Scheduling Periodic Real-Time Communication in Multi-GPU Systems

Uri Verner Avi Mendelson Assaf Schuster

Technion – Israel Institute of Technology
Technion City, Haifa 3200003, Israel, +972-48294883
{uriv, avi.mendelson, assaf}@cs.technion.ac.il

***Abstract*—Multi-GPU systems have become a popular architecture for high-throughput processing of streaming data. In many such systems, data transfers inside the compute nodes are becoming a performance bottleneck due to insufficient bandwidth. The problem is even more acute for real-time systems that sacrifice utilization and efficiency in order to achieve predictable and analyzable execution.**

Data transfer over the interconnect of a compute node is most efficient when it is streamed on multiple paths in parallel. However, this mode of operation greatly complicates the transfer time analysis due to the effects of bus contention, especially if the data transfers are asynchronous.

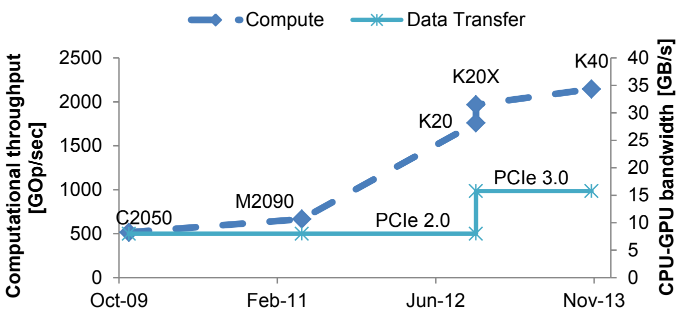
This work presents a new scheduler for periodic data transfers with deadlines that uses the system interconnect efficiently. The scheduler analyzes the data transfer requirements and their time constraints and produces a verifiable schedule that transfers the data in parallel. Experiments on realistic systems show that our method achieves up to 74 % higher system throughput than using the classic scheduling methods.

I. Introduction

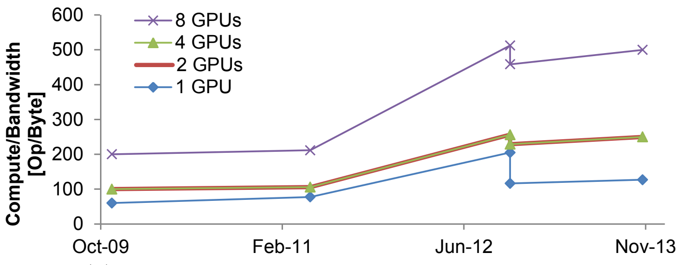
In recent years, due to the prevalence of high-rate data collection, vast amounts of streaming data need to be processed. A preferred solution for handling the rapid growth in demand for both high bandwidth and high processing rates is the use of compute accelerators, such as GPUs [8]. An important subset of these applications requires handling intensive data streams in real-time, as is the case, for example, in VoIP servers, video tracking systems, and production inspection systems.

In modern architectures, the compute power is growing much faster than the communication bandwidth, as illustrated by Figure I.1a which compares the growth of computational throughput vs. CPU-GPU bandwidth for NVIDIA Tesla-series cards[1]. The figure indicates that the bandwidth almost doubles with each PCIe generation, but clearly the slope of compute growth is steeper. The PCIe speeds are not expected to grow further before 2016.

[1]The data represents maximum values, according to manufacturer specification. Throughput values are for floating-point operations, such as ADD, MUL, and FMA. Bandwidth values are for unidirectional data transfer. The chart axes are scaled proportionally, for fair gradient comparison.



(a) Growth of GPU compute vs. data transfer bandwidth



(b) Multi-GPU compute vs. data transfer bandwidth

Figure I.1: Compute throughput grows faster than data transfer bandwidth in CPU/GPU systems

The gap increases substantially for multiple GPUs. As shown in Figure I.1b, the ratio between compute throughput and CPU-GPU bandwidth grows with the number of GPUs[2]. (The aggregate bandwidth between the main memory and the GPUs is considered.) For example, while the compute throughput of eight GPUs is twice that of four, the CPU-GPU bandwidth remains the same because the system multiplexes GPU pairs on the same PCIe bus. The CPU-GPU bandwidth is already a limiting factor that prevents many applications from using a GPU as an accelerator, and this problem is expected to worsen in the near future. In order to ease the problem, different chip manufacturers offer the use of hybrid CPU/GPU architectures that have higher data transfer bandwidth. Unfortunately, these architectures are forced to use GPUs with lower computational throughput due to a limited

[2]Using a dual-socket system for four GPUs or more.

power envelope, and the bandwidth problem reappears when scaling to multiple GPUs.

NVIDIA announced a new interconnect technology called NVLink on March 2014, which it intends to integrate into its next generation of GPUs, starting 2016. This technology is expected to provide 80-200 GB/s point-to-point links among compatible GPUs and CPUs, which outperforms PCIe 3.0 by a factor of 5-12. However, Stacked Memory, another technology to be implemented in these GPUs, is expected to significantly speed up computations by increasing the GPU memory bandwidth, so the communication-to-computation gap may not decrease. Moreover, if NVLink is used for CPU-GPU communication, internal system buses may become a bottleneck. The specifications of this technology have not been released up to the date of writing.

This work focuses on real-time systems, which are typically modeled as a collection of simple repetitive tasks with known (worst-case) execution times. In order to simplify the scheduling and timing analysis of such systems, their architecture is abstracted as a set of homogeneous serial processors with a shared uniform-access memory. However, this abstraction often comes at the cost of inflated execution time requirements due to hiding some of the features.

In real-time systems with a distributed topology, the interconnect serves as a shared resource that is managed by a data transfer scheduler. Traditionally, such schedulers use the uniprocessor model in order to avoid bus contention, and do not schedule data transfers on different routes in parallel. This model does not fit the interconnect of a system with multiple CPUs and GPUs, where transferring data on different routes in parallel is essential to achieving high bandwidth. To make use of parallel data transfer, one could suggest the multiprocessor model, where data routes are processors that execute data transfer tasks. However, this model does not fit the interconnect either, because it assumes that the processors are homogeneous and have constant throughput. The interconnect, on the other hand, is heterogeneous, as it is composed of several domains that use different communication technologies (e.g., QPI and PCI Express), and the data throughput on each path changes in accordance with bus contention.

Contribution: This paper uses a heterogeneous system model that better represents CPU/GPU systems and presents a scheduler for periodic data transfers with real-time constraints that uses an infrastructure-aware execution method to achieve high data transfer efficiency. The new method applies a greedy strategy that iteratively combines single-data-transfer tasks into batch tasks that transfer data in multiple streams using the Batch method [10]. Our method reduces the communication time requirements of the task set and can find a valid schedule for workloads that were previously considered unschedulable. When combining the tasks, the scheduler defines the time constraints of the new batch task to be at least as restrictive as that of the original tasks, hence guaranteeing that any schedule produced using the new task set will be valid with respect to the original time constraints.

Organization: In the rest of this paper, we provide the requisite background and related work (Section II), describe the system model (Section III) and our proposed scheduling method (Section IV), present our experimental results (Section V), and conclude (Section VI).

II. Background and Related Work

A. Multi-GPU Server Architecture

Our baseline architecture is based on a multi-GPU system that includes one or more multi-core CPUs and a set of discrete GPUs. The system has a NUMA architecture: each CPU and GPU is connected to a local DRAM memory module, which it can access with higher bandwidth and lower latency than the remote modules. These components are connected via the *system interconnect*, which forms several communication domains where the components use the same architecture and protocol for communication. The domains are bridged by components that belong to more than one of them. To transfer data from or to GPU memory, a CPU configures a DMA controller (a.k.a. DMA engine) which executes the transaction. Each GPU is equipped with one or two on-board DMA controllers that serve data transfers from and to this GPU. Each DMA controller can serve only one data transfer at a time. The data route between each two endpoints is fixed. Fujii et al. [4] showed that direct I/O operations are faster than DMA controllers for small data transfers. In this paper, we focus on the more common DMA-based method.

The effective bandwidth of the interconnect depends on the system's ability to make use of its multiple data routes. When data is transferred over a single route, the bandwidth is limited by the domain with the narrowest bandwidth, but when two or more routes are used,

Table I: Bus bandwidth in a Tyan FT72B7015 system

Communication domain	Bus type	Bandwidth (GB/s)
Memory bus	DDR3	32
Processor interconnect	QPI	9.6
PCI Express bus	PCIe 2.0	8

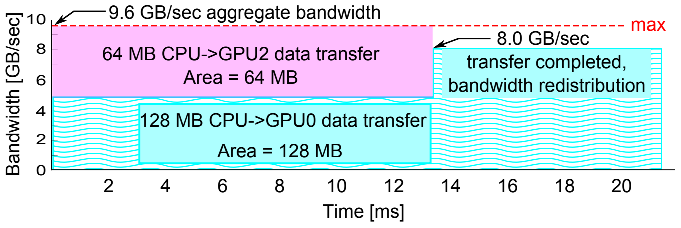


Figure II.1: Example of bandwidth distribution during the transfer of two data blocks

the aggregate bandwidth bottleneck is not necessarily in this domain. For example, Table I lists the theoretical bandwidth of the communication domains and their bus types in a Tyan FT72B7015 system. One can see that the PCI Express bus has lower bandwidth than the processor interconnect, so the CPU-GPU bandwidth is limited by the PCI Express bus. However, when transferring data from the CPU to two GPUs that use different PCIe buses, the bottleneck is in the processor interconnect. Figure II.1 shows the bandwidth distribution in this system during the transfer of two data blocks: 128 MB CPU$\rightarrow$GPU0 and 64MB CPU0$\rightarrow$GPU2. Note that the aggregate bandwidth decreases when the shorter transfer completes.

B. Scheduling

Real-time systems are typically modeled as a finite collection of simple and highly repetitive *tasks*, each of which generates, in a predictable manner, basic units of execution that are called *jobs,* with strict *deadlines.* At run-time, the *scheduling algorithm* generates a schedule for the execution of the jobs. In order to guarantee that all the jobs complete before their deadlines, a static *schedulability test* is often used.

In this work, we focus on *periodic* task systems, which are composed of tasks that generate jobs at regular periodic intervals. Such tasks are characterized by the tuple $T = (a, c, d, p)$, where a denotes the generation time of the first job, c denotes the worst-case execution time (WCET) of a job, d denotes the relative deadline from the time of job generation, and p denotes the job generation interval. If for all the tasks in a periodic system $d = p$, the system is called *implicit-deadline*, and the parameter d is omitted from the task characterization. The parameter a can also be omitted if the generation time of the first job is unknown.

Job execution is commonly modeled as either *preemptive* or *non-preemptive.* In the preemptive model, the execution of a job may be interrupted and resumed later. There is no penalty associated with such preemption. In the non-preemptive model, once a job starts execution, it executes to completion.

Earliest Deadline First (EDF) is a popular scheduling algorithm that was extensively studied for a variety of real-time task types and system configurations [7], [1], [13], and static methods for testing the validity of the schedule were introduced (e.g., by Zhang and Burns [14]). In this paper, we use a non-preemptive variation of this algorithm.

The *utilization* of a periodic task is defined as the ratio between its WCET and the job generation interval, $u_i = c_i/p_i$. Utilization is a measure of the load on the system and is often used in schedulability tests. For example, Liu and Layland [7] proved that a set of implicit-deadline periodic tasks is schedulable on a uniprocessor iff $\sum_i u_i \leq 1$, and that EDF will produce a valid schedule in this case.

The schedulability problem for non-preemptive periodic tasks with specified release times is NP-Hard [5]. We use a sufficient pseudo-polynomial schedulability test that was described by Jeffay et al. [5] to verify the schedules for data transfers with timing constraints. The test checks the utilization condition $\sum_i u_i \leq 1$, among others.

C. Related Work

Verner et al. presented a framework for efficient processing of large numbers of asynchronous data streams under strict latency constraints on CPU/GPU systems [11], [12]. The framework distributes the data streams between the CPUs and GPUs, such that each processor works on multiple streams. The processing on each GPU works in a coarse-grain pipeline, where the incoming data is accumulated in a buffer, copied to the GPU, processed, and the results are copied back to the system memory. To increase the data transfer efficiency, the framework tries to accumulate as much data as possible before copying it to the GPU; however, due to the batch processing mode, the end-to-end processing time is limited by the shortest latency bound of the streams processed on that GPU. To avoid bus contention issues when working with multiple GPUs, the framework abstracts them as a single virtual-GPU, and works with them in lock-step. Although this approach achieves higher CPU-GPU bandwidth than when using a single GPU, the data transfers would be more efficient if they were scheduled otherwise, especially in cases where the GPUs do not support bi-directional data transfer.

Elliott et al. presented GPUSync [3], a framework for scheduling the execution of real-time tasks in multi-GPU systems. The paper acknowledges the effect of bus contention on data transfer time, and the framework implements a conservative transfer time predictor that assumes worst-case contention at all times.

Verner et al. later showed that this resource sharing method (bandwidth allocation) can lead to serious under-utilization of the system interconnect [10]. They presented the Batch method for executing data transfers and computations on multi-GPU systems with high efficiency and predictable execution times. Given a set of data blocks to transfer and compute kernels, the method schedules and executes them, with the goal of minimizing the makespan (total execution time). To achieve high resource efficiency, the method executes some of the operations in parallel. The method also includes an accurate execution-time prediction algorithm that is based on a detailed model of the system interconnect. Therefore, it can be used as an execution engine with predictable execution times by a higher-level scheduler of real-time tasks. However, to make efficient use of the method, the higher-level scheduler needs to define batch jobs, i.e., groups of data transfer and compute operations that will be executed by the method.

III. System Model

This paper assumes a hard real-time system that receives multiple streams of periodically-arriving data packets and applies a processing procedure to each packet within a strict time limit from its arrival, using a platform that consists of CPUs and GPUs.

The next three subsections detail the system architecture, provide a formal definition of data streams, and describe the execution model.

A. System Architecture

In this work, we focus on a single-node multi-GPU system with an architecture described in Section II-A. The system consists of multiple CPUs and GPUs that are connected via a network of point-to-point buses and controllers, namely the system interconnect. A data transfer operation copies a data block of specified size between two endpoints (CPUs and GPUs) using the corresponding DMA controller. Multiple DMA controllers can execute data transfers in parallel, but data transfers that require the same DMA controller are transferred in FIFO order. Once a DMA transaction starts, it cannot be aborted.

B. Data Stream

A real-time data stream is a sequence of data packets that arrive to the system periodically at equal time intervals. Each packet needs to be processed by applying a procedure that can be decomposed into a series of communication and computation steps that use different system resources. Figure III.1 illustrates the processing steps in such a procedure. For simplicity, we assume that the same procedure is applied to all the packets in the stream. The processing of a packet must complete within a given time limit from its arrival (the relative deadline). In general, a data stream can consist of a finite or infinite number of data packets; in this work, we focus on infinite real-time data streams.

A data stream is characterized by the tuple $(size, period, (procstep_i)_{i=1}^{N})$, where $size$ is the size of each data packet, $period$ is the inter-arrival period, and $(procstep_i)_{i=1}^{N}$ is a list of N processing steps to be executed by the system, where each $procstep_i$ is a computation or communication step. In a computation step, compute kernels are executed on a set of processors; such a step is characterized by the set $\{(proc_k, kern_k)\}_{k=1}^{m}$. In a communication step, data blocks are transferred among CPU and GPU memories; these blocks are characterized by the set $\{(x_k, src_k, dst_k)\}_{k=1}^{n}$, where x_k denotes the size of the data block and (src_k, dst_k) denote the communication endpoints in the system. Note the difference between a data packet and a data block: a packet is a data chunk that arrives to the system from an external source, while a block is a data chunk that is transferred between processors.

In this work, we follow the offline scheduling scheme, meaning that the characterization of the data streams is known a priori. In addition, we assume a fixed assignment of computations to processors, and focus on the scheduling problem.

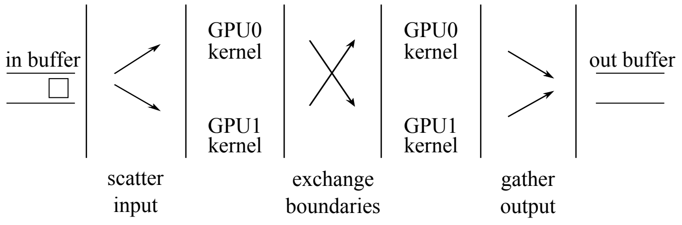


Figure III.1: Data stream processing steps

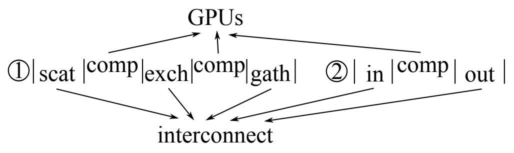


Figure III.2: Required resource for two data streams

C. Execution

In order to efficiently process multiple real-time data streams, the system (1) processes each stream in a synchronous pipeline, and (2) executes multiple operations on the same resource in parallel, a concept introduced for real-time tasks in the Batch method [10]. The system sets the cycle time in the processing pipeline of each data stream to $T = l/n$, where l is the stream's relative deadline and n is the number of processing steps. Consequently, the processing pipelines of different streams may be asynchronous.

Each processing operation requires a system resource, as illustrated in Figure III.2 for two data streams. In this example, each operation needs to be executed once in each cycle of its pipeline, and the system needs to share each resource (interconnect or compute unit) among all the operations that require it.

IV. Data Transfer Scheduling

The system processes multiple data streams by scheduling compute kernels and data transfers. Each such operation has a scheduling window of a pipeline cycle, in which it must execute. The time bounds of these scheduling windows are predefined because the pipelines have fixed cycle length. Therefore, data transfers and kernels can be scheduled independently. This paper focuses on the *data transfer* scheduling problem.

In classic task scheduling, data transfers over a shared medium are executed serially. The Batch method [10] proposes executing a batch of data transfers in parallel, and it provides an algorithm to compute the execution time of such data transfer operations.

In this section, we describe a task model for periodic data transfers, and present a new algorithm that optimizes the task set before it is fed to the scheduler. The algorithm lowers the resource demand of the task set by iteratively combining data transfers from multiple tasks into one. We use non-preemptive EDF as the task scheduling algorithm, and the Batch method as the execution engine. We have chosen a non-preemptive algorithm since, in our system model, data transfers cannot be aborted.

A. Message Model

Following previous studies on scheduling data transfers [6], [2], [9], we will use the term *message* to describe a data transfer task.

Definition IV.1. A message $m = (x, src, dst, p, a)$ is a source of data blocks of size x that arrive with period p starting at time a to be transferred from a source src to a destination dst before the arrival of the following block.

A message is, hence, an implicit-deadline periodic task (defined in Section II-B) characterized by (a, c, p), where $c = WCET(x, src, dst)$ is the worst-case time to transfer a data block of size x on the path $src \rightarrow dst$. We generalize the definition of a message to a *batch message* – an implicit-deadline periodic task where each job transfers one or more data blocks.

Definition IV.2. A batch message $m = (\{(x_k, src_k, dst_k)\}_{k=1}^{n}, p, a)$ is a source of data blocks that produces n data blocks every period p starting at time a, such that block k, of size x_k, needs to be transferred from src_k to dst_k before the arrival of the next n blocks.

A batch message is also an implicit-deadline periodic task that is characterized by (a, c, p), where $c = WCET(\{(x_k, src_k, dst_k)\}_{k=1}^{n})$. Since any message is also a batch message, we will henceforth use the term message for brevity when referring to any batch message.

B. Data Transfers to a Set of Messages

As discussed in Section III-C, each pipeline stage contains an operation that requires system resources, and this requirement needs to be satisfied before the beginning of the next pipeline cycle. Therefore, the requirement and time restrictions of the data transfer operations can be described as a set of messages. We call the set of messages, where each message describes a data transfer operation, the *elementary* message set. Note that other message sets can be used to represent the data transfer requirements in the system, but the elementary set is unique for any system; other representations can result, for example, from splitting a 12MB data transfer into four 3MB transfers.

Definition IV.3. An elementary message set $E = \{m_i : (x_i, src_i, dst_i, p_i, a_i)\}_{i=1}^{n}$ describes the data transfer requirement of the system. For each data transfer operation there is one corresponding message in E that represents its data transfer requirement and timing constraints.

One can view the elementary message set as the basic specification of the communication requirements from the system. The system can optimize these messages by splitting them into smaller data blocks and combining blocks from different messages into batch messages. By construction, a schedule of an elementary message set is valid if and only if it satisfies the system's requirements and time constraints for data transfers.

In order to optimize the communication, the system can generate other message sets that use the interconnect more efficiently; however, it needs to make sure that the message sets are compatible with the time constraints, i.e., the elementary set. For example, the following two message sets describe the same resource requirement:

$$E = \left\{ \begin{array}{c} (\text{10MB}, \text{CPU0}, \text{GPU1}, \text{15ms}, 0), \\ (\text{10MB}, \text{GPU1}, \text{CPU0}, \text{15ms}, 0), \\ (\text{15MB}, \text{CPU0}, \text{GPU2}, \text{20ms}, 0) \end{array} \right\}$$

$$M = \left\{ \begin{array}{c} \left(\left\{ \begin{array}{c} (\text{10MB}, \text{CPU0}, \text{GPU1}), \\ (\text{10MB}, \text{GPU1}, \text{CPU0}) \end{array} \right\}, \text{15ms}, 0 \right), \\ (\text{15MB}, \text{CPU0}, \text{GPU2}, \text{20ms}, 0) \end{array} \right\}.$$

Note that message set E has three messages with one data block transfer each, and M has two messages, one of which was generated by combining two messages in E.

C. Message Set Transformation Algorithm

So far, algorithms described in the literature schedule messages at the elementary set level. These schedulers do not overlap data transfers because it is assumed that each data transfer needs to use the interconnect exclusively. This causes the interconnect to be used inefficiently and may also cause the system to fail the schedulability test. Our goal is to reduce the execution time demand of the message set, hence *reducing* the load on the system, i.e., the utilization. Note the difference between *efficiency* and *utilization* – efficient data transfer makes full use of the bandwidth of the interconnect, while system utilization is considered to be too high if the time requirement for using the interconnect cannot be satisfied.

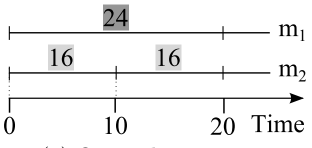
Algorithm 1 describes the message set transformation algorithm in our scheduler. The algorithm transforms the elementary message set into one with lower utilization by iteratively combining messages that can be transferred in parallel. It iteratively transforms the message set by combining messages, hence increasing the aggregate bandwidth and shortening the transfer time. The transforming operations, which will be described in the next subsection, preserve the message set's property of being compatible with the elementary message set. Our algorithm uses a greedy approach, where the optimization parameter is the *sum-utilization* of the message set ($\sum_i u_i$). The sum-utilization is the fraction of time that the resource is required over an infinite period of time.

D. Message Combining Operations

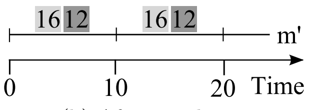
Two messages can be *combined* into a message that includes data transfers from both original messages. Combining messages can reduce the utilization without relaxing the scheduling restrictions. We distinguish between

Algorithm 1 Message set transformation algorithm

```
function R = reduceUtilization(E)
% E - Elementary message set
    R = E;
    gain = gainMatrix(R);
    % for msgs i,j, gain(i,j) stores
    % u(m_i) + u(m_j) - u(combine(m_i,m_j))
    % where u(m_i) is the utilization of m_i
    [dumax, i_m1, j_m2] = max(gain);
    while dumax > 0   % dumax: max util gain
        R = combine(R,i_m1,j_m2);
        gain = update(gain,R,i_m1,j_m2);
        [dumax, i_m1, j_m2] = max(gain);
    end
end
```



(a) Original message set



(b) After combining

Figure IV.1: Synchronous combining example

synchronous and *asynchronous* message combining. Synchronous combining is more efficient, but can only be applied if the messages fulfill certain alignment conditions that are not required in asynchronous combining.

1) Synchronous Combining: Consider a case of two messages illustrated in Figure IV.1a. The numbers in squares indicate the block sizes in MB and the vertical marks indicate their arrival times. The synchronous method suggests taking the arrival time of the message with the shortest period, m_2, as the baseline and fitting the other message to it. This is done by splitting m_2's data block into smaller blocks and aligning their scheduling bounds to m_2. As illustrated in Figure IV.1b, the block arrival times of the two messages are fully synchronous, so they can be combined. A necessary condition for synchronously combining messages is, hence, that the block arrival times of the message with the longer-or-equal period will coincide with those of the other message.

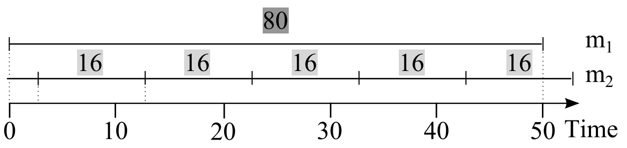
Definition IV.4. We define *synchronous combining* as an operation that receives two messages

$$m_1 = (\{(x_k, src_k, dst_k)\}_{k=1}^{n_1} \quad , p_1, a_1)$$
$$m_2 = (\{(x_k, src_k, dst_k)\}_{k=n_1+1}^{n_1+n_2}, p_2, a_2),$$

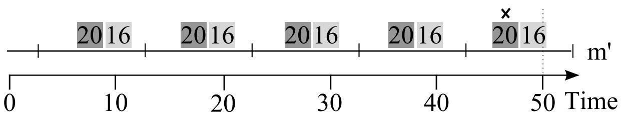
where $p_1 = np_2$ ($n \in \mathbb{N}^+$) and $a_1 \equiv a_2 \pmod{p_2}$, and returns

$$m' = \begin{pmatrix} \{(x_k/n, src_k, dst_k)\}_{k=1}^{n_1} \quad \cup \\ \{(x_k \quad , src_k, dst_k)\}_{k=n_1+1}^{n_1+n_2}, \\ p_2, \min(a_1, a_2) \end{pmatrix}.$$

2) Asynchronous Combining: Figure IV.2a shows an example of asynchronous combining. As illustrated, m_1, with the larger period, overlaps four full periods of m_2. Therefore, if we transfer one quarter of m_1's 80MB block in each of these periods, the data will be transferred before the deadline. We use this observation to combine the two messages into a single batch message (Figure IV.2b). Note that message m' reserves time for a 20MB transfer of m_1's data every 10ms, but the reserved time in the 10ms instances that are not fully overlapped by the larger period is not used at run-time.



(a) Original message set



(b) After combining

Figure IV.2: Asynchronous combining example

In general, let m_1 and m_2 be two messages with periods p_1 and p_2, such that $p_1 \geq p_2$, and these periods cannot be synchronously combined. Then, for some $n \in \mathbb{N}$ and $0 < \Delta p_{12} < p_2$, $p_1 = (n+1)p_2 + \Delta p_{12}$. We observe that, assuming arbitrary starting times, the minimum number of full p_2 periods overlapped by any instance of p_1 is n. Therefore, when combining the messages, we divide m_1's block to n pieces. Asynchronous combining can only be applied when the period of one message is at least twice as long as the other's, because otherwise a full overlap (i.e., $n \geq 1$) of one period by the other is not guaranteed.

Suppose a_1 and a_2 are the starting times of m_1 and m_2. If $a_1 \geq a_2$, then the starting time of the combined message is a_2. Otherwise, we need to align a_1 with the periods of m_2, i.e., to some time $t_i = a_2 - ip_2$; the formula for computing the starting time in this case is $\overline{a_2} = a_2 - \lceil (a_2 - a_1)/p_2 \rceil \, p_2$.

Definition IV.5. We define *asynchronous combining* as an operation that receives two messages,

$$\begin{aligned} m_1 &= (\{(x_k, src_k, dst_k)\}_{k=1}^{n_1} \quad , p_1, a_1) \\ m_2 &= (\{(x_k, src_k, dst_k)\}_{k=n_1+1}^{n_1+n_2}, p_2, a_2), \end{aligned}$$

such that $p_1 = (n+1)p_2 + \Delta p_{12}$ ($n \geq 0$, $p_2 > \Delta p_{12}$), and returns

$$m' = \begin{pmatrix} \{(x_k/n, src_k, dst_k)\}_{k=1}^{n_1} \quad \cup \\ \{(x_k \quad , src_k, dst_k)\}_{k=n_1+1}^{n_1+n_2}, \\ p_2, \overline{a_2} \end{pmatrix},$$

where $\overline{a_2} = a_2$ if $a_2 \leq a_1$, and $\overline{a_2} = a_2 - \lceil (a_2 - a_1)/p_2 \rceil \, p_2$ if $a_2 > a_1$.

V. Evaluation

We evaluate the efficiency of the proposed data transfer scheduler on two realistic applications from different domains: wafer inspection, and multi-stream encryption in real time. For each application, we demonstrate how the scheduler can be integrated into the system and measure its effect on performance.

A. *Application 1: Wafer Inspection*

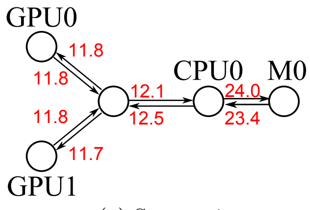
A common method in wafer-production inspection (wafer metrology) is to locate defects during the manufacturing process by analyzing images taken by high-resolution cameras. The inspection must keep pace with the production line and meet hard real-time deadlines. We built a model of a GPU-based inspection system using realistic data rates and compute requirements. Our system is composed of a set of high-resolution cameras connected to a number of multi-GPU compute nodes. Each camera produces images of size 200x1000 pixels with 16-bit encoding at a data rate of 360 MB/s (an image every 1.11 ms). The images from all the cameras are synchronously streamed into the nodes via fiberoptic-to-PCIe frame grabbers. Each image goes through five processing steps: (1) it is copied from the frame grabber to a GPU; (2) a series of image-processing algorithms, with an average demand of 1000 operations per pixel, are applied; (3) a synopsis of the results (only 4 KB per image) on each GPU is sent to the other GPUs; (4) the final defects list is produced; and (5) the list is copied to CPU memory. Steps 1 and 2 are much more time consuming than steps 3-5. The system processes each 'generation' of images synchronously on all GPUs. We consider compute nodes with four PCIe 2.0 slots (x16). One of them is used by the frame grabber and the others by NVIDIA K20m GPUs.

Prior to this work, the system processed the images one generation at a time, since previous attempts to do it in a pipeline failed due to contention for shared resources. To improve the system throughput, we first characterized the data streams according to the presented model, and derived an elementary message set to describe the communication requirements. Then, we applied Algorithm 1 to find a more efficient message set to represent the input. Finally, we used non-preemptive EDF to schedule the communication on the interconnect and the computations on each GPU.

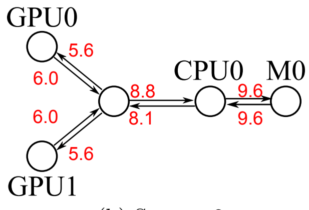
Results: By using the scheduler proposed in this work, the throughput of each compute node was increased by 67 %, with a processing capability of up to 15 cameras.

B. *Application 2: Stream Processing Framework*

We integrated the scheduling algorithm into a framework for processing asynchronous data streams under strict latency constraints on CPU/GPU systems [11], [12]. The user provides to the framework a characterization of a set of data streams that includes the data rate and latency bound (deadline) for each stream. The framework searches for a division of the streams into two groups – one to be processed by the CPUs and the other by the GPUs – that allows a valid schedule to be generated for both groups.



(a) System 1



(b) System 2

Figure V.1: Interconnect topology. Edge labels denote bandwidth in GB/s.

We integrated our scheduler into the framework and experimented with different system configurations and workloads. The framework uses a model of the system architecture to predict the execution times. Therefore, we were able to evaluate its performance on different systems by merely changing the configuration without executing on the actual hardware. The framework leaves sufficient error margins to cover for possible inaccuracies in execution time prediction, so any schedule that the framework verified to be valid can be executed on the system without deadline misses.

We examine the framework on a real streaming application, using two realistic hardware configurations, for a range of generated workloads that stress out the system.

1) System Configurations: We ran the experiments using the configurations of two existing systems. Both systems have a multi-core CPU and two NVIDIA GTX680 GPUs with one DMA controller each. The two systems differ in the bandwidth of their interconnects, which are described in Figure V.1.

2) Application: We evaluate the throughput of the framework for AES-CBC 128-bit encryption of multiple data streams. For this application, the CPU has a maximum throughput of 250 MB/sec, and each GPU has a maximum throughput of 7500MB/sec (1 MB=10^6 bytes).

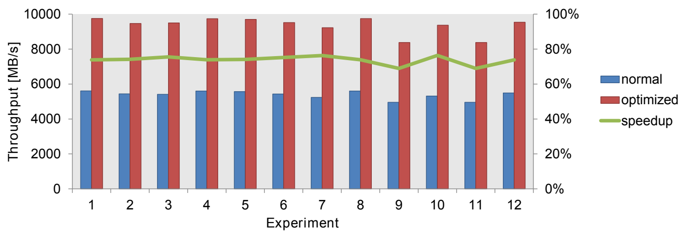
We generated a number of data stream pools, where in each pool the data rates and latency bounds of the streams were randomly generated according to one of the following distributions: constant, uniform, normal, and truncated exponential. For each distribution, we first randomly generated numbers in the range [0,1], and then projected them to the range of acceptable values. Data rates were projected to the range [0.2,6] Mbit/sec, and deadlines were projected to [0.5,200] ms. The settings for the generation of these workloads and the experiment results are described in Figure V.2.

3) Results: The two charts in Figure V.2 show the increase in throughput achieved by using the new data transfer scheduler in the two systems. In System 1, the framework was able to process 74 % more streams on

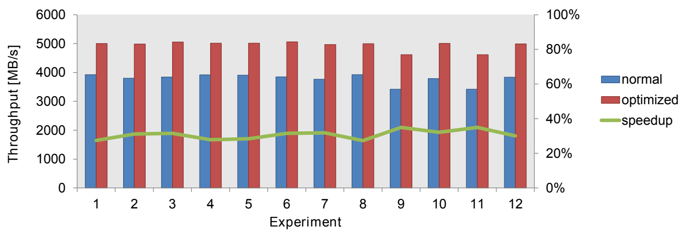
AES-CBC 128-bit encryption of multiple streams

1. constant 0.5
2. constant 0.1
3. uniform in [0,1]
4. uniform in [0.4,0.6]
5. normal AVG=0.5 STD=0.2
6. normal AVG=0.5 STD=1.0
7. normal AVG=0.2 STD=0.2
8. normal AVG=0.8 STD=0.2
9. exp STD=1.0 truncate at 5
10. exp STD=1.0 truncate at 1
11. exp STD=0.2 truncate at 1
12. deadlines: uniform in [0,1]
    data rates: constant 0.5

(a) Generation parameters for data rates and deadlines used in twelve experiments



(b) System 1



(c) System 2

Figure V.2: Throughput increase due to more efficient communication scheduling

average, and in System 2 the speedup was 31 %.

The scheduler automatically detected that some CPU-to-GPU and GPU-to-CPU data transfers can be overlapped, and that overlapping transfers in *opposite* directions for different GPUs gives the highest bandwidth. Overlapping all four transfers was not possible since bi-directional data transfer is not supported by these GPUs.

VI. Conclusions

In recent years, the compute power of GPUs grows faster than the interconnect bandwidth. As a result, the performance of many data-intensive applications is communication bounded. Previous works on real-time processing on GPU-based systems focused mainly on the problem of scheduling the computation. In this paper, we focused on the communication scheduling problem.

We presented a two-level scheduling model and an algorithm that automatically tunes the work distribution between the two levels to find the most efficient use of the system interconnect. The scheduling hierarchy allows this model to be both efficient and analyzable. The top level scheduler uses non-preemptive EDF to schedule batch jobs, and the bottom level uses the Batch method [10] to schedule the data transfers inside each batch job. Our algorithm iteratively combines tasks by joining their data transfers into batch jobs, hence moving more scheduling work towards the bottom-level scheduler, which makes use of parallel data transfer. We presented two methods for combining tasks, one for tasks with synchronous periods and the other for asynchronous. The methods construct the combined task such that its scheduling constraints are at least as strict as the original constraints, which guarantees that any valid schedule that is found for the task set will not cause deadline misses when applied in practice.

Our experiments show that our scheduler improved the overall performance of two realistic real-time systems by factors of between 31 % and 74 %.

Acknowledgments

This work was supported by the Metro450 Israeli national consortium (http://www.metro450.org.il).

References

[1] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proc. of the 11th Real-Time Systems Symposium*, pages 182–190, 1990.

[2] Marco Di Natale and Antonio Meschi. Scheduling messages with earliest deadline techniques. *Real-Time Systems*, 20:255–285, 2001.

[3] Alenn A. Elliott, Bryan C. Ward, and James H. Anderson. GPUSync: A Framework for Real-Time GPU Management. *The 34th IEEE Real-Time Systems Symposium*, pages 33–44, December 2013.

[4] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data Transfer Matters for GPU Computing. In *19th IEEE International Conference on Parallel and Distributed Systems*, 2013.

[5] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. *Real-Time Systems Symposium*, (December):129–139, 1991.

[6] John P. Lehoczky and Lui Sha. Performance of real-time bus scheduling algorithms. *ACM SIGMETRICS Performance Evaluation Review*, 14(1):44–53, May 1986.

[7] Chang L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, (1):46–61, 1973.

[8] Michael Minelli, Michele Chambers, and Ambiga Dhiraj. *Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses.* Wiley CIO. Wiley Publishing, 2013.

[9] Oliver Sinnen and Leonel A. Sousa. Communication Contention in Task Scheduling. 16(6):503–515, 2005.

[10] Uri Verner, Avi Mendelson, and Assaf Schuster. Batch Method for Efficient Resource Sharing in Real-Time Multi-GPU Systems. In *Distributed Computing and Networking*, volume 8314 of *LNCS*, pages 347–362, 2014.

[11] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the 2011 International Conference on Supercomputing*, pages 120–129, May 2011.

[12] Uri Verner, Assaf Schuster, Mark Silberstein, and Avi Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. *Proc. of the 5th Annual International Systems and Storage Conference*, pages 1–12, June 2012.

[13] Omar U. P. Zapata and Pedro M. Alvarez. EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation. *Queue*, pages 1–24, 2005.

[14] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. 58(9):1250–1258, 2009.