a development largely driven by the so-called dark silicon effect [Ha11]. This effect describes that thermal and energetic limitations force the CPUs developers to tune down compute units, if they want to increase the overall core count. Initially, heterogeneous designs added

[1] Johannes Gutenberg University Mainz, Institute of Computer Science, Staudingerweg 9, 55128 Mainz, Germany schuhknecht@uni-mainz.de

[2] Johannes Gutenberg University Mainz, Institute of Computer Science, Staudingerweg 9, 55128 Mainz, Germany tislam@students.uni-mainz.de

doi:10.18420/BTW2023-45

specialized cores with vastly different feature sets. For instance, the Sparc M7 [Co23] combines general-purpose x86_64 cores with ASIC data analytics accelerators that support only scans, selections, and semi-joins. Such feature-heterogeneous designs require a careful rethinking of the scheduling mechanism [Du19] as it has to factor in which tasks can actually be carried out by which core. Last year, even mainstream CPUs started to implement heterogeneous multi-core architectures in form of the AlderLake architecture, which resembles Intel's 12th generation Core consumer processor. Therein, while all cores are general-purpose x86_64 chips, the architecture separates them into performance cores and efficiency cores. While the faster performance cores are meant to take over highly demanding compute tasks, the efficiency cores should save energy when executing less demanding or less performance-critical tasks. In this sense, they implement a performance-heterogeneity instead of a feature-heterogeneity, on which we will focus in the following report.

As shown in Figure 1, the two core types of an AlderLake i9-12900K highly differ in clock-speed range, cache hierarchy/cache sizes, and whether hyper-threading is available or not, potentially resulting in very different performance characteristics.

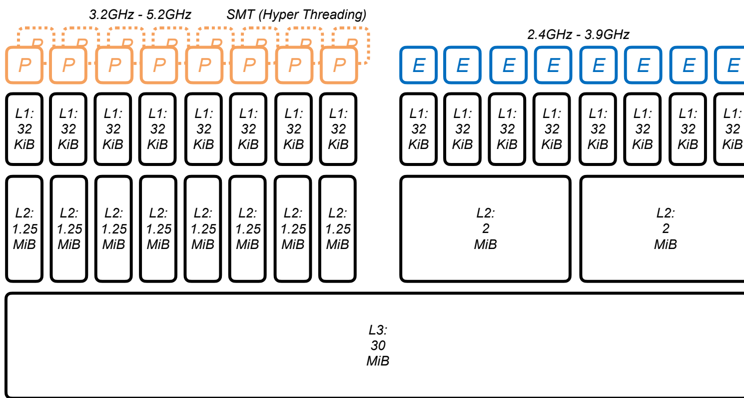
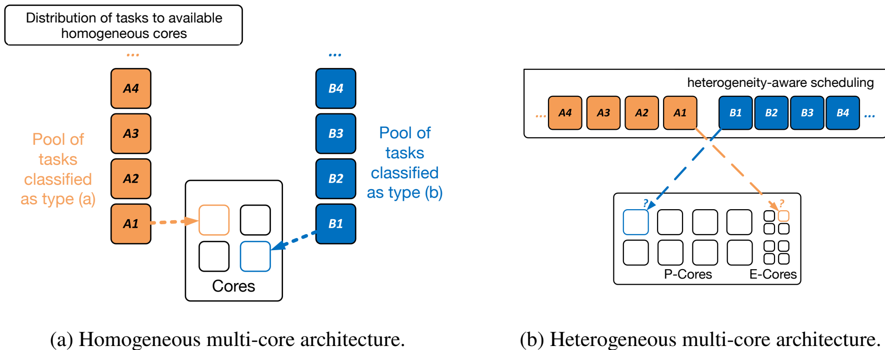


Fig. 1: A modern heterogeneous multi-core architecture (Intel AlderLake i9-12900K).

Therefore, we advocate that even though all cores implement the same instruction set, a scheduler should still be aware of the performance difference of both types of cores and assign tasks actively to the most fitting core type, as we visualize it in Figure 2. Note that operating systems such as Windows 11 and Linux (since kernel 5.18) recently became aware of the heterogeneous design and try to schedule tasks based on classification (a concept marketed as Thread Director [SP22]). In recent years, alternative general-purpose OS-level task scheduling mechanisms tailored to heterogeneous CPU architecture have been proposed [Fa21, SNN22, Ni22, THW02, CJ09, Cr12, AA17]. Typically, they are designed to optimize arbitrary heterogeneous architectures outside the database context and focus on limiting energy consumption, the amount of thread migration, and the runtime of a batch of tasks. We also want to point out that Intel's new architecture is not the first performance-

heterogeneous CPU: The ARM big.LITTLE, released several years ago, followed a similar principle. In this regard, [Mü14] analyzed how to schedule DBMS-pipelines efficiently to the different core types to optimize for energy-efficiency. With the concept now arriving on commodity x86_64 CPUs, we see the topic worth to be revisited.

The question remains whether multi-threaded applications such as DBMSs should additionally actively schedule threads on specific performance-heterogeneous cores based on their available domain knowledge on these new processors. As queries have vastly different characteristics, e.g., being compute-bound, bandwidth-bound, short/long-running, or read/write-heavy, it is likely that certain query types will utilize a certain type of core best.



(a) Homogeneous multi-core architecture.

(b) Heterogeneous multi-core architecture.

Fig. 2: Task scheduling on uniform architectures (2a) vs heterogeneous architectures (2b).

1.1 Contributions and Structure

Therefore, in the following, we would like to investigate the following questions, which also mark the contributions of this experience report:

(1) Is there a sufficient performance difference between performance cores and efficiency cores that would justify a manual core-type-aware scheduling within a DBMS?
(2) Which type of tasks perform well on which type of core? For which type of tasks is the core choice irrelevant?
(3) Do core-type-aware scheduling strategies (push-based vs pull-based) perform better than a completely unaware strategy?

The experience report is structured as follows: In Section 2, we first perform an initial benchmarking of the AlderLake architecture, in which we identify how different workloads map to the available cores. In Section 3, we present and evaluate two heterogeneity-aware scheduling strategies and compare them against an unaware strategy. In Section 4, we outline possible next steps in this topic and conclude with our early findings.

2 Benchmarking the Architecture

We start by performing a set of micro benchmarks on our performance-heterogeneous AlderLake CPU to identify which workload fits to which type of core. For all upcoming experiments, we use an Intel i9-12900K with 8 performance cores of up to 5.2GHz (with SMT aka hyper-threading) and 8 efficiency cores of up to 3.9GHz (without SMT). This results in a total of 16 physical and 24 logical cores. The machine is equipped with 128GB of DDR4-3200 RAM. As operating system, we use Arch Linux running kernel 5.17.5. Note that in this kernel version, the scheduler was not yet aware of the heterogeneous architecture – for our experiments, this does not matter as we manually perform all thread assignment in our code. A comparison with a heterogeneity-aware scheduler on a newer kernel is left for future work.

Figure 3 shows the results for executing four different workloads on each available logical core individually. Note that logical cores 0-15 resemble the (logical) performance cores, whereas cores 16-23 are the efficiency cores. No parallelism is happening here as only one core is active during each measurement. To get an intuition for the architecture, we perform the following set of micro-benchmarks on 300M integers in total: In Figure 3a, we schedule the sorting of an array with integers using `std::sort`. In Figure 3b, we copy randomly selected integers from one array into a second array. In Figure 3c, we copy the entire array of sequentially into another array using `memcpy`. In Figure 3d, we read the entire array sequentially to compute the sum of all integers. As we can see, the tasks have a different runtime on the individual logical cores, where some tasks are highly affected by the type of core while other tasks hardly show a performance difference at all. The sorting task, being compute heavy and containing random access clearly benefits from being executed on a performance core (core 0-15). Also, random copies perform better on performance cores, although the difference being less prominent. When looking at the sequential tasks, we interestingly hardly notice any difference between the core types anymore, as these tasks are rather bandwidth bound than compute bound.

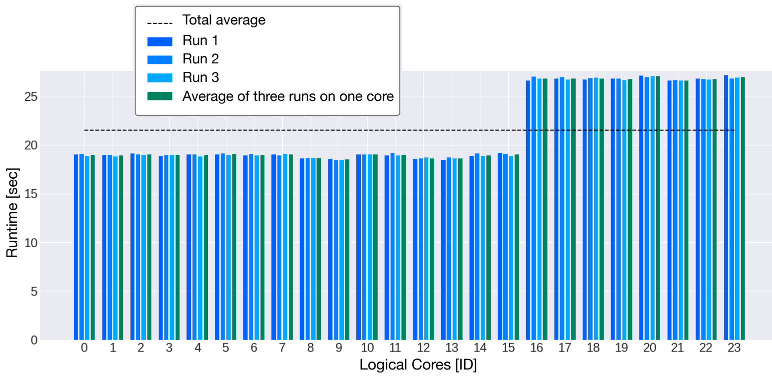
Overall, this gives us a first recommendation on how to utilize the cores: Compute-bound tasks or tasks that contain random access should preferably go to performance cores, while sequential tasks, that are mostly bandwidth-bound, could also be scheduled on efficiency cores.

3 Heterogeneity-aware Scheduling

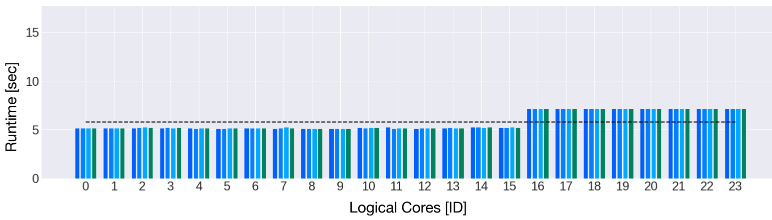
With the gained knowledge, in the following, we build and evaluate a simple scheduling mechanism for parallel query processing using two different scheduling strategies.

3.1 Setup and Task Types

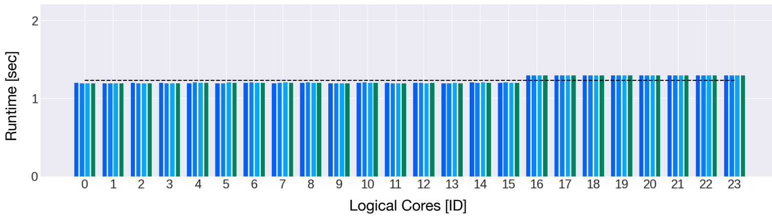
We set up a table of n integers columns in row-layout represented by a two-dimensional vector and support two types of tasks on this table: **(a) Updating transactions** that modify



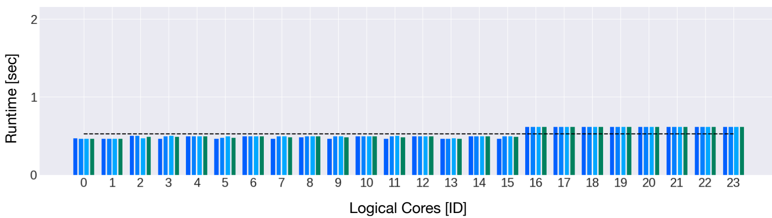
(a) Sorting (compute-heavy, random reads, random writes)



(b) Copying randomly selected integers (random reads, random writes)



(c) Copying integers sequentially (sequential reads, sequential writes)



(d) Scanning & Aggregating (sequential reads)

Fig. 3: Executing different types of workloads of each available logical core.

a certain number of rows of the table. These resemble a task from a traditional transactional workload. **(b) Read-only queries** that select a specific value of a specific column and count how often this value has been seen. These could represent a typical query from an analytical workload. When creating a test workload, we specify the ratio of tasks of type (a) in relation to tasks of type (b) and fill two pools of pending tasks of each type accordingly. Our scheduler then executes these tasks using a specific scheduling strategy.

3.2 Scheduling Strategies

We compare three different strategies in the following. The first strategy is push-based and simply ignores the heterogeneous architecture, as it uniformly distributed tasks of both types to available cores. It will serve as our baseline. Figure 4 visualizes the setup for a batch of 100 tasks. Each core maintains a pool for update transactions and read-only queries.

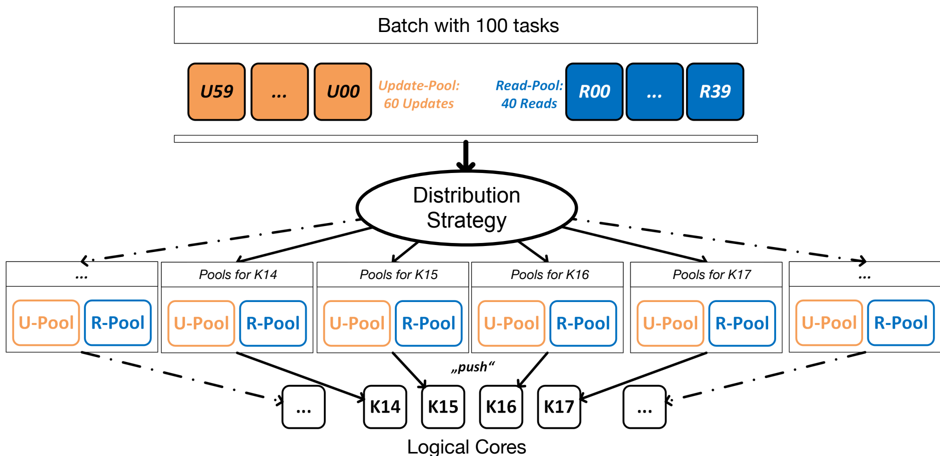


Fig. 4: Architecture of the push-based scheduler. As distribution strategy, we test both a non-aware strategy, which uniformly distributes tasks to all cores and an aware strategy, which tries to schedule update transactions to performance cores and read-only queries to efficiency cores.

The second strategy is also push-based and resembles the architecture of Figure 4, but does not uniformly distribute tasks across cores. Instead, it is aware of the heterogeneity and tries to assign update transactions to performance-cores and read-only queries to efficiency cores. Only if no core of the respective type is currently available, the task is scheduled on the other type of core. The availability of cores is recorded in status flags that are accessed by the scheduler and updated by the threads running on the cores.

The third strategy is also heterogeneity-aware, but follows a pull-based approach as visualized in Figure 5. Here, the performance cores preferably pull from the pool containing update transactions, while the efficiency cores pull from the pool containing read-only queries.

Only if no work is left in a pool, the other pool is considered. A mutex protects each pool to avoid races during the pulling of tasks.

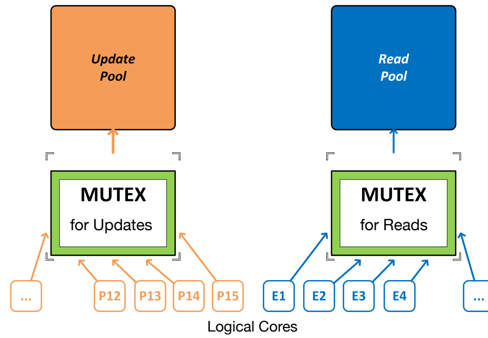


Fig. 5: Pull-based scheduler that is aware of the heterogeneous architecture.

Let us now see how the three scheduling strategies perform in comparison. We fire a batch of 180 tasks and vary the mixture between update transactions and read-only queries in steps of 25%. We use a table with 60M rows and 150 columns. Every update transaction updates 12M randomly selected rows. We observe that the strategy indeed has a significant impact on the runtime. The more update transactions we have in our batch, the more the strategy matters and the heterogeneity-aware strategies win. This makes sense, as update transactions must be actively scheduled to performance cores to yield the best runtime.

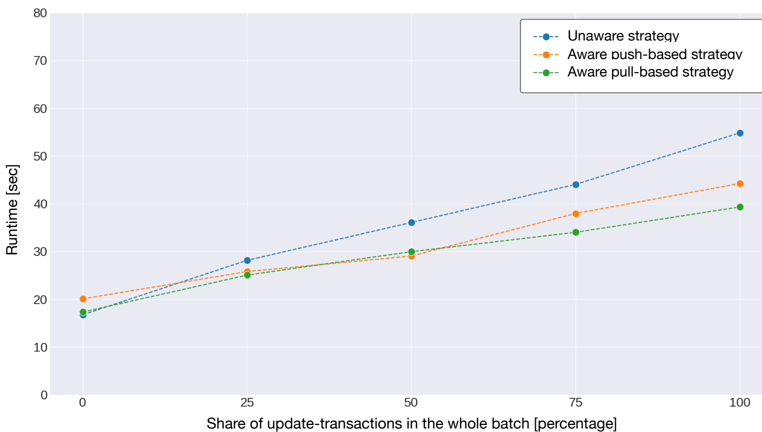
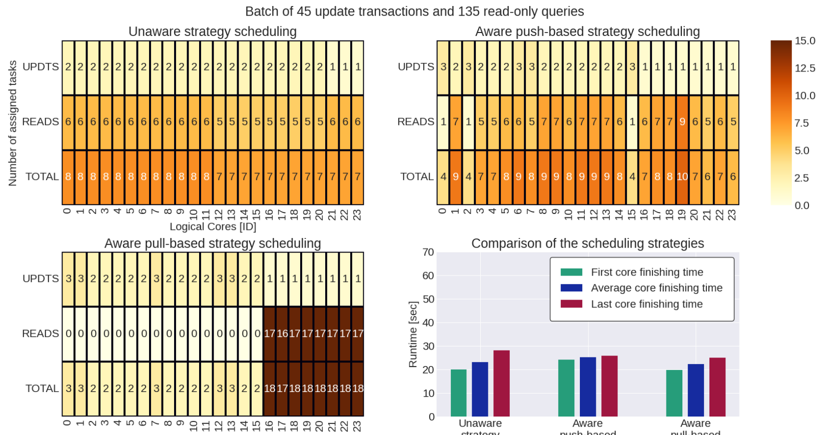


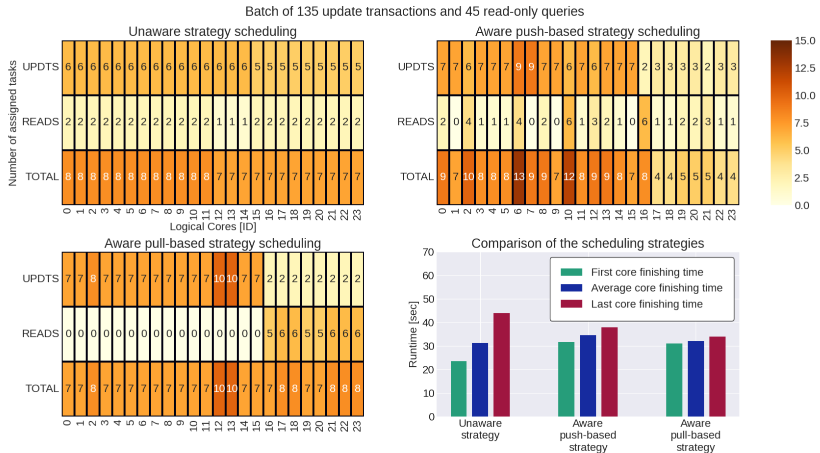
Fig. 6: Comparison of scheduling strategies under a varying mixture of update transactions and read-only queries.

To get a deeper insight in the behavior of the task scheduling, in Figure 7 we additionally plot heatmaps for the cases of 25% update transactions and 75% read-only queries (Figure 7a)

respectively 75% update transactions and 25% read-only queries (Figure 7b) that show the assignment from task (type) to logical core. Additionally to the total latency of batch, we plot the time the first task of the batch finishes as well as the average task time. We



(a) Workload Mixture: 25% update transactions, 75% read-only queries



(b) Workload Mixture: 75% update transactions, 25% read-only queries

Fig. 7: Comparison of scheduling strategies under different workloads.

observe that the strategy indeed makes a significant impact on the scheduling behavior. Both heterogeneity-aware strategies indeed try to assign update transactions to the performance cores and read-only queries to the efficiency cores. However, we also see differences between the strategies. In Figure 7a, we observe that the push-based strategy primarily distributes update transactions to performance cores, while it also assigns read-only queries to both

types of cores. The reason for this is that due to the low number of update transactions, the performance cores are soon unused and can be used to answer read-only queries as well. This behavior is different for the pull-strategy, where read-only queries are pulled only by efficiency cores in the run. In Figure 7b, we see a similar picture, however, more compute-intensive update transactions must be handled. These are indeed primarily scheduled to the performance cores by both strategies. When analyzing the whole batch, we also observe that using heterogeneity-aware strategies homogenizes the latency of individual tasks in a batch. For the naive strategy, the minimum and maximum latency differs highly, while this difference is much smaller for the aware strategies, showing that the hardware resources are efficiently utilized.

4 Outlook and Conclusion

In this experience report, we presented our initial steps in understanding the impact of a performance-heterogeneous CPU design on parallel query processing. To simplify the analysis, we focused on inter-query parallelism, i.e., we scheduled whole transactions/queries to individual cores and evaluated two strategies that try to cleverly assign fitting tasks to a specific core type. We tested two simple heterogeneity-aware scheduling strategies and showed that even on this coarse-grained level, a measurable performance boost can be observed over an unaware strategy.

Of course, this report marks only the very first step towards query parallelism on this type of hardware. As modern DBMS schedulers typically break down a transactions/query into more fine-granular compiled pipelines and schedule these individually [Le14, NF20, Ne21], a next step is to extend such a pipeline scheduler with heterogeneity-awareness. This involves on-the-fly classification of (arbitrary) pipelines, finding a suitable mapping between pipelines and core types, handling dependencies between pipelines, and ensuring a constant utilizations of all cores. Also, an important baseline for any fine-grained approach will be the new OS-level heterogeneity-aware scheduler. Here, we see a major advantage of a manual approach by being able to include domain knowledge, such as detailed transaction/query behavior, into the scheduling process. However, such a comparison is left for future work.

Bibliography

[AA17] AlEbrahim, Shaikhah; Ahmad, Imtiaz: Task scheduling for heterogeneous computing systems. J. Supercomput., 73(6):2313–2338, 2017.

[CJ09] Chen, Jian; John, Lizy Kurian: Efficient program scheduling for heterogeneous multi-core processors. In: Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009. ACM, pp. 927–930, 2009.

[Co23] Corporation, Oracle: Oracle's SPARC T7 and SPARC M7 Server Architecture: https://www.oracle.com/assets/sparc-t7-m7-server-architecture-2702877.pdf. 2023.

[Cr12] Craeynest, Kenzo Van; Jaleel, Aamer; Eeckhout, Lieven; Narváez, Paolo; Emer, Joel S.: Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In: 39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA. IEEE Computer Society, pp. 213–224, 2012.

[Du19] Dursun, Kayhan; Binnig, Carsten; Çetintemel, Ugur; Swart, Garret; Gong, Weiwei: A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. Proc. VLDB Endow., 12(12):2218–2229, 2019.

[Fa21] Fan, Zhichao; Hu, Wei; Guo, Hong; Liu, Jing; Gan, Yu: An Efficient Scheduling Algorithm for Interdependent Tasks in Heterogeneous Multi-core Systems. In: 2021 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2021, Melbourne, Australia, October 17-20, 2021. IEEE, pp. 2354–2359, 2021.

[Ha11] Hardavellas, Nikos; Ferdman, Michael; Falsafi, Babak; Ailamaki, Anastasia: Toward Dark Silicon in Servers. IEEE Micro, 31(4):6–15, 2011.

[Le14] Leis, Viktor; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In (Dyreson, Curtis E.; Li, Feifei; Özsu, M. Tamer, eds): International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. ACM, pp. 743–754, 2014.

[Mü14] Mühlbauer, Tobias; Rödiger, Wolf; Seilbeck, Robert; Kemper, Alfons; Neumann, Thomas: Heterogeneity-conscious parallel query execution: getting a better mileage while driving faster! In (Kemper, Alfons; Pandis, Ippokratis, eds): Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014. ACM, pp. 2:1–2:10, 2014.

[Ne21] Neumann, Thomas: Evolution of a Compiling Query Engine. Proc. VLDB Endow., 14(12):3207–3210, 2021.

[NF20] Neumann, Thomas; Freitag, Michael J.: Umbra: A Disk-Based System with In-Memory Performance. In: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. www.cidrdb.org, 2020.

[Ni22] Nishikawa, Hiroki; Shimada, Kana; Taniguchi, Ittetsu; Tomiyama, Hiroyuki: Simultaneous Scheduling and Core-Type Optimization for Moldable Fork-Join Tasks on Heterogeneous Multicores. IEICE Trans. Fundam. Electron. Commun. Comput. Sci., 105-A(3):540–548, 2022.

[SNN22] Salami, Bagher; Noori, Hamid; Naghibzadeh, Mahmoud: Online energy-efficient fair scheduling for heterogeneous multi-cores considering shared resource contention. J. Supercomput., 78(6):7729–7748, 2022.

[SP22] Saez, Juan Carlos; Prieto-Matías, Manuel: Evaluation of the Intel thread director technology on an Alder Lake processor. In (Serafini, Marco; Xu, Harry, eds): APSys '22: 13th ACM SIGOPS Asia-Pacific Workshop on Systems, Virtual Event, Singapore, August 23 - 24, 2022. ACM, pp. 61–67, 2022.

[THW02] Topcuoglu, Haluk; Hariri, Salim; Wu, Min-You: Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Trans. Parallel Distributed Syst., 13(3):260–274, 2002.