COD: Database / Operating System Co-Design

Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach
Gustavo Alonso, Timothy Roscoe
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
www.systems.ethz.ch

ABSTRACT

Trends in multicore processors and hardware virtualization pose severe structural challenges to system software in general, and databases in particular. On the one hand, machines are becoming increasingly heterogeneous in terms of, i.e., cache hierarchies, processor interconnect, instruction sets, etc., thereby making it increasingly difficult to develop optimal software for all possible platforms. On the other hand, virtualization forces databases to share resources with other applications without the databases having any knowledge about the run-time conditions.

As part of a long-term effort in our group to revisit the entire software stack of the data center, in this paper we explore how to enhance the interaction between databases and OS to allow the database better cope with varying hardware characteristics and runtime conditions. The goals is to integrate the database's extensive internal knowledge of its own resource requirements –including cost models– into the system-wide and run-time view of the hardware configuration and application mix available inside the OS.

1. INTRODUCTION

The last years have seen profound changes in the hardware available for running database systems.

On the hardware side, increasing core counts are at odds with the synchronization-heavy approach to concurrency used in both databases [29] and operating systems [9]. Furthermore, increasing heterogeneity both within and between systems poses additional structural problems: optimal use of resources requires detailed knowledge of the underlying hardware (memory affinities, cache hierarchies, interconnect distances, CPU/- core/die layouts, etc.), a problem aggravated by the increasing diversity of machines in the marketplace. Such a trend towards more complex, and more parallel computer architectures means that databases must work harder to efficiently exploit the available hardware: placement of data in memory, and assignment of tasks to cores, for example, now have a much more profound effect on performance. Databases perform their own memory allocation and thread scheduling, store data on raw disk partitions, and implement complex strategies to optimize and avoid I/O synchronization problems (e.g., in logging).

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2013.
6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

Extensive research has been invested over more than three decades to optimize such engine decisions. Unfortunately, and unless something changes, a great deal of additional complexity will need to be added to database engines to cope with the increasing heterogeneity within and across hardware architectures.

On the deployment side, in an age of virtualization and server consolidation, databases can no longer assume they have a complete physical machine to themselves. Databases are increasingly deployed on hardware alongside other applications: in virtual machines, multi-tenant hosting scenarios, cloud platforms, etc. As a result, the carefully constructed internal model of machine resources a DBMS uses to plan execution has become highly dependent on the runtime state of the whole machine, state which is unknown to the database and is currently available only to the operating system (OS). In our experiments we observe that even small OS-related tasks can impact the performance of an otherwise scalable database because the DBMS is unaware of these tasks. Hence, good performance in the presence of other applications requires the database to have an accurate picture of the runtime state of the whole machine. We may have reached the limits of complexity that database engines (already bloated and facing their own scaling problems because of multi-core) can absorb while trying to optimize for a given architecture and second-guessing the OS.

Database management systems and operating systems have a history of conflict. Although 35 years ago Jim Gray pointed out the significant gains that could be obtained by co-designing the DBMS and the OS [19], not much has changed since then. Modern databases find many of the abstractions provided by an OS of limited use. In parallel, the design of modern OSes pays little attention to database workloads. In part this is because databases, as the large enterprise application *par excellence*, usually run on dedicated machines and so the OS as a resource allocator and arbiter between applications has little role to play.

However, this situation is now changing. As we discuss in Section 2, the complexity and diversity of future hardware platforms has led to calls to re-architect both databases [18, 12, 24] and operating systems [9, 6]. In this paper we argue that it is necessary to rethink not only the design of operating systems and database management systems for future hardware, but also the interface between them. Only through a better interface and better interaction between the two will databases be able to exploit the sophisticated knowledge of the application with the mechanisms and ability of the OS to manage shared resources and to act as a homogeneous hardware driver across architectural differences.

Our contribution in this paper is to show the benefit of integrating recent ideas from both areas in a co-designed DBMS and OS, including, crucially, a richer interface between the two. Such integration makes sense in a wide range of contexts but specially in

data processing appliances (which provide the context for the work done in this paper [1]).

The resulting system, COD, combines a DB storage engine redesigned to operate well on multi-core machines with an OS level policy engine in charge of making suggestions and decisions regarding all deployment aspects of the DB. The interface between them allows several types of interactions: from adding information to the policy engine or registering for notifications regarding changes in the system-state to declaring DB specific cost models that the OS service then uses to answer queries about job placement and system-state. Based on the recommendations from the OS policy engine, the DB can optimize queries and restructure itself so as to be able to meet concrete SLAs of throughput and response time. The OS itself uses the same facility to spatio-temporally schedule multiple applications without interfering with the database, as it now knows what the DB needs to meet its SLAs.

2. BACKGROUND

COD is motivated by the implications for both databases and operating systems of trends in hardware and workloads, and combines several recent ideas from the OS and DBMS communities.

A current school of thought in research is devoted to redesigning system software for multicore architectures [6, 36, 26, 24], reducing inter-core memory sharing and synchronization by careful structuring of the whole system. COD follows this same line of thinking.

2.1 Databases on Multicore

Conventional databases aim for thread concurrency but not parallelism, and exploit little information about the underlying hardware. While a traditional DB can be made to scale when restricted to trivial (read-only, non-indexed, single-table) workloads [10], handling updates and more complex queries is much harder, even with a fully in-memory system. Traditional designs do not perform well at scale, due to algorithms optimized for single-CPU systems [22], scalability limits on synchronization primitives [24], architectural limitations [23], or load interaction among parallel queries [29]. In all cases, the fix requires major changes to the system.

The challenge goes beyond 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 [8]. A placement of data and operators on the cores in a machine which works well on one hardware configuration and workload, may perform poorly on another.

Column stores and *shared scans* are two recent techniques for addressing these challenges in database design, in particular for Online Analytical Processing (OLAP) and Operational Business Intelligence (BI) workloads. COD uses both techniques.

Shared scans are a simple example of a multi-query optimization technique [31]. The idea is to process a group of queries on the same table simultaneously by executing a full table scan once for all the queries in the group. Shared scans are used in systems like RedBrick [16], IBM Blink [20] and Crescando [35] and have been extensively studied [37, 34].

For instance, Crescando's "ClockScan" algorithm, the one implemented in COD, uses one scanning kernel thread pinned to each core that continuously scans a horizontal partition of the complete data set. Result tuples are aggregated from all scan threads. With modern hardware, a single scan thread can answer thousands of requests at a time and is CPU bound – the performance is limited by the time taken to execute the queries on the subset of tuples currently in the processor cache, rather than the DRAM bandwidth needed to move tuples in and out of the cache. Crescando provides excellent scalability due to the low synchronization requirements between scanning threads, and its high predictability: the performance of a shared-scan is stable and easy to model.

Column stores are attractive for OLAP and Operational BI because these workloads process large numbers of tuples to compute a given business metric. With a column store, if the query involves only a few columns of a table, only a fraction of the data has to be brought into the processor cache. As a result, column-oriented DBMS implementations, including main memory ones, are now common in industry and research, for instance MonetDB [8], C-Store [33], Vertica, and SAP's T-Rex accelerator.

The database part of COD builds on existing work [2] combining both techniques to define a deployment and work unit within the database that (1) can be allocated to a single core, (2) operates without any interference with the other work units, and (3) has fully predictable performance controllable with a few, well-defined parameters. Thanks to the robust[1] behavior of the database component, COD can make a more precise allocation of resources without the need for inefficient over-provisioning.

2.2 Opening up the OS

The OS problem that COD seeks to address can be cast as follows: given a database *and* OS, both designed to fully exploit multicore hardware, what is the best interface and best distribution of state information between them?

Abstraction of resources and the associated encapsulation of system state 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. As a result, the abstractions (virtual processors, uniform virtual memory, etc.) provided by conventional OSes have often turned out to be a poor match for the requirements of relational databases.

This is less true of the research OS work we survey below, which tends to be wary of over-abstraction, but we are aware of surprisingly little such work in the OS community that targets databases. This is despite the fact that the detailed resource calculations that databases perform would seem to make them an ideal test case for better OS designs and abstractions.

The separation of mechanism and policy in an OS has a long history going back at least to the Hydra system [25]. One thread of OS research has always sought to get better performance by exposing more information to applications in a controlled way. For example, Appel and Li [3] 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 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 [15] and Nemesis [21]. 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.

Alternatively, extensible OS kernels like SPIN [7] and VINO [32] allowed applications to inject policy code in the OS, where both the mechanism and state required to make decisions were located. However, even on uniprocessor systems, the benefits of such approaches are debatable [14].

InfoKernel [4] adopted a different approach to overcoming abstraction barriers and the "semantic gap" between OS state and application-level research management, by exposing considerable

[1]Robust in the sense of producing fully predictable execution times regardless of the query and load

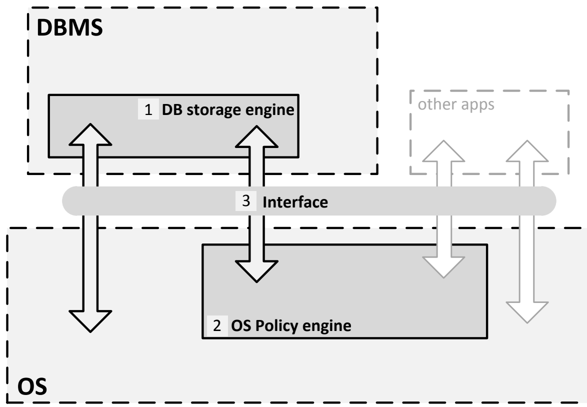

Figure 1: COD architecture. Shaded blocks denote the main components

information about the state of a conventional OS to applications.

Today, as with databases, there is considerable interest in removing scalability bottlenecks in OS designs (e.g. [10, 6, 36, 26, 17]), much of which is relevant to database/OS co-design. A key area to emerge is spatio-temporal scheduling: various ways of combining time-slicing on individual cores with longer-term assignment of tasks to processors [26, 27]; COD provides such placement as basic OS mechanism.

One example of this work, the Barrelfish OS, extends ideas from Exokernel and InfoKernel by combining a small and relatively policy-neutral kernel per core with a novel system facility, the "System Knowledge Base" (SKB). The SKB is a complete constraint logic programming (CLP) solver populated with rich hardware information and OS state, and is used, for example, to construct optimal communication patterns over the hardware [6] and configure hardware devices [30].

Although prototyped on Linux, COD builds upon the Barrelfish SKB, with the goal of deeper integration in the future [1].

3. SYSTEM OVERVIEW

The key feature of COD is the interface between the database and OS so that the database can make optimal use of the available system resources even in a dynamic, noisy environment where it shares the machine with other applications/tasks. Figure 1 illustrates the basic architecture. Such an interface cannot be easily constructed with existing products. Thus, here we resort to two experimental systems where we have access to the source code as building blocks for COD. However, we show that the ideas behind COD and what needs to be done at both sides of the interface to get the results can be easily generalized to other type of systems.

Since database engines are complex systems with many components, in this paper we focus on the storage engine, marked (1) in Figure 1. Our storage engine is a main memory, column oriented, shared scan engine designed for robustness and whose performance can be precisely controlled with a few parameters. Its design and characteristics are covered in Section 4.

The second building block is the OS Policy Engine, a service provided by the operating system, and marked (2) in Figure 1. This unifies the OS knowledge of the available hardware resources, such as cores and NUMA-domains, with information provided by applications running atop. This enables the OS to better orchestrate resource allocation both among running tasks and within a specific application. We discuss it further in Section 5.

Between the two is a rich query-based interface that not only allows for exchange of system-level information but also provides the means to create application-specific stored procedures, able to exploit both the constraint solver and optimizer as well as the overall knowledge available on the OS side. Section 6 describes the interface in more detail.

4. THE DATABASE SIDE OF COD

To be able to interact in a reasonable manner with the OS, the database side has to have a very precise idea of what it needs. Briefly, the storage manager needs (1) a working unit (e.g. a thread) that allows flexible and elastic use of resources as well as serving as deployment unit, and (2) a cost function that predicts the impact of resource allocation decisions on the overall performance (response time and throughput). Below we present the storage manager of the COD database, the *Clock-Scan Column Store* (**CSCS**) [2], which we use to obtain these features.

4.1 CSCS Architecture

Traditional database storage managers use memory pages and blocks as the exchange unit with the rest of the system. Their function is to manage the buffer pool from which every transaction thread obtains memory pages, while enforcing the mechanisms that ensure transactional durability (flushing dirty pages to disk when needed) and isolation (e.g., when enforcing snapshot isolation), in addition to prefetching and replacing pages to minimize page misses. The performance of the database depends heavily on the storage manager, since it controls the memory allocation and replacement policy. Given their architecture, storage managers must arbitrate among the competing needs of concurrent transactions. This makes them a critical component, specially in multi-core machines where concurrent transactions become truly parallel threads competing for resources. In multicores not only memory size and paging, but also core affinity and core allocation play fundamental roles in determining performance.

Unsurprisingly, these challenges are currently a hot topic in both database research and industry. Inspired by work on tailored database engines for the airline industry [35], CSCS has been built as a storage manager with a SQL-based interface instead of purely memory blocks and pages.

Derived from the flexibility and predictability requirements above, along with workload requirements, CSCS' architecture has the following characteristics:

1. **Batching of requests**: instead of assuming each transaction runs on its own thread, CSCS batches transactions and processes them as a group – an idea increasingly applied in systems like IBM Blink [28], SharedDB [18], DataPath [5], and C-Join [11, 12].

2. **Main memory storage**: As with most commercial database engines today, CSCS is a main memory storage manager. Like SharedDB and DataPath, CSCS maps individual operators to cores and well-defined memory regions, with no interaction between them beyond the necessary data flow.

3. **Shared scanning**: Like IBM Blink or Crescando [35], CSCS avoids static indexes on the data. Such systems do not pay the penalty of many indexes during insert/update transactions. By only scanning the data, they can offer an upper bound on the latency of each transaction.

4. **Column storage**: In performing scan operations, column storage offers better data locality than traditional row storage.

While none of the above techniques is new, the novelty of CSCS is that it encompasses all of them in achieving its goals: a fast, flexible and predictable storage engine.

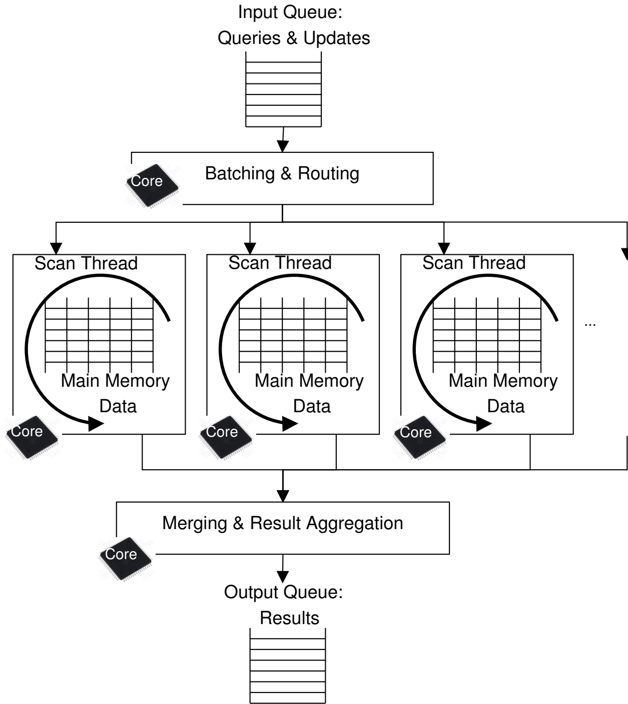

Figure 2: CSCS architecture

Figure 2 shows the CSCS architecture. Requests enter the *input queue*, where they are batched and indexed based on their predicates, while awaiting processing. *Scan threads* perform full data scans, each one over its own partition of the data. Data is partitioned first into columns (CSCS is a column store), and then horizontally in partitions, as needed for scalability (see below). After each complete scan, the scan threads read a new batch of requests and start a new data scan phase. As the scan progresses, results are passed to the *merging and aggregation* thread, which pushes them into the *output queue*. More details on the system can be found in the corresponding tech report [2].

4.2 Deployment/Performance unit

For our purposes in COD, the key feature of CSCS is the working and deployment unit used - the scan thread. Scan threads have hard-CPU affinity, and the memory for the data partition processed by each scan thread is well-defined and determined in advance. The column store approach ensures high data locality during the scan, minimizing L1 data cache misses. Therefore, under all loads, CSCS remains CPU bound, but also cache- and NUMA-sensitive. Furthermore, all requests within a batch are processed in the same amount of time, with the response time bound by the time it takes to perform a single scan matching each tuple against the index of requests and merge the results.

4.3 Properties of CSCS

The number of CSCS scan threads is determined by the amount of data in the system, expected peak throughput, and target response time bounds for processing a request. From these parameters, the number of scan threads needed can be calculated in advance, and therefore the performance of CSCS determined by design, making it both *scalable* and *predictable*.

The **scalability** of CSCS derives from the ability to linearly reduce response time by increasing the number of scan threads. For a fixed amount of data, either the same number of requests can be processed faster or more can be processed with the same response time by reducing the amount of data assigned to each scan thread and adding more scan threads. Since CSCS scans are *synchronization free*, the number of scan threads in the system has negligible effect on the overall behavior.

The **predictability** of CSCS lies in its ability to guarantee an upper bound on the response time of any request that it receives. While traditional systems are optimized for answering each request as fast as possible by using data-indexes, this does not scale for diverse workloads with many update requests: increasing the number of indexes in order to optimize any possible request degrades the update performance. Alternatively, CSCS handles such workloads in a predictable way by optimizing the most expensive operation: performing a full data scan. A full data scan requires the entire data to be read in order to fulfill each request. This latency is high, but predictable, since we always know how long it takes to scan the whole dataset. Extending this naïve processing model by batching requests on the same full data scan yields an increased throughput that more than compensates for the increased latency of each individual query [35, 5].

4.4 Embedding into COD

CSCS provides a working unit that is easily deployable and scalable. It remains to provide a cost function that can be used for allocating resources such that the database can meet its SLAs. From now on we will assume, without loss of generality, that the system's SLA is an upper bound on the response time of all requests.

In this paper, we focus on a workload comprising both update and read-only queries. It is inspired from an operational business intelligence workload from the travel industry that contains a high number of both queries and updates [35]. We use this workload for our experiments. All requests (i.e., queries or updates) are point-requests matching few records. The dataset is formed by a single table with 48 attributes.

Table 1: Throughput (queries/sec): Operational BI Workload, Vary #Cores, Vary Dataset Size (1core/1GB)

	Number of cores/Data size				
	1	8	16	32	40
	Throughput (queries/sec)				
CSCS	870	860	850	843	843

Table 1 shows the results of the benchmark when run on a 48-core AMD MagnyCours with 128GB RAM. We varied the number of cores used from 1 to 40 and varied the database size from 1 GB to 40 GB, so that each new scan thread adds 1GB of data in a new partition. The workload uses batches of 2048 queries and 256 updates. As the results show, CSCS scales with the data size, maintaining a steady throughput.

From this data we derive via polynomial fitting a cost function describing the response time of CSCS as a function of the number of scan threads (cores), tuples per thread (memory), and requests per thread (batch size). The model we use is:

$$RT[ms] = c \cdot \frac{\#tuples}{\#cores} \cdot \{a \cdot \#requests + b\}$$

For the AMD MagnyCours machine, these constants are:
$a = 0.85, b = 106.37$ and $c = 1/3750000$.

Constants a and b are machine-dependent, while c is used to normalize the total number of tuples used in the experiment.

Using this function, CSCS can delegate the deployment of scan threads across cores and the corresponding allocation of memory to the OS. Given the cost function, the OS has a precise idea of what the application needs and will try to find the best possible match to the characteristics of the underlying hardware, current system state, and resource utilization.

Last but not least, CSCS has been designed to be both proactive and reactive: in addition to stating its requirements, it can also receive notifications from the OS indicating changes to available resources. Before starting a new scan phase, CSCS checks whether it needs to reconfigure its deployment. If so, it reorganizes the data partitions, starts or stops new scan threads as needed, or reallocates scan threads and their associated data across cores.

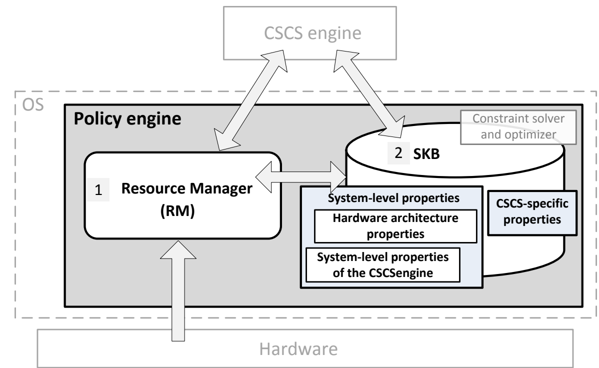

Figure 3: New OS functionality

5. THE OS SIDE OF COD

COD extends a traditional OS with new functionality aimed at widening the interface between it and applications, in particular CSCS. One critical challenge is the *distribution of knowledge* in the system: the OS holds information about system resource state and hardware performance trade-offs, whereas CSCS has detailed knowledge about its workload, via application-specific analysis and the inherent predictability of its shared-scan architecture.

For the OS to make optimal resource allocation decisions, *and* for CSCS to make optimal use of the resources available, both need access to *all* of this knowledge. However, centralizing it in the OS requires the OS to take "on trust" information from CSCS and perform CSCS's optimization for it. On the other hand, centralizing it in CSCS prevents the OS from making global resource decisions.

We resolve this tension as follows: the OS maintains detailed information about hardware (cores, memory hierarchy, performance trade-offs, etc.) and its own state (resource allocations, load, etc.) in a rich representational framework that enables it to reason about this information online to make medium-term (i.e. on the order of a few seconds) policy decisions, such as spatial placement of OS and application tasks on cores.

However, the OS also exposes this functionality to CSCS. It can perform complex application-specific calculations on this state on behalf of CSCS and other applications, allowing them to optimize usage of their own resources. Coupled with a facility for explicitly notifying the database when the system resource allocation changes, it provides CSCS with the benefits of being able to reason about the complete system state, without the cost of replicating and maintaining that state in the application. Finally, CSCS can submit hints to the OS about utility, which are stored alongside system state. In COD, we refer to this part of the system as the Policy Engine and view it as part of the operating system.

5.1 Architecture

The new OS functionality is structured as shown in Figure 3, and consists of two additional facilities: the Resource Manager (RM) and System Knowledge Base (SKB).

We borrow the concept of the SKB from the Barrelfish OS. It stores free-form predicates in a Constraint Logic Programming (CLP) engine, and permits reasoning over this information by means of logical queries extended with facilities for constraint solving and optimization. The information in the SKB falls into two categories.

First, the SKB is populated with information from the OS about the hardware obtained at startup from resource discovery and online micro-benchmarks (such as the hardware topology, memory hierarchy, and core to memory affinities), and system state (such as the set of running tasks, and their spatial assignments to cores). It can also store other named constraints and inference rules. We refer to these collectively as *system-level facts*, and CSCS can query the SKB for information about current resource allocations, but more importantly can submit complex queries to allow it to optimize execution based on the resources it has.

Second, CSCS also submits "hints" for resource allocation to the SKB in the form of additional constraints, which the OS may take into account when allocating resources: this is not intended to override OS policies (though the OS might choose to), but rather provides additional, application-specific information to the OS to help it select the best allocation from a set of policy-compliant alternatives. We refer to these as *application-level facts*.

Application-level facts are part of application's domain knowledge which might not be understood by the OS, but rather can be used by application-specific stored procedures (explained later) to compute desired system-level properties. Section 7 shows several examples where these facts are used, and explains how this information is being utilized in different scenarios.

The SKB is purely "reactive": it is essentially a repository and calculation engine for knowledge about COD as a whole. The RM, on the other hand, implements the results of resource allocation algorithms executed in the SKB. On every change of the environment (new task arrives, task terminates, task's properties change), the resource manager triggers a re-computation of the global allocation. After the re-computation, the RM notifies the CSCS (and other affected applications) about the new resource allocation decisions.

Our new OS facilities are not intended to be on the critical path either in the OS or CSCS – in other words, they do not delay regular operations in either. Rather, they provide a way to calculate medium-term policies out-of-band, such as thread placement or data partitioning based on global system knowledge which can then be implemented inside CSCS or the rest of the OS.

5.2 Implementation

The OS portion of COD has been built on version 2.6.32 of the Linux kernel for 64-bit x86 machines, though there is little that would prevent a port to an OS with more explicit allocation of resources such as Barrelfish. The RM implements spatial scheduling by means of thread pinning, populates the SKB, and triggers periodic recalculation of resource allocations. It also mediates between the SKB and CSCS, and informs CSCS of changes in resource allocation by means of upcalls. The SKB is built using the ECL^{i}PS^e CLP engine, running as a system daemon. ECL^{i}PS^e is expressive, convenient and easy to run, but complex queries can be slow. Using a modern Satisfiability Modulo Theories (SMT) solver like Z3[13] is a topic for future work.

Table 2: Message types and instances

Message Type	Example
Core functions	`rsmgr_client_connect()`
Add facts	`add_fact(var_name(value))`
Add CSCS-specific function	`add_query("f_name(f_vars):-f_content")`
(De-)Register for notification	`rsmgr_register_fn(event, filter, handler)`
Query system-specific info	`execute_query("get_nr_cores()")`
Query CSCS-specific function	`execute_query("f_name(f_param,f_result)")`

5.3 Discussion

Even though the computation of a global allocation plan is periodically done off-fast path, it is important to do that in a reasonable time. The current implementation does not restrict the ECL^{i}PS^e solver in any ways when finding valid allocations. This complete freedom implements the most general allocation policy and allows the solver to consider every possible solution. This comes at the risk of high execution times. It is infeasible to predict execution times, as this depends heavily on the requirements imposed on the system by the CSCS and possibly other applications.

For future work, we envision to restrict the solver's search space in a way that it still finds allocation solutions close to the fully flexible implementation. As an example, if CSCS needs four cores, the current implementation might chose them from a full permutation of all available cores. This is not necessary, as CSCS is not affected by the order it gets the four cores.

6. INTERFACE

In this section we present the third main component of COD that joins the CSCS and the OS Policy engine: the interface between them. Examples of how it is used are provided in Section 7. Here we provide an overview of the current scope covered by the interface, followed by a brief description of the different classes of supported functions and their intended use, as well as an outline of the planned methods for evaluation.

6.1 Scope

As described earlier, currently the interface between the database and the OS is only covering the communication between the DB storage engine (CSCS) and the OS policy engine. We are thus providing support for actions such as: retrieving information about the underlying architecture and resources, pushing down application-specific properties and cost functions in a way that the OS policy engine can reason about them, as well as some basic functionality that allows for information flow also during runtime.

This list is neither exhaustive nor exclusive, but rather is intended to illustrate the possibilities currently offered by the interface and the type of interactions that can be implemented in a co-design architecture. Later, it can be further enhanced with support for actions involving database query optimization and scheduling, as well as more advanced features needed for control and coordination with other modules of the operating system. These extensions are left for future work.

6.2 Semantics

For better overview of the current possibilities, we have grouped the supported messages into several categories and summarized them in Table 2. COD's interface currently supports the following types of messages:

Core functions provide support for: (1) Initializing communication between the CSCS and the policy engine, (2) registering CSCS as a running task, informing the policy engine that a new task has arrived, and setting up state so the RM can forward notifications to it, and (3) requesting a resource allocation suggestion from the RM that will invoke the execution of the global allocation code in the SKB, and eventually forward the decision to the affected applications.

Add facts calls enable CSCS to load information, as facts, for its own properties into the SKB. As described earlier, the policy engine distinguishes between system-level properties and application-specific facts. The interface allows for both of these to be modified and/or removed at any time during the execution of the application.

Add application-specific function calls enable CSCS to add stored procedures to the SKB that are specific to its own needs, such as the deployment cost stored procedure (presented in Listing 3). These procedures can use all CSCS-related facts and system-level properties belonging to the application.

Subscribe for events calls allow CSCS to be informed about changes occurring in the system, and filter unrelated events. RM by default will notify all applications via upcalls when the global system optimizer of the SKB changes the resources allocated to a particular task. This enables CSCS and other applications to adapt their execution plans and internal resource management accordingly and to react to the changes in the system that affect the resources they operate with.

Query system-specific information calls enable CSCS to issue queries that retrieve system-specific information.

Query application-specific functions calls allow CSCS to also query its own stored procedures, previously added to the SKB.

As we mentioned before, this list is neither exhaustive nor exclusive, but rather intended to illustrate the possibilities offered by the currently supported interface and is subject to be altered and extended in the future.

6.3 Implementation

The implementation of the interface is heavily dependent on the OS policy engine, its support and the syntax it understands. Since the OS policy engine is implemented as a user-space library, written in Prolog and uses the ECL^{i}PS^e CLP language, most of the function calls of the interface resemble the Prolog format. Exceptions are the core-functions that enable the CSCS engine to bind to the RM and the SKB and register itself as an application entering the system. All other functions contain a string of Prolog command as argument, containing both the input parameters for the function and pointers for the output variables. Parsing the response obtained from the policy engine on the client side is implemented in a similar fashion. More detailed explanation will be provided in Section 7, accompanied with an example.

6.4 Evaluation

In order to evaluate the proposed and implemented interface we intend to investigate it from two different aspects:

1. Applicability We are interested in evaluating the applicability of the proposed functionality, i.e. how it can be used, in which concrete scenarios and use-cases, the benefits of utilizing it, as well as the potential for expanding it. For that purpose, in the next section we present several scenarios, and for each we discuss in more details how the interface is used. Furthermore, we will present possible options for extending the usability by giving other examples where applicable.

2. Overhead A different aspect when evaluating the interface is the overhead that it introduces to the system. Thus, we plan to investigate the cost for making each call that affects the latency; the size of each message and the frequency of their occurrence, both of which affect the extra load imposed on the system bus. This evaluation will be presented after each use-case, as part of the discussion for the overall results obtained from that experiment.

7. EXPERIMENTS

This section presents in more details the interactions between the CSCS engine and the OS policy engine through use-cases, presenting both the advantages of this approach supported by experiments and discussion of the overhead imposed by the communication. Furthermore we conclude each scenario with ideas for extensions and possible future work directions.

More concretely, in this section we show that COD can deploy efficiently on a variety of different machine configurations without prior knowledge of hardware, and react to additional loads on the system to preserve performance.

7.1 Experimental Setup

The dataset and workload used in these experiments is the one used by Unterbrunner et al. [35]. It is generated from the traces of the Amadeus on-line flight booking system. It is characterized by a large amount of concurrent point-queries, frequent peak loads, many updates and strong latency requirements.

We used four different hardware platforms for diversity:

1. AMD Shanghai: Supermicro H8QM3-2 board with 4 quad-core 2.5GHz AMD Opteron 8380 processors, and 16GB RAM, arranged as 4GB per NUMA node.

2. AMD Barcelona: TyanThunder S4985 board with M4985 daughtercard and 8 quad-core 2GHz AMD Opteron 8350 processors and 16GB RAM across 8 NUMA nodes.

3. Intel Nehalem-EX: Supermicro X8Q86 board with 4 8-core 1.87GHz Intel Xeon L7555 processors and 128GB RAM with a NUMA node size of 32GB. Hyperthreading is disabled.

4. AMD MagnyCours: Dell 06JC9T board with 4 2.2GHz AMD Opteron 6174 processors and 128GB RAMwith a NUMA node size of 16GB. Each processor has two 6-core dies.

7.2 Deployment on different machines

Our first scenario shows how COD can adapt the deployment of CSCS to different hardware platforms using the OS policy engine, satisfying the performance SLA requirements.

Use-case description

In this use-case the main goal for CSCS is to determine the most suitable deployment strategy on a given machine so that it meets its response time SLA. In order to do that CSCS needs to derive: (a) the number of cores to run scan threads on, and (b) the correct size of its data partitions.

One can easily see when knowing the underlying architecture and having a cost function that characterizes the DB scan operation, like the one presented in section 4.4, this task is trivially solved. In that regard with this scenario we confirm the importance of deriving such DB cost models and matching them to the available hardware resources.

Implementation details

We now describe the interaction between the CSCS and the OS policy engine, and how the information exchange comes into place. As given in Listing 1, at startup CSCS binds to the OS policy engine: by registering to both the RM and the SKB (lines 1-4); and then registers its system-level properties with the SKB, namely that it is a CPU-bound task that could use all cores on the machine and highly cache- and NUMA-sensitive (lines 6-9).

Listing 1: Using the interface

```
1 rsmgr_client_connect(use_skb);
2 rsmgr_register_function();
3
4 skb_client_connect();
5
6 skb_system_fact(maxCores, MAX_CORES);
7 skb_system_fact(bound, CPU);
8 skb_system_fact(sensitive, cache);
9 skb_system_fact(sensitive, NUMA);
10
11 skb_add_fact("db_ntuple(3750000).");
12 skb_add_fact("db_tsize(315).");
13 skb_add_fact("db_nquery(2048).");
14 skb_add_fact("db_nupdate(256).");
15 skb_add_fact("db_rtime(3000).");
16
17 skb_add_fn("db_cost_fn(X,Y,Z,NrCores):-NrCores is
       (X*((0.47*Y)+265.29))/(3750000*Z)");
18
19 skb_add_fn(...query: see Listing 3...);
20 skb_execute_query(...query...);
```

It then populates the SKB with its application-specific facts such as: the size of its dataset in tuples, size of a tuple, the batch size of requests it needs to handle, the response time SLA, and the cost function as derived in Section 4.4 (lines 11-17). Finally, it registers the stored procedure (line 19) to derive the results needed: number of partitions, and the corresponding size of each partition (more details are provided in Listing 3).

Listing 2 contains Prolog code that retrieves system-level facts: list of all available cores, and list of all NUMA nodes' sizes. Both of these functions are used in the stored procedure in Listing 3.

Listing 2: Example for retrieving system-level facts

```
1 get_list_free_cpus(avail_cores):-findall(_,
     cpu_affinity(_,_,_),avail_cores).
2 get_list_numa_sizes(numa_sizes):-findall(N,
     memory_affinity(_, N, _), numa_sizes).
```

The CSCS' initial deployment stored procedure (see Listing 3) operates by first retrieving the CSCS-specific facts (lines 4-6), and then the necessary system-level facts using the example functions given in Listing 2 (lines 8-10). It continues the execution by calculating the total size of the dataset (line 12), and computing the minimum number of cores, as requested by the cost function, provided during the initialization phase (line 14). Since the CSCS is NUMA-sensitive, the dataset needs to be partitioned and distributed across the available NUMA nodes, and at least one core per NUMA node should be used - to guarantee data access locality. Consequently, each partition size must not exceed the size of a NUMA

Table 3: Derived deployments for different SLAs and hardware platforms

Hardware platform	SLA requested	Num. cores by cost function	Num. cores by NUMA node size	Total Num. cores	Size of partition	Measured mean response time
Intel Nehalem EX	2s	8	1	8	1GB	1.66s
	4s	4	1	4	2GB	3.27s
	8s	2	1	2	4GB	6.54s
AMD Barcelona	2s	8	5	8	1GB	2.18s[2]
	4s	4	5	5	1.6GB	3.55s
	8s	2	5	5	1.6GB	3.55s
AMD Shanghai	2s	8	3	8	1GB	1.68s
	4s	4	3	4	2GB	3.25s
	8s	2	3	3	2.67GB	4.33s
AMD MagnyCours	2s	8	1	8	1GB	1.87s
	4s	4	1	4	2GB	3.71s
	8s	2	1	2	4GB	7.37s

node. Thus, the stored procedure computes the minimum number of cores so that each partition fits in the smallest of all NUMA nodes (lines 16-17). The final number of cores/partitions needed is the maximum of both requirements (line 18). Once determined, the final number of cores is used to calculate the exact size of the partitions to be used (line 20). Finally, the stored procedure checks whether the total size of the dataset fits into main memory, and if the total number of cores required is in fact available in the machine. If either of these is not true, the query fails, notifying the CSCS that this machine cannot meet the desired constraints (lines 22-24) CSCS then operates on the obtained results and partitions its data accordingly.

Listing 3: CSCS' Initial deployment stored procedure

```
1  %status, nr_cores, part_size are output values.
2  dbos_cost_function(status, nr_cores, part_size):-
3
4   db_tsize(tsize), db_ntuple(ntuple).
5   db_nquery(nquery), db_nupdate(nupdate),
6   db_rtime(rtime),
7
8   get_free_memory(avail_memory),
9   get_list_free_cpus(avail_cores),
10  get_list_numa_sizes(numa_sizes),
11
12  memory is (ntuple*tsize),
13
14  db_cost_fn(ntuple,(nquery+nupdate),rtime,
        sla_nr_cores),
15
16  min(numa_sizes, min_numa_size),
17  numa_nr_cores is (memory/min_numa_size),
18  max([numa_nr_cores, sla_nr_cores], nr_cores),
19
20  part_size is (memory/nr_cores),
21
22  ( nr_cores > length(avail_cores) -> status = 1;
23    memory > avail_memory -> status = 2;
24    status = 0;
25  ).
```

During runtime, some of the application-specific properties may change: for example, the size of the batch of requests that needs to be handled. In that case CSCS simply modifies these values in the OS policy engine, and triggers a re-computation of the stored procedure.

The outcome of an application-specific stored procedure results in new CSCS' system-level properties that are added in the SKB. By design, whenever some system-specific properties change or are added/removed, the RM calls the global allocation function in the policy engine to compute concrete core IDs to be allocated to all registered applications.

In this particular use-case, the CSCS' stored procedure computes the minimum requirement of core count and the size of a partition, both of which are added as system-level properties in the SKB. After the global allocation plan completes its computation, the RM sends CSCS an upcall containing the concrete core IDs to use. Based on that, the CSCS can pin its scan threads and allocate memory from the corresponding NUMA nodes.

Experiment evaluation

We deployed COD on all machines, described in 7.1, and now present the resulting allocation of scan threads to cores, as suggested by the OS policy engine.

The CSCS engine pushed to the SKB the following application-specific facts: $30 \cdot 10^6$ tuples, each of size 315B (resulting in a total dataset size of 8GB) and a batch size of requests containing 2048 queries and 512 updates. We varied the SLA response time constraint between 2 and 8 seconds.

Table 3 shows the results of the calculation performed at the SKB and illustrates how the suggested configuration varies considerably for different SLA requests and hardware platforms. The final column of the table shows the results of the actual runs with the proposed configuration. The experiment values confirm that in every case, but one, COD does meet the SLA as predicted. [2]

In theory, this calculation could be performed entirely inside the database based on information requested by CSCS from the OS, regarding details about the underlying architecture and available resources. However, submitting a query to the OS policy engine means that CSCS does not need to understand each machine's hardware configuration. More importantly, since CSCS has now delegated useful knowledge to the OS about its own resource requirements (including how it can trade-off cores for memory), the OS is in position to do more intelligent resource reallocation, and automatically compute CSCS' deployment in response.

Moreover, in the next scenarios we will see cases where this deployment decision is heavily dependent on system runtime state

[2]The case where the SLA is not met is due to the use of the same cost function for all machines rather than tailoring it to each one of them. Some of the constants depend on CPU clock frequencies that vary from machine to machine and we have not adjusted the formula accordingly.

and the resource utilization of other applications present in the system. In this case, the OS is the only location where all this information is available and it makes little sense to pass it on to the database.

Communication and computation overhead

As presented in the implementation details of this scenario, the communication overhead that COD introduces can be calculated as the number of messages that had to be sent in order to add the required facts to the SKB and trigger computation of the application stored-procedure and global allocation plan. In this particular scenario (see Listing 1), the initialization phase requires three function calls: connection to the SKB, and registration to the RM. It then needs four calls to set up the system-level properties, and seven calls to place the application-specific facts including the cost function and the stored procedure (note that we calculate one function call per fact). Lastly, we need one call that triggers the computation of the stored-procedure, which waits upon the callback containing the results of the calculation, and the final call belongs to the upcall notification from the global allocation plan.

In total that means that we have introduced sixteen function calls for the initialization phase at deployment time, out of which four are fixed and the rest depend on our application properties. As we can see, one immediate candidate for optimization in the interface is to decrease the communication overhead by grouping the redundant calls of adding application properties to the SKB into a single function call. This optimization is left for future work.

The computation overhead for this phase solely depends on the time it takes for the SKB to calculate the output of the CSCS-stored procedure. We measured this overhead during our last experiment to be 0.18 milliseconds (see also Table 4) Since it is invoked quite infrequently, i.e. only at deployment time and when one of the application-specific facts have altered, we can conclude that this overhead is not affecting the overall performance of the system.

Conclusion and possibilities for extensions

The results confirmed that even having a rather-simplistic cost model for the scan operation can result in a good deployment when knowing the underlying architecture. Ideally, the cost function should also take into consideration other processor properties (like CPU frequency) and cache layout so that we get more accurate results when deploying on different machines.

This scenario can easily be extended to other DBMS operations, apart from the full table scan, as long as we develop the corresponding cost function that best describes the operation's dependencies on the available system resources.

7.3 Deployment in a noisy system

In the second scenario we continue the discussion of the impact of information exchange between CSCS and the OS policy engine in COD, especially when deploying in a noisy system.

Use-case description

In this use-case we show that just knowing the architecture is not enough to do smart deployment on a machine that is being shared with other tasks. One also has to take into account the current state of the system and map accordingly, otherwise the deployment decision can result in a significant drop in performance.

Implementation details

In this subsection we describe the assignment of tasks (such as scan threads) to cores, in particular in the presence of other tasks sharing

Core	0	1	2	3	4	5	6	7
Task 1	X=1	X=0	X=0	X=0	X=0	X=0	X=0	X=0
Task 2	X=0	X=1	X=0	X=0	X=0	X=0	X=0	X=0
Task 3	X=0	X=0	X=1	X=1	X=0	X=0	X=0	X=0
Task 4	X=0	X=0	X=0	X=0	X=1	X=1	X=1	X=1
Shared L3	Cache 0		Cache 1		Cache 2		Cache 3	
NUMA	Node 0				Node 1			

Figure 4: Matrix showing core to task allocation, including NUMA, cache and core affinity.

the machine with the datastore. The concrete allocation of cores and NUMA nodes is performed by the constraint satisfaction solver in the SKB.

The basis for allocation is a matrix of free variables annotated with constraints derived from system-level and application-specific facts. The structure of the matrix is itself based on the particular hardware configuration at hand. Figure 4 shows an example, similar to the one we used for evaluation in this experiment, where the number of tasks assigned to each core is constrained to be zero or one[3]. This set of policy constraints is essentially equivalent to the space-time partitioning scheme proposed for the Tessellation OS [26], though COD's technique is rather more general: it subsumes shared caches and NUMA nodes, as well as sharing cores between appropriate tasks. Given that the matrix contains initially unconstrained free variables, we are not restricted to spatial placement of tasks. With time the solver obtains concrete values for these variables to indicate which core on which NUMA-node is allocated to which task. To derive a concrete core allocation, additional requirements on the number of necessary cores and memory consumption may be registered by the CSCS or other applications. The most common constraints used in the SKB for task assignment are the following:

MaxCores defines how many cores the application supports at most. Not all applications support an arbitrary number of cores in every phase. Some phases might be single-threaded or applications might have scaling limitations making it useful to tell the OS the maximum supported number of cores for a given phase. This constraint is implemented as the sum of the task's row has to be smaller or equal to *MaxCores*. In Figure 4, tasks 1 and 2 set *MaxCores* to 1 and task 3 to 2. Task 4 does not have any restrictions.

MinCores defines the minimum number of cores to be allocated to an application. It is implemented as a row sum constraint, but also includes an admission control check. To avoid infeasible allocations, the policy code checks in advance whether the sum of all *MinCore* values is at most the number of available cores.

WorkingSetSize defines the working set size per core. This property is important for NUMA-aware core allocation to tasks. If two cores share a 4GB NUMA node and the application processes a working set size of 4GB per thread, cores must be allocated on different NUMA nodes to accommodate the data sets.

CSCS declares *MinCores* to be the value computed by the previous experiment with the application-specific stored procedure, and *WorkingSetSize* to be the corresponding partition size. With this information, the OS policy engine can allocate concrete core IDs, which are NUMA-aware and meet SLA requirements.

[3]Please note that Figure 4 is a sample illustration of a machine with eight cores, used for simplicity, and does not match the actual machine used in the experiment.

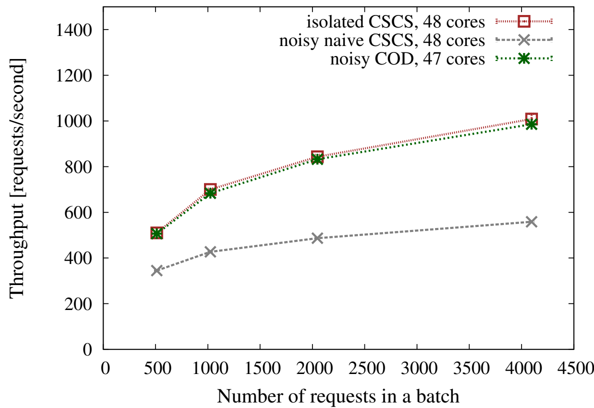

Figure 5: CSCS performance when deployed in a noisy system

Experiment evaluation

For this experiment we used the AMD MagnyCours machine, and we tested the deployment of CSCS engine with dataset size of 53GB, consisting of $180 \cdot 10^6$ tuples of size 315B each. The number of requests processed in a batch was varied between 512 queries and 128 updates to 4096 queries and 1024 updates. Prior to starting the execution of CSCS there was one other CPU-intensive task already in the system, pinned to execute on core 0.

Figure 5 presents the obtained results from the described experiment. We see the throughput of CSCS running on forty-eight cores both when spatially partitioned (CSCS is the only application in the system using these forty-eight cores) and when running in a noisy environment (in this case sharing one of the cores with a stressor program that is CPU-intensive). The results show that running CSCS collocating one of its scan threads with the other compute intensive task in the system can significantly impact performance, degrading the total throughput by almost fifty percent. The outcome is logical, since one slow scan thread in CSCS can delay the whole system, as all scan threads need to finish processing the same batch of requests before they start the next one.

This situation can be avoided only if CSCS is aware of the other tasks when deploying its threads. COD is superior than a naïve-CSCS engine because it relies on the OS policy engine which computes an allocation of forty-seven cores for CSCS, avoiding the already occupied core. This results in a performance almost as good as the original deployment in an isolated system when utilizing all forty-eight cores.

Communication and computation overhead

Communication overhead boils down to one function call that triggers the deployment and a notification response given by the OS policy engine that comes as a result from the re-calculation of the global allocation plan of the SKB.

The main overhead is the computation time in the SKB when evaluating the global allocation of resources to all registered applications. Table 4 summarizes the measured time of the computation of the global allocation plan for this experiment when having only CSCS, and when we have the CSCS plus one other application in the system.

As we can see from the results, the computation cost is quite small (in the range of milliseconds) for this setup. It can, however, grow higher as more applications enter the system, each with a specific set of system-level and application-specific facts that increase the complexity of the problem that the optimizer needs to solve.

Conclusion and possibilities for extensions

Even though in this use-case we have utilized a CPU-intensive task to create the noise in the system, that is not the only scenario where we need to be careful. Similar effects can be noticed when having a task that pollutes the caches, and thus thrashes the performance of the highly cache optimized scan threads of CSCS.

As described earlier, the decision for proper deployment of an application in a noisy system requires detailed information about the underlying hardware, the OS and system state, *and* CSCS internal properties, which COD collocates in the SKB. In a traditional database plus operating system deployment, there is no part of the system which has access to *all* this information.

7.4 Adaptability to changes

It is not only necessary that the CSCS engine is aware of the other running applications and tasks at deployment time but also during runtime. It is essential that the database can adapt to the dynamic system state so that it is not severely affected by the constantly changing noise in the machine.

Use-case description

In this experiment we show how COD can guarantee performance and maintain predictable behavior even in such a dynamic noisy environment. We will compare it with an execution of a naïve CSCS working stand alone and unaware of the changes undergoing in the system.

Implementation details

Whenever a new task enters the system it triggers a re-computation of the global allocation plan of resources. This can very often result in a decision to remove one of the cores previously allocated to CSCS, as long as the CSCS can still meet its SLA constraints i.e. satisfying the *MinCores* system-level property. In this case, the CSCS has to decide which scan threads should take over processing the affected portion of tuples.

In order to do that CSCS invokes the second application-specific stored procedure, registered at the SKB. This function (whose code with omit due to lack of space) derives to which cores the CSCS should move the affected tuples to, so that the new imbalance of load is evenly distributed to the other scan threads. It checks for memory availability on the corresponding NUMA nodes, and at the same time tries to maximize the number of sibling threads that will share the new load. Eventually, it responds with a list of core IDs to which CSCS will have to move the tuples to, as well as the corresponding number of tuples to be delegated to each of the cores. As soon as the data is re-distributed, CSCS kills the scanning thread on the core it just lost, and resumes the scan operation on the next batch of requests.

Experiment evaluation

This experiment was also conducted on the AMD MagnyCours machine, using a dataset of size 53GB ($180 \cdot 10^6$ tuples of size 315B each), and a batch size consisting of 2048 queries and 512 updates. Initially CSCS is the only application running in the system, utilizing all forty-eight cores. The SLA response time was set to be 3 seconds. The overall duration of the experiment is eighteen minutes and at about every four-five minutes we start another compute-intensive task. In the naïve run, i.e. when the experiment was executed without the OS policy engine, the external CPU intensive tasks were always scheduled at core #0. At the same time the CSCS is unaware of their entrance and thus does not react. Consequently, its performance gets degraded by almost fifty percent. In COD, the new incoming tasks are placed on separate cores, as suggested

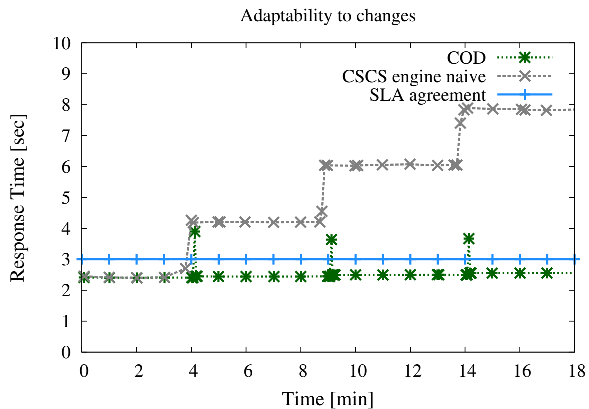

Figure 6: COD adaptability to changes in the system

Table 4: Policy engine computation overhead

Global allocation plan	time (msec)
only CSCS in the system	5.64
CSCS + 1 application in the system	13.28
CSCS-specific functions	time (msec)
initial deployment function	0.18
tuple re-distribution function	0.27

by the policy engine, and CSCS is notified every time it needs to release a core.

The results of this experiment are presented in Figure 6. It shows how the response time of CSCS, measured in seconds, changes in the course of the experiment. On one side, we can see the CSCS' naïve-run performance and how its response time increases dramatically with each new task entering the system. On the other side, we can see how the performance of CSCS integrated in COD remains steady even in the presence of other applications. The peaks that can be observed in COD are as a result of CSCS re-distributing the tuples to the other cores, still owned by the CSCS.

Communication and computation overhead

Communication overhead is the upcall from the RM that there was a change in the global allocation of resources, and the corresponding reaction from the CSCS engine invoking the tuple redistribution stored procedure, resulting in a total of two function calls.

Computation overhead on the SKB-side is the re-computation of the global allocation plan and consequently the cost of calculating the redistribution of tuples to a specific subset of CSCS-owned cores i.e. the second stored procedure of CSCS. Table 4 summarizes the measured values for this experiment. As we can see, the computation overhead for the stored procedure is almost negligible especially when compared to the time needed to do the actual re-distribution of tuples, which takes around 1.2 seconds.

Conclusion and possibilities for extensions

The possibility to easily adapt the execution of the DB storage engine as a result of receiving an upcall signal from the OS is of critical importance when sharing the machine with other tasks executed in parallel. This makes the CSCS flexible and adaptable to dynamic system state, that can maintain stable and predictable response time within the SLA agreement.

Being able to adapt as a result of receiving a signal from the OS, can easily be extended for other useful scenarios: for example, delegating to the operating system a task to monitor specific events on CSCS' behalf and getting notification when something goes beyond certain threshold. This can further enhance the database to be more resilient to dynamic changes in the utilization of resources and variations in workload.

8. CONCLUSION

The interaction between operating systems and database engines has been a difficult system problem for decades. Both try to control and manage the same resources but have very different goals. The operating system arbitrates between applications running on the machine, but inevitably has little knowledge of the applications' requirements. The database, meanwhile, tries to maximize transactional performance using deep knowledge of what the transactions do and the data needed to answer every query, but assumes its own statically configured partition of the machine to do so. The ignoring each other tactic followed in the last decades has worked because the homogeneity of the hardware has allowed databases to optimize against a reduced set of architectural specifications and over-provisioning of resources (i.e., running a database on a single server) was not seen as a problem.

With the advent of multicore and virtualization, these premises have changed. Databases will often no longer run alone in a server and the underlying hardware is becoming significantly more complex and heterogeneous. In fact, and because of these changes, both databases and operating systems are revisiting their internal architectures to accommodate large scale parallelism. Using COD as an example, we argue that the redesign effort on both sides must include the interface between the database and the OS.

COD is a proof of concept built out of several prototypes and experimental systems. Yet, COD illustrates very well what needs to be changed in both databases and operating systems to achieve a better integration as well as how such an integration could look like. As future work, we intend to investigate a negotiation protocol between the database and the OS to resolve the situation when requests cannot be met, adding runtime performance triggers on the policy engine that provide useful information to the database on how optimally it is operating, and linking the policy engine to the database query optimizer.

9. REFERENCES

[1] G. Alonso, D. Kossmann, and T. Roscoe. SwissBox: An architecture for Data Processing Appliances. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR'11, pages 32–37, January 2011.

[2] G. Alonso, D. Kossmann, T. Salomie, and A. Schmidt. Shared Scans on Main Memory Column Stores. Technical Report no. 769, Department of Computer Science, ETH Zürich, July 2012.

[3] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the fourth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IV, pages 96–107, New York, NY, USA, 1991. ACM.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 90–105, New York, NY, USA, 2003. ACM.

[5] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. L. Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD Conference*, pages 519–530, 2010.

[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 *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.

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

[9] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. hua Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *OSDI*, pages 43–57, 2008.

[10] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[11] G. Candea, N. Polyzotis, and R. Vingralek. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *PVLDB*, 2(1):277–288, 2009.

[12] G. Candea, N. Polyzotis, and R. Vingralek. Predictable performance and high query concurrency for data analytics. *VLDB J.*, 20(2):227–248, 2011.

[13] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] P. Druschel, V. Pai, and W. Zwaenepoel. Extensible Kernels are Leading OS Research Astray. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 38–, Washington, DC, USA, 1997. IEEE Computer Society.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[16] P. M. Fernandez. Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms. In *SIGMOD Conference*, page 492, 1994.

[17] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximising Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, Feb. 1999.

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

[19] J. Gray. Notes on Data Base Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1977.

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

[21] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.

[22] F. Huber and J. C. Freytag. Query Processing on Multi-Core Architectures. In *Grundlagen von Datenbanken*, pages 27–31, 2009.

[23] 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.

[24] 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.

[25] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/Mechanism separation in Hydra. In *Proceedings of the fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM.

[26] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, Mar. 2009.

[27] S. Peter, A. Schuepbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe. Design Principles for End-to-End Multicore Schedulers. In *Proceedings of the 2nd Usenix Workshop on Hot Topics in Parallelism (HotPar-10)*, Berkeley, CA, USA, June 2010.

[28] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, pages 60–69, 2008.

[29] T. Salomie, I. Subasu, J. Giceva, and G. Alonso. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *Proceedings of the Eurosys Conference*, April 2011.

[30] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter. A Declarative Language Approach to Device Configuration. *ACM Trans. Comput. Syst.*, 30(1):5:1–5:35, Feb. 2012.

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

[32] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. *SIGOPS Oper. Syst. Rev.*, 30(SI):213–227, Oct. 1996.

[33] M. Stonebraker, D. J. 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. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[34] M. Switakowski, P. A. Boncz, and M. Zukowski. From Cooperative Scans to Predictive Buffer Management. *CoRR*, abs/1208.4170, 2012.

[35] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2:706–717, August 2009.

[36] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[37] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 723–734. VLDB Endowment, 2007.