Bulletin of the Technical Committee on

Data Engineering

March 2019 Vol. 42 No. 1

IEEE Computer Society

Letters

Opinions

Special Issue on Cross-Layer Support for Database Management

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

Opinions and Viewpoints

Starting from this issue, the Data Engineering Bulletin will feature opinions and viewpoints from distinguished researchers in the database community and beyond. The landscape of data management has changed dramatically since the Internet, the explosion of big data, and the rise of machine learning. It is important to ponder on the state of database research, its impact on industrial practice, and new initiatives the database community should undertake. For this reason, we bring in thought leaders to help stimulate the discussions and move the field forward.

In this issue, we asked two renowned experts in the field, David Lomet and Alon Halevy, to share their thoughts about some history and future trends of data management. David comtemplates on the role of caching in the past and future. His analysis provides deep insights into why data caching systems continue to dominate the market and how to achieve higher performance that does not rely on simply increasing main memory cache size. Alon considers aspects of data that are important to decision making but are traditionally ignored by our research community. His pieces covers the management of subjective data, building unbiased presentations of data that are tailored to the subjective world-view of the recipient, and the subjective nature of human decision making.

The Current Special Issue

Philippe Bonnet put together an exciting issue on the inter-operations among applications, database systems, operating system, and hardware. The six articles in the issue illustrate various forms of cross-layer support.

Haixun Wang
WeWork Corporation

Letter from the Special Issue Editor

The layering of applications, database systems, operating system, and hardware, has never been an ideal separation. For decades, database systems have overriden operating system services in order to efficiently support transactional properties. However, this layering has enabled the independent evolution of database systems based on well-defined abstractions, at a performance cost that has remained negligible for decades.

Today, this layering is challenged. First, applications that require low latency in addition to high throughput cannot tolerate inefficiencies due to layer independance. When every micro-second counts, it makes sense to streamline the data path across layers. Second, techniques and policies devised for multicore CPUs, might have to be revised for large number of possibly diverse cores. Third, the dominance of virtualized environments and the emergence of programmable hardware requires novel abstractions.

There is a need to rethink the boundaries of applications, database systems, operating system, and hardware and revisit the nature of the interactions across these layers. In this issue, we have collected six articles that illustrate various forms of cross-layer support.

The first article proposes to share information across layers to optimize resource utilization. It is based on experimental results with SAP HANA and proposes a concrete instance of co-design between Database and Operating System. It identifies the need to bridge the semantic gap between application and database system.

The second article illustrates a mechanism, hyperupcalls, specifically designed to bridge such a semantic gap. Hyperupcalls have been designed to bridge the semantic gap between a hypervisor and a guest virtual machine (this work got the best paper award at Usenix ATC 2018). In this paper, the authors explore how hyperupcalls could be used from the perspective of applications and database systems.

Following these two articles from industrial research, the following three articles are authored by young faculty members. Each outlines an ambitious research agenda that revisits the role of database systems on the data, control, and compute planes of modern computers.

The third article revisits operating system support for data management on modern hardware. It argues for new forms of interface between database system and operating system. The fourth article focuses on specialized hardware and the opportunities this creates for database system design. The fifth article details the challenge of utilizing server cores efficiently for data intensive tasks, especially on servers equipped with many/diverse cores, and makes the case for cross-layer support for data-intensive task scheduling.

The sixth article focuses on a general purpose synchronization method, optimistic lock coupling, that is both simple and scales to a large number of cores. This method is applicable to most tree-like data structures and should be considered as an operating system-level facility on multi-core CPUs.

The traditional layering has favoured contributions from independent communities. The cross-layer approaches outlined in this issue, favour collaborations between database experts and experts on hardware, operating systems, compilers, and programming languages. We anticipate that the data engineering community will increasingly engage with these communities in the near future.

Philippe Bonnet
IT University of Copenhagen

Data Caching Systems Win the Cost/Performance Game

David Lomet
Microsoft Research, Redmond, WA 98052

1 Introduction

1.1 Cost/Performance

Data in traditional data caching record management systems resides on secondary storage, and is read into main memory only when operated on. This limits system performance. Main memory data stores with data always in main memory are faster. But this performance comes at an increased cost. The analysis in [7] shows how modern data caching systems can produce better cost/performance. Their exploition of a storage hierarchy hence can serve a greater diversity of data management needs at lower cost.

1.2 A Little History

Traditional data management systems were implemented using hard disk drives (HDD) coupled with small main memories. Such systems were, of necessity, designed as data caching systems. That is, data lived on HDDs, and was read into a main memory cache to be operated on. As main memories became larger and costs fell, more and more data was cached. Removing I/O cost from the path to data improved performance substantially. It exposed, however, new performance bottlenecks. For example, concurrency control and recovery (CCR) has modest execution cost compared with data I/O accesses. With the I/O cost reduced, CCR became a much larger part of operation execution path.

Database researchers, in striving for great performance, looked again at CCR. Ultimately, high performance CCR techniques were developed that depended, for effectiveness, upon there being no I/O. High latency within transactions did not fit well with these techniques. This led to main memory only data management systems, e.g., [1, 3, 10], with performance of millions of operations/sec. Main memory systems inspired an explosion of new CCR and data access techniques suitable for such systems. However, the performance gains required permanently committing main memory to the data being managed.

The main memory efforts have, however, led to technology that made it possible to achieve much higher performance in data caching systems. For example, both RocksDB [8] and Deuteronomy [4, 5, 6] use main memory techniques, e.g. latch-free data access, for high performance on data cached in main memory. In addition, both dramatically shrink write I/O cost via log structuring techniques, and avoid some read I/O by supporting "blind" updates without the pages needing to be in main memory.

2 Data Management Economics

A data management system should $ALWAYS$ be able to achieve higher performance with all data in main memory. Further, the fall-off in performance when data has to first be brought into the cache is substantial, even for a highly optimized system. So why bother with a data caching system? The answer is "better cost/performance". So the argument here is not that there is insufficient main memory to hold the data, but that there is a less costly way to manage data.

We regard data caching system operations as coming in two flavors, in-memory operations MM (data is in main memory) and secondary storage operations SS (data needs to be brought into main memory). Data management operations have both execution and storage costs. Storage costs are always paid, while execution costs are incurred only when data is operated on. MM operations have higher performance (no secondary storage data access) while SS operations have lower storage costs (flash is cheaper than DRAM).

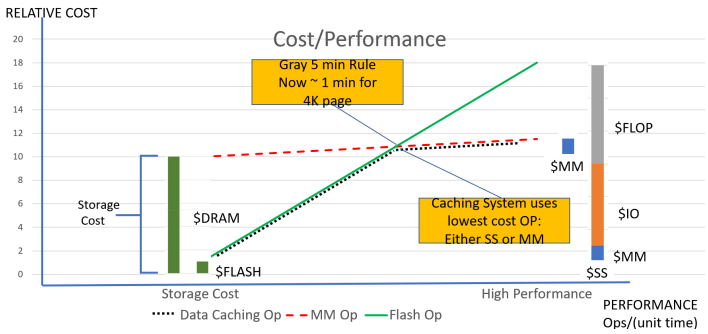


Figure 1: The cost/sec of main memory operations and secondary storage operations as execution rates change.

2.1 Data Management Cost/Sec

We analyze cost per second of data management for our two operations: MM and SS [1] The costs are of two forms, storage rental costs for data and cpu and device rental costs for operation execution.

MM **operations:** We need to rent a page of DRAM ($DRAM) and a page of flash ($FLASH) because the data is also on secondary storage. MM operation execution cost is $MOP.

$$\$MM = (\$DRAM + \$FLASH) + N * \$MOP \tag{1}$$

SS **operations:** We need to rent a page of flash ($FLASH) but not a page of DRAM since we are bringing the data into DRAM only as needed. So the cost in DRAM is insignificantly small. The cost of an SS operation involves its storage cost, now flash memory, with cost $FLASH and its execution cost $SOP. $SOP consists of MM operation execution cost $MOP plus the cost of bringing the data into DRAM. This second cost is $FLOP, the cost of an SSD's IO operation, plus $IO, the cost of the cpu executing its IO path .

$$\$SS = \$FLASH + N * (\$MOP + \$FLOP + \$IO) \tag{2}$$

We can now calculate the breakeven point- i.e., how many op/sec N we need to execute for the costs to be equal. For lowest cost, we switch between MM and SS operations at that point. We solve for $1/N$, the interval between accesses when costs are equal.

$$T_i = \frac{1}{N} = \frac{\$SOP - \$MOP}{\$DRAM} = \frac{\$FLOP + \$IO}{\$DRAM} \tag{3}$$

T_i yields our version of the "5-minute" rule. T_i is approximately one minute (the updated "5 minute rule" [2]). The smaller the breakeven time, the sooner the cost is reduced by evicting the page. A data caching system can use the breakeven point in choosing the lower cost operation. It approximates this point with its caching policy. The cost trade-off between MM and SS operations is shown in Figure 1. Capturing operation costs permits us to compare costs at performance points away from breakeven. This demonstrates why flash cost $FLASH is so important.

At low access rates, the cost of storage dominates the execution cost. Since main memory is $10X$ more costly than flash, an MM operation is $11X$ more expensive than an SS operation at very low access rates. At high access rates, it is the execution cost that dominates. $SOP is about $12X$ the cost of $MOP, with $FLOP approximately $6X$ of $MOP, and $IO about $5X$ of $MOP. The number of IOPS provide by an SSD has dramatically increased, reducing $FLOP. Thus CPU cost ($IO) for an I/O is now a significant part of $SOP cost.

Trading space (cost) for time (cost) can be useful for improving systems, but how much space for how much performance? We analysed other technologies in [7] which showed similar cross-over points frequently

[1] A more detailed analysis is in [7].

exist where relative cost advantage also changes between systems. Using more main memory, as done by the main memory MassTree key-value store results in its performance being higher than Deuteronomy's. In our experiments, the breakeven point for a page at about $T_i = 3$ seconds. Hence MassTree is better for very hot data, but not for data accessed less frequently. Data compression saves storage cost at the expense of increasing execution cost, hence good for very cold data, even though decompression adds to execution cost.

3 Concluding Thoughts

Cost/performance is usually more important than sheer performance. And storage costs are a very big part of overall costs, since most data is cold. Further, the hot data set typically changes over time. Cost effective data management means reducing storage costs for cold data and reducing execution cost for hot data. That is what data caching systems do. Data caching systems dominate the market via better cost/performance, despite main memory systems having higher performance.

There is another message here. Research may dramatically increase data caching system performance. For example, reducing the cost of moving data between storage hierarchy levels could be a huge win, enabling new and cheaper storage technologies to play an important role in data management. That research agenda might succeed in producing a "one size" system that fits (almost) all [9] of the data management market.

4 Acknowledgements

I want to thank Haixun Wang for inviting me to write this opinion piece. Thanks also to my Deuteronomy colleagues, particularly Justin Levandoski, Sudipta Sengupta, and Ryan Stutsman. Together, we built a great data caching system.

References

[1] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, M. Zwilling: Hekaton: SQL server's memory-optimized OLTP engine. SIGMOD 2013: 1243-1254

[2] J. Gray, G. R. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. SIGMOD 1987: 395-398

[3] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mülhe, T. Mühhlbauer, W. Rödiger: Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. IEEE Data Eng. Bull. 36(2): 41-47 (2013)

[4] J. Levandoski, D. Lomet, and S. Sengupta, The Bw-Tree: A B-tree for New Hardware Platforms, ICDE 2013, pp. 302–313.

[5] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. PVLDB 6(10): 877-888 (2013).

[6] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, R. Wang: High Performance Transactions in Deuteronomy. CIDR 2015.

[7] D. Lomet: Cost/performance in modern data stores: how data caching systems succeed. DaMoN 2018: 9:1-9:10

[8] RocksDB: A persistent key-value store for fast storage environments. `http://rocksdb.org/`

[9] M. Stonebraker, U. Cetintemel, "One size fits all": an idea whose time has come and gone. ICDE 2005.

[10] M. Stonebraker, A. Weisberg: The VoltDB Main Memory DBMS. IEEE Data Eng. Bull. 36(2): 21-27 (2013)

The Ubiquity of Subjectivity

Alon Halevy

1 Introduction

Data has become an integral part of our lives. We use data to make a wide spectrum of decisions that affect our well-being, such as what to wear today, where to go for dinner and on vacation, which charity to support, and who to vote for in the elections. Ideally, we'd like to think that if we just had all the facts, decisions would be easy and disagreements would be quickly settled. As a research community, our goal has been to fuel such decision making by focusing on extracting, managing, querying and visualizing data and facts. I argue here that we need to acknowledge the subjective aspects of data and decision making and broaden our agenda to incorporate them into the systems we build.

Subjectivity is prevalent in at least three levels. First, the data itself may be subjective: there is no ground truth about whether a restaurant is romantic or a travel destination is relaxing. We need to develop techniques to extract, manage and query subjective data. Second, presentation of the data can be subjective either by introducing bias (perhaps intentionally or even maliciously), or by tailoring the presentation to the frame of mind of the recipient. Third, human decision making is inherently subjective and based on emotions. We need to understand how to model the emotional aspect of decision making in our systems.

The following sections will expand on each of these topics. I have already done some research on the first topic [5] and so my comments on it will be more concrete. However, I believe all three areas are equally important.

2 Subjective data

When we compare between multiple options (e.g., hotels, restaurants, online courses), we typically lean on subjective data that describes the *experiential aspects* of these options. Today, this data resides in online reviews. For example, reviews will give insights on whether a restaurant has a quiet ambience, a hotel has helpful staff, or a school has good faculty. While today's online shopping sites make significant attempts at surfacing key quotes from reviews and even classifying them into several categories, a user cannot express experiential conditions in the query (e.g., *hotels up to $250 a night with helpful staff*). Hence, we end up sifting through many reviews until we get exhausted and settle for sub-optimal choices.

Database systems traditionally focus on the *objective* attributes of entities (e.g., the price and location of a hotel, or the cuisine of a restaurant). One exception, which captures a narrow aspect of subjective data, is modeling reviewers' opinions as star ratings, and letting users query the aggregate ratings. Subjective data introduces additional technical challenges because it is expressed in free text and is very nuanced. Hence, in addition to the inherent subjectivity of the data, users may express similar opinions with very different words. The Natural Language Processing (NLP) community has spent significant effort on subjective data. That community investigated problems such as identifying subjective data [14], studying how subjectivity affects the meaning of words [12], learning extraction patterns for it [13], and creating summaries of online reviews [6]. Management of subjective data presents an interesting opportunity to combine techniques from databases and NLP.

The first challenge involving subjective data is answering complex queries. Roughly speaking, from a query processing perspective, the NLP community has addressed the problem of retrieving individual facts (e.g., finding comments about the friendliness of the hotel staff). However, to answer more complex queries, a system needs to be able to combine multiple query predicates and to aggregate opinions across many reviews. Combining multiple subjective predicates (e.g., *hotels with quiet rooms and friendly staff*) is similar in spirit to the problem of combining multiple sources of fuzzy information [2]. Aggregating subjective data is crucial because users

typically want to get an overview of the reviews at hand. The problem of aggregating reviews is fascinating because it requires mapping textual expressions to some meaningful scale. Of course, we may also want to restrict whose opinions we consider in answering the query. For example, I may want to only consider reviews of people *like me*, or people who are experienced travelers.

The second challenge regarding subjective data is that the schema is much more fluid and subjective. Since experiential aspects of an entity can be extremely varied and nuanced, it's not clear which attributes to include in the schema and what their precise meaning is. Some attributes may be related to each other or mostly determined by others (e.g., FRIENDLYSTAFF and HELPFULSTAFF). Hence, a subjective database system cannot assume that every query it receives can be answered with its schema. In some cases, it will need to find attributes in the schema that are closest to the query terms. In other cases, when the system has low confidence that the schema can be used to answer the query, it will need to fall back on the actual text in the reviews. In some ways, this challenge is similar to how web search engines straddle the boundary between structured and unstructured data. When a search engine can confidently map a query to an entity in their knowledge graph, the engine presents a knowledge card with structured data. When the engine thinks the answer appears in a paragraph of text or an HTML table on the Web, they present that result prominently. But when both of these fail, they fall back on links to ranked Web pages.

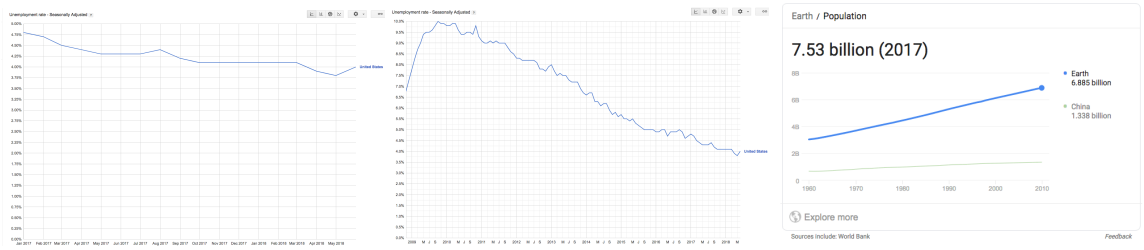
Finally, at the core of the subjective evaluation of data is also the subjective view of the user asking the query. A hotel that would be considered clean for one person may not cut it for another, and obviously opinions differ on cafes. Hence, when user profiles are available, a subjective database should tailor its answers to the user. User profiles may be collected passively from previous queries, clicks or purchases, or they may be constructed by answering a few survey questions. The next two sections dive deeper into the topic of the user perception.

3 Subjective presentation

To be useful, data needs to be communicated to a user either textually, orally or visually. There are two dimensions to subjectivity in the presentation: *faithfulness*–whether the data is presented without bias, and *effectiveness*–whether the data is presented in a way that is likely to resonate with the frame of mind of the recipient.

In terms of faithfulness, there are several common methods to create a misleading presentation, including omission, alluding to the *straight-line instinct*, and improper comparison [8]. An example of omission is in Figure 1(a) showing that unemployment rates in the US have been going down since Donald Trump took office in January, 2017, whereas Figure 1(b) shows that it is merely a continuation of the trend that began during the previous administration. An example of alluding to the straight-line instinct would be to show the world population growth graph in Figure 1(c) and letting the readers extrapolate that it will grow indefinitely. In fact, the common estimates show that the world population will actually peak around 2050. An example of improper comparison would be to state that the GDP of the USA grew by 2.3% in 2017 but without presenting growth numbers for comparable economies. Note that in some cases search engines already show comparable points when presenting structured data.

In the above examples the presenter seems at least slightly nefarious by hiding relevant aspects of the data (see [9] for a more detailed discussion of fairness in data analysis). But subjectivity in presentation can also be introduced in order to frame the data in a way that can resonate better with the recipient. Such subjectivity could make the difference between making a convincing case or falling on flat ears. Lakoff and Wehling discuss the framing issue in the context of political debates [4]. They discuss the example of the healthcare debate in American politics and claim that once the healthcare is framed as a product, there is little chance of convincing conservatives that it should be available to everyone and that everyone should be forced to buy healthcare. After all, a government should not force its citizens to buy *any* product, be it peanut butter, striped socks, or healthcare. In contrast, if healthcare is framed as an issue of freedom, which is central to the conservative doctrine, then



(a) (b) (c)

Figure 1: (a) is the unemployment rate in the USA since 2017 and (b) is the rate since 2009. (c) is the world population over time.

conservatives would see the merits. After all, you are not truly free if a medical condition can completely deplete all your assets.

The above discussion leads to several research problems which can be stated at a general level as follows: (1) *is P a faithful presentation of the data D?* (2) *Does a presentation P of data D support an argument A*? (3) *is the presentation P of data D effective for the user U?* Note that effectiveness is different from relevance–data can be relevant, but the user may still ignore it if not presented effectively. In a particular context, these problems will be made more specific. For example, we will have some space of possible presentations (e.g., graphs spanning different time periods), and for elements of that space we can consider specific questions, such as would a presentation of a graph of a variable over time incorrectly lead to inducing wrong conclusions with the straight line instinct? Could one falsely conclude a particular pattern without looking at a broader time scale? What are appropriate comparison points for a particular datum?

4 Subjective data use

We would like to think that our decision making is rational and based on hard facts. In practice, however, it is well known from psychology and Neuroscience that our emotions, which express many of our subjective preferences, play a large part in our decision making [3]. In fact, it has been shown that *without* emotions we cannot make even simple decisions. A famous case in point was Antonio Damasio's patient named Elliot [1]. Elliot suffered a particular type of damage to his brain following the removal of a tumor and was unable to feel any emotion, even when he was shown very disturbing images. While Elliot's IQ remained in tact, he was not able to make simple decisions such as how to prioritize work items or choose an item from a restaurant menu.

I am not suggesting that the intricacies of decision making can be reformulated as a data management problem, but I think we can do a much better job at incorporating the emotional aspect of decision making into the systems we build.[2] From a computational perspective, decision making involves exploring a large state space of possible outcomes, such as choosing a hotel to stay during a trip, or the best method for securing child care for your baby. The first challenge to decision making arises because the number of possible states may be too large to consider, especially under time and attention pressure. Second, we may have incomplete or only probabilistic knowledge about these possible states, making the comparison even sketchier. Finally, many of the choices we make in life (e.g., between job offers or romantic prospects) are not really directly comparable to each other.

To reach decisions effectively, we prune the search space using heuristics that may not be conscious [10]. For example, we may proceed by specifying conditions on aspects of the problem (i.e., conditions on schema attributes) that exclude many options (including some good ones!) until we have a small enough set of choices to consider in detail [11]. Developing an understanding of how to use this and other heuristics effectively while and

[2]Rosalind Picard already pointed in this direction in 1997 in her original book on Affective Computing [7], (page 220).

still guaranteeing quality decisions presents an exciting area of research that our current data science toolkit set is well poised to investigate.

5 Conclusion

To realize the full potential of the vast amounts of data available to us today, systems need to be able to manage subjective data and to understand how prospective consumers of the data think and make decisions subjectively. I've tried to outline a few concrete steps on this very broad research agenda. Of course, as we tackle these problems, we should also keep in mind that ultimately the data we have is merely an abstraction of the world and there will be other factors that are not included in the data that will influence our decisions and actions. In a nutshell, this is a call to consider concepts from psychology, behavioral economics and neuroscience in the design of tools that enable decision making based on data.

6 Acknowledgements

I'd like to thank my colleagues at Megagon Labs for inspiring many of the ideas described above: Wang-Chiew Tan, Vivian Li, Adi Zief-Balteriski, Yuliang Li and Jingfeng Li. Thanks to Phil Bernstein, Anna-Lisa Gentile, Rada Mihalcea, Haixun Wang and Dan Weld for several discussions relating to the topics covered and to Haixun for encouraging me to write this article.

References

[1] A. Damasio. *Descartes' Error*. Springer, 1994.

[2] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226. ACM, 1996.

[3] D. Kahnnman. *Thinking Fast and Slow*. Penguin Books, 2012.

[4] G. Lakoff and E. Wehling. *The Little Blue Book: The Essential Guide to Thinking and Talking Democratic*. Free Press, 2012.

[5] Y. Li, A. X. Feng, J. Li, S. Mumick, A. Halevy, V. Li, and W.-C. Tan. Subjective databases. *arXiv preprint https://arxiv.org/abs/1902.09661*, 2019.

[6] B. Liu. *Sentiment Analysis and Opinion Mining*. Morgan & Claypool, 2012.

[7] R. Picard. *Affective Computing*. The M.I.T Press, 1995.

[8] H. Rosling, O. Rosling, and A. R. Rönnlund. *Factfulness: Ten Reasons We're Wrong About the World–and Why Things Are Better Than You Think*. Flatiron Books, 2018.

[9] J. Stoyanovich, S. Abiteboul, and G. Miklau. Data responsibly: Fairness, neutrality and transparency in data analysis. In *Proceedings of EDBT*, pages 718–719, 2016.

[10] R. H. Thaler and C. R. Sunstein. *Nudge: Improving Decisions About Health, Wealth, and Happiness*. Penguin Books, 2009.

[11] A. Tversky. Elimination by aspects: A theory of choice. *Psychological Review*, 79(4):281–299, 1972.

[12] J. Wiebe and R. Mihalcea. Word sense and subjectivity. In *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, 2006.

[13] J. Wiebe and E. Riloff. Finding mutual benefit between subjectivity analysis and information extraction. *IEEE Trans. Affective Computing*, 2(4):175–191, 2011.

[14] J. Wiebe, T. Wilson, R. F. Bruce, M. Bell, and M. Martin. Learning subjective language. *Computational Linguistics*, 30(3):277–308, 2004.

From the Application to the CPU: Holistic Resource Management for Modern Database Management Systems

Stefan Noll [#+*], Norman May [#], Alexander Böhm [#], Jan Mühlig [+], Jens Teubner [+*]

[#] *SAP SE, Germany*

{stefan.noll,norman.may,alexander.boehm}@sap.com

[+] *Databases and Information Systems Group, TU Dortmund University, Germany*

jan.muehlig@tu-dortmund.de

jens.teubner@cs.tu-dortmund.de

[*] *Informatik Centrum Dortmund e.V., Germany*

Abstract

With their capability to perform both high-speed transactional processing and complex analytical workloads on the same dataset and at the same time, Operational Analytics Database Management Systems give enormous flexibility to application developers. Particularly, they allow for the development of new classes of enterprise applications by giving analytical insights into operational data sets in real time. From a database system point of view though, these applications are very demanding, as they exhibit a highly diverse combination of different query workloads with inhomogeneous performance and latency requirements. In this article, we discuss the practical implications and challenges for database architects and system designers. We propose solutions that—by sharing semantics between the application, the database system, the operating system, and the hardware—allow to manage complex and resource-intensive workloads in an efficient and holistic way.

1 Introduction

Driven by the idea of enabling both transactional processing as well as real-time analytical query workloads in the context of a single system [1], a new class of database management systems has emerged, referred to as operational analytics database management systems [2]. These systems envision to simplify the data management landscape by consolidating multiple, disparate use cases. Consequently, they allow the creation of novel business applications which seamlessly combine both real-time analytics and transaction processing [3]. From a database system designer's point of view, however, delivering high performance for these modern applications puts up additional challenges because their workload characteristics are highly diverse and demanding.

In particular, static workload management schemes and simple heuristics perform suboptimally considering end-to-end performance [4]. The reason for this is that even the same query can have different priorities from an application's point of view, depending on the context it is being executed in. A simple primary-key based

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

lookup operation can be issued, e.g, in the context of a *mission-critical* and time-sensitive OLTP transaction such as the *interactive* data entry by a business user, but also as part of a *non-critical* batch transaction that runs for hours in the *background*. A complex analytical query can be sent in the context of, e.g., a *scheduled* quarter-end close report (making it rather uncritical from an execution time point of view), but also as part of a *user-driven*, interactive dashboard application where virtually every millisecond counts.

Motivated by these observations, we share the opinion of other researchers that individual hard- and software-solutions throughout the data processing stack fail to address the complex challenges of these systems in isolation [5, 6]. Particularly, we believe that they are best addressed by a holistic approach, and in collaboration between the application, database management system, and the underlying hardware [7].

Outline. This article is structured as follows. In Section 2, we discuss how to bridge the semantic gap between the application and the database system. Communicating additional workload context information and priorities allows the database system to perform prioritization, scheduling, and resource management that is in line with the expectations of the application. Next, Section 3 discusses a similar technique which allows the database system to share semantic information about data access patterns with the underlying hardware (i.e., the processor) by using CPU cache partitioning. This enables the more efficient utilization of CPU caches, specifically for heterogeneous, highly concurrent workloads. In Section 4, we highlight different approaches for database management systems to interact with the operating system, e.g., to improve scaling and robustness. While commercial database systems usually choose to circumvent the OS exploiting detailed workload information, another strategy is bringing the database system and the operating system closer together with a radical co-design. We conclude the article in Section 5.

2 Workload Management in SAP HANA

Analytical and transactional workloads do not only have different resource demands, they are also linked to different performance expectations. *Transactional* workloads are characterized by operations which consume relatively little memory and compute power. However, they are usually sensitive to lock contention. Unfortunately, this resource contention leads to fluctuations in statement response times, while applications expect predictable throughput and response time characteristics. In our experience, customers are willing to sacrifice peak performance in return for better predictability of statement response times. *Analytical* workloads, on the other hand, typically are demanding regarding CPU and memory consumption. In particular, single analytical ad-hoc statements may consume excessive amounts of memory or CPU. This is particularly problematic for operational analytics DBMS when demanding analytical statements take away CPU clock cycles and slow down concurrent transactional workloads [4].

To balance or prioritize different workloads and to comply with a *service level agreement* (SLA), commercial DBMS usually employ *workload management* to manage resources available to single statements, applications or database users, see [8] for a good overview. In SAP HANA, workload management works on a fine-grained level [9] because the system is optimized for modern multi-core architectures with extensive multi-threading [10]. Limiting, e.g., the number of working threads or the amount of memory available to the engine, can be managed for entire workloads or individual statements. While this creates a powerful tuning option for the user, the system has to track and enforce resource consumption in all of its components. In the following, we briefly present two mechanisms for implementing workload management using the example of SAP HANA: *workload classes* and *admission control*.

Workload Classes. In SAP HANA *workload classes* address 1) the requirement of avoiding the excessive resource consumption of single statements and 2) the requirement of isolating the resource demands of different applications, database users, etc. Workload classes are containers of configuration parameters like priority, query

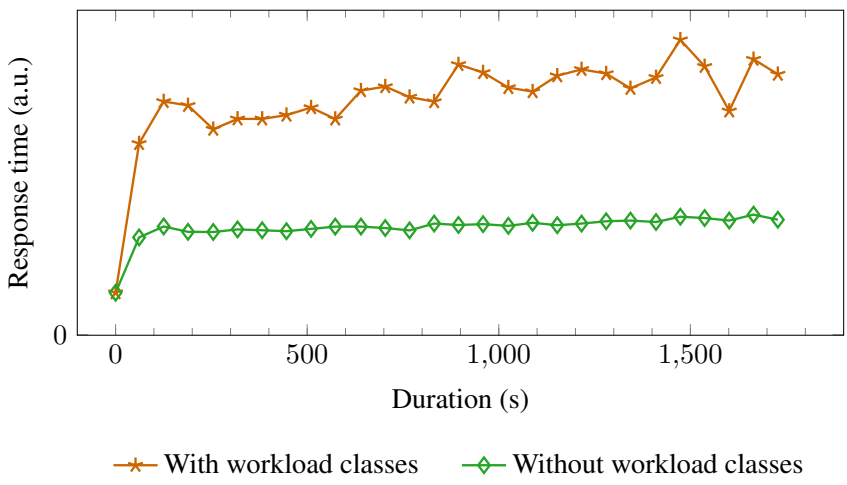


Figure 1: Average response time of web requests with and without using workload classes.

timeout, limits of memory consumption, or limits of concurrency per statement or per group of statements. They are defined in the database catalog and can be applied to all statements of a database connection. Workload classes depend on context information passed by the client application: session variables, key-value pairs maintained per database connection, e.g., `APPLICATION="ETL"`. This context information is matched to a *workload mapping* which defines the set of session variables to consider and the mapping to a workload class.

The example in Listing 1 defines a workload class `ETLWorkloadClass`. Statements executed in the context of this workload class may not consume more than 5 GB of memory and will not use more than one thread. The corresponding workload mapping `ETLWorkloadMapping` will apply this workload class if the session variable `APPLICATION="ETL"` is set in the database connection. We have confirmed in many customer scenarios that workload classes can significantly help to achieve high performance and robust system behavior.

```
CREATE WORKLOAD CLASS "ETLWorkloadClass"
   SET 'STATEMENT MEMORY LIMIT' = '5',
       'STATEMENT THREAD LIMIT' = '1';

CREATE WORKLOAD MAPPING "ETLWorkloadMapping"
       WORKLOAD CLASS "ETLWorkloadClass"
   SET 'APPLICATION NAME' = 'ETL';
```

Listing 1: Simple example of using workload classes in SAP HANA.

Experimental Results. Figure 1 illustrates how workload classes can improve response time of web requests in peak load situations. We measure the average response time per web request: Using JMeter we issue continuously 50 web requests per second. In addition, we slowly increase analytical and ETL load by adding five queries to each workload every five minutes. Eventually, we reach a very high system load with close to 100 % CPU utilization. Using workload classes, we limit the ETL and the analytical workload to one thread and 5 GB of main memory per SQL query (cf. Listing 1). Thus, the ETL and analytical load is handled using a best-effort strategy with reduced resource usage. The results demonstrate that enabling workload class management keeps the average response time of the web requests at a low, predictable value.

Admission Control. Workload classes influence the resource consumption during execution when a statement was already admitted by the database processes. However, a reasonably sized system may still experience short

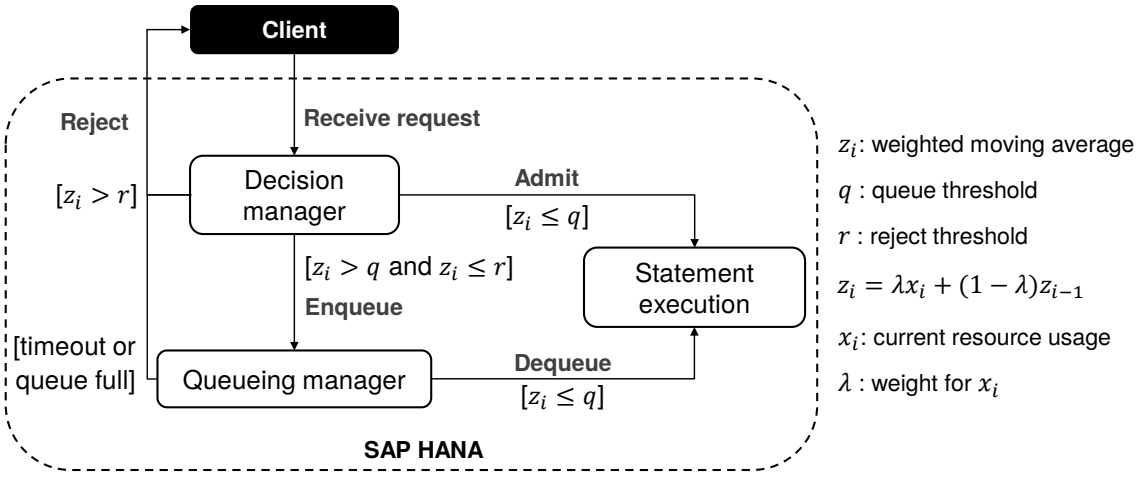


Figure 2: Request processing in SAP HANA using admission control.

peaks in CPU or memory consumption that may result in contention of memory, CPU or latches. Hence, in SAP HANA we complement workload classes with an *admission control* mechanism. A schematic overview is depicted in Figure 2.

The admission control mechanism maintains a weighted moving average z_i for all monitored resources of the host such as CPU utilization or total memory consumption. When SAP HANA receives a new database request, it checks the weighted moving average against thresholds for each resource. If each z_i is below its corresponding queuing threshold q, the statement is admitted for immediate execution. If the statement cannot be executed immediately, each z_i is checked against its corresponding reject threshold r. If the resource usage is below the reject threshold, the statement is queued; otherwise it is rejected immediately. SAP HANA periodically checks if the resource consumption has decreased below the queuing threshold. When this is the case, a batch of requests is fetched from the queue and scheduled for execution. Database requests may also be rejected when the queue exceeds a configurable size or when the request is queued for too long.

Admission control samples the current resource usage x_i every second. The value x_i is taken into account with, e.g., a weight of $\lambda = 0.7$. Note that the sampling interval in combination with the weighted moving average lessens the impact of peak load situations, while the thresholds assure that the system reaches a high load without being overloaded. In addition, admission control assumes that the client application implements a reasonable strategy to retry rejected database requests. We could verify in productive setups that admission control helps to avoid contention issues in peak load situations, and that customers profit from a more robust system behavior.

Conclusion. Different workloads have different resource demands and performance characteristics. To manage hardware resources efficiently or to comply with SLAs of various workloads, commercial systems employ *workload management*. Using SAP HANA as an example, we present two techniques. *Workload classes* limit, e.g., the amount of memory or number of threads for an entire workload or an individual statement by enforcing limits across all components of the DBMS. In addition, *admission control* manages incoming request before they are admitted into the system, monitors CPU and memory consumption, and ultimately accepts or rejects a request to improve response times in peak load scenarios.

3 CPU Cache Partitioning

Modern microprocessors feature a sophisticated hierarchy of caches to hide the latency of memory access. However, multiple cores within a processor usually share the same last-level cache (LLC). We observed that this

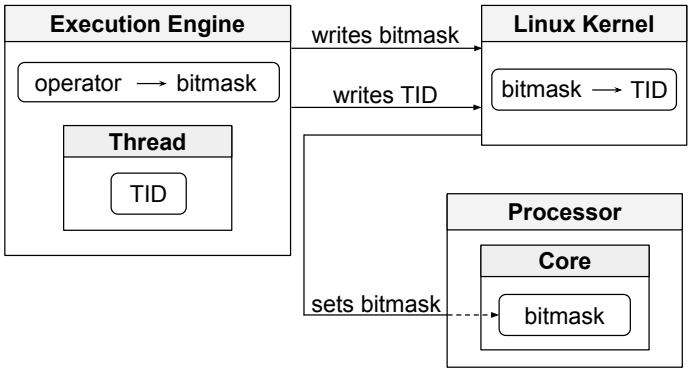


Figure 3: Interaction between the execution engine of SAP HANA, the Linux kernel and the processor.

can hurt performance and predictability, especially in concurrent workloads whenever a query suffers from cache pollution caused by another query running on the same processor: the throughput of cache-sensitive operators may degrade by more than $50\,\%$ [11]. In particular, some workloads are highly sensitive to the available amount of CPU cache (e.g., random accesses to a small hash table), contrary to cache-insensitive operations such as a sequential scan of a large memory area. The good news is that hardware manufacturers allow fine-grained control of cache allocation by offering mechanisms such as Intel's *Cache Allocation Technology* (CAT) [12]. By integrating a cache partitioning mechanism, which partitions the cache for individual database operators, into the execution engine of a prototype version of SAP HANA, we were able to improve the overall system performance by up to $38\,\%$.

Cache-Sensitivity. We analyzed the cache-sensitivity of individual operators using the example of the *column scan* operator, the *aggregation with grouping* operator and the *foreign key join* operator. Our results show that *column scans* do not profit from a large LLC. This observation does not come as a surprise because, by nature, scans read data exactly once from DRAM without re-using it. *Aggregations*, by contrast, can be highly sensitive to the size of the LLC. The algorithm that we consider is based on hashing, and is most cache-sensitive whenever the size of the hash tables is comparable to the (configured) LLC size. Only if the hash table is either very small or very large, cache sensitivity becomes less significant. Finally, the cache sensitivity of *foreign key joins* heavily depends on the input data, i.e., the cardinality of the primary keys: If the size of the bit vector used internally is comparable to the size of the cache, the algorithm becomes cache-sensitive. Otherwise the operator does not profit from a large LLC.

Cache Partitioning. Traditionally, the user has little control over the cache, as it is entirely managed by hardware. Techniques such as *page coloring* [13] offer the possibility of partitioning the cache by allocating memory in specific memory pages, known to map to a specific portion of the cache. But those techniques require significant changes to the OS and the application. Plus, re-partitioning the cache at runtime requires copying the data [14, 15].

With the microarchitecture codenamed "Haswell" Intel introduced *Cache Allocation Technology* (CAT) [12] to partition the *last-level cache* of a processor. We use the Linux kernel interface of CAT (available since version 4.10 [16]) to integrate cache partitioning into the execution engine of a prototype version of SAP HANA. A schematic overview of how we retrofitted cache partitioning into an execution engine is illustrated in Figure 3.

The execution engine of SAP HANA uses a thread pool of worker threads to execute *jobs* [10], e.g., an operator. We annotate a job with information of its *cache usage*. The approach is similar to *workload classes*, but its granularity differs. While *workload classes* annotate entire queries, the cache partitioning mechanism annotates jobs, internal units of work often featuring a distinctive memory access pattern. Currently, we distinguish between

three categories of jobs: *(i)* jobs which are not cache-sensitive and pollute the cache such as the *column scan*; *(ii)* jobs which are cache-sensitive and profit from the entire cache such as the *aggregation with grouping* operator for most cases; and *(iii)* jobs such as the *foreign key join* operator which can be both cache-polluting and cache-sensitive depending on the query or data. By default, a job belongs to *(ii)* to avoid regressions. During the execution of a job, the engine maps a job to a bitmask and passes the bitmask and the TID of the worker thread to the Linux kernel. The Linux kernel associates the bitmask with the TID allowing it to update a core's bitmask during thread scheduling. Integrating cache partitioning into the code base of an industry-strength system, described into more detail in [11], requires only a small effort: our actual implementation consists of less than 1000 lines of code.

While we derived the cache partitioning scheme from an experimental analysis, the application of existing characterization methods for *describing* the cache usage pattern of a database operator could be investigated. For instance, Chou and DeWitt [17] propose the query locality set model based on the knowledge of various patterns of queries to allocate buffer pool memory efficiently. Others propose the cache miss ratio as an online model for characterizing workloads or operators [18, 19, 20].

Experimental Results. Figure 4 illustrates some of our experimental results. Our evaluation confirms that, e.g., *aggregations* are sensitive to *cache pollution* caused by, e.g., *column scans*. Aggregations are most sensitive to cache pollution whenever the size of their performance-critical data structures is comparable to the size of the LLC (cf. Figure 4b). Note that columns in SAP HANA are dictionary-compressed. A compressed column contains indexes which reference values in a dictionary. A dictionary is a sorted, unique sequence of the actual domain values. Because the *aggregation* operator decompresses its input to compute the aggregate, it performs many random accesses to the dictionary of the aggregated column. Moreover, *aggregations* frequently access hash tables used for grouping.

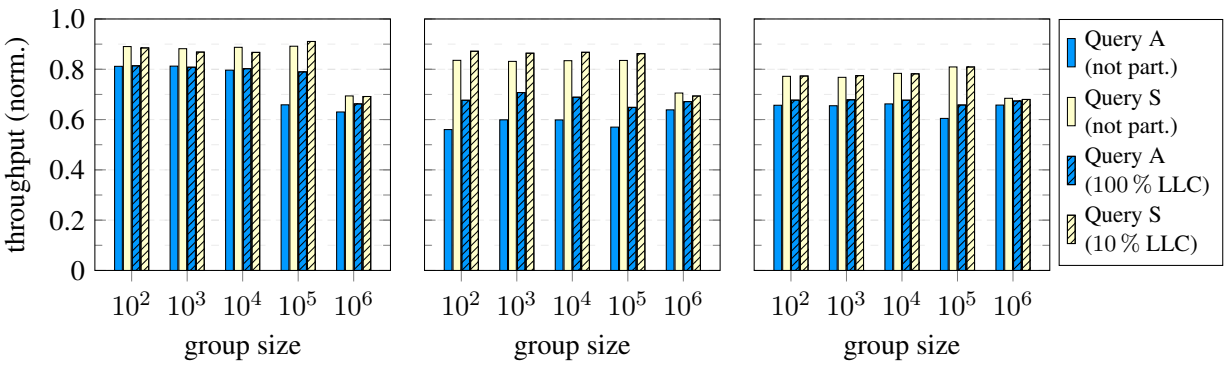
We observed that the throughput of the *aggregation* query may drop below $60\,\%$ compared to running isolated in the system. By utilizing cache partitioning and restricting the *column scan* to $10\,\%$ of the available LLC, we can improve the throughput of the *aggregation* query by up to $21\,\%$. In addition, the *column scan* operator profits from the fact that the *aggregation* operator consumes less memory bandwidth: The throughput of the *column scan* increases slightly by up to $6\,\%$, too. We determine that the overall cache hit ratio increases and that the LLC misses per instruction decrease because the *aggregation* performs fewer accesses to main memory and more accesses to the cache.

While the *column scan* operator always pollutes the cache, other operators such as the *join* operator (not shown here) only cause cache pollution whenever its frequently accessed data structures fit in the L1 or L2 cache [11]. In these cases, we can eliminate cache pollution as well and significantly improve performance by restricting the *join* to a small portion of the LLC. Generally, the search for the "best" partitioning in any given situation will depend on accurate *result size estimates*.

Conclusion. In-memory database operators exhibit different performance characteristics depending on the available cache size. We demonstrate how to integrate a cache partitioning mechanism into the execution engine of an existing DBMS with low expenditure and show in our evaluation that our approach avoids cache pollution and significantly reduces cache misses improving overall system performance. Ultimately, our results illustrate that integrating cache partitioning into a DBMS engine is worth the effort: it may improve but never degrades performance for arbitrary workloads containing scan-intensive, cache-polluting operators.

4 Interaction with the OS

The interaction of the database management system with the operation system is crucial for achieving high concurrency, high scalability and robust performance. While commercial database systems usually choose to



(a) 4 MiB Dictionary (b) 40 MiB Dictionary (c) 400 MiB Dictionary

Figure 4: Throughput of Query A (*aggregation with grouping*) and Query S (*column scan*) when executed concurrently (normalized to their throughput when running isolated). We vary the size of the dictionary of the aggregated column, we vary the number of groups, and we disable or enable cache partitioning.

circumvent the OS and exploit detailed workload information, another strategy is bringing the database system and the operating system closer together with a radical co-design. First, we discuss how a database management system may take control of memory management and task scheduling using the example of SAP HANA. Second, we present concepts as well as first experimental results for the bare metal runtime *MxKernel*, a shared platform for implementing crucial components for both the DBMS and the OS.

4.1 Bypassing the OS

Instead of relying on the operating system, SAP HANA takes care of, e.g., the memory management and task scheduling itself—similar to other vendors such as IBM [21] and Oracle [22]—to achieve high concurrency, high scalability and robust performance. In the following, we briefly present how SAP HANA "bypasses" the OS using the example of memory management and task scheduling.

Memory Management. In contrast to general-purpose allocators which do not scale to thousands of cores and usually specialize in allocating small blocks, the memory management of SAP HANA is specifically built for the needs of a database system. It is primarily optimized for high concurrency and scalability in multithreaded and NUMA environments and provides robust performance for the allocation of different sizes of memory requests and their defragmentation (compaction and garbage collection) during the long operating time of the system (cf. [23]). In addition, the tailor-made implementation supports tracking and monitoring memory operations to provide fine-grained memory usage statistics. This facilitates limiting the memory consumption globally, per instance, per process, or SQL statement for, e.g., the implementation of workload classes from Section 2. Furthermore, the memory statistics can be used for debugging memory leaks or memory corruption as well as for analyzing performance characteristics related to memory usage.

The memory management pre-allocates memory by requesting chunks of memory (using `mmap`) from the operating system. From this point on, the OS is no longer involved. The cached memory is distributed across different memory pools and completely managed by the database system. To reduce lock conflicts, each CPU has its own pool. However, pools can still take memory from other pools.

Task Scheduling. The task scheduling mechanism of SAP HANA utilizes its domain-specific knowledge of database internals to efficiently schedule tasks for highly concurrent, both analytical and transactional workloads. To mitigate the problem of blocking tasks in transactional workloads, the task scheduler dynamically adapts the

number of threads in the self-managed thread pool and prioritizes short-running OLTP queries. At the same time, the scheduler makes use of a dynamic concurrency hint for analytical workloads, which results in a lower number of tasks for OLAP queries avoiding synchronization, communication and scheduling costs [24]. In general, the task scheduler relies on the scheduler of the Linux kernel to map threads to cores, but it, e.g., manages the thread placement to NUMA nodes explicitly and supports task stealing to deal with under-utilization [25].

4.2 DB/OS Co-Design

Most DBMS such as SAP HANA implement, e.g., their own memory management or task scheduling to achieve high concurrency, high scalability as well as robust performance. Bypassing the OS comes at a price, however. First, the DBMS re-implements features already existing in the OS. This means that program code may be duplicated, which in return creates unnecessary maintenance and development costs. In addition, with the OS and the DBMS having their own implementations it becomes increasingly difficult to share information. Hence, the DBMS might have detailed information about its workloads, but at the same time it lacks the comprehensive hardware and system information available to the OS and vice versa. Second, bypassing the OS works best if the DBMS is the only application. However, as soon as the DBMS is co-running with another application on the same machine, e.g., in a cloud scenario or an on-premise infrastructure, dynamically managing a machine's hardware resources becomes very difficult.

A solution could be the introduction of a central component to which each application as well as the OS communicate their resource demands. Such a component could then possess detailed information about each of the applications' workloads running on the OS *and* about the OS itself. Consequently, a radical *co-design* of OS and DBMS (as well as other applications) could address the problem of dynamic resource management on a shared machine. A first step into this direction is the bare-metal platform *MxKernel*, which we introduce in the following.

Architecture of MxKernel. Usually, DBMS are built on top of an underlying OS which is used to abstract the machine's hardware such as CPU architectures or complex memory hierarchies. This abstraction can result in the loss of information relevant to the DBMS. For example, applications running on top of the OS have less knowledge about parallel executed applications, the load factor of the machine and the actual hardware structure. Using external libraries like `libnuma` [26] might help to gain additional hardware information such as local and remote memory areas or the NUMA distribution of CPU cores. But the use is more like a crutch and does not solve the problem holistically. It remains, e.g, difficult to explore and utilize cache hierarchies or topology structures of various modern hardware [5].

A first step towards solving these problems in a holistic way is the bare-metal platform MxKernel. Figure 5 depicts an overview of the architecture. The platform's core is a basis layer, which enables to run all applications as well as the OS side by side. Note that running an OS is optional. At the same time, MxKernel provides a detailed view of the hardware and an interface to which an application can communicate its requirements regarding hardware resources or performance expectations.

In particular, it implements services for hardware compatibility and a mechanism for control flow execution. Moreover, its architecture allows, e.g., the OS and the DBMS to share data structures and algorithms such as B-Trees, which can be used for primary key indexing by the DBMS or for implementing a file system by the OS.

MxTasks. The platform MxKernel introduces *MxTasks* to abstract from and to create an interface to the control flow execution. MxTasks describe small units of work representing a more lightweight alternative to POSIX Threads. They are executed atomically by implementing a run-to-completion semantic. Usually, MxTasks feature only a small sequence of instructions. As a result, their runtime, memory accesses, and resource requirements are easier to predict and to define than threads. In addition, MxTasks can be coupled with precise metadata about, e.g., its memory access patterns, its accessed data structures and its preferred NUMA region. Thus, by annotating

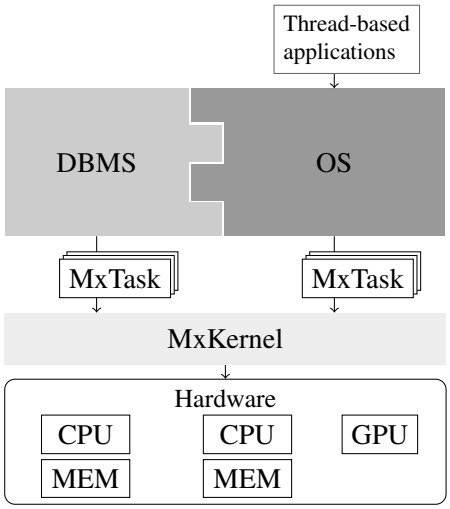


Figure 5: MxKernel provides a small basis layer for managing shared hardware resources between all applications running on a shared machine, including, e.g., the OS and the DBMS.

MxTasks with workload information we employ the same concept of sharing semantics between application, OS, DBMS, and hardware as discussed for *workload classes* in Section 2 or for *jobs* in Section 3.

MxKernel creates an execution plan that synchronizes MxTasks for shared-resource accesses and optimizes them for cache and NUMA locality. Thus, MxKernel functions as a central component for resource management which receives (workload) information in the form of MxTasks with metadata. Its execution framework then tries to fulfill the requirements of all applications as well as possible. Applications running on top of an (optional) OS, however, can still choose to use threads instead of MxTasks for compatibility reasons.

Furthermore, MxTasks represent a way to exploit heterogeneous systems by providing different implementations for distinct processing units such as GPUs or CPUs. Based on system load and the runtime property of a MxTask, MxKernel could then choose the most suitable available hardware for execution.

Experimental Results. As a first use case, we evaluated an index data structure based on the B^{link}-tree [27]. We implement the insert algorithm described by Lehman and Yao [27]. However, we modify it to support MxTasks: We spawn a new task for each traversal of a node. This means that we create a task for looking up the child node to traverse next. When a task finally finds the matching leaf node, we insert the new value.

While concurrent POSIX Threads accessing the same node of the B^{link}-tree have to be protected using latches, MxTasks benefit from the run-to-completion semantic: Mapping the traversal of a tree node to a specific processor core ensures that only one task accesses a node at a time. In addition, we benefit from cache locality because one core accesses the same data structure multiple times.

We consider two different implementations of the B^{link}-tree: a version based on POSIX Threads executed on Linux, and a version based on MxTasks running on MxKernel. The workload consists of $16 \cdot 10^6$ insert operations using unique key-value pairs. We execute the experiment on a system with two Intel Xeon E5-2690 processors with 8 CPU cores each, and simultaneous multithreading enabled.

The results illustrated in Figure 6 show that while the thread-based version exhibits a higher throughput until using eight threads, the throughput starts dropping after using four threads. We explain the differences with the latch contention of the thread version: Every node has to be protected by a latch to prevent inconsistencies. MxTasks, on the other hand, have no need to use latches. Every node of the B^{link}-tree is assigned to one of the cores and every MxTask accessing a node will be executed on the mapped core without suspension.

In addition, we notice that the throughput degrades slightly after using cores from the second NUMA node.

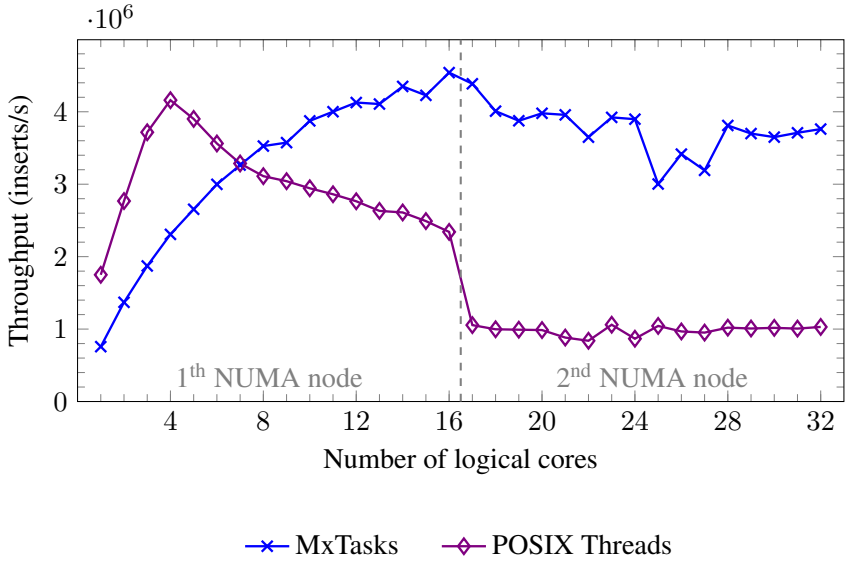


Figure 6: Throughput of insert operations into a B^{link}-tree. While the throughput drops down when using more than four POSIX Threads, the throughput stagnates when using MxTasks from the second NUMA region.

We attribute this to the increased overhead of cache coherency. For managing MxTasks and tracking their execution state, we use a wait-free queue for every core. Pushing MxTasks from a queue of one core to the queue of another core, triggers the cache coherency mechanism of the processor which costs additional execution time. This effect occurs as soon as another NUMA node is involved. Note that the throughput of the thread-based version, on the other hand, drops significantly with more than 16 cores.

Conclusion. The interaction between DBMS and OS are crucial for creating high-performance systems. We briefly present how a commercial system such as SAP HANA bypasses the OS and implements, e.g., its own memory management and task scheduling to guarantee robust performance and scaling by exploiting domain-specific knowledge, or to track and to enforce the consumption of hardware resources. Another strategy is the co-design of DBMS and OS. As a first step towards this goal, we introduce the bare-metal platform *MxKernel*. On top of it, the DBMS and the OS run side by side with an interface to communicate their hardware and runtime requirements to. In return, MxKernel manages hardware resources exclusively.

5 Conclusion

Operational analytics database management systems with the capability to perform both high-speed transactional processing and complex analytical workloads on the same dataset and at the same time, impose new challenges and practical implications for system designers. In this article, we discussed several solutions that allow to manage these resource-intensive workloads in an efficient and holistic way. In particular, we explicitly share workload information between application, OS, DBMS, and hardware which allows us to manage resources efficiently and to improve performance and predictability.

By sharing context information between the application and the DBMS, systems can implement an application-aware resource management and quality of service features. SAP HANA employs *workload classes* for limiting the resource consumption of queries issued by individual applications. In addition, we presented its *admission control* mechanism to manage the amount of requests the system handles at a given time in order to avoid contention in peak load situations.

By sharing cache usage information between the DBMS and the hardware, we implement a mechanism that *dynamically partitions* the *cache* per individual operator. We demonstrated that this may improve the overall throughput of highly concurrent workloads, where we restrict the cache usage of scan-intensive operators and increase the cache capacity for cache-sensitive operators.

Finally, we discussed how the database system interacts with the operating system. While DBMS traditionally take control over individual features of the OS and implement, e.g., a custom memory management or task scheduling, other research directions explore their co-design. In particular, we presented the bare-metal platform *MxKernel*. On top of it, the DBMS and the OS run side by side and communicate their hardware and runtime requirements to the platform. In return, MxKernel takes care of managing all hardware resources.

Acknowledgments

This work was supported by DFG, Deutsche Forschungsgemeinschaft, grant number TE 1117/2-1.

References

[1] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. *SIGMOD*, 2009, pp. 1–2.

[2] A. Böhm, J. Dittrich, N. Mukherjee, I. Pandis, and R. Sen. Operational Analytics Data Management Systems. *VLDB*, vol. 9, no. 13, pp. 1601–1604, 2016.

[3] H. Plattner and B. Leukert. The In-Memory Revolution: How SAP HANA Enables Business of the Future. Springer, 2015.

[4] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K. Sattler. Scaling up Mixed Workloads: a Battle of Data Freshness, Flexibility, and Scheduling. *TPCTC*, 2014, pp. 97–112.

[5] J. Giceva, T. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. COD: Database / Operating System Co-Design. *CIDR*, 2013.

[6] K. Kara, J. Giceva, and G. Alonso. FPGA-based Data Partitioning. *SIGMOD*, 2017, pp. 433–445.

[7] A. Böhm. Novel Optimization Techniques for Modern Database Environments. *BTW*, 2015, pp. 23–24.

[8] M. Zhang. Autonomic Workload Management for Database Management Systems. *Ph.D. thesis*, Queen's University, 2014.

[9] SAP SE. SAP HANA Administration Guide. `https://help.sap.com/viewer/6b94445c94ae495c83a19646e7c3fd56/2.0.03/en-US`, 2018.

[10] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *VLDB*, vol. 8, no. 12, pp. 1442–1453, 2015.

[11] S. Noll, J. Teubner, N. May, and A. Böhm. Accelerating Concurrent Workloads with CPU Cache Partitioning. *ICDE*, 2018, pp. 437–448.

[12] Intel Corporation. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *White paper*, 2015.

[13] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *VLDB*, vol. 2, no. 1, pp. 373–384, 2009.

[14] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems. *HPCA*, IEEE Computer Society, 2008, pp. 367–378.

[15] X. Zhang, S. Dwarkadas, and K. Shen. Towards Practical Page Coloring-based Multicore Cache Management. *EuroSys*, ACM, 2009, pp. 89–102.

[16] Intel Corporation. User Interface for Resource Allocation in Intel Resource Director Technology. Documentation of the Linux Kernel, `https://www.kernel.org/doc/Documentation/x86/intel_rdt_ui.txt`, 2017.

[17] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *VLDB*, 1985, pp. 127–141.

[18] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. *ASPLOS*, ACM, 2009, pp. 121–132.

[19] S. Manegold, P. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. *VLDB*, 2002, pp. 191–202.

[20] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. *ASPLOS*, ACM, 2004, pp. 177–188.

[21] IBM. The DB2 UDB memory model – How DB2 uses memory. `https://www.ibm.com/developerworks/data/library/techarticle/dm-0406qi/`, 2018.

[22] Oracle. Database Administrator's Guide – Managing Memory. `https://docs.oracle.com/database/121/ADMIN/memory.htm`, 2018.

[23] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory Management Techniques for Large-Scale. Persistent-Main-Memory Systems. *VLDB*, vol. 10, no. 11, pp. 1166–1177, 2017.

[24] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. *ADMS*, 2013, pp. 36–45.

[25] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Adaptive NUMA-aware Data Placement and Task Scheduling for Analytical Workloads in Main-Memory Column-Stores. *VLDB*, vol. 10, no. 2, pp. 37–48, 2016.

[26] A. Kleen. A NUMA API for Linux. *White paper*, SUSE Labs, 2004.

[27] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, 1981.

Leveraging Hyperupcalls To Bridge The Semantic Gap: An Application Perspective

Michael Wei
VMware Research

Nadav Amit
VMware Research

Abstract

Hyperupcalls are a mechanism which we recently proposed to bridge the semantic gap between a hypervisor and its guest virtual machines (VMs) by allowing the guest VM to provide the hypervisor safe, verifiable code to transfer information. With a hyperupcall, a hypervisor can safely read and update data structures inside the guest, such as page tables. A hypervisor could use such a hyperupcall, for example, to determine which pages are free and can be reclaimed in the VM without invoking it.

In this paper, we describe hyperupcalls and how they have been used to improve and gain additional insight on virtualized workloads. We also observe that complex applications such as databases hold a wealth of semantic information which the systems they run on top of are unaware of. For example, a database may store records, but the operating system can only observe bytes written into a file, and the hypervisor beneath it blocks written to a disk, limiting the optimizations the system may make: for instance, if the operating system understood the database wished to fetch a database record, it could prefetch related records. We explore how mechanisms like hyperupcalls could be used from the perspective of an application, and demonstrate two use cases from an application perspective: one to trace events in both the guest and hypervisor simultaneously and another simple use case where a database installs a hyperupcall so the hypervisor can prioritize certain traffic and improve response latency.

1 Introduction

Previously, we have described Hyperupcalls [4], a tool used to bridge the *semantic gap* in virtualization, where one side of an abstraction (virtual machines) must be oblivious to the other (hypervisors). The abstraction of a virtual machine (VM), enables hosts known as *hypervisors* to run multiple operating systems (OSs) known as *guests* simultaneously, each under the illusion that they are running in their own physical machine. This is achieved by exposing a hardware interface which mimics that of true, physical hardware. The introduction of this simple abstraction has led to the rise of the modern data center and the cloud as we know it today. Unfortunately, virtualization is not without drawbacks. Although the goal of virtualization is for VMs and hypervisors to be oblivious from each other, this separation renders both sides unable to understand decisions made on the other side, a problem known as the semantic gap.

The semantic gap is not limited to virtualization - it exists whenever an abstraction is used, since the purpose of the abstraction is to hide implementation details. For example, in databases, a SQL query has limited insight

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

into how the query planner may execute it, and the query planner has limited insight into the application that sent the SQL query. Insight into the application may help the database execute the query optimally. If the database could see that an application was overloaded, it might be able to delay execution of the query and use the resources for other applications. Likewise, if the application could see that a column was being reindexed, it may perform the query on another column, or defer that query for later.

In virtualization, addressing the semantic gap is critical for performance. Without information about decisions made in guests, hypervisors may suboptimally allocate resources. For example, the hypervisor cannot know what memory is free in guests without understanding their internal OS state, breaking the VM abstraction. As a result, in virtualized environments, many mechanisms have been developed to bridge the semantic gap. State-of-the-art hypervisors today typically bridge the semantic gap with *paravirtualization* [10, 35], which makes the guest aware of the hypervisor. Several paravirtual mechanisms exist and are summarized in Table 1. Another mechanism hypervisors may leverage is *introspection*, which enables the hypervisor to observe the behavior of virtual machines without prior coordination.

Each one of these mechanisms used in virtualization may be used elsewhere where the semantic gap exists. For example, a paravirtual mechanism such as a hypercall might be similar to an application informing a database using an RPC call, whereas an upcall may involve a database informing an application about it's internal state. While introspection would be more complicated to implement in the context of applications such as databases, one can imagine a database attempting to infer the workload of an application by measuring its query rate, or an application trying to determine if a database is overloaded by measuring response latency.

In this paper, we describe the design and implementation of hyperupcalls [1], a technique which enables a hypervisor to communicate with a guest, like an upcall, but without a context switch, like VMI. This is achieved through the use of verified code, which enables a guest to communicate to the hypervisor in a flexible manner while ensuring that the guest cannot provide misbehaving or malicious code. Once a guest registers a hyperupcall, the hypervisor can execute it to perform actions such as locating free guest pages or running guest interrupt handlers without switching into the guest. We believe that hyperupcalls are useful from the perspective of an application - both as a tool for an application such as a database to gain insight about its clients, and for applications to use in a virtualized environment, breaking the semantic gap between the application, guest operating system and hypervisor.

Hyperupcalls are easy to build: they are written in a high level language such as C, and we provide a framework which allows hyperupcalls to share the same codebase and build system as the Linux kernel that may be generalized to other operating systems. When the kernel is compiled, a toolchain translates the hyperupcall into verifiable bytecode. This enables hyperupcalls to be easily maintained. Upon boot, the guest registers the hyperupcalls with the hypervisor, which verifies the bytecode and compiles it back into native code for performance. Once recompiled, the hypervisor may invoke the hyperupcall at any time.

We have previously shown that hyperupcalls enable a hypervisor to be proactive about resource allocation when memory is overcommitted, enhance performance when interprocessor interrupts (IPIs) are used, and enhance the security and debuggability of systems in virtual environments [4]. In this paper, we show an application use case: we design a hyperupcall which is installed by memcached to prioritize the handling and reduce the latency of certain requests.

2 Communication Mechanisms

It is now widely accepted that in order to extract the most performance and utility from virtualization, hypervisors and their guests need to be aware of one another. To that end, a number of mechanisms exist to facilitate communication between hypervisors and guests. Table 1 summarizes these mechanisms, which can be broadly characterized by the requestor, the executor, and whether the mechanism requires that the hypervisor and the

[1] Hyperupcalls were previously published as "hypercallbacks" [5].

Requestor	Paravirtual, Executed by: Hypervisor	Paravirtual, Executed by: Guest	Uncoordinated: Introspection
Guest	*Hypercalls*	Pre-Virt [24]	HVI [42]
HV	**Hyperupcalls**	*Upcalls*	VMI [18]

Table 1: *Hypervisor-Guest Communication Mechanisms.* Hypervisors (HV) and guests may communicate through a variety of mechanisms, which are characterized by who initiates the communication, who executes and whether the channel for communication is coordinated (paravirtual). *Italicized* cells represent channels which require context switches.

guest coordinate ahead of time. We note that many of these mechanisms have analogs in other places where the semantic gap may exist. For example, a hypercall might be similar to a query notification in a database, and an upcall might resemble an out-of-band request to a service.

In the next section, we discuss these mechanisms and describe how hyperupcalls fulfill a need for a communication mechanism where the hypervisor makes and executes its own requests without context switching. We begin by introducing state-of-the-art paravirtual mechanisms in use today.

2.1 Paravirtualization

Hypercalls and upcalls. Most hypervisors today leverage paravirtualization to communicate across the semantic gap. Two mechanisms in widespread use today are *hypercalls*, which allow guests to invoke services provided by the hypervisor, and *upcalls*, which enable the hypervisor to make requests to guests. Paravirtualization means that the interface for these mechanisms are coordinated ahead of time between hypervisor and guest [10].

One of the main drawbacks of upcalls and hypercalls is that they require a context switch as both mechanisms are executed on the opposite side of the request. As a result, these mechanisms must be invoked with care. Invoking a hypercall or upcall too frequently can result in high latencies and computing resource waste [3].

Another drawback of upcalls in particular that the requests are handled by the guest, which could be busy handling other tasks. If the guest is busy or if a guest is idle, upcalls incur the additional penalty of waiting for the guest to be free or for the guest or woken up. This can take an unbounded amount of time, and hypervisors may have to rely on a penalty system to ensure guests respond in a reasonable amount of time.

Finally, by increasing the coupling between the hypervisor and its guests, paravirtual mechanisms can be difficult to maintain over time. Each hypervisor have their own paravirtual interfaces, and each guest must implement the interface of each hypervisor. The paravirtual interface is not thin: Microsoft's paravirtual interface specification is almost 300 pages long [26]. Linux provides a variety of paravirtual hooks, which hypervisors can use to communicate with the VM [46]. Despite the effort to standardize the paravirtualization interfaces they are incompatible with each other, and evolve over time, adding features or even removing some (e.g., Microsoft hypervisor event tracing). As a result, most hypervisors do not fully support efforts to standardize interfaces and specialized OSs look for alternative solutions [25, 31].

Pre-virtualization. Pre-Virtualization [24] is another mechanism in which the guest requests services from the hypervisor, but the requests are served in the context of the guest itself. This is achieved by code injection: the guest leaves stubs, which the hypervisor fills with hypervisor code. Pre-virtualization offers an improvement over hypercalls, as they provide more flexible interface between the guest and the hypervisor. Arguably, pre-virtualization suffers from a fundamental limitation: code that runs in the guest is deprivileged and cannot perform sensitive operations, for example, accessing shared I/O devices. As a result, in pre-virtualization, the hypervisor code that runs in the guest still needs to communicate with the privileged hypervisor code using hypercalls.

2.2 Introspection

Introspection occurs when a hypervisor or guest attempts to infer information from the other context without directly communicating with it. With introspection, no interface or coordination is required. For instance, a hypervisor may attempt to infer the state of completely unknown guests simply by their memory access patterns. Another difference between introspection and paravirtualization is that no context switch occurs: all the code to perform introspection is executed in the requestor.

Virtual machine introspection (VMI). When a hypervisor introspects a guest, it is known as VMI [18]. VMI was first introduced to enhance VM security by providing intrusion detection (IDS) and kernel integrity checks from a privileged host [9, 17, 18]. VMI has also been applied to checkpointing and deduplicating VM state [1], as well as monitoring and enforcing hypervisor policies [32]. These mechanisms range from simply observing a VM's memory and I/O access patterns [22] to accessing VM OS data structures [14], and at the extreme end they may modify VM state and even directly inject processes into it [19, 15]. The primary benefits of VMI are that the hypervisor can directly invoke VMI without a context switch, and the guest does not need to be "aware" that it is inspected for VMI to function. However, VMI is fragile: an innocuous change in the VM OS, such as a hotfix which adds an additional field to a data structure could render VMI non-functional [8]. As a result, VMI tends to be a "best effort" mechanism.

HVI. Used to a lesser extent, a guest may introspect the hypervisor it is running on, known as hypervisor introspection (HVI) [42, 37]. HVI is typically employed either to secure a VM from untrusted hypervisors [38] or by malware to circumvent hypervisor security [36, 28].

2.3 Extensible OSes

While hypervisors provide a fixed interface, OS research suggested along the years that flexible OS interfaces can improve performance without sacrificing security. The Exokernel provided low level primitives, and allowed applications to implement high-level abstractions, for example for memory management [16]. SPIN allowed to extend kernel functionality to provide application-specific services, such as specialized interprocess communication [11]. The key feature that enables these extensions to perform well without compromising security, is the use of a simple byte-code to express application needs, and running this code at the same protection ring as the kernel. Our work is inspired by these studies, and we aim to design a flexible interface between the hypervisor and guests to bridge the semantic gap.

2.4 Hyperupcalls

This paper introduces hyperupcalls, which fulfill a need for a mechanism for the hypervisor to communicate to the guest which is coordinated (unlike VMI), executed by the hypervisor itself (unlike upcalls) and does not require context switches (unlike hypercalls). With hyperupcalls, the VM coordinates with the hypervisor by registering verifiable code. This code is then executed by the hypervisor in response to events (such as memory pressure, or VM entry/exit). In a way, hyperupcalls can be thought of as upcalls executed by the hypervisor.

In contrast to VMI, the code to access VM state is provided by the guest so the hyperupcalls are fully aware of guest internal data structures— in fact, hyperupcalls are built with the guest OS codebase and share the same code, thereby simplifying maintenance while providing the OS with an expressive mechanism to describe its state to underlying hypervisors.

Compared to upcalls, where the hypervisor makes asynchronous requests to the guest, the hypervisor can execute a hyperupcall at any time, even when the guest is not running. With an upcall, the hypervisor is at the mercy of the guest, which may delay the upcall [6]. Furthermore, because upcalls operate like remote requests,

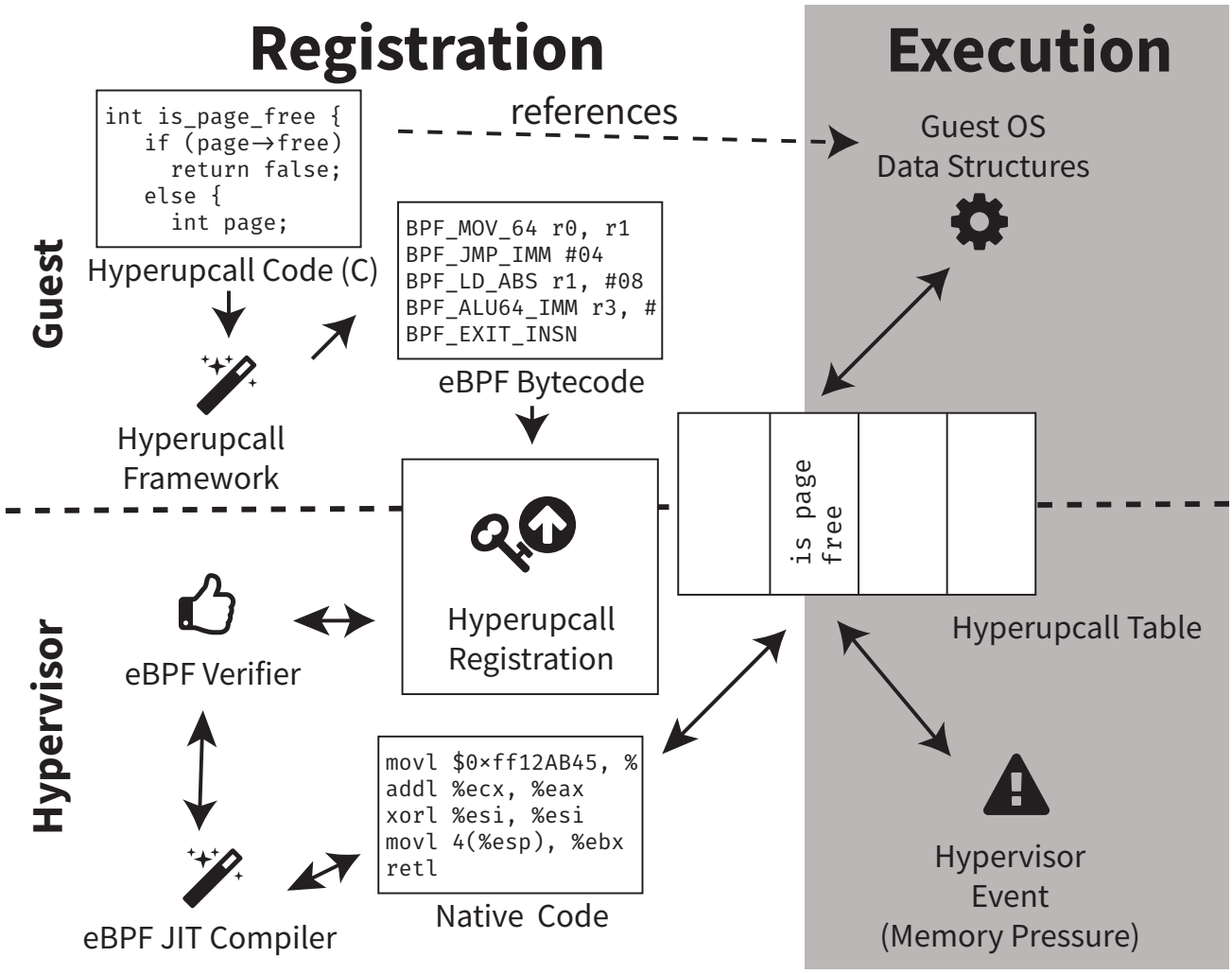


Figure 1: *System Architecture.* Hyperupcall registration (left) consists of compiling C code, which may reference guest data structures, into verifiable bytecode. The guest registers the generated bytecode with the hypervisor, which verifies its safety, compiles it into native code and sets it in the VM hyperupcall table. When the hypervisor encounters an event (right), such as a memory pressure, it executes the respective hyperupcall, which can access and update data structures of the guest.

upcalls may be forced to implement OS functionality in a different manner. For example, when flushing remote pages in memory ballooning [41], the canonical technique for identifying free guest memory, the guest increases memory pressure using a dummy process to free pages. With a hyperupcall, the hypervisor can act as if it were a guest kernel thread and scan the guest for free pages directly.

Hyperupcalls resemble pre-virtualization, in that code is transferred across the semantic gap. Transferring code not only allows for more expressive communication, but it also moves the execution of the request to the other side of the gap, enhancing performance and functionality. Unlike pre-virtualization, the hypervisor cannot trust the code being provided by the virtual machine, and the hypervisor must ensure that execution environment for the hyperupcall is consistent across invocations.

In some ways, hyperupcalls may resemble SQL stored procedures: they are a block of code installed across the semantic gap, and code is more expressive than a simple request. One important distinction between hyperupcalls and stored procedures is that hyperupcalls have access to the guest state and data structures, whereas stored

procedures do not. This allows hyperupcalls to be much more expressive and dynamic compared to simple code transfer mechanisms.

3 Architecture

Hyperupcalls are short verifiable programs provided by guests to the hypervisor to improve performance or provide additional functionality. Guests provide hyperupcalls to the hypervisor through a *registration* process at boot, allowing the hypervisor to access the guest OS state and provide services by *executing* them after verification. The hypervisor runs hyperupcalls in response to events or when it needs to query guest state. The architecture of hyperupcalls and the system we have built for utilizing them is depicted in Figure 1.

We aim to make hyperupcalls as simple as possible to build. To that end, we provide a complete *framework* which allows a programmer to write hyperupcalls using the guest OS codebase. This greatly simplifies the development and maintenance of hyperupcalls. The framework compiles this code into verifiable code which the guest registers with the hypervisor. In the next section, we describe how an OS developer writes a hyperupcall using our framework. Some of the implementation details, especially in regards to how we verify that the code is safe and execute the hyperupcall, are out of the scope of this paper. For more details, see [4].

3.1 Building Hyperupcalls

Guest OS developers write hyperupcalls for each hypervisor event they wish to handle. Hypervisors and guests agree on these events, for example VM entry/exit, page mapping or virtual CPU (VCPU) preemption. Each hyperupcall is identified by a predefined identifier, much like the UNIX system call interface [33].

3.1.1 Providing Safe Code

One of the key properties of hyperupcalls is that the code must be guaranteed to not compromise the hypervisor. In order for a hyperupcall to be safe, it must only be able to access a restricted memory region dictated by the hypervisor, run for a limited period of time without blocking, sleeping or taking locks, and only use hypervisor services that are explicitly permitted.

Since the guest is untrusted, hypervisors must rely on a security mechanism which guarantees these safety properties. There are many solutions that we could have chosen: software fault isolation (SFI) [40], proof-carrying code [30] or safe languages such as Rust. To implement hyperupcalls, we chose the enhanced Berkeley Packet Filter (eBPF) VM.

We chose eBPF for several reasons. First, eBPF is relatively mature: BPF was introduced over 20 years ago and is used extensively throughout the Linux kernel, originally for packet filtering but extended to support additional use cases such as sandboxing system calls (`seccomp`) and tracing of kernel events [21]. eBPF enjoys wide adoption and is supported by various runtimes [12, 29]. Second, eBPF can be provably verified to have the safety properties we require, and Linux ships with a verifier and JIT which verifies and efficiently executes eBPF code [43]. Finally, eBPF has a LLVM compiler backend, which enables eBPF bytecode to be generated from a high level language using a compiler frontend (Clang). Since OSes are typically written in C, the eBPF LLVM backend provides us with a straightforward mechanism to convert unsafe guest OS source code into verifiably safe eBPF bytecode.

3.1.2 From C to eBPF — the *Framework*

Unfortunately, writing a hyperupcall is not as simple recompiling OS code into eBPF bytecode. However, our framework aims to make the process of writing a hyperupcalls simple and maintainable as possible. The framework provides three key features that simplify the writing of hyperupcalls. First, the framework takes care

of dealing with guest address translation issues so guest OS symbols are available to the hyperupcall. Second, the framework addresses limitations of eBPF, which places significant constraints on C code. Finally, the framework defines a simple interface which provides the hyperupcall with data so it can execute efficiently and safely.

Guest OS symbols and memory. Even though hyperupcalls have access to the entire physical memory of the guest, accessing guest OS data structures requires knowing where they reside. OSes commonly use kernel address space layout randomization (KASLR) to randomize the virtual offsets for OS symbols, rendering them unknown during compilation time. Our framework enables OS symbol offsets to be resolved at runtime by associating pointers using address space attributes and injecting code to adjust the pointers. When a hyperupcall is registered, the guest provides the actual symbol offsets enabling a hyperupcall developer to reference OS symbols (variables and data structures) in C code as if they were accessed by a kernel thread.

Global / Local Hyperupcalls. Not all hyperupcalls need to be executed in a timely manner. For example, notifications informing the guest of hypervisor events such as a VM-entry/exit or interrupt injection only affect the guest and not the hypervisor. We refer to hyperupcalls that only affect the guest that registered it as local, and hyperupcalls that affect the hypervisor as a whole as global. If a hyperupcall is registered as local, we relax the timing requirement and allow the hyperupcall to block and sleep. Local hyperupcalls are accounted in the vCPU time of the guest similar to a trap, so a misbehaving hyperupcall penalizes itself.

Global hyperupcalls, however, must complete their execution in a timely manner. We ensure that for the guest OS pages requested by global hyperupcalls are pinned during the hyperupcall, and restrict the memory that can be accessed to 2% (configurable) of the guest's total physical memory. Since local hyperupcalls may block, the memory they use does not need to be pinned, allowing local hyperupcalls to address all of guest memory.

Addressing eBPF limitations. While eBPF is expressive, the safety guarantees of eBPF bytecode mean that it is not Turing-complete and limited, so only a subset of C code can be compiled into eBPF. The major limitations of eBPF are that it does not support loops, the ISA does not contain atomics, cannot use self-modifying code, function pointers, static variables, native assembly code, and cannot be too long and complex to be verified.

One of the consequences of these limitations is that hyperupcall developers must be aware of the code complexity of the hyperupcall, as complex code will fail the verifier. While this may appear to be an unintuitive restriction, other Linux developers using BPF face the same restriction, and we provide a helper functions in our framework to reduce complexity, such as `memset` and `memcpy`, as well as functions that perform native atomic operations such as `cmpxchg`. A selection of these helper functions is shown in Table 2. In addition, our framework masks memory accesses (§3.4), which greatly reduces the complexity of verification. In practice, as long as we were careful to unroll loops, we did not encounter verifier issues while developing the use cases in (§4) using a setting of 4096 instructions and a stack depth of 1024.

Hyperupcall interface. When a hypervisor invokes a hyperupcall, it populates a context data structure, shown in Table 3. The hyperupcall receives an `event` data structure which indicates the reason the callback was called, and a pointer to the guest (in the address space of the hypervisor, which is executing the hyperupcall). When the hyperupcall completes, it may return a value, which can be used by the hypervisor.

Writing the hyperupcall. With our framework, OS developers write C code which can access OS variables and data structures, assisted by the helper functions of the framework. A typical hyperupcall will read the `event` field, read or update OS data structures and potentially return data to the hypervisor. Since the hyperupcall is part of the OS, the developers can reference the same data structures used by the OS itself—for example, through header files. This greatly increases the maintainability of hyperupcalls, since data layout changes are synchronized between the OS source and the hyperupcall source.

Helper Name	Function
send_vcpu_ipi	Send an interrupt to VCPU
get_vcpu_register	Read a VCPU register
set_vcpu_register	Read a VCPU register
memcpy	memcpy helper function
memset	memset helper function
cmpxchg	compare-and-swap
flush_tlb_vcpu	Flush VCPU's TLB
get_exit_info	Get info on an VM_EXIT event

Table 2: *Selected hyperupcall helper functions.* The hyperupcall may call these functions implemented in the hypervisor, as they cannot be verified using eBPF.

Input field	Function
event	Event specific data including event ID.
hva	Host virtual address (HVA) in which the guest memory is mapped.
guest_mask	Guest address mask to mask bits which are higher than the guest memory address-width. Used for verification (§3.4).
vcpus	Pointers to the hypervisor VCPU data structure, if the event is associated with a certain VCPU, or a pointer to the guest OS data structure. Inaccessible to the hyperupcall, but used by helper functions.
vcpu_reg	Frequently accessed VCPU registers: instruction pointer and VCPU ID.
env	Environment variables, provided by the guest during hyperupcallregistration. Used to set address randomization offsets.

Table 3: *Hyperupcall context data*. These fields are populated by the hypervisor when a hyperupcall is called.

It is important to note that a hyperupcall cannot invoke guest OS functions directly, since that code has not been secured by the framework. However, OS functions can be compiled into hyperupcalls and be integrated in the verified code.

3.2 Compilation

Once the hyperupcall has been written, it needs to be compiled into eBPF bytecode before the guest can register it with the hypervisor. Our framework generates this bytecode as part of the guest OS build process by running the hyperupcall C code through Clang and the eBPF LLVM backend, with some modifications to assist with address translation and verification:

Guest memory access. To access guest memory, we use eBPF's direct packet access (DPA) feature, which was designed to allow programs to access network packets safely and efficiently without the use of helper functions. Instead of passing network packets, we utilize this feature by treating the guest as a "packet". Using DPA in this manner required a bug fix [2] to the eBPF LLVM backend, as it was written with the assumption that packet sizes are $\leq$64KB.

Address translations. Hyperupcalls allow the hypervisor to seamlessly use guest virtual addresses (GVAs), which makes it appear as if the hyperupcall was running in the guest. However, the code is actually executed by the hypervisor, where host virtual address (HVAs) are used, rendering guest pointers invalid. To allow the use of guest pointers transparently in the host context, these pointers therefore need to be translated from GVAs into HVAs. We use the compiler to make these translations.

To make this translation simple, the hypervisor maps the GVA range contiguously in the HVA space, so address translations can easily be done by adjusting the base address. As the guest might need the hyperupcall

to access multiple contiguous GVA ranges—for example, one for the guest 1:1 direct mapping and of the OS text section [23]—our framework annotates each pointer with its respective "address space" attribute. We extend the LLVM compiler to use this information to inject eBPF code that converts each of the pointer from GVA to HVA by a simple subtraction operation. It should be noted that the generated code safety is not assumed by the hypervisor and is verified when the hyperupcall is registered.

Bound Checks. The verifier rejects code with direct memory accesses unless it can ensure the memory accesses are within the "packet" (in our case, guest memory) bounds. We cannot expect the hyperupcall programmer to perform the required checks, as the burden of adding them is substantial. We therefore enhance the compiler to automatically add code that performs bound checks prior to each memory access, allowing verification to pass. As we note in Section 3.4, the bounds checking is done using masking and not branches to ease verification.

Context caching. Our compiler extension introduces intrinsics to get a pointer to the context or to read its data. The context is frequently needed along the callback for calling helper functions and for translating GVAs. Delivering the context as a function parameter requires intrusive changes and can prevent sharing code between the guest and its hyperupcall. Instead, we use the compiler to cache the context pointer in one of the registers and retrieve it when needed.

3.3 Registration

After a hyperupcall is compiled into eBPF bytecode, it is ready to be registered. Guests can register hyperupcalls at any time, but most hyperupcalls are registered when the guest boots. The guest provides the hyperupcall event ID, hyperupcall bytecode and the virtual memory the hyperupcall will use.

3.4 Verification

The hypervisor verifies that each hyperupcall is safe to execute at registration time. Our verifier is based on the Linux eBPF verifier and checks three properties of the hyperupcall: memory accesses, number of runtime instructions, and helper functions used.

3.5 Execution

Once the hyperupcall is compiled, registered and verified, it may be executed by the hypervisor in response to an event. There are some complexities to executing hyperupcalls, for accessing remote CPU states and dealing with locks. In general, the hypervisor can run the hyperupcall to obtain information about the guest without waiting on a response from the guest.

4 Use Cases and Evaluation

Previously, we presented several use cases for hyperupcalls which primarily involved the guest operating system: enabling a hypervisor to be proactive about resource allocation when memory is overcommitted, enhancing performance when interprocessor interrupts (IPIs) are used, and increasing the security and debuggability of systems in virtual environments [4]. From the application perspective, we also provided a modified version of the `ftrace` utility to help application developers see both hypervisor and guest events in a unified view. In this section, we focus on the application use cases of hyperupcalls: first we present both the frace utility previously presented and a new use case where we enable memcached to install a hyperupcall to prioritize certain traffic. This enables the hypervisor to safely understand application traffic and perform actions based on application state without coupling the hypervisor with application code.

4.1 Unified Event Tracing

Event tracing is an important tool for debugging correctness and performance issues. However, collecting traces for virtualized workloads is somewhat limited. Traces collected inside a guest do not show hypervisor events, such as when a VM-exit is forced, which can have significant effect on performance. For traces that are collected in the hypervisor to be informative, they require knowledge about guest OS symbols [13]. Such traces cannot be collected in cloud environments. In addition, each trace collects only part of the events and does not show how guest and hypervisor events interleave.

To address this issue, we run the Linux kernel tracing tool, `ftrace` [34], inside a hyperupcall. `Ftrace` is well suited to run in a hyperupcall. It is simple, lockless, and built to enable concurrent tracing in multiple contexts: non-maskable interrupt (NMI), hard and soft interrupt handlers and user processes. As a result, it was easily be adapted to trace hypervisor events concurrently with guest events. Using the `ftrace` hyperupcall, the guest can trace both hypervisor and guest events in one unified log, easing debugging. Since tracing all events use only guest logic, new OS versions can change the tracing logic, without requiring hypervisor changes.

Tracing is efficient, despite the hyperupcallcomplexity (3308 eBPF instructions), as most of the code deals with infrequent events that handles situations in which trace pages fill up. Tracing using hyperupcalls is slower than using native code by 232 cycles, which is still considerably shorter time than the time a context switch between the hypervisor and the guest takes.

Tracing is a useful tool for performance debugging, which can expose various overheads [47]. For example, by registering the `ftrace` on the VM-exit event, we see that many processes, including short-lived ones, trigger multiple VM exits due to the execution of the `CPUID` instruction, which enumerates the CPU features and must be emulated by the hypervisor. We found that the GNU C Library, which is used by most Linux applications, uses `CPUID` to determine the supported CPU features. This overhead could be prevented by extending Linux virtual dynamic shared object (vDSO) for applications to query the supported CPU features without triggering an exit.

4.2 Scheduler Activation

Our scheduler activation hyperupcall prototype performs scheduler activation by increasing the virtual machine priority for a short time when a packet which matches a condition arrives to the guest. memcached registers a hyperupcall with the guest, which in turn registers a hyperupcall with the hypervisor on a guest packet receiving event. In our prototype implemetation, this hyperupcall simply boosts the VM priority using a helper function, but we could have also performed other actions such as inspect the packet contents or access guest data structures. We increase the VM priority (`cpu.weight`) from a default value of 10 to 100 for 100ms.

Figure 2 shows the results. We used a memcached server in a guest with a single VCPU and increased the level of overcommittment of the physical CPU the machine was running on. The guest and multiple background tasks (iperf) were executed in different `cgroups` with equal priority. We assume in our experiment that the same user owns all the different guests. A workload generator (memcslap) was dispatched every second to issue 100 requests to the VM. Each experiment as conducted 50 times and the average execution time and standard deviation are shown.

Overall, this use case demonstrates that an application can use hyperupcalls to prioritize requests to the guest in an application-specific manner, greatly reducing latency and variability. We believe that we have only scratched the surface, and there are many other use cases of hyperupcalls which can be used to enhance applications. Since hyperupcalls can seamlessly integrate into the codebase of the application and are able to access guest state, developing new use cases is greatly simplified. We are currently working on making it easier for the hyperupcall to access application state, as well as methods for the guest OS to safely register multiple hyperupcalls on the same event.

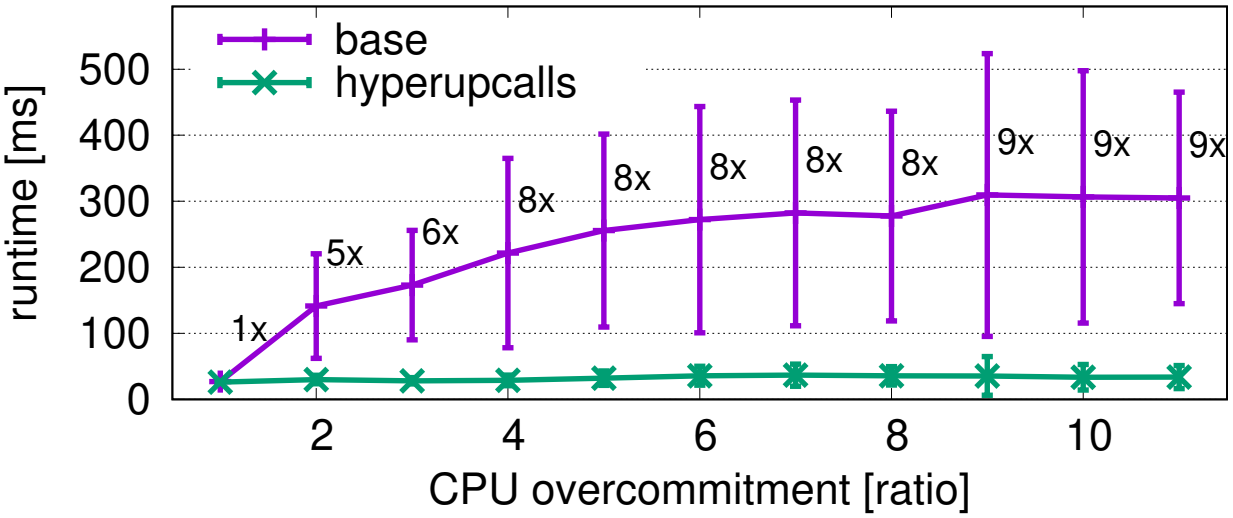


Figure 2: The runtime of 100 memcached requests with varying level of CPU overcommitment, with and without scheduler activation hyperupcalls. The slowdown ratio is presented above the lines.

5 Availability

We are in the process of open sourcing the hyperupcalls infrastructure and intend to make it available to the public soon. In the meantime, interested parties may contact the authors to obtain a pre-release version. Currently, the hyperupcalls infrastructure we intend to release is built for Linux virtual machines, and is integrated into the source code of the Linux kernel. Additional work is necessary to fully expose this interface to other applications.

There are several contexts which the hyperupcalls infrastructure could be used in other applications. An application wishing to install its own hyperupcalls, as in the example use cases we developed, could do so using an interface provided by the operating system. However, the operating system would need to have a mechanism for multiplexing hyperupcalls registered to the same event, or not support multiplexing at all. Another context that hyperupcalls could be used by applications is for applications to use the hyperupcalls concept of running verified trusted code. Our infrastructure does not yet support this, but could serve as an example system for leveraging eBPF in applications to run code to bridge the semantic gap.

6 Conclusion

Bridging the semantic gap is critical for performance and for the hypervisor to provide advanced services to guests. Hypercalls and upcalls are currently used to bridge the gap, but they have several drawbacks: hypercalls cannot be initiated by the hypervisor, upcalls do not have a bounded runtime, and both incur the penalty of context switches. Introspection, an alternative which avoids context switches can be unreliable as it relies on observations instead of an explicit interface. Hyperupcalls overcome these limitations by allowing the guest to expose its *logic* to the hypervisor, avoiding a context switch by enabling the hyperupcall to safely execute guest logic directly.

We have built a complete infrastructure for developing hyperupcalls which allow developers to easily add new paravirtual features using the codebase of the OS. This infrastructure could easily be extended to be used by applications as well, enabling applications to provide the hypervisor insight into their internal state. They can also be used to bridge the semantic gap outside of virtualization, for example, for services such as databases to gain more insight about the applications which use them.

References

[1] Ferrol Aderholdt, Fang Han, Stephen L Scott, and Thomas Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 414–423, 2014.

[2] Nadav Amit. Patch: Wrong size-extension check in BPFDAGToDAGISel::SelectAddr. `https://lists.iovisor.org/pipermail/iovisor-dev/2017-April/000723.html`, 2017.

[3] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[4] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (ATC)*, pages 97–112, 2018.

[5] Nadav Amit, Michael Wei, and Cheng-Chun Tu. Hypercallbacks: Decoupling policy decisions and execution. In *ACM Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 37–41, 2017.

[6] Kapil Arya, Yury Baskakov, and Alex Garthwaite. Tesseract: reconciling guest I/O and hypervisor swapping in a VM. In *ACM SIGPLAN Notices*, volume 49, pages 15–28, 2014.

[7] Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using Xen, OpenVX, and Xenserver. In *IEEE International Conference on Advances in Computing and Communications (ICACC)*, pages 247–250, 2014.

[8] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *IEEE Symposium on Reliable Distributed Systems*, pages 82–91, 2010.

[9] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5):670–684, 2011.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[11] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. *ACM SIGOPS Operating Systems Review (OSR)*, 29(5):267–283, 1995.

[12] Big Switch Networks. Userspace eBPF VM. `https://github.com/iovisor/ubpf`, 2015.

[13] Martim Carbone, Alok Kataria, Radu Rugina, and Vivek Thampi. VProbes: Deep observability into the ESXi hypervisor. VMware Technical Journal `https://labs.vmware.com/vmtj/vprobes-deep-observability-into-the-esxi-hypervisor`, 2014.

[14] Andrew Case, Lodovico Marziale, and Golden G Richard. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010.

[15] Tzi-cker Chiueh, Matthew Conover, and Bruce Montague. Surreptitious deployment and execution of kernel agents in windows guests. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 507–514, 2012.

[16] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. *Exokernel: An operating system architecture for application-level resource management*, volume 29. 1995.

[17] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *IEEE Symposium on Security and Privacy (SP)*, pages 586–600, 2012.

[18] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *The Network and Distributed System Security Symposium (NDSS)*, volume 3, pages 191–206, 2003.

[19] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 147–156, 2011.

[20] Nur Hussein. Randomizing structure layout. https://lwn.net/Articles/722293/, 2017.

[21] IO Visor Project. BCC - tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc, 2015.

[22] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2006.

[23] Andi Kleen. Linux virtual memory map. Linux-4.8:Documentation/x86/x86_64/mm.txt, 2004.

[24] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. *Pre-virtualization: Slashing the cost of virtualization*. Universität Karlsruhe, Fakultät für Informatik, 2005.

[25] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472, 2013.

[26] Microsoft. Hypervisor top level functional specification v5.0b, 2017.

[27] Aleksandar Milenkoski, Bryan D Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. Experience report: an analysis of hypercall handler vulnerabilities. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111, 2014.

[28] Preeti Mishra, Emmanuel S Pilli, Vijay Varadharajan, and Udaya Tupakula. Intrusion detection techniques in cloud environment: A survey. *Journal of Network and Computer Applications*, 77:18–47, 2017.

[29] Quentin Monnet. Rust virtual machine and JIT compiler for eBPF programs. https://github.com/qmonnet/rbpf, 2017.

[30] George C Necula. Proof-carrying code. In *ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL)*, pages 106–119, 1997.

[31] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304, 2011.

[32] Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. Ibmon: monitoring VMM-bypass capable infiniband devices using memory introspection. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, pages 25–32, 2009.

[33] Dannies M. Ritchie and Ken Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[34] Steven Rostedt. Debugging the kernel using Ftrace. LWN.net, http://lwn.net/Articles/365835/, 2009.

[35] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):95–103, 2008.

[36] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen hypervisor. *BlackHat USA*, 2008.

[37] Jiangyong Shi, Yuexiang Yang, and Chuan Tang. Hardware assisted hypervisor introspection. *SpringerPlus*, 5(1):647, 2016.

[38] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In *ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV)*, pages 45–48, 2016.

[39] VMware. open-vm-tools. https://github.com/vmware/open-vm-tools, 2017.

[40] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 27, pages 203–216, 1994.

[41] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review (OSR)*, 36(SI):181–194, 2002.

[42] Gary Wang, Zachary John Estrada, Cuong Manh Pham, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Hypervisor introspection: A technique for evading passive virtual machine monitoring. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[43] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 33–47, 2014.

[44] Rafal Wojtczuk. Analysis of the attack surface of Windows 10 virtualization-based security. BlackHat USA, 2016.

[45] Britta Wuelfing. Kroah-Hartman: Remove Hyper-V driver from kernel? Linux Magazine `http://www.linux-magazine.com/Online/News/Kroah-Hartman-Remove-Hyper-V-Driver-from-Kernel`, 2009.

[46] Xen Project. XenParavirtOps. `https://wiki.xenproject.org/wiki/XenParavirtOps`, 2016.

[47] Haoqiang Zheng and Jason Nieh. WARP: Enabling fast CPU scheduler development and evaluation. In *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*, pages 101–112, 2009.

Operating Systems Support for Data Management on Modern Hardware

Jana Giceva
Imperial College London
j.giceva@imperial.ac.uk

Abstract

For decades, database management systems have found the generic interface, policies and mechanisms offered by conventional operating systems at odds with the need for efficient utilization of hardware resources. The existing approach from the OS-side of "one-size-fits-all" interface and policies fails to meet modern data management workload's performance expectations, and the "overwriting the OS policies" approach from the DB-side does not scale with the increasing complexity of modern hardware and deployment trends.

In this article, we present two approaches on how to improve the systems support for database engines. First, we extend the OS with a policy engine and a declarative interface to improve the knowledge transfer between the two systems. Such extensions allow for easier deployment on different machines, more robust execution in noisy environments and better resource allocation without sacrificing performance guarantees. Second, we show how we leverage a novel OS architecture to develop customized OS kernels that meet the needs of data management workloads. Finally, we discuss how both approaches can help to address the pressing challenges for data processing on heterogeneous hardware platforms.

1 Introduction

Generally, today's operating systems multiplex applications with little to no information about their requirements. They migrate, preempt, and interrupt threads on various cores, trying to optimize some system-wide objectives (e.g., load balancing the work queues on individual cores and across the NUMA nodes [27]). As such, the OS has no notion about how its decisions affect the performance of the applications primarily due to the limited communication between the two layers [15].

As a result, database engines that run on commodity operating systems often experience performance problems, which are caused by the generic OS policies [44]. First, when executing in a noisy environment alongside other applications, the default OS policies for resource management can often cause performance degradation [19] or inefficiencies in resource usage [16, 27]. Second, even when running in isolation, databases often override the generic OS policies (e.g., by pinning threads to cores, allocating memory from a particular NUMA node, or pinning pages to avoid swapping, etc. [37, 22]). The problem with such user-side optimizations is that they are often tailored to a specific architecture, which makes portability to other platforms a daunting

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

task [28, 41]. Third, frequently the applied optimizations are fragile as they rely on assumptions on what the OS kernel mechanisms and policies do (e.g., HyPer [21] leverages an efficient OS-assisted snapshotting). Consequently, any change in the OS policies can cause performance bugs that are difficult to identify and debug.

In light of modern hardware trends of increased hardware heterogeneity and machine diversity, pushing all the complexity up to the developer or within the database engine does not scale. Furthermore, as databases are often deployed in the cloud, alongside other applications and tasks, they can no longer assume to have full ownership of the underlying machine's resources and any scheduling decision they do may be at odds with the noisy system environment and result in unpredictable performance.

In this article we argue that it is time to revisit the interface between operating systems and databases and address the modern challenges in a holistic manner crossing various layers across the systems stack. More specifically, we propose a solution that first addresses the semantic gap that exists between the database engine and the operating system by leveraging (1) a powerful declarative interface between the two layers allowing for bi-directional information flow, and (2) an OS policy engine that unifies the knowledge present in the database (workload characteristics, access patterns, cost models, data distribution, etc.) with the knowledge of the OS about the underlying hardware and the runtime system-state. Furthermore, we present a novel OS architecture that allows for OS kernel customization (i.e., policies, mechanisms and services) based on the specific requirements of the database system or its workloads. Our design is inspired by recent advancements in operating systems, which enable systems to run a specialized kernel on a subset of the resources on a given machine. This enables the database to get considerably more control over the full OS stack, which can then be tuned to achieve both better performance and stronger guarantees. Finally, we argue how both design principles are suitable to target modern hardware resource dis-aggregation challenges, raising a few interesting research directions.

2 Background

Databases and operating systems have a decades-long conflict when it comes to resource management and scheduling. Even though they initially started with the same goal – providing efficient access of data written in files – they took different approaches to addressing the problem. For many years this was not perceived as an issue as the two systems were targeting different workloads and machines. This shaped the role of monolithic databases and operating systems as we know them today. However, the economic advantage of off-the-shelf hardware has led to today's situation where a database runs on top of conventional OS. The key problem is that the OS works with very little knowledge about the workload requirements and properties. Its primary role is to schedule resources among multiple applications and to provide isolation protection. As such, it sees the database as yet another program and offers the same generic mechanisms and policies, which often lead to sub-optimal performance numbers.

Recent trends in both hardware architectures and resource dis-aggregation over fast network interconnects as well as economies of scale and deployment in the cloud are pressing both layers of the system stack to rethink their internal designs. The last decade, in particular, has seen profound changes in the available hardware as we reached the power wall limitation and CPU frequencies stopped scaling. In response, hardware architects introduced multiple cores, heterogeneous compute resources and accelerators. Similarly, with the rise of the memory wall and the gap between DRAM and CPU frequencies, machines emerged with more complex cache hierarchies, non-uniform cache coherent memory, etc. Consequently, the system software (both DBs and OSs) has to adapt and embrace the new hardware landscape as an opportunity to rethink its architecture model and design principles. For example, to improve performance, novel scheduling decisions within a database engine [24, 37] and certain relational algorithms has shifted towards hardware awareness in modern machines [36, 5, 46, 32]. Optimal use of resources today requires detailed knowledge of the underlying hardware (e.g., memory affinities, cache hierarchies, interconnect distances). Absorbing such complexity has now become the burden of the programmer and the problem gets further aggravated with the increasing diversity of micro-architectures.

On the deployment side, in the age of cloud and server consolidation, databases can no longer assume to have a complete physical machine to themselves. They are increasingly more often deployed and offered as services on the cloud, where they run alongside other applications. Consequently, the carefully constructed internal model of machine resources a database typically uses to plan the execution of its query plans and physical relational operators has become highly dependent on the runtime state of the whole machine. This state, however, is unknown to the database and is only available in the operating system which has an overview of all active applications and orchestrates the resource allocation.

In that context, we make the following observations. First, databases can no longer take full ownership of resource management, allocation and scheduling, partly because of increasing hardware complexity and portability issues, and partly because databases today are running in noisy environments (e.g., in the cloud), alongside other applications. Second, there is a big semantic gap between what each layer of the system stack knows – the database engine about its workload properties and requirements and the operating systems about the underlying hardware and runtime system-state – and the rigid interface between them does not allow for rich information flow. Fourth, the one-size-fit-all generic policies and mechanisms offered by the OS for a wide range of workloads do not work for performance sensitive applications, like data processing engines. And fifth, the heavy OS stack is no longer suitable for the new generation hardware, with heterogeneous (rack-scale) resource dis-aggregation. These are the issues we address as part of our work and discuss in the article.

3 Overview of proposed solution

More specifically, we propose customizing the operating system for data-processing applications and enriching the interface between the two layers to allow for better information flow. This way the operating system can adjust its allocation policies while reasoning about the workload requirements in addition to its optimization for system-wide objectives. To achieve that, we built a proof-of-concept system that makes the following contributions:

First, we show how the semantic gap between data processing engines and the operating system can be avoided by introducing a declarative interface for mutual information exchange. To do that we explored how to best integrate some of the extensive knowledge that a database engine has about its workload requirements (e.g., cost models, data dependency graphs, resource requirements, etc.) into the OS. The goal is to enable the OS to reason both about the particular requirements and properties of the database and about the system-wide and runtime view of the hardware platform and the current application mix. We achieve that by introducing a policy engine in the OS and a resource monitor that facilitates the communication between the two layers. A rich *query-based interface* then enables any application (including the database) to interact with the policy engine. More specifically, it allows the database to (i) query for details about the underlying hardware resources, (ii) rely on the policy engine to absorb the hardware complexity and diversity, and provide suitable deployment decisions, and (iii) push database-specific logic down to the OS in the form of stored procedures that enables it to adjust and react to noisy system environments (§ 4). The system architecture is shown in Figure 1.

Second, inspired by the multikernel OS design [6], in § 5 we propose a novel OS architecture that enables dynamic partitioning of the machine's resources (e.g., CPUs, memory controllers, accelerators, etc.) into a *control plane*, running a full-weight operating system along with an OS policy engine, and a *compute plane*, consisting of specialized light-weight OS stacks. The objective is to enable customization of the compute-plane OS both for the properties of the underlying hardware (i.e., potentially heterogeneous compute units) and for the specific requirements of the workload (e.g., customized scheduler or memory management). By design the allocation of resources between the control and compute plane is dynamic and can be adapted at runtime based on the changing workload requirements. To demonstrate the benefits of such control-compute plane OS architecture, we present a light-weight OS with a kernel-integrated runtime (Basslet), which we run on the compute plane, that is customized for parallel data analytics.

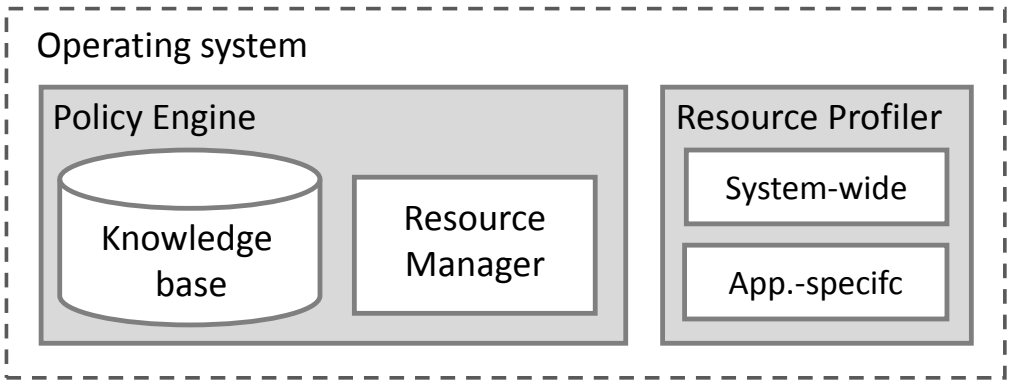


Figure 1: Policy Engine

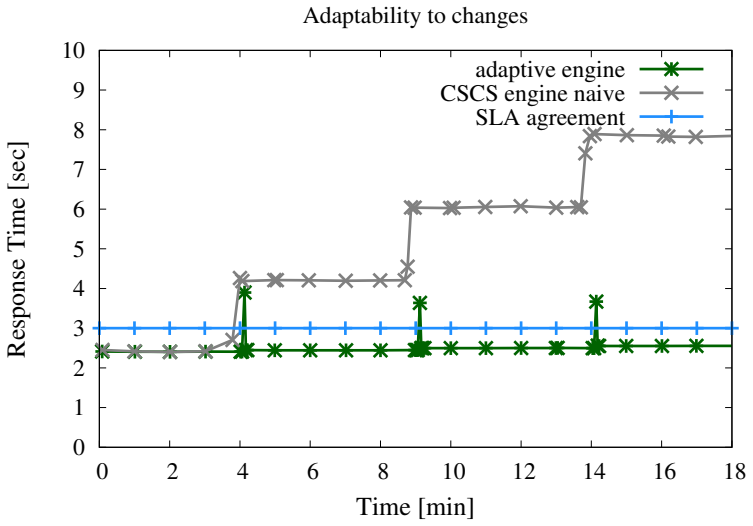
4 OS policy engine

The OS policy engine is designed to enable both the OS itself and the database running on top to better grasp the properties of the available hardware resources and reason about the real-time system state.

More specifically, it consists of a *knowledge base* that contains information about (1) machine specific facts (e.g., topology of the machine, number of cores and memory per NUMA node, the cache hierarchy, etc.), (2) application-related facts (e.g., information whether an application is compute- or memory-bound, sensitive to sharing the caches, etc.), and (3) current system-state (e.g., number of active applications, their memory usage, etc.). This information is used both by the knowledge base to build a detailed model of the machine and its resources, and by a set of algorithms and solvers that compute resource allocation schedules. For the knowledge base we borrow and extend the concept of System Knowledge Base (SKB) from the Barrelfish OS [42, 1], which stores data in the format of free-form predicates in a Constraint Logic Programming (CLP) engine. This enables various solvers to reason about the information available by issuing logical queries to perform constraint solving and optimization.

The *resource manager* is responsible for communicating with the applications (in our case the database engine), triggering resource allocation computations in the knowledge base, and executing the decided policies by invoking the available OS mechanisms. Often, the policy engine relies on a *resource profiler* to measure the capacities of hardware resources (e.g., the maximum attainable DRAM bandwidth achievable per NUMA node), monitor their current utilization, and enable applications to measure their resource requirements or footprints.

Finally, the new *interface* is declarative and allows for richer two-way information exchange between the DBMS and the OS policy engine. It covers a wide range of actions from retrieving information about the underlying architecture to pushing down application-specific cost-models and dataflow dependencies so that the OS policy engine can reason about them. Furthermore, by allowing stored procedures it enables database-specific logic to be computed in the OS-side that leverages the most up-to-date system-state. Finally, it supports the retrieval of application-specific resource usage profiles as measured by the resource profiler and enables a continuous information flow between the two layers at runtime in the form of notifications and updates. This way the OS can do a better job when deploying the application's threads onto a range of different machines and provide efficient resource allocation without affecting the application's performance of predictability; and the database can adapt itself based on the current system state, which is especially important in noisy environments.



(a)

Scheduler	CPUs	Tput	Response Time[s]		
		[WIPS]	50th	90th	99th
Depl Alg.	6	428	15	23	36
SharedDB	44	425	14	22	36
Linux	48	317	8	72	82

(b)

Figure 2: (a) Adaptability to noisy environments. (b) Deployment of complex query plan

Examples

We demonstrate the benefits of the policy engine and importance of leveraging unified knowledge from both the database engine and the operating system with two different examples.

Use-case 1: Adaptability to noisy environments

In the first use-case we show how a storage engine (that we call CSCS) can be adjusted to interact more closely with the OS policy engine to achieve good performance and maintain predictable runtime even in dynamic environment where other applications enter the system and begin using resources [15]. Before starting the execution, the storage engine communicates with the policy engine its properties (i.e., its scan threads are CPU-bound, cache- and NUMA-sensitive, the SLO for latency is 3ms, etc.), a cost function that calculates the scan time given the specific workload properties on a particular machine, and a stored procedure for data redistribution among the remaining scan threads whenever a CPU core resource is revoked at the expense of a newly entered task in the system.

When a new task enters the system it registers with the resource manager and asks for a CPU core. The resource manager notifies the knowledge base of the new changes and triggers a re-computation of the resource allocation plan. In this re-computation the policy engine checks that even if it takes away a core from the storage engine, the scan time is still going to be below the runtime SLO and allocates one of the cores to the new application. The database storage engine is notified of the change and invokes the stored procedure to decide how to redistribute the data that was scanned by the thread that just lost its CPU core. The stored procedure retrieves information about the remaining available cores and checks the availability of memory on the corresponding

NUMA nodes. It then redistributes the data to the chosen threads and resumes execution. In Fig. 2a we compare the behaviour of a naive CSCS storage engine that does not react to changes in the system state and continues executing as if nothing happened (despite another CPU-intensive task entering the system every 4 minutes and pinning its thread to core 0 where a scan thread runs) leading to significant reduction in response time. The adaptive-engine line shows how the storage engine behaves when coordinating with the policy engine. Its response time remains relatively steady even in the presence of other tasks with spikes observed at the time when a new task enters the system. We explain the spikes as a result of the storage engine redistributing the data to the other cores, as suggested by the stored procedure. Nevertheless, even when losing a scan thread (core), the storage engine can easily resume executing with a latency well within the required SLA requirements.

Use-case 2: Efficient deployment on multicore machine

The second use-case demonstrates the benefits of using (1) the *resource profiler* to capture the resource requirements of database operations, (2) the *OS policy engine* and its knowledge of the underlying machine model and (3) the *DB engine*'s knowledge of the data-dependency graph between relational operators in a complex query plan, to compute a close to optimal deployment of the query plan on a given multicore machine [14].

Good resource management and relational operator deployment requires awareness of the thread's resource requirements [3, 29, 26]. As a result of tuning the algorithm's implementation to the underlying hardware, databases have also become more sensitive to the resources they have at hand and poor scheduling can lead to performance degradation [23, 15]. In order to capture the relevant characteristics for application threads, the resource monitor generates so-called *resource activity vectors* (RAVs). At present, they capture the usage of the most important resources (CPU and memory bandwidth usage), but can be easily extended to other resources when needed (e.g., network I/O utilization, cache sensitivity, etc.). The approach was inspired by the notion of activity vectors, initially proposed for energy-efficient scheduling on multicore machines [30].

The deployment algorithm for a given query plan runs in the OS policy engine and aims to minimize the computational- and bandwidth- requirements for the query plan, provide NUMA-aware deployment of the relational operators and enhance data-locality. As input, it uses (1) the data-dependency graph of the relational operators as provided by the database engine, (2) the RAVs for each operator as generated by the resource monitor, and (3) a detailed model of the underlying machine as kept in the OS policy engine. The algorithm consists of four phases, where the first two compute the required number of cores (corresponding to the *temporal scheduling* sub-problem), the third phase approximates the minimum number of required NUMA nodes and the fourth phase computes the final placement of the cores on the machine so that it minimizes DRAM bandwidth usage – the *spatial scheduling* sub-problem.

We evaluated the effectiveness of the algorithm by deploying a TPC-W global query plan as generated by SharedDB [13] (with 44 relational operators) on the AMD Magnycore machine (four 2.2 GHz AMD Opteron 6174 processors and 128 GB RAM, each processor has two 6-core dies, or 48 cores in total). We compare the performance of running the workload against two baselines: (1) using the default Linux scheduler and (2) using the standard operator-per-core deployment used by systems like SharedDB to provide guarantees for predictable performance and tail latencies. The results are shown in Tab. 2b. The presented values for average throughput and latency percentiles (50th, 90th, and 99th) show that the performance of the system was not compromised by the significant reduction in allocated resources (44 for SharedDB default scheduler down to 6 cores for our algorithm), which is important for databases and their SLOs. Please note that the performance of the query plan when the Linux scheduler was in charge of the deployment is poorer in both absolute throughput performance and stability than the other two approaches. This is because the OS can use all 48 cores on the machine and often migrates threads around based on some system-wide metric which leads to higher tail latencies.

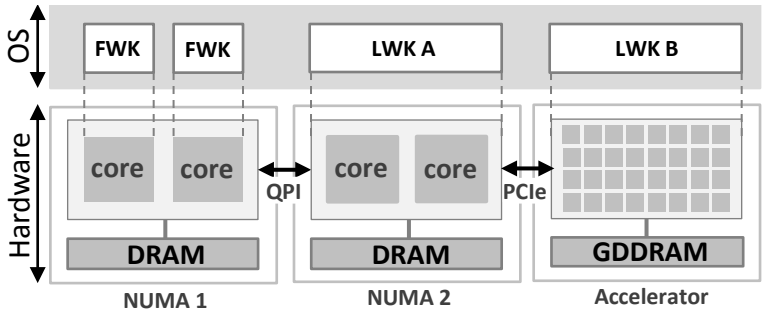


Figure 3: Illustrating Badis – an adaptive OS architecture, based on the multikernel model. The cores on NUMA node 1 each execute a separate kernel of the full-weight kernel (FWK). The cores on NUMA node 2 execute a common instance of a specialized light-weight kernel (LWK A). The computation units on the HW accelerator run a different version of the light-weight kernel – optimized for the particular hardware platform (LWK B).

5 Customized OS

In the previous section we showed the benefits of using the unified knowledge from both the DB engine and the OS policy engine. While the design can bring significant advantages in a noisy environment and when scheduling jobs on a multicore system, it still does not alter the fact that the resource manager of the OS policy engine needs to use the full-blown OS s tack with all of its generic mechanisms. Recent advancements in operating system design enable us to configure and specialize the OS system stack (i.e., apply changes in both kernel- and user-space) for particular workload classes. Some new operating systems are based on the multikernel design [6], which run a separate kernel on every core [1, 47, 9]. In the Barrelfish OS the state on each core is (relatively) decoupled from the rest of the system – a multikernel can run multiple different kernels on different cores [49].

5.1 Novel OS architecture and the case for customized kernels

The novel OS architecture (Badis) we propose leverages the flexibility of the Barrelfish multikernel design that enables us to have an optimized lightweight OS co-exist in the same system as other general purpose OS kernels. We show the design in Fig. 3. In a nutshell, Badis splits the machine's resources into a *control plane* and a *compute plane*. The control plane runs the full-weight OS stack (FWK), while the compute plane consists of customized lightweight kernels (LWKs). The compute plane kernels provide selected OS services tailored to a particular workload and a noise-free environment for executing jobs on behalf of applications whose main threads run on the control plane's FWK. Additionally, as we discuss later, Badis' modular design makes it suitable to address HW heterogeneity and enables OS customization for different compute resources.

To demonstrate the benefits, we designed and implemented a customized OS stack for executing parallel analytical workloads. Even though, in our work we identified multiple opportunities for improvement of resource management and scheduling (e.g., for CPUs, memory, and various hardware devices) [16], in the first prototype we focused primarily on managing CPU resources. More specifically, for parallel analytical workloads we identified the following requirements:

- The need for *run-to-completion* tasks, which is important for both synchronization-heavy workloads, where preemption can lead to the well-know convoy effect [8], and data-intensive jobs that are cache-sensitive, where preemption can often lead to cache pollution and expensive DRAM accesses. In one of our experiments, we measured the indirect cost of a context-switch on machines with large last-level caches to be as expensive as 6ms, which is equivalent to the quantum on modern Linux schedulers.

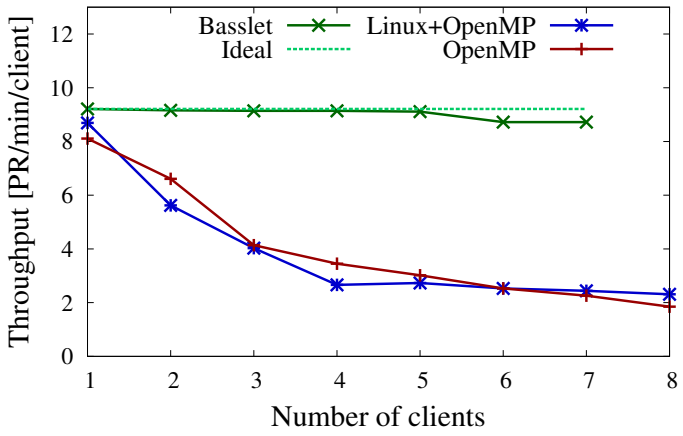


Figure 4: Throughput scale-out when executing multiple `PR`s using internal OpenMP parallelism vs. Linux+OpenMP scheduler vs. kernel-integrated runtime, as compared to ideal scale-out.

- The need for *co-scheduling* for a group of threads that work on the same operation and especially for data processing workloads that have synchronization steps where a single straggler can impact performance.
- The need for *spatial isolation*. In particular, when running in a noisy environment alongside other application threads which also use the memory subsystem. Such interaction can often result in destructive resource sharing [23, 45]. Hence, we claim that there is more to allocation than just cores and one should also account for other resources such as shared caches and DRAM bandwidth. Given the properties of modern multicore machines one such hardware island [38] is a NUMA node.

5.2 Implementation and evaluation

To achieve those requirements in the customized OS kernel, we proposed extending the UNIX-based process model to also support OS task and ptask (parallel task) as program execution units. This way the database can explicitly specify that a job needs to be executed without preemption – OS task, or that a pool of user-level threads that execute a common parallel job should be co-scheduled until completion – OS ptask. We implemented it as part of a kernel-based runtime, which can execute parallel analytical jobs on behalf of the data processing engine. Each customized kernel is spawned on a separate NUMA node (hardware island) for spatial isolation. As per design, the light-weight kernels run on the compute plane, while the FWK on the control plane offers a traditional thread-based scheduling. The boundary between the two planes, as well as the type of compute-plane kernels, can be changed at runtime depending on the workload mix requirements. Note that the cost of switching a kernel is as expensive as a context switch [49]. Such a dynamic architecture makes the system's stack suitable for scheduling hybrid workloads (e.g., operational analytics), where different kernels can co-exist at the same time, each one customized for a particular workload.

To demonstrate that not only database engines can benefit from such a customized kernel integrated runtime, we evaluated the system with GreenMarl [18], a graph application suite running on OpenMP. More specifically, we execute PageRank (PR) on the LiveJournal graph, which is the largest available social graph from the SNAP dataset [25]. The experiment evaluates the efficiency of using the customized compute plane kernel compared to the performance of the same workload, when executed using either the default OpenMP or the Linux scheduler. All experiments were ran on the same AMD Magnycours machine as before. The workload is as follows: we measure the performance when a single client runs a PageRank algorithm on one NUMA node – 6 cores. For

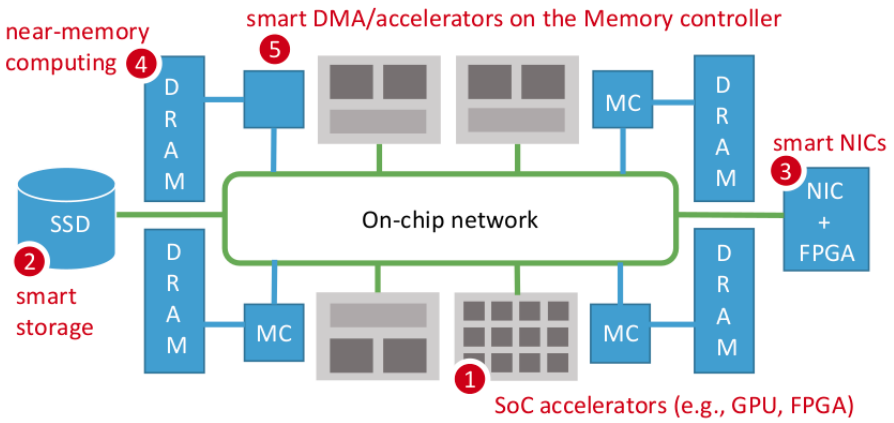


Figure 5: Active heterogeneous hardware

every subsequent client (i.e., another instance of PR) we allow the system to use additional 6 cores of another NUMA node. The response variable is the throughput as perceived per client (i.e., the inverse of the total time needed to execute all PR jobs). The results are presented in Fig. 4. They show that the interference among the clients when OpenMP or OpenMP+Linux schedule the resources, increases as we add more clients despite having sufficient resources (there are 8 NUMA nodes and at most 8 PRs running in the system). In contrast, when using the kernel integrated runtime we can achieve almost linear per-client throughput scale-out until seven clients. The final six cores, belonging to the first NUMA node, are dedicated for the control plane.

While the discussion here focused on managing CPU resources for analytical workloads, in [16] we also discuss opportunities for managing memory as well as providing more transparent access to other devices.

5.3 Related work

The HPC community has long argued that their workloads are sensitive to OS "noise" when working at large scale [17]. Thus, they proposed using light-weight kernels [39, 20, 12] that are customized for sensitive applications. Researchers have also explored the design space of multikernel-based OSes by having the specialized light-weight kernels run alongside full-weight kernels like Linux inside the same system [34, 48, 11].

While our prototype is implemented over a multikernel, customized new schedulers can be applied on Linux if we leverage some recent proposals for fast core reconfigurability [33]. Currently, in our work we have not directly addressed I/O issues, but the architecture allows to easily integrate the control/data plane ideas proposed by systems like Arrakis [35] and IX [7]. Similar approaches were also explored in Solros [31] for workloads with high disk and network bandwidth requirements when running on co-processors.

6 Future outlook and research directions

In this section, we look at recent and future developments of hardware technologies and deployment trends. We argue for a holistic solution across the system stack (from the data processing layer, down to runtime and operating systems and eventually hardware) in order to efficiently address the coming challenges and hide the increasing complexity from the developer's side.

Modern machines have an abundance of hardware heterogeneity and they are only going to get more diverse. In Fig. 5 we show all the places where *active hardware* components can be found today in addition to the regular CPUs: (1) system-on-chip accelerators like GPUs and FPGAs, (2) smart storage mediums (e.g., smart SSDs), (3) programmable NICs or NICs with an FPGA attached, as bump in the wire, offloading compute (4) to where the

data sits (e.g., near-memory computing [2]) or (5) as the data moves between DRAM and the processor's cache (e.g., accelerators on the memory controller [4]), etc.

Despite this outlook, today's commodity operating systems still hide the underlying hardware complexity and diversity as much as possible from the applications running above. While this made sense in the past, such an approach today is very restrictive and leads to under-utilization of the available hardware capacity. We argue that the Badis OS architecture is well-suited for such hardware platforms – as opposed to treating all the active components as devices with external drivers (as with GPGPUs [40] or NICs [35, 7]), we should have the OS manage their computational capacities in the *control plane* and export the device services directly to the applications via customized *compute planes* [10]. Recent work in the OS community has also proposed extending the multikernel model for heterogeneous computing [43] and building data-centric OSs for accelerators [31].

Furthermore, it is important to note that the declarative interface between databases and operating systems becomes even more relevant in the case of hardware heterogeneity. Especially when a data-processing engine can offload part of the computation to an active compute component. Constructing cost-models that match the performance/cost metrics for using an accelerator and pushing down such information to the OS policy engine, can make the scheduling and resource management of these resources much more effective. If this is also accompanied with the data-dependency graph as we have shown for other hybrid systems [10], the underlying OS can absorb the complexity of memory management and task allocation on behalf of the developer and achieve much higher performance and more efficient resource usage.

Going beyond database engines, many machine learning, data mining and graph processing applications can benefit from similar cross-layer optimizations across the systems stack, including the operating system. We are currently exploring how such workloads can benefit by sharing information about their cost models or dataflow graphs to the OS policy engine when executing on heterogeneous computing platforms (e.g., TPUs or FPGAs).

7 Conclusion

The interaction between database engines and operating systems has been a difficult problem for decades, as both try to control and manage the same resources but with different goals. For long time, databases had the luxury to ignore the OS and overwrite the generic policies thanks to hardware homogeneity and over-provisioning of resources (i.e., running a database alone on a dedicated server machine). With the latest trends in hardware development (e.g.,from multicore to various accelerators) and workload deployment (e.g., multi-tenancy and server consolidation in the cloud), these assumptions are no longer valid. Hence, we argue that *now* is the time for a holistic approach that crosses multiple layers of the system stack and in particular one that revisits the interface between database management and operating systems.

In this article, as main problems we identified the knowledge gap that exists between the two systems and the rigid interface that does not allow for richer information flow as well as the generic policies offered by conventional operating systems for a wide range of applications. To address these issues we proposed Badis, an OS control- compute-plane architecture that allows for customization of the compute-plane OS stack for a particular workload or underlying hardware platform, and a powerful OS policy engine that resides on the control plane, which is able to reason about the database specific properties and requirements. With a series of experiments we demonstrated the benefits of the approach of unifying the knowledge of the two layers both for efficient deployment on modern machines and for maintenance of good and predictable performance in noisy environments. Looking forward, we believe that the proposed design principles are going to become even more relevant in the context of active hardware and resource dis-aggregation, and extend beyond the requirements of traditional data management workloads.

References

[1] Barrelfish Operating System, 2019. `www.barrelfish.org`, accessed 2019-01-20.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *ISCA '15*, pages 336–348, 2015.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB '99*, pages 266–277, 1999.

[4] K. Aingaran et al. M7: Oracle's next-generation sparc processor. *IEEE Micro*, 35(2):36–45, 2015.

[5] C. Balkesen. *In-memory parallel join processing on multi-core processors*. PhD thesis, ETH Zurich, 2014.

[6] 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 *SOSP*, pages 29–44, 2009.

[7] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. OSDI'14, pages 49–65.

[8] Blasgen, Mike and Gray, Jim and Mitoma, Mike and Price, Tom. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, 1979.

[9] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *SOSP*, pages 12–25, 1995.

[10] Daniel Grumberg. Customized OS kernel for data-processing on modern hardware, 2018.

[11] B. Gerofi, M. Takagi, Y. Ishikawa, R. Riesen, E. Powers, and R. W. Wisniewski. Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing. ROSS '15, pages 5:1–5:8, 2015.

[12] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. SC '10, pages 1–10, 2010.

[13] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: killing one thousand queries with one stone. *VLDB*, 5(6):526–537, Feb. 2012.

[14] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of Query Plans on Multicores. *PVLDB*, 8(3):233–244, 2014.

[15] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. COD: Database/Operating System Co-Design. In *CIDR '13*, 2013.

[16] J. Giceva, G. Zellweger, G. Alonso, and T. Rosco. Customized OS Support for Data-processing. In *The 12th International Workshop on Data Management on New Hardware*, pages 2:1–2:6, 2016.

[17] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. SC '10, pages 1–11, 2010.

[18] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362, 2012.

[19] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 263–276, 2015.

[20] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, catamount. In *In Cray User Group*, pages 16–19, 2005.

[21] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[22] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. SIGMOD '15, pages 691–706, 2015.

[23] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *PVLDB*, 2(1):373–384, Aug. 2009.

[24] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD '14*, pages 743–754.

[25] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[26] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR '13*, 2013.

[27] J. Lozi, B. Lepers, J. R. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: a decade of wasted cores. In *EuroSys'16*, page 1, 2016.

[28] D. Makreshanski, J. J. Levandoski, and R. Stutsman. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *PVLDB*, 8(11):1298–1309, 2015.

[29] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *PVLDB '00*, 9(3):231–246, 2000.

[30] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys '10*, pages 153–166, 2010.

[31] C. Min, W. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. Solros: A Data-centric Operating System Architecture for Heterogeneous Computing. In *EuroSys*, pages 36:1–36:15, 2018.

[32] I. Mueller. *Engineering Aggregation Operators for Relational In-memory Database Systems*. PhD thesis, Karlsruhe Institute of Technology (KIT), 2016.

[33] S. Panneerselvam, M. Swift, and N. S. Kim. Bolt: Faster Reconfiguration in Operating Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 511–516, Santa Clara, CA, July 2015. USENIX Association.

[34] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. W. Wisniewski. Fusedos: Fusing lwk performance with fwk functionality in a heterogeneous environment. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 211–218, Washington, DC, USA, 2012. IEEE Computer Society.

[35] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *OSDI*, pages 1–16, 2014.

[36] O. Polychroniou and K. A. Ross. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *SIGMOD*, pages 755–766, 2014.

[37] D. Porobic, E. Liarou, P. Tozun, and A. Ailamaki. ATraPos: Adaptive transaction processing on hardware Islands. In *ICDE '14*, pages 688–699, 2014.

[38] D. Porobic, I. Pandis, M. Branco, P. Tozun, and A. Ailamaki. OLTP on Hardware Islands. *PVLDB '12*, 5(11):1447–1458.

[39] R. Riesen, A. B. Maccabe, B. Gerofi, D. N. Lombard, J. J. Lange, K. Pedretti, K. Ferreira, M. Lang, P. Keppel, R. W. Wisniewski, R. Brightwell, T. Inglett, Y. Park, and Y. Ishikawa. What is a Lightweight Kernel? ROSS '15, pages 9:1–9:8, 2015.

[40] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *SOSP*, pages 233–248, 2011.

[41] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD '10*, pages 351–362, 2010.

[42] A. Schuepbach. *Tackling OS Complexity with Declarative Techniques*. PhD thesis, ETHZ, Dec. 2012.

[43] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *OSDI*, pages 69–87, 2018.

[44] M. Stonebraker. Operating System Support for Database Management. *Commun. ACM*, 24(7):412–418, July 1981.

[45] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *ISCA*, pages 283–294, 2011.

[46] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par 2011 Parallel Processing*, pages 160–169. Springer, 2011.

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

[48] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen. mOS: An Architecture for Extreme-scale Operating Systems. ROSS '14, pages 2:1–2:8, 2014.

[49] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling cores, kernels, and operating systems. In *OSDI*, pages 17–31, Oct. 2014.

The Glass Half Full: Using Programmable Hardware Accelerators in Analytics

Zsolt István
IMDEA Software Institute, Madrid, Spain
{first.lastname@imdea.org}

Abstract

There have been numeorus proposals to accelerate databases using specialized hardware in past years but often the opinion of the community is pessimistic: the performance and energy efficiency benefits of specialization are seen to be outweighed by the limitations of the proposed solutions and the additional complexity of including specialized hardware, such as field programmable gate arrays (FPGAs), in servers. Recently, however, as an effect of stagnating CPU performance, server architectures started to incorporate various programmable hardware components, ranging from smart network interface cards, through SSDs with offloading capabilities, to near-CPU accelerators. The availability of heterogeneous hardware brings opportunities to databases and we make the case that there is cause for optimism. In the light of a shifting hardware landscape and emerging analytics workloads, it is time to revisit our stance on hardware acceleration.

In this paper we highlight several challenges that have traditionally hindered the deployment of hardware acceleration in databases and explain how they have been alleviated or removed altogether by recent research results and the changing hardware landscape. We also highlight that, now that these challenges have been addressed, a new set of questions emerge around the integration of heterogeneous programmable hardware in tomorrow's databases, for which answers can likely be found only in collaboration with researchers from other fields.

1 Introduction

There is a rich history of projects aiming to specialize computers (or parts of computers) to databases. Notable examples include the Database Machine from the seventies [1], Gamma [2], the Netezza data appliance [3], the Q100 DB processor [4], and Oracle Rapid [5] most recently. These works demonstrate orders of magnitude increase in energy efficiency and better performance thanks to a hardware/software co-design approach. However, CPUs, until very recently, enjoyed a performance scaling in line with Moore's law and the time and effort of designing and delivering specialized hardware was not economical. This changed with the stagnation in CPU performance [6] and the simultaneous increase in networking speeds in the last decade that has created a clear need for hardware acceleration.

Initially, the move to the cloud worked against hardware acceleration for databases due to the cloud's reliance on commodity hardware and the need to cater to many different users and applications. Recently, however, new

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

data-intensive workloads emerged in the cloud (most notably machine learning), that suffered from stagnating CPU performance and could benefit from various types of compute or networking acceleration. If we look at today's cloud offering and datacenters, an exciting, heterogeneous landscape emerges: Machine learning workloads in the Google Cloud are accelerated with Tensor Processing Units (TPUs) [7], increasing energy efficiency by at least an order of magnitude when compared to GPUs. Amazon, Baidu and Huawei all offer Field Programmable Gate Arrays (FPGAs) by the hour in their cloud to users[1] to implement custom accelerators. Microsoft, in project Catapult [8], has been deploying FPGAs in the Azure Cloud to accelerate their infrastructure and machine learning pipelines. Furthermore, Intel has been experimenting with including small programmable elements on their Xeon CPUs [9] that can be tailored to accelerate different user applications.

These recent developments mean that multi-purpose programmable hardware accelerators are entering the mainstream and, from the point of view of the database, they can be exploited without having to incur additional deployment costs. Specialized hardware is most often used to accelerate compute-bound operations and the ongoing shift in the analytical database workloads towards machine learning [10][11][2] brings significantly more compute-intensive operations than the core SQL operators. What's more, there are proposals for using machine learning methods to replace parts of the decision making and optimization processes inside databases [12]. These emerging operators bring new opportunities in hardware acceleration both inside databases and for user workloads. Furthermore, now that hardware acceleration of real-world workloads is economically feasible, new challenges emerge around deep integration of programmable hardware in databases.

In this paper we make the case that there is cause for optimism, thanks to the two trends mentioned above, namely, datacenters becoming increasingly heterogeneous and workloads opening towards machine learning. These, combined with the state of the art in hardware acceleration for databases, tackle most of the past hindrances of programmable hardware adoption. We will focus on FPGAs as a representative example and discuss how several significant challenges have been alleviated recently. In the final part of this paper we highlight open questions around the topics of resource management and query planning/compilation in the presence of programmable hardware accelerators.

2 Background

2.1 Programmable Hardware in the Datacenter

The wide range of programmable hardware devices proposed and already deployed in datacenters can be categorized depending on their location with regards to the data source and CPU into three categories (see Figure 1): on-the-side, in data-path and co-processor.

The most traditional way we think about accelerators is as being *on-the-side* (Figure 1.1), attached to the processor via an interconnect, for instance PCIe. Importantly, in this deployment scenario the CPU owns the data and explicitly sends it to the accelerator, resulting usually in significant additional latency per operation due to communication latency and data transformation overhead. This encourages offloading operations at large granularity and without requiring back and forth communication between the CPU and the accelerator. GPUs are a common example of this kind of accelerator and were shown to be useful, for instance, to offload LIKE-based string queries [13]. There have also been proposals that deploy FPGAs this way for data filtering and decompression, e.g., in the work by Sukhwani et al. [14].

Another way of placing acceleration functionality in the architecture is *in data-path* (Figure 1.2). This can be thought of as a generalized version of near-data processing [15], and the goal of the accelerator is to filter or transform data at the speed that it is received from the data source. Designs that can't guarantee this could end up

[1] At the moment of writing it costs around $1.65/h to rent an Amazon EC2 F1 instance.

[2] For instance, Microsoft SQL Server now includes machine learning plug-ins. `https://docs.microsoft.com/en-us/sql/advanced-analytics/what-is-sql-server-machine-learning`

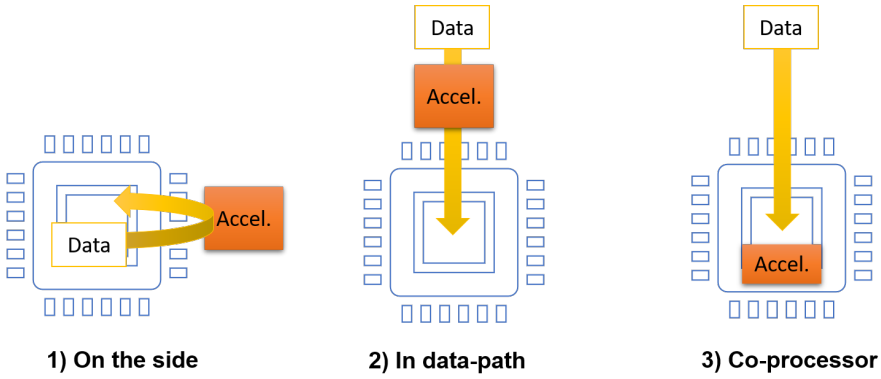


Figure 1: Programmable hardware accelerators can be deployed either as "on-the-side" accelerator (e.g., GPUs), as "in data-path" accelerator (e.g. smart NICs, smart SSD), or as co-processor (e.g. in Oracle DAX or Intel Xeon+FPGA).

slowing down the entire system [16]. Much of the research effort in this space has been centered around in-SSD processing [17][18], but more recently there have been efforts in using RDMA network interface cards (NICs) to accelerate distributed databases [20][21]. These NICs are limited to data manipulation acceleration, but there are efforts to make NICs and networking hardware in general more programmable [22]. This will allow offloading in the future complex, application-specific, operations.

The third deployment option, namely, *co-processor* (Figure 1.3), is also becoming increasingly available in the form of CPUs that integrate domain-specific or general-purpose programmable co-processors: The Oracle DAX [23] is an example of the former because it implements database-specific operations, such as data decompression, scan acceleration, comparison-based filtering, on data in the last level cache. Thanks to its specialized nature, it occupies negligible chip space and does not increase the cost of the CPU. As opposed to the DAX, the Intel Xeon+FPGA [9] platform offers an FPGA beside the CPU cores for general-purpose acceleration. The FPGA has high bandwidth cache-coherent access to the main memory and can be reprogrammed in different ways. This creates acceleration opportunities without the usual overhead of the on-the-side accelerators.

2.2 Field Programmable Gate Arrays

FPGAs are chips that can be programmed to implement arbitrary circuits and historically have been used to prototype and validate designs that would result later in Application-Specific Integrated Circuits (ASICs). They have recently become a target for implementing data processing accelerators in datacenters thanks to their flexibility (their role can change over time, as opposed to an ASIC) and orders of magnitude better energy efficiency than that of traditional CPUs [24]. FPGAs are composed of look-up tables (LUTs), on-chip memory (BRAM) and digital signal processing units (DSPs). All these components can be configured and interconnected flexibly, allowing the programmer to implement custom processing elements (Figure 2). It is not uncommon to have small ARM cores integrated inside the programmable fabric either, e.g., in Xilinx's Zynq product line.

FPGAs offer two types of parallelism: first, pipeline parallelism means that complex functionality can be executed in steps without reducing throughput. The benefit of FPGAs in this context is that the communication between pipeline stages is very efficient thanks to the physical proximity and availability of on-chip memory to construct FIFO buffers. The second type of parallelism that is often exploited on FPGAs is data-parallel execution. This is like SIMD (single instruction multiple data) processing in CPUs, but it can also implement a SPMD (single program multiple data) paradigm if the operations are coarser grained. What makes FPGAs interesting for acceleration is that these two types of parallelism can be combined even inside a single application module to provide both complex processing and scalable throughput.

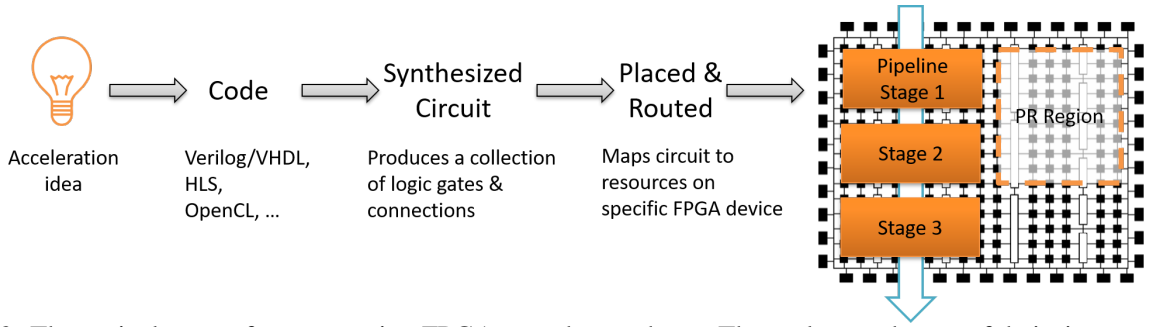


Figure 2: The typical steps of programming FPGAs are shown above. The tools spend most of their time mapping the synthesized circuit onto the FPGA. This is because the chip is composed of many programmable gates and memories that have to be configured and connected together in a 2D space, ensuring that signals can propagate correctly within clock periods.

As Figure 2 shows, FPGAs are programmed by synthesizing a circuit from a hardware definition language, such as Verilog or VHLD, and creating a "bitstream" for a specific device type that defines the behavior of every logic resource on the chip. This is an expensive step as it requires the tool to lay out the circuit on the "chip surface" and define connections and routing of these connections between circuit elements. Since FPGAs have flexible clocking options and the programmer is free to define a target frequency (e.g., 300MHz), the tools have to set up routing such that signals are propagated within the clock periods (which can become impossible with too high frequencies).

It is also possible to perform partial reconfiguration (PR), meaning that only a portion of the FPGA's resources are reprogrammed (illustrated on the right-hand side of Figure 2). This means that, for instance, in a database use-case a hardware-accelerated operator can be replaced with another one without having to bring the device offline. PR, however, comes with limitations: the regions can only be defined at coarse granularity, their size can't be redefined at runtime and their reprogramming requires milliseconds.

One important limitation of FPGAs is that all application logic occupies chip space and there is no possibility of "paging" code in or out dynamically. This means that the complexity of the operator that is being offloaded is limited by the available logic resources (area) on the FPGA. This also applies to the "state" of an algorithm that is often stored as data in the on-chip BRAM memories. These can be accessed in a single clock cycle, but if the data doesn't fit in the available BRAM, high latency off-chip DRAM has to be used.

3 Sources of Pessimism

Many early projects of FPGA-based database acceleration propose deploying them as on-the-side accelerators for row stores [14][25][26] and they demonstrate that FPGAs are able to successfully accelerate selection, projection, group-by aggregation, joins and even sorting, by an order of magnitude when compared to MySQL and Postgres, for instance. However, the **benefits are significantly reduced once one factors in the cost of communication over PCIe and the software overhead** of preparing the data for the FPGA to work on (sometimes pre-parsing, often copying pages).

In traditional, on-the-side deployments, the high latency communication (microseconds over PCIe) often forces designs to move entire operators onto the FPGA, even if only parts of the operator were a good match for the hardware. This leads to complications, because even though FPGAs excel at parallel and pipelined execution, they behave poorly when an algorithm requires iterative code or has widely branching "if-then-else" logic. In the case of the former, CPUs deliver higher performance thanks to their higher clock rates. In the case of the latter, the branching logic needs to be mapped to logic gates that encode all outcomes, resulting in very large circuits. Since the space on the FPGA is limited, the larger circuits result in reduced parallelism, which in turn leads to

lower throughput. This means that even though FPGAs could be successful in accelerating the common case of an algorithm, they might not be able to handle corner cases, and in practice this **leads to uncertainty in the query optimizer or even to wasted work, if an unexpected corner case is encountered during execution**.

In parallel with accelerator-based efforts, there have been numerous advances in the space of analytical databases. Today, column-oriented databases, such as MonetDB [27], are widely deployed and typically outperform row-oriented ones by at least an order of magnitude and can take advantage of many-core CPUs efficiently. As a result, the **speedups that FPGAs offer when targeting core SQL operators have shrunk**[3] and often are not enough to motivate the additional effort of integrating specialized hardware in the server architecture.

For the above reasons, FPGA-based acceleration ideas are often received with pessimism. However, changes in the hardware available in datacenters and the cloud, as well as the changes in database architecture and user workloads, create novel opportunities for FPGA-based acceleration. In the next section we discuss these in more detail and provide examples of how they can be exploited.

4 Reasons for Optimism

4.1 Changing Architectures

With the increasing adoption of distributed architectures for analytical databases, as well as the disaggregation efforts in the datacenter [28], there are numerous opportunities for moving computation closer to the data source to reduce the data movement bottlenecks. These bottlenecks arise from the fact that the access bandwidths are higher closer to the data source than over the network/interconnect and they can be eliminated by pushing filtering or similar data reduction operations closer to source. Thus, **the main goal of accelerators in the data-path is to reduce the amount of data sent to the processor, while maintaining high data access bandwidths.**

The data source is often (network-attached) flash storage and recent projects, for instance, YourSQL [17], BlueDBM [19] and Ibex [18], show that it is possible to execute SQL operations as the data is moving from storage to processing at high bandwidth. Another use-case that can benefit from data reduction in a similar way is ETL. Recent work [29] has demonstrated that specialized hardware can be used to offer a wide range of ETL operations at high data rate, including: (de)compression, parsing from formats such as CSV or JSON, pattern matching and histogram creation.

In Ibex we deployed an FPGA between an SSD and the CPU, offering several operators that can be plugged into MySQL's query plans. As Figure 3 shows, these include scans, projection, filtering and group-by aggregation, and were chosen in a way that ensures that processing in hardware will reduce the final data size for most queries. For this reason, Ibex does not accelerate joins, since these would potentially result in larger output than input and slow down the system this way. The rest of the operations are all performed at the rate of the data arriving from storage.

As opposed to on-the-side accelerators, in this space there are two possible options for who "owns" the data. In the case of smart SSDs, data is typically managed by the host database [17][18]. In contrast, in the case of distributed storage accessed over the network, it is possible to explore designs where the data is both processed and managed by the specialized hardware device as, for instance, in Caribou [30][31], our distributed key-value store that is built using only FPGAs.

In Caribou, the FPGAs implement, in addition to network line-rate data processing, a hash table data structure and memory allocator necessary for managing large amounts of data, as well as, data replication techniques to ensure that no records are lost or corrupted in case of device failures or network partitions. This results in a high throughput, energy efficient distributed storage layer that, even though is built using FPGAs, can be used as a drop-in replacement for software-based solutions [31].

[3] Using specialized hardware can still compete with multi-cores if we factor in energy efficiency (Operations/s/Watt) but in many cases the metric that is of interest is database throughput and response time.

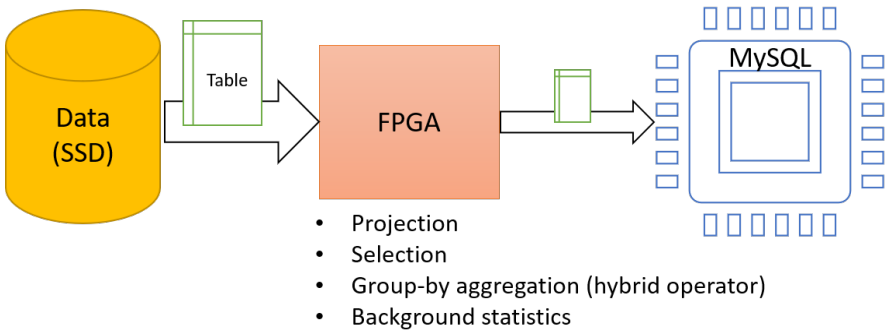


Figure 3: In Ibex we showcase several operations that can be performed on the data as it is read from storage with the goal of reducing the number of tuples that arrive at the CPU.

In many ways, in-data-path accelerators provide similar acceleration options as the on-the-side ones because data is still moved over a network (similarly to an interconnect in the case of the latter) that requires processing it in large enough batches to warrant the latency overhead. However, if FPGAs are deployed as co-processors, this overhead is drastically reduced and new opportunities open up, since the latency to the FPGA is in the same order of magnitude as a cross-socket memory access. The Centaur platform [32], for instance, exposes the FPGA of an Intel Xeon+FPGA platform using an efficient "hardware thread" API. As a result, in this co-processor scenario, **the database can offload functionality as if spawning a parallel thread and the FPGA can be used for processing even just a handful of tuples** – as we point out in the next subsection, there are emerging use-cases where this low latency acceleration is a game-changer.

4.2 Emerging Compute-Intensive Workloads

The examples in the previous subsection showed how to reduce the data access bottleneck with an in data-path accelerator targeting common SQL operators. It is unclear, however, if this strategy can be applied for co-processors as well. Modern database engines, that make use of the multi-core CPUs and their wide SIMD units, are rarely compute bound once the data is loaded into main memory. Unless reading data from storage, **offloading core SQL operators is unlikely to bring orders of magnitudes improvements in performance. There is, however, cause for optimism if we look beyond such operators and in the direction of machine learning**, both training and inference.

A significant portion of machine learning pipelines operate on relational data and the case has been made that there is a benefit in integrating these pipelines directly in the database [10]. Furthermore, there is also interest in including such components in the internal modules of the databases [12], to perform optimizations depending on the workload characteristics and the model. Since this could require on-line retraining that, without hardware acceleration, could hurt user throughput significantly, new opportunities open up for FPGAs. Acceleration of training as part of user workloads is being explored, for instance in Dana [10]. The iterative and computation-heavy nature of training operators makes them less sensitive to the latency issues introduced by using on-the-side accelerators and therefore could revive the interest in these acceleration platforms. Amazon, for instance, is already offering FPGAs running Xilinx's OpenCL-based compute framework as PCIe-attached accelerators.

In the "ML-backed" database scenario it will also be paramount to be able to take decisions with low latency using learned models – this further motivates the use of FPGAs. Even though GPUs are a de-facto standard for machine learning acceleration, when it comes to low latency inference, FPGAs can offer benefits since they do not require batching in their processing modules: recent work by Owaida et al. [33] and Umuroglu et al. [34] demonstrates, for instance, how FPGAs can be used very efficiently to accelerate inference using decision trees, respectively, neural networks.

4.3 Hybrid Approaches to Acceleration

Since all functionality, regardless whether used or not, occupies chip space on the FPGA, **corner cases often can't be efficiently handled in hardware. For this reason, it is important to design accelerators such that they behave predictably even if the particular instance of the problem can't be fully handled.** As we illustrate below with two examples from our work, state of the art solutions overcame such cases by splitting functionality between FPGA and software, such that the part on the FPGA remains beneficial to execution time regardless of the input data contents or distribution.

In Ibex [18] we used a hybrid methodology to implement a group-by operator that supports `min`, `max`, `count` and `sum` (in order to compute `avg`, we used query rewriting to compute the count and sums). This operator is built around a small hash table that collects the aggregate values. In line with FPGA best-practices, the hash table is of fixed size and is implemented in BRAM. The reason for this is that this way it is possible to guarantee fixed bandwidth operation, regardless of the data contents, because the FPGA doesn't have to pause processing to resize the table.

Unfortunately, this approach has a drawback: if a query has just one more group than the size of the hash table, the FPGA can't be used – and this information is often not available up front. We overcome this situation by post-processing the results of the group-by operator on the FPGA in software. The hardware returns results from the group by aggregation unit in a format that allows the database to perform an additional aggregation step without having to apply projections on the tuples or parse them in the first place (see Figure 4). If during the hash table operations collisions are encountered that can't be solved, a partial aggregate is evicted from the table and sent to the software post-processor. Once all the data has been processed on the FPGA, the contents of the hash table are sent to the software post-processor to compute the final groups. This results in a behavior where, if all the groups could be computed on the FPGA, the final software step has to perform virtually no work (assuming that the number of resulting groups is significantly smaller than the cardinality of the table), and otherwise the software executes the group by aggregation as if there wasn't any FPGA present (though still benefits from projections and selections).

The regular expression-based LIKE operator that we implemented in MonetDB [35] running on top of the Intel Xeon+FPGA platform is another example of the hybrid operator methodology. If the expression can't be encoded in its entirety on the FPGA, because, for instance, it contains too many characters (such as the bottom example in Figure 5), we cut it at the last possible wildcard and process the first part of the expression on the FPGA and the second part in software. For each string, the FPGA operator returns an index that signifies the end of the location where the regular expression matched the string. The software can pick up processing from this

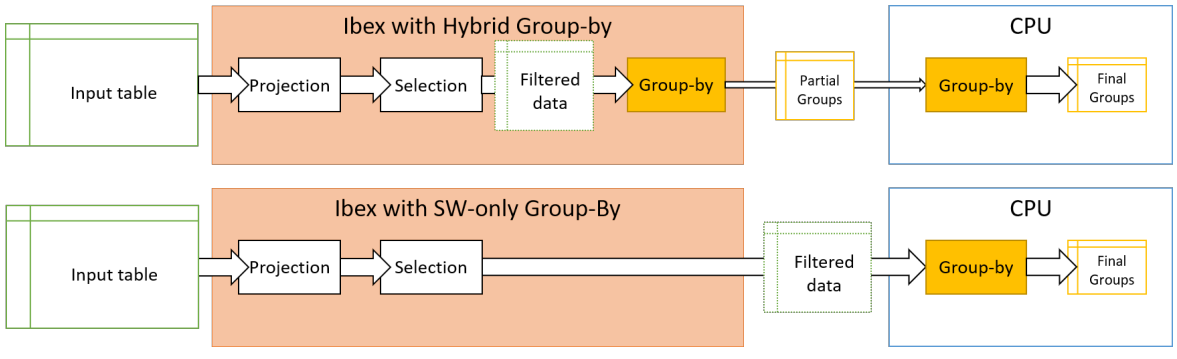


Figure 4: By implementing operators in a way that allows hybrid computation, the FPGA accelerator can reduce data sizes over the bottleneck connection to the CPU in most cases. In this example of Ibex's group-by operator, if we would choose an "all or nothing approach", moving the data to be aggregated to the CPU could become the bottleneck.

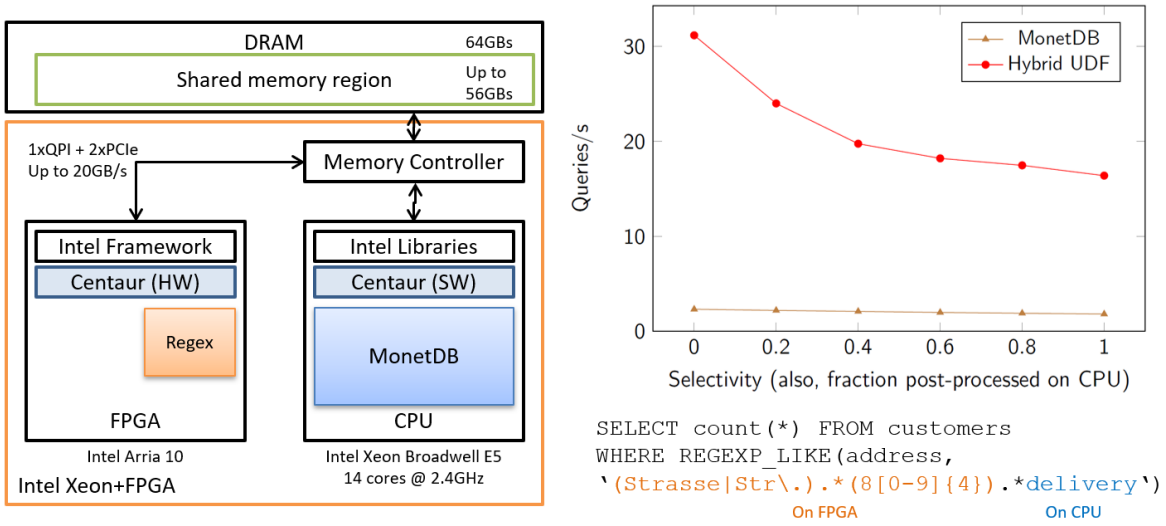


Figure 5: Even if only part of the regular expression fits on the FPGA it is worth to offload it because the post-processing becomes cheaper, resulting in an overall faster execution.

point in case of hybrid processing and match the rest of the expression. In case the entire expression fits on the FPGA, however, the software has no additional work to do. In Figure 5 we illustrate how, when compared to a single-threaded execution in MonetDB, the hybrid solution is always faster than the software-only one (for more details see [35]).

One aspect that makes the integration of programmable hardware in databases challenging is the change in the predictability of query runtimes. Therefore, in our work we aim to design circuits whose throughput is not affected by the problem instance they work on. This way the query optimizer can predict the rate at which data will be processed/filtered on the FPGA and with this information it can reliably decide when to offload. One example of such a design is the regular expression module we presented above. Since the overhead of compiling regular expressions to circuits and then performing partial reconfiguration (PR) could take longer than executing an entire query, we took a different approach: we created a "universal" automaton that could implement any expression, within some limits on the number of distinct characters to detect and the number of states. Small on-chip memories are used to describe the state machine and the characters of the regular expression, and their contents can be loaded at runtime in nanoseconds. We laid out this state machine as a pipeline, that processes one character per clock cycle, regardless of the contents of the on-chip memories. The conversion from a regular expression written by a user to the configuration parameters is performed in software but is orders of magnitude cheaper than circuit synthesis.

5 The Road that Lies Ahead

5.1 Managing Programmable Hardware

How to best integrate hardware that, even though reprogrammable, will never be as flexible as software? Should the operating system/hypervisor control it, or can we build future databases that do this?

Even though there are efforts in the FPGA community to speed up the process of partial reconfiguration, it is unlikely that the overhead of this operation will ever be as small as that of a software context switch. As a result, databases must find ways to adapt to the idea of running on specialized hardware that, even though, can be reprogrammed, doesn't have the flexibility of software. The main question that needs to be answered in this space is who will "own" the acceleration functionality, because this also defines whether the database needs only to be

able to compile its queries to take advantage of the accelerators, or whether it could also synthesize fully custom accelerators depending on the workload.

If it is the OS/hypervisor that controls the accelerator, then the database still has to be able to adapt to different underlying hardware acceleration functionality, that will likely be both designed and managed by the infrastructure/cloud provider. In this scenario, the database has to create query plans that take advantage of the specific machine's acceleration opportunities. For this, it is likely that we can reuse techniques that are already present in databases for compiling code for different target CPU features such as SIMD units [40].

Alternatively, if the database takes full ownership of the accelerator, it will have more responsibility but also greater opportunities. Instead of relying on the cloud provider to design general-purpose acceleration units that might or might not match the database's needs, the database developer can design and synthesize the right ones and integrate them tighter with the database. What's more, the database could even generate and synthesize workload-specific accelerators at runtime.

In DoppioDB [32][36] we explored the case where the database manages the accelerator. The role of the operating system is to set up a basic infrastructure on the FPGA, configuring it with several "slots" that can be filled in using partial reconfiguration (we call these slots hardware threads because the interface to them in software is similar to a function call on a new thread). Once the database has started, the FPGA gets access to the process's virtual memory space and the database can explicitly manage what tasks the different slots perform, choosing in our prototype from a small library of available operators. In DoppioDB, instead of focusing only on the usual SQL operators like selection or joins, we began exploring how one could extend what the database is capable of, targeting machine learning type of operators, such as training a model using stochastic gradient descent or running inference with decision trees. This functionality was exposed using a UDF mechanism, but in the future could be integrated much tighter with the database. The research question that emerges is how to populate the hardware operator library and what granularity these operators should have. Recent work by Kara et al. [42] shows that it is possible to offload sub-operators successfully to the FPGA. However, the identification of generic enough sub-operators that can be deployed on an accelerator and parameterized/composed at runtime remains an open challenge.

5.2 Compilation/Synthesis for Programmable Hardware

Are there reusable building blocks that would make query compilation easier for programmable hardware? Should databases have their own DSLs from which to generate hardware accelerators?

The second big question is how to express acceleration functionality for database use-cases in an efficient way. As opposed to CPUs or GPUs where the architecture (ISA, caches, etc.) is fixed, in an FPGA it is not. This adds a layer of complexity to the problem of compiling operators, as well as query planning in general. Given even just the heterogeneity of modern CPUs and their different SIMD units, there is already a push for databases to incorporate more and more compiler ideas [40][41].

The side effect of bringing more ideas from compilers into databases is that it will likely also be easier to integrate DSLs for hardware accelerators [37][38][39] into the database. However, many of these solutions are targeting compute kernels written in languages such as OpenCL [37], that are a better fit for HPC and machine learning type functionality than database operations. Therefore, novel ideas are needed that bridge the space between databases and languages/compilers for specialized hardware. One possible direction to explore is related to the design of the Spatial language and compiler [38]. Spatial approaches the problem of writing parallel code for accelerators in a way that accounts for the fact that circuits are physically laid out on the chip. Given that query plans are often composed by a set of sub-operators that are parameterized differently to implement, for instance, different join types, these could be an intermediate step between SQL and hardware circuit that allows the database to offload a pipeline of such sub-operators to the FPGA in an automated manner.

Another aspect that makes translating operators to hardware-based accelerators challenging comes from the fact that not all functionality will fit on the device. This is true regardless whether we target an FPGA, a P4-based

switch or SmartNIC, or an ASIC-based solution such as the DAX. Therefore, even if the best case of an operator can be efficiently translated to hardware, corner cases will have to be handled without significantly impacting performance. For this reason, the challenge of compilation is also related to the ideas discussed before around hybrid execution and query planning. Frameworks that compile queries to such platforms will have to provide software-based post-processing functionality to ensure that corner cases are gracefully handled. The challenge in this hybrid computation is to find suitable points where to split the functionality in an automated way.

6 Conclusion

In this paper we made the case that the use of specialized hardware in analytical databases has a positive outlook, even though it has been approached pessimistically for a long time. To support this argument, we discussed the past and future challenges of including a specific kind of hardware accelerator, namely FPGAs, in databases.

To address fears that deploying FPGAs always brings high overheads that reduce their "raw" speedup, we highlighted how, in today's distributed database landscape, they can be used to reduce bottlenecks of data movement by positioning them in data-path. Since they can process data at the rate at which it is retrieved from the data source, they never slow down data access, even if there is no opportunity for acceleration. We also discussed the opportunities that novel machine learning workloads bring. Their operators are typically compute bound on CPUs and using FPGAs we can achieve significant speedups even when compared to an entire socket with multiple cores. Finally, to demonstrate that it is possible to design FPGA-based operators that behave gracefully even if the entire functionality of the operator doesn't fit on the device, we discussed two examples from our previous work that implement hybrid computation across FPGA and CPU (a group-by operator and a regular expression matcher).

We also identify two areas in which significant progress has to be made for the inclusion of heterogeneous hardware in databases to become truly widespread. One is finding ways to actively manage the programmable hardware underneath the database, shaping it to workloads using partial reconfiguration and parameterizable circuits. The second question is about finding the right programming primitives for hardware accelerators in the context of database operators, to avoid designing from scratch each new accelerator idea and to allow the database to offload parts of a query more flexibly at runtime. It is unlikely that we can provide answers for both questions only from inside the database community and will have to instead collaborate with researchers working in the areas of operating systems, programming languages and compilers.

Acknowledgments

Our cited work and many of the lessons learned are a result of the author's collaboration with current and past members of the Systems Group at ETH Zürich, in particular, Gustavo Alonso, David Sidler, Louis Woods and Jana Giceva.

References

[1] J. Banerjee, D. Hsiao and K. Kannan. DBC: A Database Computer for Very Large Databases. *IEEE Transactions on Computers*, 6, pp. 414-429, IEEE, 1979.

[2] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.I. Hsiao, R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and data engineering*, 2(1), pp. 44-62, 1990.

[3] P. Francisco. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Red Books*, 2011.

[4] L. Wu, A. Lottarini, T.K. Paine, M. Kim, K.A. Ross Q100: The Architecture and Design of a Database Processing Unit. *ASPLOS'14*, pp. 255-268, 2014.

[5] S.R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, E. Sedlar et al. A many-core architecture for in-memory data processing. *MICRO'17*, pp. 245-258, ACM, 2017.

[6] H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam and D. Burger. Dark silicon and the end of multicore scaling. *ISCA'11*, pp. 365-376, IEEE, 2011.

[7] K. Sato, C. Young, D. Patterson. An in-depth look at Google's first Tensor Processing Unit (TPU). *Google Cloud Big Data and Machine Learning Blog, 12*, 2017.

[8] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. *NSDI'18*, USENIX, 2018.

[9] P.K. Gupta. Accelerating datacenter workloads. *FPL'16*, 2016.

[10] D. Mahajan, J.K. Kim, J. Sacks, A. Ardalan, A. Kumar, H. Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. *Proceedings of the VLDB Endowment*, 11(11), 2018.

[11] J.M. Hellerstein, C. Re, F. Schoppmann, D.Z. Wang, E. Fratkin, A. Gorajek, K.S. Ng, C. Welton, X. Feng, K. Li,et al. The MADlib analytics library: or MAD skills, the SQL. *PVLDB*, 5(12), pp. 1700–1711, 2012.

[12] T. Kraska, M. Alizadeh, A. Beutel, E. Chi, J. Ding, A. Kristo, V. Nathan, et al. SageDB: A learned database system. *CIDR'19*, 2019.

[13] E. Sitaridi, K. Ross. GPU-accelerated string matching for database applications. *Proceedings of the VLDB Endowment*, pp. 719-740, 2016.

[14] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Brezzo, S. Asaad, D.E. Dillenberger. *Database analytics: A reconfigurable-computing approach*, IEEE Micro, 34(1), pp. 19-29, 2014.

[15] M. Oskin, F.T. Chong, T. Sherwood. Active pages: A computation model for intelligent memory. *IEEE Computer Society*, Vol. 26, No. 3, pp. 192-203, 1988.

[16] G. Koo, K.K. Matam, H.V. Narra, J. Li, H.W. Tseng, S. Swanson, M. Annavaram. Summarizer: trading communication with computing near storage. *MICRO'17*, pp. 219-231, ACM, 2017.

[17] I. Jo, D.H. Bae, A.S. Yoon, J.U. Kang, S. Cho, D. Lee, J. Jeong. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12), pp. 924-935, 2016.

[18] L. Woods, Z. Istvan, G. Alonso. Ibex: an intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment*, 7(11), pp. 963-974, 2014.

[19] S.W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu. BlueDBM: Distributed Flash Storage for Big Data Analytics. *ACM TOCS* 34(3), 7, 2016.

[20] C. Barthels, S. Loesing, G. Alonso, D. Kossmann. Rack-scale in-memory join processing using RDMA. *SIGMOD'15*, pp. 1463-1475, ACM, 2015.

[21] A. Dragojevic; D. Narayanan; M. Castro. RDMA Reads: To Use or Not to Use?. *IEEE Data Eng. Bull.*, vol. 40, no 1, pp. 3-14, 2017.

[22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), pp. 87-95, 2014.

[23] K. Aingaran, S. Jairath, D. Lutz. Software in Silicon in the Oracle SPARC M7 processor. *Hot Chips Symposium (HCS'16)*, pp. 1-31, IEEE, 2016.

[24] J. Teubner and L. Woods. Data processing on FPGAs. *Synthesis Lectures on Data Management*, 5(2), pp. 1-118, 2011.

[25] J. Casper, K. Olukotun. Hardware acceleration of database operations. *FPGA'14*, pp. 151-160, ACM, 2014.

[26] C. Dennl, D. Ziener, J. Teich Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. *FCCM'13*, pp. 25-28, IEEE, 2013.

[27] P.A. Boncz, M. Zukowski, N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR*, Vol. 5, pp. 225-237, 2005.

[28] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, S. Kumar. Flash storage disaggregation. *EUROSYS'16*, 2016.

[29] Y. Fang, C. Zou, A.J. Elmore, A.A. Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. *MICRO'17*, pp. 55-68, ACM, 2017.

[30] Z. Istvan, D. Sidler, G. Alonso. Caribou: intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11), pp. 1202-1213, 2017.

[31] Z. Istvan. Building Distributed Storage with Specialized Hardware Doctoral dissertation, ETH Zurich, 2018.

[32] M. Owaida, D. Sidler, K. Kara, G. Alonso Centaur: A framework for hybrid CPU-FPGA databases. *FCCM'17*, pp. 211-218, IEEE, 2017.

[33] M. Owaida, H. Zhang, C. Zhang, G. Alonso. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. *FPL'17*, IEEE, 2017.

[34] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. *FPGA'17*, pp. 65-74, ACM, 2017.

[35] D. Sidler, Z. Istvan, M. Owaida, G. Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. *SIGMOD'17*, pp. 403-415, ACM, 2017.

[36] D. Sidler, Z. Istvan, M. Owaida, K. Kara, G. Alonso. doppioDB: A hardware accelerated database. *SIGMOD'17*, pp. 1659-1662, ACM, 2017.

[37] M. Wong, A. Richards, M. Rovatsou, R. Reyes. Khronos's OpenCL SYCL to support heterogeneous devices for C++, 2016.

[38] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, K. Olukotun. Spatial: a language and compiler for application accelerators. *PLDI'18*, pp. 296-311, ACM, 2018.

[39] O Mencer. Maximum performance computing for Exascale applications. *ICSAMOS'12*, 2012.

[40] H. Pirk, J. Giceva, P. Pietzuch. Thriving in the No Man's Land between Compilers and Databases. *CIDR*, 2019.

[41] H. Pirk, O. Moll, M. Zaharia, S. Madden Voodoo - A vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment*, 9(14), pp. 1707-1718, 2016.

[42] K. Kara, J. Giceva, G. Alonso. Fpga-based data partitioning *SIGMOD'17*, pp. 433-445, ACM, 2017.

Scheduling Data-Intensive Tasks on Heterogeneous Many Cores

Pınar Tözün
IT University of Copenhagen
pito@itu.dk

Helena Kotthaus
TU Dortmund University
helena.kotthaus@tu-dortmund.de

Abstract

Scheduling various data-intensive tasks over the processing units of a server has been a heavily studied but still challenging effort. In order to utilize modern multicore servers well, a good scheduling mechanism has to be conscious of different dimensions of parallelism offered by these servers. This requires being aware of the micro-architectural features of processors, the hardware topology connecting the processing units of a server, and the characteristics of these units as well as the data-intensive tasks. The increasing levels of parallelism and heterogeneity in emerging server hardware amplify these challenges in addition to the increasing variety of data-intensive applications.

This article first surveys the existing scheduling mechanisms targeting the utilization of a multicore server with uniform processing units. Then, it revisits them in the context of emerging server hardware composed of many diverse cores and identifies the main challenges. Finally, it concludes with the description of a preliminary framework targeting these challenges. Even though this article focuses on data-intensive applications on a single server, many of the challenges and opportunities identified here are not unique to such a setup, and would be relevant to other complex software systems as well as resource-constrained or large-scale hardware platforms.

1 Introduction

Utilizing the processors of commodity servers well is crucial to avoid wasting resources, energy, and money in data centers regardless of their scale [16]. As a result, quest to remove the bottlenecks of data management systems causing underutilization of modern commodity servers have been the focus of many past and ongoing work [1]. One of the essential challenges in this quest is scheduling various data-intensive tasks effectively over the processing units that are available to these tasks. The fundamental evolution of the server hardware and the increasing variety of the data-intensive applications over the recent years amplify this challenge.

Server hardware has gone through major advances over the years as illustrated in Figure 1a. These advances have stemmed from Moore's Law [27], which is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. To exploit the increase in the transistor counts in a unit area, initially, computer architects focused on boosting the performance of a single core while designing chips (left-hand side of Figure 1a). Around 2005, however, Dennard Scaling [10], which states that as the transistors get smaller their power density in a unit area remains constant, came to a halt. Increasing the complexity of a processor core became non-viable since it raised concerns about power draw and heat dissipation. To overcome

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

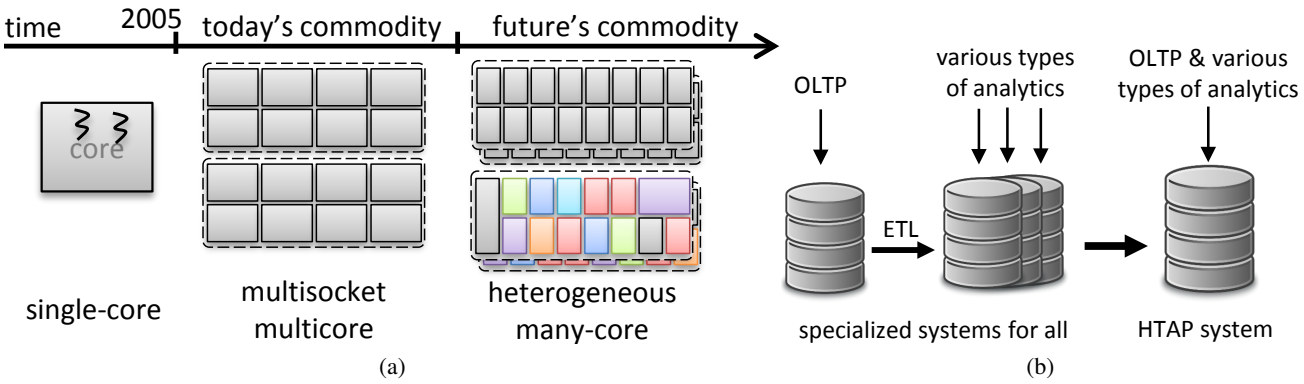


Figure 1: (a) Evolution of server hardware over time following Moore's Law and Dennard Scaling. Each break in the timeline represents a disruptive period for processor evolution due to power concerns. (b) Ways to deploy data-intensive applications.

this limitation, computer architects started to add more and more cores on a single processor [29] and more and more processors in servers (middle part of Figure 1a). Multicore processors have enabled the continuation of Moore's Law despite the halt of Dennard Scaling. Unfortunately, the limits of the traditional multicore processor design is also upon us. Adding more and more cores to a processor cannot be the only path to overcome the halt of Dennard Scaling since we will not be able to power all of those cores up simultaneously. This trend is also known as dark silicon [11]. To overcome this limitation, we must design cores that are more energy-efficient. One way to achieve this by specializing cores, reducing energy spent per instruction, for certain tasks [20]. Then, as part of the commodity servers, one can utilize the specialized cores in addition to the general-purpose ones. The emerging server hardware landscape, therefore, will likely to be composed of a heterogeneous set of processing units (as illustrated by the different colors in right-hand side of Figure 1a); each specialized to execute a specific task very well, with opportunities for extreme levels of parallelism.

In parallel to the evolution of commodity server hardware that data-intensive applications typically run on, the applications themselves and how they are deployed have also changed over time as illustrated in Figure 1b. Transaction and analytical processing used to be the two broad categories of data-intensive applications. Analytics, in turn, have several distinct sub-categories such as online analytical processing, data warehousing, machine learning, graph analytics, etc. Traditionally, they have been deployed separately and data moved from an operational system (such as an online transaction processing system) to various types of analytics systems using an extract-transform-load (ETL) process. The reason for this separation is that optimal system design for serving transactional and different types of analytical tasks are different (e.g., row stores for OLTP, column stores for OLAP, NoSQL for unstructured data, etc.). In recent years, however, the popularity of the data-intensive applications such as real-time inventory/pricing/recommendations, fraud detection, risk analysis, IoT, AI, etc. require data management systems that can run fast transactions and analytics simultaneously. As a result, there is an increasing demand for data management systems that can handle hybrid transactional and analytical processing (HTAP) efficiently [30].

Designing a scheduling mechanism that is able to leverage the heterogeneity of processing units for the variety of the data-intensive tasks to be executed in the emerging hardware and software landscapes is a difficult but significant challenge to tackle. The goal of this article is to derive some guidelines to overcome this challenge in the context of a single node of commodity server hardware. First, Section 2 surveys some of the existing work that target scheduling of data-intensive tasks on modern homogeneous multicore servers. Then, Section 3 discusses emerging heterogeneous server hardware landscape and considers existing work in the context of such hardware. Finally, Section 4 illustrates a framework for scheduling diverse set of (or hybrid) data-intensive tasks over diverse set of (or heterogeneous) processing units focusing on the resource estimation challenges.

2 Scheduling Data-Intensive Tasks over Different Dimensions of Parallelism

As previously mentioned, we view the effective scheduling of different data-intensive tasks as a significant factor when it comes to effective utilization of the resources of modern server hardware. Any mechanism that targets effective scheduling must be able to answer the following questions.

***What** to schedule?* This question determines the unit of scheduling. What is the granularity of the task to be executed on a specific processing unit? Is it the whole data-intensive task required by a client request or is it part of it? If it is a part of it, what is the size of that part?

***Where** to schedule?* This challenge handles the mapping between tasks and processing units. This mapping has both a *static* and a *dynamic* part. The static mapping targets the question of what the most effective processing unit/units to execute a task is/are. The answer to this question assumes that every kind of processing unit is available in infinite amounts. In practice, however, we rarely have all kinds of processing units in a single server and the hardware resources are finite. The dynamic mapping must consider the question of whether the ideal processing units for a task are available at the exact time that we have to execute that task. In addition, in the case of unavailability, what are the next best alternatives?

***How** to schedule?* This challenge provides the necessary execution and communication primitives, especially if multiple processing units are involved in executing a task. What are the primitives to utilize while scheduling a task or parts of a task? Which level(s) of the system stack these primitives come from?

The following subsections survey the scheduling mechanisms proposed in recent work that depart from the conventional wisdom when it comes to the answers to the questions above. Section 2.1 and Section 2.2 focus on utilizing the resources of a single core and multiple uniform cores, respectively.

2.1 Implicit/Vertical Parallelism

Before Dennard Scaling made it problematic to put more complexity within a core due to heat dissipation concerns, exploiting Moore's Law meant boosting the performance of a single core. This resulted in parallelism opportunities within a core through techniques like instruction level parallelism, pipelining, out-of-order execution, simultaneous multithreading, etc. We refer to this kind of parallelism as *implicit/vertical* parallelism as the different tasks are time-multiplexed on the same core instead of being run concurrently in the same execution cycle. The main insight behind this kind of parallelism is overlapping various stall times with other work instead of a core wasting the execution cycles being idle. For example, as a core waits for fetching an instruction or data item from memory due to it not being present in L1 caches, one can overlap this waiting time with another instruction or data fetch request from the same task or execute another task on the same core. In addition, mostly hardware manages this kind of parallelism and software has the luxury to be oblivious to it. Therefore, before multicores emerged, the software systems got faster with each new generation of servers without having to make fundamental design changes.

On the other hand, for many data-intensive applications being oblivious to implicit parallelism leads to severe underutilization of the micro-architectural resources of servers. Several workload characterization studies emphasize the high rates of memory access related stall times due to either instruction or data accesses for data-intensive applications [12, 37]. Similarly, techniques like simultaneous multithreading may even hurt performance if not used carefully [42]. Multiple data-intensive tasks sharing the same resources in a core simultaneously may put more pressure on caches due to their aggregate data and instruction footprint. Therefore, there is value to rethink the way we design and schedule data-intensive tasks even when utilizing implicit parallelism.

Figure 2 illustrates alternative ways of scheduling a data-intensive task on a single core (a & b) and on multiple cores (c & d). The figure assumes that tasks run over a setup that has fast I/O (e.g., DRAM, NVRAM, low-latency SSD) and hence do not require context switching due to slow I/O (e.g., HDD). In the figure, a data-intensive task is at the granularity of a whole transaction or analytical query. The task has three sub-tasks A, B, C. In the interest of our discussion, let's assume that these sub-tasks are at a granularity where their instruction

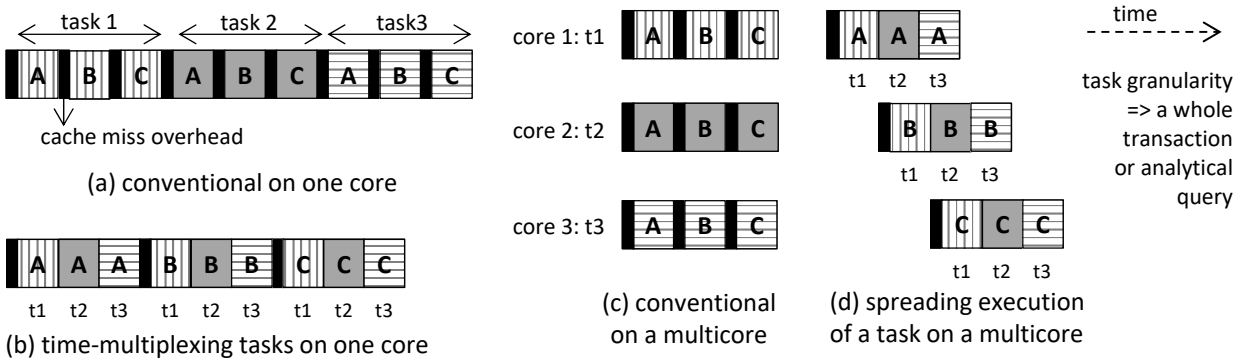


Figure 2: Different ways of scheduling data-intensive tasks. In the context of this illustration a task is at the granularity of a transaction or an analytical query, where A, B, and C are the sub-tasks of this task.

or data set sizes can fit in L1-I or L1-D, respectively. This section discusses Figure 2(a & b) since the focus is on implicit parallelism, whereas the discussion of Figure 2(c & d) is in Section 2.2.

Figure 2(a) depicts the more conventional way of scheduling tasks. In this case, the tasks run without any interruptions on a core as a whole one after the other based on their priority in a task queue in the system. They take turns thrashing the caches since each executes sub-tasks A through C in order independent of the other tasks. Thus, each sub-task incurs overhead due to cache misses. In this case, the answer to the *what* question is the whole task while the *where* question doesn't matter as there is only a single core, and the answer to the *how* question is mainly left to the default mechanisms supported by the operating system.

Figure 2(b) shows an alternative way to schedule the sub-tasks, which time-multiplexes them with the goal of maximizing cache locality. The first, *lead*, task executes A incurring cache miss overhead as previously. However, instead of proceeding to execute B, the first task context switches allowing, in turn, the second and third tasks to execute instead. The second and third tasks find sub-task A in L1 and thus incur no overhead due to misses. Once all three tasks execute the first sub-task, execution proceeds to the second one and so on.

The core idea of time-multiplexing the tasks on a single core to improve cache locality has been studied and shown to be effective in the context of both instruction (L1-I) and data (L1-D) [4, 17, 18, 24] locality. The main insight behind this idea is that similar tasks share common instructions or data or both. As a result, they can benefit from constructive sharing of the cache resources to improve locality. Fewer cache misses lead to better utilization of the micro-architectural resources that enable implicit parallelism within a core since a smaller portion of the overall execution time is spent on stalls. Even if a technique that focuses on instruction cache locality may hinder data cache locality or vice-versa, the benefits of one may outshine the overhead of the other, or the locality may improve at the higher levels of the cache hierarchy despite the hindered L1 locality thanks to constructive sharing [39].

Achieving constructive sharing for different concurrent tasks in a system is not straightforward. The first challenge is the underlying assumption of the tasks would have similar sub-tasks to be executed. For data-intensive applications, this is not an issue. No matter how different the output or high-level functionality of one data-intensive task from another, data management or processing systems typically compose a subset of predefined sub-tasks to serve a task. Figure 3 and Figure 4 show some examples within the same or across different applications/tasks. Transactions are composed of sub-tasks such as probing and scanning an index, inserting a tuple to a table, updating a tuple, etc. Traditional analytical queries are composed of projections, selections, joins, etc. These sub-tasks themselves have common sub-tasks such as hash table lookup, data partitioning, sorting, etc. across different types of sub-tasks or workloads. There may be frequently accessed tables or indexes or metadata used by several of these sub-tasks. Overall, there are many opportunities for constructive sharing in data-intensive applications.

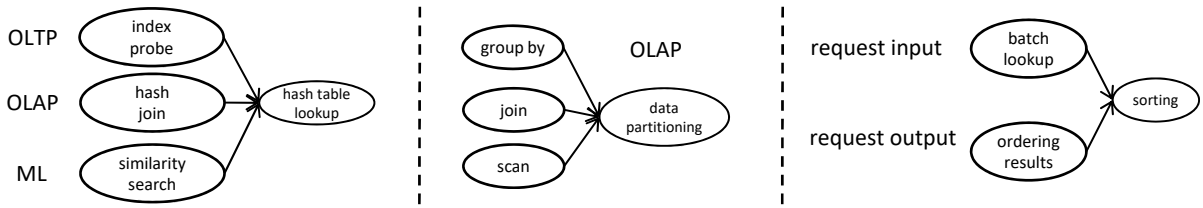


Figure 3: Examples of common sub-tasks across different types of data-intensive applications. Hash table lookup is common across OLTP, OLAP, and machine learning. Data partitioning is common across many OLAP tasks. Sorting is common both across data-intensive applications and while ordering input/output values from various client requests.

Determining the granularity of sub-tasks to time-multiplex at runtime is a harder challenge (to answer the *what* question) as well as orchestrating the runtime scheduling in a lightweight manner (to answer the *how* question). Regarding the former challenge, previous work either considers the granularity of database operators [18], rely on monitoring the cache behavior at runtime to determine when the L1 cache starts to become full [4], or perform profiling [17]. Regarding the latter challenge, previous work either adopts hardware mechanisms [4] to sidestep the overheads of default context switching primitives of the operating system, or develop specialized context switching at the kernel-level [17].

Finally, despite increasing the throughput, time-multiplexing a batch of tasks on one core increases the average latency, especially for the *lead* task. One has to take into account the priority or latency requirements of the data-intensive tasks when deploying these types of scheduling mechanisms. This challenge is definitely under-studied in the literature.

2.2 Explicit/Horizontal Parallelism

The switch to multicore processors forced traditional software systems, including data management systems, to go through fundamental design changes in order to exploit the kind of parallelism offered by having multiple cores in a processor [1]. We refer to this kind of parallelism as *explicit/horizontal* parallelism as it allows different tasks to run simultaneously in the same execution cycle. Unlike implicit parallelism, this type of parallelism has to be managed more carefully at the software side to reap the benefits. In this dimension of parallelism, majority of the efforts from previous work focus on removing scalability bottlenecks that arise due to concurrent threads accessing shared data. Complementary to such scalability problems, this article focuses on the work that targets scheduling of data-intensive tasks on multicores.

Let's start by following the discussion from Figure 2, Figure 2(c & d) illustrate alternative ways of scheduling data-intensive tasks on multicores. Figure 2(c) depicts the more conventional way when there is no I/O. Each task is scheduled to a different core in the system and executed as a whole on that core. As a result, each task exhibits cache misses since neither the instruction nor the data footprint of data-intensive tasks fit in L1 caches. This way of scheduling stems from traditional data management systems treating various data management tasks as large indivisible units of work on a single server. This monolithic view of such tasks eventually leads to sub-optimal resource management decisions even on today's homogeneous commodity server hardware [39].

In the case of Figure 2(c), just like Figure 2(a), the answer to the *what* question is the whole task and the answer to the *how* question is mainly left to the default mechanisms supported by the operating system. On the other hand, having multiple cores makes the *where* question more difficult to answer. Traditional systems typically pick the next available/idle core to schedule the next task to be executed.

Figure 2(d), on the other hand, spreads the computation of a data-intensive task over multiple cores and utilizes the aggregate L1-I cache capacity of the multicores while executing this task. The main insight behind this idea is, as in Section 2.1, the observation that data-intensive tasks in general share common sub-tasks (or

code [39]). As long as there are enough cores so that the aggregate L1-I capacity can hold all code segments, a task can migrate to the core whose L1-I cache holds the code segment the task is about to execute. For example, as Figure 2(d) shows, the first, *lead*, transaction can execute sub-task A first on core 1, then migrate to core 2 where it would execute sub-task B, then migrate to core 3 where it would execute sub-task C. The second and third tasks can follow in a pipelined fashion, finding sub-tasks A, B, and C, in cores 1, 2, and 3, respectively. While the lead task incurs an overhead when fetching the code segments for the first time, the other tasks do not. Even though, the migrations may diminish data locality at the L1-D level, as long as they happen within a processor/socket, long-latency data misses from the last-level cache either stay the same or get reduced as a result of constructive data sharing across similar tasks [39].

The core idea of spreading the computation over multiple cores to improve instruction cache locality, is initially studied in the context of separating kernel code from application code in [7]. SLICC [3] and ADDICT [39] have taken this idea further to also localize the common application code across concurrent data-intensive tasks over specific cores. These work in fact target improving cache locality, minimizing stall times due to cache misses, and hence, improving utilization of implicit parallelism within a core like the work described in Section 2.1. However, they exploit explicit parallelism to achieve their target. Similarly, the staged execution mechanisms such as QPipe [18], which are originally developed with implicit parallelism in mind, are later adapted to utilize explicit parallelism as well [14, 34]. Furthermore, separating the tasks to be executed by the kernel and the data-intensive application into common sub-tasks, and running these sub-tasks over separate specific cores is also studied in the context of effective operating system and database system co-design [15].

In addition to strengthening the techniques that target improving (instruction or data) locality for data-intensive tasks, explicit parallelism also allows exploiting intra-task parallelism. In other words, the independent sub-tasks of a data-intensive task can run concurrently over multiple cores. To prevent underutilization of ever increasing explicit parallelism offered by multicores or many cores, intra-task parallelism is essential. Viewing tasks as a black-box and just focusing on optimizing for inter-task parallelism is ineffective while scaling up on servers with 100s or 1000s of cores.

A common way to achieve intra-task parallelism is to partition the data to be processed by a data-intensive task and assign different threads to each partition [25, 35]. Data partitioning is only one dimension when targeting intra-task parallelism, though. The other, slightly more challenging, dimension is to detect the independent sub-tasks within a task that can run concurrently. Figure 4 gives an example of how to parallelize the sub-tasks of the `payment` transaction from the industry-standard TPC-C benchmark [40], which is utilized by the DORA/PLP mechanisms [31, 32]. The three update operations over the different tables (customer, district, and warehouse) have no dependency on each other and can run in parallel while the insert operation over the history table must run after these three. Previous work adapted the SQL frontend of Postgres to determine the independent sub-tasks of transactions automatically [31]. Expanding this methodology to more complex data-intensive tasks is still a challenge.

All the mechanisms that involve multiple cores in the execution of a transaction whether it is to improve cache locality or intra-task parallelism or both, have the same challenges as the mechanisms that time-multiplex sub-tasks on the same core (Figure 2(b)). Therefore, the answers to the *what* and *how* questions here are the same as in Section 2.1 (i.e., finer-granularity sub-tasks and lighter-weight context switching). Answering the *where* question, on the other hand, requires runtime monitoring and bookkeeping to know which core has the instructions a sub-task needs or which cores are assigned to which database operators beforehand. Furthermore, explicit parallelism in the era of multisocket multicore hardware with non-uniform memory access (NUMA), also brings the challenge of minimizing communication overheads across the sub-tasks. Involving multiple cores in the execution of a data-intensive tasks require orchestrating the sub-tasks, which requires communication across cores that may not be able to communicate as fast as some other cores. Naive ways of scheduling sub-tasks ignoring hardware topology, especially NUMA, may hinder overall performance even if it utilizes explicit parallelism well [33, 35]. Therefore, one must definitely take the hardware topology into account while answering the *where* question.

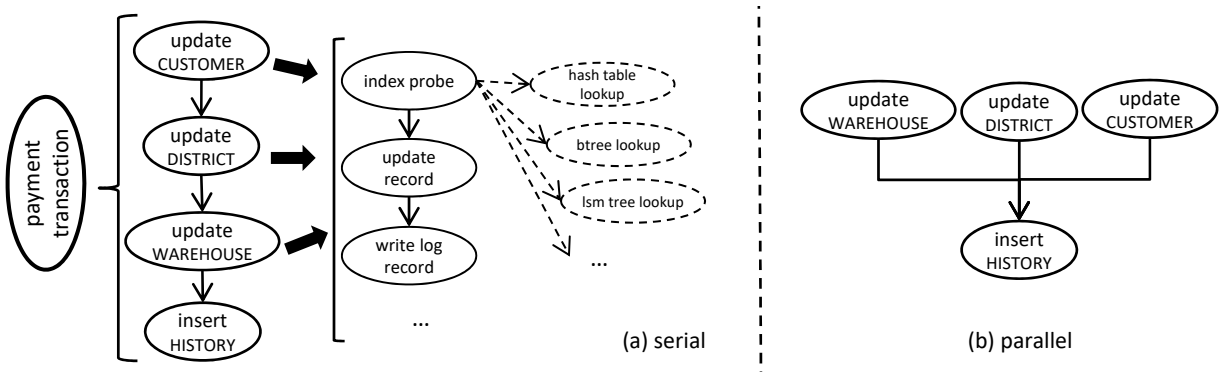


Figure 4: TPC-C's `payment` transaction. Each node represents a sub-task in `payment`. `Customer`, `District`, `Warehouse`, and `History` are the tables in TPC-C. (For brevity the iteration over the `Customer` table in this transaction is omitted). (a) The serial execution plan also illustrating the sub-tasks of `payment` at different granularities. `payment` performs operations such as update and insert. An update performs actions such as index probe, update record, write log, etc. These actions also have sub-tasks at a finer-granularity. (b) The parallel execution plan for `payment` exploiting intra-transaction parallelism among the three independent updates.

3 Toward Heterogeneous Parallelism

As mentioned in Section 1, adding more and more cores to a processor cannot be the only path for progressing commodity servers anymore. Moore's Law is slowing down. The main reason is once again power-related even though there are other physical constraints as well (e.g., fabrication costs for transistors that get smaller and smaller). The supply voltage required to power all the transistors up does not decrease at a proportional rate. Even if we can still add more and more cores on processors, we will not be able to power all of them up simultaneously [11]. Optimizing energy per instruction has to be the key in this new era.

One option to achieve more energy-efficient hardware is to adopt simpler and more low-power cores in emerging processors. However, such processing units are not suitable for latency critical applications. In addition, solely focusing on the energy-efficiency of an individual core or processor is not going to give us energy-proportionality [5]. We have to focus on how much energy it takes to run tasks to completion. The low-power cores might end up spending more energy at the end of the day for running a set of tasks compared to power-hungry cores since it takes them longer to execute tasks due to being slower [23].

The better long-term solution for energy-efficiency is to build servers with a variety of processing units, where each unit is specialized to accelerate specific tasks. On such servers, one would pick the cores to power-up based on the tasks currently running, while shutting down the idle cores that are specialized for other types of tasks. Orchestrating task scheduling dynamically over such heterogeneous hardware intensifies an already challenging problem on homogeneous hardware (as Section 2 focused on). In addition, economic feasibility of specialized hardware is always a concern since specialization limits the market of a system, despite being more efficient, as opposed to being general-purpose. As a result, processor specialization for data-intensive tasks was unpopular in industry up until recently. However, this attitude has been changing [2, 20]. Therefore, there is pressing need to develop scheduling mechanisms that not only consider the diversity of the data-intensive tasks, but also the diversity of the processing units.

The scheduling approaches surveyed in Section 2 are inspiring and preliminary steps toward the efficient utilization of processors with many diverse parallelism opportunities. The common denominators (and the common root of the associated challenges) across all of these mechanisms are that (1) they view data-intensive tasks at a finer sub-task granularity (e.g., *update* operation of *payment* transaction instead of the whole *payment*

transaction) and (2) they adopt lighter-weight and hardware-topology-aware techniques for dynamic scheduling of these sub-tasks instead of relying on the traditional operating system defaults.

Splitting data-intensive tasks into their sub-tasks (see examples in Figure 3 and Figure 4) help in identifying the common sub-tasks across concurrent transactions or analytical queries. This, in turn, enables more opportunities for constructive instruction and data sharing, intra-task parallelism, and mapping a unit work to a core that would benefit the most from running on that core. Furthermore, it is an essential preliminary step to discover the frequent critical sub-tasks that justify building new specialized hardware for. Therefore, even though detecting the right granularity for sub-tasks and orchestrating more things at runtime is a big challenge, this challenge is worthwhile to address and study in more depth. It is the only way to answer the *what* question and aides answering the *where* question in the context of emerging heterogeneous hardware landscape.

After mapping a certain granularity of sub-tasks to the available processing units at runtime, one has to perform the actual scheduling and coordination of these sub-tasks efficiently. Otherwise, no matter how optimal the mapping is, it is not going to be beneficial. Specializing context switching or thread migrations for data-intensive tasks either at the level of the kernel or hardware has been tried (as also mentioned in Section 2.1). These techniques and others that specialize the same routines should be revisited in more detail in the context of heterogeneous hardware. It is highly likely that the common operating system layers and mechanisms will also evolve with such hardware. Therefore, it is important to have a holistic view while developing mechanisms to efficiently coordinate sub-tasks in order to achieve lightweight coordination and minimize replication of functionality across layers. In addition, exploiting more and more processing units should not turn the development of a data-intensive application into an unproductive process. Ensuring the correct and efficient instruction and data stream on a specific processing unit should be handled through high-level language primitives for the application developer and smart query compilation within the data management system [19]. Tackling these two challenges is the way to answer the *how* question for heterogeneous many cores.

Next, we discuss an end-to-end framework that takes the challenges of scheduling varying data-intensive tasks on heterogeneous hardware into account by mainly focusing on the resource estimation challenge, which also aides the *where* question.

4 A Framework for Running Data-Intensive Tasks on Emerging Hardware

Even though the previous sections focus on the utilization of a single server hardware, resource-aware scheduling is an active field of research often tailored specifically for different hardware platforms, from small embedded systems [38] up to clusters [9]. Executing tasks with varying resource demands in parallel can lead to inefficient resource utilization, especially on heterogeneous hardware. More precisely, the correlation between the resource demand of a task and its completion time is often highly non-linear, once the task is executed concurrently with other ones. In order to solve the very challenging problem of finding an optimal resource mapping for a single task, new resource-aware scheduling strategies are required, which efficiently map tasks to heterogeneous parallel architectures, taking their particular resource demands into account. Independent of the actual objective, i.e., making the execution of hybrid tasks more efficient or designing an efficient parallelization strategy for a specific task, the underlying motivation remains the same, namely, optimizing the execution of tasks on heterogeneous hardware architectures while respecting given latency requirements.

In data management systems, hybrid data-intensive tasks often arrive dynamically providing only inaccurate information about their resource utilization behavior. Unlike classical scheduling problems, such systems require mapping methods that (1) interact with a resource model to map tasks to suitable resources at runtime and (2) adjust mapping decisions dynamically depending on the system load.

Many existing work that tackle the problem of scheduling tasks on parallel architectures aim to optimize either the execution of tasks having similar characteristics (e.g., only transactional workloads) or the scheduling of tasks with hybrid characteristics (transactional and analytical workloads) on homogeneous parallel systems.

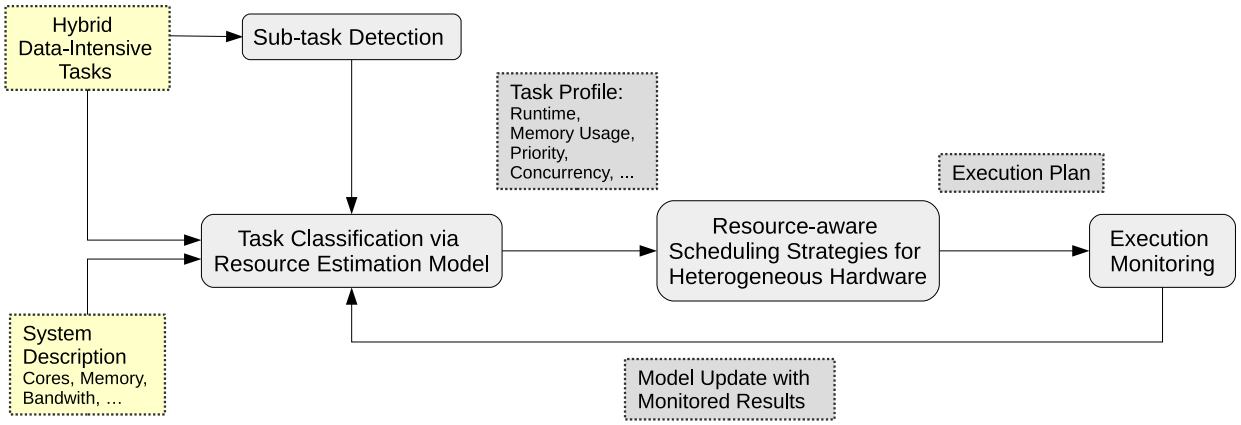


Figure 5: Framework for resource-aware scheduling strategies guided by a resource estimation model for hybrid data-intensive tasks executed on heterogeneous hardware.

For example, database systems such as SAP HANA or HyPer are designed to efficiently execute hybrid tasks using an optimized workload management system for servers with homogeneous cores [36]. Future scheduling strategies, however, should consider both the heterogeneity of workloads and of hardware in order to find an efficient resource-aware mapping exploiting the full potential of the underlying parallel architecture.

To enable a resource-efficient scheduling of data-intensive tasks over complex heterogeneous hardware architectures, we focus on a framework, illustrated in Figure 5, that includes scheduling mechanisms guided by resource estimation models. Regarding different aspects of our strategy, we survey previous approaches harvesting already existing results that shall serve as a guideline.

To generate resource estimation models distinct machine learning methods or analytical models can be applied to predict the resource demands for different possible task-to-core(s) mapping strategies. Analytical models can be more accurate than machine learning models when applied to estimate the runtime of concurrently executed tasks [41]. However, machine learning models are typically preferred in more complex heterogeneous hardware scenarios, since the complexity of the analytical models can increase rapidly in such cases. When building models that are capable of describing the complete data management system behavior, several aspects need to be taken into consideration. For models with high multidimensionality and thus high complexity, a dimensionality reduction, through machine learning techniques such as clustering or classification, is suggested [8]. For example, as shown by previous work [26], a workload forecasting strategy based on machine learning techniques can try to predict the expected arrival rate of certain types of tasks in a data management system and use clustering to reduce the model complexity. In general, machine learning methods, especially for resource estimations, are not only applicable to data management systems running on a single server. They can also be applied in the context of cluster management, e.g., to classify heterogeneous workloads that would achieve an efficient resource utilization [9].

Figure 5 visualizes the optimization cycle of the resource-aware scheduling framework. It consists of four main steps.

In the *first* step, the sub-task detection tries to identify possible sub-tasks from the incoming data-intensive tasks (transactional and different types of analytical) to take possible parallelization strategies into account.

To determine an efficient mapping of tasks to available hardware, it is necessary to be aware of each task's (estimated) resource demand. To this effect, a flexible resource estimation model is constructed in the *second* step, which estimates the resource demands of each task and sub-task based on previously executed similar tasks. Since the available hardware architecture influences the runtime behavior of a task, the model also uses the hardware description as input, and later should also be able to adapt to hardware changes. As mentioned above, such a model can utilize machine learning techniques instead of analytical cost models, since the analytical

models could become very complex when they have to cope with many different heterogeneous hardware parts in one system. Depending on the model's estimations, tasks can be classified into different types of groups based on their resource demands to later efficiently map them to suitable hardware resources or queues that are assigned to each group. Therefore, as an output the model produces task profiles including different resource utilization characteristics and an execution priority that can be used for latency critical tasks. A similar framework for resource-aware scheduling that focuses on scheduling parallel parameter optimization of machine learning algorithms with heterogeneous tasks have been studied in [22]. This framework uses a regression model to estimate the runtime of tasks and computes an execution priority for each task. This priority is then used as input to schedule these tasks in a way that minimizes CPU idling on homogeneous clusters. Further work on this framework showed that its runtime estimation mechanism also works for heterogeneous hardware. However, for heterogeneous hardware the random forest regression model is found to be more effective since task execution times form a discontinuous model because of the additional categorical variable that represents the processor type [21]. Besides runtime estimation, the estimation of multiple performance metrics via machine learning techniques for specific query plans on homogeneous hardware is proposed in [13]. In [28] a detailed overview of different machine learning techniques applicable for estimating multiple metrics for highly concurrent OLTP workloads on homogeneous systems is given, which could be interesting as well for resource estimations on heterogeneous systems.

As depicted in Figure 5, the obtained task profiles including multiple metrics serve as inputs for the resource-aware scheduling strategies in the *third* step. Resulting from this, an execution plan is created that efficiently maps tasks to suitable hardware resources. While creating the execution plan, this step also determines whether to use intra-task parallelism or the degree of parallelism for a task based on the available hardware resources and topology. As mentioned in Section 2.2, due to sub-task coordination efforts and non-uniform core-to-core communication costs, parallel execution plan of a task may not always result in faster execution compared to running a task serially.

Since the profiles are only estimated, under- or overestimation (e.g., of execution times) may occur. In such cases, a task may need to be rescheduled or stopped to guarantee latency requirements. This service is performed by the execution monitoring provided by the *last* step of our strategy. Here, an adaptive operator replacement technique using machine learning for runtime estimation, as presented in [6], could be applied, where operator mappings are dynamically adjusted on heterogeneous co-processors. Moreover, execution monitoring is also used to gather information about the system behavior, such as CPU or memory utilization, and to measure the de facto resource utilization of tasks at runtime. After a task or a group of tasks has finished their execution, the results are collected to iteratively refine the resource estimation model. Evidently, the quality of the scheduling strategy depends on the accuracy of the resource estimation. Hence, the model update entails more reliable estimations over time for the future predictions of the new incoming tasks.

5 Conclusion

In this article, we focused on scheduling data-intensive applications with different types of tasks over the processing units of emerging heterogeneous server hardware. Existing scheduling proposals that diverge from conventional methods when utilizing the resources of a single server with homogeneous multicores already give us essential insights. Therefore, their challenges should be revisited in the context of emerging heterogeneous hardware. More specifically, moving forward we should focus on the following: (1) identifying sub-tasks of data-intensive tasks across different applications, especially the common ones that allow constructive sharing of instructions and data, (2) efficient orchestration of these sub-tasks at runtime, (3) dynamic models to guide us during task-to-core(s) mapping decisions, and (4) a holistic approach across hardware, operating systems, and data management/processing systems to keep the systems' layers lightweight. This article navigated these items giving a high-level overview. Our goal is to tackle them in more detail in the future.

Acknowledgments

This work was partly supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876, projects C5 and A3. The authors would like to thank Jens Teubner, Philippe Bonnet, and Danica Porobic for providing valuable feedback.

References

[1] A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. *Databases on Modern Hardware: How to Stop Underutilization and Love Multicores*. Morgan & Claypool Publishers, 2017.

[2] G. Alonso and P. Bailis. Research for practice: FPGAs in datacenters. *CACM*, 61(9):48–49, 2018.

[3] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *MICRO*, pages 188–198, 2012.

[4] I. Atta, P. Tözün, X. Tong, A. Ailamaki, and A. Moshovos. STREX: Boosting Instruction Cache Reuse in OLTP Workloads through Stratified Transaction Execution. In *ISCA*, pages 273–284, 2013.

[5] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40:33–37, 2007.

[6] S. Breß, B. Köcher, M. Heimel, V. Markl, M. Saecker, and G. Saake. Ocelot/HyPE: Optimized Data Processing on Heterogeneous Hardware. *PVLDB*, 7(13):1609–1612, 2014.

[7] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In *ASPLOS*, pages 283–292, 2006.

[8] Y. Chen, A. Ganapathi, and R. Katz. Challenges and Opportunities for Managing Data Systems Using Statistical Models. In *IEEE DeBull*, volume 34, pages 53–60, 2011.

[9] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*, pages 127–144, 2014.

[10] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc. Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions. *IEEE J. Solid-State Circuits*, pages 256–268, 1974.

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, pages 365–376, 2011.

[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, pages 37–48, 2012.

[13] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, pages 592–603, 2009.

[14] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries With One Stone. *PVLDB*, 5(6):526–537, 2012.

[15] J. Giceva, G. Zellweger, G. Alonso, and T. Roscoe. Customized OS support for data-processing. In *DaMoN*, pages 2:1–2:6, 2016.

[16] J. Hamilton. Perspectives - Overall Data Center Costs. `https://perspectives.mvdirona.com/2010/09/overall-data-center-costs/`.

[17] S. Harizopoulos and A. Ailamaki. STEPS Towards Cache-Resident Transaction Processing. In *VLDB*, pages 660–671, 2004.

[18] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, pages 383–394, 2005.

[19] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *PVLDB*, 6(9):709–720, 2013.

[20] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *CACM*, 62(2):48–60, 2019.

[21] H. Kotthaus. *Methods for Efficient Resource Utilization in Statistical Machine Learning Algorithms*. PhD thesis, TU Dortmund University, 2018.

[22] H. Kotthaus, J. Richter, A. Lang, J. Thomas, B. Bischl, P. Marwedel, J. Rahnenführer, and M. Lang. RAMBO: Resource-Aware Model-Based Optimization with Scheduling for Heterogeneous Runtimes and a Comparison with Asynchronous Model-Based Optimization. In *LION*, pages 180–195, 2017.

[23] W. Lang, J. M. Patel, and S. Shankar. Wimpy Node Clusters: What About Non-Wimpy Workloads? In *DaMoN*, pages 47–55, 2010.

[24] J. R. Larus and M. Parkes. Using Cohort-Scheduling to Enhance Server Performance. In *USENIX*, pages 103–114, 2002.

[25] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[26] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD*, pages 631–645, 2018.

[27] G. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(6), 1965.

[28] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and Resource Modeling in Highly-concurrent OLTP Workloads. In *SIGMOD*, pages 301–312, 2013.

[29] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *ASPLOS*, pages 2–11, 1996.

[30] F. Özcan, Y. Tian, and P. Tözün. Hybrid Transactional/Analytical Processing: A Survey. In *SIGMOD*, pages 1771–1775, 2017.

[31] I. Pandis, P. Tözün, M. Branco, D. Karampinas, D. Porobic, R. Johnson, and A. Ailamaki. A Data-Oriented Transaction Execution Engine and Supporting Tools. In *SIGMOD*, pages 1237–1240, 2011.

[32] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-Free Shared-Everything OLTP. *PVLDB*, 4(10):610–621, 2011.

[33] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive Transaction Processing on Hardware Islands. In *ICDE*, pages 688–699, 2014.

[34] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing Data and Work Across Concurrent Analytical Queries. *PVLDB*, 6(9):637–648, 2013.

[35] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *PVLDB*, 8:1442–1453, 2015.

[36] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K.-U. Sattler. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *TPCTC*, pages 97–112, 2014.

[37] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*, pages 387–402, 2016.

[38] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource-Aware Task Scheduling. *ACM TECS*, 14(1):5:1–5:25, 2015.

[39] P. Tözün, I. Atta, A. Ailamaki, and A. Moshovos. ADDICT: Advanced Instruction Chasing for Transactions. *PVLDB*, 7(14):1893–1904, 2014.

[40] TPC Benchmark C Standard Specification. `http://www.tpc.org/tpcc`.

[41] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *PVLDB*, 6(10):925–936, 2013.

[42] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving Database Performance on Simultaneous Multithreading Processors. In *VLDB*, pages 49–60, 2005.

Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method

Viktor Leis, Michael Haubenschild*, Thomas Neumann
Technische Universität München Tableau Software*
{leis,neumann}@in.tum.de mhaubenschild@tableau.com*

Abstract

As the number of cores on commodity processors continues to increase, scalability becomes more and more crucial for overall performance. Scalable and efficient concurrent data structures are particularly important, as these are often the building blocks of parallel algorithms. Unfortunately, traditional synchronization techniques based on fine-grained locking have been shown to be unscalable on modern multi-core CPUs. Lock-free data structures, on the other hand, are extremely difficult to design and often incur significant overhead.

In this work, we make the case for Optimistic Lock Coupling as a practical alternative to both traditional locking and the lock-free approach. We show that Optimistic Lock Coupling is highly scalable and almost as simple to implement as traditional lock coupling. Another important advantage is that it is easily applicable to most tree-like data structures. We therefore argue that Optimistic Lock Coupling, rather than a complex and error-prone custom synchronization protocol, should be the default choice for performance-critical data structures.

1 Introduction

Today, Intel's commodity server processors have up to 28 cores and its upcoming microarchitecture will have up to 48 cores per socket [6]. Similarly, AMD currently stands at 32 cores and this number is expected to double in the next generation [20]. Since both platforms support simultaneous multithreading (also known as hyperthreading), affordable commodity servers (with up to two sockets) will soon routinely have between 100 and 200 hardware threads.

With such a high degree of hardware parallelism, efficient data processing crucially depends on how well concurrent data structures scale. Internally, database systems use a plethora of data structures like table heaps, internal work queues, and, most importantly, index structures. Any of these can easily become a scalability (and therefore overall performance) bottleneck on many-core CPUs.

Traditionally, database systems synchronize internal data structures using fine-grained reader/writer locks[1]. Unfortunately, while fine-grained locking makes lock contention unlikely, it still results in bad scalability because

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]In this work, we focus on data structure synchronization rather than high-level transaction semantics and therefore use the term *lock* for what would typically be called *latch* in the database literature. We thus follow common computer science (rather than database) terminology.

lock acquisition and release require writing to shared memory. Due to the way cache coherency is implemented on modern multi-core CPUs, these writes cause additional cache misses[2] and the cache line containing the lock's internal data becomes a point of physical contention. As a result, any frequently-accessed lock (e.g., the lock of the root node of a B-tree) severely limits scalability.

Lock-free data structures like the Bw-tree [15] (a lock-free B-tree variant) or the Split-Ordered List [19] (a lock-free hash table) do not acquire any locks and therefore generally scale much better than locking-based approaches (in particular for read-mostly workloads). However, lock-free synchronization has other downsides: First, it is very difficult and results in extremely complex and error-prone code (when compared to locking). Second, because the functionality of atomic primitives provided by the hardware (e.g., atomically compare-and-swap 8 bytes) is limited, complex operations require additional indirections within the data structure. For example, the Bw-tree requires an indirection table and the Split-Ordered List requires "dummy nodes", resulting in overhead due to additional cache misses.

In this paper we make the case for *Optimistic Lock Coupling (OLC)*, a synchronization method that combines some of the best properties of lock-based and lock-free synchronization. OLC utilizes a special lock type that can be used in two modes: The first mode is similar to a traditional mutex and excludes other threads by physically acquiring the underlying lock. In the second mode, reads can proceed optimistically by validating a version counter that is embedded in the lock (similar to optimistic concurrency control). The first mode is typically used by writers and the second mode by readers. Besides this special lock type, OLC is based on the observation that optimistic lock validations can be interleaved/coupled—similar to the pair-wise interleaved lock acquisition of traditional lock coupling. Hence, the name Optimistic Lock Coupling.

OLC has a number of desirable features:

- By reducing the number of writes to shared memory locations and thereby avoiding cache invalidations, it **scales well** for most workloads.
- In comparison to unsynchronized code, it requires few additional CPU instructions making it **efficient**.
- OLC is **widely applicable** to different data structures. It has already been successfully used for synchronizing binary search trees [4], tries [14], trie/B-tree hybrids [17], and B-trees [22].
- In comparison to the lock-free paradigm, it is also **easy to use** and requires few modifications to existing, single-threaded data structures.

Despite these positive features and its simplicity, OLC is not yet widely known. The goal of this paper is therefore to popularize this simple idea and to make a case for it. We argue that OLC deserves to be widely known. It is a good default synchronization paradigm—more complex, data structure-specific protocols are seldom beneficial.

The rest of the paper is organized as follows. Section 2 discusses related work, tracing the history of OLC and its underlying ideas in the literature. The core of the paper is Section 3, which describes the ideas behind OLC and how it can be used to synchronize complex data structures. In Section 4 we experimentally show that OLC has low overhead and scales well when used to synchronize an in-memory B-tree. We summarize the paper in Section 5.

[2]The cache coherency protocol ensures that all copies of a cache line on other cores are invalidated before the write can proceed.

2 Related Work

Lock coupling has been proposed as a method for allowing concurrent operations on B-trees in 1977 [2]. This traditional and still widely-used method, described in detail in Graefe's B-tree survey [8], is also called "latch coupling", "hand-over-hand locking", and "crabbing". Because at most two locks are held at-a-time during tree traversal, this technique seemingly allows for a high degree of parallelism—in particular if read/write locks are used to enable inner nodes to be locked in shared mode. However, as we show in Section 4, on modern hardware lock acquisition (even in shared mode) results in suboptimal scalability.

An early alternative from 1981 is a B-tree variant called B-link tree [10], which only holds a single lock at a time. It is based on the observation that between the release of the parent lock and the acquisition of the child lock, the only "dangerous" thing that could have happened is the split of a child node (assuming one does not implement merge operations). Thus, when a split happens, the key being searched might end up on a neighboring node to the right of the current child node. A B-link tree traversal therefore detects this condition and, if needed, transparently proceeds to the neighboring node. Releasing the parent lock early is highly beneficial when the child node needs to be fetched from disk. For in-memory workloads, however, the B-link tree has the same scalability issues as lock coupling (it acquires just as many locks).

The next major advance, Optimistic Latch-Free Index Traversal (OLFIT) [5], was proposed in 2001. OLFIT introduced the idea of a combined lock/update counter, which we call *optimistic lock*. Based on these per-node optimistic locks and the synchronization protocol of the B-link tree, OLFIT finally achieves good scalability on parallel processors. The OLFIT protocol is fairly complex, as it requires both the non-trivial B-link protocol and optimistic locks. Furthermore, like the B-link tree protocol, it does not support merging nodes, and is specific to B-trees (cannot easily be applied to other data structures).

In the following two decades, the growth of main-memory capacity led to much research into other data structures besides the venerable B-tree. Particularly relevant for our discussion is Bronson et al.'s [4] concurrent binary search tree, which is based on optimistic version validation and has a sophisticated, data structure-specific synchronization protocol. To the best of our knowledge, this 2010 paper is the first that, as part of its protocol, interleaves version validation across nodes—rather than validating each node separately like OLFIT. In that paper, this idea is called "hand-over-hand, optimistic validation", while we prefer the term Optimistic Lock Coupling to highlight the close resemblance to traditional lock coupling. Similarly, Mao et al.'s [17] Masstree (a concurrent hybrid trie/B-tree) is also based on the same ideas, but again uses them as part of a more complex protocol.

The Adaptive Radix Tree (ART) [12] is another recent in-memory data structure, which we proposed in 2013. In contrast to the two data structures just mentioned, it was originally designed with single-threaded performance in mind without supporting concurrency. To add support for concurrency, we initially started designing a custom protocol called Read-Optimized Write Exclusion (ROWEX) [14], which turned out to be non-trivial and requires modifications of the underlying data structure[3]. However, fairly late in the project, we also realized, that OLC *alone* (rather than as part of a more complex protocol) is sufficient to synchronize ART. No other changes to the data structure were necessary. Both approaches were published and experimentally evaluated in a followup paper [14], which shows that, despite its simplicity, OLC is efficient, scalable, and generally outperforms ROWEX.

Similar results were recently published regarding B-trees [22]. In this experimental study a simple OLC-based synchronization outperformed the Bw-tree [15], a complex lock-free synchronization approach. Another recent paper shows that for write-intensive workloads, locking often performs better than lock-free synchronization [7]. These experiences indicate that OLC is a general-purpose synchronization paradigm and motivate the current paper.

[3]Note that ROWEX is already easier to apply to existing data structures than the lock-free approach. The difficulty depends on the data structure. Applying ROWEX is hard for B-trees with sorted keys and fairly easy for copy-on-write data structures like the Height Optimized Trie [3]—with ART being somewhere in the middle.

3 Optimistic Lock Coupling

The standard technique for inter-thread synchronization is mutual exclusion using fine-grained locks. In a B-tree, for example, every node usually has its own associated lock, which is acquired before accessing that node. The problem of locking on modern multi- and many-core processors is that lock acquisition and release require writing to the shared memory location that implements the lock. This write causes exclusive ownership of the underlying cache line and invalidates copies of it on all other processor cores. For hierarchical, tree-like data structures, the lock of the root node becomes a point of physical contention—even in read-only workloads and even when read/write locks are used. Depending on the specific data structure, number of cores, cache coherency protocol implementation, cache topology, whether Non-Uniform Memory Access (NUMA) is used, locking can even result in multi-threaded performance that is worse than single-threaded execution.

The inherent pessimism of locking is particularly unfortunate for B-trees: Despite the fact that logical modifications of the root node are very infrequent, every B-tree operation must lock the root node during tree traversal[4]. Even the vast majority of update operations (with the exception of splits and merges), only modify a single leaf node. These observations indicate that a more optimistic approach, which does not require locking inner nodes, would be very beneficial for B-trees.

3.1 Optimistic Locks

As the name indicates, optimistic locks try to solve the scalability issues of traditional locks using an optimistic approach. Instead of always physically acquiring locks, even for nodes that are unlikely to be modified simultaneously, after-the-fact validation is used to detect conflicts. This is done by augmenting each lock with a version/update counter that is incremented on every modification. Using this version counter, readers can optimistically proceed before validating that the version did not change to ensure that the read was safe. If validation fails, the operation is restarted.

Using optimistic locks, a read-only node access (i.e., the majority of all operations in a B-tree) does not acquire the lock and does not increment the version counter. Instead, it performs the following steps:

1. read lock version (restart if lock is not free)
2. access node
3. read the version again and validate that it has not changed in the meantime

If the last step (the validation) fails, the operation has to be restarted. Write operations, on the other hand, are more similar to traditional locking:

1. acquire lock (wait if necessary)
2. access/write to node
3. increment version and unlock node

Writes can therefore protect a node from other writes.

As we observed in an earlier paper [14], because of similar semantics, optimistic locks can be hidden behind an API very similar to traditional read/write locks. Both approaches have an exclusive lock mode, and acquiring a traditional lock in shared mode is analogous to optimistic version validation. Furthermore, like with some implementations of traditional read/write locks, optimistic locks allow upgrading a shared lock to an exclusive lock. Lock upgrades are, for example, used to avoid most B-tree update operations from having to lock inner nodes. In our experience, the close resemblance of optimistic and traditional locks simplifies the reasoning about optimistic locks; one can apply similar thinking as in traditional lock-based protocols.

[4] To a lesser extent this obviously applies to all inner nodes, not just the root.

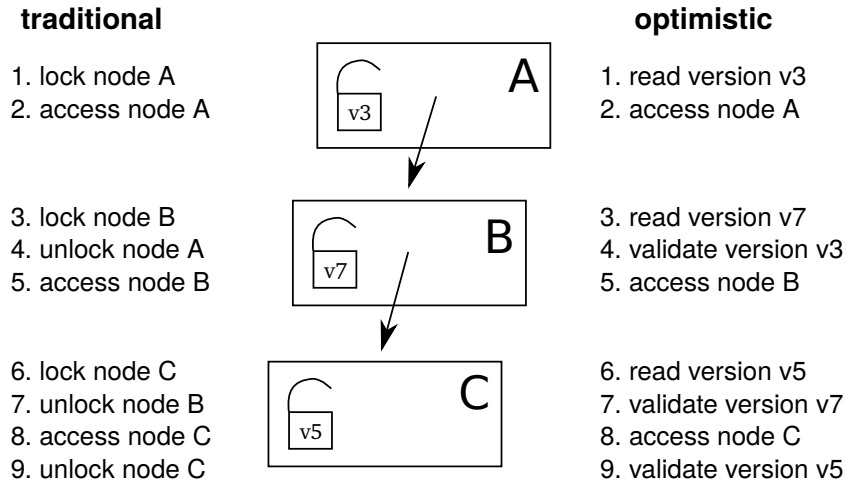


Figure 1: Comparison of a lookup operation in a 3-level tree using traditional lock coupling (left-hand side) vs. optimistic lock coupling (right-hand side).

3.2 Lock Coupling with Optimistic Locks

The traditional and most common lock-based synchronization protocol for B-trees is lock coupling, which interleaves lock acquisitions while holding at most two locks at a time. If, as we observed earlier, optimistic locks have similar semantics as traditional locks, it is natural to ask whether lock coupling can be combined with optimistic locks. And indeed the answer is yes: One can almost mechanically translate traditional lock coupling code to optimistic lock coupling code. This is illustrated in Figure 1, which compares the traversal in a tree of height 3 using traditional and optimistic locks. As the figure shows, the main difference is that locking is translated to reading the version and that unlocking becomes validation of the previously read version. This simple change provides efficient lock-free tree traversal without the need to design a complex synchronization protocol.

It is important to emphasize the conceptual simplicity of OLC in comparison to data structures that use custom protocols like the Bw-tree [15]. To implement lock-free access, the Bw-tree requires an indirection table, delta nodes, complex splitting and merging logic, retry logic, etc. OLC, on the other hand, can directly be applied to B-trees mostly by adding the appropriate optimistic locking code and without modifying the node layout itself. Therefore, OpenBw-Tree, an open source implementation of the Bw-tree, requires an order of magnitude more code than a B-tree based on OLC[5]. Given how difficult it is to develop, validate, and debug lock-free code, simplicity is obviously a major advantage.

3.3 Correctness Aspects

So far, we have introduced the high-level ideas behind OLC and have stressed its similarity to traditional lock coupling. Let us now discuss some cases where the close similarity between lock coupling and OLC breaks down. To make this more concrete, we show the B-tree lookup code in Figure 2. In the code, `readLockOrRestart` reads the lock version and `readUnlockOrRestart` validates that the read was correct.

One issue with OLC is that any pointer speculatively read from a node may point to invalid memory (if that node is modified concurrently). Dereferencing such a pointer (e.g., to read its optimistic lock), may cause a segmentation fault or undefined behavior. In the code shown in Figure 2, this problem is prevented by the extra check in line 25, which ensures that the read from the node containing the pointer was correct. Without this

[5]Both implementations are available on GitHub: `https://github.com/wangziqi2016/index-microbench`

```
1  std::atomic<BTreeNode*> root;
2
3  // search for key in B+tree, returns payload in resultOut
4  bool lookup(Key key, Value& resultOut) {
5     BTreeNode* node = root.load();
6     uint64_t nodeVersion = node->readLockOrRestart();
7     if (node != root.load()) // make sure the root is still the root
8        restart();
9
10    BTreeInner<Key>* parent = nullptr;
11    uint64_t parentVersion = 0;
12
13    while (node->isInner()) {
14       auto inner = (BTreeInner*)node;
15
16       // unlock parent and make current node the parent
17       if (parent)
18          parent->readUnlockOrRestart(parentVersion);
19       parent = inner;
20       parentVersion = nodeVersion;
21
22       // search for next node
23       node = inner->findChild(key);
24       // validate 'inner' to ensure that 'node' pointer is valid
25       inner->checkOrRestart(nodeVersion);
26       // now it safe to dereference 'node' pointer (read its version)
27       nodeVersion = node->readLockOrRestart();
28    }
29
30    // search in leaf and retrieve payload
31    auto leaf = (BTreeLeaf*)node;
32    bool success = leaf->findValue(key, resultOut);
33
34    // unlock everything
35    if (parent)
36       parent->readUnlockOrRestart(parentVersion);
37    node->readUnlockOrRestart(nodeVersion);
38
39    return success;
40 }
```

Figure 2: B-tree lookup code using OLC. For simplicity, the restart logic is not shown.

additional validation, the code would in line 27 dereference the pointer speculatively read in line 23. Note that the implementation of `checkOrRestart` is actually identical to `readUnlockOrRestart`. We chose to give it a different name to highlight the fact that this extra check would not be necessary with read/write locks.

Another potential issue with optimistic locks is code that does not terminate. Code that speculatively accesses a node, like an intra-node binary search, should be written in a way such that it always terminates—even in the presence of concurrent writes. Otherwise, the validation code that detects the concurrent write will never run. The binary search of a B-tree, for example, needs to be written in such a way that each comparison makes progress. For some data structures that do not require loops in the traversal code (like ART) termination is trivially true.

3.4 Implementation Details

To implement an optimistic lock, one can combine the lock and the version counter into a single 64-bit[6] word [14]. A typical read operation will therefore merely consist of reading this version counter atomically. In C++11 this can be implemented using the `std::atomic` type.

On x86, atomic reads are cheap because of x86's strong memory order guarantees. No memory fences are required for sequentially-consistent loads, which are translated (by both GCC and clang) into standard `MOV` instructions. Hence, the only effect of `std::atomic` for loads is preventing instruction re-ordering. This makes version access and validation cheap. Acquiring and releasing an optimistic lock in exclusive mode has comparable cost to a traditional lock: A fairly expensive sequentially-consistent store is needed for acquiring a lock, while a standard `MOV` suffices for releasing it. A simple sinlock-based implementation of optimistic locks can be found in the appendix of an earlier paper [14].

OLC code must be able to handle restarts since validation or lock upgrade can fail due to concurrent writers. Restarts can easily be implemented by wrapping the data structure operation in a loop (for simplicity not shown in Figure 2). Such a loop also enables limiting the number of optimistic retry operations and falling back to pessimistic locking in cases of very heavy contention. The ability to fall back to traditional locking is a major advantage of OLC in terms of robustness over lock-free approaches, which do not have this option.

In addition to the optimistic shared mode and the exclusive mode, optimistic locks also support a "shared pessimistic" mode, which physically acquires the lock in shared mode (allowing multiple concurrent readers but no writers). This mode is useful for table (or range) scans that touch many tuples on a leaf page (which would otherwise easily abort). Finally, let us mention that large range scans and table scans, should be broken up into several per-node traversals as is done in the LeanStore [11] system.

Like all lock-free data structures, but unlike traditional locking and Hardware Transactional Memory [9, 16, 13], OLC requires care when deleting (and reusing) nodes. The reason is that a deleting thread can never be sure that a node can be reclaimed because other threads might still be optimistically reading from that node. Therefore, standard solutions like epoch-based reclamation [21], hazard pointers [18], or optimized hazard pointers [1] need to be used. These memory reclamation techniques are, however, largely orthogonal to the synchronization protocol itself.

[6] Even after subtracting one bit for the lock status, a back-of-the-envelope calculation can show that 63 bits are large enough to never overflow in practice.

Table 4: Performance and CPU counters for lookup and insert operations in a B-tree with 100M keys. We perform 100M operations and normalize the CPU counters by that number.

	threads	M op/s	cycles	instruc-tions	L1 misses	L3 misses	branch misses
lookup (no sync.)	1	1.72	2028	283	39.1	14.9	16.1
lookup (OLC)	1	1.65	2107	370	43.9	15.1	16.7
lookup (lock coup.)	1	1.72	2078	365	42.3	16.9	15.7
insert (no sync.)	1	1.51	2286	530	59.8	31.1	17.3
insert (OLC)	1	1.50	2303	629	61.2	31.1	16.5
insert (lock coup.)	1	1.41	2473	644	61.0	31.0	17.2
lookup (no sync.)	10	15.48	2058	283	38.6	15.5	16.0
lookup (OLC)	10	14.60	2187	370	43.8	15.8	16.8
lookup (lock coup.)	10	5.71	5591	379	54.2	17.0	14.8
insert (no sync.)	10	-	-	-	-	-	-
insert (OLC)	10	10.46	2940	656	62.0	32.5	16.8
insert (lock coup.)	10	7.55	4161	667	75.0	28.6	16.2

4 Evaluation

Let us now experimentally evaluate the overhead and scalability of OLC. For the experiments, we use an in-memory B+tree implemented in C++11 using templates, which is configured to use nodes of 4096 bytes, random 8 byte keys, and 8 byte payloads. Based on this B-tree, we compare the following synchronization approaches:

- an OLC implementation[7]
- a variant based on traditional lock coupling and read/write locks
- the unsynchronized B-tree, which obviously is only correct for read-only workloads but allows measuring the overhead of synchronization

Note that earlier work has compared the OLC implementation with a Bw-tree implementation [22] and other state-of-the-art in-memory index structures.

We use a Haswell EP system with an Intel Xeon E5-2687W v3 CPU, which has 10 cores (20 "Hyper-Threads") and 25 MB of L3 cache. The system is running Ubuntu 18.10 and we use GCC 8.2.0 to compile our code. The CPU counters are obtained using the Linux perf API[8].

Table 4 compares the performance and CPU counters for lookup and insert operations in a B-tree with 100M keys. With *single-threaded* execution, we observe that all three approaches have very similar performance. Adding traditional or optimistic locks to unsynchronized B-tree code results in up to 30% of additional instructions without affecting single-threaded performance much.

As Figure 3 shows, the results change dramatically once we use multiple threads. For lookup, the scalability of OLC is near-linear up to 20 threads, even though the system has only 10 "real cores". The OLC scalability for insert is also respectable (though not quite as linear because multi-threaded insertion approaches the memory bandwidth of our processor). The figure also shows that the results of traditional lock coupling with read/write

[7] An almost identical OLC implementation is available on github: `https://github.com/wangziqi2016/index-microbench/tree/master/BTreeOLC`

[8] We use the following convenience wrapper: `https://github.com/viktorleis/perfevent`

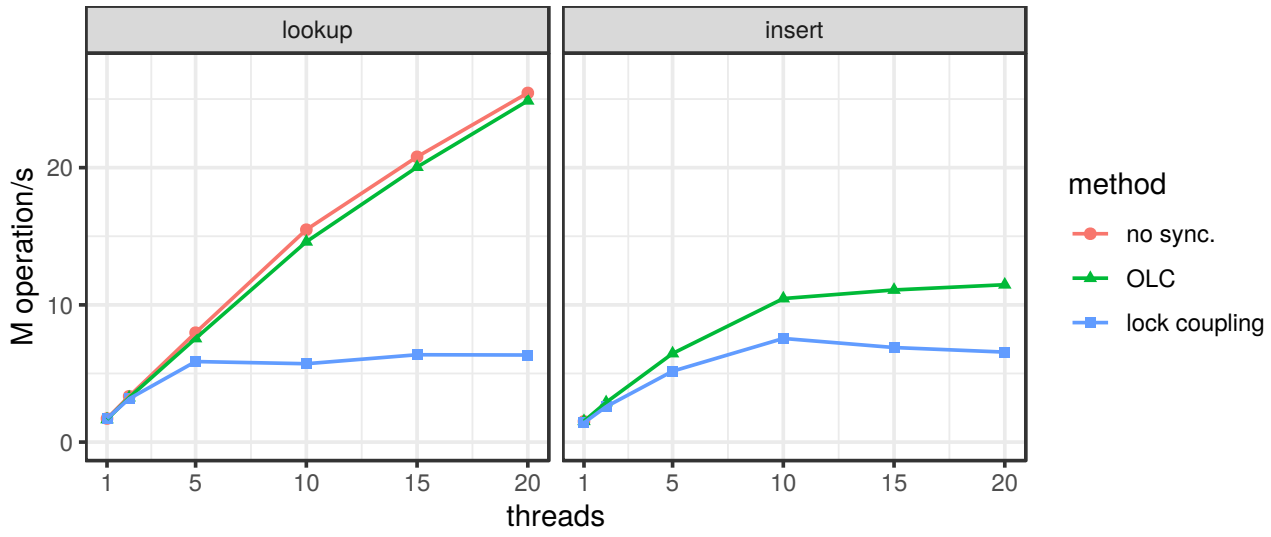


Figure 3: Scalability on 10-core system for B-tree operations (100M values).

locks are significantly worse than OLC. With 20 threads, lookup with OLC is 3.9× faster than traditional lock coupling.

5 Summary

Optimistic Lock Coupling (OLC) is an effective synchronization method that combines the simplicity of traditional lock coupling with the superior scalability of lock-free approaches. OLC is widely applicable and has already been successfully used to synchronize several data structures, including B-trees, binary search trees, and different trie variants. These features make it highly attractive for modern database systems as well as performance-critical systems software in general.

References

[1] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *SPAA*, 2016.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9, 1977.

[3] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *SIGMOD*, 2018.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPOPP*, 2010.

[5] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, 2001.

[6] I. Cutress. Intel goes for 48-cores: Cascade-AP with multi-chip package coming soon. https://www.anandtech.com/show/13535/intel-goes-for-48cores-cascade-ap, 2018 (accessed January, 2019).

[7] J. M. Faleiro and D. J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *CIDR*, 2017.

[8] G. Graefe. Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4), 2011.

[9] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with Intel® transactional synchronization extensions. In *HPCA*, 2014.

[10] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4), 1981.

[11] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. Leanstore: In-memory data management beyond main memory. In *ICDE*, 2018.

[12] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.

[13] V. Leis, A. Kemper, and T. Neumann. Scaling HTM-supported database transactions to many cores. *IEEE Trans. Knowl. Data Eng.*, 28(2), 2016.

[14] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *DaMoN*, 2016.

[15] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, 2013.

[16] D. Makreshanski, J. J. Levandoski, and R. Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *PVLDB*, 8(11), 2015.

[17] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[18] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 2004.

[19] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3), 2006.

[20] A. Shilov. AMD previews EPYC 'Rome' processor: Up to 64 Zen 2 cores. https://www.anandtech.com/show/13561/amd-previews-epyc-rome-processor-up-to-64-zen-2-cores, 2018 (accessed January, 2019).

[21] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[22] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. Andersen. Building a Bw-tree takes more than just buzz words. In *SIGMOD*, 2018.

Call for Papers

35th IEEE International Conference on Data Engineering

8-12 April 2019, Macau SAR, China

The annual IEEE International Conference on Data Engineering (ICDE) addresses research issues in designing, building, managing and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences. The 35th ICDE will take place at Macau, China, a beautiful seaside city where the old eastern and western cultures as well as the modern civilization are well integrated.

Topics of Interest

We encourage submissions of high quality original research contributions in the following areas. We also welcome any original contributions that may cross the boundaries among areas or point in other novel directions of interest to the database research community:

- Benchmarking, Performance Modelling, and Tuning
- Data Integration, Metadata Management, and Interoperability
- Data Mining and Knowledge Discovery
- Data Models, Semantics, Query languages
- Data Provenance, cleaning, curation
- Data Science
- Data Stream Systems and Sensor Networks
- Data Visualization and Interactive Data Exploration
- Database Privacy, Security, and Trust
- Distributed, Parallel and P2P Data Management
- Graphs, RDF, Web Data and Social Networks
- Database technology for machine learning
- Modern Hardware and In-Memory Database Systems
- Query Processing, Indexing, and Optimization
- Search and Information extraction
- Strings, Texts, and Keyword Search
- Temporal, Spatial, Mobile and Multimedia
- Uncertain, Probabilistic Databases
- Workflows, Scientific Data Management

Important Dates

For the first time in ICDE, ICDE2019 will have two rounds' submissions. All deadlines in the following are 11:59PM US PDT.

First Round:

Abstract submission due: May 25, 2018
Submission due: June 1, 2018
Notification to authors (Accept/Revise/Reject): August 10, 2018
Revisions due: September 7, 2018
Notification to authors (Accept/Reject): September 28, 2018
Camera-ready copy due: October 19, 2018

Second Round:

Abstract submission due: September 28, 2018
Submission due: October 5, 2018
Notification to authors (Accept/Revise/Reject): December 14th, 2018
Revisions due: January 11th, 2019
Notification to authors (Accept/Reject): February 1, 2019
Camera-ready copy due: February 22, 2019

General Co-Chairs
Christian S. Jensen, Aalborg University
Lionel M. Ni, University of Macau
Tamer Özsu, University of Waterloo

PC Co-Chairs
Wenfei Fan, University of Edinburgh
Xuemin Lin, University of New South Wales
Divesh Srivastava, AT&T Labs Research

For more details: http://conferences.cis.umac.mo/icde2019/

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314

Non-profit Org.
U.S. Postage
PAID
Los Alamitos, CA
Permit 1398