He..ro DB: A Concept for Parallel Data Processing on Heterogeneous Hardware

Michael Müller[1], Thomas Leich[2], Thilo Pionteck[3], Gunter Saake[3], Jens Teubner[4], and Olaf Spinczyk[1(✉)]

[1] ESS Group, Institute of Computer Science, Osnabrück University, Osnabrück, Germany
{michael.mueller,olaf.spinczyk}@uos.de
[2] Harz University of Applied Sciences, Wernigerode, Germany
tleich@hs-harz.de
[3] Otto-von-Guericke-University, Magdeburg, Germany
thilo.pionteck@ovgu.de, saake@iti.cs.uni-magdeburg.de
[4] DBIS Group, Department of Computer Science, TU Dortmund University, Dortmund, Germany
jens.teubner@cs.tu-dortmund.de

Abstract. Due to the growing demand on processing power and energy efficiency by today's data-intensive applications developers have to deal with heterogeneous hardware platforms composed of specialized computing resources. These are highly efficient for certain workloads but difficult to handle from the software engineering perspective. Even state-of-the-art database management systems do not exploit *all* heterogeneous hardware components, as their characteristics differ significantly. They are thus hard to integrate within a coherent database architecture.

To address this problem, we propose a design concept that is based on a layered system software architecture: He..ro DB transforms a data-flow graph that describes the data-processing application to a task-based execution plan. Task implementations for the different computing resources and a reasonable degree of parallelism are chosen automatically based on available resources. The concept can cover any hardware configuration and application scenario. It is versatile and offers opportunities for independent optimization on each layer.

Keywords: Heterogeneous many-core systems · Data processing · Databases · Task-parallel programming

1 Introduction

The hardware industry is trying to cope with the growing demand on computational power and energy efficiency of today's data-intensive applications

This work has been carried out in the course of the priority program 2037 *Scalable Data Management for Future Hardware* funded by the German Research Foundation (DFG). The authors would like to thank the DFG for funding and all the project partners for the helpful discussions.

A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 82–96, 2020.
https://doi.org/10.1007/978-3-030-52794-5_7

by developing hardware that is inherently parallel and also heterogeneous. For example, the Xilinx Ultrascale+ combines four ARM cores, two ARM Cortex R5 cores and an FPGA fabric on one chip.

Especially data-intensive cyber-physical systems could benefit from the efficiency of these modern hardware platforms. However, we are not aware of any database management system that could make use of *all* heterogeneous resources at once in parallel for boosting its query processing performance. A novel software architecture would be needed. However, for the designers of system software this kind of platform raises a lot of new research questions. In this paper we will address the following:

RQ1: How can the available hardware resources be fairly assigned to isolated **concurrent applications**?
RQ2: Is it possible to **abstract** from individual resource types without losing the ability to exploit a computing resource's specific strengths?
RQ3: With which **execution model** can applications – especially data-intensive programs with high demands on computing power – make use of and benefit from the available heterogeneous resources in a coordinated and parallel manner?

To facilitate system software engineering for heterogeneous platforms that are running data-intensive application, this paper presents a design concept, namely the He..ro-DB architecture and a preliminary evaluation of an early prototypical implementation. By a layered software architecture different concerns become cleanly separated and optimizations are possible on each of these layers independently. The He..ro-DB architecture assumes that data-processing operations can be expressed as a data-flow graph containing logical operators and edges along which data flows from data sources to data sinks, e.g. database tables. We sketch an architecture that is able to transform this high-level description into an optimized execution plan that consists of elementary operations, which we call *tasks*. During this transformation process the most appropriate degree of functional and data parallelism is estimated and the most promising selection of optimized task implementations for the heterogeneous computing resources is made.

The outline of this paper is as follows: To motivate the need for the presented conceptual design framework Sect. 2 will discuss existing approaches to integrate heterogeneous computing resources into data-processing systems. As none of the existing systems is able to exploit all available (heterogeneous) computing resources, we will describe the He..ro DB system software architecture in Sect. 3. The validation of the approach is based on a concrete application scenario. Section 4 will explain how that would be handled in a He..ro-based system software stack. Section 5 will reflect the presented architecture by discussing the design space, the decisions that have been made, and the remaining freedom for concrete implementations. Finally, we will present the results of a performance evaluation that we conducted with the He..ro DB prototype implementation in Sect. 6 and our conclusions in Sect. 7.

2 Related Work

The Utilization of heterogeneous computing resources, such as FPGAs and GPUs, to accelerate database operations has been investigated intensively in the recent years. So all major operators have been realized as FPGA functions or GPU kernels, such as sorting [5,12,16], selection [8,15] and join [5,9,18]. Although significant performance improvement could be achieved, none of them allowed to accelerate a whole query.

To accelerate a whole query, query compilers for accelerators have been investigated. So Sukhwani et al. [17], Glacier [11] and Hawk [3] provide a query compiler for FPGAs (the former) and GPUs. Though these solutions are limited to specialized database machines, as they do not provide an execution model that allows load balancing or dynamic adaptation.

With OmniDB [19] and Ocelot [7], attempts have been made to create a fully heterogeneous DBMS. As both use a hardware-oblivious approach though, they cannot make full use of the special characteristics each accelerator provides. By using a hardware-sensitive approach, though sacrificing portability, CoGaDB [2] and Hype [4] can improve the utlization of accelerator hardware even more, leading to better system performance. However, these solutions do not take concurrently running applications into account and are thus exposed to performance degradation by interference with other applications.

OpenCL[1] has made the notion of a kernel, a closed unit of parallel work, the de-facto standard execution model for heterogeneous programs. It allows the application programmer to offload certain functions (kernels) to an accelerator. Although OpenCL provides a good way of abstracting hardware details while still preserving most of its characteristics, it does not provide any means for multiprogramming or scheduling. FluidiCL [13], StarPU [1] and Harmony [6] try to fill this gap by providing a coherent programming model and runtime for heterogeneous tasks. These runtimes schedule tasks or kernels on a given accelerator depending on the expected load and timing requirements of the application. Though, none of them consider multiprogramming, i.e. the concurrent execution of isolated applications.

So far none of the mentioned solutions provide a holistic approach allowing database applications and other concurrent applications to fully leverage the potential of modern parallel and heterogeneous hardware, while providing abstractions for heterogeneous processors, sophisticated load balancing and isolation of concurrent applications without losing the ability to distribute heterogeneous resources among them. This encourages the motivation for our proposal which addresses all the mentioned challenges.

3 He..ro-DB Architecture

This section describes the proposed architecture in detail. It starts by giving an overview and continues with a discussion of each layer in a bottom-up manner.

[1] See https://www.khronos.org/opencl.

3.1 Overview

The He..ro-DB architecture seperate resource management (Layer 0), task scheduling (Layer 1) and query planing (Layer 2) from eachother in their own functional layer. Applications run in isolated resource containers, called cells, having their individual task scheduler and query planer, if needed. This allows maximium flexibility regarding the choice of implementations. So each application may have its own tailored scheduling and query planing. As the management of hardware resources has to be done application-independently Layer 0 exists only once as shared layer for all applications.

3.2 Layer 0: Resource Partitioning

Layer 0 is responsible for assigning the available hardware resources to application cells running concurrently on the same hardware platform.

Provided Functions: For a special-purpose system with only a single application, Layer 0 may have almost no functionality. But most modern systems are more complex and isolation of system components is needed, as for cyper-physical systems where mission-critical control functions shall be isolated from other parts of the system. Below is a list of functions that a Layer 0 Implementation would provide;

Isolation: In order to avoid propagation of errors between application cells and system software components as well as security issues, temporal and spatial isolation are required. A typical way of achieving this is by virtualization of resources, such as CPU and main memory. In today's heterogeneous hardware landscape not all computing resources support virtualization on the hardware level. In this case access can be granted only through a special API, which fully controls resource usage.

Prioritization: Since the criticality of applications in a system may differ, the Resource partitioning shall take priorities into account. So a high-priority application (e.g. an interactive user-application) shall be granted more resources than a low-priority one (e.g. a background task). The resource management shall also withdraw resources from lower priority applications when needed by an application of higher priority. Realtime applications might be supported by static assignment of isolated hardware resources.

Mapping: Software components running within one cell are more likely to interact with each other than components in different cells. Therefore, Layer 0 should optimize the placement of cells with the system. For example, in a Non-Uniform Memory Architecture (NUMA) it would make sense to take locality with NUMA regions into account. The same holds for memory areas and I/O devices used by a cell.

Stabilization: A cell reconfiguration (adding, withdrawing, or replacing resources) is always costly. Therefore, Layer 0 must implement strategies to keep cells as stable as possible. The minimum amount of resources that are

assigned to a cell is a certain percentage of all resources and depends on prioritization. However, cells produce load dynamically and it is only necessary to provide these resources if actually needed. In such "full load" situations a cell can be given even more resources if other cells are not under full load at the same time. Here a good balance between reactivity and stabilization must be found. The load situation with the cells must be constantly monitored with low overhead.

The list of functions is not intended to be complete. It motivates the need for a global resource management software layer that simplifies application development by handling several operational concerns transparently.

Interface: Layer 0 shall provide a *Cell Management Interface* (CMI) and a *Resource Introspection Interface* (RII). CMI is needed to start, configure, and stop application cells. RMI can be used by cells to get information about the *physical* resources assigned to them. Changes in resource assignment can be signaled to the affected domain. If resources have to be withdrawn, the affected cell will be granted a reasonable amount of time to stop using the resources. Layer 1 is designed in a manner that allows resources to be released quickly (see next section) and to exploit the specific features of the assigned physical hardware components.

Structure/Implementation: The *Resource Partitioning* layer resembles an exokernel. Both share the same principle that there is almost no abstraction of the underlying hardware, to allow optimizations in the applications running on top. This distinguishes Layer 0 from hypervisors which virtualize a complete computer system, hiding the characteristics of the actual hardware, and also from monolithic and microkernel operating systems which usually only provide an abstract machine.

Although Layer 0 is a kind of exokernel, its resource management strategies differ greatly. The original exokernel was designed for machines with a single or only a few CPUs and with uniform memory access. However, the resource partitioning of He..ro DB is designed for heterogeneous many-core systems with complex memory hierarchies and non-uniform memory access that call for new ways of assigning hardware resources.

3.3 Layer 1: Task-Based Runtime System

The purpose of Layer 1 is to provide a runtime system for each cell with a programming model for arbitrary applications that want to make use of heterogeneous computing, memory, and I/O resources in a coherent and automatically optimized way. Figure 1 shows the components of Layer 1 and its interfaces to Layer 0 and 2.

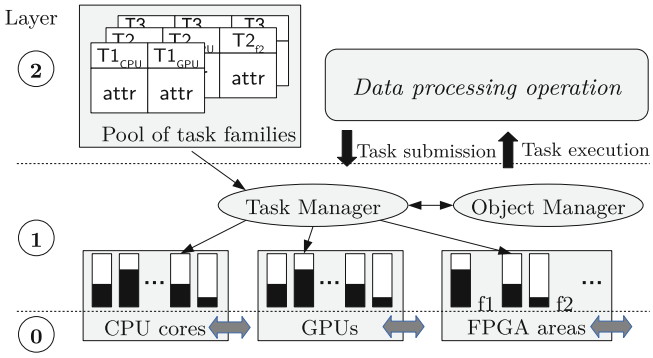

Fig. 1. Structure and interface of the *Task-Based Runtime System*

Provided Functions: The main function provided by Layer 1 is the execution of *tasks* that are submitted by Layer 2 components, such as data processing operations. In this context we define a task as a *finite and non-preemptible* computation on any of the heterogeneous resources of the machine that may read or write data structures in memory (*data objects*) and may have parameters. A task is a member of a *task family* and has a number of *attributes* that describe its usage of resources and interactions with other tasks. Each task in a task family implements a semantically equivalent computation, such as sorting an array of integers or scanning a table for a specific entry, in a different way. The main purpose of this concept is to group variants of data-processing code that is implemented and optimized for different heterogeneous hardware components. In Fig. 1, for instance, the task T1 exists in two variants (family members): One that can execute on a CPU and one GPU implementation. Task families must not be empty, but don't have to be complete.

The *Task Manager* is responsible for a number of scheduling and placement decision needed to execute a task. The *Object Manager* component keeps track of the available memory resources and provides a cost model for memory transfers. During operation the task and object manager have to deal with the following issues:

Load Balancing: If Layer 2 submits a GPU task, but the only assigned GPU is already in use for another task, the task manager may either put the new task into a waiting list or choose another member from the task's family and run the same computation on a different hardware resource. By this mechanism the load on the heterogeneous computing resources can be balanced.

Optimization: Load balancing shall reduce the performance of the system as little as possible. Therefore, the task placement decision must take a cost model into account that considers necessary copying of memory objects and the expected resource usage of the different task family members. The latter information shall be provided by the developer or a dynamic profiler in task attributes.

Synchronization: If tasks have data dependencies or require mutual exclusion while accessing a certain memory object, an elegant way to achieve serialization is to simply insert the tasks in the *same* waiting list. This scheme avoids costs that would otherwise be induced by lock-based synchronization. However, load balancing and optimization might sometimes outweigh the benefits of this kind of task synchronization.

Adaptation: The resource partitioning Layer 0 aims at keeping the assigned resources stable. Nevertheless, Layer 1 must be able to deal with elasticity, i.e. dynamic resource availability. Layer 0 describes the available resources and signals changes by the RII. When a resource is added, Layer 1 can use that from that moment on for the execution of new tasks. Resource withdrawal is more challenging. Layer 1 must make sure that the resource is no longer used within a short period of time. Otherwise, Layer 0 would simply terminate the cell. This is done by resubmitting the tasks that are currently on the waiting list of the resources and waiting for the running task to finish.

Interface: The *Task Execution Interface* (TEI) provides functions for submitting tasks. These are executed asynchronously at a later point in time. For each task there are static attributes, such as the kind of computing resource needed and the expected execution costs. Parameters provided during task submission include references to input and output data objects and synchronization dependencies to other tasks.

Besides this, the *Resource Introspection Interface* (RII) from Layer 0 is also provided as a Layer 1 interface. Thereby, Layer 2 components, such as data-processing operations, can create task execution plans with optimized and well-balanced long-term task-to-resource mappings. As a consequence – if available resources and the load situation are stable and as planned by Layer 2 – Layer 1 will never have to send a task to a different resource than it was intended to run on by Layer 2.

Layer 1 also provides a *Load Introspection Interface* (LII). It describes the current load situation and can be used to inform Layer 2 components if tasks from that component have to be executed often on other computing resources than intended. This allows Layer 2 to create a new execution plan that takes the current resource availability and load situation into account. By this means there is a feedback from Layer 0 up to Layer 2, for instance, when a newly started high-priority application (cell) consumes so many resources that re-planning on all levels becomes necessary.

Structure/Implementation: The *Task-Based Runtime System* can be regarded as a lightweight library operating system or Unikernel [10], because there is one instance per application cell. The runtime system can assume that the application tasks are cooperative. There is no need for user/supervisor-mode separation or other protection means on this level. Traditional operating system features such as file systems or network protocol stacks can be implemented either on top of Layer 1 within an application cell or in a separate global system

service cell, which would also make use of the task-based runtime system, as it facilitates the handling of dynamic resource availability.

3.4 Layer 2: Data Processing

The *Data Processing* layer maps the operations of a data-processing application to the task-based execution model of Layer 1.

Provided Functions: For each operation, such as a query on an in-memory database, Layer 2 creates a state machine as shown in Fig. 2. The following functions are involved:

Planning: The first performed step is the step-wise transformation of a data-flow-oriented logical query plan into a physical execution plan that exploits task as well as data parallelism and all available heterogeneous computing and memory resources simultaneously. It also considers the current resource availability and the expected execution costs of the task implementations. An example for this can be found in Sect. 4.

Task Families: During planning a high-level operation must be broken down into primitive operations for which there is a task-based implementation in the pool of task families (see Fig. 1). These implementations are the building blocks of any performed database operation. They are part of Layer 2 and assumed to be known by its planning component.

Execution: When the plan is ready, the respective tasks must be submitted with the *Task Execution Interface* (TEI) of Layer 1. Tasks are represented at runtime by special memory objects called task objects. These memory objects hold the task parameters and all other dynamic task attributes. Layer 2 creates, destroys, and modifies these objects with the help of the Layer 1 object manager.

Dynamic Re-planning: In case that Layer 1 signals a high percentage of tasks that had to be executed on a different computing resource than it was planned, a re-planning will be triggered.

Interface: The logical query plan is provided by the data-processing application as a set of linked memory objects through the *Data Processing Interface* (DPI). The planning and execution steps are triggered by submitting a built-in Layer 2 task that will create further sub-tasks. Hence, planning can also benefit from all heterogeneous computing resources and parallelism. The results of the operation are passed to the applications via memory objects and callback tasks that are triggered when results are ready.

Structure/Implementation: The first and most challenging part of the implementation is the *planning*. High-level operations must be broken down into primitive operations and task families that implement these primitives must be found.

Based on an estimation of data volume, costs per task, costs for memory object transfers, and the available resources, the plan must to transformed into a task graph. Finally, the planned tasks must be executed by passing the respective task family members to Layer 1.

4 Case Study

We validate the presented architecture by explaining how a simple hypothetical data-procession operation would be executed on an exemplary heterogeneous hardware platform. This time we handle the architectural layers from top to bottom.

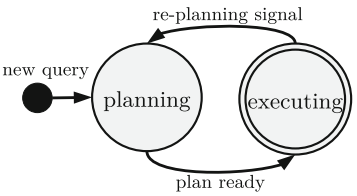

Fig. 2. Per-operation state machine

4.1 Scenario

The hardware platform consists of a multi-core CPU, a GPU that supports multiple independent execution contexts, and an FPGA with two equally-sized re-configurable regions. An in-memory database table R shall be scanned for entries that fulfill the conditions P_1, P_2, and P_3: $\sigma_{P_1 \wedge P_2 \wedge P_3}(R)$. A conventional query optimizer, which only works on the logical level, can turn this relational algebra expression into the data-flow-oriented execution plan shown in Fig. 3 (left part). We regard this as the input for Layer 2.

4.2 Layer 2

Layer 2 is responsible for planning and for submitting tasks as already shown in Fig. 2. Before planning the execution of a new query, Layer 2 gets information on currently available CPU, GPU, and FPGA resources from Layer 1 (LII). If another query is already being executed, it is likely that it uses *all* available cell resources. Therefore, both, the old and the new query execution, must be (re-)planned with an adequate fraction of the resources and considering priorities. Similarly, re-planning is necessary when a query execution terminates and additional resources become available or in situations in which Layer 0 has to resize the cell. In the following, we will focus on planning a single query and its execution.

Planning. During the planning phase the logical data-flow-oriented execution plan (Fig. 3) is searched for structural patterns that would allow a graph transformation. For example, a specific primitive could be replaced by a task node or a number of cooperating tasks. The tasks are chosen from the pool of task families.

It is also possible that a transformation replaces a group of nodes. In our example scenario, the three nodes "Selection σ_{P_2}", "Selection σ_{P_3}", and "Intersection" would be matched and replaced be an efficient FPGA-based implementation of this compound operation.

Eventually, all logical primitives will be replaced by task nodes. It is a complex optimization problem to apply transformations in the right order to minimize the resource consumption during the execution phase. In order solve this problem, Layer 2 needs a cost model so that alternative paths in the search space can be compared. Therefore, estimates for execution costs must be provided by the developer of each task implementation. This metadata is also stored in the pool of task families. Furthermore, costs for transferring data between different memories are provided by the object manager in Layer 1. Based on this, it can be decided to replace the three aforementioned nodes by an FPGA-based selection task and two data transfer tasks as shown on the right-hand side of Fig. 3.

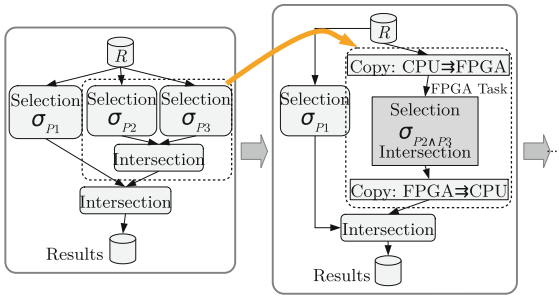

Fig. 3. Execution plan transformation

After replacing all primitives with the "cheapest" available task implementations, performance optimizations by exploiting data parallelism would be taken into account. For example, the cheapest implementation of "Selection σ_{P_1}" would in our scenario be a CPU-based task, because the data transfer costs from CPU memory to GPU memory might be prohibitively high. The planner would consider instantiating multiple parallel CPU tasks that perform the selection after splitting the data into chunks. However, the compound operation on the FPGA can be instantiated only once, because only one FPGA region with the necessary hardware structure is available. The optimal amount of parallel selection tasks on CPU cores depends on the data rate of the FPGA task, because the results of both sides will have to be merged. Both data paths should produce results with the same rate. In our scenario the planner predicts that the FPGA will handle the data faster than the parallel CPU cores. Therefore, it also uses the GPU for a part of the data to further boost the data rate of "Selection σ_{P_1}". Again, nodes for splitting, transferring data, and merging would be added. Figure 4 shows the final execution plan.

Execution. Typically, the plan execution will be performed in a pipelined manner. For example, in our scenario the FPGA has only a small amount of on-chip RAM. It is not possible to copy all the data from R to the FPGA memory at once. As a consequence, the task for selecting rows that match P_2 and P_3 will have to be triggered many times. The execution engine is responsible for allocating the necessary task object, setting up task parameters, and triggering follow-up tasks upon completion. Data dependencies in the execution plan define the order in which tasks must be submitted.

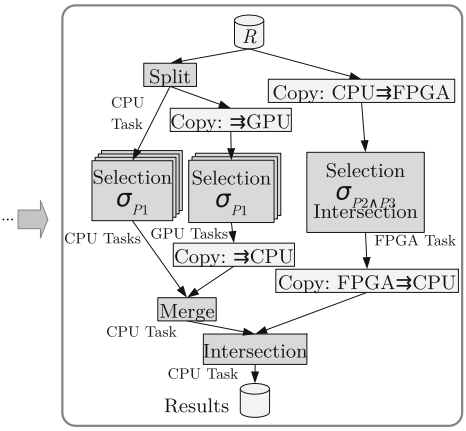

Fig. 4. Final execution plan for the case study

Tasks can use data objects that may reside in any memory region as input or output. The Layer 1 object manager is responsible for copying data from one region to another if necessary. This means that the data transfer nodes in the execution plan can be ignored. They are only needed for cost estimation.

4.3 Layer 1

Layer 1 manages work queues for all computing resources. Besides assigning tasks to these queues the task manager monitors the load. For example, in our scenario the costs calculated by the planner on Layer 2 might have been too imprecise. It might turn out that all CPU cores are overloaded. In this case, Layer 1 would access the pool of task families and replace a CPU task by a GPU task. The object manager would automatically perform the necessary data transfers.

Layer 1 is thus capable of performing short-term load balancing. However, if the rate of these replacements exceeds a threshold, Layer 1 would be informed to trigger re-planning.

The implementation handles all tasks that are submitted within the cell. This means that also other application tasks or tasks submitted by a library operating system contribute to dynamically changing load situations.

4.4 Layer 0

Layer 0 monitors the resource usage of all cells. While the cell is executing the query, it might withdraw resources from other cells that produce a low load. For example, a second cell might have been running with CPU cores at a low clock speed. While the query is running, Layer 0 might decide to withdraw CPU cores from the second cell, increase the clock of the remaining cores, and add the withdrawn cores to the cell that executes the query. A signal mechanism will be used to inform Layer 1 asynchronously.

5 Discussion

This section will reflect on the design decisions that we made and issues that were intentionally not addressed.

Decision: Task abstraction The ability to abstract from arbitrary heterogeneous computing resources requires a universal abstraction. The "task" can model the execution of a function on a CPU, a kernel on a GPU, or the data flow through gates on an FPGA. Tasks are smaller units of computation than threads and it is, thus, easier to annotate data structures used for input and output.

Decision: Layer 0 and 1 support arbitrary tasks It would be unrealistic to assume that the complete hardware platform is always dedicated to data processing. Therefore, we created a functional hierarchy that first handles resource partitioning and global management functions on Layer 0. If these features are not needed, Layer 0 could be reduced to a simple introspection mechanism that describes the available (static) hardware resources to the layers above.

Layer 1 implements the task execution model. If there was only a data-processing application on a dedicated system, this application would still benefit from the functions provided here. Supporting location transparent identification and automatic transfer of data objects simplifies the design of all software layers above.

Decision: Dynamic resources Layer 1 and 2 assume that resources can be withdrawn. This can have multiple reasons: First, Layer 0 might decide to assign some resources to another cell. However, even in systems with only one application, resources might be dynamic. For example, due to thermal issues not all computing resources can always run at full speed. The system software might need to throttle certain hardware components, which makes it necessary to deal with this problem. Furthermore, in future manycore systems, computing resources might permanently fail or be intentionally turned off to control aging.

Assuming dynamic resource availability makes cost calculations for data processing operations unreliable. However, this situation is not new to optimizers in DBMS and can be dealt with by re-planning.

Decision: Task Families We are aware that OpenCL allows developers to program an algorithm that could run on either CPU, GPU, or FPGA. In our opinion this approach is orthogonal to the concept of task families. The members of a task family could be generated from the same source code, e.g. from OpenCL code, or from completely differently code. The only assumption is that the functional behavior (input-to-output transformation for given parameters) is the same. It is not necessary that a task family has members for all computing resources – any subset is sufficient. By not assuming that task family members are created from the same source code there is room for arbitrary implementation optimizations.

Not addressed features: Various system optimizations are possible on Layer 0, which are left to individual implementations. So far our prototype only posseses a task scheduler that schedules a set of tasks in a way that the makespan is minimized. More sophisticated strategies can be implemented on Layer 1 that also consider the memory hierarchy. As query optimization is a research area on its own, we sketched only a few ideas for inspiring developers.

6 Prototype Performance

In a set of early experiments with our prototype implementation we have evaluated the performance of selection operators. Figure 5 shows that combining CPU cores and a GPU actually pays off (right column). The execution platform was a notebook with Intel Core i7 CPU with eight cores and an integrated GPU[2]. He..ro DB was executed directly on the hardware without any other system software. Our prototype lacks Layer 0. Therefore, Layer 1 is a self-made bare-metal tasking framework.

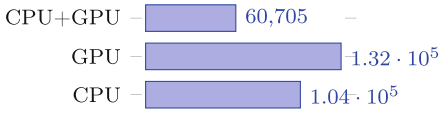

Fig. 5. Makespan for processing units in μs

The test data was a 16 MiB sized table from the TPC-H benchmark [14]. 16 queries of three different kinds were executed randomly. The implementations of the selection operators were provided as a task family: Code for the GPU was written in OpenCL while the code for the CPU was written in C++. The task scheduler could thus decide at runtime where the next task (operator) is to be executed. In the combined CPU/GPU run this decision was based on an execution time estimate, which is contributed by each task itself. With this the scheduler can estimate for each execution unit when the new task would start to be executed, based on the length of the task queue, and when it would be finished. The processor, which would finish the task first, is chosen.

During the experiments it turned out that for the highly memory bound selection operator, the integrated GPU is only three times faster than an i7 CPU core. For compute-intensive tasks we have seen much higher speedups on the same platform. This makes us believe that we follow the right approach, because (A) the combined use of CPU core and accelerators improves the performance and (B) the scheduling decision is non-trivial—meaning that a specialized system software component is needed.

7 Conclusions

This paper addressed the problem of exploiting all available computing resources on a modern heterogeneous hardware platform for data-intensive applications.

[2] Only seven CPU cores were used for task execution, as the eighth core was needed for benchmark control and time measurement.

The proposed He..ro DB is a design concept that is based on a layered system software architecture. It can be used as a blueprint for future system designs and supports independent optimizations on all of its layers. Some of the presented ideas have already been implemented in prototype systems by the authors.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. CCPE **23**(2), 187–198 (2011)
2. Breß, S.: The design and implementation of CoGaDB: a column-oriented GPU-accelerated DBMS. Datenbank-Spektrum **14**(3), 199–209 (2014). https://doi.org/10.1007/s13222-014-0164-z
3. Breß, S., Köcher, B., Funke, H., Rabl, T., Markl, V.: Generating custom code for efficient query execution on heterogeneous processors. arXiv preprint arXiv:1709.00700 (2017)
4. Breß, S., Saake, G.: Why it is time for a HyPE: a hybrid query processing engine for efficient GPU coprocessing in DBMS. Proc. VLDB Endow. **6**(12), 1398–1403 (2013)
5. Casper, J., Olukotun, K.: Hardware acceleration of database operations. In: Proceedings of the FPGA 2014, pp. 151–160, New York, NY, USA. ACM (2014)
6. Diamos, G.F., Yalamanchili, S.: Harmony: an execution model and runtime for heterogeneous many core systems. In: Proceedings of HPDC 2008, pp. 197–200. ACM (2008)
7. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. Proc. VLDB Endow. **6**(9), 709–720 (2013)
8. István, Z., Sidler, D., Alonso, G.: Runtime parameterizable regular expression operators for databases. In: FCCM 2016, pp. 204–211, May 2016
9. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: GPU join processing revisited. In: 8th International Workshop on DaMoN (DaMoN 2012), pp. 55–62, New York, NY, USA. ACM (2012)
10. Madhavapeddy, A., et al.: Unikernels: library operating systems for the cloud. SIGPLAN Not. **48**(4), 461–472 (2013)
11. Mueller, R., Teubner, J., Alonso, G.: Glacier: a query-to-hardware compiler. In: ACM SIGMOD International Conference on Management of Data (SIGMOD 2010) (2010)
12. Mueller, R., Teubner, J., Alonso, G.: Sorting networks on FPGAs. VLDB J. **21**(1), 1–23 (2012). https://doi.org/10.1007/s00778-011-0232-z
13. Pandit, P., Govindarajan, R.: Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. In: 12th International Symposium on Code Generation and Optimization (CGO 2014), pp. 273:273–273:283, New York, NY, USA. ACM (2014)
14. Poess, M., Floyd, C.: New TPC benchmarks for decision support and web commerce. ACM SIGMOD Rec. **29**(4), 64–71 (2000)
15. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using FPGAs. In: FCCM 2001, pp. 227–238, March 2001
16. Sukhwani, B., et al.: Large payload streaming database sort and projection on FPGAs. In: IEEE International Symposium on Computer Architecture and High Performance Computing, pp. 25–32. IEEE (2013)

17. Sukhwani, B., et al.: A hardware/software approach for database query acceleration with FPGAs. Int. J. Parallel Progr. **43**(6), 1129–1159 (2015). https://doi.org/10.1007/s10766-014-0327-4
18. Ueda, T., Ito, M., Ohara, M.: A dynamically reconfigurable equi-joiner on FPGA. IBM Tehnical Report RT0969 (2015)
19. Zhang, S., He, J., He, B., Lu, M.: OmniDB: towards portable and efficient query processing on parallel CPU/GPU architectures. Proc. VLDB Endow. **6**(12), 1374–1377 (2013)