8 Communication Awareness

The ubiquity of connected devices and parallel computing platforms challenges efficient and reliable execution of machine learning algorithms. If machine learning workloads are executed merely locally, a system does not always have sufficient resources at its disposal to perform the necessary operations fast enough. Furthermore, at a smaller scale, multiple hardware components these days are interconnected via on-chip or off-chip networks to create many-core systems. Communication, synchronization, and offloading have thus become essential in designing embedded systems under communication and resource constraints.

This chapter presents (1) the timing predictability of embedded systems and (2) the communication architecture in heterogeneous CPU/GPU environments. Synchronization with resource sharing, communication with potential failures, and probabilistic timing information are presented in Section 8.1. Bandwidth limitations of different execution models and coprocessor-accelerated optimization are presented in Section 8.2.

https://doi.org/10.1515/9783110785944-008

8.1 Timing-Predictable Learning and Multiprocessor Synchronization

Kuan-Hsun Chen
Junjie Shi

Abstract: With the increasing demand for time-predictable machine learning applications, e.g., object detection in autonomous driving systems, such a trend poses several new challenges for resource synchronization in real-time systems, especially when hardware accelerators like Graphics Processing Units (GPUs) are considered as shared resources. When the shared resources have relatively high utilization, conventional synchronization mechanisms might result in performance downgrade.

We thus propose the emerging Dependency Graph Approach (DGA), where the precedence constraints of all the computation segments are pre-proceeded. Such a non-work-conserving approach can schedule long critical sections, which may be even longer than the period of another task. This is not the case in all the other work-conserving protocols typically in use. Throughout numerical experiments, we show that DGA outperforms all the other conventional protocols in all the evaluated configurations when shared resources are highly utilized.

Additionally, a system does not always have sufficient resources at its disposal to perform the necessary operations fast enough if machine learning workloads are executed merely locally. One sound approach is to offload heavy workload to powerful remote servers and expect the inference outcome can be received in time. However, since this approach highly depends on network connectivity and responsiveness, typically only non-critical tasks are offloaded, whose timing requirements are less strict than those of critical tasks. Against such a pessimistic design, we present two novel offloading protocols that offload both critical and non-critical tasks. They handle uncertain connections while providing certain timing guarantees.

To achieve a timing-predictable design, typical timing analyses always consider the worst-case execution pattern to derive timing guarantees. But this approach is often too restrictive for some machine learning applications with soft timing constraints. To mitigate the pessimism, we develop several timing analyses of the probability of deadline misses and the deadline miss rate, two important metrics considered in the literature to quantify timeliness.

8.1.1 Introduction

Under the von Neumann programming model, shared resources that require mutually exclusive accesses, e.g., shared files, data structures, etc., have to be protected by applying synchronization (e.g., *binary semaphores*) or locking (e.g., *mutex locks*) mechanisms. Protected code segments that have to access shared resource(s) mutually exclusively are called *critical sections*. For uni-processor real-time systems, longstanding protocols developed in the 90s, are the current state of the art. These are namely the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) by Sha et al. [623], as well as the Stack Resource Policy (SRP) by Baker [37].

Along with the development of multiprocessor platforms, multiprocessor resource synchronization and locking protocols have been proposed and extensively studied. These include Distributed-PCP (DPCP) [592], Multiprocessor PCP (MPCP) [591], Multiprocessor SRP (MSRP) [238], Flexible Multiprocessor Locking Protocol (FMLP) [58], $O(m)$ Locking Protocol (OMLP) [69], and Multiprocessor resource sharing Protocol (MrsP) [101].

However, the performance of aforementioned protocols highly depends on 1) how the tasks are partitioned and prioritized, 2) how the resources are shared locally and globally, and 3) whether a job/task being blocked should spin or suspend itself. In the literature, conventional synchronization mechanisms might result in performance downgrade, since most of them are designed for sporadic tasks with relatively low utilization for critical sections, which are often not able to represent emerging heavy-loaded machine learning applications. We thus propose a novel concept called DGA, which can serve high utilization of critical sections well.

Moreover, when the workload of a critical section, e.g., a machine learning workload on GPU, is extremely high, a system does not always have sufficient resources at its disposal to perform the necessary operations fast enough. A sound solution is to offload heavy workload to powerful remote servers and wait for the outcome of the inference processes. However, the performance and stability of this approach highly depends on the quality of the network. To improve flexibility, we propose several adaptive protocols to ensure that the timing requirements of safety- and mission-critical tasks are not violated even in the case of connectivity issues while obtaining the benefits of offloading computation shares.

Last but not least, to achieve a timing-predictable design, conventional timing analyses always focus on the worst-case execution pattern to derive hard timing guarantees. However, such analyses are sometimes too pessimistic when systems can accept rare deadline misses, e.g., for soft real-time systems. Limited deadline misses on many machine learning applications might only lead to performance degradation, e.g., for image and voice recognition on smart edge devices. Some end users might only feel inconvenienced without further serious consequences. However, people might still wonder how resilient the considered system is with respect to deadline misses in a probabilistic argument. To obtain the probability of deadline misses, we innovate a

well-known convolution-based approach based on multinomial distribution, and adopt several concentration inequalities to derive analytical upper bounds to further improve the efficiency of calculation.

Overall, the presented contributions in this chapter are as follows:

Dependency Graph Approach (DGA) We propose a novel method for periodic real-time task systems, which predestines the sequence where tasks can access resources. The conducted numerical simulations show that DGA outperforms all state-of-the-art approaches in most evaluated configurations, especially when utilization of critical sections is relatively high.

Offloading protocols for unreliable connection We present two offloading protocols that offload both critical and non-critical tasks. They deal with uncertain connections while providing certain timing guarantees. A case study on a robotic system demonstrates the applicability of the proposed protocols under various configurations.

Deadline-miss analyses A novel approach based on the multinomial distribution is proposed that calculates the deadline miss probability with drastically better analysis runtime without any precision loss. Furthermore, we propose several analytical bounds based on various concentration inequalities. The evaluation shows that our approaches are applicable for significantly larger task sets while preserving the quality of derived results, compared with conventional convolutional-based approaches.

8.1.2 Related Work

For multiprocessor systems, many resource synchronization and locking protocols are extensions of these aforementioned well-known uni-processor protocols. For example, Rajkummar et al. [592] proposed DPCP, where each resource is assigned on a processor statically, and critical sections are executed on the corresponding processor where the requested resource is assigned on. The extension MPCP [591] enables tasks to execute their critical section locally. In order to minimize the usage of stack memory in real-time systems, Gai et al. [238] proposed MSRP. Block et al. [58] introduced FMLP, where resources are divided into two groups, i.e., long and short. For short resources, critical sections are executed in a non-preemptable manner and tasks spin on their processors while waiting for resources. For long resources, tasks suspend themselves into a First In First Out (FIFO) queue while waiting. Brandenburg and Anderson [69] proposed OMLP, which ensures $O(m)$ maximum pi-blocking for any task set. Burns et al. [101] proposed MrsP, which allows tasks to progress other tasks that have occupied the same requested resource, in order to reduce the blocking time. A comprehensive survey of multiprocessor real-time locking protocols can be found in [68].

Besides fully relying on local computational power, offloading computation to remote servers is a reasonable solution to ease the pressure of resource constraints on embedded systems. In 2012, a cloud-assisted system for autonomous driving was firstly studied by Kumar et al. [406]. In 2015, Esen et al. [203] presented a software architecture named *Control as a Service* in which all control functions are completely moved to the cloud. In 2018, Adiththan et al. [4] proposed an adaptive offloading technique for control applications that makes all offloading decisions online based on a network performance monitor. Recently, Al Maruf and Azim [469] proposed a strategy for task offloading in multiprocessor mixed-criticality systems with dynamic scheduling policies under overload conditions. For real-time systems that allow offloading, one concept for modeling this particular local system view is *self-suspension* [127]. One of the state-of-the-art models can be applied such as the dynamic self-suspension model (e.g. [324], [442]), the segmented self-suspension model (e.g. [611]), or a hybrid model, e.g. [84]. For a detailed overview, see [127, 128].

To safely derive probabilistic timing guarantees, which enable a tradeoff between system safety and hardware costs, several techniques have been developed in the literature. Diaz et al. [177] developed a framework for calculating the deadline miss probability based on convolution for periodic task systems. In addition, Tanasa et al. [657] used the Weierstrass Approximation to approximate any arbitrary execution time distributions and applied a customized decomposition procedure to search all the possible combinations. However, the two approaches can derive only the probability of deadline misses with 7 and 25 jobs in the hyper-period, respectively. For sporadic real-time task systems, in which two consecutive jobs of a task do not have to be released periodically, Axer et al. [27] proposed evaluating the response-time distribution and iterating over the activations of job releases for non-preemptive fixed-priority scheduling. Maxim et al. [476] provided a probabilistic response time analysis by assuming a probabilistic minimum inter-arrival as well as probabilistic worst-case execution times (WCETs) for the fixed-priority scheduling policy. Ben-Amor et al. [46] extended the probabilistic response time analysis in [476] by considering precedence-constrained tasks. These convolution-based approaches are in general not scalable due to the huge number of jobs in the interval of interest.

8.1.3 Dependency Graph Approach

In this subsection, the dependency graph approach is presented in detail, including the primary design of DGA, the extension for supporting periodic task systems, and the corresponding scheduling algorithms.

8.1.3.1 Primary Design of DGA

We consider a set of *n frame-based* real-time tasks $\mathbf{T} = \{\tau_1, \ldots, \tau_n\}$ that is scheduled on M identical (homogeneous) processors. Each task is described by $\tau_i = ((C_{i,1}, A_{i,1}, C_{i,2}), T_i, D_i)$. The given tasks release their jobs at the same time and have the same period and relative deadline. Specifically, each task τ_i releases a job (at time 0 for notational brevity) with the following properties:

- $C_{i,1}$ is the execution time of the first non-critical section of the job.
- $A_{i,1}$ is the execution time of the (first) non-nested critical section of the job, in which a binary semaphore or a mutex $\sigma(\tau_{i,1})$ is used to control the access to the critical section.
- $C_{i,2}$ is the execution time of the second non-critical section of the job.

A sub-job is a critical section or a non-critical section. Therefore, each job of task τ_i has three sub-jobs. We assume the task set $\mathbf{T}$ is given and a constrained deadline is considered, i.e., $D_i \leq T_i$. We also make the following assumptions:

- For each task τ_i in $\mathbf{T}$, $C_{i,1} \geq 0$, $C_{i,2} \geq 0$, and $A_{i,1} \geq 0$.
- The execution of the critical sections guarded by one binary semaphore s must be sequentially executed under a total order. That is, if two tasks share the same semaphore, their critical sections must be executed one after another without any interleaving.
- The execution of a job cannot be parallelized, i.e., a job must be sequentially executed in the order of $C_{i,1}, A_{i,1}, C_{i,2}$.
- There are in total Z binary semaphores.

The dependency graph approach consists of the following two steps:

- In the first step, a directed graph $G = (V, E)$ is constructed. A subjob (i.e., a critical or a non-critical section) is a vertex in V and the edges in E describe the precedence constraints of these jobs. The subjob $C_{i,1}$ is a predecessor of the subjob $A_{i,1}$, and $A_{i,1}$ is a predecessor of the subjob $C_{i,2}$. If two jobs of τ_i and τ_j share the same binary semaphore, i.e., $\sigma(\tau_{i,1}) = \sigma(\tau_{j,1})$, then either the subjob $A_{i,1}$ is the predecessor of $A_{j,1}$ or the subjob $A_{j,1}$ is the predecessor of $A_{i,1}$. All the critical sections guarded by a binary semaphore form a chain in G, i.e., the critical sections of the same binary semaphore follow a total order. Therefore, we have the following properties in set E:
  - The two directed edges $(C_{i,1}, A_{i,1})$ and $(A_{i,1}, C_{i,2})$ are in E.
  - Suppose that $\mathbf{T}_k$ is the set of tasks that require the same binary semaphore s_k. Then, the $|\mathbf{T}_k|$ tasks in $\mathbf{T}_k$ follow a certain total order π such that $(A_{i,1}, A_{j,1})$ is a directed edge in E when $\pi(\tau_i) = \pi(\tau_j) - 1$.

  Figure 8.1 provides an example of a task dependency graph with one binary semaphore. Since there are Z binary semaphores in the task set, the task dependency graph G has in total Z connected subgraphs, denoted as $G_1, G_2, \ldots, G_z$. In

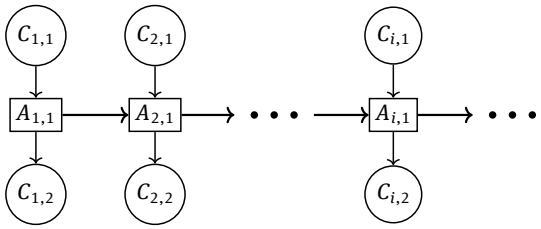

Fig. 8.1: A task dependency graph for a task set with one binary semaphore.

each connected subgraph G_ℓ, the corresponding critical sections of the tasks that request critical sections guarded by the same semaphore form a chain and have to be executed sequentially. For example, in Figure 8.1, the dependency graph forces the scheduler to execute the critical section $A_{1,1}$ prior to any of the other three critical sections.

– In the second step, a corresponding schedule of G on M processors is generated. The schedule can be based on system restrictions or user preferences, i.e., it can be based on either preemptive or non-preemptive schedules, or on either global, semi-partitioned, or partitioned schedules.

Algorithms to Construct G The objective of constructing dependency graph, i.e., G, is to minimize the makespan, i.e., the latest finishing time of all tasks, with the assumption that the number of *virtual processors* is the same as the number of tasks, based on uni-processor non-preemptive scheduling. For each task, $C_{i,1}$ is considered as release time r_i, and $C_{i,2}$ is considered as delivery time. There are several existing algorithms to derive good approximations of $G^\star$, where $G^\star$ is the graph with the optimal makespan: 1) the **extended Jackson's rule** [289], which is a polynomial-time algorithm with 2-approximation [377]; 2) the **Potts** [583], which is a polynomial-time 1.5-approximation algorithm [289]; 3) and the improvement of the approximation ratio to 4/3 by Hall and Shmoys [289].

8.1.3.2 Extension to Periodic Task Systems

To increase the applicability, we extend the DGA to handle multiprocessor synchronization for *periodic* real-time task systems. That is, we unroll the jobs of all the tasks in one hyper-period and then construct a dependency graph of these jobs. Suppose that the hyper-period H of a task set is the least common multiple (LCM) of the periods of all the tasks in this set. For each task τ_i that requests (at least) one resource, we create H/T_i jobs of task τ_i. For the ℓ-th job of task τ_i, we set its release time to $(\ell-1)T_i$ and its absolute deadline must be no later than $(\ell-1)T_i + D_i$. Since the jobs for one task should not have any execution overlap with each other, we only need one virtual processor or

dedicated shop for them, but the release time constraint is added for each job. The three methods in Section 8.1.3.1 can still be applied by adding the release time constraint for each job. Afterward, a dependency graph for all the jobs in one hyper-period is generated. In the end, the schedules are generated offline. And the generated schedules will be repeated in the upcoming hyper-periods.

Please note that such an extension can be applied to any periodic real-time task system but that it comes at the cost of space and computation, due to the increasing number of jobs in one hyper-period.

8.1.3.3 Scheduling Algorithms

In the following, we show three scheduling algorithms for the same dependency graph(s) under different system specifications.

List-EDF Here, we show how to schedule the unrolled dependency graphs over the hyper-period. For the ℓ-th job of τ_i, J_i^ℓ has three subjobs $J_{i,1}^\ell, J_{i,2}^\ell, J_{i,3}^\ell$ that represent the related subjobs $C_{i,1}, A_{i,1}, C_{i,2}$, respectively. The release time of the first subjob is $J_{i,1}^\ell$ is $(\ell-1)T_i$, and the absolute deadline of the last subjob $J_{i,3}^\ell$ is $(\ell-1)T_i + D_i$. Regarding the release times of the second and third subjob, we initially set the earliest possible time the job may be released based on the WCETs of the other subjobs. Meanwhile, regarding the deadline of the first and second subjob, we initially assign the latest possible time the subjob can finish while still allowing schedulability. To be precise, the release time of $J_{i,2}^\ell$ is set to $(\ell-1)T_i + C_{i,1}$, the release time of $J_{i,3}^\ell$ is set to $(\ell-1)T_i + C_{i,1} + A_{i,1}$, the absolute deadline of $J_{i,2}^\ell$ is set to $(\ell-1)T_i + D_i - C_{i,2}$, and the absolute time of $J_{i,1}^\ell$ is set to $(\ell-1)T_i + D_i - C_{i,2} - A_{i,1}$.

We assume that each dependency graph $\mathbf{G}_s$ for a binary semaphore s is constructed for the corresponding jobs released (strictly) within one hyper-period H. If $H_s < H$, then $\frac{H}{H_s}$ copies of $\mathbf{G}_s$ are applied in a consecutive order to represent the precedence constraints of the critical sections. For notational brevity, we denote $r_{i,j}^\ell$ as the release time of the subjob $J_{i,j}^\ell$ and $d_{i,j}^\ell$ as the absolute deadline of $J_{i,j}^\ell$. If the absolute deadline of an immediate predecessor of $J_{i,j}^\ell$, denoted as $IPre(J_{i,j}^\ell)$, is larger than $d_{i,j}^\ell$, the absolute deadline of the immediate predecessor should be reassigned to $d_{i,j}^\ell$ minus the WCET of $J_{i,j}^\ell$. This is a standard procedure for scheduling jobs subject to release dates and precedence constraints. Details can be found in [36].

We assume that the absolute deadline assignment is adjusted accordingly so that $d_{i,j}^\ell$ for the subjob $J_{i,j}^\ell$ is always greater than the absolute deadline of $IPre(J_{i,j}^\ell)$. Scheduling $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_z$ on M homogeneous (identical) processors is a special case of the classical scheduling problem $P|prec; r_j|L_{\max}$, i.e., scheduling a set of jobs with specified release times and precedence constraints on M identical processors, minimizing the maximum lateness. One possible scheduling strategy is to use the List scheduling developed by Graham [269] in combination with Earliest Deadline First scheduling (EDF). A List schedule works as follows: Whenever a processor idles and there are

subjobs eligible to be executed (i.e., all of their predecessors in the dependency graph have finished), one of the eligible subjobs is executed on the processor. If more subjobs than processors are available, we prioritize the subjobs that have the earlier absolute deadlines. If two subjobs have the same absolute deadline, the one with the larger remaining workload has a higher priority. We call this scheduling algorithm List-EDF.

Federated-Based Partitioning Algorithm Federated scheduling was proposed by Li et al. [430] in order to schedule parallel real-time task systems with internal precedence constraints that can be modeled as a Directed-Acyclic Graph (DAG). The foremost intention of this scheduling algorithm is to provide provably good approximations with respect to an optimal scheduling algorithm while considering implementation constraints, e.g., cache hit-rates and memory accesses during runtime. The idea of federated scheduling is to assign DAGs (in our case the DAGs resulting from the dependency graph construction) that need to utilize more than one processor (so-called *heavy* graphs) to those processors exclusively. Analogously, the graphs that can be feasibly scheduled on a single processor are denoted as *light* graphs and are scheduled jointly on the remaining processors, i.e., non-exclusively allocated processors. After this initial partition, the actual scheduling is done by a work-conserving scheduler on the assigned processors. If the graphs in both the *heavy* group and the *light* group can be scheduled feasibly, the corresponding partition is returned. Otherwise, there is no feasible partition.

Worst Fit-Based Heuristic In addition, a worst-fit heuristic is proposed in which the tasks are partitioned one by one. The tasks are first sorted according to a sorting strategy. After that, they are partitioned to the available processors using a worst-fit strategy, i.e., each task is assigned to the processor with the currently lowest utilization. Again, Partitioned-EDF (P-EDF) scheduling is applied to verify whether the resulting partition on M processors is feasible.

We proposed two sorting strategies: 1) sort all the tasks decreasingly with regard to the tasks' utilization, no matter which resources they request; 2) sort the graphs decreasingly with regard to the graph utilization and then sort the tasks inside each graph decreasingly with regard to the task utilization. In our proposed heuristic, both sorting strategies are applied. If the partition P generated by the first sorting strategy is not applicable, i.e., if the task set is not schedulable on M processors based on the current partition P using P-EDF, the second sorting strategy and the resulting partition $P^{'}$ are considered, and P-EDF is applied to verify the new partition $P^{'}$ once again. The algorithm only returns infeasible when both aforementioned sorting strategies cannot generate a schedulable partition. Otherwise, the task set is schedulable and the partition is returned. Again, if a time-driven schedule is created, the schedule can be returned as well.

8.1.3.4 Evaluation

We randomly generated task sets based on the number of processors M, shared resources Z, and relative utilization of the critical sections H as parameters. In our evaluation, we considered $M \in \{4, 8, 16\}$, $Z \in \{4, 8, 16\}$, and $H \in \{[5\,\% - 10\,\%], [10\,\% - 40\,\%], [40\,\% - 50\,\%]\}$.

For a given configuration of M, Z, and H, we generated task sets with $10 \times M$ tasks for each total utilization value $\sum_{\tau_i \in \mathbf{T}} U_{\tau_i} \in [0, M]$ with a step 5 %, applying the RandFixedSum method [199]. We enforced the total utilization $U_{\tau_i} \leq 0.5$ for each task τ_i. To determine the subtask utilization of one task, i.e., $U_{C_{i,1}}$, $U_{C_{i,2}}$, and $U_{A_{i,1}}$, we first decided the utilization of the critical section $U_{A_{i,1}}$ by randomly drawing a percentage of the task's total utilization U_{τ_i} based on the parameter H. Next, the remaining utilization U_{C_i} was split by drawing $U_{C_{i,1}}$ randomly uniform from $[0, U_{C_i}]$ and setting $U_{C_{i,2}}$ to $U_{C_i} - U_{C_{i,1}}$. The resource that each critical section of a task requests was selected randomly from all the available resources. In addition, we generated two kinds of task sets according to their settings of available periods:

Periodic task sets with semi-harmonic periods The task periods T_i are selected randomly from a set of semi-harmonic periods, i.e., $T_i \in \{1, 2, 5, 10\}$, which is a subset of the periods used in automotive systems [86, 290, 392, 606, 662].

Frame-based task sets As a special case of periodic task sets, all the tasks have the common period 1. Hence, i.e., $C_{i,1} = U_{C_{i,1}}$, $A_{i,1} = U_{A_{i,1}}$, and $C_{i,2} = U_{C_{i,2}}$.

For each of these setting of periods, 54 configurations are considered in total. For each of the utilization step values, 1000 task sets were randomly generated.

Evaluated Approaches To construct the dependency graphs, `POTTS` [583] is applied. Other evaluated methods to schedule the tasks sets were: 1) `FED-P-EDF`: the algorithm based on federated scheduling; 2) `WF-P-EDF`: the algorithm based on global worst-fit partitioning; 3) `LIST-EDF`: the List schedule based approach; 4) `ROP-FP`: Resource-Oriented Partitioned under Fixed-Priority [82]; 5) `ROP-EDF`: ROP under Dynamic-priority; 6) `LP-GFP-FMLP` [58]; 7) `LP-GFP-PIP` [194]; and 8) `GS-MSRP` [704].

Evaluation Results Only a subset of the results is presented, as the other results show similar trends. The evaluation results for periodic task systems are shown in Figure 8.2. If the workload of the critical sections is increased (Figure 8.2-(a) to (c)), the performance of all methods is reduced, and the difference between methods is decreased as well. The reason is that, when $\beta = [40\,\% - 50\,\%]$, the execution time of the critical section for tasks with period 10 time units can be large, i.e., longer than 2 time units. Therefore, tasks with period 1 time unit directly miss the deadline by default for all other approaches, no matter what kind of partitioning algorithm is applied. The performance drops down quickly when the utilization is increased and the critical section workload is large, as shown in Figure 8.2 (c).

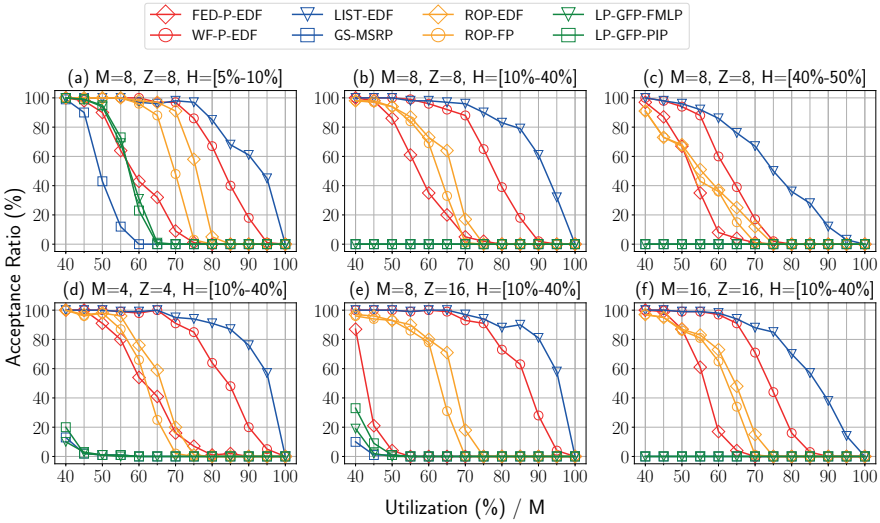

Fig. 8.2: Schedulability of different approaches for periodic task sets.

The evaluation results for frame-based task systems are shown in Figure 8.3. The proposed worst-fit heuristic `WF-P-EDF` outperforms `ROP-EDF` and other partitioned scheduling methods significantly. Furthermore, Figure 8.3 shows that `WF-P-EDF` has a good performance compared with `LIST-EDF`. In most cases, both `LIST-EDF` and `WF-P-EDF` can reach a 100 % acceptance ratio even with a 95 % utilization per processor.

8.1.4 Offloading Protocols for Unreliable Connection

In this subsection, two offloading protocols are presented in detail, addressing two system requirements: 1) the *service protocol*, which provides as much service for non-critical tasks as possible at any point in time, and 2) the *return protocol*, which allows a fast return to normal system behavior in the case of an unsuccessful offloading operation.

8.1.4.1 System Model

We consider a cyber-physical system comprising a set of tasks $\mathcal{T}$ that can be divided into two subsets with different requirements, namely, the set of *critical* tasks $\mathcal{T}_{crit}$, and the set of *non-critical* tasks $\mathcal{T}_{non}$, such that $\mathcal{T} = \mathcal{T}_{crit} \cup \mathcal{T}_{non}$ and $\mathcal{T}_{crit} \cap \mathcal{T}_{non} = \emptyset$. While for each $\tau_k \in \mathcal{T}_{crit}$ timing constraints must be satisfied at any point in time, for each $\tau_k \in \mathcal{T}_{non}$ timing violations may be unpleasant but not hazardous. According to the classification of tasks into two subsets, we specify two different system execution behaviors, i.e., *normal* and *local* execution behavior. When the system exhibits normal

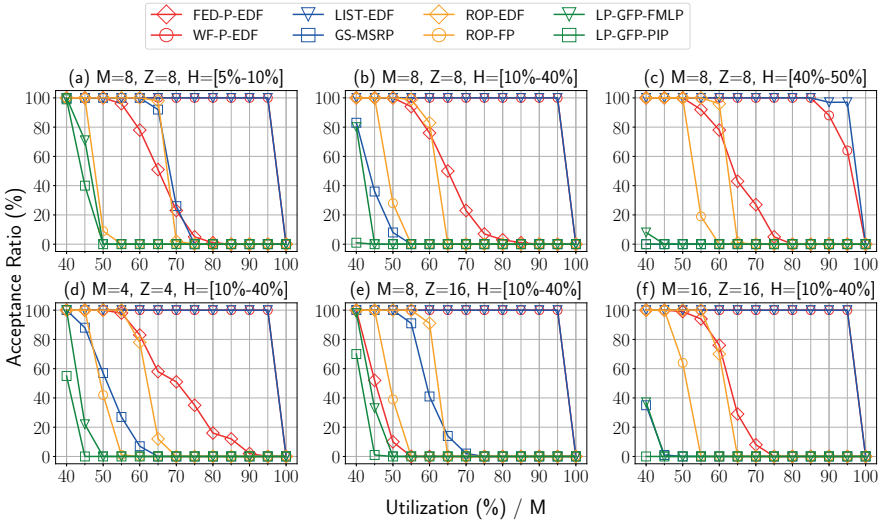

Fig. 8.3: Schedulability of different approaches for frame-based task sets(1).

execution behavior, all timing requirements of all tasks are satisfied at any point in time, whereas, if the system exhibits local execution behavior, timing guarantees can only be given for all critical tasks $\tau_k \in \mathcal{T}_{crit}$.

Each recurrent real-time task $\tau_k \in \mathcal{T}$ is assumed to have a sporadic arrival pattern and is characterized by a tuple $\left(C_{k,1}, C_{k,s}, C_{k,2}, S_k, p_k, q_k, D_k, T_k\right)$:

- Each τ_k releases an infinite number of task instances denoted as *jobs*. T_k indicates the minimum inter-arrival time of τ_k.
- D_k describes the relative deadline of τ_k. A constrained-deadline task system is considered, in which $D_k \leq T_k$ for each task τ_k.
- $C_{k,1}$ and $C_{k,2}$ denote the WCETs of the first and second *computation segments*, respectively.
- $C_{k,s}$ is the WCET of the typically offloaded task share if executed on the *local* system.
- p_k and q_k are the WCETs of the pre- and post-processing routines, which are executed before and after the offloading operation of a job of task τ_k, respectively.
- S_k is the offloading or *suspension* time of τ_k.

We assume that $T_k \geq D_k > 0$ and $C_{k,1}, C_{k,s}, C_{k,2}, S_k, p_k, q_k \geq 0$. Moreover, we assume that WCET of pre- and post-processing routines are less than or equal to the WCET of local execution, i.e., $p_k + q_k \leq C_{k,s}$. Furthermore, the WCET of a job of task τ_k under any possible execution scenario is greater than 0, i.e., $C_{k,1} + C_{k,s} + C_{k,2} > 0$ and $C_{k,1} + p_k + q_k + C_{k,2} > 0$. For notational brevity, we denote $C_k^\sharp = C_{k,1} + C_{k,s} + C_{k,2}$ and $C_k^\flat = C_{k,1} + p_k + q_k + C_{k,2}$.

In addition, we assume that the local cyber-physical real-time system, termed *local system*, is a uniprocessor system, in which tasks are scheduled according to a preemptive

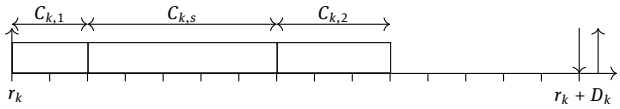

Fig. 8.4: A job of task τ_k is executed locally (local execution behavior).

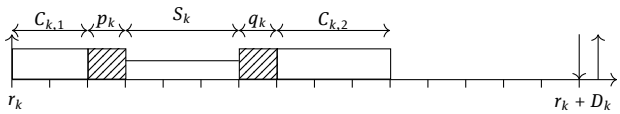

Fig. 8.5: An offloading operation of a job of task τ_k is performed successfully (normal execution behavior).

fixed-priority policy. More precisely, each task is assigned a unique priority, i.e., all jobs of task τ_k have the same priority. If at any point in time multiple jobs are ready, i.e., eligible for being executed on the local system, the job having the highest priority is executed. For each task τ_k, the unique set of the higher-priority tasks is denoted as $hp(\tau_k)$.

For a job of task τ_k arriving at time r_k the following execution scenarios are possible:

- The job is *executed locally* (Figure 8.4). In this case, the WCET of the job released at time r_k is $C_{k,1} + C_{k,s} + C_{k,2}$, i.e., $C_k^\sharp$.
- The job is *offloaded*. In this case, the job is first executed locally for up to $C_{k,1}$ execution time units and thereon enters the pre-processing routine for up to p_k execution time units. Suppose that the first computation segment as well as the pre-processing routine are finished at time ρ. Then, the considered job is offloaded to the remote system at time ρ. The actual offloading operation can be either *successful* or *unsuccessful*:
  - *Offloading is successful* if the computation result or *offloading response* is returned to the local system until time $\rho + S_k$. In this case, the offloading response is post-processed for up to q_k time units and the second computation segment is executed for up to $C_{k,2}$ time units (Figure 8.5). Accordingly, the execution time of the job of τ_k on the local system is at most $C_k^\flat$.
  - *Offloading is unsuccessful* otherwise. In this case, at time $\rho + S_k$, a local re-execution of the offloaded task share is performed for up to $C_{k,s}$ time units followed by the execution of the second computation segment for up to $C_{k,2}$ time units. In this case, the execution time of the job of τ_k on the local system is at most $C_k^\sharp + p_k$.

8.1.4.2 Recovery Protocols

Cyber-physical systems are applied throughout a broad range of areas, each exhibiting individual requirements and thus a need for situationally appropriate system behav-

ior. For safety-critical cyber-physical systems, the timeliness of critical tasks must be guaranteed under any circumstances - even in the event of an unsuccessful offloading operation. Since in this case a larger amount of local resources is required, less resources remain to serve the non-critical tasks, as we explained in Section 8.1.4.1. However, depending on the actual system characteristics, timing constraints for non-critical tasks tend to be less strict. For instance, it is possible that a non-critical task misses its deadline, but that the results are still useful up to a certain degree [83, 87]. Nevertheless, it may be desirable to return to the normal execution behavior and to re-establish timing guarantees for both critical and non-critical tasks as soon as possible, especially since a non-critical task is not necessarily unimportant and thus should provide functionally and temporally correct results most of the time. Further discussion on the relation between criticality and importance can be found in [204].

Against this backdrop, we propose two recovery protocols allowing the system to satisfy its requirements under local execution behavior and to return to normal execution behavior:

- The *service protocol* aims to provide as much service as possible for non-critical tasks, even under local execution behavior.
- The *return protocol* aims to minimize the amount of time, in which the system exhibits local execution behavior after an unsuccessful offloading operation.

Independent of the actual protocol, we assume that the local system exhibits normal execution behavior at time 0, such that offloading is enabled for all tasks in $\mathcal{T}$. The schedule considers the execution of all tasks until the first moment $\gamma_{1,\searrow}$ in which the offloading operation of a certain task τ_k is unsuccessful. That is, a job of task τ_k, which has offloaded its computation at time $\gamma_{1,\searrow} - S_k$, does not receive the offloading response until time $\gamma_{1,\searrow}$ (Figure 8.6). Immediately after $\gamma_{1,\searrow}$, the local system exhibits local execution behavior. Until time $\gamma_{1,\searrow}$, three scenarios are possible for each incomplete job of all critical tasks τ_i in $\mathcal{T}_{crit}$:

- *The job of τ_i has not been offloaded*: In this case, no offloading operation will be performed for this job, but it is executed locally instead. Since it is possible that the pre-processing routine for offloading is already active at time $\gamma_{1,\searrow}$, the WCET of this job is upper-bounded by $C_{i,1} + p_i + C_{i,s} + C_{i,2}$, i.e., $C_i^\sharp + p_i$.
- *The job of τ_i is already offloaded, but no offloading response was received until time $\gamma_{1,\searrow}$*: In this case, the offloading process is aborted and the job is executed locally as of time $\gamma_{1,\searrow}$. Therefore, the WCET of this job is upper-bounded by $C_{i,1}+p_i+C_{i,s}+C_{i,2}$, i.e., $C_i^\sharp + p_i$.
- *The job of τ_i is already offloaded and the offloading response has been received prior to time $\gamma_{1,\searrow}$*: In this case, the job continues its final processing. Therefore, the WCET of this job is upper-bounded by $C_{i,1} + p_i + q_i + C_{i,2}$, i.e., $C_i^\flat$.

After $\gamma_{1,\searrow}$, timing guarantees are provided only for $\mathcal{T}_{crit}$. Moreover, offloading is inhibited for all critical tasks in the near future of $\gamma_{1,\searrow}$ due to the currently unreliable

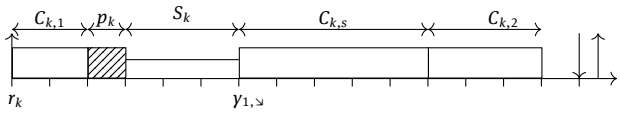

Fig. 8.6: An unsuccessful offloading operation of τ_k resulting in the transition to the local system behavior at time $\gamma_{1,\searrow}$.

connection leading to the missing offloading response. The offloading decision for non-critical tasks, however, depends on the applied recovery protocol:

Service Protocol Under the *service protocol*, offloading is inhibited for all instances of all tasks that are active as long as the system exhibits local execution behavior. The task share of each $\tau_i \in \mathcal{T}$ that is offloaded under normal execution behavior is executed locally within $C_{i,s}$ units of execution time. Since this leads to a higher workload on the local system, timeliness cannot be guaranteed for any non-critical task. Nevertheless, no non-critical task is aborted.

Return Protocol The *return protocol* does not inhibit offloading for *all* tasks, only for critical ones under local execution behavior. Non-critical tasks, by contrast, are offloaded regardless, but neither a re-execution nor a re-transmission is performed if an offloading response is not received in time. More precisely, the second subtask of τ_i is executed only if an offloading response is received, and aborted otherwise. Moreover, a job of τ_i in $\mathcal{T}_{non}$ is aborted whenever it misses its deadline.

As of time $\gamma_{1,\searrow}$, the local system exhibits local execution behavior until the point in time $\gamma_{1,\nearrow}$, in which timing guarantees can be given again for all tasks in $\mathcal{T}$. In the proposed protocols, two options are considered for the transit from local to normal execution behavior. They should be chosen based on the actual system requirements:

Abort-Transit This option aims to re-establish the normal system execution behavior as quickly as possible. Suppose that $\gamma_{1,\nearrow}$ is the earliest moment (after $\gamma_{1,\searrow}$) in which there is no incomplete job from $\mathcal{T}_{crit}$ at $\gamma_{1,\nearrow}$. All released but not yet finished instances of non-critical tasks are discarded.

Idle-Transit This option re-establishes the normal system execution behavior at the earliest moment $\gamma_{1,\nearrow}$ (after $\gamma_{1,\searrow}$) in which there is no incomplete job from $\mathcal{T}$ at $\gamma_{1,\nearrow}$.

We note that the above transitions are well-defined and the local system exhibits normal and local execution behavior in an interleaving manner.

8.1.4.3 Evaluation

In this subsection, we perform a case study on a robotic system to compare the acceptance ratio of schedulability over different protocols. More comprehensive numerical simulations can be found in the original paper [612].

Tab. 8.1: Periodic, implicit-deadline tasks; measurements of a Robotnik RB-1 Base robot platform. Note that the frequency of task τ_{laser} is 15.5 Hz.

Task	WCET [ms]	Period [ms]
τ_{laser}	6.732	64.516
τ_{odom}	1.046	60.0
τ_{tf}	0.333	60.0

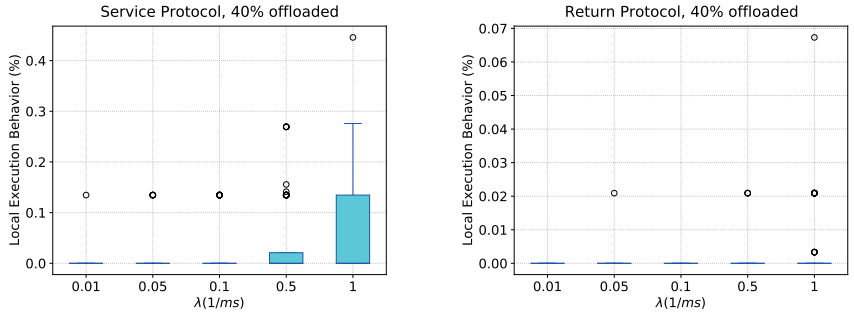

Fig. 8.7: The percentage of time the robot exhibits local execution behavior during the simulation for different probabilities of unsuccessful offloading operations and different percentages of offloaded workload under the service and the return protocol with 40 % offloaded workload per task.

Case Study on a Robotic System We adopt a Robotnik RB-1 Base robot platform [598], which uses the well-known Robot Operating System (ROS) [601]. We simulated the navigation of the robot in a virtual map and measured the timing data of the `move_base` node during a time frame of 60 seconds by using the Real-Time Scheduling Framework for ROS (ROSCH) [607] and RESCH [362]. We obtained three periodic, implicit-deadline tasks, as shown in Table 8.1, which are transformed into self-suspending tasks analogously to the tasks in experiment 1), and we considered the cases that 40 %, and 60 % of the task workload are offloaded. Moreover, we assume that $\mathcal{T}_{crit} = \{\tau_{odom}\}$ and $\mathcal{T}_{non} = \{\tau_{laser}, \tau_{tf}\}$. We simulate the system behavior using the event-based miss rate simulator from experiments 1) with $\lambda = 0.1 \cdot \frac{1}{ms}$. For each offloading case, the simulation was repeated 100 times.

Under the return protocol, Figure 8.8 shows that the amount of offloaded workload has no significant impact on the time that the system exhibits local execution behavior. Under the service protocol, we can observe that the time that the system exhibits local execution behavior is increased along with the increasing amount of offloaded workload. Overall, the derived results suggest that the amount of offloaded workload per task has a strong impact on the system execution behavior under the service protocol and thus should be taken into consideration at system design time.

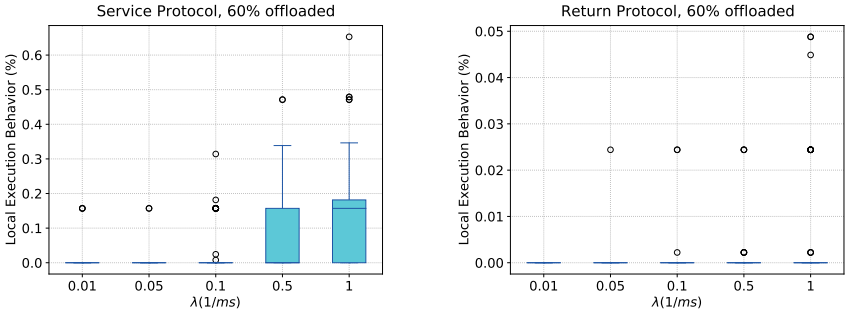

Fig. 8.8: The percentage of time the robot exhibits local execution behavior during the simulation for different probabilities of unsuccessful offloading operations and different percentages of offloaded workload under the service and the return protocol with 40 % and 60 % offloaded workload per task.

8.1.5 Probability-Based Timing Analysis

In this subsection, we present a multinomial-based approach to efficiently calculate the deadline miss probability. Additionally, three analytical approaches are presented, i.e., Chernoff bound, Hoeffding's inequality, and Bernstein's inequality.

8.1.5.1 System Model and Notation

We consider a given set of n independent periodic (or sporadic) tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ in a uniprocessor system. Each task τ_i releases an infinite number of task instances, called jobs, and is defined by a tuple $((C_{i,1}, \ldots, C_{i,h}), D_i, T_i)$, where D_i is the relative deadline of τ_i and T_i is its minimum interarrival time. In addition, each task has a set of h distinct execution modes $\mathcal{M}$ and each mode j with $j \in \{1, \ldots, h\}$ is associated with a different WCET $C_{i,j}$. We assume those execution modes to be ordered increasingly according to their WCETs, i.e., $C_{i,m} \leq C_{i,m+1}\ \forall m \in \{1, \ldots, h-1\}$. Furthermore, we assume that each job of τ_i is executed in one of those distinct execution modes. To fulfill its timing requirements in the j^{th} execution mode, a job of τ_i that is released at time t_a must be able to execute $C_{i,j}$ units of time before $t_a + D_i$. The next job of τ_i must be released at $t_a + T_i$ for a periodic task and for a sporadic task the next job is released at or after $t_a + T_i$. In this work, we focus on *implicit-deadline* task sets, i.e., $D_i = T_i$ for all tasks, and *constrained-deadline* task sets, i.e., $D_i \leq T_i$ for all tasks. We assume that a job execution is aborted as soon as the absolute deadline is reached, to ensure that there is no 'domino effect' to jeopardize the execution of the other jobs.

We assume a preemptive fixed-priority scheduling policy is used in the considered system. The tasks are indexed according to their priority, i.e., τ_1 has the highest and τ_n has the lowest priority. In addition, $hp(\tau_k)$ denotes the set of tasks with higher priority than τ_k and $hep(\tau_k)$ is $hp(\tau_k) \cup \{\tau_k\}$. $\mathbb{P}_i(j)$ denotes the probability that a job of task τ_i is executed in mode j with related WCET $C_{i,j}$ and we assume that each job is executed

in exactly one of these distinct execution modes, i.e., $\sum_{j=1}^{h} \mathbb{P}_i(j) = 1$. In addition, we assume that these probabilities are independent from each other according to the following definition:

Definition 27 (Independent Random Variables)**.** *Two random variables are (probabilistically) independent if the realization of one does not have any impact on the probability of the other.*

Particularly, for a newly arriving job the probability of the execution modes is independent from the execution mode of the jobs of previous jobs.

8.1.5.2 Definition of Deadline Miss Probability

To derive the probability of deadline misses, we look for the probability that the accumulated workload S_t over an interval of length t is at most t, where S_t can be calculated by the sum of random variables, i.e., the sum of probabilistic WCETs from all tasks $\tau_i \in hep(\tau_k)$ over. That is, the situation where S_t is larger than t for an interval of length t and hence $\mathbb{P}(S_t > t)$ is the overload probability at time t. To upper bound the probability that this test fails, the minimum probability among all time points at which the test could fail should be derived. Hence, the probability of a deadline miss Φ_k can be upper bounded by

$$\Phi_k = \min_{0<t\leq D_k} \mathbb{P}(S_t > t) \tag{8.1}$$

When analytical bounds are in use, we seek $\mathbb{P}(S_t \geq t)$ instead of $\mathbb{P}(S_t > t)$. By definition $\mathbb{P}(S_t \geq t) \geq \mathbb{P}(S_t > t)$, so these values can be used directly when looking for an upper bound of $\mathbb{P}(S_t > t)$.

8.1.5.3 A Multinomial-Based Approach

Conventionally, the probability of deadline misses can be derived from convolution-based approaches [476]. In such approaches, the underlying random variable represents the execution mode of each single job. This state space in fact can be transformed into an equivalent space that describes the states on a task-based level by proving the invariance when considering equivalence classes for each task. As a result, we introduce a novel approach that is based on the multinomial distribution. For the simplicity of presentation, we only highlight the insight of the aforementioned transformation.
The traditional convolution-based approach determines the *overload probability* by successively calculating the probability for all other points of interest in the analysis interval. However, the probability for t is evaluated based on the resulting states after all jobs in the analysis interval are convoluted. With respect to t, the intermediate states are not considered. By utilizing this insight, we can merge the states to efficiently calculate the vector representing the possible states at time t. If the number of jobs for a task is known, all possible combinations and the related probabilities can be calculated

directly using the multinomial distribution. The rationale is to construct a tree based on the tasks, which means that the number of children on each level depends on the number of jobs the related task releases.

8.1.5.4 Analytical Upper Bounds

In the following, we demonstrate how common concentration inequalities used in machine learning, statistics, and discrete-mathematics can be used to derive analytical bounds on $\mathbb{P}(S_t \geq t)$.

Chernoff Bound can be exploited to over-approximate the probability that a random variable exceeds a given value. This statement is summarized in the following lemma:

Lemma 28 (Lemma 1 from Chen and Chen [131]). *Suppose that S_t is the sum of the execution times of the $\rho_{k,t} + \sum_{\tau_i \in hp(\tau_k)} \rho_{i,t}$ jobs in $hep(\tau_k)$ at time t. In this case*

$$\mathbb{P}(S_t \geq t) \leq min_{s>0} \left(\frac{\prod_{\tau_i \in hep(\tau_k)} (mgf_i(s))^{\rho_{i,t}}}{\exp(s \cdot t)} \right) \tag{8.2}$$

It is in general pessimistic and there is no guarantee for the quality of the approximation. To find the optimal value of s to minimize the right-hand side in Equation 8.2, it has been proven as a log-convex optimization problem [129].

Hoeffding's Inequality derives the targeted probability that the sum of independent random variables exceeds a given value. For completeness, we present the original theorem here:

Theorem 29 (Theorem 2 from [319]). *Suppose that we are given M independent random variables, i.e., $X_1, X_2, \ldots, X_M$. Let $S = \sum_{i=1}^{M} X_i$, $\bar{X} = S/M$ and $\mu = \mathbb{E}[\bar{X}] = \mathbb{E}[S/M]$. If $a_i \leq X_i \leq b_i$, $i = 1, 2, \ldots, M$, then for $s > 0$,*

$$\mathbb{P}(\bar{X} - \mu \geq s) \leq \exp\left(-\frac{2M^2 s^2}{\sum_{i=1}^{M} (b_i - a_i)^2} \right) \tag{8.3}$$

Let $s' = sM$, i.e, $s = s'/M$. Hoeffding's Inequality can also be stated with respect to S:

$$\mathbb{P}(S - \mathbb{E}[S] \geq s') \leq \exp\left(-\frac{2s'^2}{\sum_{i=1}^{M} (b_i - a_i)^2} \right) \tag{8.4}$$

By adopting Theorem 29, we can derive the probability that the sum of the execution times of the jobs in $hep(\tau_k)$ from time 0 to time t is no less than t. The detailed proof can be found in [85].

Bernstein's Inequality generalizes the Chernoff bound and the related inequality by Hoeffding and Azuma. The original corollary is also stated here:

Theorem 30 (Corollary 7.31 from [232]). *Suppose that we are given L independent random variables, i.e., $X_1, X_2, \ldots, X_L$, each with zero mean, such that $|X_i| \leq K$ almost surely for $i = 1, 2, \ldots, L$ and some constant $K > 0$. Let $S = \sum_{i=1}^{L} X_i$. Furthermore, assume that $\mathbb{E}[X_i^2] \leq \theta_i^2$ for a constant $\theta_i > 0$. Then for $s > 0$,*

$$\mathbb{P}(S \geq s) \leq \exp\left(-\frac{s^2/2}{\sum_{i=1}^{L} \theta_i^2 + Ks/3}\right) \tag{8.5}$$

The proof can be found in [232]. Note, however, that the result in [232] is stated for the two-sided inequality, i.e., as upper bound on $\mathbb{P}(|S| \geq s)$. Here, the one-sided result, which is a direct consequence of the proof in [232] (page 198), is tighter. Similarly, it can also be used to derive the probability of deadline misses. The detailed proof can also be found in [85].

Final Remark Considering the required runtime and the accuracy of different approaches, when a given task set needs to be analyzed, we suggest first running *Chernoff's*, *Hoeffding's*, and *Bernstein's* bounds. If a sufficiently low deadline miss probability cannot be guaranteed from these analytical bounds, we propose running the multinomial-based approach with equivalence class union in parallel on multiple machines by partitioning the time points equally.

8.1.6 Summary

In this section, we showed a novel resource-sharing protocol for multiprocessors, named DGA, that can serve a high utilization of critical sections while guaranteeing the given hard real-time constraints. In addition, we presented adaptive protocols for computation offloading that are able to countermeasure the unreliable connection. Unlike conventional analyses for hard real-time systems, our innovated convolution-based approach is able to efficiently derive safe upper bounds for the probability of deadline misses under soft real-time constraints.

8.2 Communication Architecture for Heterogeneous Hardware

Henning Funke
Jens Teubner

Abstract: In this section, we look at distributed processing on a smaller scale. Even a single-machine system today internally looks—and behaves—like a distributed system: multiple *processing modules* of different flavors (*e.g.*, CPUs, GPUs, FPGAs) interact with *memory modules*, which are scattered over the system, through an *interconnect* that is comprised of, say, QPI, PCIe, or "real" network links. In such environments, *communication* quickly becomes the limiting factor—not only for observable performance, but also for other system aspects, such as energy consumption.

We specifically look at communication patterns in heterogeneous CPU/GPU environments, and we illustrate how novel processing models can minimize communication overhead in such systems, which in turn results in significant performance improvements for real-world settings.

In GPU-accelerated, data-intensive systems, the PCIe link is often perceived as the limiting factor. Oftentimes, successions of fine-granular GPU kernel invocations amplify the problem, since they tend to cause multiple round-trips over the bandwidth-limited link. As we see in this section, unnecessary round-trips can be avoided by fusing fine-granular kernels into larger work units that can hide costly PCIe transfer times (query compilation can be a device to implement kernel fusion).

Eliminating the PCIe bottleneck, however, only exposes the GPU's *on-chip communication* as the new bottleneck to GPU-assisted data processing. Here, the data-parallel processing modes of graphics processors and the synchronization between parallel units are the cause for redundant round-trips over the precious on-chip communication interfaces. These bottlenecks can be avoided when processing models are deliberately designed to be *communication aware*. A specific example is a novel combination of pipelining/streaming processing models with the data-parallel nature of GPUs, which aligns particularly well with the semantics of database query execution. For real-world settings, this results in a reduction of memory access volumes by factors of up to 7.5× and shorter GPU kernel execution times by factors of up to 9.5×.

8.2.1 Introduction

Graphics Processing Units (GPUs) are frequently used as powerful accelerators for database query processing. As the arithmetic throughput of the coprocessor peaks in the teraflop range, it becomes a challenge to provision enough data. For this reason,

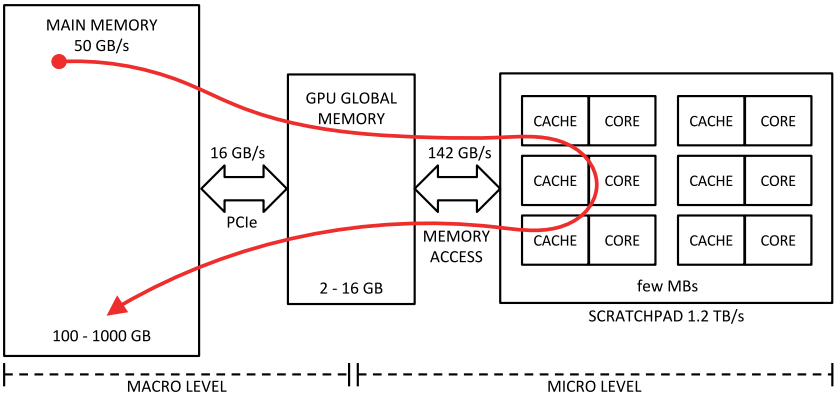

Fig. 8.9: The path of a tuple through the memory levels of a coprocessor environment.

hardware vendors equip graphics cards with high-bandwidth memory that has read and write rates of hundreds of GB/s. Still, memory intensive applications such as query processing fall behind regarding the cost of data movement for different reasons. Figure 8.9 shows the path of relational data through the hierarchical memory levels in a typical coprocessor system. Along the path, several bandwidth and capacity constraints need to be considered to achieve scalability and performance:

PCIe / OpenCAPI / NVLink A widely-acknowledged problem is the data transfer bottleneck between the host system and the coprocessor [270], typically via PCIe. Due to the coprocessor's limited memory capacity, data transfers are necessary *during* computations. With an order of magnitude between internal and external memory bandwidth, database developers are challenged with data locality-aware algorithms that efficiently use inter-processor communication. Recent technologies, i.e., OpenCAPI and NVLink, increase the bandwidth over PCIe, shifting the bottleneck toward GPU global memory.

GPU Global Memory The fine-grained data parallelism of a GPU typically requires that kernels perform additional passes over the data. Performing multiple passes, however, can significantly inflate memory loads and can cause a bandwidth bottleneck especially for random memory accesses.

Main Memory Integrated GPU-style coprocessors are a recent development to directly access the memory of the host CPU. Such an *Accelerated Processing Unit (APU)* allows the use of massively parallel processing without additional data transfers. However, the available memory bandwidth is lower than that of a dedicated GPU (30 GB/s vs. hundreds of GB/s).

Scratchpad Memory[1] Scratchpad memory is located on-chip and placed next to each compute unit of a GPU. It can be controlled as an explicit cache for low-level computations and offers a very high bandwidth. However, the capacity is limited to 16 kB–96 kB per core, which makes it challenging to use it for large-scale computations.

8.2.2 Contributions

With *HorseQC*, we developed a database query compiler that accounts for the hierarchical memory structure of modern coprocessor environments and for their inherent bandwidth limitations. In this section, we elaborate on the key building blocks of *HorseQC*, which can serve as a poster child in bandwidth-aware systems for other application contexts as well.

Specifically, we *(a) analyze the bandwidth limitations* in different database execution models; *(b)* demonstrate a *query compiler* for a coprocessor-accelerated database engine; *(c)* show how database sub-tasks can be realized in a *single pass over the data* (thus avoiding expensive memory round-trips); and *(d)* integrate these contributions in a *fully working system* that we use to evaluate our work.

Coprocessor-enabled database engines are typically classified by the *macro execution model* that they use to orchestrate the processing of query plans. Orthogonally, we devise a *micro execution model* that can be paired with different existing macro execution models, enhancing their communication- and resource-awareness.

8.2.3 Macro Execution Model

We begin by looking at macro execution models that have been employed in the past. To evaluate a relational query operator, state-of-the-art systems will select a number of primitives and execute the corresponding kernel sequence on the GPU. To feed the kernels with data, the macro execution model defines how data transfers will be interleaved with kernel executions. Here, the data movement from kernel to kernel may result in additional bandwidth demand compared with conventional systems. To understand the effect, we study the implications that existing macro execution models have on the use of bandwidth at multiple levels (PCIe, GPU global memory, etc.). We profiled the execution of Query 3.1 as a poster child from the star schema benchmark (SSB) [543]. The query was executed at scale factor 10 with CoGaDB [74] on a NVIDIA

1 We use the term *scratchpad memory* to disambiguate *shared memory* for CUDA and *local memory* for OpenCL.

GTX970 GPU.[2] In the following, we discuss three macro execution models: *run-to-finish*, *kernel-at-a-time*, and *batch processing*.

Algorithm 8: Run-to-finish execution of two successive kernels.

```
1 RUN-TO-FINISH – input: R, output: P
2 move R Host → GPU
3 tmp ← op1(R)                          /* invoke first GPU kernel */
4 P ← op2(tmp)                          /* invoke second GPU kernel */
5 move P GPU → Host
```

8.2.3.1 Run-To-Finish (Not Scalable)

A straightforward way to execute a sequence of kernels is to first transfer all input, execute the kernels, and finally transfer all output. The approach, illustrated in Algorithm 8, has the advantage that intermediate data remains in GPU global memory in-between kernel executions and no significant PCIe transfers are necessary. However, run-to-finish has the disadvantage that it works only if *all* input, output, and intermediate data is small enough to fit in GPU memory. Run-to-finish macro execution models are used, e.g., by Ocelot [302], CoGaDB [74], and others. The *lack of scalability* leads us to evaluate the following execution models.

Algorithm 9: Kernel-at-a-time achieves scalability by transferring I/O for each kernel through PCIe.

```
1 KERNEL-AT-A-TIME – input: R, output: P
2 foreach r_i in R=r_1 ∪ ··· ∪ r_m do
3 |   move r_i Host → GPU
4 |   m_i ← op1(r_i)                    /* invoke first GPU kernel */
5 |   move m_i GPU → Host (assemble into M)
6 foreach m_j in M=m_1 ∪ ··· ∪ m_n do
7 |   move m_j Host → GPU
8 |   p_j ← op2(m_j)                    /* invoke second GPU kernel */
9 |   move p_j GPU → Host (assemble into P)
```

8.2.3.2 Kernel-At-A-Time

To process large data on coprocessors, we can execute each kernel on blocks of data. The pseudocode of this approach is shown in Algorithm 9. Processing blocks of data requires algorithm choices that can deal with partitioned inputs. Joins or aggregations, for instance, can be processed in this mode only if their internal state (e.g. a hash table) can fit in GPU global memory.

2 We measured 146.1 GB/s GPU global memory bandwidth in a host system with 16 GB/s bidirectional PCIe bandwidth.

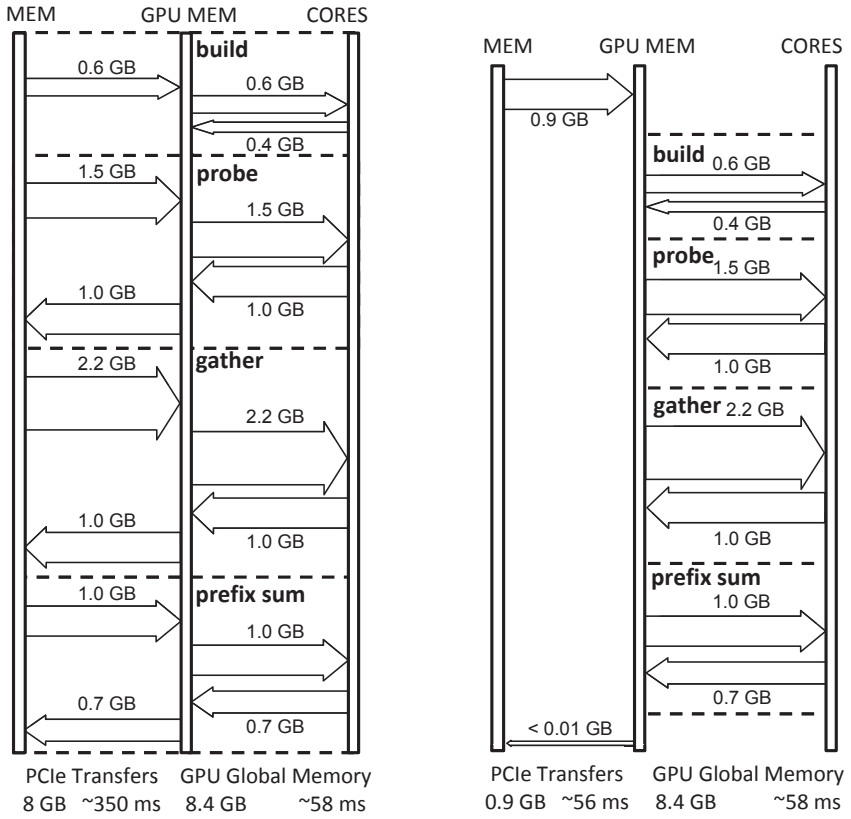

Fig. 8.10: Data movement for processing SSB Query 3.1. While the throughput of a is limited by PCIe transfers, b exposes GPU global memory access as the next bottleneck.

We analyze the data movement of kernel-at-a-time for SSB Query 3.1. Blocks are first moved via PCIe from the host to the coprocessor and then read by the kernel from GPU global memory (output passes both levels vice-versa). In this way, the data volumes for GPU global memory accesses equal the data volume transferred via PCIe, plus the cost to build up the hash tables in GPU global memory (0.4 GB here). Figure 8.10a shows the resulting data movement.

In the figure, the arrows annotated with data volumes represent PCIe transfers and GPU global memory accesses. We aggregated the data volumes by kernel types (e.g. scan, gather) and show only the most important kernels responsible for 88.2% of the memory traffic. Given a PCIe bandwidth of 16 GB/s, all PCIe transfers together require at least 350 ms to complete. This exceeds the aggregate time for GPU global memory access by a factor of 5.8×. For kernel-at-a-time processing *the PCIe link is clearly the bottleneck*.

Kernel-at-a-time processing is used to scale out individual operators [358]. Unified Virtual Addressing (UVA) produces the same low-level access pattern, though it is transparent to the system developer.

8.2.3.3 Batch Processing

We can alleviate PCIe bandwidth limitations by rearranging the operations of kernel-at-a-time. Instead of running kernels until a column is processed, we can short-circuit the transfer of intermediate results to the host. Batch processing achieves this by reusing the output of the previous operation (op1) as input for the next operation (op2) instead of transferring to the host. This is applicable whenever intermediate batch results can be kept within GPU global memory. The corresponding pseudocode is shown in Algorithm 10.

Algorithm 10: Batch processing executes multiple kernels for each block that is transferred via PCIe.

```
1 BATCH PROCESSING – input: R, output: P
2 foreach r_i in R=r_1 ∪ ··· ∪ r_m do
3 |   move r_i Host → GPU
4 |   tmp_i ← op1(r_i)                    /* invoke first GPU kernel */
5 |   p_i ← op2(tmp_i)                    /* invoke second GPU kernel */
6 |   move p_i GPU → Host (assemble into P)
```

We analyze the data movement cost with the example of SSB Query 3.1. The GPU global memory load is the same as for kernel-at-a-time processing, because each kernel reads and writes I/O to GPU global memory. We obtain the PCIe transfer cost using the transfer volumes of the input columns of the query and the output of the final result. Figure 8.10b shows the resulting data movement cost. Batch processing reduces the amount of PCIe transfers by a factor of 8.8×. This shows that transferring data in blocks *and* performing multiple operators per block allows scalability and increases the efficiency compared to kernel-at-a-time.

Batch processing macro execution models have been used for coprocessing by GPUDB [728] and Hetero-DB [735]. Wu et al. [711] described the concept as *kernel fission* and identify opportunities to omit PCIe transfers automatically.

Limitations The lower amount of PCIe traffic can expose GPU global memory bandwidth as the next limitation. Batch processing reduces the PCIe transfer cost, but the amount of GPU global memory accesses remains unaffected. The memory access volume inside the device is now an order of magnitude larger. Despite the high bandwidth, this takes longer to process than the PCIe bus transfers (Figure 8.10b). For this reason, batch processing SSB Query 3.1 is *not* limited by PCIe transfers, but by accesses to the (high-speed) GPU global memory. Since in typical query plans, I/O and hashing opera-

tions both address the same GPU global memory, the situation may arise frequently in real-world workloads.

Tab. 8.2: Number of passes over the input data for benchmark queries. Out of 25 queries, 9 are definitely limited by GPU global memory.

Query	Passes	Query	Passes	Query	Passes
ssb11	7.5	ssb34	2.2	tpch5	7.2
ssb12	6.9	ssb41	7.4	tpch6	6.2
ssb13	6.7	ssb42	3.9	tpch7	**9.0**
ssb21	**9.6**	ssb43	3.5	tpch9	**9.0**
ssb22	**9.2**	tpch1	**15.5**	tpch10	5.8
ssb23	**9.1**	tpch2	**14.5**	tpch15	6.3
ssb31	**11.0**	tpch3	5.2	tpch18	**38.5**
ssb32	7.9	tpch4	6.6	tpch20	**10.5**
ssb33	7.5				

8.2.4 Micro Execution Model

Tuning the macro level helps to remove the main bottleneck for scalability: data transfers over PCIe. However, the macro level change exposes a new bottleneck: the memory bandwidth of GPU global memory. To utilize the GPU global memory bandwidth more efficiently, we need to apply additional micro-level optimizations using *micro execution models* and combine them with the macro execution model (batch processing) to achieve scalability *and* performance.

Existing micro-level optimizations such as *vector-at-a-time* processing [749] and *query compilation* [529] utilize memory bandwidth more efficiently by leveraging pipelining in on-chip processor caches. Therefore, both techniques are promising candidates for opening up the bottleneck of limited GPU global memory bandwidth. However, vector-at-a-time processing and query compilation are designed in the context of CPUs. While it is highly desirable to apply both techniques in the context of GPUs, mapping the techniques from CPU to GPU is challenging, as we discuss below.

Vector-At-A-Time To mediate the interpretation overhead of Volcano and the materialization overhead of operator-at-a-time, vector-at-a-time uses batches that fit in the processor caches. First, this reduces the number of `getNext()` calls from one per tuple to one per batch. Second, this makes materialization cheap because operators pick up the cached results of previous operators. On CPUs, vector-at-a-time benefits from batch sizes that are large enough to limit the function call overhead and small enough to fit in the CPU caches.

On GPUs, the compromise between tuple-at-a-time and full materialization strategies is not a sweet spot, however. Kernel invocations are an order of magnitude more expensive than CPU function calls. Furthermore, GPUs need much larger batch sizes to facilitate over-subscription and out-of-order execution. This leads to the problem that batches, which fit in the GPU caches, are too small to be processed efficiently. Alternatively, more recent GPUs support *pipes* to move a local execution context from one kernel to another. This has been used by GPL [557] for query processing. However, this technique still introduces an overhead for switching the execution context. In addition, it is limited to a depth of 2–32 kernels depending on the microarchitecture.

Query Compilation Query compilation is a common tool for avoiding excessive memory transfers during query processing. Compiling code for incoming queries becomes feasible with low-level code generation and achieves performance close to hand-written code. The compilation strategy of Neumann [529] keeps intermediate results in CPU registers and passes data between operators without accessing memory at all. The generated code processes full relations or blocks of tuples using a sequential tight loop.

To use query compilation on GPUs, we must integrate fine-grained data parallelism into compiled queries. The parallelization strategy of HyPer [425], however, uses a coarse-grained approach, so that it does not break with the concept of tight loops. In fact, HyPer does not use SIMD instructions [529] and thus omits fine-grained data parallelism. Even on CPUs with a moderate degree of parallelism in SIMD instructions, database researches are challenged by integrating query compilation and SIMD instructions [487, 639].

In summary, using a micro-level technique for efficient on-chip pipelining on GPUs remains a challenge. Applying any of the commonplace techniques makes it necessary to combine at least three things that are hardly compatible: fine-grained data-parallel processing, extensive out-of-order execution, and deep operator pipelines. To achieve our goal of mitigating the GPU global memory bottleneck, we need to develop a new micro execution model.

8.2.5 Data-Parallel Query Compilation

In the following, we show a micro-level execution strategy that reduces GPU global memory access volumes by means of pipelining in on-chip memory. To this end, we show the approach of our query compiler *HorseQC* and its integration with the operator-at-a-time execution engine of CoGaDB [74].

8.2.5.1 Fusion Operators

HorseQC extends the operator-at-a-time approach with the concept of *fusion operators*, operators that embrace multiple relational operations. A fusion operator replaces a

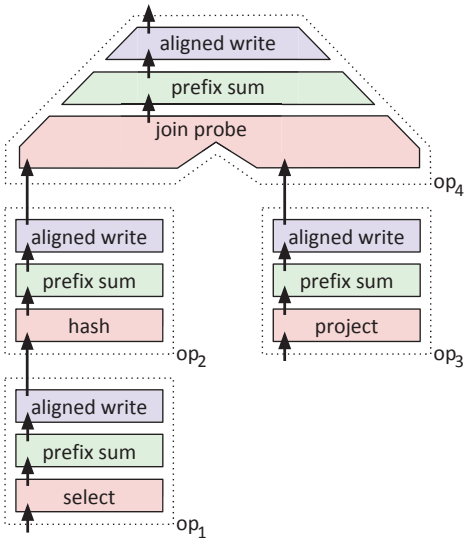

Fig. 8.11: Operator-at-a-time.

sequence of conventional operators in the physical execution plan with a micro-level-optimized pipeline. The data movement within a fusion operator can be improved by applying different micro level execution models.

8.2.5.2 Micro-Level Pipeline Layout

To keep matters simple, we first apply query compilation with the operator-at-a-time primitives described by He et al. [300]. This choice is not limiting as other data-parallel primitives may be used instead. However, a commonality of different primitive sets is that they use *relational primitives* with relational functionality (e.g. select) and *threading primitives* with thread coordination functionality (e.g., map, prefix sum, gather).

State Of The Art We look at a query with two input tables and a total of four relational operators $\texttt{op}_1, \cdots, \texttt{op}_4$. Operator-at-a-time runs three primitives per operator (cf. Figure 8.11 on the right): The first pass executes the relational primitive (e.g., select, project) and counts the number of outputs of each thread. The second pass computes a *prefix sum* to obtain unique per-thread write positions. The third pass performs an *aligned write*. This means that the output values are written into a dense array and may include executing the relational primitive for a second time to produce the output values. Thus, the query is processed in twelve operations with separate GPU global memory I/O.

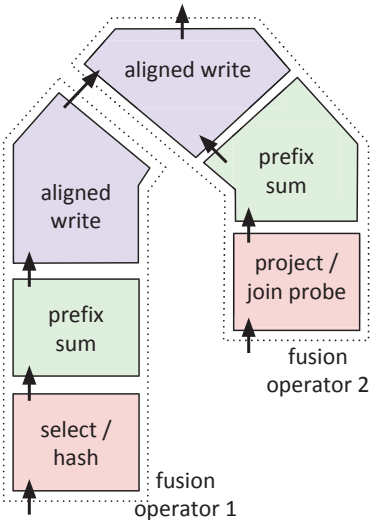

Fig. 8.12: Multi-pass QC.

Multi-Pass Query Compilation By grouping operations that are applied to the same input table, the query may be processed with two fusion operators. Within each fusion operator, we apply the following query compilation strategy (cf. Figure 8.12): We extract the prefix sum from the operators and execute it only once between all relational primitives and all aligned writes. The relational primitives are then compiled into one kernel called `count`, which is executed before the prefix sum. The aligned writes are compiled into one kernel called `write`, which is executed after the prefix sum. In this way, we apply *kernel fusion* [689] to the four relational primitives and to the four aligned writes. The same query is processed with six operations and the operations in compiled kernels communicate through on-chip memory instead of GPU global memory.

8.2.6 Memory Access and Limitations

In Figure 8.13, we illustrate the bandwidth characteristics of our example query when using code generation with three phases. The figure shows the behavior of the three-phase micro execution model described above with the batch processing macro execution model. To analyze the implications of forwarding intermediate results in the generated kernels through registers and scratchpad memory, we extended the illustration with an additional GPU-internal layer of memory.

GPU global memory access has previously been the bottleneck for query execution. Here the `count` kernel accesses 1.7 GB in GPU global memory, the prefix sum computation accesses 0.8 GB in GPU global memory, and the `write` kernel accesses 1.9 GB in GPU global memory. This is a reduction by a factor of 1.9× compared with batch

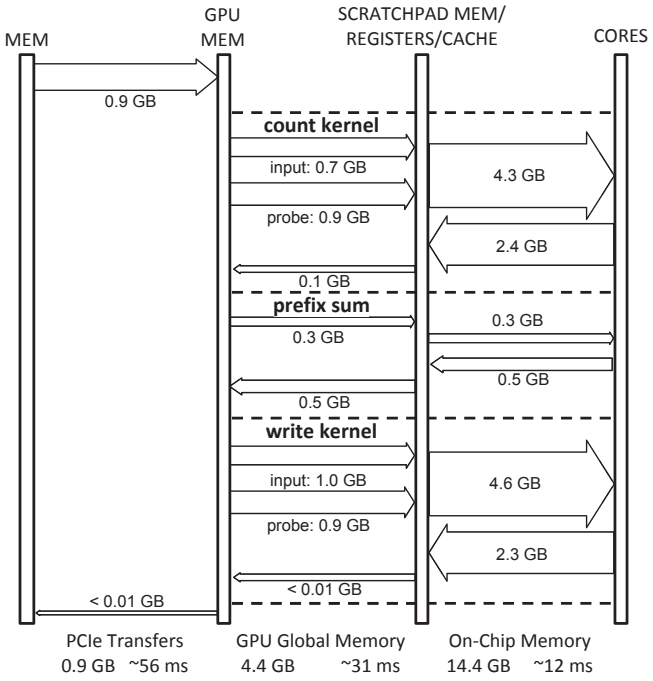

Fig. 8.13: Data movement for data-parallel query compilation with three phases.

processing. In the generated kernels, a substantial amount of memory traffic has moved to on-chip memory. In on-chip memory, the access volume of 14.4 GB is not a limiting factor due to the extremely high bandwidth of 1.2 TB/s of scratchpad memory.

Although the reduced GPU global memory traffic may suggest that the approach eliminates the bottleneck, real-world queries still experience limitations. In fact, Section 8.2.10.6 shows that compilation with three phases can still not saturate PCIe for 9 out of 12 SSB queries. This is because the query complexity prevents the strategy from utilizing the full GPU global memory bandwidth. Therefore, we investigate ways to further increase the processing efficiency in the next section.

8.2.7 Processing Pipelines in One Pass

The previous execution model relied on a typical programming concept of GPUs that executes operations with multiple kernels. The kernels that execute the actual work for the operations are interleaved with kernels that execute prefix sum computations. To further improve the processing efficiency, we have to break with this concept. With a new micro execution model, we avoid round trips to GPU global memory, which are caused by multi-pass implementations. This enables us to radically reduce GPU global memory traffic and lift the bandwidth bottleneck.

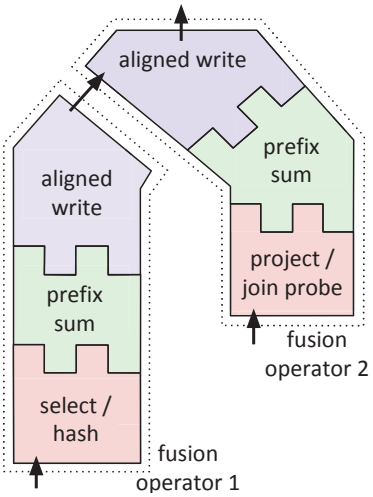

Fig. 8.14: Compound kernel.

Compound Kernel Kernel fusion brought reduction operations (e.g. prefix sum) as boundaries into the spotlight. Previously, we computed the prefix sum *between* two generated kernels to obtain write positions. Instead of two separate kernels, we now generate only one *compound kernel* that integrates the prefix sum computation (cf. Figure 8.14), which eliminates multiple passes. Computing write positions *within* a generated kernel makes it possible to process pipelines in one pass without intermediate materialization. In this way, each fusion operator is executed by a single compound kernel. In the following, we look at implementation strategies for reduction operations that enable fully pipelined processing.

Atomic Prefix Sum The separation into multiple reduction kernels with intermediate materialization impedes pipelining. To introduce a pipelined implementation, let us first look at a very simple sequential prefix sum:

```
for(i=0; i<n; i++)
  if(flags[i]) prefix_sum[i] = sum++;
```

The sequential prefix sum loops through the array `flags` while writing *and* incrementing `sum` for every valid entry. Figure 8.15a illustrates the use of the prefix sum for a dense write of selected input elements. When parallelizing the `for`-loop, this implementation runs into the issue of many threads trying to increment `sum` at the same time. To resolve this parallel dependency, atomic operations can be used to isolate parallel modifications of the same memory address. Atomic operations ensure a consistent state, yet are executed in an undefined order. The following code executes an *atomic prefix sum* to compute unordered, dense write positions:

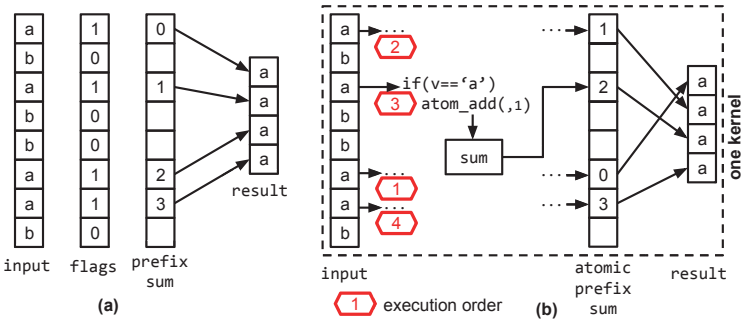

Fig. 8.15: The computation of a prefix sum for writing selected elements to a dense array (a) can be parallelized using atomic operations (b).

```
if(is_selected) wp = atom_add(&sum, 1);
```

Threads contribute an offset of `1` to the sum at address `&sum` by executing the expression conditionally. Each `atomic_add(..)` returns the previous state of `sum`. Thus, threads immediately obtain a unique global write offset as `wp` in register. This is illustrated in Figure 8.15b.

The use of atomic operations causes a break with the semantic of the prefix sum because the result has *no defined order*. For the relational semantic, however, only the *uniqueness* of output positions is critical. Output permutations lead to non-aligned GPU global memory access where adjacent threads do not write to adjacent memory addresses. The impact on write throughput, however, is limited, because the filter semantics lead to non-aligned access for separate prefix sums as well.

8.2.7.1 Memory Access and Limitations

The compound kernel micro execution model further reduces GPU global memory access by a factor of 2.4× to 1.8 GB (see Figures 8.13 and 8.16). Compared with operator-at-a-time, this is a reduction by a factor of 4.7×. Pipelining the prefix sum avoids round trips to GPU global memory that are necessary in the three-phase micro execution model. The compound kernel has only a minimal GPU global memory access volume for input, output, and hash-table access. Now the on-chip traffic is balanced with the GPU global memory traffic when relating each memory volume to the available bandwidth.

The described approach heavily relies on atomic operations. This has the disadvantage of causing limitations for parallelism. Although the execution order is undefined, the operations *are* sequentialized and reducing n values takes $O(n)$ parallel steps. However, Egielski et al. [195] showed that recent hardware support makes atomic operations competitive with parallel algorithms. Still, the integrated prefix sum puts significant pressure on the atomic functional units, which prevents pipeline kernels from utilizing

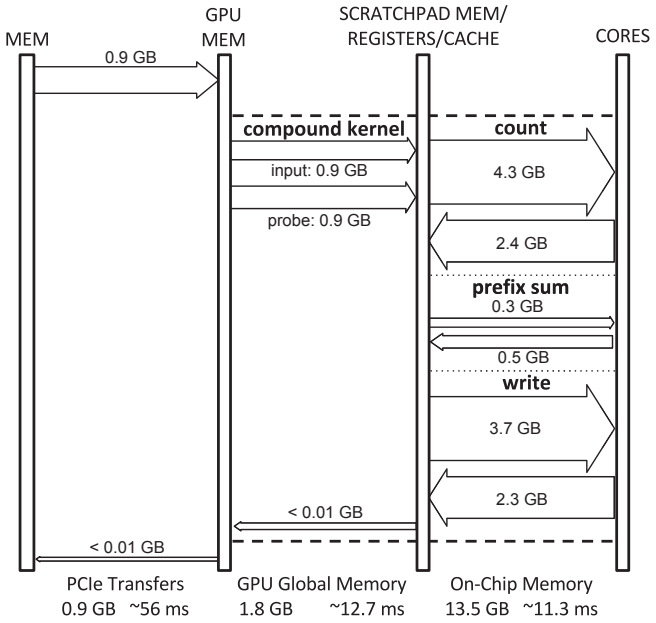

Fig. 8.16: Data movement for query compilation with one pass. The compound kernel reduces data movement by 4.7×.

full GPU global memory bandwidth. In the following, we address this issue and show how the efficiency of parallel reductions in compound kernels can be increased.

8.2.8 Efficient Pipelined Reductions

We have showed a way to pipeline reductions in generated kernels using atomic operations. This benefits the memory efficiency, but also reveals the atomic functional units of a GPU to be a bottleneck. This is especially critical because several operations that are combined in the compound kernel rely on atomic isolation as well. Specifically, the state-of-the-art implementations of hash joins and hash aggregations [358] use atomic operations to isolate hash table inserts.

This section addresses performance bottlenecks that occur when utilizing atomic reductions to pipeline relational operators. We show a new technique called *local resolution, global propagation*, that is used by *HorseQC* to pipeline prefix sums, single tuple aggregation, and grouped aggregation efficiently. The approach reduces the pressure on atomic functional units and offers tunability regarding hardware and thread-group granularity. We describe the approach in the following.

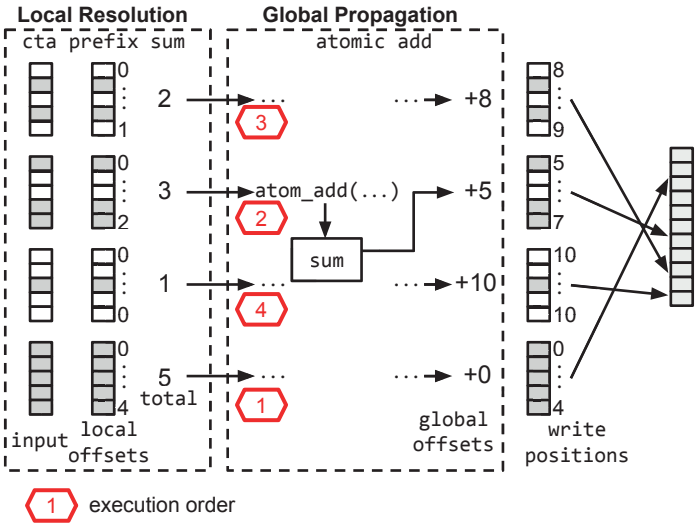

Fig. 8.17: Computing write positions with local resolution (local offset), global propagation (global offset).

8.2.8.1 Local Resolution, Global Propagation

Like other efficient GPU implementations such as in CUB [489], local resolution with global propagation consists of two levels of reductions. In contrast to other techniques, however, it always uses pipelined techniques on *both* levels. Local resolution is an additional pre-reduction step, computed by a local thread group, whereas global propagation is the same atomic reduction as described in Section 8.2.7. We use the term *Collaborative Thread Array* (CTA) for the thread groups in local resolution. CTAs can either match the workgroup (AMD) or thread-block (NVIDIA) size of the GPU kernel or work on a finer granularity.

The following code, illustrated in Figure 8.17, executes an atomic prefix sum using local resolution, global propagation:

```
l_os = cta_prfx,(flags, &cta_total);  //local res.
if,(cta_thread_idx == 0)
  g_os = atom_add,(&sum, cta_total);  //global prop.
wp = l_os + g_os;
```

First, each CTA executes `cta_prfx` to compute a local prefix sum on `flags`. This is the local resolution step. We implement `cta_prfx` with SIMD reductions (cf. Intra-Warp Scan Algorithm by Sengupta et al. [622]). The function returns the local offset `l_os` and the sum of all flags assigned to the CTA `cta_total`. Second, one thread of each CTA adds `cta_total` atomically to a global counter `sum`. This is the global propagation step.

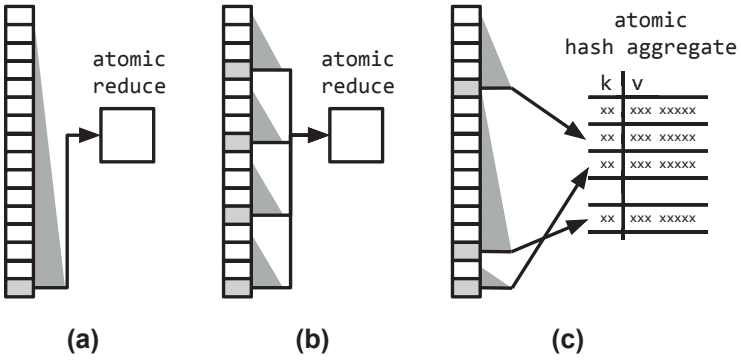

Fig. 8.18: Local resolution mechanisms: (a) Work-efficient reduction (b) SIMD reduction (c) segmented reduction.

The call to `atom_add` returns the global offsets `g_os`. Finally, the write position `wp` is the sum of `l_os` and `g_os`.

Compared with the simple atomic prefix sum, we now add pre-aggregates instead of 1/0 flags to `sum`. Accordingly, each atomic add obtains ranges of output indices instead of a single index. The process is analogous to *allocating* segments of output memory to CTAs. The order of the allocations is undefined, however. (See the execution order in Figure 8.17.) This leads to an output that is ordered *within segments* and permuted *between segments*. Further investigation reveals that, due to the GPUs stream processing engine, the permutations exhibit locality, leading to semi-ordered output data.

Local Resolution Mechanisms The mechanisms used for local resolution are interchangeable. This makes it possible to tune pipelined reductions and to apply them in different operations. Figure 8.18a and 8.18b show the integration of work-efficient reductions [56] and SIMD reductions [622]. Both techniques have different thread group granularities and we can choose between them to adapt to the hardware parallelism of different processors. Figure 8.18c shows the use of pipelined segmented reductions for grouping. First, segmented reductions compute pre-aggregates in scratchpad memory. Second, global propagation inserts the pre-aggregates into a hash table with an atomic operation. The ability to control scratchpad memory opens up a new design space for grouping algorithms in pipelined computations (e.g. handling frequent items). A similar approach PLAT [722] aggregates frequent grouping keys in a table local to each CPU core.

8.2.9 DBMS Integration

We integrated our query compiler *HorseQC* into the open source DBMS CoGaDB, leveraging the built-in code generator *Hawk* [75]. The DBMS uses a columnar data layout

and processes full columns operator-at-a-time on GPUs and CPUs. We use the front-end and the storage layer of CoGaDB; *HorseQC* adds a compiler-based execution engine.

We added two components to the DBMS: 1. a query compiler that compiles fusion operators to GPU code (cf. Section 8.2.4); and 2. a *translation layer* that identifies fusion operators and drives the query compiler. Currently, there are two different workflows for the translation layer:

1. CoGaDB parses the SQL code for a query and generates a query plan. The translation layer applies the produce/consume model [529] to the query plan to determine fusion operators. We use this approach for the SSB queries and TPC-H Q6.
2. The translation layer parses a JSON file that describes the query plan including the fusion operators. This enables us to process queries when (1) cannot handle the queries via SQL (e.g. correlated subqueries or automatic unnesting). This is used for the other TPC-H queries.

When the fusion operators are defined, the translation layer drives the query compiler to compile and execute. Finally, the decompression of dictionary compressed columns and sorting are executed by CoGaDB's original execution engine.

8.2.10 Evaluation

Section 8.2.3.1 showed that query coprocessing in existing macro execution models is sensitive to memory bandwidth bottlenecks on various hierarchical levels. We proposed several micro execution models that allow to remove memory indirections to achieve a more efficient use of bandwidth. In this section, we evaluate our approaches and carefully assess bandwidth and throughput in identifying several benefits.

The experimental study is structured as follows. First, we evaluate the *micro execution models* and we execute specific queries to analyze the *reduction performance* of the proposed techniques in Experiments 1 and 2. Then, we evaluate the micro execution models for the SSB and TPC-H benchmarks in Experiments 3 and 4. Next, we analyze the *integration* of our micro execution model with the batch processing *macro execution model*. In doing so, we analyze the *real-world benefits* of our approach with a comparison of end-to-end performance in Experiment 5 and a scalability analysis in Experiment 6. Note that all experiments, except for Experiment 6, were executed with scale factor 10.

8.2.10.1 Processing Techniques

This section describes three micro execution models built into *HorseQC*. The goal is to use them within macro execution models to improve performance. Therefore, it is crucial to achieve a higher throughput than PCIe when executing queries. We show the benefit of our approaches by comparing them with an operator-at-a-time micro

Tab. 8.3: Coprocessors used in the evaluation.

Model	Type	Architecture	Cores	Scratch pad (KB)	B/W (GB/s)
GTX970 (NV)	GPU	Maxwell	13	96	146.1
GTX770 (NV)	GPU	Kepler	8	48	167.6
RX480 (AMD)	GPU	Ellesmere	32	32	104.9
A10 (AMD)	APU	Godavari	8	32	18.7

execution model. In this way, we analyze the benefit of moving data transfers between relational operators to the on-chip level.

Multi-pass The first approach separates reductions from the generated kernels, which leads to an execution in multiple passes (Section 8.2.5). Each reduction is executed on materialized data using the `boost::compute` library.

Pipelined The second approach integrates reductions into a fully pipelined kernel using atomic operations (Section 8.2.7). By using atomic operations for each reduction input, the approach is an instance of local resolution, global propagation that has no local resolution step.

Resolution The third approach increases the efficiency of pipelined reductions with local resolution methods such as pre-aggregation (Section 8.2.8). We differentiate between local resolution implementations using Resolution:SIMD for SIMD reductions and Resolution:WE for work-efficient reductions.

Operator-at-a-time We use CoGaDB 0.4.1, which processes full columns of data in each operator with CUDA kernels. It features a run-to-finish macro execution model and an operator-at-a-time micro execution model.

8.2.10.2 Baselines

PCIE transfer The *PCIe transfer time* is the time it takes to transfer input and output data between the host's main-memory and GPU global memory. It is the target time used by micro execution models for balancing throughput and PCIe bandwidth. The PCIe transfer time is shown in each graph with a dashed line (- - -).

Memory bound The *GPU global memory bound execution time* is the time it takes to access the data. As each approach has to read the input columns and write the output columns, the baseline is a lower bound on the kernel execution time. We indicate it with a solid line (——) in each graph.

Listing 8.1: Query 1 is a simple selection and projection query inspired by the star schema benchmark.

```
SELECT lo_extprice * lo_discount + lo_tax AS revenue
FROM   lineorder
WHERE  lo_quantity BETWEEN 25 - x AND 25 + x
```

8.2.10.3 System Configuration

For the experiments, we use three dedicated GPUs with PCIe gen 3.0 links and one APU that accesses main-memory directly. Table 8.3 specifies the GPU models and shows hardware properties. The amount of scratchpad is available *per core*. The reported bandwidth refers to GPU global memory for the GPUs and to main-memory for the APU. It was measured using on-GPU `memcpy` of 1 GB data. We measured bidirectional PCIe transfers between CPU and GPU as 12.1 GB/s.

Both NVIDIA GPUs GTX770 and GTX970 run in a system with an Intel Xeon E5-1607 CPU. We use the NVIDIA 364.19 driver and CUDA Toolkit 7.5 with OpenCL drivers. The AMD RX480 GPU is placed in a separate system with the A10-7890K APU. We use the AMDGPU-Pro 16.40 driver for the GPU and the fglrx 15.201 driver for the APU. Each system is running Ubuntu 14.04 and uses the boost library 1.61.

We used the profiling tools `nvprof 2.0.28` for NVIDIA hardware and `CodeXLGpuProfiler V4.0.511` for AMD hardware to measure kernel execution times, PCIe transfers, and GPU global memory access. For the measurements of kernel execution times, we used both tools to profile individual kernels and sum up the kernel execution times when multiple kernels were involved.

8.2.10.4 Experiment 1: Pipelined Prefix Sum

We compare several pipelined prefix sum techniques with one non-pipelined technique for a query that filters and projects one table. This allows us to analyze the benefit of integrating prefix sum computations into single-pass kernels. We execute Query 1, shown in Listing 8.1, and vary the selectivity in the range [0, 1] using `x`. By running the experiment on four GPUs, we aim to assess the best local resolution mechanisms for a given hardware. Figure 8.19 shows the results.

Observations Pipelined techniques perform better than Multi-pass in most cases. Integrating the prefix sum computation into single-pass kernels reduces the kernel execution times by a factor of up to 6.3×. While processing with Multi-pass takes up to 328.6 % of the PCIe time, Resolution:SIMD uses only 101.3 % of the PCIe time in the worst case (selectivity 1.0, RX480). This shows that the approach can saturate the bus bandwidth for a variety of configurations. On the A10 there are no PCIe transfers and Resolution:SIMD increases the overall throughput by factors of up to 1.6× over Multi-pass.

The results show that the local resolution step reduces the performance impact of atomic operations. This becomes visible for higher selectivity factors. Pipelined has higher executions times because the strategy executes one atomic addition per output. However, Resolution:SIMD and Resolution:WE show good performance across all selectivities due to local resolution.

Resolution:SIMD achieves the shortest kernel execution times in most cases and allows memory bound processing on the GTX970. On the GTX770, lowering the output

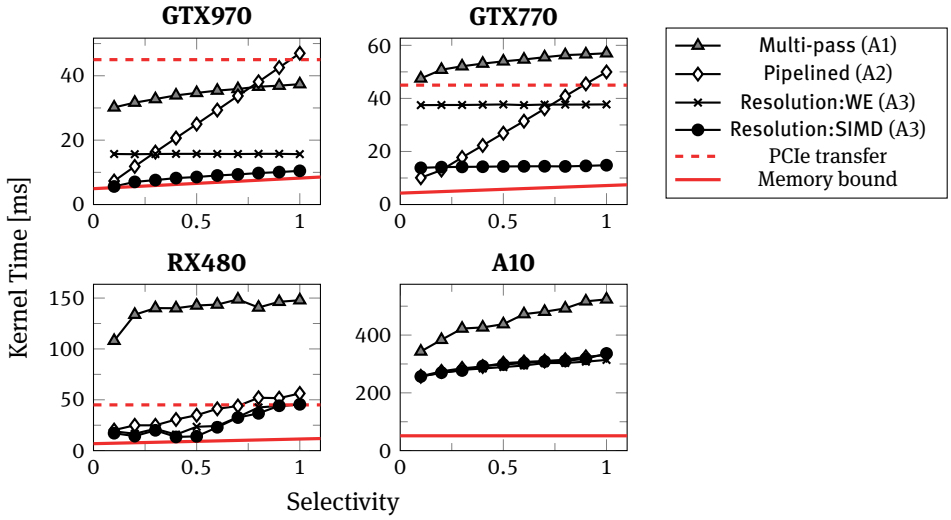

Fig. 8.19: Projection query executed with different approaches. Integrating prefix sums into kernels allows fastest execution.

size down to 0 does not affect the execution time. We conclude that the GTX770 is compute-bound earlier than the GTX970. The higher memory bandwidth of the GTX770 leads to an increased throughput for atomic operations and Pipelined can outperform Resolution:SIMD for selectivities below 10 %. On the RX480 and on the A10 there is no definite advantage for one of the reduction techniques. In the following, we use only Resolution:SIMD and skip the other techniques for a clear presentation.

8.2.10.5 Experiment 2: Pipelined `GROUP BY`

We evaluate the effect of pipelined `GROUP BY` aggregations using Operator-at-a-time, Pipelined, and Resolution. The query groups all tuples of `lineorder` according to the computed attribute `lo_orderkey%x` into sums. We vary the number of groups by increasing x from 2 to 16 384. We show the results of the experiment on a GTX970 GPU in Figure 8.20.

Observations The execution times of Operator-at-a-time do not depend on the group size. The main cost factor is sorting the input columns. Pipelined shows up to 11.1× lower execution times but only for larger group sizes. For group sizes below 64, we observe high execution times. This is caused by the heavy contention of parallel aggregation hash-table inserts.

The bottleneck is resolved by Resolution which uses pre-aggregations to reduce the contention. The results show that execution times reduce by a factor of up to 126×. However, the local pre-aggregations have a limited effect on larger group numbers. This

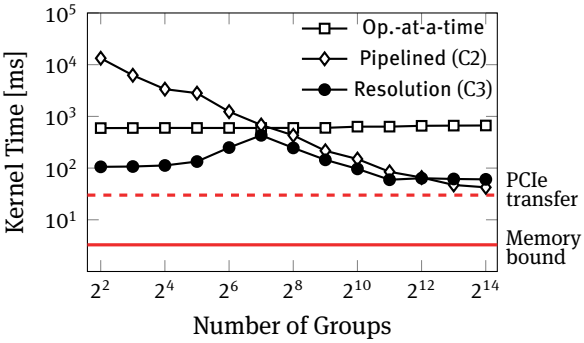

Fig. 8.20: Performance of grouped aggregations.

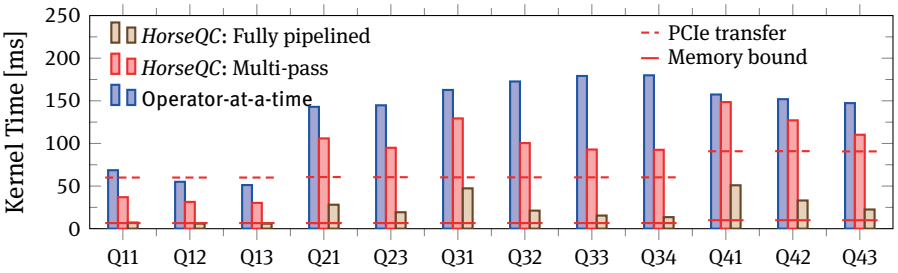

Fig. 8.21: Performance of SSB queries.

explains the spike at 128 groups, where both pre-aggregation and contention have an effect. While the approaches cannot saturate PCIe when aggregating a full table, filters reduce the cost of grouping for real-world queries.

8.2.10.6 Experiment 3: Star Schema Benchmark

The previous experiments showed that pipelining specific reduction operations helps to increase the throughput of query processing. In this experiment, we analyze whether this behavior carries over to real-world situations. To this end, we execute the SSB Queries[3] on the GTX970 GPU.

We use Operator-at-a-time and two variants of our query compiler. *HorseQC*: Multi-pass uses pipeline breaking implementations for reductions (A1, B1 and C1). *HorseQC*: Fully pipelined integrates all pipeline operations in one kernel (using A3, B3 and C2). We show the results of the experiment in Figure 8.21.

3 We could not process SSB Query 2.2 as we do not yet support range predicates on dictionary compressed columns.

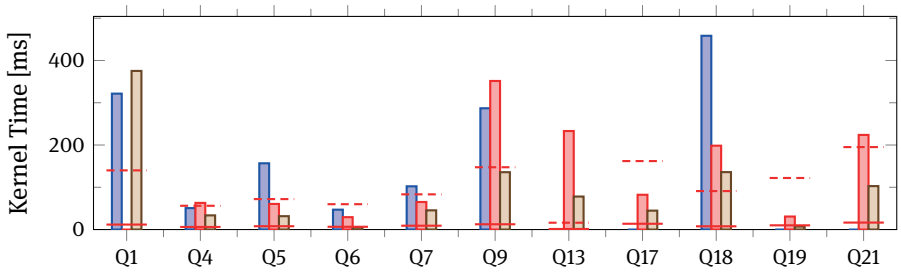

Fig. 8.22: Performance of TPC-H queries.

Observations The bandwidth analysis in Section 8.2.3.1 showed that 4 out of 12 queries are limited by GPU global memory access in operator-at-a-time processing.

- The kernel execution times of Operator-at-a-time show that compute and latencies make the problem worse. While PCIe would allow execution times between 60.6 ms to 90.9 ms, the kernel execution times take longer for 10 out 12 queries with up to 295.5 %.
- *HorseQC*: Multi-pass improves over Operator-at-a-time and uses only 50.5 % of the PCIe bandwidth transfer time in the best case and 215.5 % in the worst case. This shows that without efficient pipelining of reduction operations, the benefit of query compilation is limited.
- *HorseQC*: Fully pipelined lowers all kernel execution times to a level that is consistently lower than PCIe transfer times. This shows that compiling pipelines into one kernel with local resolution, global propagation provides an execution approach with sufficient throughput. Processing takes 9.7 % of the PCIe transfer time in the best case and 78.1 % in the worst case. For Queries 1.1, 1.2, and 1.3, kernel execution is memory bound by GPU global memory access.

8.2.10.7 Experiment 4: TPC-H Queries

We execute and profile queries from the TPC-H benchmark to show the effect when relaxing the specific assumptions of the star schema benchmark (e.g. using one centralized table). We select a subset of queries based on the work by Boncz et al. [61] to capture challenging aspects of the TPC-H benchmark: Q1, Q4, Q13, and Q21 contain heavy aggregation; Q9 and Q18 contain heavy joins; and Q4, Q19, and Q21 contain parallelism bottlenecks. We modified 4 queries, because *HorseQC* currently does not support all operations, e.g., `like` expressions. The results of the experiment are shown in Figure 8.22. For Q1, there is no result for *HorseQC*: Multi-pass, because the strategy ran out of GPU memory. The results shown for Operator-at-a-time are for all TPC-H queries supported by the DBMS.

Observations The PCIe and memory-bound baselines show larger variations than for the SSB benchmark. This is mainly caused by the join structure, e.g., Q13 joins three small tables, while Q17, Q18, and Q21 join multiple instances of the largest `lineitem` table.
The kernel execution times show that *HorseQC* can improve over operator-at-a-time by factors of up to 8.6×. For Q1, Q4, and Q9, there are cases where Operator-at-a-time has shorter kernel execution times than compiled strategies. Further investigation showed that in these cases Operator-at-a-time moves some operators to the CPU, which means that the measurements cover a limited amount of operations.

Comparing the variants of the query compiler, we observe that *HorseQC*: Fully pipelined consistently improves over *HorseQC*: Multi-pass by a factor of up to 5.4×. *HorseQC*: Fully pipelined achieves lower execution times than PCIe transfer times for 8 out of 11 queries. For Q1, Q13, and Q18, the PCIe bandwidth cannot be fully saturated. This is because the queries contain grouped aggregations of unfiltered columns (cf. Experiment 2). The execution times of *HorseQC*: Fully pipelined take 5.6 % of the PCIe transfer time in the best case and 268.1 % in the worst case.

8.2.10.8 Experiment 5: Scalability

Due to the deeply integrated storage layer implementations of the host DBMS CoGaDB, we were unable to build a fully scalable version of *HorseQC*. For this reason, we perform a separate experiment that integrates the Resolution micro execution model with the batch processing macro execution model for the star join from SSB Query 3.1. Decoupling this experiment allows us to apply the rules for coprocessor data management by Yuan et al. [728] and to measure end-to-end performance for larger datasets.

The star join recombines three dimension tables and one fact table with an overall selectivity of 3.4 %. We build hash tables for the dimension tables in GPU global memory. The fact table resides in pinned host memory and each column is partitioned into blocks of 0.5 MB, 2 MB, or 8 MB. The blocks are transferred asynchronously via PCIe into an inner kernel that computes the star join by probing each dimension hash table.

Figure 8.23 shows the end-to-end execution times for each block size when executing the experiment. We observe that execution times grow linearly with increasing scale factors and that block sizes larger than 2 MB can saturate the PCIe bandwidth. The computation does not become a bottleneck for the examined scale factors. With a block size of 4 MB and scale factor 300, the size of intermediate data in GPU global memory is only 473 MB. Therefore, we expect the approach to scale to even larger databases with linear performance.

8.2.10.9 Experiment 6: End-to-End Performance

To make a comparison with other database systems, we execute the TPC-H queries with different database systems and measure end-to-end performance. We compare MonetDB5 Dec2016-SP3 executed on CPUs, and CoGaDB 0.41 and *HorseQC* executed

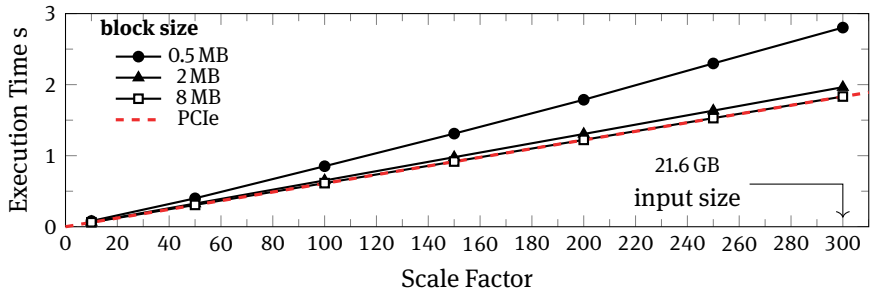

Fig. 8.23: End-to-end performance of star join computation for different scale factors.

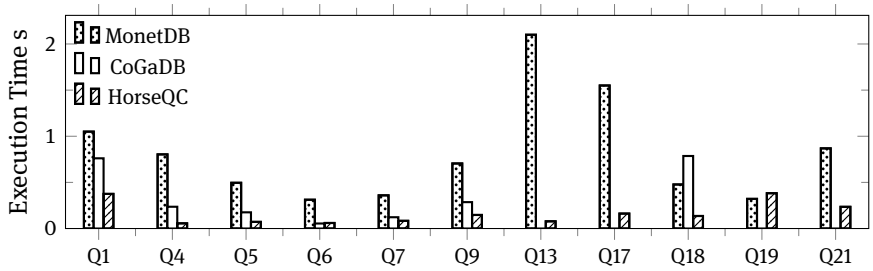

Fig. 8.24: End-to-end performance of TPC-H queries.

on GPUs. Both competitors feature an operator-at-a-time approach. We perform the measurements with warm caches. MonetDB runs on a workstation-class system with an Intel Xeon E5-1607 CPU and 32 GB RAM. CoGaDB and *HorseQC* run on the GTX970. The results are shown in Figure 8.24.

Observations For the supported queries, *HorseQC* is up to 5.8× faster than CoGaDB. While CoGaDB uses GPU global memory as a cache for frequently used columns, *HorseQC* does not cache data between queries. This shows that *HorseQC* uses memory and interconnects more efficiently. For Q6 there is no improvement, because query execution is PCIe bound.

HorseQC has lower execution times than MonetDB by a factor of up to 26.9×. Despite moving data through the PCIe bottleneck, the additional bandwidth resources of GPU global memory offer an acceleration. For Q19, MonetDB has a lower execution time than *HorseQC*. This shows that for queries with a low complexity, it is more effective to process data directly than moving it over PCIe.

8.2.11 Discussion

In the previous experiments, we evaluated our new approaches for querying compilation on coprocessors. Across all experiments, we were able to show improvements of query compilation over operator-at-a-time processing. Operator-at-a-time has a low memory efficiency due to large materialization volumes and repetitive operations. Therefore, the approach cannot efficiently utilize the memory systems surrounding the coprocessor.

While naive compilation techniques increase the memory efficiency, reductions and prefix sums split operator pipelines into multiple passes. In this way, the approach inherits the drawbacks of operator-at-a-time. This becomes visible because kernel execution times frequently exceed PCIe transfer times.

We demonstrated a query compilation technique that merges the operators of a pipeline into one compound kernel. When combined with efficient reduction techniques, the compound kernel achieves substantial advantages over other processing approaches. With upcoming OpenCAPI and NVLink interconnects, these improvements to GPU-local processing are essential in order to take advantage of the increased bandwidth of the new hardware. In the evaluation setting, the PCIe bandwidth can be saturated for all SSB queries. For the TPC-H benchmark, the approach is an improvement over operator-at-a-time and naive compilation, but saturates PCIe in only 8 out of 11 queries. We conclude that the compound kernel works particularly well with star join queries.

8.2.12 Summary

In this section, we showed query processing techniques that help to balance the data movement cost and compute throughput on GPU-style coprocessors. We measure the data transfer volumes in different scalable processing approaches to assess bandwidth bottlenecks. While naive scalable execution techniques are limited by PCIe bandwidth, batch processing is limited by GPU-local throughput. To address the bottleneck, we propose micro execution models that benefit from on-chip pipelining. Naive query compilation techniques allow simple code generation but inherit the memory-intensity of operator-at-a-time. We introduce compound kernels that merge several pipeline phases into one efficient kernel.