

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

Copyright 2019 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

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.

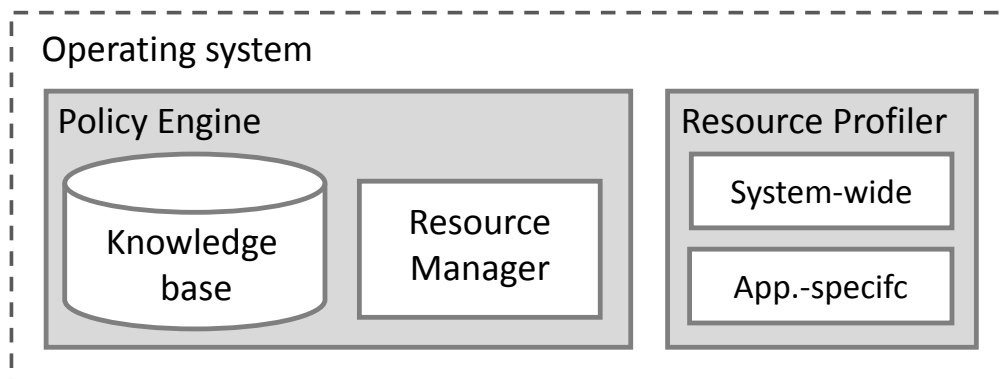


Figure 1: Policy Engine

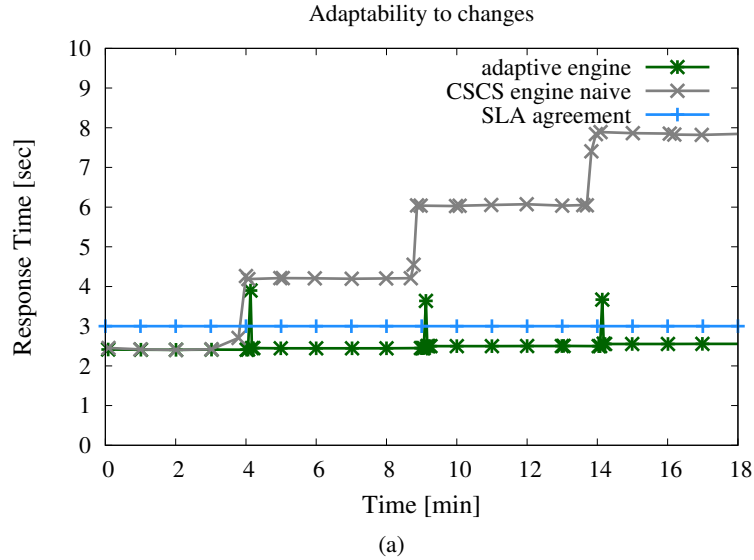
4 OS policy engine

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 [WIPS]	Response Time[s]		
			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.

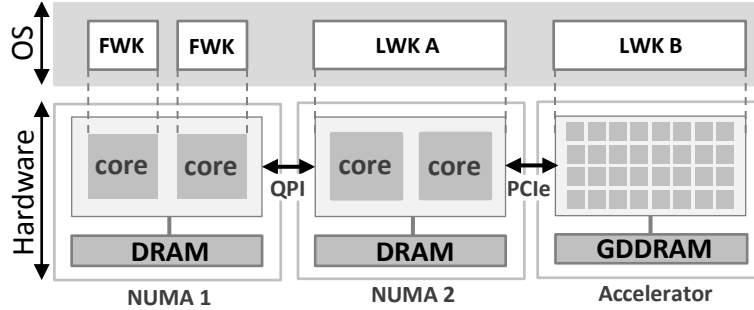


Figure 3: Illustrating Badis – an adaptive OS architecture, based on the multikernel model. The cores on NUMA node 1 each execute a separate kernel of the full-weight kernel (FWK). The cores on NUMA node 2 execute a common instance of a specialized light-weight kernel (LWK A). The computation units on 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 stack 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.

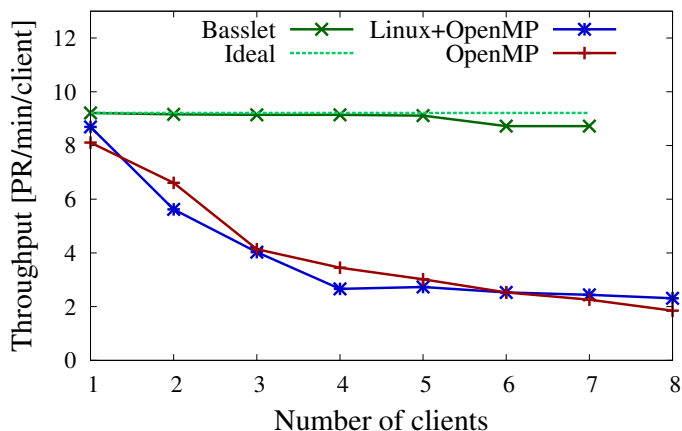


Figure 4: Throughput scale-out when executing multiple PRs using internal OpenMP parallelism vs. Linux+OpenMP scheduler vs. kernel-integrated runtime, as compared to ideal scale-out.

- 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

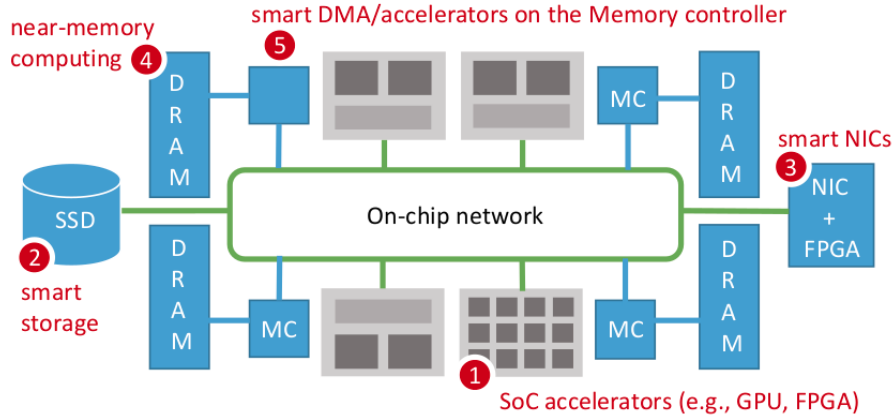


Figure 5: Active heterogeneous hardware

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. Singhanian. 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.