Towards Database / Operating System Co-Design

Jana Giceva, Adrian Schüpbach, Gustavo Alonso, Timothy Roscoe
Systems Group, ETH Zurich

ABSTRACT

Trends in multicore processors pose serious structural challenges to system software such as databases and operating systems. In this paper we revisit the decades-old problem of the interaction between a database and an operating system in the new context of the changes imposed on both by multicore architectures. Using existing prototypes of a multicore database and multicore operating system, we explore how they can efficiently interact so that the database can make optimal placement and deployment decisions without knowledge of the machine architecture. Our results show superior memory and interconnect utilization, and dramatic performance improvements in the presence of competing system tasks.

1. INTRODUCTION

In this paper, we report on early efforts to rethink the relationship between databases and operating systems in the light of modern hardware trends. Multicore architectures represent a significant departure from traditional mainstream hardware platforms. The intrinsic parallelism of multicores is at odds with the synchronization approach to concurrency of modern system software, including databases and operating systems [7, 17].

Furthermore, increasing heterogeneity both within and between multicore platforms poses additional problems: optimal use of resources requires detailed knowledge of the underlying hardware (memory affinities, cache hierarchies, interconnect distances, CPU/-core/socket layouts, etc.), aggravated by the increasing diversity of machines in the marketplace.

As a result, significant efforts are being devoted to redesigning system software for multicore architectures. We believe that the redesign of operating systems and databases is an excellent opportunity to revisit the decades-old problem [10] of database-operating system co-design.

Before tackling many of the areas where databases and operating systems have collided in the past (scheduling, thread management, I/O control, paging, memory allocation, etc.), we see the question of where and what maintains accurate information on the underlying hardware architecture as a key step towards a tighter co-design of databases and operating systems. This is based on the following observations: (1) Both databases and operating systems should be re-architected to exploit the intrinsic parallelism of multicore hardware. Such a system needs to be aware of the many architectural characteristics. (2) Along with the number of cores, heterogeneity both among cores and machines will increase. A database then needs to maintain accurate configuration and deployment information for every possible architecture where it could run. (3) With multicore machines being used for more purposes than single application servers, load interaction with other applications or even OS-tasks makes it difficult for the DB to implement elastic deployments. Such flexibility implies that the DB will require an accurate picture of the run time state of the whole machine.

In this paper we start to explore how a database and an operating system can interact to make the best possible use of the available resources and, as a first step, focus on memory and CPU resources. Our vehicle for this work is a new integrated system called *Cod*.

Cod delegates the task of maintaining accurate models of the underlying hardware and the current machine status to the OS. Above this, rather than spawning threads and data partitions according to static configuration or heuristics, Cod's DB instead uses a novel interface between the DB and the OS to state up front what it needs (degree of multiprogramming, size of memory, affinities between memory and processing units). In response, the OS produces an optimal mapping of the request to the underlying hardware.

The OS/DB interface in Cod supports a wide range of interactions, and will expand as we explore the design space. Currently it can be used to inquire about available resources, recommended partitioning strategies, mappings for a given set of jobs and associated memory access requirements, and the current status of the system. Based on this, the database can optimize queries and restructure itself so as to be able to meet concrete service level agreements of throughput or response time.

Initial experiments are encouraging, and show that Cod can efficiently use a diversity of machine configurations under a variety of different workloads (both itself and other applications sharing the machine).

2. MOTIVATION AND RELATED WORK

Database management systems (DBMS) and operating systems have a long history of conflicts. Nearly 35 years ago, Jim Gray pointed out the significant gains that could be obtained by designing the database and the operating system [10] together. Unfortunately, the same difficulties pointed out by Gray at the time remain the reason why databases today run on top of general purpose operating systems.

Despite this, databases are an interesting case for application-OS co-design since they are highly concerned with resource management issues. Considerable research has gone into allowing databases

to predict their own performance based on workload, as the basis for cost-based query optimization.

To our knowledge, there is little work today, from either side, on improving the interactions between databases and operating systems. Part of the reason may be that databases, as large enterprise application *par excellence*, usually run on a dedicated machine. Such a static resource allocation has allowed database designers to work around most of the barriers to system knowledge presented by commodity operating systems by simply bypassing the OS. Thus, modern databases do most of their own memory allocation and thread scheduling, work directly on raw disk partitions, and implement complex strategies to optimize and avoid I/O synchronization problems (e.g., in logging). Less widely acknowledged but also a telling sign of the gap between the DB and the OS is the fact that databases have gone as far as implementing their own type and data formats (such as Oracle's number representation) due to the inefficiencies of general purpose solutions. Today we would argue that there is essentially no meaningful interaction between the database and the operating system.

However, the stalemate arrived at over the last decades between the OS and the DB is now quickly changing. Not only multicore, but also developments like virtualization, network attached storage, large main memories and higher memory bandwidth, radical changes in the memory hierarchy (multicore caches, SSD devices, and -in the near future- persistent random access memory), and the need for more power- and space-efficient use of computing resources make it impossible to ignore the need for a richer DB/OS interface. We may have reached the limits of complexity that database engines (already somewhat bloated and facing their own scaling problems) can absorb while trying to second-guess and bypass the operating system.

At the root of many of these problems is, unsurprisingly, the allocation of resources to particular tasks. As the number of cores increases and the memory/cache hierarchies and affinities become more diverse; choosing good placement and allocation of resources is both more challenging and more important for performance.

Conventional database engines are designed for thread concurrency but not parallelism (see below), and typically exploit little information about the underlying hardware. Representing and using such information efficiently is complicated by the diversity of modern hardware: machines exhibit many different ways to group CPUs, cores, and caches, and connect them via a wide range of topologies. Furthermore, in an age of virtualization and server consolidation, databases may not be able to assume that they have a complete physical machine to themselves. As we show in Section 5.2, even small OS-related tasks can impact the performance of a scalable database if the DBMS is unaware of them.

All these issues point to the urgent need to revisit the interface between the database and operating system, and explore the advantages of making it richer than it is today.

2.1 Databases on Multicore

Both the developments mentioned above and several new application areas have triggered a set of changes in the design of database systems [21]. Here we focus on those related to multicore architectures.

While it may be possible to modify a traditional database engine to scale if it is restricted to trivial (read-only, non-indexed, single-table) workloads [8], handling updates and more complex queries is considerably harder, even with a fully in-memory system, and many traditional designs do not perform well on many cores. Multiple different reasons have been identified [13, 15, 14, 17], but solving them requires in all cases drastic changes to the engine.

The challenge goes beyond simple scalability: modern hardware is complex and diverse, and databases increasingly need to be aware of cache architectures and system interconnects to be able to optimize query processing (e.g.[6]). A placement of data and query operators on the processors in a machine which works well on a particular hardware configuration and workload may perform poorly on another.

Column stores [6, 20] and shared scans [19, 11, 23], are widely viewed as two crucial techniques for addressing the multicore challenge in database design. We use both techniques in Cod.

2.2 Opening up the OS

Abstraction of resources has long been regarded as a core operating system function, and this has tended to go hand-in-hand with the OS determining resource allocation policy. Within an OS, of course, separation of mechanism and policy has a long and distinguished history going back at least to the Hydra system [16].

There has also been a parallel thread of OS research and design which has sought to get better performance by exposing more information to applications in a controlled way. For example, Appel and Li [1] proposed a better interface to virtual memory which still located much of the paging policy in the kernel, but nevertheless allowed applications (specifically, garbage-collected runtimes) to do a much better job of managing their own memory.

Architecturally, one way to do this is to remove abstractions from the kernel, and instead implement as much OS functionality (and consequently, policy) as possible in user-space libraries linked into the application – an approach used in Exokernel [9] and Nemesis [12]. This opens up the space for application-specific policies, but by itself does not solve the problem of how each application can map its requirements onto the available resources.

Perhaps the closest OS facility to the one we employ in Cod is InfoKernel [2], which advocated exposing considerable information about OS state to applications in a controlled way.

We conjecture that one reason these approaches so far have had limited impact is that they have been developed independently of any particular application's requirements. Their weakness has generally been the need for applications to specify their own resource requirements, something notoriously hard to do.

We build on these ideas but, in contrast, we approach the problem from a co-design perspective: given a database *and* an OS designed to fully exploit multicore hardware, what is the best interface, and the best distribution of state information, between them.

3. BACKGROUND

We agree with many who argue that the move to multicore architectures requires rethinking the design of both operating systems [5] and database systems [22].

However, Cod itself is not built from scratch. We leverage two recent prototypes which have individually explored the design space for multicore databases and for operating systems. As well as already targeting future hardware, as a research artifact each is also more amenable to the kinds of changes we would like to make in considering the database-plus-OS as a whole. Experience has shown that more mature systems like Oracle, PostgreSQL, Windows, or Linux would alone exhibit higher performance on current hardware, but would be much harder to modify architecturally or to use to identify interaction problems between the DB and the OS – our goal in this paper.

We provide a brief background on these two building blocks for our work here: a column-store implementation of the Crescando in-memory query processor, and the Barrelfish multikernel.

3.1 The CSCS engine

The database we consider is a prototype of an in-memory, column oriented, shared scan engine. The shared scan strategy is inspired by that of Crescando [23], while the column-store memory layout is similar to that used in MonetDB [6]. The main memory approach mirrors the strategy used in SAP's T-Rex accelerator. This prototype is a good match for multicore machines as it eliminates any need for synchronization and its performance can be controlled with great accuracy solely from the amount of data allocated to each core.

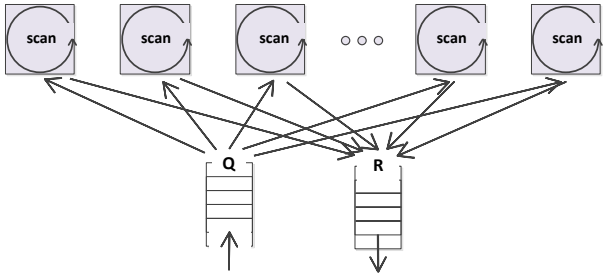

Figure 1: CSCS execution model

CSCS, the database prototype we use in Cod, is designed to scale well with core count and relies on a large main memory. The data set is horizontally partitioned, and for each partition a scanning thread is dedicated to process the requests on a separate core. Each scanning thread implements a columnstore version of Crescando's ClockScan algorithm [23], and is purely CPU-bound on current hardware. It is a shared-nothing parallel database on a single machine.

As shown in Figure 1, requests (both queries and updates) are buffered in the system's input queue and then propagated to the scanning threads in batches. The scanning threads perform a three-way join over columns, tuples and request predicates. This join is efficiently implemented by dynamically indexing the requests based on their predicates, and performance is further enhanced by processing queries in order of decreasing selectivity.

We keep intermediate state in position lists, which denote the matching tuples of the currently scanned data-partition. To ensure that the size of the position list does not grow beyond the size of the L1 cache, the CSCS datastore splits the partitions into datastore chunks and processes them by scanning one at a time – a technique inspired by vectorization [6, 20]. When the scan is done, the results from all scanning threads are aggregated into the output queue. Consequently, the response time for a batch of queries is determined by the scan thread with the longest processing time.

Furthermore the CSCS engine has well-defined and predictable performance characteristics: it is primarily CPU bound (not memory bound), mandates hard CPU affinity, and each thread operates solely on one particular chunk of the data in main memory. Finally, the prototype structure of CSCS simplifies the process of extending it and modifying it to leverage communication with the underlying operating system regarding the system architecture.

3.2 The Barrelfish OS

We implement our shared-scan column store over the Barrelfish open-source research operating system [4, 3]. Barrelfish is a "multikernel", internally structured as a distributed system of cores communicating via messages. The OS component of each core consists of a kernel-mode "CPU driver", which shares no state nor communicates with any other core, together with a varying set of user-mode processes which implement most of the OS functionality.

Architecturally, a multikernel is a good match for the CSCS data store: both are, at heart, shared-nothing architectures which seek to decouple activities on separate cores from each other as much as possible. As well as scaling well on hardware with many cores, this separation is useful methodologically: it reduces unpredictable cross-core interference effects and allows us to quantify the benefit of design changes more precisely.

However, two further aspects of Barrelfish also serve to make it attractive from the perspective of experimenting with OS / database co-design.

Firstly, allocation of physical resources to applications in Barrelfish is more explicit than in mainstream OS designs – a feature it shares with older, uniprocessor research OSes like Exokernel [9] and Nemesis [12].

From a design perspective, therefore, a multikernel like Barrelfish makes it easier to consider the database and OS as a single design task. Explicit placement of threads on cores, explicit allocation of cores to tasks (via upcalls), and user-level memory management (including self-paging) mean that Cod can manage such physical resources precisely and efficiently.

From a methodological perspective, it is also much easier to quantify effects and identify the causes of performance anomalies than in a system with stronger abstraction barriers between OS and applications.

Secondly, Barrelfish incorporates a service called the System Knowledge Base (SKB) [18]. The SKB is populated with a range of information about the hardware and OS, and can be queried by both the OS itself and applications. It also supports a rich data model based on logic programming and Prolog, constraint satisfaction, and optimization. While this might be considered overkill for a production OS, it affords us great flexibility in communicating information between the OS and database, and so serves as a valuable prototyping facility.

In particular, in this paper we exploit information in the SKB concerning the memory hierarchy (cache sharing, NUMA nodes, etc.), and the current application mix in the system (which processes are running, and which cores they are currently running on) to enable closer cooperation between the database and OS. We also exploit the SKB's facilities for constrained optimization.

4. SYSTEM DESCRIPTION

Cod is an implementation of a co-designed DB and OS. Figure 2 shows the architecture which is based on the Barrelfish OS [4] and the CSCS engine. We are going to explain the architecture in more detail in this section.

4.1 Where to locate knowledge

We distinguish two types of knowledge: machine architecture knowledge and application requirements and properties knowledge. In Cod we decided to push the former type of knowledge completely to the OS level. On one hand, the SKB on Barrelfish already has a lot of machine architecture knowledge available. This information consists of loaded configuration files, runtime data and online querying of installed hardware. The combination of this information is exposed in a standardized data layout independent of the concrete underlying hardware details providing an easy-to-use abstract information to its applications. On the other hand, the database should not need to know details about the hardware in order to take decisions, but rather rely on some abstract information acquired upon request.

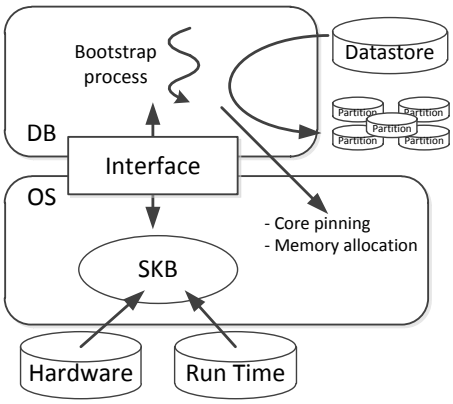

Figure 2: Cod architecture

4.2 Information to be exchanged

The knowledge about the worker thread and dataset properties is within the CSCS engine. An important thread property is that it is compute bound (rather than memory or I/O bound), which cannot be known in advance by an OS. Dataset properties include the dataset size and whether the data is partitioned or is yet to be partitioned in a suitable way. If data is pre-partitioned, the CSCS engine knows the number of partitions as well as the size of each partition. Furthermore the CSCS engine also knows which scanning threads operate on which partition.

As discussed earlier, the knowledge about the available number of cores and existing NUMA regions is at the OS level, i.e. in the SKB. Core information also includes cache sizes and whether they are shared.

It is therefore important to exchange information between the DB and the OS. The DB needs to tell the OS that it wants to run compute intensive threads as well as the total number and size of each partition it wants to use. The SKB service of the OS then uses these dataset and thread properties to produce a list of possible optimal mappings to the available resources and delegates the suggestions up to the DB. Based on the more abstract information it gets back from the OS, the DB can optimally place the scanning threads and assign the appropriate memory affinities.

4.3 Interfaces

We have extended the existing interface of the SKB by introducing several functions that ease the communication and information flow between the database and the OS. In this section we present the main three functions, their interfaces and briefly discuss the motivation for having each one.

(1) `get_cache_size()` takes as a parameter the cache level. The SKB, then, looks up the details for it within its knowledge base and returns the requested size. Knowing the size of the L1 cache for example helps in determining the appropriate size of the scanning database chunks so that we ensure that the position lists fit in it.

(2) `suggest_number_of_partitions()` takes as parameters the total size of the dataset and whether the workload is compute or memory bound. These values can be used by the SKB constraint solver, along with its knowledge of the system architecture and the runtime state of the OS, to determine the optimal number of partitions, to which the database should be split, so that we maximize parallelism.

(3) `suggest_resource_allocation()` provides the SKB with application-specific information and requirements such as the number of partitions, the size of each partition, whether the workload is compute or memory bound, and a flag for cache sharing preference. In some cases cache sharing is undesirable, especially when threads have conflicting interests and pollute their common cache. Furthermore, since the scanning thread algorithm of the CSCS is compute-bound, the SKB should not consider cores which are heavily used by other applications. For instance, the SKB knows that all of the OS's house-keeping tasks are scheduled to run on core 0 and will therefore know that this core is not a suitable candidate for other cpu-intensive applications. The SKB constraint solver takes into account the values of the function input parameters, the runtime state of the OS and its own knowledge of the underlying hardware architecture to produce a list of possible mappings that meet the DB needs. It eventually returns a list of available cores, and the corresponding NUMA domains where the dataset partitions should be placed.

In the following section we briefly describe how these are used by the database.

4.4 The column store

Having already described the interfaces and the information that needs to be exchanged between the CSCS engine and the SKB service of Barrelfish, we now present how it all fits together.

We built Cod by porting the CSCS engine to Barrelfish and extending it with a bootstrap process which initially configures the system.

Cod begins its execution by getting information about the size of the datastore it needs to deploy and learns if it is already pre-partitioned or not. In this description we consider that the datastore was not pre-partitioned beforehand so that we explain the entire communication flow between the CSCS engine and the SKB.

First the bootstrap process initializes the communication channel with the SKB. It then queries the SKB to learn about the optimal number of partitions given the size of the datastore (suggest_number_of_partitions()) that will maximize the parallelism on the chosen machine, and then based on the response of the SKB it splits the datastore in that many horizontal partitions.

Once we have the partitions, the bootstrap process needs to spawn the scanning threads, pin them on a CPU core and assign a memory region for each partition. In order to determine an efficient resource allocation, the bootstrap process queries the SKB using the suggest_resource_allocation() function. The SKB responds with a suggestion for resource allocation and a deployment layout. The bootstrap process uses this information to spawn the scanning threads to the specified cores, and to set the appropriate memory affinities.

5. EXPERIMENTS

In this section, we present two experimental results illustrating how Cod can exploit the co-design of a database and an operating system to obtain better performance resource utilization and performance without prior knowledge of the given hardware platform and software environment.

We obtained the dataset and workload used by Unterbrunner et. al [23], and used it for the experiments presented in this section. It is generated from the traces of the Amadeus on-line booking system.

In this paper we focus on a single hardware platform. Namely, for both experiments we used the AMD Shanghai machine. It is a 4x4 (4 socket, 4 cores per socket) machine with 4 2.5GHz AMD Opteron 8380 processors having 6MB shared L3 cache. The machine has 16GB RAM, with NUMA nodes of size 4GB.

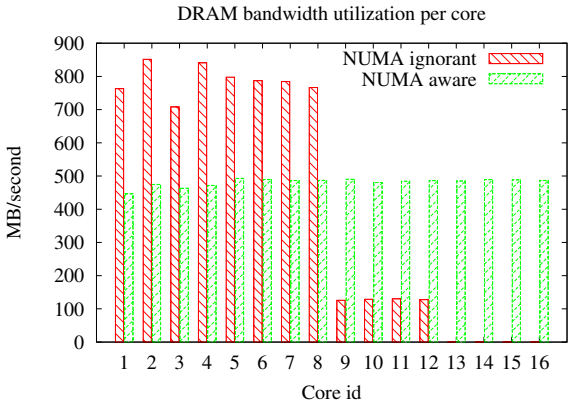

Figure 3: DRAM bandwidth utilization

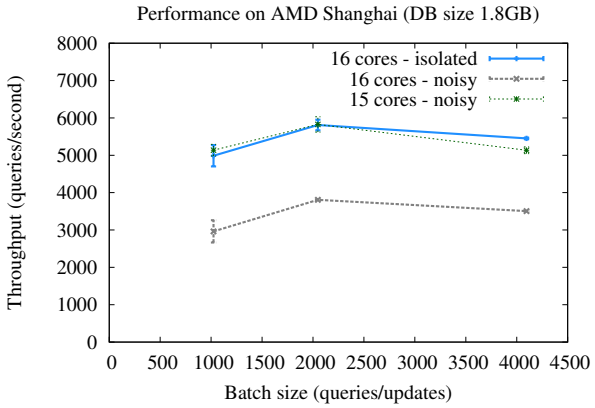

Figure 4: DB alone vs. sharing CPU with another application

5.1 Smart memory allocation

This first experiment shows the impact of informed memory allocation on database performance. We compare the performance of Cod performing naïve (OS default) memory allocation for partitions with an approach that queries the SKB for optimal placement. In the naïve approach, Cod allocates memory in a sequential manner without considering the NUMA layout or the location of the cores on which its threads are running. In the informed case, Cod queries the SKB service for information on both where to place the scan threads, and where to allocate main memory for the data partitions. We measure the DRAM bandwidth utilization using the hardware performance counters available on the cores.

For this experiment we used all 16 cores of the AMD Shanghai machine. The datastore had a total size of 7GB, resulting in 16 partitions of 438MB each. The workload batch size (#Queries, #Updates) was set to (2048, 256).

The DRAM bandwidth utilization in these experiments is shown in Figure 3. It illustrates the utilization measured per core.

On one side we can see that the naïve approach (NUMA ignorant) of allocating memory results in a very uneven distribution of load across the NUMA nodes. In fact, considerable bandwidth pressure is imposed on the first two NUMA nodes, where most of the datastore memory ends up being stored. On the other side, in the informed approach (NUMA aware) we can see that the utilization is uniformly distributed among the four memory controllers.

These results are, of course, to be expected: without specifying memory affinities, the datastore initialization processes populate the data structures in main memory in a sequential manner and thus most of the datastore data is allocated in the first two NUMA nodes. In contrast, when using NUMA-aware memory allocation, the correct dataset partitions are allocated locally on each NUMA node hosting a scanning process.

The results confirm that smart memory allocation results in a better and more uniform utilization of memory and interconnect resources. Furthermore, it also provides the flexibility for easy deployment on a range of other different machines and architectures. Such NUMA-aware allocation is rare in modern databases, but we note that even with facilities like Linux's `libnuma` facility, the database requires considerable intelligence (and therefore complexity) to take advantage of such knowledge. In contrast, Cod not only achieves smart allocation with minimal code complexity (the inference engine in the SKB performs most of the heavy lifting in this case), the design of Cod (shared-nothing, continuous scans) makes the effect of NUMA-aware allocation more significant.

5.2 Deployment in a noisy system

With this experiment we illustrate the effect that arises when another system task runs on the same machine as the database, thereby creating a conflict for resources.

On a 16-core machine (AMD Shanghai), a naïve deployment would partition the database into sixteen contexts (to maximize parallelism) and spawn a scan thread on each core, without consulting with the OS as to whether there are other tasks in the system already scheduled on some of the CPUs.

In this experiment we show that the naïve deployment can significantly reduce performance, especially in the presence of another competing system task. Furthermore, we show the benefits of an alternative approach where Cod queries the SKB in order to obtain an allocation of cores and memory which avoids such conflicts. In our particular example, the SKB will suggest a configuration using fifteen partitions, based on the number of available cores, and specify the exact cores and memory regions to be used.

For the purpose of this experiment we used a datastore of size 1.8GB and we varied the workload batch size of (#Queries, #Updates) as {(1024, 128), (2048, 256), (4096, 512)}

The results confirm the importance of smart resource allocation on the performance of Cod when running in a noisy system. In Figure 4 we see the throughput of the column-store running on sixteen cores both when isolated and when sharing the machine with another task. It shows the performance of the system as the load (workload batch size), imposed on the system, increases. We can see that running the CSCS engine on all 16 cores in a noisy system can significantly impact performance, degrading it by almost 30 percent. The figure also shows the performance of the column-store in the same noisy system when deployed on fifteen cores – a load-aware layout suggested by the SKB.

The difference in throughput when running the CSCS engine on fifteen (noisy) and sixteen (isolated) is almost negligible with smaller workload batch sizes, as the system is still underloaded, and can only be seen when processing around 4000 requests in a batch.

The significant drop in performance when running the CSCS engine in a noisy system on all sixteen cores comes as a consequence of the internal design of the CSCS engine. As described earlier, the overall performance of the system depends on the scanning time of the slowest scanning thread that processes the batch of requests. As expected, if we have another CPU-intensive task operating on one of the cores, it competes for the CPU with one of our scanning threads and thus slows down the execution of the whole CSCS en-

gine. Therefore, using a deployment that avoids such conflicting allocations, results in a system that significantly outperforms the originally intuitive and naïve approach.

6. CONCLUSION

Multicore systems pose serious challenges to both operating systems and databases, and many believe that a complete redesign is necessary to exploit the intrinsic parallelism. We argue that this is an opportunity to revisit the problem of DB/OS co-design aiming for better utilization of the available resources. With this paper we start exploring how the first and crucial aspect of this co-design can be accomplished with Cod.

Cod combines a DB and an OS, both designed to work well on top of multiple cores, into an entity that is more tightly integrated. Through a simple but powerful interface we demonstrate how a database can make optimal use of the system resources (mainly CPU and memory) on different multicore architectures by exploiting the architecture knowledge available at the OS-level (even in the presence of other resource-competing tasks in the system), and combining it with the database's internal knowledge of its own performance characteristics and workload.

We feel Cod is a good foundation for future work exploring how to evolve the OS/DB interface. We are extending the SKB interface to support other interactions relevant to both the database and the operating system such as: scheduling, synchronization, and network and I/O access. On the DB-side, we are exploring how much the extended interface can be exploited to enhance database "elasticity". A further intriguing question is whether the operating system can be usefully involved in query optimization decisions.

Finally, databases are a good choice for application/OS co-design: they are critical components of systems infrastructure, and have sophisticated internal models of both the system for which they are configured and the demands of their current workload. However, other applications share some of these characteristics, such as some advanced programming language runtimes. This is a broader area we are beginning to explore.

7. REFERENCES

[1] A. Appel and K. Li. Virtual memory primitives for user programs. ASPLOS-IV, pages 96–107, 1991.

[2] A. Arpaci-Dusseau, R. Arpaci-Dusseau, N. Burnett, T. Denehy, T. Engle, H. Gunawi, J. Nugent, and F. Popovici. Transforming policies into mechanisms with infokernel. SOSP '03, pages 90–105, 2003.

[3] Barrelfish Project. The Barrelfish research operating system. `http://www.barrelfish.org/`, October 2011.

[4] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP '09, pages 29–44, 2009.

[5] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[6] P. Boncz, M. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51:77–85, December 2008.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, 2008.

[8] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. OSDI'10, pages 1–8, 2010.

[9] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. SOSP '95, pages 251–266, 1995.

[10] J. Gray. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, pages 393–481. 1977.

[11] W. Han, W. Kwak, J. Lee, G. Lohman, and V. Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1:188–200, August 2008.

[12] S. Hand. Self-paging in the nemesis operating system. OSDI '99, pages 73–86, 1999.

[13] F. Huber and J. C. Freytag. Query processing on multi-core architectures. In *Grundlagen von Datenbanken*, pages 27–31, 2009.

[14] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. A new look at the roles of spinning and blocking. In *DaMoN*, pages 21–26, 2009.

[15] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[16] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. SOSP '75, pages 132–140, 1975.

[17] T. Salomie, I. Subasu, J. Giceva, and G. Alonso. Database Engines on Multicores, Why Parallelize When You Can Distribute? EuroSys'11, 2011.

[18] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of MMCS'08*, 2008.

[19] T. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13:23–52, 1988.

[20] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. VLDB '05, pages 553–564, 2005.

[21] M. Stonebraker and U. Çetintemel. "one size fits all": An idea whose time has come and gone (abstract). In *ICDE*, pages 2–11, 2005.

[22] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). VLDB '07, pages 1150–1160, 2007.

[23] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. pages 706–717, 2009.