Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads

Iraklis Psaroudakis
École Polytechnique Fédérale de Lausanne
iraklis.psaroudakis@epfl.ch

Tobias Scheuer
SAP AG
tobias.scheuer@sap.com

Norman May
SAP AG
norman.may@sap.com

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
anastasia.ailamaki@epfl.ch

ABSTRACT

Task scheduling typically employs a worker thread per hardware context to process a dynamically changing set of tasks. It is an appealing solution to exploit modern multi-core processors, as it eases parallelization and avoids unnecessary context switches and their associated costs. Naïvely bundling DBMS operations into tasks, however, can result in sub-optimal usage of CPU resources: highly contending transactional workloads involve blocking tasks. Moreover, analytical queries assume they can use all available resources while issuing tasks, resulting in an excessive number of tasks and an unnecessary associated scheduling overhead.

In this paper, we show how to overcome these problems and exploit the performance benefits of task scheduling for main-memory DBMS. Firstly, we use application knowledge about blocking tasks to dynamically adapt the number of workers and aid the OS scheduler to saturate CPU resources. In addition, we show that analytical queries should issue a low number of tasks in cases of high concurrency, to avoid excessive synchronization, communication and scheduling costs. To achieve that, we maintain a concurrency hint, reflecting recent CPU availability, that partitionable analytical operations can use as a limit while adjusting their task granularity. We integrate our scheduler into a commercial main-memory column-store, and show that it improves the performance of mixed workloads, by up to 12.5% for analytical queries and 370% for transactional queries.

1. INTRODUCTION

In the era of big data, the key evaluation criterion for database management systems (DBMS) is their performance when evaluating an ever-increasing number of on-line analytical processing (OLAP) and on-line transactional processing (OLTP) queries. The challenge for DBMS is to leverage the increase of the processing power of modern multi-socket multi-core processors to maximize performance and efficiently service incoming queries over growing datasets.

Typically, the execution engine of a DBMS uses a single logical thread for short-lived latency-sensitive transactional queries. Long-running operations, such as complex transactional queries or analytical queries, may be parallelized using more logical threads. For highly concurrent workloads, simply issuing logical threads and leaving scheduling to the operating system (OS), leads to high creation costs and numerous context switches. The latter are incurred by the OS time sharing policy that balances the usage of a limited number of available hardware contexts among a higher number of threads using time slices [2, 25].

For resource management, including CPU, DBMS typically employ a query admission control to limit the number of processed queries. A query admission control, however, is a mechanism that operates on a per-query level, and can only indirectly avoid an excessive number of threads. It does not control resource utilization of queries after they have been admitted. Task scheduling [2, 6, 10, 13, 21] is a more appealing solution, as it uses a number of threads to process all operations for the whole run-time of an application. Tasks, which encapsulate operations, are stored in task pools, and worker threads are employed by the task scheduler to process the tasks. Task scheduling can be a complementary solution to query admission control, or even an alternative if tasks can be prioritized (see Section 4).

Moreover, task scheduling is well-suited for the recent wave of main-memory DBMS that forfeit disk-based storage in favor of performance. By removing I/O bottlenecks, main-memory DBMS can focus completely on optimizing CPU and memory utilization. Task scheduling can prove a powerful tool for main-memory DBMS, as it can automate the efficient usage of CPU resources, especially of modern shared-memory multi-core processors [2, 10, 13, 32], and can help developers easily parallelize database operations.

Recent popular task scheduling frameworks include the OpenMP API [3, 10] and Intel Thread Building Blocks (TBB) [2]. Their main advantage is that developers express partitionable operations, that can be parallelized with a variable number of tasks, using a high level of abstraction such as data parallelism. The high level of abstraction helps to automatically adjust the task granularity of analytical parti-

tionable operations, such as aggregations or hash-joins. The high level of abstraction, however, also turns out to be a disadvantage, as it cannot be used straightforwardly by already developed applications. Integration into a commercial DBMS would require a re-write of large portions of code, which is a process with significant cost and time considerations. Moreover, partitionable operations in commercial DBMS typically define their task granularity independently, without the use of a central mechanism for data parallelism. This is the common case, as optimizing the granularity of a single DBMS paritionable operation alone involves considerable research effort (see [12], for example, for partitioning in hash-joins). In this paper, we show how to adjust task granularity in a main-memory DBMS, in a non-intrusive manner, without the need of a high level of abstraction. We supply partitionable operations with a hint reflecting recent CPU availability, that can be used to adjust their task granularity. Our experiments show that when partitionable operations use this concurrency hint, overall performance for analytical and mixed workloads is significantly improved.

Furthermore, recent task schedulers, e.g. Intel TBB, use a fixed number of worker threads, equal to the number of hardware contexts. This is a standard technique to avoid overcommitment of CPU resources. The fixed concurrency level, however, is only suited for CPU-intensive tasks that rarely block [2]. We show that tasks in DBMS often block due to synchronization, especially in heavily contending OLTP workloads. Thus, the fixed concurrency level can result in under-utilization of CPU resources in DBMS workloads. In this paper, we show how the task scheduler can detect the inactivity periods of tasks and dynamically adapt its concurrency level. Our scheduler gives control to the OS of additional workers when needed to saturate CPU resources.

Contributions. We apply task scheduling to a commercial main-memory DBMS. Our experiments show the benefits of using task scheduling for scaling up main-memory DBMS over modern multi-socket multi-core processors, to efficiently evaluate highly concurrent analytical and transactional workloads. Our main contributions are:

- We show that a fixed concurrency level for task scheduling is not suitable for a DBMS, as tasks often block, especially in highly contending OLTP workloads. Our scheduler adapts its concurrency level, by detecting blocked tasks, and giving control to the OS of additional worker threads to saturate CPU resources.
- We show that using a hint reflecting recent CPU availability helps to adjust the task granularity of partitionable analytical operations. The concurrency hint improves overall performance significantly in cases of high concurrency, by reducing costs related to communication, synchronization and book-keeping.
- We show how we integrate our task scheduler into SAP HANA [18], a commercial main-memory DBMS. We show that our task scheduler improves the performance of highly concurrent analytical workloads (TPC-H [5]) by up to 16%, and of highly concurrent mixed workloads by up to 12.5% for analytical queries (TPC-H) and 370% for transactional queries (TPC-C [4]).

Paper organization. In Section 2, we present related work. In Section 3, we give an overview of SAP HANA, a commercial main-memory DBMS, and how we apply our task scheduler to SAP HANA. Next, we present the general architecture of our scheduler in Section 4, how we handle blocking tasks using a flexible concurrency level in Section 5, and how we use concurrency hints to aid task creators of partitionable operations adapt their task granularity in Section 6. In Section 7, we show our experimental evaluation. Finally, in Section 8, we present our conclusions.

2. RELATED WORK

Task scheduling for parallel programs and parallel systems is a broad field of research. Early related work focuses on static scheduling [19, 23], which is typically done at compile time and assumes that basic information about tasks (such as processing times, dependencies, synchronization, and communication costs) and the target machine environment are known in advance. Given perfect information, a static scheduling algorithm attempts to produce the optimal assignment of tasks to processors, that ideally balances their loads and minimizes scheduling overheads and memory referencing delays [20]. Perfect information, however, is hard to obtain for modern shared-memory multiprocessors and modern applications where tasks may be generated dynamically and at a fast pace. For example, queries in a DBMS arrive dynamically, and information about them can only be estimated, often with high relative errors. More recent related work focuses on dynamic scheduling, which is done on-the-fly at run-time [23, 35, 38]. Dynamic task scheduling does not require information about supplied tasks a priori, has less overhead than static scheduling, and provides automatic load-balancing and improved portability between different hardware architectures [27]. Dynamic task scheduling may also use run-time measurement to re-adapt scheduling decisions [30, 33, 38] for better data locality [38] or NUMA (non-uniform memory access) awareness [11, 34]. Our work studies the practical application and evaluation of dynamic task scheduling for the specific case of a main-memory DBMS on a single shared-memory multiprocessor machine. We focus on the minimization of context switches, handling blocked tasks, and adjusting task granularity for highly concurrent workloads, and leave optimizations for data locality and NUMA awareness for future work.

The common design of recent dynamic task schedulers involves task pools where tasks are submitted dynamically at run-time, and the scheduler employs a set of threads to work on the tasks [21]. There are two main categories of task schedulers [16]: breadth-first schedulers [31] and work-first schedulers [13] with various work-stealing techniques [6, 28, 32, 34] for load-balancing and improved predictability in real-time applications [27]. In breadth-first schedulers, when a task is created, it is placed in the task pools and the parent task continues execution. In work-first schedulers, the thread of the parent task switches to execute child tasks, for potentially better data locality [13]. Our scheduler combines both approaches: when a task generates a group of new tasks, the parent thread takes upon one of the new tasks, following the work-first approach, while the rest of the tasks are dispatched to the task pools, following the breadth-first approach, for load-balancing.

Hoffmann et al [21] provide a survey on different task pool implementations. Distributed task pools with stealing achieve best performance, as they minimize synchronization overheads and stealing amends load-balancing issues. We

follow a similar approach for our scheduler. Johnson et al [22] decouple contention management from scheduling and load management, to combine the advantages of spin and blocking locks. Our scheduler is not concerned with how locks are used, but uses the information about blocked tasks to dynamically adjust its concurrency level.

In highly concurrent analytical workloads with a large number of partitionable operations issuing tasks, granularity of tasks plays an important role in communication, synchronization, and scheduling costs [1, 14, 15, 26, 28]. Our experimental results corroborate these observations for main-memory DBMS. While we address workloads with long-running analytical queries as well as short-running OLTP-queries, we cannot rely on special implementations to avoid locking as proposed in [39]. Recent task scheduling frameworks such as OpenMP [3, 10] and Intel Thread Building Blocks [2] regulate task granularity by requiring the developer to express parallelism in a higher-level abstract manner and use this information. We assume that there is no central mechanism for data parallelism, and that partitionable operations define their task granularity independently. We use a concurrency hint, reflecting recent CPU availability, to adjust the task granularity of partitionable operations.

3. OVERVIEW OF SAP HANA

SAP HANA is a commercial main-memory DBMS, that aims to efficiently support OLTP and OLAP workloads [18]. It supports a multitude of data formats and provides facilities for an extensive spectrum of enterprise applications, under a flexible and componentized architecture [17].

The DBMS architecture is depicted in Figure 1, (see [18, 36] for an overview). SAP HANA provides four main-memory storage engines: a column-store, suited for OLAP-dominant and mixed workloads, a row-store, suited for OLTP-dominant workloads [37], as well as a graph engine and a text engine [18]. The transaction manager uses multi-version concurrency control (MVCC) [36]. Further components provide extensions for enterprise applications, and various application interfaces such as SQLScript and calculation models.

In this architecture realizing parallelism is the key to good scalability. For scaling out, SAP HANA supports distributed

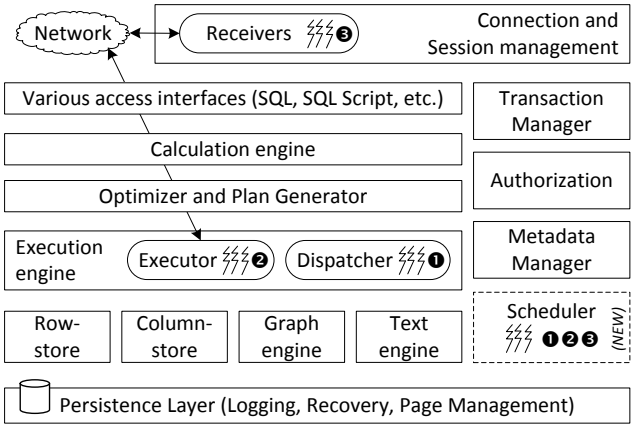

Figure 1: The general database architecture of SAP HANA. Our new scheduler integrates the three main thread pools of SAP HANA.

query execution plans. Scalability on every node is achieved through multi-threaded algorithms that exploit data parallelism inherent in many database operations. These algorithms are implemented with a special focus on hardware-conscious design and high scalability on modern multi-core processors. Furthermore, quick response times for short-running queries is provided by an efficient session management and client interface [24].

Although the architecture of SAP HANA enables the loose coupling of all components and their selective leverage, the components are independent as far as execution is concerned. Some components use their own thread pools, while they assume they can fully exploit system resources, unaware of concurrent operations of other components. The three major thread pools are the following. (1) The *Dispatcher* is a simple task graph scheduler used typically for parallelizing partitionable analytical operations. (2) The *Executor* is a task graph scheduler that processes plans of operations that can potentially be distributed across machines. Plan nodes can use the Dispatcher for parallelizing on one machine. (3) The *Receivers* are threads that process received network requests. Short-running transactions are typically completely executed within a Receiver. For more complex transactions, longer analytical or distributed queries, the Receiver uses the Executor, which in turn uses the Dispatcher.

The independence of these three thread pools poses several problems. Firstly, when all thread pools are active, the number of logical threads may surpass available hardware contexts, over-committing CPU resources and resulting in context switching costs. Secondly, DBMS administrators can be confused while configuring how many threads each pool should use. A good configuration is highly dependent on the workload. Thirdly, developers need to learn three different thread implementations, and battle with parallelizing their operations across each one of them. We design our new scheduler to address the aforementioned issues, and better scale up on a single shared-memory multiprocessor system by integrating the different thread pools of SAP HANA.

Our scheduler constitutes a new component in the general architecture of a DBMS (see Figure 1), and is orthogonal to other components such as the Persistency Layer or the Transaction Manager. It is important to note that our scheduler does not compromise any transactional correctness or persistency semantics.

4. SCHEDULER ARCHITECTURE

To support fast scheduling of all heterogeneous general-purpose tasks, we opt for a dynamic task scheduler that does not require or process any a priori execution information about the tasks, except for potentially a directed acyclic graph (DAG) defining their correlations and ultimately their order of execution. The DAG can take any form, with the only restriction of having a single root node. This does not prevent the creation of single-node graphs. Each node in the task graph can contain any piece of code. A node can potentially spawn a new task graph, or issue itself again to the scheduler. The developer is free to co-ordinate synchronization among tasks, since we take care to maintain a flexible concurrency level (see Section 5). Optionally the developer can assign a priority for the task graph, which results in a decreased or increased probability of being chosen for execution. The developer then dispatches the root node to the

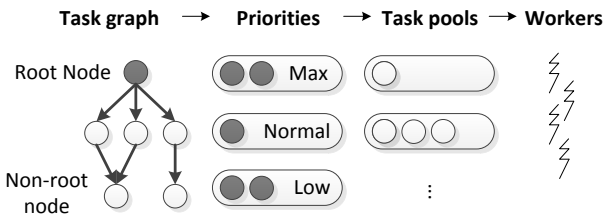

Figure 2: The data structures used by the scheduler.

scheduler, and he can wait for execution of the task graph or continue without waiting.

The scheduler maintains two sets of queues, depicted in Figure 2. The first set contains one queue per priority and holds the root nodes of the submitted graphs that have not yet been initiated for execution. The second set contains queues that hold the non-root nodes to be executed. The second set actually constitutes the main distributed task pools for our scheduler. The task pools can further be sorted by node depth, in order to favour execution of deep-running graphs, or by the timestamp of the owning query, in order to favour execution of earliest queries. For our experiments, we sort the task pools by node depth, because resource-intensive queries tend to create deep-running graphs, and we take care to finish these queries early, in order to free up the resources such as the memory.We use distributed task pools to reduce synchronization contention. Currently, we create as many task pools as the number of sockets. More task pools can also be created if the number of hardware contexts in one socket is high and results in synchronization contention for the task pool. Each worker thread is assigned to a specific task pool in a round-robin fashion according to its ordinal identifier. If the worker thread finds its assigned task pool empty, it starts querying other task pools, in a round-robin fashion, and steals tasks [21].

When the task pools are empty, a free worker retrieves a root node from the queues of priorities, with a probability that favours prioritized root nodes. This probability is configurable, in order to prevent starvation of root nodes with low priorities. We note that in our experiments of Section 7, all tasks have the same priority. After executing the root node, the worker thread continues executing the first descendant for better data locality, while the rest of the descendants are dispatched randomly to the task pools for load-balancing. When the task pools are not empty, a free worker retrieves his next task from the task pools. When a non-root node is executed, the worker checks which descendants are ready for execution, takes upon the first of them and dispatches the rest to the task pools.

Integration. Our simple design allows a fast integration of all three main thread pools of SAP HANA into our scheduler (see Section 3). Without having to worry about synchronization (see Section 5), we quickly bundle old tasks and generic blocks of code into tasks for our new scheduler. As is standard for task schedulers [2], we do not bundle I/O-bound operations into tasks. These operations are executed by separate threads that are handled by the underlying OS scheduler. Since these threads do not reserve any worker threads from our scheduler, we can keep the system busy with CPU-intensive tasks while there are I/O operations. It is easy to detect I/O-bound operations in main-memory DBMS, as general query execution is CPU-bound or memory-bound. Heavy I/O operations, such as savepoints, are only done periodically and in the background to minimize the disruption of the general performance of the database [18]. Thus, I/O-bound operations are traced mainly inside the persistence or network layer.

The most significant difficulties we encountered were propagating exceptions in the task graph to the creator thread that may wait for the task graphs associated with a query, and inheriting thread-local storage of the creator thread to the workers handling nodes of the task graph. Thread-local storage in SAP HANA is used to store the transactional MVCC details of the query, which are used by the Transaction Manager (see Figure 1).

Scheduling policies. The design with the two sets of queues allows us to provide different scheduling policies. Our default policy dictates that a free worker thread should retrieve its next task from the task pools, if they are not empty, with a high probability. In our experiments, this probability is set to maximum (1.0). Thus, new graphs are initiated after older graphs have completed. This policy favours throughput over latency, and takes care to finish earlier initiated graphs that may hold resources, such as memory, before initiating new graphs. The administrator, however, may want to be fair to the latencies of all incoming queries, and thus we provide him with a setting to decrease the aforementioned probability, so that free worker threads can execute a root node even if the task pools are not empty. Even though this policy is more fair for all incoming queries, general performance and throughput is hurt, due to increased contention from many concurrent queries. Our default policy provides semantics similar to a light-weight admission control manager for CPU resources. We use the default policy for our experiments, as it provides best results.

Watchdog thread. To control workers, but also to monitor the state of execution, we reserve an additional watchdog thread. The watchdog typically sleeps, but wakes up periodically to gather information and potentially control worker threads, similar to the notion of centralized scheduling [20]. We use light-weight mechanisms for monitoring, based on statistical counters, such as the number of waiting and executing tasks, how many tasks each worker thread has executed etc. These counters are updated using atomic instructions by each worker thread and the watchdog.

5. DYNAMIC ADJUSTMENT OF CONCURRENCY LEVEL

Typical task schedulers employ a number of worker threads equal to the number of hardware contexts. This level of concurrency is suitable for task schedulers whose aim is to handle CPU-intensive tasks that do not block frequently [2]. Our aim, however, is to integrate already-developed general-purpose code into tasks. We need to handle tasks that can include heavy usage of synchronization primitives and locks. A representative example are highly concurrent and contending transactional workloads, such as TPC-C [4]. When tasks are inactive, our scheduler takes care to overlap inactivity periods with additional worker threads and saturate CPU resources. Next, we describe the different states of inactivity that a task can be in, and how our scheduler handles them, by adapting its concurrency level at run-time.

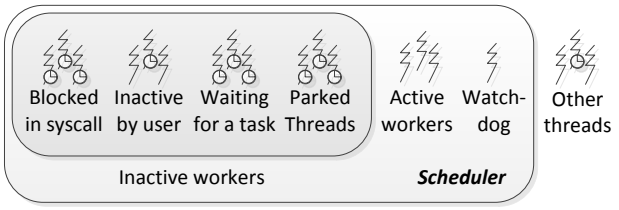

Figure 3: The scheduler's types of worker threads.

Blocked workers. The OS scheduler is the first to know when a thread blocks after a system call for a synchronization primitive. It then cedes the CPU to another thread waiting for its time slice. If we set a fixed number of worker threads equal to the number of hardware contexts, blocked threads will not be overlapped by other threads, as the OS scheduler does not have knowledge of any other working threads in our application. This results in under-utilization of CPU resources, as the OS scheduler could potentially schedule another worker thread while a worker thread blocks. Additionally, since we do not know how the developer synchronizes tasks, a fixed concurrency level can lead to potential deadlocks, if the interdependency edges between nodes are not correctly used. For example, if a node in a task graph requires a conditional variable from another node at the same level of the task graph, the latter node may not be scheduled in time if the nodes in the level are more than the available hardware contexts. Deadlocks can also happen if code synchronizes heavily between different task graphs.

To avoid deadlocks and under-utilization of CPU resources, we argue that a scheduler handling general-purpose tasks should not use a fixed concurrency level. Our watchdog periodically checks for blocked worker threads, and activates additional worker threads, that get scheduled by the OS immediately and overlap the inactivity period. Thus, we co-operate with the OS by voluntarily adjusting the concurrency level, and giving control to the OS of additional worker threads when needed to saturate CPU resources. We exploit both the advantages of task scheduling and the OS scheduler: Task scheduling ensures that the number of working threads is small enough so that costly context switches are avoided. By dynamically adjusting the concurrency level, we exploit the capability of the OS scheduler to quickly cede the CPU of a blocked thread to a new worker thread.

To detect blocks efficiently, we do not use OS synchronization primitives directly. DBMS typically encapsulate these in user-level platform-independent data structures. For example, SAP HANA on Linux uses a user-level semaphore based on atomic instructions, that calls the futex facilities of Linux when needed. We leverage these user-level synchronization primitives to know when a worker thread is about to call a potential system call that could block.

Active concurrency level. We define:

$$\text{concurrency level} = \text{total number of worker threads}$$

The concurrency level is variable. There can be a number of inactive workers, such as blocked threads, and a number of active workers. We are mainly interested, however, in keeping the total number of active workers as close as possible to the number of hardware contexts, in order to saturate CPU resources. For this reason, we define:

$$\begin{aligned}\text{active concurrency level} = \text{concurrency level} \\ - \text{inactive workers}\end{aligned}$$

When threads resume from inactivity, they are considered again in the active concurrency level, which can at times be higher than the number of hardware contexts.

Parked threads. In order to fix a high active concurrency level, the scheduler gets the chance to pre-empt a worker when it finishes a task. We cannot pre-empt a worker in the middle of a generic task, as it can be in a critical section and the consequences are unpredictable. Instead of ending the thread, we keep it suspended, in a *parked* state. The watchdog is responsible for monitoring if the active concurrency level gets low and waking up parked threads. Parked threads overcome the costs of creating logical threads, which include the expensive allocation of their stacks.

Other inactive threads. Apart from blocked and parked threads, we define two additional states of inactivity. Firstly, there can be tasks that wait for another task graph. This inactivity state is comparable to OpenMP's suspend/resume points (e.g. `taskwait`) [10], or to TBB's wait methods (e.g. `wait_for_all`) [2]. Secondly, we give the developer the opportunity to explicitly define a region of code as inactive. This is useful for code regions that are not CPU-intensive, such as the I/O-bound commit part of a transaction that needs to write in the redo log on disk. We note that in our experiments, we do not specify these I/O-bound parts of a transactional task as inactive. For both these cases of inactive threads, a new worker thread is activated immediately if allowed by the active concurrency level, instead of being activated by the watchdog. We note that while a worker thread is blocked, parked, or waiting for another task graph, it is also considered inactive by the OS scheduler. A code region, however, that is defined by the developer as inactive, pertains only to our scheduler's accounting for its active concurrency level, while the OS considers the relevant worker thread as runnable and schedules it.

All the aforementioned types of workers are shown in Figure 3. The total number of inactive workers is defined as:

$$\begin{aligned}\text{inactive workers} = \text{blocked workers} \\ + \text{inactive by user} \\ + \text{workers waiting for a task graph} \\ + \text{parked workers}\end{aligned}$$

Avoiding too many active threads. We note that activating additional worker threads in place of inactive workers may not always be optimal. If the inactivity period of a worker is short, and the newly activated worker begins executing a large task, then when the first worker returns from inactivity, there will be two worker threads active. If this situation is repeated many times, the active concurrency level can get much higher than available hardware contexts, leading to context switching costs from the OS scheduler.

For this reason, it is important to handle inactivity states carefully. The inactivity states where the developer specifies a code region as inactive, and where a task waits upon another task, are typically not too short. These inactivity states lower the active concurrency level, and immediately activate additional worker threads that increase the active concurrency level up to the number of available hardware

contexts. The duration of the inactivity state of blocked threads, however, is generally unknown, and can be too short. Thus, blocked tasks are handled differently: they only lower the active concurrency level, and do not immediately spawn additional workers. The active concurrency level is increased only when the watchdog checks it periodically and attempts to fix it by activating additional worker threads, or in case another inactive worker thread resumes activity in the meanwhile and thus increases the active concurrency level and is allowed to continue working on next tasks. Thus, too short block periods are typically hidden between the intervals of the watchdog and of active tasks. Even in the bad case that the active concurrency level gets too high, this is quickly fixed when active workers finish their current tasks and are pre-empted and parked in order to fix the active concurrency level.

To support our intuition, our experiments with SAP HANA have an active concurrency level that is most of the time equal to the number of hardware contexts (see Section 7). We note that the watchdog interval we use in our experiments is 20ms. We have experimented with larger intervals as well, but have not noticed significant differences in the active concurrency level. Nevertheless, in order to avoid worst-case scenarios, the watchdog can check periodically if the active concurrency level gets much higher than the number of hardware contexts.

6. DYNAMIC ADJUSTMENT OF TASK GRANULARITY

Partitionable operations can be parallelized using a variable number of tasks. Many analytical operations in a DBMS fall into this category, e.g. aggregations and hashing. If a column needs to be aggregated, it can be split into several parts which can be processed in parallel independently. This is a classic example of data parallelism using a fork-join programming structure [9].

For this kind of partitionable operations, a number of tasks lower than the number of available hardware contexts, i.e. a coarse granularity of tasks, can under-utilize CPU resources. A higher number of tasks (up to the number of available hardware contexts) means that the partitionable operation can potentially use more CPU resources and decrease its latency. Using a fine granularity, however, can potentially introduce additional costs for communication, synchronization and scheduling [1, 14, 26]. Thus, a balance is required for the task granularity.

Task schedulers like Intel TBB [2] can greatly help in case of partitionable operations. As the developer expresses partitionable operations through higher-level algorithmic structures and data parallelism, the framework employs a centralized mechanism for adjusting task granularity.

In commercial DBMS, however, the majority of partitionable operations do not use a central mechanism for data parallelism. This is the common case because optimizing the granularity of a single DBMS paritionable operation alone involves considerable research effort (see [12], for example, for partitioning in hash-joins).

There are typically distinct components for partitionable operations that handle data parallelism and granularity independently. In SAP HANA, for example, each partitionable operation employs heuristics to find the right task granularity, based on factors such as data size, communication costs, and the number of hardware contexts of the system. We should note that the degree of parallelism does not affect the choice of a query plan of the optimizer in SAP HANA.

In this paper, we are not concerned with how each component calculates task granularity, but with how task granularity affects performance when numerous concurrent queries, possibly with other partitionable operations, are being processed. The problem we notice is that partitionable operations calculate the number of tasks irrespective of other concurrent tasks. In the worst case, every operation can dispatch a number of tasks equal to the number of hardware contexts. Our experiments show that this practice results in a myriad of tasks in cases of high concurrency, and increased book-keeping and scheduling costs. Thus, task creators for partitionable operations need to adjust their task granularity by taking into consideration other concurrent tasks. To solve this problem, our scheduler provides information about the state of execution to partitionable operations that they can use for the calculation of task granularity.

Concurrency hint. If a partitionable operation issues more tasks than can be handled by free worker threads at the moment, there is little to no benefit for parallelism, and redundant scheduling costs. The intuition is that in cases of high concurrency, when the system is fully loaded and free worker threads are scarce, partitionable operations should opt for a very coarse granularity in order to minimize the number of tasks to be processed.

Our scheduler can provide task creators with information about the current availability of computing resources, as it has knowledge of the active concurrency level of the whole DBMS. Thus, it can give a hint to task creators about the maximum number of tasks they should create at the moment. The watchdog is responsible for calculating the *concurrency hint*, which is an exponential moving average of the free worker threads in the recent past. The free worker threads and the concurrency hint are defined as:

$$\begin{aligned}\text{free worker threads} = max\{0, &\text{number of hardware contexts} \\ &- \text{active concurrency level}\}\end{aligned}$$

$$\begin{aligned}\text{concurrency hint} = a * &\text{free worker threads} \\ &+ (1.0 - a) * \text{previous concurrency hint} \\ &\text{where } 0 \leq a \leq 1.0\end{aligned}$$

Due to the dynamic nature of our workers, which can change status often and quickly, an average can give better results than an absolute value. For our experiments, we use an exponential moving average, with equal weight for the free workers threads of the previous observations and the currently observed number of free worker threads (i.e. $a = 0.5$). The sampling rate is configured at 50ms in our experiments, which provides reasonable smoothing over the recent past, and also quickly captures changes in the number of free worker threads. Due to the fact that the exponential moving average captures all past observations, we take care to reset it to the number of hardware contexts when it surpasses a predefined threshold. This threshold is set to 90% of the number of hardware contexts for our experiments.

Splitting. It can happen that a partitionable operation gets a low hint. If resources are freed up later on, it will under-utilize CPU resources. To correct this behaviour, we follow a strategy similar to the data parallelism concept of

splitting ranges in Intel TBB or the lazy task creation [28]. Each task needs to check the concurrency hint periodically. If the hint gets high, the task should decide if it should split into two more nodes. Those two nodes can recursively split again. At the moment, we have implemented splitting for the simple cases of aggregations and calculations. Nevertheless, since we are interested in highly-concurrent workloads, when the system is fully loaded, the absence of splitting for the rest of partitionable operations does not affect performance significantly in our experiments.

7. EXPERIMENTAL EVALUATION

We compare the following variations of SAP HANA:

- Multiple-Pools, which is the original version of SAP HANA, with the different thread pools showed in Section 1. This serves as our baseline.
- Single-Pool-NoSys, which integrates the different thread pools of Multiple-Pools into tasks for our new scheduler. This variation assumes workers blocked on synchronization primitives as working, and includes them in the active concurrency level of the scheduler. Also, this variation defines the concurrency hint as the number of hardware contexts, to simulate the old behaviour.
- Single-Pool, which is like Single-Pool-NoSys, but uses a flexible concurrency level by assuming workers blocked on synchronization primitives as inactive.
- Single-Hints, which is like Single-Pool, but with the concurrency hint following the exponential moving average of free workers in the recent past. Partitionable analytical operations adjust their task granularity according to the concurrency hint. This variation is the best version of our new scheduler for SAP HANA.

Our server has eight ten-core processors Intel Xeon E7-8870 at 2.40 GHz, with hyper-threading enabled, and 1 TB of RAM. Each core has a 32KB L1 cache and a 256KB L2 cache. Each processor has a 30MB L3 cache, shared by all its cores. The OS is a 64-bit SMP Linux (SuSE), with a 2.6.32 kernel. Unless stated otherwise, every data point in our graphs is an average of multiple iterations with a standard deviation less than 10%. Our measurements for context switches and CPU times are gathered from Linux. The total number of instructions retired are gathered from Intel Performance Counter Monitor. For all expriments, we warm up the DBMS first and there are no thinking times. We use transaction level snapshot isolation with repeatable reads. We make sure that all queries and clients are admitted, and we disable query caching because our aim is to evaluate the execution of the queries and not query caching.

7.1 Analytical workload

We use the TPC-H benchmark [5] with a scaling factor 10, stored in a column-store. We measure performance by varying the number of concurrent queries, and measuring the response time of each variation from the moment we issue the queries until the last query returns successfully. Queries are instantiated from the 22 TPC-H query templates in a round-robin fashion, with the same parameters for each query template for stable results, but without query caching. We start measuring from 32 concurrent queries, to include

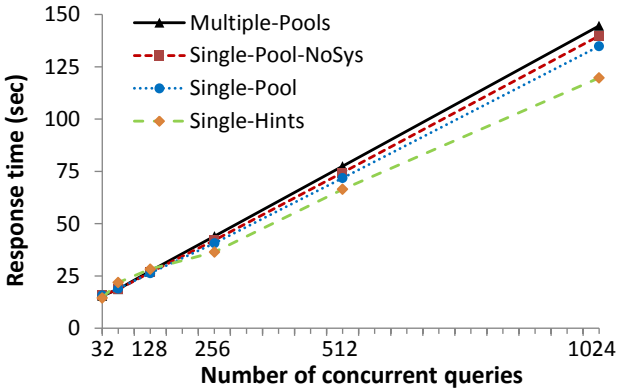

Figure 4: Experiment with TPC-H queries.

all query templates, up to 1024. The results are shown in Figure 4. In Figure 5, we include performance measurements for the case of 1024 queries. Our scheduler does not affect cache-related behaviour significantly and cache miss rate in all variations stays at similar levels.

Single-Pool-NoSys improves performance of Multiple-Pools by only 3% for high concurrency. The main improvement comes from reducing lock contention, as shown by the reduction in system CPU time. The number of context switches has increased, even though we integrate all thread pools of Multiple-Pools into a single task scheduler. We attribute this to the fixed concurrency level. When workers block on synchronization primitives, the OS does not have knowledge of any additional workers to schedule. It replaces the time slice of a blocked worker with any non-CPU-intensive thread, outside the scheduler, with a small time slice. This is also reflected in the increased idle CPU time.

Single-Pool, which has a flexible concurrency level, overcomes this problem and improves performance of Multiple-Pools by 7%. When many worker threads block, the watch-

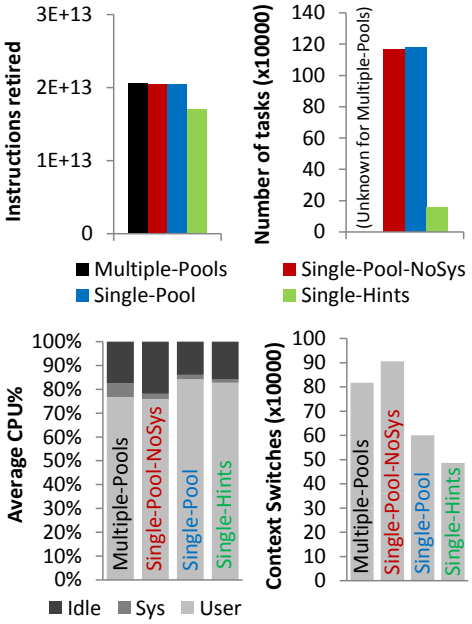

Figure 5: Measurements for the case of 1024 TPC-H concurrent queries.

dog issues more worker threads and gives the chance to the OS scheduler to schedule CPU-intensive worker threads with new tasks and full time slices. That is reflected in the decreased idle CPU time, and the fewer context switches.

Single-Hints results in the best performance improvement of Multiple-Pools by 16%. The coarser task granularity leads to a reduction of the total number of tasks by 86%. We achieve a significant reduction in unnecessary book-keeping and scheduling costs, which is reflected in the 16% reduction of the total number of instructions retired. Furthermore, we corroborate previous related work that a coarser granularity results in less costs for synchronization and communication [1, 14, 26], since system CPU time is further decreased.

We notice the effect of not splitting tasks (see Section 6) for the case of 64 and 128 concurrent queries of Figure 4, where the standard deviation for Single-Hints is up to 30%. In a few iterations, a partitionable operation that got a low hint was left in the end alone, under-utilizing CPU resources and prolonging response times. We remind that we have enabled splitting for aggregations and calculations in SAP HANA, and we plan to enable splitting for more partitionable operations. In our experiments with Single-Hints, however, all partitionable operations use the concurrency hint as a limit in order to show the effect of hints for high concurrency when the effect of not splitting is not obvious.

To better understand the effect of the flexible concurrency level and hints throughout the whole experiment, we show the timelines for Single-Pool-NoSys and Single-Hints for the case of 1024 queries in Figure 6. For Single-Pool-NoSys, we notice the effect of bursts of too many tasks being issued to the scheduler. The redundant scheduling, communication, and book-keeping costs of these bursts of numerous tasks result in erratic behaviour of the CPU utilization. In contrast, the timeline for Single-Hints presents a much smoother run-time. The majority of the tasks are issued by all queries in the beginning of the experiment, and they are gradually scheduled until the end of the experiment. The CPU utilization line is more stable. Additionally, the flexible concurrency level results in a few more worker threads that raise the CPU utilization average in comparison to Single-Pool-NoSys. Nevertheless, we still note that CPU utilization is not fully saturated, and we still have idle time.

Thus, we note that the standard scheduling technique of having an active number of workers equal to the number of hardware contexts does not fully saturate CPU resources, since idle CPU time is significant in all variations. A similar observation about idle time has been done in a recent evaluation of modern column-stores [8]. This is attributed to the fact that DBMS operations are not purely computationally intensive [7], and the OS can exploit a few more threads to overlap stalls (e.g. for memory). We plan to investigate how to dynamically raise the active concurrency level, to minimize idle CPU time, while not deteriorating context switching costs and cache miss rate.

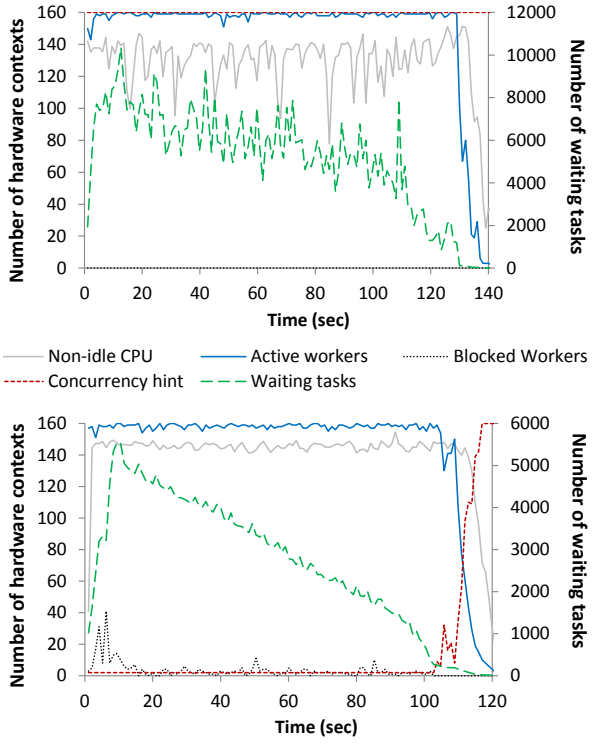

Figure 6: Timelines for Single-Pool-NoSys (above) and Single-Hints (below), for the case of 1024 concurrent TPC-H queries.

7.2 Mixed workload

We measure the performance of the different variations by running a throughput experiment for 15 minutes for TPC-H and TPC-C [4] concurrently on disjoint datasets. This use case can happen if a hot transactional dataset is copied on the same server for analytics. Our intention is to see how each workload behaves when they co-exist on the same server, and how scheduling affects their performance.

For TPC-C, we use a database of 200 warehouses, stored in a column-store, 200 clients, and measure the total average successful new-order transactions per minute (tpmC). Our TPC-C driver is based on a previous project [29]. Before every experiment, we re-load our initially generated TPC-C database, in order for all experiments to start with the same columnar data and have no data in their delta storages [36].

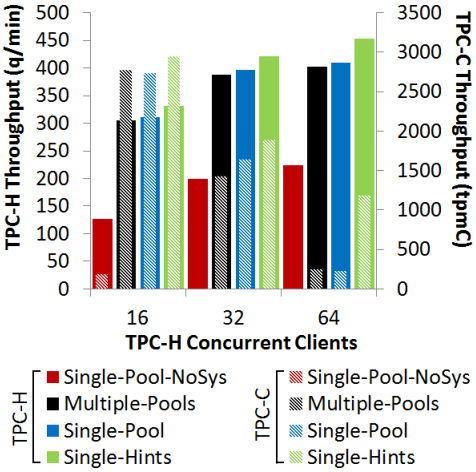

Figure 7: Throughput experiments with a mixed workload, consisting of TPC-H and TPC-C clients on disjoint datasets. Each experiment involves a TPC-H throughput (bar with solid color) and a TPC-C throughput (bar with pattern).

We turn off merge operations for the column-store [36], in order to avoid unpredictable periods of inactivity due to the TPC-C tables being merged and locked, and achieve a stable behaviour for all variations. Due to the absence of merge operations, we note that the tpmC slowly degrades in every experiment, but as this behaviour is stable for all variations, and because each variation starts the experiment with the same data and conditions, this does not prevent us from comparing the total average tpmC. For TPC-H, we use a scaling factor of 10 and vary the number of concurrent clients, in order to see the effect that long-running analytical tasks have on the short-lived tasks of TPC-C. Clients issue queries from the TPC-H Q1-Q22 templates in a global (not per client) round-robin fashion, as in Section 7.1. We report the average of TPC-H queries completed per minute. The results are shown in Figure 7. We note that CPU resources are completely saturated for the case of 32 and 64 concurrent TPC-H clients, and that standard deviation for some cases of very low TPC-C throughput is up to 50%. In Figure 8, we show measurements for the case of 64 TPC-H clients.

Overall, we observe that increasing the number of concurrent TPC-H clients overshadows the TPC-C clients. This is due to the fact that we do not change the number of TPC-C clients in this experiment, and overwhelm the database with an increasing number of resources-intensive TPC-H clients. Also, TPC-C queries are typically executed in a single task, while TPC-H queries usually pertain a task graph with multiple tasks. This results in many more tasks for TPC-H than TPC-C. This trend serves as motivation to find ways to give the DBMS administrator the possibility of favouring either transactional or analytical queries. Note that we cannot do this simply with the queues of priorities of our scheduler, since Receivers handle all incoming queries in the same way and bundle them into tasks for the scheduler.

We observe that Single-Pool-NoSys results in the worst performance, due to its fixed concurrency level. Idle CPU time reaches 70%, and redundant context switches occur

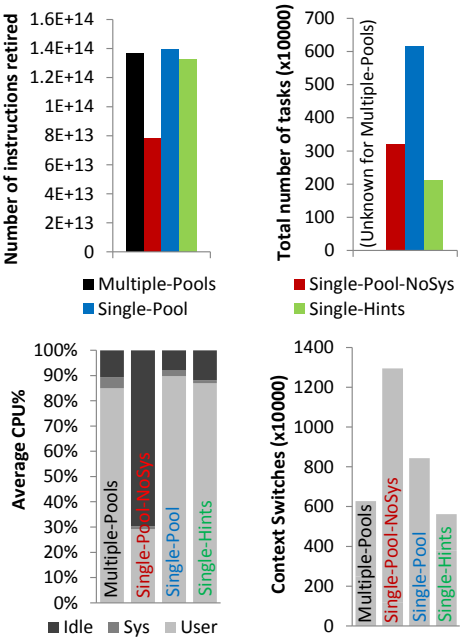

Figure 8: Measurements for 64 TPC-H clients.

similar to Section 7.1. This is due to the fact that in TPC-C, many clients contend for modifying common data. A lot of workers are blocked, and the OS scheduler cannot overlap them with other tasks. The flexible concurrency level of Single-Pool corrects this behaviour, processes more tasks, and can achieve similar or better performance than Multiple-Pools. This also shows that our light-weight implementation does not hurt the performance of short-lived transactions. It also reduces the number of context switches in comparison to Single-Pool-NoSys, but their number is still higher than that of Multiple-Pools, as the additional worker threads created due to inactive tasks are more than the threads of Multiple-Pools, due to the flexible concurrency level.

The concurrency hint of Single-Hints results in a significant reduction of the total number of TPC-H tasks, giving room to TPC-C tasks to be queued up faster in the scheduler. For 64 concurrent TPC-H clients, Single-Hints improves TPC-H throughput by 12.5% and TPC-C throughput by 370% in comparison to Multiple-Pools, with approximately the same number of instructions (since the system is fully loaded for all variations for the majority of the duration of the experiment). Note that hints alone do not affect TPC-C queries, as they are executed in a single task.

8. CONCLUSIONS

We show how to efficiently employ task scheduling for general-purpose tasks, without a central mechanism for data parallelism, in order to handle both short-lived transactions and long-running analytical tasks in highly concurrent workloads in a main-memory DBMS. Our results are backed up by an evaluation on a commercial DBMS: SAP HANA.

As transactional tasks can heavily use synchronization primitives, we show that the concurrency level of the task scheduler should not be fixed, but be flexible. When worker threads block, more worker threads should be issued, giving control to the OS scheduler of additional tasks to saturate CPU resources. Furthermore, for analytical partitionable operations, we observe that task granularity significantly affects communication, synchronization, and scheduling costs in cases of high concurrency. For this reason, our scheduler gives a hint to the task creators of partitionable operations, reflecting the level of CPU contention. Using this hint, partitionable operations re-adjust their task granularity, to avoid excessive communication, synchronization, and scheduling costs for high concurrency, and avoid under-utilization of CPU resources in cases of low concurrency.

9. REFERENCES

[1] Intel articles - Granularity and Parallel Performance. http://software.intel.com/en-us/articles/granularity-and-parallel-performance.

[2] Intel Thread Building Blocks – Documentation – User Guide – The Task Scheduler – Task-based Programming. http://threadingbuildingblocks.org/documentation.

[3] OpenMP API for Parallel Programming. http://www.openmp.org/.

[4] TPC-C Benchmark: Standard Specification, v. 5.11. http://www.tpc.org/tpcc/.

[5] TPC-H Benchmark: Standard Specification, v. 2.14.3. http://www.tpc.org/tpch/.

[6] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proc. of the 12th*

annual ACM Symposium on Parallel Algorithms and Architectures, pages 1–12, 2000.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. of the 25th Int'l Conf. on Very Large Data Bases*, pages 266–277, 1999.

[8] I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Scaling up analytical queries with column-stores. In *Proc. of the 6th Int'l Workshop on Testing Database Systems*, 2013.

[9] S.-l. Au and S. P. Dandamudi. The Impact of Program Structure on the Performance of Scheduling Policies in Multiprocessor Systems. *Int'l Journal of Computers and Their Applications*, 3(1):17–30, 1996.

[10] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A Proposal for Task Parallelism in OpenMP. In *Proc. of the 3rd Int'l Workshop on OpenMP: a Practical Programming Model for the Multi-Core Era*, pages 1–12, 2008.

[11] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for NUMA-aware contention management on multicore systems. In *Proc. of the 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 557–558, 2010.

[12] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proc. of the 2011 ACM SIGMOD Int'l Conf. on Management of Data*, pages 37–48.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 207–216, 1995.

[14] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *Proc. of the 17th annual Int'l Symposium on Computer Architecture*, pages 239–248, 1990.

[15] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *Proc. of the 4th Int'l Workshop on Data Management on New Hardware*, pages 25–34, 2008.

[16] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *Proc. of the 4th Int'l Conf. on OpenMP in a New Era of Parallelism*, pages 100–110, 2008.

[17] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[18] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[19] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 4:287–326, 1979.

[20] B. Hamidzadeh and D. Lilja. Dynamic scheduling strategies for shared-memory multiprocessors. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems*, pages 208–215, 1996.

[21] R. Hoffmann, M. Korch, and T. Rauber. Performance Evaluation of Task Pools Based on Hardware Synchronization. In *Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*, 2004.

[22] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *Proc. of the 15th edition of Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2010.

[23] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[24] J. Lee, Y. S. Kwon, F. Färber, M. Muehle, C. Lee, C. Bensberg, J. Y. Lee, A. Lee, and W. Lehner. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *IEEE 29th Int'l Conf. on Data Engineering*, pages 1165–1173, 2013.

[25] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007.

[26] H.-W. Loidl and K. Hammond. On the granularity of divide-and-conquer parallelism. In *Proc. of the 1995 Int'l Conf. on Functional Programming*, pages 135–144, 1995.

[27] S. Mattheis, T. Schuele, A. Raabe, T. Henties, and U. Gleim. Work stealing strategies for parallel stream processing in soft real-time systems. In *Proc. of the 25th Int'l Conf. on Architecture of Computing Systems*, pages 172–183, 2012.

[28] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proc. of the 1990 ACM Conf. on LISP and Functional Programming*, pages 185–197, 1990.

[29] A. Morf. Snapshot Isolation in Distributed Column-Stores. Master's thesis, ETH Zurich, 2011.

[30] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. In *Proc. of the 4th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 422–431, 1993.

[31] G. J. Narlikar. Scheduling Threads for Low Space Requirement and Good Locality. In *Proc. of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 83–95, 1999.

[32] D. Neill and A. Wierman. On the Benefits of Work Stealing in Shared-Memory Multiprocessors, 2011. Project Report at Carnegie Mellon University. `http://www.cs.cmu.edu/~acw/15740/paper.pdf`.

[33] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 155–174, 1996.

[34] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins. OpenMP task scheduling strategies for multicore NUMA systems. *Int'l Journal of High Performance Computing Applications*, 26(2):110–124, 2012.

[35] M. A. Palis, J.-C. Liou, and D. S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, 1996.

[36] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data*, pages 731–742, 2012.

[37] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proc. of the 21st Int'l Conf. on Data Engineering*, pages 2–11, 2005.

[38] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1):4:1–4:28, 2012.

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