A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores

Kayhan Dursun
Brown University
kayhan@cs.brown.edu

Carsten Binnig
TU Darmstadt
carsten.binnig@cs.tu-darmstadt.de

Ugur Cetintemel
Brown University
ugur@cs.brown.edu

Garret Swart
Oracle Corporation
garret.swart@oracle.com

Weiwei Gong
Oracle Corporation
weiwei.gong@oracle.com

ABSTRACT

Currently, we face the next major shift in processor designs that arose from the physical limitations known as the "dark silicon effect". Due to thermal limitations and shrinking transistor sizes, multi-core scaling is coming to an end. A major new direction that hardware vendors are currently investigating involves specialized and energy-efficient hardware accelerators (e.g., ASICs) placed on the same die as the normal CPU cores.

In this paper, we present a novel query processing engine called *SiliconDB* that targets such heterogeneous processor environments. We leverage the Sparc M7 platform to develop and test our ideas. Based on the SSB benchmarks, as well as other micro benchmarks, we compare the efficiency of *SiliconDB* with existing execution strategies that make use of co-processors (e.g., FPGAs, GPUs) and demonstrate speed-up improvements of up to 2×.

PVLDB Reference Format:
Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, Garret Swart, Weiwei Gong. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Motivation: Within the last decade, databases have undergone a major shift in designs which is mainly caused by two hardware trends: (1) Increases in main-memory capacities made it possible to hold even large databases in RAM, thus eliminating the I/O bottleneck when accessing data on secondary storage (such as hard disks). However, this also required databases to reconsider some fundamental decisions, such as how data should be laid out for efficient access by leveraging the upper levels (i.e., caches) of the memory hierarchy. (2) Also, processor designers started to exploit Moore's Law from a different perspective by increasing the number of cores, rather than focusing on single-core performance by improving clock-frequencies.

The advent of multi-core and multi-socket processor machines has led to a multitude of parallelization strategies in databases [1, 2], including new query scheduling paradigms such as morsel-driven execution [13] and also other optimizations to best leverage non-uniform memory access (NUMA) architectures [1, 2, 18].

Currently, we face the next major shift in processor designs which originates from physical limitations known as the *dark silicon* effect [9]. Due to thermal limitations and shrinking transistor sizes, the multi-core scaling is coming to an end, since not all the cores in a processor can be powered up at the same time. A major direction that hardware vendors are therefore investigating to tackle this problem is to place specialized hardware accelerators, that are less power hungry (e.g., implemented as ASICs), on the same die together with normal CPU cores and thus allow all processing units to be powered up at the same time. One example of such a heterogeneous multi-core environment is the Sparc M7 processor [15], which combines normal CPU cores with an ASIC that implements a Data Analytics Accelerator (called DAX) for typical main memory database operations. Another example is the Intel HARP platform [12], which combines FPGAs with normal CPU cores. We believe that in the near future there will be many more of these specialized system-on-a-chip (SoC) designs that follow the same paradigm of combining heterogeneous cores.

However, developing efficient data management systems for these emerging heterogeneous multi-core processors demands a critical rethinking of the architectural design and processing assumptions. In this paper, we take a first step and investigate how parallel query execution strategies should be designed for heterogeneous environments that combine normal cores with specialized ASIC-based accelerators.

One major challenge is that the state-of-the art parallel query execution strategies, such as morsel-based execution, have been developed for homogeneous multi-cores and are not optimal in heterogeneous settings. One difference is that the specialized hardware accelerators often provide only a *limited set of functions* at runtime and thus can not be viewed as yet another general-purpose core in the architecture. For example, the DAX engine in the Sparc M7 processor implements only two database operations (selection

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

and semi-join). A second major difference is that specialized accelerators typically expose themselves as *passive units* that cannot actively make work requests when they become idle.

As a result, existing query execution strategies for homogeneous multi-cores, are not directly applicable for modern heterogeneous multi-core systems. Furthermore, query execution strategies that leverage accelerators [3, 6] are not optimally suited for heterogeneous multi-cores as well, since they assume that these accelerator units are located at the end of the PCI bus and not tightly integrated with general-purpose cores. For such co-processor environments, databases typically assume that data transfers between the CPUs and co-processors are a major bottleneck due to slow interconnects. Therefore, existing solutions instead implement coarse-grained query processing strategies that offload big parts of the execution to the co-processors to minimize data transfer costs. Unlike co-processor environments with slow data transfers, heterogeneous multi-cores commonly allow the accelerators to access main memory via the same memory bus and sometimes even share the same lowest level caches (i.e., as in Sparc M7).

Contributions: We present a query processing engine called *SiliconDB* that implements a novel parallel query execution scheme for the emerging heterogeneous multi-core processors. Specifically, we make the following contributions: (1) We developed a new query execution scheme based on the notion of morsels but adapted for heterogeneous multi-cores that addresses the problem that specialized hardware accelerators provide only a *limited set of functions.* (2) In order to integrate passive DAX engines, we developed an adaptive push-based scheduling strategy that leverages a queueing-theory based cost model to effectively serve *passive units* with work elements in the morsel-driven scheme. The main challenge here is to identify an optimal number of work elements to be pushed to the accelerators to fully utilize both the accelerators and the regular cores. (3) We additionally propose novel query optimization techniques to address cases where a classical query optimizer would produce a plan that would result in an under-utilization of accelerators. We show that there are cases where a more expensive plan, but one that better leverages all available resources, would yield lower total query execution times.

In this work, we use the Sparc M7 processor as our development and testing platform. However, we believe that the architecture of *SiliconDB* and the underlying approaches are general and can be used to support other instantiations of heterogeneous multi-core processors as well. Explicitly demonstrating this is a line of future work.

Outline: The rest of this paper is structured as follows: In Section 2, we discuss the architecture of *SiliconDB*. In Section 3, we present the details of our query execution strategy for heterogeneous multi-cores based on the Sparc M7 processor. Afterwards, in Section 4, we discuss the details of our new scheduling model that efficiently utilizes all available compute resources; i.e., regular cores and DAX engines in our case. In Section 6, we then present the results of our experimental evaluation based on the SSB benchmark and other micro-benchmarks. Then, we conclude the paper in Section 9. Also, while we refer to related work throughout this paper, we provide a detailed discussion in Appendix 7.

2. SILICONDB OVERVIEW

In this section, we provide an overview of *SiliconDB*, an execution engine designed to address the challenges of emerging heterogeneous multi-core environments. We use the Sparc M7 as our target platform to evaluate our ideas, and assume a columnar layout, though our ideas should be readily adaptable to other similar platforms (such as Intel HARP) and storage layouts.

2.1 System Architecture

Emerging heterogeneous multi-core platforms commonly host general-purpose CPU-cores and specialized accelerators on the same die, where all these processing units typically have access to the same memory regions in a NUMA fashion and they even share the same last-level caches, as in the Sparc M7 environment. In order to attribute to NUMA-awareness and to leverage the effects of cache locality, a key idea behind *SiliconDB*'s architecture is the grouping of cores and accelerators into the so-called processing units (PUs), where each processing unit implements a morsel-based parallel execution scheme (as depicted in Fig. 1).

In a traditional morsel-based parallel execution scheme, cores that reside within the same processing unit (i.e., that have access to the same NUMA region) have access to one shared work queue. During query processing, all cores actively pull work elements from those queues. Unlike this model, *SiliconDB* needs to deal with two major challenges. First, the accelerators of a processing unit cannot process arbitrary work elements, since they may be functionally restricted and can support only a certain set of operations. For example, in the case of Sparc M7, the DAX (Data Analytics Accelerators) engines are designed to execute only scans and semi-joins over in-memory columnar data. Second, accelerators cannot actively pull work from the queues because they are implemented as passive units.

To address the first challenge, our query compiler splits a pipelined query plan (such as the one shown in Figure 2) into so called sub-pipelines where each sub-pipeline is annotated with information that denotes whether it can be executed by a DAX engine or not. Furthermore, each processing unit in *SiliconDB* comes with a set of so called "function-specific work queues" as well as a core-only work queue, instead of using only one shared work queue per processing unit. Function-specific queues are those for the work elements that are supported by accelerators.

After query compilation, *SiliconDB* first fills the work queues with work elements and then initiates query processing. A work element in *SiliconDB* refers to a small block of data that needs to be processed (i.e., a fixed number of values of a column in our case). While the work elements in function-specific queues can be processed either by a general-purpose core or an accelerator (e.g., when we have a scan or a join queue in case of Sparc M7), the work elements in a core-only work queue can be processed only by general purpose cores. A major challenge for our query execution scheme is to schedule these work elements residing in different queues in such a way to ensure that all available compute resources are fully utilized.

To address the second challenge and serve the passive processing units, *SiliconDB* uses cores to actively push work elements to the accelerators. If done naively, cores would schedule individual work elements one at a time, which would be a direct implementation of a morsel-based execution for pas-

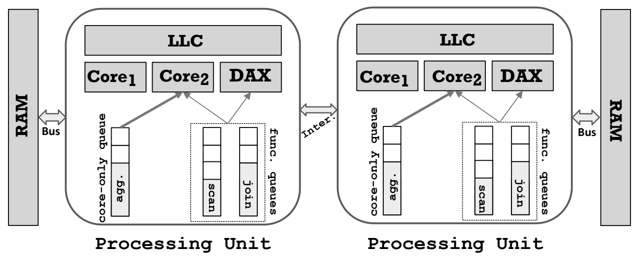
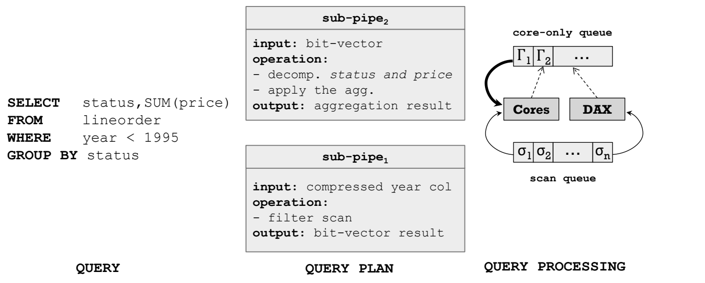


Figure 1: System Architecture



```
SELECT   status,SUM(price)
FROM     lineorder
WHERE    year < 1995
GROUP BY status
```

sub-pipe$_2$
input: bit-vector
operation:
- decomp. status and price
- apply the agg.
output: aggregation result

sub-pipe$_1$
input: compressed year col
operation:
- filter scan
output: bit-vector result

Figure 2: Query Processing

sive units. However, in such a naive scheme, the scheduling and synchronization overheads for individual work elements not only represent non-negligible processing costs but additionally cause the under-utilization the accelerators, where they often need to wait for the next item to be scheduled after completing the processing of the previous element. We therefore developed a new cost-based scheduling strategy that leverages queuing theory to decide how many work elements should be pushed to an accelerator at one time. The main challenge here is that, when too many work elements are pushed to an accelerator, the cores might become idle towards the end of the query execution, which would result in a sub-optimal query runtime.

2.2 Query Processing Example

In order to intuitively show how *SiliconDB* works, we discuss the execution using a simple example. Figure 2 depicts the lifecycle of a SQL query consisting of a scan and aggregation operator. As we described before, the resource-aware query compiler first splits the query plan into two separate sub-pipelines. In our example, the first sub-pipeline of the plan represents a scan on a column `year`, which produces a bit-vector for the selected rows. Afterwards, the subsequent sub-pipeline uses these bit-vector results to identify the selected values of the column `price` table and executes the aggregation operation on them.

In this example, we assume that the query is executed in one processing unit of *SiliconDB* composed of only one core and one accelerator (as shown on the right hand side of Figure 2). For morsel-driven fine-grained parallel query processing, the processing unit uses one function-specific queue (the scan queue) to keep the work elements of the scan operator and one core-only queue to hold the aggregation work-elements that can only be processed by cores. For execution, *SiliconDB* first populates the scan-queue. Following our fine-grained operator processing model, *SiliconDB* initially splits the scan operator into multiple work-elements and places them into the scan queue to start the processing.

Afterwards, *SiliconDB* triggers several execution threads that actively start pulling work from the scan queue: one worker thread for the normal cores and one thread that is a passive-unit handler to serve the accelerator. The worker thread and the passive-unit handler are initially pinned to physical CPU-cores while they can switch their handling roles at any point of time. For execution, the core-handler as well as the passive-unit handler thread pull work elements from the scan queue and executes these work elements. After finishing one work element, all threads (core or passive unit handler) create a new aggregation work element that consumes the result of the scan. Once the first elements become available in the core-only queue, the core handler thread starts pulling these new elements from the core-only queue instead of working on scan work elements any further. This not only helps us to better utilize all resources but also pipeline the results between two sub-pipelines and thus leverage data locality if intermediate results are still in the cache.

3. QUERY PROCESSING

In this section, we discuss the details of *SiliconDB*'s morsel-driven query processing engine and explain all the steps from query compilation to query execution including different scheduling strategies. In the end, we describe some optimizations we implemented to further improve query processing performance of *SiliconDB*.

3.1 Query Compilation

In the original morsel-driven query processing scheme [13], an incoming query is first compiled into several execution pipelines [14], where each pipeline processes small chunks of the input at-a-time, named as morsels. This morsel-based execution scheme lets the system to parallelize execution across multiple cores in an efficient way.

While *SiliconDB* compiles query plans into execution pipelines in a similar fashion, it additionally decomposes them into several *sub-pipelines* in order to address the different functionality characteristics of compute units (cores and accelerators) residing in the emerging multi-core environments. For instance, the specialized accelerators of the Sparc M7 processor (DAX engines) are implemented as ASICs to support only a limited set of operators, thus cannot be used to execute full pipelines. Before we describe how we split pipelines into sub-pipelines in *SiliconDB*, we first describe the functions that a DAX engine can support for query processing.

In the version we used for this paper, the DAX engines support the following three database-specific functions:

(1) Scan function: This function scans an in-memory array (e.g., representing an input column of a table) and applies a filter on its elements in order to detect the ones that match a given constant or satisfy a range predicate (i.e., greater than/smaller than). The function returns a bit vector with bits set to one representing the elements of the input column that match the condition applied.

(2) Select function: The select function also takes an in-memory array as input and selects elements from that array using a given bit vector that defines the selected entries.

(3) Translate function: This function can be used to execute a semi-join of two in-memory arrays. We will describe

more details about this function later in the paper.

In regards to these functionality characteristics, *SiliconDB* separates the functions of a given pipeline that can be executed by DAX engines and those that they cannot support into different sub-pipelines. The compiler tags the former ones as being DAX-executable and the latter being core-only ones. For instance, for the example query in Figure 2 the compiler generates two sub-pipelines: Here sub-pipe_1 represents a DAX-executable operation that scans the `year` column of the input table and applies a filter, whereas sub-pipe_2 depicts the core-only executable operation of the query which applies an aggregation function on the filtered tuples of the base column.

Here it is important to note that *SiliconDB* is not restricted to execute DAX-executable sub-pipelines only on DAX engines, but also has the option to use general-purpose cores for this purpose if necessary. The decision of what type of compute resource (cores or DAX engines) to use for a DAX-executable operation is given by our query execution scheme and it depends on the state of query processing at-a-given point of time as we discuss in the following subsection. We will also describe how processing DAX-executable sub-pipelines in this way instead of greedily pushing them only to DAX engines helps *SiliconDB* to provide better load balancing of query execution.

3.2 Query Execution

The query execution engine of *SiliconDB* is comprised of the so called *processing units* as shown in Figure 1. Each processing unit defines a logical processing component of *SiliconDB*'s processing model, where they each consist a group of compute resources (CPU cores and DAX engines) that have access to the same memory region. Accordingly, *SiliconDB* partitions tables of a database across processing units and stores these partitions in their local memory regions to enable the benefits of NUMA-awareness.

Furthermore, the execution engine defines multiple work queues per processing unit (i.e., per memory region) as the norm of query processing unlike a classical morsel-driven scheme that uses only one work queue per NUMA region. To this end, *SiliconDB* incorporates each processing unit with one *core-only queue* to hold work elements for *core-only sub-pipelines* that can be processed only by general purpose cores. It also defines a set of *function-specific queues* to contain work elements of *DAX-executable sub-pipelines* that can be executed by all resources (both DAX engines and CPU cores). In *SiliconDB*, we provide one function-specific queue for each DAX-executable function (e.g., a scan, a select, and a join queue).

In order to process the work elements residing in these work-queues, *SiliconDB* pins one worker thread to each available strand of a core and also deploys a DAX handler thread inside each processing unit. It is important to note that strands in a Sparc processor are similar to hyper-threads in Intel-based platforms, but they typically offer more parallelism. While all these execution threads can actively pull work elements from corresponding work queues, only the worker threads are defined to process them directly, whereas the DAX Handler threads are required to push them into the passive DAX engines and trigger their executions asynchronously on them. DAX Handlers also need to monitor the completion of each work element before scheduling the next element on the corresponding DAX engines.

In the following section, we describe how *SiliconDB* implements the scheduling of work elements inside a processing unit using these worker and DAX handler threads.

3.3 Work Scheduling

The main goal of *SiliconDB*'s query processing framework is to effectively utilize all compute resources (cores and DAX engines) and minimize the time frames that any of these units would need to stay idle. In the following, we explain the details of the scheduling policies that the worker and DAX-handler threads follow towards satisfying this goal.

Worker Threads. As we pointed out in previous section, *SiliconDB* defines worker threads to actively pull and process work elements from core-only queues, as in a classical morsel-driven model, however with the possibility of accessing function-specific queues as well that contain DAX-executable work elements.

In regards to optimal resource utilization, here the main challenge is to decide on the order of work queues that the worker threads should be pulling the work elements from. To this end, each processing unit defines a priority map that drives its worker threads to pull work-elements from specific work queues at a specific state of query processing. More specifically, the priority mapping defines that a worker thread should first try to pull work elements from the core-only queue. In case no work elements are available in the core-only queue, the worker threads then start pulling work elements from the other function queues and work cooperatively with DAX engines on the same sub-pipelines (instead of staying idle). Algorithm 1 summarizes the general process that the worker threads follow to implement this scheduling scheme.

Algorithm 1: Query Processing in *SiliconDB*

```
   Input: List of Work Queues $Q$ ordered by Priority Mapping
1  Algorithm processQuery($Q$):
2      while $Q$.allProcessed() == false do
3          cur_queue ← $Q$.GetNonEmptyQueue() ;
4          if cur_queue.hasNextElement() then
5              work_element ← cur_queue.nextElement() ;
6              processWorkElement(work_element) ;
7              scheduleFollowupWork(sub_pipe, work_element) ;
8          end
9      end
10 end
11 def scheduleFollowupWork(sub_pipe, work_element):
12     if sub_pipe.hasFollowing() then
13         fw_pipe ← sub_pipe.getFollowing() ;
14         fw_queue ← fw_pipe.GetNonEmptyQueue() ;
15         fw_queue.addElement(work_element.getFollowing()) ;
16     end
17 end
```

The input to the query processing algorithm is an ordered-list of work queues that was generated using the priority-mapping. To this end, work elements are processed with respect to the orders of the queues they belong in this list. At the start of each iteration, the processing algorithm returns the first non-empty queue based on the priority-mapping that contains some work elements (line 3). Afterwards, the algorithm executes the sub-pipeline for the selected work element (line 4-8) and upon its completion, the algorithms calls a procedure to schedule follow-up work elements as encoded by the query plan (line 7). The idea of the handling procedure is that it triggers the creation of follow-up work elements for a subsequent sub-pipeline in the query plan if such as sub-pipeline exists (13-15). For example, in Figure 2, the execution of a work element of the lower sub-pipeline creates a work element for the upper sub-pipeline once the lower sub-pipeline finished its execution.

DAX Handler Threads. As we described in previous section, DAX handler threads work in a different way than worker threads in order to address the passive processing nature of DAX engines. Since they cannot actively request new work-elements, DAX handler threads are responsible to optimally schedule work elements on these accelerators. Additionally they need to observe the state of the execution of the previously scheduled elements and carefully decide when to schedule new ones.

In order to comply with the main goal of *SiliconDB*, one important challenge for DAX handlers is to keep DAX engines optimally utilized during query execution. To this end, *SiliconDB* implements DAX handlers to follow an adaptive push-based scheduling strategy, which we describe in Section 4 in greater detail.

3.4 Optimizations of Execution

We conclude this section by discussing some additional optimizations we implemented to further improve the performance of *SiliconDB*'s query execution engine.

Pipelining for Sub-Pipelines. As discussed previously, when being executed in *SiliconDB*, work elements materialize the output of a sub-pipelines into memory and push a follow-up work element that consumes the output into the corresponding work queue. However, pushing a follow-up work element into a queue might cause unnecessary additional overhead and prevent cache-locality. Therefore, we extend the worker threads to provide a fusion mode between work elements of different query sub-pipelines if applicable. For instance, if a DAX-executable work element is executed by a worker thread and triggers a follow-up work element (line 15 in Algorithm 1) that is core-only executable, the worker thread immediately processes this work element instead of placing it first into the core-only queue.

Furthermore, we also adapted the work element handling of DAX-handlers to provide better cache locality since DAX engines and normal cores share the same last level caches (LLC). The idea is that a DAX-handler can push work elements to the front of the work queues queues to maximize the chance that the output is still in the LLC. For instance, when the DAX-handler finishes the processing of a scan work element in Figure 2, the DAX-handler pushed the follow-up aggregation work element to process the output to the front of the core-only work queue. This way, the likelihood that the input for the aggregation work element is still present in the LLC when the work element is pulled by on one of the worker threads is maximized.

Work Stealing. As we described in Section 2, normal cores and DAX engines are grouped into the so-called processing units to attribute to the effects of NUMA-awareness and enable cache-locality when processing work elements. In addition to polling work elements from the local work queues, we allow all worker threads and the DAX-handler of a processing unit to additionally steal work elements of a remote processing unit if all work queues of the local processing unit are empty. This way, we can mitigate the chance that individual processing units remain idle while others are overloaded (e.g., due to data skew).

4. ASYNCHRONOUS SCHEDULING

In this section, we provide the details of the scheduling model of *SiliconDB* for DAX handler threads and describe how we incorporate it with the rest of the framework in order to comply with the main goal of *SiliconDB* (i.e., optimal utilization of compute resources). To this end, we look into the problem from two perspectives:

(1) Implementing an optimal handling strategy so that system resources are not sacrificed just for observing the DAX engines (cf. Section 4.1).

(2) Providing an adaptive scheduling model that effectively utilizes the accelerators during query processing (cf. Section 4.2 and cf. Section 4.3).

4.1 Handling Model

In *SiliconDB*, we implement two different strategies to handle the scheduling of work-elements on DAX engines:

(1)*Separate Handler:* In each processing unit, one core is reserved to pull work elements from function-specific queues that hold DAX-executable work elements and assign them to a DAX engine. Furthermore, alternating with scheduling work, the handler thread monitors the status of running work elements and potentially creates a follow-up work element once another work element was finished.

(2)*Piggybacked:* In this model, there is no dedicated handler thread to assign work to the accelerators. Instead, all the worker threads in a processing unit share the responsibility of scheduling work for the DAX engines. The idea is that each worker thread checks whether or not the DAX engine finished processing a work element, and if a new work element should be scheduled to a DAX engine before polling a work elements for itself.

4.2 Scheduling Policy

DAX handlers can submit single or several work elements to a DAX engine at a time, which first places them into an internal hardware queue and then assign these elements to their hardware execution pipelines for processing. It is important to note that the Sparc M7 processor implements each DAX engine to have four of these pipelines letting them to process multiple work elements simultaneously. Therefore, the actual processing of work elements on the pipelines is handled by some internal scheduling mechanisms, which cannot be directly controlled by DAX handlers.

For the overall query processing scheme, the main challenge is (1) to ensure an internal hardware state where all the execution pipelines are effectively utilized and (2) the number of work elements waiting for service in the internal hardware queues are at minimum levels. While each one of these goals can be satisfied independently by applying simple heuristics, an optimal solution requires more sophisticated schemes to find a sweet-spot between these two important metrics.

For instance, the system could make sure that the utilization is always high by submitting a large number of work elements in each round but with the risk of having them to pile-up in the internal queues. This not only would increase the average execution time of work elements in accelerators, but would also hurt the overall query processing performance since DAX engines might become stragglers and dominate the overall query runtime.

Similarly, the size of the internal queues can be reduced by having the system to initially submit as many work elements as the number of execution pipelines and then to supply new ones one at a time for each completed work element. However, it is clear that there is a risk for DAX engines to become underutilized with this approach since DAX handlers first need to detect that a work element is completed before they can submit a new one to a DAX engine. Moreover, there is some internal overhead associated with scheduling of work elements on DAX engines which would increase the latency for each element as a result.

Therefore, the main challenge of query scheduling over DAX engines is to find an optimal number of work elements that the DAX handlers should submit to these accelerators in every iteration. For the ease of representation, we describe our ideas around the model where *SiliconDB* uses a separate thread for DAX Handlers. However, the ideas are directly applicable for the *piggybacked* case as well.

The main idea behind *SiliconDB*'s scheduling policy for the passive hardware accelerators is to define the value of an internal scheduling parameter called the *q-size*, which refers to the maximum number of work elements that can reside in the internal hardware queues of these accelerators. Accordingly, in every iteration the DAX handler thread monitors how many work elements are completed and uses the value of *q-size* to decide the number of new work elements that should be submitted to the corresponding DAX engines. Next, we describe the procedure that *SiliconDB* follows in order to assign the value of this important *q-size* parameter.

4.3 Optimal Queue Size

Before providing any further details, it is important to note that *SiliconDB* needs to adjust the value of *q-size* periodically, since the DAX engines will be processing work elements with different characteristics during query execution (i.e., different job-types or data characteristics). Moreover, the performance of DAX engines are sensitive to other runtime effects, such as the contention on the memory bus that is shared between multiple execution threads. Therefore, *SiliconDB* implements the DAX handler threads to follow an adaptive scheduling model where they continuously observe the DAX engines to collect statistics during query processing. Then they use these observations in order to adjust scheduling decisions accordingly (i.e., updating the value of *q-size*) by leveraging a cost model that we define.

Runtime Statistics: Since the internal state of DAX engines are not exposed to the operating system, we make use of an analytical model to get estimations specifically on the two important metrics that we described in Section 4.2. In the rest of this section and in our cost models, we refer to these parameters as *utilization* and *items-waiting* to represent the utilization of the internal execution pipelines and the number of work elements that is waiting for service in the internal hardware queues. To be able to estimate these metrics for each DAX engine, we leverage a queueing model with limited capacity defined with the *(M/M/s/k)* mathematical notation.

The parameters of a typical queueing model consists of the system's arrival (M) and service (M) rates, the number of units that can serve the system simultaneously (s), and the maximal queue length (k) that is allowed, all respective to their orders in the notation. Then with these parameters in place, a queueing model can provide estimations about the internal state of a corresponding system. As any queue-based system can leverage queueing theory to get estimates about its performance and then uses them to improve service, in *SiliconDB* we leverage a queueing model to estimate the aforementioned *utilization* and *items-waiting* metrics and use them in a cost model to improve the scheduling scheme that the DAX Handlers follow.

In this regard, *SiliconDB* actively collects statistics about query processing such as how often DAX handlers submit work elements (to derive the arrival-rate) and how long it takes DAX engines to process these elements on average (to derive the service-rate). Then it provides these statistics as the input of the queueing-model along with two parameters representing the number of execution pipelines in each DAX engine (server-size) and the current *q-size* defined by the DAX handler.

Now in the following, we describe the details of our cost-model that the DAX handler threads use to find optimal values for the *q-size* that is based on the *utilization* and *items-waiting* estimations as provided by the queueing model and some throughput statistics about query performance.

Cost Model: As we mentioned before, *SiliconDB* leverages a cost model that provides a runtime estimation value for executing a sub-pipeline that consists of a set of work elements. Then the asynchronous scheduling model of DAX handler threads uses this cost model to pick a *q-size* that gives the minimum runtime.

We adapt the parameter *q-size* at query runtime such that the overall processing time needed for all work elements of a sub-pipeline is minimized. The cost model is then applied in regular time windows W to adjust *q-size*.

The following equations show how the runtime is estimated based on the estimations of *utilization* and *items-waiting* for the DAX engines. For our cost model, we assume that cores and DAX engines can cooperatively work on work elements of a sub-pipeline and that sub-pipelines need to materialize their output before the next sub-pipeline starts.

$$\mathsf{tput}_{\mathsf{overall}} = \mathsf{tput}_{\mathsf{cores}} + \mathsf{tput}_{\mathsf{dax}} \cdot \frac{\mathsf{utilization}^{\mathsf{W+1}}}{\mathsf{utilization}^{\mathsf{W}}}$$

$$\mathsf{cost}_{\mathsf{runtime}} = \frac{\mathsf{N}}{\mathsf{tput}_{\mathsf{overall}}} + \frac{\mathsf{tput}_{\mathsf{dax}}}{\mathsf{items-waiting}}$$

As depicted with the equations above, the main components of the cost model are the $tput_{cores}$ and $tput_{dax}$ parameters, which represent the throughput that cores and DAX engines produce for an observation window, W. While the *utilization* and *items-waiting* parameters are provided by the queueing model as we described above, the scheduling model monitors the throughput parameters at query runtime and then calculates an overall runtime estimation for the remaining N work elements.

The main idea of the cost model is that the *q-size* affects the *utilization* of DAX engines and thus their throughput as well while the throughput of the cores remain stable. The change (increase or decrease) in throughput is given by the ratio between $utilization^{W+1}$ and $utilization^{W}$, which defines the estimated throughput of the estimated utilization for window $W + 1$, after changing *q-size* over the estimated utilization of the current window W.

Therefore, the cost model allows us to estimate the overall throughput $tput_{overall}$ for the next window $W + 1$ that results from choosing a new *q-size*. The throughput can then be used to compute an estimate for the total runtime for processing the remaining N work elements for a given sub-pipeline. The runtime estimate is thus given by $(N/tput_{overall})$ plus the additional time it requires to process the already scheduled work elements (*items-waiting*) that would reside at the internal hardware queues of the DAX engines as estimated by the queuing model. Algorithm 2 shows the overall procedure that *SiliconDB*'s adaptive scheduling model follows to find the optimal *q-size* by leveraging the cost model we just described.

Algorithm 2: Finding the optimal queue size

```
   Input  : Set of work elements Processed in the Previous
            Window, $W$
   Input  : Submission and Completion TimeStamps of work
            elements Processed by Accelerators , $TS$
   Output: The New Optimal Queue Size, $q\_size$
1  Algorithm adjustQueueSize($W$, $TS$):
2  $a\_rate \leftarrow$ calculateArrivalRate($TS$) ;
3  $s\_rate \leftarrow$ calculateServiceRate($TS$) ;
4  $tput \leftarrow$ calculateThroughputRates($W$) ;
5  foreach $newQSize \in possibleQSizes$ do
6      ($utilization$, $items\text{-}waiting$) $\leftarrow$
        QueueingModel($a\_rate$, $s\_rate$, $newQSize$);
7      $curRuntime \leftarrow$ cost($tput_{overall}$, $utilization$, $items\text{-}waiting$);
8      if $curRuntime \leq minRuntime$ then
9          $minRuntime = curRuntime$;
10         $q\_size = curQSize$;
11     end
12 end
13 return $q\_size$;
```

In order the find the most optimal *q-size* for the current state, the scheduler applies a linear search over the possible values of it, uses the estimation models as described and picks the *q-size* which results in the minimal estimated runtime. It is important to note that linear search is possible since the search space is sufficiently small. We simply apply a neighbor search starting with the current *q-size* and increment/decrement its value linearly.

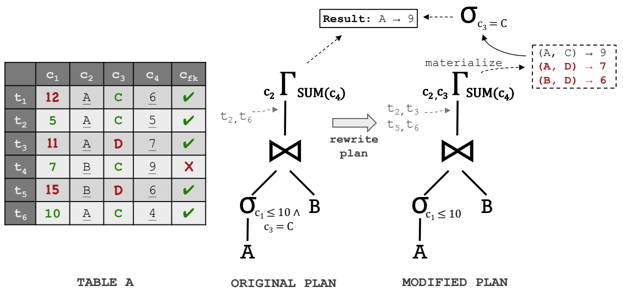
In Algorithm 2, we summarize the steps the scheduler follows towards finding the most optimal *q-size*:

(1) For each observation window it first calculates the average arrival and service rates (line 2-3) for the work elements based on some runtime statistics and also the estimated throughput rates of cores and DAX engines (line 4).

(2) Then for each possible *q-size*, it first uses the queuing-model to estimate the *utilization* and *items-waiting* parameters using the observed arrival and service rates (line 6),

(3) and then using the estimated *utilization* and *items-waiting* values along with the observed throughput of compute units, it calculates a runtime value for the execution of work elements using the cost model we described before (line 7).

In the end, it returns the *q-size* value that produced the minimal estimated runtime.



	c_1	c_2	c_3	c_4	c_{fk}
t_1	12	A	C	6	✓
t_2	5	A	C	5	✓
t_3	11	A	D	7	✓
t_4	7	B	C	9	✗
t_5	15	B	D	6	✓
t_6	10	A	C	4	✓

Figure 3: Additional Materialization

5. QUERY OPTIMIZER

We now explore the query optimization perspective and describe why existing query optimizers should be reconsidered for heterogeneous multi-cores. We also propose new query optimization techniques and demonstrate their potential in these new environments.

5.1 Overview

Since existing query optimizers generate query plans assuming a homogeneous processor environment, there are a variety of reasons why these plans may not be applicable or optimal on heterogeneous multi-cores. For instance, a query plan might suggest the use of the operators that are not supported by the accelerators, which would render them useless for processing this plan since they can not be utilized at all.

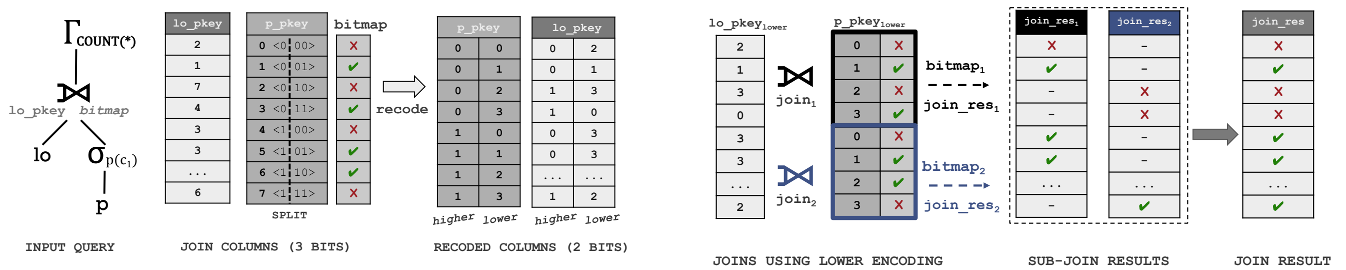
We thus suggest adapting the optimizer to take into account which compute resources can be leveraged to process the plan. While this is a non trivial problem on its own and would require a major change of the optimizer from the ground up, we look into this problem around the characteristics of the Sparc M7 processor with its DAX engines and propose simple heuristics that rewrite a query plan produced by a classical optimizer into one that better utilizes our processor. With such a rewrite, we show that we can reduce the overall runtime as demonstrated by our experiments.

5.2 Heuristic 1: Additional Materialization

First, we propose to insert extra scan operators at the end of a query plan in order to better utilize the DAX-Units. The query optimizer would normally regard this approach as an inefficient way of executing the query.

We motivate our heuristics using an example query that has an aggregation on top of a hash-join between two relations. Fig. 3 represents the original query plan that was produced by the query optimizer, which suggests applying all the filters first to the relation that probes into the hash-table. In order to execute this plan on the Sparc M7 environment, the query processor would initially be able to leverage the DAX engines, but then they would stay idle during the latter parts of execution after all the scans at the bottom of the query plan are completed.

To address this problem, i.e., to allow the system to leverage the DAX engines in the later stages of query processing, we propose to modify the query plan by inserting an extra scan operator at the end to filter some temporary results produced by the aggregation.



(a) Recoding Join Columns

(b) Executing Partitioned Joins

Figure 4: Operator Re-Coding

In the example, we illustrate this idea as completely pulling-up the second scan ($c_3 = C$) defined on the first relation and having the aggregation to first materialize its output results into a temporary buffer. Following this modification, we have the system leverage the DAX engines in order to filter out of these results (i.e., the second and third tuples in the temporary output) to produce the final query result.

To implement this type of an optimization, it is clear that the system should carefully examine if the benefits of applying it would outweigh the costs associated with it, and then provide a rewrite strategy accordingly. We show the promise of such a query optimization technique for our target platforms. We also provide the results of a micro-benchmark we implemented in this regard in Section 6.3.

5.3 Heuristic 2: Join Re-Coding

While DAX engines can support the functionality of a semi-join operator, they are designed to process inputs of at most 16-bit encodings. For FK-PK type of joins, this restriction requires the smaller PK relation to have at most 2^{16} keys, otherwise accelerators cannot be utilized during the execution of this operator.

We propose a technique that re-codes the inputs of the semi-join operator so that they are represented with smaller bit-encodings and adapt the query plan accordingly to have the system support queries that could not be executed by the DAX engines otherwise.

In Figure, 4 we depict an example to show how we implement this technique in *SiliconDB*. Here, the input query includes a semi-join between the *lineorder* and *part* tables of the *SSB-Benchmark*. The inputs to the operator include a bitmap result representing the selected tuples of the *part* relation and the bit-compressed *lo_pkey* foreign-key column of the *lineorder* table. Then, as to follow the implementation of a semi-join, the operator uses the values of *lo_pkey* to index into the bitmap vector and produces an output bit-result representing the tuples of the *lineorder* that satisfies the join condition. For the ease of representation, we assume that the DAX engines can support bitmap inputs represented with at most 2-bits, but the *p_pkey* column has 8 distinct values and thus requires 3-bit encoding.

Now our goal in this example is to represent the values of the *p_pkey* column with an encoding of 2-bits and process the join accordingly so that the system can leverage the DAX engines. As we show in Fig. 4a, the idea is to create a split point over the original bit-encodings and represent the original columns as with two sub-columns that contain the high- and low-end bits of the actual encoding in respect to the applied split point.

After the *p_pkey* and *lo_pkey* columns are re-coded according to this process, we adapt the query plan to process the join in two steps represented as $join_1$ and $join_2$ in Fig. 4b. Both of these sub-joins share the use of lo_pkey_{lower} sub-column as one side of their inputs. However, they are required to use different portions of the re-coded p_pkey_{lower} as their bitmap inputs ($bitmap_1$ and $bitmap_2$) in order to reflect it as a PK-column as required by the join operator (note the repeating pattern of the values).

After their execution, both $join_1$ and $join_2$ generate bit-vector results ($join\_res_1$ and $join\_res_2$), but with some false-positives due to the fact that they were required to index into different portions of the original bitmap column. In order to eliminate these false-positives and correctly generate the final join result ($join\_res$), here we use the re-coded lo_pkey_{higher} sub-column as the final step.

6. EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental evaluation of the techniques presented in this paper. The main goal of this evaluation is to: (1) compare the proposed fine-grained execution model based on morsels against two alternative techniques that are typically used for heterogeneous environments, (2) show the significance of the adaptive scheduling model, (3) demonstrate the promise of new query optimization ideas and (4) highlight the benefits of some other optimization techniques we discussed throughput the paper.

Setup: The prototype of *SiliconDB* is implemented in C++ and compiled using GCC 4.82. All experiments have been executed on a machine at Oracle with 128GB of RAM and one SPARC M7 processor (32 cores, 8 DAX engines) running Solaris 11.3 as the operating system. In the SPARC M7 processor, 4 cores and 1 DAX engine share one 8 MB L3 cache. Moreover, each of the cores supports 8, the so called strands (which are similar to hyper-threads) resulting in a total of 256 hardware threads for one server with 32 cores. In most of our experiments, we were limited to use only 8 cores and 2 DAX engines since we only had access to parts of a remote machine at Oracle. However, we were able to execute some experiments on the full machine and we indicate the setup we used for each experiment.

6.1 Exp1: Star Schema Benchmark

In this experiment, we present the benefits of *SiliconDB*'s query processing model and compare our execution scheme against two different execution models used in the past for heterogeneous environments based on CPUs and GPUs (sitting at the end of the PCI bus).

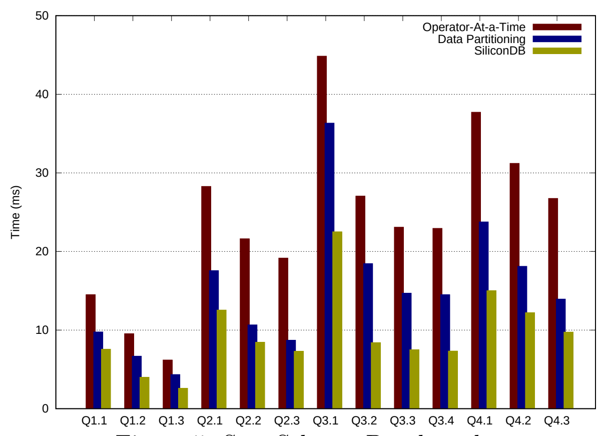


Figure 5: Star-Schema Benchmark

(1) Operator At-a-Time [4, 5]: In this scheme, the complete execution of each operator is pushed to either cores or accelerators, depending on which resource has better execution performance on the specific operator. This model represents a coarse-grained processing scheme, commonly seen in heterogeneous co-processor environments. It is also important to note that the processing model that the execution engine of an Oracle database follows in the Sparc M7 environment aligns closest with this scheme as it pushes all scans and semi-joins to the DAX Engines.

(2) Data Partitioning [10]: This model partitions the input of each operator between cores and accelerators before the processing of each operator starts. This model lets both cores and accelerators process some operators simultaneously, but in a blocking manner that avoids pipelining between operators completely. Moreover, the amount of work is statically assigned to cores/accelerators.

In order to provide fair comparisons, we implemented both of these schemes in *SiliconDB* as a different mode of processing. Both for (1) and (2), we use our compilation approach to generate plans with sub-pipelines and ensured that all the work elements of a specific operator are consumed before the execution of a parent operator is started. Regarding the scheduling of work elements for the *operator-at-a-time* technique, we created and pushed all the work elements of the scan and semi-join operators into the accelerators, while the rest of the query plan is executed on normal cores. For the second technique, we pre-assigned the work elements of the scan and semi-join operators between cores and accelerators depending on their expected idealized performance ratio for processing these operators.

In order to show the effectiveness of *SiliconDB*'s query processing model over these two techniques, we ran the queries from the complete SSB benchmark with a scale factor of SF = 10 and provide the total runtime results in Figure 5. In this experiment, we used a setup involving 8 cores and 2 DAX engines.

As expected, the *operator-at-a-time* technique shows the worst performance due to its coarse grained processing model, which causes cores to stay idle while accelerators are processing work elements or vice versa. The *data-partitioning* technique improves over the *operator-at-a-time* model because it can support the co-processing of operators between cores and accelerators, thus providing improved utilization. *SiliconDB* outperforms both of these techniques by up-to 3.2x improvements over the *operator-at-a-time* model and a 2.3x speed-up over the *data-partitioning* technique. This is mainly due to the *SiliconDB*'s dynamic query scheduling model, which can adapt to run-time conditions and provide better utilization of all compute-resources. In order to better point out this fact, we present the details of resource utilization during the execution of Query 1.1 in Figure 6. This clearly shows that our scheduling strategy better utilizes all available compute resources and thus minimizes the overall runtime.

6.2 Exp. 2: Adaptive Scheduling

In this experiment, we show the significance of *SiliconDB*'s adaptive scheduling model. Our main goal is to show that finding an optimal *q-size* is necessary for an optimal query processing scheme and the value of the *q-size* should be adjusted dynamically during query execution.

We first present the result of an experiment in Figure 7 showing that the optimal *q-size* depends on different characteristics of the query workload. Here, the y-axis presents the total runtime of a scan operation that applies a filter on a compressed column with a specific bit-encoding; i.e. each line represents a different case, $8-bit$ and $16-bit$ encoding respectively. We report the results for each possible *q-size* (from 1 to 16), and the results depict how these values effect the total runtime. In both cases, we see the effect of *utilization* and *items-waiting* metrics as we discussed in Section 4.3 and that the optimal *q-size* value occurs when the system finds a sweet-spot between them. Also even more importantly, our cost model (shown in Figure 7 as well) is able to pick the optimal *q-size* value in both cases. The reason why different *q-size* are important is due to the different types of scan items processed by the system.

6.3 Exp. 3: Query Rewrites

We now show the effects of our rewrite heuristics for query optimization techniques. All experiments in this section are executed again using 8 cores and 2 DAX engines.

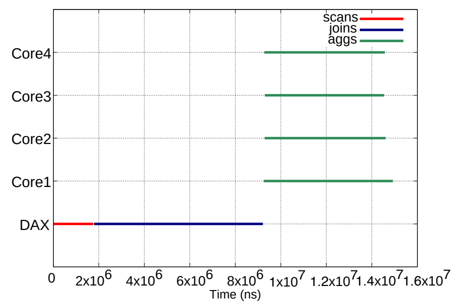
Exp. 3a - Join Re-Coding

In the first part, we executed a query which has a PK-FK join between the *lineorder* and *part* tables of the SSB benchmark. Implementing this operator as a semi-join requires to encode the smaller *part* relation's join-column with 18-bits of encoding as the bitmap of the semi-join operator. However, as we mentioned, the DAX-Units can support only bitmaps of up-to 16-bits, so we first execute the query with default settings which requires the joins to be handled by CPU-cores only.

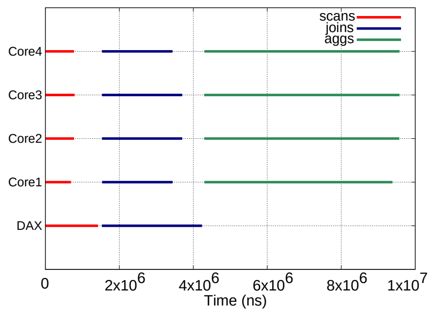
To that end, we applied our operator splitting approach, where we create four different small joins out of the single join operator and process the query in this manner. One disadvantage here is that the system now needs to process more joins instead of a single one, but the benefit is that we will be able to utilize more resources. The results of this rewrite are shown in Figure 8a in terms of their resource utilization and total run-time. We see that even if the recoding approach does more work in total, it is able to reduce the total run-time by utilizing more resources.

Exp. 3b - Inserting Materialization Operators

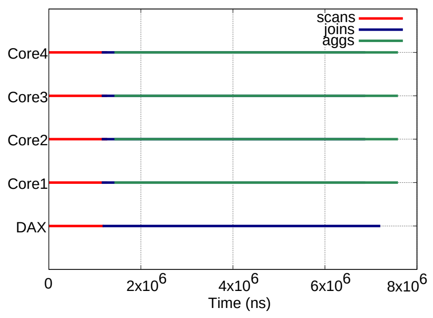
For the second experiment, we applied the rewrite which adds additional materialization operators to better leverage



(a) Operator At-a-Time



(b) Data Partitioning



(c) SiliconDB

Figure 6: Resource Utilizations During SSB Query 1.1

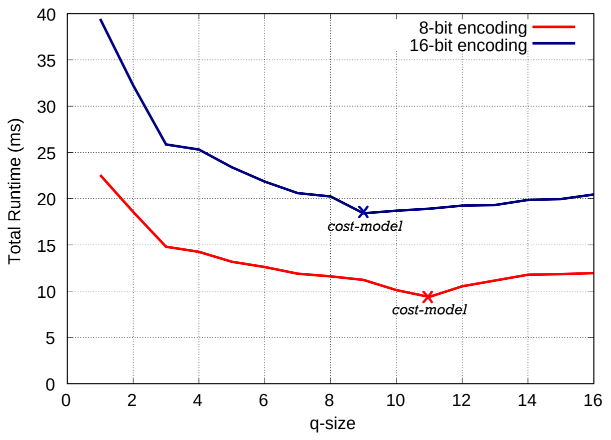
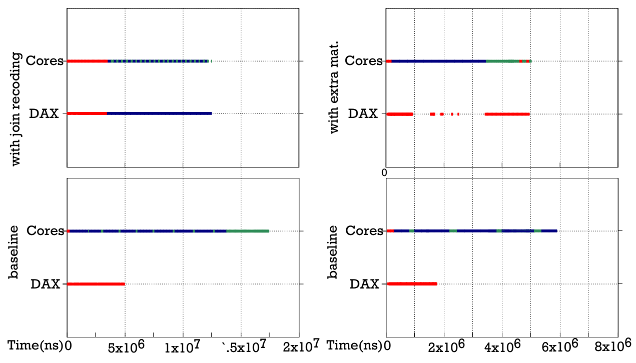


Figure 7: Optimal Queue Sizes



(a) Join Re-Coding (b) Extra Materialization

Figure 8: Rewriting Query Plans

DAX engines in the later phases of executing a query plan. To this end, we used the query from Figure 2 which has an aggregation on top of a scan, so the system would be able to use the DAX engines only at the start of the query. To show the effects of our suggested operator-insertion technique, we modified the query plan to have an additional having statement after the aggregation operator, so the system would first materialize some of the aggregation results and then need to apply another filtering to produce the final results.

In this experiment, we use a fixed ratio of 0.5 on how much of the output data the aggregation should materialize. In order to make more informed decisions, we would need an additional cost-model based decision model. In this experiment, our main goal is just to show the promise of query plan adaptation for better resource utilization. The results of this experiment are shown in Figure 8b. Again, we can see that the rewrite reduces the overall runtime and increases the utilization of the DAX engines in the later phases.

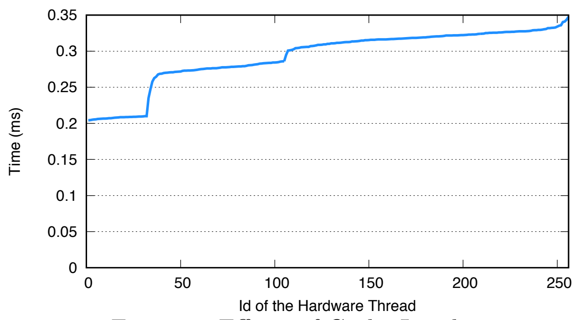


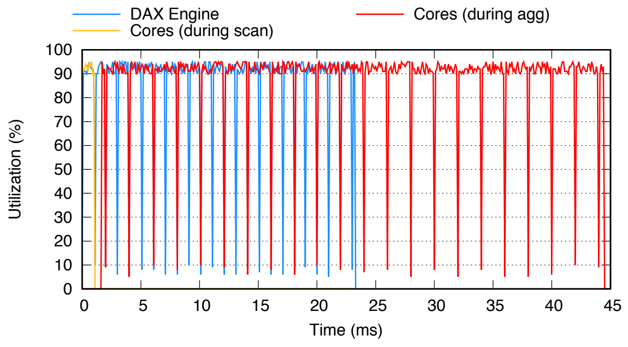
Figure 9: Effects of Cache-Locality

6.4 Exp. 4: Micro Benchmarks

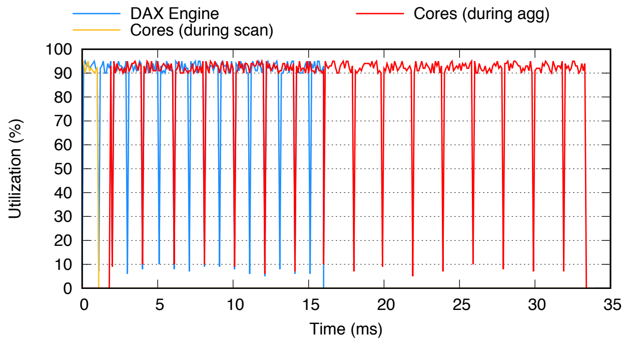
Exp. 4a - Effects of Cache-Locality

In order to reveal the effects of cache locality, we executed the query in Figure 2. For running the query, we used the full machine but selected only 1 DAX engine and 1 core as follows. For executing the scan sub-pipeline, we used 256 different configurations: in each of the configurations we used the same DAX engine but for the the aggregation we pinned the worker thread to a different dedicated strand — recall that the full machine has 32 cores and each core has 8 strands, resulting in a total of 256 strands. The output of the DAX scan was sized so that it fits in the L3 cache assigned to the DAX engine. We repeated the experiment for all available 256 hardware threads.

The results of the experiment are shown in Figure 9. The x-axis shows to which hardware thread we pinned the software thread and the y-axis represents the average runtime of the software thread. We observe that the runtime of the aggregation when the software thread is pinned to one of the first 32 strands is significantly lower. The reason is that these strands are executed by the first 4 cores that share the cache with the DAX engine. In our results, we can also see an additional increase in runtime when using the strands 128 to 256. The reason for this are NUMA effects that result from different latencies to access the two NUMA regions using an on-chip network on the SPARC M7 processor.



(a) DAX-Handler



(b) Piggybacked

Figure 10: Efficiency of Handling Strategies

Exp. 4b - Efficiency of the Handling Model

In this experiment, we show the benefits of our execution strategies discussed in Section 4.1 when using the two variants: Separate DAX Handler and Piggybacked. In order to show the utilization of cores/accelerator we used 4 cores and 1 DAX engine. The results of this experiment are shown in Figure 10. While the utilization of both schemes is similar, the overall runtime of the piggybacked version is about 25% lower since all four cores can be used to execute meaningful work instead of reserving one core as a DAX handler.

7. RELATED WORK

Query Processing on Co-Processors Environments: There exists a large body of work on how to leverage specialized co-processors for analytical database workloads (e.g., FPGAs [19], GPUs [3, 7], etc.). The main limitation of existing co-processors is the PCI bottleneck, thus the main goal of these solutions that integrate co-processors in an end-to-end manner into a DBMS has been to reduce the overall communication [3, 5]. However, even when using these schemes, the high speed-up rates reported for individual database operators often become negligible when considering the overall runtime that includes the communication costs between CPUs and the co-processors [6].

Some recent work has addressed more coupled environments where the cores and the co-processor units are integrated on a single chip [16, 17, 20, 11, 21]. For instance, in [16] and [17], to show the effects of using an integrated GPU, the authors propose specialized scan and aggregation operations. They were able achieve a 3× performance boost compared to an architecture where the GPU sits at the end of the PCI bus, even though the integrated GPU has 4× lower computational power than a discrete GPU. Different from our paper, these approaches focus on the integration of GPUs which can still execute arbitrary functions whereas *SiliconDB* can also efficiently integrate accelerators which provide a limited set of functions that brings new challenges as described in this paper. Furthermore, we believe that *SiliconDB* can also be efficiently used for coupled CPU-CPU platforms as well. Demonstrating this claim with implementation and experimentation is beyond the scope of this paper.

Similarly in [10], authors provide a co-processing scheme for hash-joins in a similar environment. Their solution splits the execution of a hash-join into four steps and have the CPU and the GPU units to co-execute each one of these steps in a fine-grained manner. Before starting each step, it uses a cost-model to find a ratio in order to split the execution between the CPU and GPU units. While this work shares the same high-level goal (i.e., utilizing all compute-resources effectively), our approach differs in various aspects. First, our focus is not to provide solutions only for specific operators but for the execution of a whole query pipeline that would optimally leverage all compute resources. Also, all the related work described above depend on static cost-model decisions that use low-level hardware parameters in order to distribute the workload between cores and co-processors. On the other hand, our scheduling strategies are designed to be adaptive at query runtime and to avoid static decisions before the actual execution pipeline starts.

Fine-Grained Query Processing Models: As we mentioned throughout the paper, in [13], the authors propose a novel query processing model that splits the input of a query pipeline into equal sized partitions, called morsels, and then schedules these fine-grained elements on worker-threads that execute them in parallel. The main idea is to provide a scheduling model that is fully elastic at query run-time, so that the system can adapt its decisions accordingly. In this way, the system is able to utilize all CPU resources effectively by applying techniques such as work-stealing. While our query processing model builds on the ideas from this paper, it differs in the way we address the new challenges arising due to the characteristics of the emerging heterogeneous multi-core environments, as we described in Section 3 in more detail.

Also, our scheduling scheme has similarities with the approach used in QPipe [8]. While QPipe uses a similar queue-based concept to schedule work for different database operators (similar to our function-specific work queues), there are some important differences: First, the focus of the scheduling strategies in QPipe is on the sharing of data and work between queries, whereas our focus is to maximize the utilization of all cores and accelerators. Second, different processing units might share the same queue to enable dynamic scheduling decisions, which is a key aspect of *SiliconDB* that allows it to adapt to different co-processor architectures.

8. DISCUSSION

Since we develop and test our ideas specifically on the Sparc M7 platform, one could argue whether or not the techniques we implemented in this paper are applicable to other platforms such as Intel Harp that combines an FPGA with normal cores or other designs such as APUs which combine CPUs and GPUs.

In order to apply the design presented in this paper with *SiliconDB*, we require that the platform in question provides well-defined semantics regarding the characteristics of its accelerators. More specifically, the functionality they can support (e.g., the type of operations they can execute) and their scheduling mechanisms, if they implement active or passive models (as in the case with DAX engines). Here the former is particularly important for *SiliconDB*'s query compilation and execution models in order to properly generate query sub-pipelines and incorporate them with function-specific work-queues, while the latter is crucial for the adaptation of the query scheduler in order to handle the scheduling of accelerators. In the following, we first discuss these points from the perspective of environments that provide re-configurable accelerator units, specifically FPGAs such as Intel Harp and the end with some thoughts on APUs.

One challenge to adapt *SiliconDB* for FPGAs would be if the query operations provided by the FPGA would change at query run-time. However, due to the high reconfiguration costs of FPGAs this is usually not the case for query processing systems. Thus, the capability of FPGAs are typically considered fixed at query runtime, just in an ASIC-based architecture. In case new architectures provide reconfiguration options with negligible costs, this would create an interesting venue for future work, where the system could decide to change the functionality of the underlying FPGAs depending on the state of the query workload. For instance, if the FPGAs are configured to support scans initially, but the system starts to serve more joins than there are scans, it might decide to re-configure the hardware to support joins to better leverage the hardware space.

For APUs, the situation is slightly different since the GPU can provide kernels for all database operations. However, we think that the scheduling model of *SiliconDB* would still be beneficial in order to leverage all compute resources (CPU cores and GPUs) and adapt to the different speeds.

9. CONCLUSIONS

In this paper, we presented *SiliconDB* that implements novel parallel query execution strategies for heterogeneous environments that combine normal cores with specialized ASIC-based accelerators on the same socket. We designed, implemented and experimentally tested our proposals on top of the Sparc M7 processor.

Our design and approaches respect the functional and execution-level limitations of accelerator units, and aim to maximize the collective utilization of all processing elements in the system in an attempt to improve end-to-end performance. To this end, we propose cost-based scheduling models, and study query plan modifications that improve system utilization at the expense of small increases in total execution costs. Based on the SSB benchmarks, we showed that *SiliconDB* provides a speed-up of a factor of up to 2× compared to alternative state-of-the-art parallel execution strategies that have been developed for heterogeneous environments.

10. REFERENCES

[1] M. Albutiu and other. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[2] C. Balkesen et al. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[3] S. Breß. The design and implementation of cogadb: A column-oriented gpu-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[4] S. Breß et al. Automatic selection of processing units for coprocessing in databases. In *Advances in Databases and Information Systems - 16th East European Conference, ADBIS 2012, Poznań, Poland, September 18-21, 2012. Proceedings*, pages 57–70, 2012.

[5] S. Breß et al. Ocelot/hype: Optimized data processing on heterogeneous hardware. *PVLDB*, 7(13):1609–1612, 2014.

[6] S. Breß et al. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, pages 1891–1906, 2016.

[7] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1603–1618, New York, NY, USA, 2018. ACM.

[8] K. Gao et al. Simultaneous Pipelining in QPipe: Exploiting Work Sharing Opportunities Across Queries. In *ICDE*, 2006.

[9] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[10] J. He et al. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10), Aug. 2013.

[11] J. He et al. In-cache query co-processing on coupled CPU-GPU architectures. *PVLDB*, 8(4):329–340, 2014.

[12] Intel. Harp. https://www.ece.cmu.edu/ calcm/carl/lib/exe/fetch.php? media=carl15-gupta.pdf.

[13] V. Leis et al. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[14] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *VLDB*, 2011.

[15] Oracle. Sparc m7. www.oracle.com/SPARC-M7.

[16] J. Power et al. Implications of emerging 3d gpu architecture on the scan primitive. *SIGMOD Record*, 44(1):18–23, 2015.

[17] J. Power et al. Toward gpus being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *DaMoN*, pages 1–8, 2015.

[18] I. Psaroudakis et al. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 10(2):37–48, 2016.

[19] I. Psaroudakis et al. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 2016.

[20] S. Tang et al. Elastic multi-resource fairness: balancing fairness and efficiency in coupled CPU-GPU architectures. In *SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 875–886, 2016.

[21] K. Zhang et al. DIDO: dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 671–682, 2017.