Processing Real-time Data Streams on GPU-based Systems

Uri Verner
Processing Real-time Data Streams on GPU-based Systems

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Uri Verner

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tevet 5776 Haifa December 2015
This research was carried out under the supervision of Prof. Assaf Schuster and Prof. Avi Mendelson, in the Faculty of Computer Science.

**LIST OF PUBLICATIONS**

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


**ACKNOWLEDGEMENTS**

I would like to thank my advisors, Prof. Assaf Schuster and Prof. Avi Mendelson, for their wise support, guidance, and life experience. I would also like to thank Prof. Mark Silberstein for interesting me in this subject and mentoring me during the first two years of my graduate studies; his effort and contribution to my research are greatly appreciated. My studies would not have been as pleasant without the devoted care of Yifat and Sharon. I enjoyed working with you and thank you wholeheartedly for your help.

Finally, I would like to thank my family for their love and support. I thank my grandparents for being a source of inspiration to me. I thank my father for his advice and encouragement. Lastly, I wish to thank my wife Sveta for her understanding, love, support, and patience, which enabled me to complete this work.

The generous financial help of the Metro 450 consortium, the Hasso Plattner Institute, and the Technion is gratefully acknowledged.
Contents

List of Figures

List of Tables

List of Listings

Abstract

Abbreviations and Notations

1 Introduction

1.1 Motivating Example: Wafer Production Inspection

1.2 Heterogeneous Systems

1.3 Real-Time Systems

1.4 Stream Processing

1.5 Contribution

1.5.1 Hard Real-Time Data Stream Processing Framework

1.5.2 CPU/GPU Work Distribution

1.5.3 Communication Scheduling on Heterogeneous Interconnects

1.5.4 Generalization to Scheduling Parallel Resources

1.6 Organization

2 Background and Related Work

2.1 Heterogeneous Systems

2.1.1 Multicore CPUs

2.1.2 Graphics Processing Units (GPUs)

2.1.3 CPU-GPU Systems

2.1.4 Multi-GPU Systems

2.1.5 System Interconnect

2.2 Real-Time Scheduling

2.2.1 Introduction

2.2.2 Task Model

2.2.3 Computing Platform Model
2.2.4 Other Scheduling Problems ........................................ 29

2.3 Real-Time Scheduling in CPU-GPU Systems ...................... 30
  2.3.1 Overview ...................................................... 30
  2.3.2 Limited Control over GPU Resources ......................... 31
  2.3.3 Estimation of Execution Time ................................ 33
  2.3.4 Resource Scheduling .......................................... 34

3 System Model ......................................................... 37
  3.1 Hardware Model .................................................. 37
  3.2 Real-Time Data Stream .......................................... 37
  3.3 Packet Processing Model ........................................ 38

4 Stream Processing in a CPU-GPU System ............................ 41
  4.1 GPU Batch Scheduling ............................................ 42
    4.1.1 Schedulability .............................................. 43
    4.1.2 GPU Empirical Performance Model ......................... 43
  4.2 CPU Scheduling .................................................. 45
    4.2.1 Scheduling Algorithms ..................................... 45
  4.3 RECTANGLE Method for CPU/GPU Stream Distribution ............ 51
    4.3.1 Considerations in Workload Distribution Between a CPU and a GPU 51
    4.3.2 RECTANGLE Method ........................................ 53
    4.3.3 GPU Performance Optimizations ............................ 54
    4.3.4 Real-Time Stream Processing Framework .................. 56
  4.4 Evaluation ....................................................... 57
    4.4.1 Experimental Platform ..................................... 57
    4.4.2 Setting Parameters ......................................... 58
    4.4.3 Results ..................................................... 58
    4.4.4 Performance Model ......................................... 61
    4.4.5 Time Breakdown ............................................ 61
    4.4.6 How Far Are We From the Optimum? ......................... 62
  4.5 Related Work ..................................................... 62
  4.6 Conclusions ....................................................... 64
  4.7 Rectangle Method Algorithm Pseudo-code ........................ 65

5 Stream Processing in a Multi-GPU System ............................ 67
  5.1 Virtual GPU ..................................................... 67
  5.2 Split Rectangle .................................................. 69
  5.3 Joined Rectangles ............................................... 70
  5.4 Loose Rectangles ................................................ 70
  5.5 Evaluation of Virtual GPU ...................................... 70
    5.5.1 Experimental Platform ..................................... 70
    5.5.2 System Performance ......................................... 71
5.5.3 Comparison to Baseline Methods ............................................. 71
5.5.4 Scaling to More GPUs ............................................................. 73
5.5.5 Transfer Rates ................................................................. 74
5.5.6 Accuracy of Performance Model .......................................... 74
5.6 Summary .............................................................................. 76

6 Data Transfer Scheduling on the Interconnect ................................. 79
6.1 Introduction ........................................................................... 79
6.2 Problem Definition .................................................................. 82
   6.2.1 System Model ................................................................. 82
   6.2.2 Problem Statement .......................................................... 86
6.3 Data Transfer Scheduling Method: Overview ................................ 86
6.4 Batch Method for Data Transfer and Computations ....................... 87
   6.4.1 Sharing Resources in Real-time Systems .............................. 89
   6.4.2 Interface ......................................................................... 90
   6.4.3 Implementation ............................................................... 91
   6.4.4 Performance of the Batch Method for the Example in Section 6.4.1 . 95
   6.4.5 Evaluation ................................................................. 95
   6.4.6 Discussion ................................................................. 101
6.5 Controlling Bandwidth Distribution to Reduce Execution Times ....... 102
   6.5.1 Problem Definition .......................................................... 103
   6.5.2 Formula for the Optimal Bandwidth Distribution Factor ........... 104
   6.5.3 Evaluation of Bandwidth Distribution Method ....................... 105
   6.5.4 Conclusions .................................................................. 107
6.6 Task Set Optimization .............................................................. 108
   6.6.1 Data Transfers to a Set of Batch Messages ....................... 108
   6.6.2 Message Set Transformation Algorithm ................................ 109
   6.6.3 Message Combining Operations ........................................ 109
   6.6.4 Message Set Scheduling and Validity Test ............................ 111
6.7 Evaluation ........................................................................ 112
   6.7.1 Application 1: Wafer Inspection ....................................... 112
   6.7.2 Application 2: Stream Processing Framework ...................... 112
6.8 Conclusions ...................................................................... 115
6.9 MATLAB Implementation .......................................................... 117
   6.9.1 Batch Run-Time Scheduling Algorithm .......................... 117
   6.9.2 Interconnect Model ........................................................... 121
   6.9.3 Computation of Data Rates ............................................. 124
   6.9.4 Non-Preemptive EDF Schedulability Test ........................... 126
# Extended Multiprocessor Model for Efficient Utilization of Parallel Resources

## 7 New Class of Scheduling Problems

### 7.1 Introduction

### 7.2 Machine Environment

### 7.3 Job Characteristics

### 7.4 Optimality Criteria

### 7.5 Summary

## 7.3 Scheduling with Turbo Boost

### 7.3.1 Turbo Boost Overview and Related Works

### 7.3.2 Model and Scheduling Problem Definition

### 7.3.3 Existing Scheduling Techniques for Identical Machines

### 7.3.4 Scheduling Challenges of the New Model

### 7.3.5 Frequency-Aware Scheduling Method (FAS)

## 7.4 Validity and Optimality

### 7.4.1 Validity

### 7.4.2 Optimality

## 7.5 Evaluation

### 7.5.1 Methodology

### 7.5.2 Accuracy of the Performance Model

### 7.5.3 Synthetic Benchmarks

### 7.5.4 Web-log analysis

### 7.5.5 Simulation of Performance on Other CPUs

## 7.6 Extending Amdahl’s Law for Multicores with Turbo Boost

### 7.6.1 Introduction

### 7.6.2 Background

### 7.6.3 Extended Amdahl’s Law

### 7.6.4 Evaluation

## 7.7 Discussion and Future Work

## 8 Discussion and Conclusion

# Hebrew Abstract
List of Figures

1.1 Wafer metrology tool .................................................. 8
1.2 Main topics addressed in this dissertation .......................... 12

2.1 GPU architecture example: NVIDIA Maxwell GM204 .......... 18
2.2 GPU memory hierarchy .................................................. 18
2.3 Architecture of a heterogeneous multi-GPU compute node ....... 24

3.1 Data stream model ....................................................... 38
3.2 Processing model ......................................................... 38

4.1 Pipelined processing on a GPU ........................................ 42
4.2 SM utilization is non-linear in the number of jobs .................. 44
4.3 Effective utilization of jobs with different sizes .................... 44
4.4 Data transfer rates in a CPU-GPU system with Intel P45 chipset .... 46
4.5 CPU/GPU stream partitioning example using the Rectangle method ...... 53
4.6 Influence of rate variance on kernel execution time ............... 55
4.7 Performance as a function of the number of CUDA thread blocks .... 56
4.8 Evaluation of workload distribution methods for stream sets with exponential rate distribution with average 500ms .............. 58
4.9 Comparison of workload distribution methods for stream sets with exponential rate distribution with average 50ms .................. 59
4.10 Evaluation of workload distribution methods for normal distribution .... 60
4.11 Evaluation of workload distribution methods for low level of parallelism ... 60
4.12 Evaluation of workload distribution methods for stream sets with different numbers of streams .............................................. 60
4.13 Determining a safety margin for kernel time estimation .......... 61
4.14 GPU breakdown of pipeline stages for exponential and normal rate distributions .................................................... 62
4.15 GTX285 optimal throughput for AES-CBC stream processing ....... 63
4.16 Throughput for AES-CBC stream processing using a GTX285 and two CPU cores ......................................................... 63

5.1 Illustration of multi-GPU distribution methods ...................... 68
5.2 Mapping streams to SMs of a virtual GPU .......................... 68
5.3 Evaluation of Virtual GPU method ..................................... 72
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Performance model estimation error</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Multi-GPU distribution methods</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Maximum transfer rates and kernel throughput</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance model estimation error</td>
<td>76</td>
</tr>
<tr>
<td>6.1</td>
<td>Performance comparison of the different methods</td>
<td>100</td>
</tr>
<tr>
<td>7.1</td>
<td>Classic machine type classification</td>
<td>131</td>
</tr>
<tr>
<td>7.2</td>
<td>New machine environment types</td>
<td>133</td>
</tr>
<tr>
<td>7.3</td>
<td>Classification of scheduling problem and selection of scheduling algorithm</td>
<td>139</td>
</tr>
<tr>
<td>7.4</td>
<td>Work redistribution effect on execution time</td>
<td>147</td>
</tr>
<tr>
<td>7.5</td>
<td>Frequency specification: 8-core Intel Xeon E5-2690</td>
<td>158</td>
</tr>
<tr>
<td>7.6</td>
<td>Frequency specification: 12-core Intel Xeon E5-2658 v3</td>
<td>158</td>
</tr>
<tr>
<td>7.7</td>
<td>Execution time [sec]</td>
<td>160</td>
</tr>
<tr>
<td>7.8</td>
<td>Package energy consumption [J]</td>
<td>160</td>
</tr>
</tbody>
</table>
# List of Listings

2.1 GPU kernel for computing the cubes ($x^3$) of floating-point numbers (CUDA C) 20
2.2 GPU management code for kernel example ................................. 20
4.1 Earliest Deadline First - First Fit (EDF-FF) ............................. 48
4.2 RECTANGLE method algorithm pseudo-code ............................. 65
6.1 API for batch operations ....................................................... 91
6.2 Transfer rate computation for concurrent data transfers ............... 94
6.3 Message set transformation algorithm ................................... 109
6.4 Batch run-time scheduling (MATLAB) ..................................... 117
6.5 Interconnect model (MATLAB) ................................................ 121
6.6 Computation of data rates (MATLAB) ..................................... 124
6.7 Non-preemptive EDF schedulability test (MATLAB) ..................... 126
Abstract

Processing massive streams of data is an important problem in modern computer systems, and in particular for applications that process big data. Many such applications, including data analytics, production inspection, and fraud detection, require that the response times be below given thresholds. Meeting these constraints makes the scheduling problem a very challenging task. Each data stream generates a sequence of data packets that need to be processed within a given time from its arrival. GPUs are very promising candidates for providing the required processing power. However, their use for processing real-time streams is complicated by various factors, such as obscure operation mechanisms, complex performance scaling, and contention for bandwidth on a shared interconnect.

The goal of this dissertation is to develop an approach to efficient processing of real-time data streams on heterogeneous computing platforms that consist of CPUs and GPUs. To achieve this goal, we develop and investigate a processing model, scheduler, and framework. We present several work distribution techniques that statically assign the streams to the CPU or GPU in a way that simultaneously satisfies their aggregate throughput requirements and the deadline constraint of each stream alone. These methods serve as the basis for four new methods for partitioning work in multi-GPU systems, each presenting different compromise between the complexity of the algorithm and achievable throughput. We design a generic real-time data stream processing framework that implements our methods, and use it to evaluate them with extensive empirical experiments using an AES-CBC encryption operator on thousands of streams. The experiments show that our scheduler yields up to 50% higher system throughput than alternative methods.

Another major challenge in using multiple GPUs for real-time processing is in scheduling recurrent real-time data transfers among CPUs and GPUs on a shared interconnect. We develop a new scheduler and execution engine for periodic real-time data transfers in a multi-GPU system. The scheduler is based on a new approach where the basic unit of work sent for execution is a batch of data-block transfer operations. In our experiments with two realistic applications, our execution method yielded up to 7.9x shorter execution times than alternative methods. The scheduler analyzes the data transfer requirements and produces a verifiable schedule that transfers the data in parallel, and achieves up to 74% higher system throughput than existing scheduling methods.

The CPU contributes to the processing capability of the system by providing easily managed and predictable compute power. However, a recent trend in CPU design is to include a
frequency scaling mechanism such as Turbo Boost that changes the performance-scaling curve. We characterize the processing speeds with Turbo Boost, and present an offline task scheduler that minimizes the total execution time. We also extend Amdahl’s law for speedup and energy consumption to take into account the effects of Turbo Boost. Finally, we generalize the new resource model and define a new class of scheduling problems that enables more efficient use of parallel resources by accurately characterizing their performance, thereby laying the foundation for further research.
Abbreviations and Notations

ALU — Arithmetic Logic Unit
ASIC — Application-Specific Integrated Circuit
API — Application Program Interface
CBC — Cipher Block Chaining
CPU — Central Processing Unit
CU — Compute Unit
DBF — Demand Bound Function
DDR — Double Data Rate
DIMM — Dual In-line Memory Module
DMA — Direct Memory Access
DRAM — Dynamic Random-Access Memory
EDF — Earliest Deadline First
FIFO — First In, First Out
FLOPS — Floating Point Operations per Second
FP — Floating Point
FPGA — Field-Programmable Gate Array
Gbit — Gigabit \((10^9 \text{ bits})\)
GDDR — Graphics Double Data Rate
GOp — Giga Operations \((10^9 \text{ operations})\)
GPGPU — General-Purpose computation on Graphics Processing Units
GPU — Graphics Processing Unit
GUI — Graphical User Interface
HPC — High-Performance Computing
I/O — Input / Output
ISA — Instruction Set Architecture
IP — Internet Protocol
LP — Linear Programming
Mbit — Megabit \((10^6 \text{ bits})\)
MIC — Many Integrated Core
ms — millisecond or milliseconds
NIC — Network Interface Card
Op — Operation(s)
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCI Express</td>
</tr>
<tr>
<td>PTX</td>
<td>Parallel Thread Execution</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QPI</td>
<td>QuickPath Interconnect</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RDMA</td>
<td>Remote Direct Memory Access</td>
</tr>
<tr>
<td>RM</td>
<td>Rate Monotonic</td>
</tr>
<tr>
<td>RTSPS</td>
<td>Real-time Data Stream Processing System</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous Dynamic Random-Access Memory</td>
</tr>
<tr>
<td>sec</td>
<td>second or seconds (time unit)</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single-Instruction Multiple-Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single-Instruction Multiple-Threads</td>
</tr>
<tr>
<td>SLA</td>
<td>Service-Level Agreement</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Socket Layer</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Modern data centers experience a continuously increasing demand for real-time processing of data streams that originate from “big data” sources such as social networks, web traffic, video streams, financial markets, and sensor arrays. Processing the data streams fast and in large volumes produces new high-value insights that enable systems and organizations to quickly react to changing conditions. Real-time data stream processing systems (RTSPSs) apply processing operations on such streams on-the-fly under strict processing-latency constraints posed by the application. RTSPSs can solve problems of fraud detection, algorithmic trading, production inspection, video and sensor monitoring, secure voice-over-IP (VoIP), risk management, and many others. Due to the growing demand for real-time stream processing, many of the applications require a high-performance computing platform designed to provide high throughput at sustainable power consumption. Recently, heterogeneous multiple-processor compute nodes have become prevalent in such platforms, due to their ability to effectively handle various types of workloads. Especially notable is the widespread adoption of GPUs, which effectively handle the highly parallel parts of the computation, and are thus prime candidates for processing multiple data streams. Despite their performance advantage, efficiently utilizing heterogeneous platforms that include such compute accelerators under strict latency constraints, as required in RTSPSs, is challenging due to their diverse and asymmetric architecture, their high latency, and the complexity of the accelerators.

The goal of this dissertation is to develop an approach to efficient processing of real-time data streams on a computing platform that consists of CPUs and GPUs.

1.1 Motivating Example: Wafer Production Inspection

Semiconductor devices are made on round silicon wafers. The principal process in wafer fabrication is the creation of layers of patterns that implement circuits. The creation of a layer is performed in a series of steps that include depositing material onto the surface of the wafer, lithography (creating patterns using light), and etching (removal of redundant material). The patterns are created at very high resolution and the production process must be very accurate.
Due to the serial nature of wafer processing, the wafers are measured and inspected for damage in between production steps in order to avoid redundant processing. The wafer inspection process (metrology) is limited in time, as it must keep up with the production line. The metrology process includes measurement of the critical dimensions of the pattern (length of a transistor gate), searching for fabrication defects and their classification, and thin film measurement. Figure 1.1 illustrates a wafer metrology tool. The inspected area is accurately positioned below an array of sensors such as high-resolution line cameras and x-ray sensors, which collect measurement data at rates of multiple giga-pixels per second. The measurement data is streamed into a nearby computing platform for processing by way of image processing, model reconstruction, or methods that determine whether the wafer has passed the test.

From a computing perspective, the workload can be characterized as multiple data streams with high processing demands and strict processing latency constraints. Traditionally, such computing platforms were based on customized electronic boards or CPU-based rack servers. Current compute technologies can hardly handle the data generated in current process technology and with a wafer size of 300 mm. For future process generations and wafer size of 450 mm [Jon15], new data transfer and processing power technologies may be needed to meet these requirements.

1.2 Heterogeneous Systems

In modern computer systems, performance is achieved through parallelism and heterogeneity. Many data centers consist of a set of heterogeneous machines connected by a high-speed network. Each machine contains an interconnected set of processing units, memory, and I/O devices. The processing units can be a combination of general purpose processors, many-core
stream processors (GPUs), digital signal processors (DSPs), application-specific integrated circuits (ASICs), field programmable gate arrays (FPGAs), x86-based coprocessors, and other specialized accelerators. Each processing unit type has distinct architectural features that make it the better choice for certain types of workloads when considering, for example, computational throughput, latency, power efficiency, and cost. Making effective use of the diverse suite of processing units is a challenging task that involves work partitioning and mapping, scheduling, and synchronization, among others [KPSW93]. As a case study for a high-performance heterogeneous system, this work examines multi-GPU workstations that combine CPUs and discrete GPUs.

GPUs have gained popularity as efficient accelerators in a number of data-intensive application categories, including high-performance computing (HPC), image processing, and machine learning. Starting from fixed-function hardware designed for 2D and 3D graphics, GPUs evolved into fully-programmable many-core processors capable of general-purpose computations. Architecturally, GPUs are different from CPUs; besides having many more execution units than a CPU, each execution unit in a GPU interleaves tens of data-parallel threads to hide the latency of long-running operations such as accesses to off-chip memory. GPUs are natural candidates for speeding up processing of multiple data streams in parallel. However, their unique architectural characteristics hinder their straightforward application in stream processing systems and introduce further complications with the addition of per-stream latency bounds. Efficient use of the GPU’s computational resources for processing multiple data streams is complicated by the following factors:

- **Low single thread performance** – GPUs are optimized for throughput at the cost of single-thread performance. Thus, guaranteeing restricted processing times of individual streams is challenging.

- **Load balancing issues** – to reduce the amount of control logic, GPUs group execution units in single-instruction multiple-data (SIMD) arrays. Each such array executes threads in groups called *warps* (in CUDA terminology), one at a time, such that all the threads in a warp execute the same serial function on different data. When some thread in a warp completes execution earlier than others, its subsequent cycles on the execution unit are wasted, until the last thread in the warp completes execution. The streams can produce data items at different rates that result in computational tasks with proportionally different sizes. Thus, it is hard to assign the tasks to threads in a way that balances the load between the execution units.

- **Non-linear throughput scaling** – for better hardware utilization, GPUs require many concurrently active warps. Thus, effective throughput depends non-linearly on the number of concurrently processed streams, making run-time prediction difficult.

- **Limited kernel concurrency** – current NVIDIA GPUs limit the number of concurrent kernel executions. Therefore, data streams need to be batched and processed synchronously. However, data streams are asynchronous by nature in both their data arrival
times and deadlines. Thus, processing all the streams within their individual time limits is challenging. Moreover, since GPU thread scheduling is managed in hardware, it is hard to predict the execution time of concurrent kernel calls.

These factors need to be addressed to make efficient use of the GPU’s computational resources for processing multiple data streams.

In a CPU-GPU compute node, the system interconnect consists of linked domains where the components use the same communication technology, such as PCI Express, QuickPath Interconnect (QPI), HyperTransport, memory buses, and NVLink [Fol14]. In modern architectures, including in multi-GPU systems, the compute power is growing much faster than the communication bandwidth, which generates increasingly higher pressure on the interconnect. Thus, making effective use of the interconnect is crucial for performance. However, its diverse architecture makes it difficult to attain predictable data transfer times when multiple data transfers are in progress on intersecting paths, likewise making it difficult to use the available bandwidth efficiently. This work also addresses the problem of effective use of the interconnect for data transfers with predictable transfer times.

1.3 Real-Time Systems

The logical correctness of real-time systems is verified on a model of the system comprised of jobs and processors. A job denotes a processing step that can execute on a single processor for a finite amount of time, and has a deadline, which is the time at which its execution must be finished. In real-time computing, heterogeneous multiprocessor computing platforms are modeled as unrelated parallel machines [CPW99, Bur91, SAA+04]. According to this model, the platform consists of processors that execute jobs independently and may have different execution times for the same job. However, GPU-based RTSPSs are incompatible with previous models in a number of ways:

- **Batch processing** – GPUs are designed to work in batch mode, i.e., execute groups of jobs, while in the classic model processors execute one job at a time.

- **Variable job execution time** – The execution time of a job on a processor is fixed in the classic model [Fun04, Rar14]. As we show later in this work, this model incorrectly represents the performance of CPUs and GPUs on workloads that are relevant to stream processing.

- **Heterogeneous interconnect** – The interconnect of a multi-GPU compute node spans several communication domains that use different technologies. This heterogeneity complicates the computation of the expected latency for data transfer operations and fundamentally changes the communication scheduling problem studied in previous works.

In hard real-time systems, a rigorous analysis of the timing of all the critical system components is required. Currently, commercially available programmable GPUs do not enable such analysis.
and are only suitable for non-critical real-time systems. The timing analysis in this work is done under the assumption that the system works reliably and with consistent performance.

The application of methods developed for the unrelated parallel machines model in an RT-SPS results in low throughput and may lead to violation of temporal correctness, due to inefficient utilization of the computation and communication resources as well as imprecise performance modeling. This work aims to develop new methods that guarantee temporal correctness for all the admitted streams and maximize the system throughput under this constraint.

1.4 Stream Processing

Stream processing is a difficult problem from the system design perspective. This is because the data processing rate should not fall behind the aggregate throughput of arriving data streams, otherwise leading to buffer explosion or packet loss. Stateful processing, which is the focus of this work, also requires the previous processing results to be available for computations on newly arrived data. Even more challenging is the problem of real-time stream processing. In such applications, each stream specifies its processing latency bound, which restricts the maximum time arrived data may stay in the system. This requirement fundamentally changes the system design space, rendering throughput-optimized stream processing techniques inappropriate for several reasons. First, tight deadlines may prevent computations from being distributed across multiple computers because of unpredictable network delay. Furthermore, schedulability criteria must be devised in order to predetermine whether a given set of streams can be processed without violating their deadline requirements and exceeding the aggregate system throughput. Runtime predictions must thus take into account every aspect of the processing pipeline. A precise and detailed performance model is therefore crucial. The runtime prediction problem, hard in the general case, is even more challenging here: to allow deadline-compliant processing, the predictions must be conservative, in conflict with the goal of higher aggregate throughput. Many of the challenges of stream processing are common to various real-time data stream applications; hence this work considers a generic workload model where each stream represents a periodic source of data packets that need to be processed within a given latency bound.

1.5 Contribution

This dissertation presents new methods that enable efficient utilization of high-performance heterogeneous platforms in RTSPSs, and the main topics it addresses are summarized in Figure 1.2.

1.5.1 Hard Real-Time Data Stream Processing Framework

We design a framework for hard real-time processing of multiple streams on heterogeneous platforms with multiple CPUs and a GPU accelerators. This was the first use of accelerators in
hard real-time computations [MV15]. The core of the framework is an algorithm that finds an assignment of input streams to processors that satisfies the latency-bound requirements. The algorithm partitions the streams into subsets, one for each GPU and another for all the available CPUs. Each subset is validated to be schedulable using an accurate performance model of the system, taking into account all the steps of the computation and the inter-processor communication. We implemented the stream processing framework in C/C++ and MATLAB and tested it on a multiple-stream AES-CBC encryption service that ran on a computing platform comprised of Intel CPUs and NVIDIA GPUs.

1.5.2 CPU/GPU Work Distribution

Non-linear throughput scaling in accelerators makes the stream partitioning problem computationally hard in general, even in a system with a single GPU, as it requires every two subsets of streams to be tested for schedulability. The exact solution requires exhaustive search in an exponential space of subsets and does not scale beyond a few input streams.

We develop a fast polynomial-time heuristic for scheduling thousands of streams on a system with a single GPU, and later extend it to multi-GPU systems. Each stream is represented using its deadline and rate properties as a point in the two-dimensional (rate-deadline) space. The heuristic finds a rectangle in this space, so that all the streams with their respective points in the rectangle are schedulable on the accelerator, and the remaining streams are schedulable on the CPUs. We show that this simple heuristic effectively handles many performance-critical challenges in CPU/GPU work distribution. With multiple GPUs, the distribution of streams can be optimized by using multiple rectangles; however, this further increases the search complexity and introduces the need for efficient scheduling of the communication on the interconnect. We develop several methods for stream partitioning in multiple-GPU systems that provide different tradeoff points between search complexity and throughput.
1.5.3 Communication Scheduling on Heterogeneous Interconnects

Each stream periodically produces data-packets that require processing from the system. The processing includes data transfer operations, such as its transfer to GPU memory, and computational operations that are executed on the CPUs and GPUs. Scheduling all these operations together to make efficient use of all the different resources is a very difficult task. We show that the problem can be alleviated using a pipeline-based approach that splits the problem into two simpler problems – computation scheduling and communication scheduling. Such splitting becomes possible because the two types of operations use different system components.

Efficient use of the interconnect is essential for a high-throughput system. However, the topology of the interconnect is such that transferring data on multiple paths at the same time, which is necessary to achieve high bandwidth, makes it difficult to predict the execution time of each individual data transfer operation, required for temporal correctness.

To attain both efficiency and guaranteed temporal correctness, we defined a new concept that we call batch periodic data-transfer tasks. Each invocation of the task combines the data transfer needs of multiple data streams. Batch tasks can achieve high bandwidth by transferring data on different paths at the same time, while having predictable execution times. We develop a two-level scheduler for the communication part of stream processing. The top level binds the communication requirements of data streams into batch tasks and schedules them using a version of the earlier-deadline-first (EDF) algorithm. The bottom level schedules the data transfers in the batch, with the aim of optimizing the total execution time.

We build an accurate performance model of the system interconnect in a multi-GPU system. For each bus, our model computes the distribution of bandwidth between the active data transfer operations, and it is able to reliably predict the completion time of each data transfer operation when multiple such operations are in progress. The model is configured with a topology graph of the interconnect that includes the bandwidth on each bus. This graph can be generated using our set of targeted data transfer benchmarks that measure the effective bandwidth in various combinations of simultaneous data transfers.

1.5.4 Generalization to Scheduling Parallel Resources

In scheduling, parallel resources such as multi-core CPUs are modeled as a set of processors, where each processor executes jobs one at a time independently of the other processors. In this model, assuming equal-speed processors, the throughput is linear in the number of active processors. Our performance analysis of GPUs and the interconnect shows that, in these resources, the throughput scales sub-linearly with the number of active execution units. We then show that this observation is also true for multi-core CPUs. Most modern CPUs include an integrated frequency scaling mechanism such as Turbo Boost. Turbo Boost opportunistically increases the cores’ operating frequency (speed) when the power, current, and temperature of the chip are below certain limits. Effectively, higher frequencies are achieved when fewer cores are active, so the speed of each core depends on the number of active cores; when more cores are active, each core works slower. A failure to account for dependencies between the processing units in
the resource model used for scheduling negatively affects the utilization of the resource, even with an optimal scheduling algorithm. We define a new resource model for scheduling that characterizes the speed of each processor as a function of the state of the other processors (such as the number of active processors). To the best of our knowledge, our work was the first to study the problem of scheduling multiprocessors where the speed of any processor depends on the other processors.

Our contributions in this area are as follows:

- We propose an extension of Graham’s $\alpha|\beta|\gamma$ classification of scheduling problems [GLLRK79] to a new class of problems, where processor speeds depend on the execution state of the multiprocessor. These new problems model many hardware resources more accurately and enable higher utilization.

- We characterize the computational throughput of CPUs with Turbo Boost technology and define a minimum-makespan scheduling problem that can be applied to reduce execution time in a number of applications. For this problem, we develop a new efficient scheduling algorithm, prove its optimality for three main sub-classes of the problem, and show that it achieves shorter makespan than existing methods.

- We show that different CPU models need to be scheduled differently.

- We extend Amdahl’s law for multicore processors with Turbo Boost. Amdahl’s law and its extension for energy are commonly used to obtain an initial evaluation of the expected maximum speedup and energy improvement that result from parallelizing sequential codes. Our refined formulas provide a tight upper bound on the expected parallelization benefits, closing a gap of up to 16% between the actual and predicted values.

1.6 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides background on system architecture and scheduling, and reviews relevant prior work. Chapter 3 describes the system model that is considered in this work. Chapters 4 and 5 present our stream processing framework and its underlying methods for CPU/GPU work distribution in single- and multi-GPU systems. Chapter 6 discusses the challenges in scheduling data transfers on the interconnect for real-time data streams, and presents our two-level scheduler and execution engine for hard real-time data transfers. Chapter 7 lays the foundation for a new class of scheduling problems where processors influence each other’s performance, using the problem of makespan minimization of compute tasks on multi-core CPUs as a case study. Finally, Chapter 8 summarizes our results and discusses future work.
Chapter 2

Background and Related Work

In this chapter, we discuss background material and prior work on topics related to this dissertation. We briefly review the required hardware architecture background for CPU/GPU systems, discuss relevant real-time scheduling problems and algorithms, and survey challenges and models in data stream processing.

2.1 Heterogeneous Systems

Multiprocessor systems can be classified as homogeneous or heterogeneous. A homogeneous system has only one type of compute unit (CU), while a heterogeneous system includes more than one type. In a heterogeneous system, each CU type has unique features and strengths, and their combination provides advantages in performance and power for various applications.

Heterogeneous multiprocessor systems can be categorized using two parameters: instruction set architectures (ISAs) and locality of their CUs (integrated or discrete). An ISA defines the instructions that a processor can execute. Thus, in a system where all the CUs have the same ISA, a program can be executed on any CU; the difference between the CUs in such a system is in their microarchitectures. In contrast, in systems with CUs that have different ISAs, code that was compiled for one CU may need to be recompiled or even redesigned to execute on another CU. The CUs in a heterogeneous system are called integrated if they are located on the same die; otherwise, they are called discrete. Integrated CUs commonly share resources and have benefits such as power efficiency, low data transfer overhead, effective resource management, and low response times. Such systems are found in many system-on-a-chip (SoC) processors, such as ARM’s big.LITTLE (same ISA) [ARM], AMD’s APU (different ISAs) [AMDa], NVIDIA’s Tegra K1 (different ISAs) [NV14b], and Intel’s Broadwell (different ISAs) [Min15]. In contrast, discrete CUs are located on different dies, commonly use separate memories, communicate via an interconnect, and provide larger thermal headroom and higher performance. Such CUs can be found in x86 CPU + Xeon Phi (same ISA) [Int12] and CPU + discrete GPU (different ISAs) systems.

This thesis considers discrete heterogeneous systems that combine latency-oriented and throughput-oriented CUs. We focus on systems with multi-core CPUs and discrete GPUs; how-
ever, many of the methods developed here remain applicable for systems with other throughput-oriented CUs, such as FPGA boards [Bit] and x86-based coprocessors [Int12].

2.1.1 Multicore CPUs

Microprocessor design shifted from single-core to multicore after the method of increasing the operating frequency was no longer effective. Further increase in frequency caused unsustainable increase in power density and heat generation, and did less to increase performance due to limited Instruction-Level Parallelism (ILP) and the increasing gap between processor and memory speeds. The efforts to improve performance in multicore CPUs take in several directions, including the following:

- Increasing the number of cores (currently, up to 18 cores in Xeon series CPUs [Int15]).
- Adding auxiliary processing units on die, such as Advanced Vector Extension (AVX) [Cor15], an integrated GPU [Min15], and AES encryption accelerators [XD10].
- Improving the power performance by optimization of energy consumption at the microarchitecture level and use of advance power management mechanisms, such as SpeedStep and Turbo Boost in Intel’s Sandy Bridge [RNA+12]. Turbo Boost utilizes the unused power budget of idle cores to increase the frequency of the active cores.

CPUs currently use DDR SDRAM memory that is spread across multiple DIMMs for higher bandwidth. Intel has recently announced a new non-volatile stacked memory technology called 3D XPoint, which might replace the technology used today.

2.1.2 Graphics Processing Units (GPUs)

Modern GPUs are fully programmable processors that offer high performance and energy efficiency. A GPU is composed of one or more parallel processing units called Streaming Multiprocessors (SMs). An SM is designed to execute up to thousands of user-defined threads concurrently and is based on the Single-Instruction Multiple-Threads architecture, where instructions are executed for a number of threads at a time. The threads are organized in fixed-size groups that are called warps, and the number of threads in a warp is defined by the GPU architecture; for example, the warp size in current NVIDIA GPUs is 32 [LNOM08] while in some AMD GPUs it is 64 [MH11]. The SM jointly creates, schedules, and executes the threads in each warp. A warp executes one instruction at a time across all of its threads for which this instruction is the next to be executed. All the threads within a warp start from the same program address, but their execution paths can diverge and converge back; for example, due to taking different branches of an if-else statement. In case of such divergence, the different paths are executed one after the other by their corresponding threads. Each SM manages a pool of warps and interleaves their execution to avoid occupying the execution elements for multiple cycles when a warp accesses off-chip memory and during other long operations. However, if
there are insufficient warps ready for execution, the utilization of the execution elements may be affected.

We describe the high-level architecture of a GPU on the example of NVIDIA’s GM204 (Maxwell) hardware architecture illustrated in Figure 2.1. The external package of the GPU processor contains an array of SMs, a work distribution engine that assigns thread blocks to SMs, four memory controllers, and a PCIe interface. Each SM is composed of four warp execution units that share a fast scratchpad memory that is called “shared memory,” a combined cache for texture and normal (by address) memory accesses, and an instruction cache. Each warp execution unit contains a warp scheduler capable of dispatching two instructions per warp every clock cycle, an instruction buffer, and several vector pipelines with multiple lanes for executing different instructions. These include 32-bit integer (INT32) and floating-point (FP32) arithmetic operations, memory load/store (LD/ST), complex FP32 functions such as logarithm and exponent (SFU), and 64-bit floating-point operations (FP64). The GPU memory is organized hierarchically, as illustrated in Figure 2.2. It includes an off-chip DRAM memory located outside the GPU, an L2 cache located in the GPU package, and a fast scratchpad memory, an L1 cache, and a register file located inside each SM. Integrated GPUs share the off-chip DRAM memory with the CPU, while discrete GPUs have separate memories that are located on the GPU board alongside the GPU; this memory is commonly referred to as the GPU memory. High-end GPUs contain high-bandwidth GDDR off-chip memories that can stream data to the SMs with bandwidth of more than 300 GB/s [NV115b], and emerging stacked memory technology is expected to increase it several times [Fol14, Joo15].

GPUs are used not only for graphics but also for general-purpose computations. In this dissertation, we consider the use of GPUs for data-stream processing computations that are not necessarily graphics-related; hence, we limit the discussion to GPUs capable of general-purpose computations. General-purpose computations on Graphics Processing Units (GPGPU) are supported by various languages, programming models, and runtime environments, including CUDA [NV15a], OpenCL [HM15], and OpenACC [Ope13]. A number of different notations are used for GPU-related terms; in this dissertation we use the CUDA notation.

To invoke a GPU, an application that runs on a CPU calls a GPU kernel. A CUDA C implementation of a simple kernel function that computes the cube (third power) of each number in an array is illustrated in Figure 2.1. The kernel generates multiple threads, each of which executes the kernel function. The threads are partitioned into equal-size work groups called thread blocks, and the collection of thread blocks is called a grid. Each thread block constitutes an independent work unit that is assigned to an SM by the GPU’s high-level work distribution engine. The threads within each thread block are created and evicted at the same time, and executed on the same SM. These threads can cooperate by sharing data through the shared memory and synchronize their execution. The thread blocks and the threads within each thread block are indexed using a 1D, 2D, or 3D indexing scheme; for example, a kernel may be called with 1024 thread blocks of 256 threads each, such that the thread blocks are indexed in 2D as \((bx, by) \in [0..31, 0..31]\) and the threads within each thread block are indexed as \((tx, ty) \in [0..15, 0..15]\). 2D and 3D indexing facilitates mapping of threads to data that is organized in.
Figure 2.1: NVIDIA Maxwell GM204 architecture \textsuperscript{[NV114a]}

Figure 2.2: GPU memory hierarchy
multidimensional arrays. Within each thread block, the threads are partitioned into warps by index according to a linear indexing scheme (row major); e.g., in a 16x16 thread block running on an NVIDIA GPU (warp size 32), the first two rows of threads \((ty \in \{0, 1\})\) constitute the first warp.

The CUDA programming model refers to the GPU as a coprocessor of the main program that runs on the CPU. The CPU program manages the GPU memory, initiates data transfers between the main memory and the GPU memory, and triggers kernel execution. In CUDA terminology, the CPU-side subsystem is called the host and the GPU-side subsystem is called the device. Consequently, the main program, which runs on the CPU and manages the GPU, is called the host program. The main memory is called the host memory and the GPU on-board memory that is accessible to the CPU via PCI Express is called device memory.

A possible management code for the kernel example is shown in Figure 2.2. The host program allocates a buffer in GPU memory and copies data to it using a call to the CUDA runtime. It then makes a kernel call by specifying the name of the kernel function, execution parameters that specify the number of thread blocks to launch and the number of threads in each thread block, and the function parameters. The kernel call is an asynchronous operation and the host program continues immediately to the cudaMemcpy command, which blocks until the kernel has finished and copies the results back to the host. The allocated buffer is released after the operation.

A GPU device driver mediates between the host program and the device. To run a kernel or perform a data transfer, the program calls a function from the GPGPU runtime library, either directly or through a higher-level interface. The runtime runs in a protected environment (user space in Linux) that does not allow it to operate the device directly. Therefore, it passes the request to the device driver, which operates with elevated privileges (kernel space in Linux), via a mechanism provided by the OS. The driver arbitrates the access to the device, usually by means of queues and ring buffers, and communicates with the device using PCI Express read and write requests and by receiving CPU interrupts from the device. The device driver’s implementation is closely related to the GPU hardware and it is usually provided by the GPU manufacturer.

Kernel calls and data transfer requests from the host program can be passed via asynchronous (non-blocking) function calls. After executing an asynchronous operation, the program can query whether the operation completed through the runtime API. Alternatively, it can provide a callback function that will be called when the operation completes. The host and each device implement the release consistency memory model, whereby the data transferred to and from a GPU during kernel execution may be inconsistent with that observed by the running kernel. This means that consistency is enforced only at the kernel boundaries.
2.1.3 CPU-GPU Systems

Architecture

CPUs and GPUs are both fully programmable CUs, but they are optimized for different types of workloads: CPUs are designed as practical general-purpose processors that control the computer’s hardware devices and handle various types of tasks effectively, while GPUs are designed...
as computation accelerators and optimized for highly-parallel tasks with low irregularity (e.g., matrix multiplication).

CPU-GPU systems are being developed in both integrated and discrete configurations. Integrated designs such as Tegra K1 [NVI14b], Intel Core i7-3720QM [She13], and AMD FX-7600P [AMDb] can achieve better power/performance ratio, but discrete designs such as Xeon E7-8870 (18-core) + Tesla K40 are less power constrained, achieve higher overall performance, and are hence preferred in high-performance systems. A common configuration for a discrete CPU-GPU system consists of a CPU, main memory, and a PCI Express (PCIe) expansion board with a GPU processor and off-chip memory. We find a discrete CPU-GPU platform more suitable for high-throughput RTSPs because such systems have high processing demands. Therefore, the discussion in this thesis is limited to discrete GPUs.

Heterogeneous Computing

Traditionally, the use of GPUs in general purpose computing was limited to accelerating specific compute-heavy portions of the program. However, recent studies have shown that techniques that use the CPU and GPU in close collaboration can yield additional performance. Such techniques are called heterogeneous computing techniques (HCTs). A recent survey of CPU-GPU heterogeneous computing techniques by Mittal and Vetter [MV15] describes the following main reasons for CPU-GPU collaboration:

- **Leveraging unique architectural strengths.** CPUs and GPUs are better suited for different types of applications. CPUs are equipped with up to tens of advanced cores and large caches, while GPUs have a large number of in-order cores, smaller caches, and work at lower frequency. HCTs can provide high performance for a large variety of applications.

- **Matching application requirements to CU features and limitations.** The relevant factors include long CPU-GPU data transfer times, branch divergence, and short latency requirements, which can cripple GPU efficiency. Thus, the characteristics of each CU need to be considered in the CPU/GPU work distribution.

- **Improving resource utilization.** When using only the CPU or the GPU, the average resource utilization is low and energy is being wasted. Intelligent management of the computational and memory resources of both CUs can improve the overall performance of the system.

- **Making use of advances in CPU design.** With an increasing number of cores and wider vector processing units, CPUs provide significant performance benefits even for workloads that map well to GPUs. Applying optimizations on both CPUs and GPUs can provide considerable speedup for the application.

The methods developed in this work are designed in line with heterogeneous computing principles and address all four of the listed benefits.
The main challenges in CPU-GPU heterogeneous computing include the following [MV15]:

- CU-architecture related challenges such as work distribution and load balancing, data transfer management, executing operations in parallel, and taking into account limitations of CUs.
- Application-specific challenges such as providing sufficient parallelism, mapping algorithms to programming model, scheduling, and handling data dependencies.
- Achieving optimization targets, such as performance, fairness, and energy savings.

This work deals with many of the mentioned architecture-related and application-specific challenges, and targets performance optimization in RTSPs.

Mittal and Vetter survey the techniques for partitioning the workload in a CPU-GPU system, which include the following:

- Static or dynamic work mapping. In static work mapping, the assignment of the tasks to the CUs is fixed and done at design time, while in dynamic work mapping, it is done at runtime.
- Tailoring the work distribution to the capabilities of each CU by considering its relative performance; selecting the best CU to execute a task by considering its performance, capabilities, and the task requirements; dynamic work assignment based on load on each CU; and aggregation of data transfers for higher efficiency.
- Pipelining of operations to execute data transfers and computations on CPUs and GPUs in parallel.

This work implements static work mapping, distributes the workload in accordance with the capabilities of each CU and the requirements of each task, and uses pipelining to achieve parallel execution.

Role of the GPU in the System

As mentioned previously, the common mode of operation in a CPU-GPU system is that the GPU is used as a coprocessor. In CUDA, the capabilities of a GPU kernel to access system resources are limited to reading and writing data from the host memory and the memories of supported peer PCI Express devices (primarily other GPUs) [NVI15a]. CUDA does not provide operating-system functions to the GPU, such as task scheduling, access to the file system, and operation of I/O devices. Providing such capabilities to the GPU would enable more effective execution of these operations, which currently have to be implemented in the host program.
Providing OS capabilities to GPUs is challenging. The interface for operating a GPU is unknown to the OS, because it is not disclosed by the vendor and varies between GPU models. Rossbach et al. [RCS+11] called for the computational resources of the GPUs to be managed by the operating system, alongside the CPUs. The authors developed PTask [RCS+11], a framework for execution of compute tasks on the GPU. PTask uses a dataflow programming model, where the programmer defines a graph of tasks and their data dependencies, and the framework manages the task execution and handles the scheduling, memory allocation, and data movement. PTask is implemented as part of the kernel on top of the runtime API, which provides GPU resource management capabilities to the OS. PTask aims to provide fairness and isolation for non-real-time compute tasks. Silberstein et al. [SFKW13] called for increased OS support for GPUs, and implemented GPUfs, an infrastructure that enables programmers to access the file system directly from a GPU kernel. The framework presents to the programmer a POSIX-like API with operations on files such as open, close, read, and write. The API requires that all the threads in a warp participate in executing each operation. This allows for effective collaborative mechanisms for the bookkeeping and data movement tasks to be implemented. The framework uses a CPU-side kernel module that handles remote procedure calls (RPCs) from the GPU. GPUfs caches the data in both the CPU and GPU and implements a weak consistency model that allows a CPU and a GPU to work on a file at the same time. Kim et al. [KHH+14] introduced GPUnet, which provides a socket-based network API to GPU kernels. Using GPUnet, kernels can communicate with CPUs and GPUs in remote machines. The framework implements efficient data transfer mechanisms, and implements direct communication between the GPU and network interface card (NIC) via peer-to-peer DMA transactions.

2.1.4 Multi-GPU Systems

Placing multiple CPUs and GPUs in the same node is a common way to improve performance, increase the size of GPU memory, and save floor space, energy, and cost. Figure 2.3 illustrates an example of a high-level architecture of a general-purpose multi-GPU compute node. The node consists of a main board with four CPU sockets. Each socket connects a CPU to a main memory module via a memory bus, other CPUs via the QuickPath Interconnect (QPI), and a GPU via PCI Express (PCIe). Here, two of the CPUs are also connected to a high-bandwidth Network Interface Card (NIC) via PCIe. The communication infrastructure that connects all the system components is called the system interconnect. Each CPU accesses its local memory directly via a memory bus, while distant CPUs and I/O Hubs (IOH) access it indirectly via the CPU interconnect, which in this example is based on QPI. Data transfers from and to GPU memory are executed by DMA controllers (a.k.a. DMA engines) over PCIe buses. The number of GPUs in such systems can be increased using PCIe switches that act as port multipliers.

2.1.5 System Interconnect

The interconnect of a CPU-GPU system is composed of several communication domains that use different technologies.
CPU Interconnect

Processor interconnect technologies such as Intel’s QuickPath Interconnect (QPI) and Hyper-Transport provide high-bandwidth connectivity among CPUs and between CPUs and I/O devices. In Intel Nehalem-based systems, the communication path between a CPU and a GPU passes through an I/O hub, which is connected to the CPU via QPI on one side and to the GPU via PCI Express on the other. In Sandy Bridge and later generations, PCI Express devices connect to a CPU directly. HyperTransport can also be used to connect the CPU directly to an external device, such as an FPGA core. Both technologies are based on point-to-point links and aggregate multiple lanes per link for higher bandwidth; for example, a HyperTransport 3.1 link provides up to 25.6 GB/s in each direction [Hyp15, Hyp10], and a QPI link in a Haswell server provides 19.2GB/s [Int14a]. For increased bandwidth, it is possible to use several links in parallel.

PCI Express

PCI Express (PCIe) is a peer-to-peer communication architecture with a hierarchical tree structure that provides a high bandwidth connection to I/O devices [PS10, BAS04]. At the top of a PCIe system is a single Root Complex, which connects the PCIe fabric to the CPU and memory subsystem though one or more PCIe ports. In CPU-GPU systems, the Root Complex is usually located either in the uncore part of a CPU or in an I/O hub that is a part of the chipset. Each port of the Root Complex can be connected to an endpoint (such as a GPU or NIC) via a PCIe bus or shared between several endpoints via a switch or a hierarchy of switches that redirect the incoming traffic to the appropriate buses. A PCIe bus is a full-duplex link for transmit-
ting packets between two components. Currently, most discrete GPUs have a 16-lane (x16) PCIe 2.0 or 3.0 connection, and have a theoretical data bandwidth of 8 GB/s and 15.75 GB/s, respectively. However, the maximum achievable throughput is lower.

PCIe endpoints can read and write CPU memory and communicate with peer endpoint devices. The communication protocol is based on packets. For example, an endpoint may transmit a memory read request to the Root Complex, which later returns a completion packet with the read data. The packets may pass through switches that would forward them to the according egress ports. PCIe provides Quality of Service (QoS) features that provide the capability to route packets from different applications with different priorities and even with deterministic latencies and bandwidth. These include assigning different traffic classes to packets in order to prioritize them during arbitration at the switches.

GPUs that have two DMA controllers support bi-directional data transfer, while GPUs that have a single DMA can transfer data only in one direction at a time. GPUs from the same PCI Express domain (tree) can exchange data directly, while most systems do not support direct data transfer between GPUs in different domains due to chipset limitations. A common solution for transferring data between GPUs that reside in different domains is to stage the data in CPU memory, using two DMA controllers. Normally, PCIe devices only exchange data with CPU memory by Remote Direct Memory Access (RDMA), but the GPUDirect RDMA (NVIDIA) [GPU10] and DirectGMA (AMD) [Mat11] technologies enable a direct path for data exchange over PCIe between a compatible GPU and a third-party device, such as a network card (e.g., Mellanox ConnectX-4 [Mel14]) or storage adapter, and provide higher bandwidth and shorter latency than the traditional method.

NVIDIA announced the development of a high-speed point-to-point proprietary communication technology called NVLink to be integrated in their Pascal series GPUs. The technology is expected to provide 5-12 times higher bandwidth than PCIe 3.0 for GPU-GPU and CPU-GPU communication, currently only with some IBM Power CPUs [Fol14]. The topology of the links is configurable and links can be ganged for higher bandwidth between a pair of devices.

2.2 Real-Time Scheduling

2.2.1 Introduction

A defining characteristic of a real-time system is operating under timing constraints. Such systems are designed by creating and analyzing a schedule that lists the times at which each operation takes place. Real-time scheduling theory has its origins in the late 1950s. At that time, the systems were scheduled by cyclic execution of a fixed schedule created for a set of immediately-available tasks [SAÅ+04]; for example, McNaughton [McN59], Schild and Fredman [SF62], and Root [Roo65] developed methods for scheduling independent single-instance tasks with deadlines on uniprocessor and multiprocessor systems. Later, in the 1970s and 1980s, there was a transition from cyclic execution to fixed-priority scheduling that built upon the seminal work of Liu and Layland [LL73] (1973). The paper considered the problem of scheduling a set
of periodically-recurring tasks on a uniprocessor, and presented two fundamental scheduling algorithms: Rate Monotonic (RM) and Earliest Deadline First (EDF). Their work considered the execution of a set of independent implicit-deadline periodic tasks. The two algorithms are preemptive and priority-driven; that is, at every instant, they execute the task with the highest priority that is ready for execution, interrupting lower-priority tasks. RM is a fixed-priority (also called static-priority) scheduling algorithm; it assigns a priority to each task once and for all. EDF is a dynamic-priority algorithm; it assigns a priority to each invocation of a periodic task that is inverse to the proximity of its deadline. Liu and Layland [LL73] proved that RM is optimal among fixed-priority algorithms and EDF is globally optimal for the considered task model, in the sense that if a task set can be scheduled to meet all deadlines by any algorithm (for RM, any fixed-priority algorithm), then their algorithm (RM or EDF) will also produce a valid schedule for this task set. Later, Dertouzos [Der74] demonstrated that EDF is optimal for any periodic and non-periodic task set on a uniprocessor.

In a multiprocessor system, the scheduling algorithm needs to determine the assignment of tasks to processors at each time instant. The task assignment techniques follow the partitioned or global scheduling approaches [RM00]. In partitioned scheduling, each task is statically assigned to a single processor, and each processor is scheduled independently. In global scheduling, jobs can migrate from one processor to another. Dhall and Liu [DL78] showed that the minimum guaranteed utilization when using RM in a global scheduling scheme is low (equivalent to the processing capacity of a single processor) due to what is called the “Dhall effect”, while in a partitioned scheme it is significantly higher. This result had a considerable impact on the multiprocessor scheduling research and biased it towards partitioned scheduling for several years [SAÅ04], until Phillips et al. [PSTW97] showed that the “Dhall effect” is due more to the high utilization of tasks than to global scheduling. Optimal partitioning of tasks between processors (the allocation problem) is akin to the Bin Packing problem, which is known to be NP-complete [GJ90]. Therefore, heuristic methods for Bin Packing such as First-Fit, Best-Fit, Next-Fit, and Worst-Fit, are commonly used to partition the tasks [ZA05]; for example, Worst-Fit iteratively assigns an unassigned task to the least occupied processor. Global scheduling techniques include Proportionate-fair (Pfair) [BCPV96] based algorithms and simpler algorithms based on EDF and RM. Pfair-based algorithms generate job preemptions and migration overhead and are complex to implement; however, their maximum utilization is higher. A survey of the common variations of EDF and RM for multiprocessors can be found in [ZA05].

Graham et al. [GLLRK79] (1979) published an important survey and classification of scheduling problems. The classification has since been expanded and it is used to organize and categorize scheduling problems to this day [CPW99, Bru07, BK15]. We describe this classification in more detail in Chapter 7, where we also discuss the need for a new computing platform model to match the characteristics of modern processors and propose an extension to the classification parameters.

The conventional approach for scheduling real-time systems uses an abstract model where tasks that represent system processes are composed of work elements called jobs that are as-
signed to processors for periods of time and need to be completed before their associated deadlines. The model may include additional constraints on the jobs’ execution, including job ordering dependencies, processor limitations, and others. Three main types of quality-of-service requirement models are considered in the literature: hard real-time, soft real-time, and mixed-criticality real-time. In hard real-time scheduling problems, the objective is to construct a schedule that strictly meets all the execution requirements: all the jobs must be executed without violating any of the timing or other constraints. An offline schedulability test checks whether the schedule generated by the scheduling algorithm for the given task set is valid (meets all the requirements). Since the system may be required to react to events whose exact times are not known until runtime, the schedulability test approves the system only if the scheduling algorithm is guaranteed to generate a valid schedule for every possible completion of the missing information. In soft real-time scheduling problems, it is acceptable for some jobs to miss deadlines and the objective is to satisfy some optimization criteria, such as a minimum number of deadline misses or minimum cumulative tardiness (deviation from deadline). In mixed-criticality scheduling problems, the execution requirements must be satisfied for subsets of the task set at different levels of assurance [BD15, BBD+12]; this is currently an active research topic.

2.2.2 Task Model

A task is a collection of jobs that jointly implement a system function. A real-time job \( J = (a, e, d) \) is characterized by the following parameters:

- \( a \) – release time (also called arrival time): the time instant at which the job becomes available for execution.

- \( e \) – execution requirement: a scalar measure of the amount of required processing resources, such as computation cycles, execution time on some processor, or communication bandwidth. By convention, in uniprocessor and homogeneous systems the execution requirement is normalized to the computing capacity of the processor, and in heterogeneous systems it is normalized to the capacity of the first processor; i.e., a work unit takes one unit of time to execute (on the first processor). In many hard real-time systems, the execution requirement denotes the worst-case execution time (WCET).

- \( d \) – deadline: the time instant by which the job’s execution needs to be completed. A situation where a job’s execution was not completed by its deadline is called a deadline miss. The deadline in this characterization is given as an absolute time point.

Jobs can be preemptive or non-preemptive. The execution of a preemptive job can be interrupted at any time and resumed later on the processor where it was executed or on another processor, without any execution overhead. In contrast, a non-preemptive job executes without interruption on the processor it was assigned to until completion.

Many real-time systems handle recurring tasks that generate jobs in a predictable manner. In particular, RTSPSs process streams of data that are characterized by recurring arrival of data
packets from regular data sources such as video cameras. The execution requirements in such systems are commonly modeled as periodic or sporadic task systems.

A periodic task system \( \tau = \{\tau_1, ..., \tau_n\} \) consists of a finite set of periodic tasks. Each periodic task generates a job at regular periodic intervals, and the deadline of each job, relative to its release time, is the same for all the jobs. A periodic task \( \tau_i = (e_i, d_i, p_i) \) generates an infinite series of jobs \( J_i^0, J_i^1, ... \) with execution requirement \( e_i \) that are released at time intervals (periods) of \( p_i \). Let the release time of \( J_i^0 \) be \( a_i^0 \). Then, for every integer \( k > 0 \), job \( k \) has release time \( a_i^k = (a_i^0 + k \cdot p_i) \). The parameter \( d_i \) is the task’s deadline and specifies the deadlines of all the jobs relative to their release time; a job deadline specified relative to its release time is called a relative deadline. Thus, for any \( k \geq 0 \), the absolute deadline of job \( J_i^k \) is \( d_i^k = (a_i^k + d_i) \).

There are precedence constraints between the jobs in a periodic task: a job can start executing only after the preceding job was completed, even if its release time is earlier. If the release time of job 0 is known, the task is said to be concrete.

A sporadic task system \( \tau = \{\tau_1, ..., \tau_n\} \) consists of a finite set of sporadic tasks. Sporadic tasks are similar to periodic tasks, except \( p_i \) denotes the minimum release separation time between consecutive jobs; i.e., it is known that \( a_i^{k+1} \geq (a_i^k + p_i) \) but the exact release times are not specified. The deadline of a job in a sporadic task is defined as \( d_i \) time units from its release time. By definition, any periodic task is also a sporadic task.

A (periodic or sporadic) task is said to have a constrained, implicit, or arbitrary deadline if \( d_i = p_i \), \( d_i = p_i \), or \( d_i > 0 \), respectively.

The following properties of a task are extensively used in schedulability tests:

**Utilization** The utilization \( u_i \) of a task \( \tau_i \) is the ratio \( e_i/p_i \). Utilization is the fraction of a processor required to serve a given task over the task’s period and over a sufficiently long time interval.

**Density** The density \( \delta_i \) of a task \( \tau_i \) is the ratio \( (e_i/\min(d_i, p_i)) \). Density is the utilization if \( d_i \geq p_i \), or the fraction of a processor required to serve a given task over the period between a job’s arrival and its deadline if \( d_i < p_i \).

### 2.2.3 Computing Platform Model

Computing platforms can be classified as uniprocessor or multiprocessor. A uniprocessor system contains a single processor (processing unit) for executing the tasks. Multiprocessor systems contain multiple processors and come in different configurations such as homogeneous and heterogeneous. The conventional processor model makes the following assumptions:

(a) at every time instant, each processor executes at most one job;

(b) at every time instant, each job may not be processed on more than one processor.

Each processor acts as an independent job execution unit with some computing capacity by which the job execution time is determined.
Multiprocessor systems are categorized by the computing capacities of their processors as follows:

**Identical** All the processors have the same computing capacity. This model characterizes homogeneous systems.

**Uniform** The computing capacity of each processor is characterized by its speed. The execution time of a job is inverse to the speed of the processor that executes it. This model is commonly used to describe processors that use the same ISA.

**Unrelated** The computing capacity of each processor is characterized by its speed and job type. The execution time of a job on one processor is unrelated to its execution time on another processor. This model is commonly used to describe heterogeneous systems with processors that use different ISAs.

### 2.2.4 Other Scheduling Problems

We briefly describe two classes of scheduling problems that use other task and computing platform models that are related to this work.

**Time-Dependent Scheduling**

In time-dependent scheduling, the processing time of a job is variable and depends on the starting time of its execution. This occurs when a delay in the execution of a job causes an increase or decrease in its execution time. The book “Time-Dependent Scheduling” by Gawiejnowicz [Gaw08] provides a detailed description of time-dependent scheduling models, problems, and algorithms.

In time-dependent scheduling, the processing time of a job $J_i$ is given by a function of its starting time. In the most general form, which corresponds to the unrelated processors model, the processing time of $J_i$ on processor $j$ is given by

$$p_{i,j}(t) = g_{i,j}(t),$$

(2.1) where $g_{i,j}(t)$ is an arbitrary non-negative function of the job starting time $t$, where $t \geq 0$. It is assumed that jobs are non-preemptive. The above representation form is rarely used because it does not give any information about changes in the processing times. The following description of job processing times is more commonly used:

$$p_{i,j}(t) = a_{i,j} + f_{i,j}(t),$$

(2.2) where $a_{i,j} \geq 0$ denotes the constant part of the execution time. Two important classes of time-dependent scheduling problems are scheduling jobs with *deteriorating processing times*, where the functions $f_{i,j}(t)$ are non-decreasing, and scheduling jobs with *shortening processing times*, where $f_{i,j}(t)$ are non-increasing. All the parameters of the problems are assumed to be known.
to the scheduler. Applications of time-dependent scheduling include repayment of multiple loans \cite{GKD87} and recognizing aerial threats using radar \cite{HLW93}.

**Batch Scheduling**

In some scheduling systems, a machine can jointly process sets of jobs, which are called batches. The completion time of all the jobs in a batch is defined as the completion time of the entire batch \cite{Bru07}. Execution of a batch involves a fixed setup time, and the problem is to group the jobs into batches and to schedule these batches. Two types of batching problems are considered in the literature: p-batching problems and s-batching problems. The problems differ in the jobs’ execution model. In p-batching problems, the jobs are executed in parallel (without interference), while in p-batching problems the jobs are executed serially.

### 2.3 Real-Time Scheduling in CPU-GPU Systems

A number of recent studies dealt with the problem of using GPUs in real-time systems. We first survey several notable works in the area, and then discuss the challenges and how they were addressed in previous studies.

#### 2.3.1 Overview

In his dissertation \cite{Ell15}, Glenn Elliott made a fundamental contribution to understanding the challenges in using GPUs in real-time systems. Elliott carefully analyzed the costs involved in operating the GPU, including CPU interrupt handling, DMA controller configuration, and the effect of inter-processor communication on CPU performance. Elliott et al. developed GPUSync \cite{EWA13}, a framework for scheduling real-time sporadic tasks on clusters of CPUs and of GPUs. The framework is implemented as an extension of the LITMUS$^{RT}$ \cite{Bra11} real-time Linux kernel. GPUSync takes a synchronization-based approach to the scheduling problem; it arbitrates access to the GPUs’ processing units and the DMA controllers using $k$-exclusion locks. Such a lock gives access to the resources to up to $k$ jobs at the same time. To get access to a resource, a job must first obtain one of the $k$ tokens, which are allocated by a priority-based mechanism. The resource is further protected by a basic lock that ensures that at most one token-holding job is using the resource at any instant. GPU-using tasks can migrate between GPUs for load balancing.

Kato et al. presented TimeGraph \cite{KLRI11}, RGEM \cite{KL11}, and Gdev \cite{KMMB12}. TimeGraph is an early work on using GPUs for real-time tasks and is aimed at real-time graphics applications. This GPU command scheduler was implemented in an open-source device driver and included two task scheduling priority policies: Predictable Response Time (PRT) and High Throughput (HT). In PRT a new job with the highest priority has to wait only for the completion of the job currently in execution, while in HT it may have to wait for several lower-priority jobs that were issued earlier by the same task. PRT provides greater predictability, while HT provides greater efficiency. TimeGraph is limited in that it does not support concurrent data transfer.
and kernel execution and multiple GPUs, and is intended to improve the responsiveness of interactive applications such as games and video players. RGEM is a run-time execution model for soft real-time GPGPU applications that is implemented entirely in user space. It supports tasks with fixed priorities. RGEM supports simultaneous data transfer and kernel execution. To reduce blocking times that result from large data transfers, it splits the data into chunks and schedules them according to given task priorities. Gdev provides advanced non-real-time GPU resource management features to the OS. It uses alternative implementations of the GPU environment (runtime and device driver), and hence is transparent to the user. Gdev offers a priority-aware scheduler and advanced GPU memory management services such as chunking of large data transfers, elimination of redundant data copies on the host, faster direct I/O operations for small data transfers, and page swapping to enable allocation of more memory than the physical memory on the GPU.

GPUs and their software environment are designed to perform large computational tasks at high throughput. To use them in a real-time system, it is necessary to meet the challenges of control over GPU resources, computation of worst-case execution time, and resource scheduling. Next, we describe these challenges in more detail and review the methods used by others to address them.

2.3.2 Limited Control over GPU Resources

Kernel calls and data transfer requests

Kernel calls and data transfer requests from the host program are executed by the GPU and DMA controllers, which are located on the GPU board. Thus, the time it takes to pass the request to the device and configure it for execution is considerable and limits its responsiveness.

DMA configuration is one of the most time-consuming setup tasks. Aumiller et al. [ABKR12], Fujii et al. [Fan+13], Kato et al. [KAB13], and Rath et al. [RKL+14] showed that direct PCIe reads and writes can be an effective alternative to DMA operations for low-latency data transfers; using such methods Rath et al. [RKL+14] achieved response times of below 8 µs for processing sensor data on a GPU.

Resource management and device arbitration for concurrent requests are performed by the device driver. Unfortunately, GPU device drivers are supplied by the vendors in closed-source form and are not easily modifiable. The host program has very limited tools for controlling the execution of kernels and data transfer operations:

1. It can define precedence dependencies between the operations using work queues (streams in CUDA, command queues in OpenCL); operations in different queues can be executed concurrently, which enables, for example, overlapping of data transfer and kernel execution.

2. CUDA supports relative priorities between the queues, so that as thread blocks in low-priority queues finish, waiting blocks in higher-priority queues are scheduled in their
place [NV115a]; however, the host program cannot explicitly define dependencies between the thread blocks.

3. It can schedule the invocation of kernel calls and data transfer requests; however, their execution is scheduled by the device driver and is not within the program’s control. Nor can kernel calls and data transfers cannot be withdrawn or aborted once they have been enqueued, which severely limits the possibility of using a dynamic scheduling approach.

To gain control over the resources, Kato et al. used alternative implementations of the GPU environment (runtime and/or device driver) in their GPU resource management frameworks TimeGraph [KLR111] (for graphics) and Gdev [KM112] (for general-purpose computations). This approach has a number of drawbacks. First, since GPU vendors do not publish a specification for the protocol between the GPU and the device driver, the protocol used by the open-source drivers is reverse-engineered from the official drivers, so the open-source drivers might not support all the GPU features. Second, official GPU drivers are frequently updated to support new models and features, so the open-source drivers require continuous maintenance. Elliott et al. used a different approach in GPUSync [Ell15, EWA13]. This framework manages the GPU resources by itself and uses the runtime and closed-source driver merely as intermediate layers to the GPU. Specifically, it runs a kernel-space scheduler that intercepts calls to the GPGPU runtime library and passes them to the runtime only when it is time for them to execute. This approach may be more practical as long as the GPU interfaces remain obscure.

**Preemption**

A number of studies focused on kernel preemption. Calhoun and Jiang [CJ12] emulated kernel preemption by saving the kernel execution state (shared memory and registers) and exiting the kernel voluntarily every time quantum; such techniques have high overhead and/or result in long time between scheduling decisions. Basaran and Kang [BK12] used a kernel slicing technique for kernel preemption, where the grid is split into smaller sub-grids that are launched as separate kernels, and correct block indices are available through special variables. Zhong and He [ZH14] made the index corrections transparent to the programmer through modification of the low-level code (PTX or SASS). These methods incur lower preemption overhead but may also lead to lower GPU utilization and require code modification to set preemption points. Tanasic et al. [TGC14] proposed architectural changes to NVIDIA GPUs to enable kernel preemption.

Data transfers to and from GPU memory are also non-preemptive and can take up to hundreds of milliseconds to complete. In their real-time GPU scheduler RGEM [KL11], Kato et al. proposed to break the DMA operations into smaller chunks to reduce priority inversion, an approach later adopted by others [ABKR12, BK12, EWA13]. However, when DMA operations are used, the data transfer time cannot be smaller than the DMA configuration time, so decreasing the chunk size beyond a certain point has very little effect on latency. Kato et al. chose 1MB chunks for RGEM based on benchmarks. Using direct PCIe reads and writes it is possible to perform short data transfers faster [FAN13].
Thread scheduling

The GPU’s internal thread scheduling mechanisms are not accessible to the host program nor fully disclosed by the GPU vendors. Thread blocks can be assigned to the SMs in any order, and warps that are ready for execution can be interleaved arbitrarily. The host program has limited influence on the execution of the threads:

1. It determines the granularity of an SM task by specifying the thread block dimensions.

2. It can restrict the number of thread blocks resident on an SM by artificially inflating the amount of resources each thread block requires. For example, it can cause SMs to execute thread blocks one at a time. This technique prevents contention between thread blocks for SM resources and facilitates the timing analysis. However, it might result in a significant performance loss due to low utilization of the SM and memory bandwidth.

2.3.3 Estimation of Execution Time

Accurate estimation of worst-case execution time (WCET) of GPU operations is very difficult due to (1) fixed and opaque execution mechanisms; (2) asynchronous CPU-GPU communication protocols; and (3) resource sharing and concurrent execution.

Due to the lack of transparency in the operation of the GPGPU runtime, device driver, GPU work distribution engine, and warp scheduler, formal analysis of the WCET impossible is not possible and research has focused on empiric performance models and statistical analysis.

Computing an upper bound on the invocation latency is very challenging because (1) the resource management mechanisms in the device driver are not revealed, and (2) the driver communicates with the GPUs and DMA controllers using asynchronous message-based communication protocols that use PCIe read/write operations and CPU interrupts. These difficulties can be ameliorated using the approaches proposed by Kato et al. [KMMB12] and Elliott et al. [Ell15]: using an alternative open-source GPU environment and managing the resources in the host program instead of the device driver.

Kernel execution time is difficult to predict due to the multitude of resources and massive parallelism. By design, GPUs create dozens of warps that compete for the compute elements and memory buses in order to hide latency and achieve high utilization; hence, accurate modeling and analysis of the execution would be extremely difficult. Traditional worst-case execution time (WCET) analysis for CPUs is inapplicable in this case because it is assumes a small number of threads. Berezovskyi et al. [BSBT14] recently used measurements, statistical analysis, and extreme-value theory to provide a probabilistic WCET. Betts and Donaldson [BD13] presented two techniques for estimating the WCET of a kernel. The first technique estimates the earliest time the last warp is sent for execution and the execution time of this warp, using measurements. The second technique uses a model of the GPU scheduler, trace data, and timestamps collected at certain places in the code to estimate the effects of resource contention and compute the WCET. Huangfu and Zhang [HZ15] proposed a method that selectively bypasses GPU caches to improve the WCET analysis. Membarth et al. [MKH12] analyzed the WCET using
a hardware model of the GPU and a control-flow graph of the kernel that was generated from PTX code using the Ocelot framework [KDY10].

### 2.3.4 Resource Scheduling

Resource scheduling in a real-time CPU-GPU system includes scheduling computations on the CPUs and GPUs and scheduling data transfers between the CUs in a way that meets the processing requirements and the user-provided latency constraints. Existing works on the problems of real-time scheduling in CPU-GPU systems include [KLRI11, KL11, LA14, EA12, EWA13, Ell15, HBK+14, ZHH15].

We have already discussed some aspects of resource scheduling in GPUSync, TimeGraph, and RGEM; here we details them in further detail. GPUSync [EWA13] considers the problem of scheduling sporadic tasks in a CPU-GPU system as a multiprocessor scheduling problem and uses a synchronization based approach where the schedulers are implemented in $k$-exclusion locks. The framework supports any job-level fixed priority scheduling algorithm. RGEM [KL11] uses priority queues to arbitrate access to the GPU and DMA controllers.

Han et al. [HBK+14] presented a real-time framework for multi-GPU systems called GPU-SPARC. The system considers a constrained-deadline sporadic task model where each real-time task is composed of a chain of CPU, GPU, or data transfer operations, and uses the Deadline Monotonic (DM) non-preemptive fixed-priority scheduling algorithm. The framework applies kernel slicing to split the kernel between multiple GPUs in order to reduce its execution time. This technique has larger overhead than the single-GPU kernel splitting because additional data transfers are required to make the data available to the kernels on all the GPUs.

Zhang et al. [ZHH15] recently presented a design of a real-time system for processing multiple data streams. Their design builds upon our work and extends it by accounting for network processing overhead and adding an admission control mechanism. The system processes batches of streams together in a processing pipeline that includes a data transfer of the input to the GPU memory, execution of a GPU kernel, and a data transfer of the output back to the main memory.

Chang et al. [CCKF13] studied the problem of scheduling implicit-deadline sporadic tasks on a platform that consists of multiple islands of homogeneous cores, where the aim is to minimize the number of required islands. The cores in each island share a fast memory of limited size and all the cores share a large global memory. The authors present scheduling algorithms that provide asymptotic approximation bounds on the number of required islands. In platforms with multiple CPUs and GPUs, this approach can be applied to the CPUs. The organization of memories and cores in GPUs is generally compatible with their platform model, but their use of the traditional one-task-per-core execution model is not compatible with the GPU’s thread-level parallelism.

Kim et al. [KANR13, KRK13] proposed an adaptive GPU resource management framework for sporadic real-time tasks. Instead of running each GPU task on all the SMs, the framework allocates a subset of the GPU’s SMs for each task.
Kuo and Hai [KH10] presented an EDF-based algorithm to schedule real-time tasks in a heterogeneous system. Their work is based on a system with a CPU and an on-chip DSP. This algorithm is not compatible with our system model because of the high latency of CPU to GPU communication.

In a recent dissertation, Raravi [Rar14] studies the problem of scheduling implicit-deadline sporadic tasks on heterogeneous systems with several types of processors. This model is similar to the unrelated processors model, except that each processor has one of a limited set of types, which defines its compute capabilities. A system with several CPU and GPUs can be considered as a 2-type system. For such systems, the study considers task assignment problems that have different assumptions on task migration and use of shared resources. All these problems assume that no more than one task is executed on the same processor at the same time.
Chapter 3

System Model

Our system applies a data-processing application to a set of real-time data streams. It aims to achieve the maximum throughput while satisfying the real-time constraint of each stream.

3.1 Hardware Model

We consider a heterogeneous system composed of an interconnected set of discrete compute units (CUs), each with a local memory module, such as a CPU-GPU or multi-GPU system.

The system contains one or more multi-core CPUs, one or more GPUs, and optional high-throughput network cards, each with a local memory module that is accessible to the other devices via the interconnect. One of the CPU cores is dedicated to controlling the job execution; it schedules the kernel executions and data transfers, and sends them for execution. Each CPU core, except those dedicated to a specific task, and each GPU device, is considered as a separate processor that executes the compute kernels assigned to it, one at a time.

A GPU executes jobs in batch mode; it receives a batch of jobs (data and instructions) that was previously copied to its local memory, and executes a kernel that collectively processes the data, making use of its multiple execution units. If needed, the results are then copied to a destination buffer on a different device. The batches are executed one at a time in a non-preemptive manner. Jobs in the batch are scheduled on the GPU’s SIMD units by a hardware scheduler. Thus, we have no control over or information about the order in which the jobs will be executed; we assume that the scheduler strives to maximize GPU throughput.

3.2 Real-Time Data Stream

We consider an application model in which the system needs to process a set of hard real-time data streams \( S = \{s_i\}_{i=1}^N \) that generate data for processing. Each data stream \( s_i = (q_i, t_i, op_i, l_i) \) is a source of data packets of size \( q_i \) that arrive to the system sporadically, at least \( t_i \) time units apart. Each packet needs to be processed using an operator \( op_i \) within a hard deadline of \( l_i \) time units (latency bound) from its arrival. Any CU is capable of executing every operator. For each stream, the operator reads and updates a state in the memory of the CU where it is executed.
Packets that belong to the same stream must be processed in the order of their arrival to preserve the correctness of the state, but can be processed in parallel with packets from other streams. The model is illustrated in Figure 3.1.

At the beginning, the system receives a characterization of the stream set that includes the parameters \((q_i, t_i, \text{op}_i, l_i)\) for each stream and begins processing it only after verifying that it can satisfy all the time constraints. The stream set is either constant or changes infrequently, and can be updated only if the following conditions are met:

- the system has sufficient resources to compute a new work plan without violating any old time constraints;
- the system verified that it can satisfy all the time constraints for the new stream set;
- the stream states can be migrated to match the new work plan without violating old time constraints, until the change comes into effect.

Upon arrival to the system, the packets are accumulated in a buffer until being sent for processing. The packet accumulation buffer of each stream resides on one of the memory modules in the system, such as in the main memory, in a GPU’s on-board memory, or on the network card. For simplicity, it is assumed that all such buffers are stored in the main memory, unless otherwise noted.

### 3.3 Packet Processing Model

The processing model of the packets is illustrated in Figure 3.2. For every stream \(s_i\), the system periodically inspects the buffer at intervals of \(T_i\) time units, and if any packets are present it
removes all the accumulated packets from the buffer and sends them as a job $J_k^i = (a_k^i, e_k^i, d_k^i)$ to the processing queue of a CU. An execution engine handles all aspects of execution of jobs from the queues on the respective CUs; it handles the related data transfers and state migrations and initiates the computation. The deadline of the job is $(l_i - T_i)$ time units from its creation time because the time between packet arrival and buffer inspection is included in the processing latency; i.e., $d_k^i = (a_k^i + l_i - T_i)$. For simplicity, we assume that processing requirement of packets from $s_i$ are proportional to their size with factor $\lambda_i$; hence, the size of a job that processes $n^i_j$ packets of size $q_i$ is $e^i_j = (\lambda_i \cdot n^i_j \cdot q_i)$. Jobs belonging to the same stream need to be processed sequentially, while jobs belonging to different streams can be processed in parallel.
Chapter 4

Stream Processing in a CPU-GPU System

This chapter studies the problem of processing a fixed set of real-time data streams on a system with a single multicore CPU and a single GPU. We define the real-time stream processing problem as follows:

**Problem 1.** Given a set of input streams \( S = \{ s_i : (q_i, t_i, op_i, l_i) \} \) and a system with a CPU and a GPU (the CUs), set the job creation period for each stream \( T_i \), find an assignment of each stream to a CU, and find a feasible schedule for each CU, if such a schedule exists, or return *failure* otherwise.

The scheduling problem in this case can be characterized as scheduling sporadic tasks with arbitrary deadlines. The existing CPU-only scheduling approaches \([ABB08, LL73, BCPV96]\) cannot be applied in our setup because they schedule jobs one or a few at a time. In contrast, GPUs require batches of thousands of jobs to be packed together, raising the question of how to select those to be executed together so that all of them comply with their respective deadlines. We use the term *stream* interchangeably when referring to a data stream or the sequence of jobs that corresponds to it.

Multiprocessor systems are known to be tightly coupled and use shared memory for interprocess communication. The time of such communication is low compared to task execution times. For this reason, a multiprocessor may use a global (centralized) scheduler. In contrast, the memory in our system is divided to several distant physical units – the main memory, shared by the CPU cores, and DRAM memory on the GPU board, shared by the SMs of that GPU. Global scheduling is not practical in our system for two reasons:

1. The standard GPU model is bulk synchronous; GPUs are not optimized for dynamic update of data in GPU memory while a kernel is running. Therefore, a running execution on a GPU must be completed before a new batch can be executed. Hence, the new batch for the GPU should be statically created in advance on a CPU, under the constraint of timely completion of all jobs in the batch. Even if an efficient mechanism for sending new jobs to a running kernel existed, it would be very hard to guarantee that their completion
Figure 4.1: Batches are processed in a four-stage pipeline

time will not exceed their deadlines due to scheduling dependencies between the new jobs, running jobs, and jobs added in the future. Nor can other new mechanisms for using GPUs, such as running multiple kernels simultaneously, be currently applied due to lack of isolation between tasks at the kernel, thread-block, and warp levels.

2. The stream state is carried from one job in a stream to the next job in the same stream. The overhead of moving jobs between CUs might become a bottleneck due to slow communication between the CPU and the GPU. Furthermore, since a steady state is assumed, migration can be avoided with proper batch creation.

These considerations led us to choose the static partitioning method, where multiple streams are statically assigned to a particular CU.

### 4.1 GPU Batch Scheduling

The streams on the GPU are processed in batches-of-jobs, or batches. A batch is a collection of jobs that are sent together for execution on the GPU in the same kernel instance. Every batch goes through a four-stage synchronous pipeline: data aggregation in main memory, data transfer to local GPU memory, kernel execution, and transfer of results to main memory. The duration of a pipeline period, which is defined by the system, is denoted by $T$. The pipeline stages are illustrated in Figure 4.1.

Let $S$ be a set of streams assigned to be processed on the GPU. The incoming packets from streams in $S$ are sent for processing on the GPU as a batch at the end of every data aggregation period, every $T$ time units. Thus, the buffer collection periods of all the streams in $S$ are equal to $T$, i.e., $\forall s_i \in S, T_i = T$. The processing latency for the packets is at most $4T$ because there are four pipeline stages, so in order to guarantee that no packet misses its deadline, we require that $4T \leq l_i$ for every stream in $S$, i.e., $T \leq \frac{1}{4} \min_i l_i$. $T$ must be at least as long as any of the four pipeline stages. Given a candidate value of $T$, we compute the expected batch of jobs and rely on a GPU performance model to calculate the length of each pipeline stage. From a scheduling perspective, the batch jobs are the only jobs scheduled on the GPU. To differentiate between regular jobs and batch jobs we denote the sizes of the jobs that compose a batch job by $W = (W_i)_{i=1}^N$.

Our design for efficient job batching was guided by the following principles:
1. Batch as many jobs as are ready to be executed. Batches with many independent jobs have higher parallelism and provide more opportunities for throughput optimization by the hardware scheduler.

2. For every job, aggregate as much data as possible (this means aggregate as long as possible). This is because batch invocation on a GPU incurs some overhead, which is better amortized when the jobs are large. Moreover, transferring larger bulks of data between a GPU and a CPU improves the transfer throughput.

Distribution of job sizes in a batch affects the load balancing on the GPU. We batch jobs in a way that minimizes their overall execution time, while guaranteeing that all batches complete on time.

### 4.1.1 Schedulability

We test GPU schedulability for \( S = \{s_i\}_{i=1}^{N} \) in 4 steps:

1. Set pipeline and packet-aggregation period \( T = \frac{1}{2} \min_{s_i \in S} s_i \).
2. Calculate job sizes in a batch \( W = (W_i)_{i=1}^{N} \) where \( W_i = \lambda_i \left\lceil \frac{T}{s_i} \right\rceil q_i \).
3. Estimate the data transfer time (input and results) \( T_D \) and kernel execution time \( T_K \).
4. If the system supports bi-directional data, \( S \) is schedulable if \( \max(T_D, T_K) \leq T \); otherwise it is considered as not schedulable.

### 4.1.2 GPU Empirical Performance Model

The GPU performance model is used by our schedulability testing function to estimate data transfer time and kernel execution time. It is not a general model: it was built specifically for our batch execution algorithm. In the model we assume that the jobs in a given batch are distributed among the SMs so that each SM is assigned the same number of jobs, regardless of job size. Such a distribution is equivalent to a random one that ignores load balancing considerations. At runtime, the thread-block scheduler on the GPU distributes the streams between the SMs dynamically, so the estimated execution time gives us an upper bound. The problem is therefore reduced to estimating the runtime of the jobs on a single SM, and taking the longest one among the SMs as the estimate of the runtime of the whole batch on a GPU.

We now consider execution of jobs on an SM. For convenience, we express the performance of an SM as utilization, namely, the fraction of the maximum throughput achievable by a single SM. The maximum throughput for a given GPU stream processing kernel can easily be obtained by measuring the throughput of that kernel on tens of thousands of identical jobs.

We use the term utilization function to describe the dependency of the SM utilization on the number of jobs invoked on that SM. The utilization function can be calculated by means of a lookup table generated by benchmarking a GPU kernel for different numbers of jobs, or using an interpolation function of these measurements. Figure 4.2 demonstrates the utilization
function for the AES-CBC kernel with four threads per data stream. The saw-like form of this graph is due to the execution of multiple jobs together in a single warp. Thus, in Figure 4.2, \( \lfloor x \rfloor \) is the rounding of the number of jobs \( x \) to the closest multiple of the number of jobs per warp that is higher than or equal to \( x \).

When we run jobs of different sizes, the utilization function becomes more complex. The number of running jobs on an SM dynamically decreases during execution, and therefore the utilization changes as well. Consider the example in Figure 4.3. It shows 6 jobs of different sizes invoked at the same time. Utilization at a certain time depends on the number of running jobs. In this example, the execution can be divided into 3 utilization periods. During the execution of job1, six jobs are running, so the utilization is \( U(6) = 12\% \). The time of this period is \( t_1 = 6 \cdot W_1 \cdot (12\% \cdot OPS_{SM})^{-1} \), where the size of job1 \( (W_1) \) is measured in number of operations, and \( OPS_{SM} \) is the maximum throughput of a single SM in operation/sec. Four jobs are left in the second time period. The length of this period is \( t_2 = 4 \cdot (W_3 - W_1) \cdot (8\% \cdot OPS_{SM})^{-1} \). The length of period \( t_3 \) is calculated similarly. SM processing time is the sum of all these periods.

We estimate the kernel execution time of a batch of jobs as follows. Given a batch of jobs \( W = (W_i)_{i=1}^N \), where \( W_i = \lambda_i \left\lfloor \frac{q_i}{r_i} \right\rfloor q_i \), the jobs are partitioned into equally-sized per-
SM subsets $\{ \{ w_i \}^n_{i=1} \}_{c=1}^{\#SMs}$. Let $\{ w_i \}^n_{i=1}$ be one of the subsets. We assume w.l.o.g. that $w_1 \geq w_2 \geq \ldots \geq w_n$, and define $w_{n+1} := 0$. The estimated execution time of this subset is calculated using the following formula:

$$T_K^{SM}(\{ w_i \}^n_{i=1}) = \sum_{i=1}^{n} (w_i - w_{i+1}) t(i) = \sum_{i=1}^{n} \frac{(w_i - w_{i+1}) \cdot i}{U(i) H}.$$ (4.1)

Here, $t(i)$ is the time it takes to perform one computation per job across $i$ jobs concurrently on an SM, $U(i)$ is the SM utilization function on the GPU, and $H$ is the maximum throughput of an SM in computations per millisecond. The kernel execution time is the maximum of the per-SM execution times:

$$T_K = \max_{1 \leq c \leq \#SMs} T_K^{SM}(\{ w_i \}^n_{i=1}).$$ (4.2)

To estimate the data transfer time, $T_D$, the performance model uses a lookup table that contains a column of data sizes and two columns for achievable transfer rates (one for each direction). Figure 4.4 shows transfer rate and latency benchmark results for the configuration described later in Section 4.4.1. The effective bandwidth increases with chunk size, until saturation. It is interesting to see that the throughput is not symmetric for transfers to and from the GPU. For chunks of size 4 KiB or smaller, the transfer time ranges around 20\(\mu\)s, and full bandwidth is achieved for chunks 8 MiB and larger. The GPU's single DMA engine supports only uni-directional data transfer, so the bi-directional throughput equals the average throughput of the two directions.

In our system, incoming data for GPU-assigned streams is automatically stored contiguously in memory. This is achieved by allocating contiguous blocks of memory (256 B) to store incoming data from these streams. Consequently, the data for a whole batch can be transferred to the GPU as a data chunk in a single operation. We evaluate the precision of our model in Section 4.4.4.

**Implications on task scheduling** Note that the existing batch scheduling problems s-batching and p-batching are not suitable in this case, because the execution time of a batch on the GPU is neither equal to the maximum nor to the sum of the execution times of the jobs.

### 4.2 CPU Scheduling

#### 4.2.1 Scheduling Algorithms

The CPU continuously monitors the buffer, and collects the packets immediately after their arrival; hence, we define $T_i = 0$ for every stream $s_i$ assigned to the CPU. As soon as a packet arrives from a stream that was assigned to the CPU, the CPU appends a new job to the corresponding processing queue.
Figure 4.4: Data transfer rates in a CPU-GPU system with Intel P45 chipset
From the point of view of the execution engine, a stream $s_i = (q_i, t_i, l_i, o_i)$ that is processed on the CPU is a sporadic task $\tau_i = (e_i, d_i, p_i)$ with the following characteristics:

- job size $e_i = \lambda_i p_i$ (normalized to single-core computation capacity);
- relative deadline $d_i = l_i$;
- minimum release separation time $p_i = t_i$.

In our model, the latency bound is not limited by the period, so the tasks have arbitrary deadlines. We describe the set of streams assigned to the CPU as a sporadic task system and use an existing task scheduling algorithm to schedule the packet processing.

A CPU schedulability testing function is used in our system to validate that streams it assigned to the CPU will be scheduled for processing in a valid schedule. That is, all jobs will be executed and no job will miss its deadline. The schedulability-testing function must match the CPU scheduling algorithm. We use [BB09] as theoretical background for the analysis of schedulability of streams on multiprocessors. We now describe two common EDF-based scheduling algorithms for multiprocessors, and their schedulability tests.

**Earliest-Deadline-First First-Fit (EDF-FF)**

The Earliest-Deadline-First First-Fit (EDF-FF) algorithm, shown in Listing 4.1, is an EDF-based partitioned scheduling algorithm for multiprocessors. The algorithm partitions tasks to processors such that the schedule produced on each processor using EDF is feasible. In EDF, each arrived job that has not been executed to completion has a priority that is inverse to its absolute deadline. That is, the job with the closest deadline has the highest priority and is executed. Job preemption is allowed.

First-Fit is a heuristic for the Bin-Packing problem, which is NP-Hard. EDF-FF attempts to partition the tasks to processors using a minimal number of processors, such that EDF-schedulability on each processor is guaranteed. If the number of necessary processors $m$ returned by EDF-FF is smaller than or equal to the number of available processors $m'$ ($m \leq m'$), then the task system is schedulable using this method. Otherwise, it is considered unschedulable. EDF-FF uses the following well-known test for EDF-schedulability on uniprocessors of tasks with arbitrary deadlines that uses the task densities (recall $\delta_i = e_i / \min(d_i, p_i)$) [BB08].

**Theorem 2 (Uniprocessor density test).** A task system $\tau = \{\tau_i : (e_i, d_i, p_i)\}_{i=1}^N$ is EDF-schedulable on a uniprocessor, provided that:

$$\sum_{\tau_i \in \tau} \delta_i \leq 1 \ .$$

(4.3)

The EDF-schedulability test in Theorem 2 is sufficient but not necessary. Any task system that passes the test is EDF-schedulable, but an EDF-schedulable task system might not pass the test. Exact tests (sufficient and necessary) for EDF-schedulability on a uniprocessor
(e.g., in [ZB09]) can be used in EDF-FF instead of the density test (Theorem 2), but their time complexity is higher.

Input: A task set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$
Output: $m$ (number of processors required),
$\{P_1, P_2, ..., P_m\}$ (partition of tasks to processors)
i:=1; m:=1; /* $i$ - Task index */
$\Delta_m:=0$; $P_m = \emptyset$; /* $\Delta_m$ - Total density on processor $m$ */
while ($i \leq n$)
    q:=1; /* $q$ - First-Fit processor number */
    while (($\Delta_q + \delta_i$)>1) /* EDF schedulability test */
        q:=q+1;
    end
    if (q>m)
        m:=q;
        $\Delta_m:=0$; $P_m = \emptyset$;
    end
    $\Delta_q:=\Delta_q + \delta_i$; $P_q := P_q \cup \{\tau_i\}$;
    i:=i+1;
end
return $(m, \{P_1, P_2, ..., P_m\})$

Listing 4.1: Earliest Deadline First - First Fit (EDF-FF)

Global EDF

In Global EDF, at runtime, arrived jobs are put into a queue that is shared by all the processors. Each job in the queue has a priority that is inverse to its absolute deadline, i.e., a job with a closer deadline has higher priority. At every moment, the $m$ highest priority jobs are selected for execution on the $m$ processors. Job preemption is allowed, and a preempted job returns to the queue and may resume execution on a different processor.

A well-known test for global EDF-schedulability of tasks is the following theorem [SB02, GFB03], which was proven to extend to tasks with deadlines larger than their period [BB08].

**Theorem 3** (Multiprocessor density test). A task system $\tau = \{\tau_i : (e_i, d_i, p_i)\}_{i=1}^N$ is global-EDF-schedulable on a platform comprised of $m$ unit-capacity processors, provided that:

$$\sum_{\tau_i \in \tau} \delta_i \leq m - (m - 1) \max_{\tau_i \in \tau} \delta_i .$$

As in the uniprocessor density test, the multiprocessor density test is sufficient, but not necessary. We now show an improved schedulability test that can be used for streams that satisfy the condition described next.

**Improved Schedulability Test** In the remainder of this section, we describe an improved schedulability test proposed by Baker and Baruah [BB09]. The test can be used for stream sets
where, for each stream, the latency bound is not shorter than the packet inter-arrival period, i.e., for every stream $s_i: (q_i, t_i, op_i, l_i), l_i \geq t_i$.

We first define the terms used in this test:

\[ DBF(\tau_i, t) \] For any interval length $t$, the demand bound function (DBF) bounds the maximum cumulative execution requirement by jobs of $\tau_i$ that both arrive in, and have deadlines within, any interval of length $t$.

\[ DBF(\tau_i, t) = \max(0, \left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1) e_i \] (4.5)

\[ LOAD(\tau) \] A measure of processor load, based upon the DBF function.

\[ LOAD(\tau) = \max_{t \geq 0} \left( \frac{1}{t} \sum_{\tau_i \in \tau} DBF(\tau_i, t) \right) \] (4.6)

\[ \mu(\tau) \] A bound on processor load. In the simpler test in Theorem 3, $\mu(\tau)$ is a lower bound on the total density of processor demand at which a task may miss a deadline. In Theorem 4, a tighter bound on processor load is presented.

\[ \mu(\tau) = m - (m - 1) \max_{\tau_i \in \tau} \delta_i \] (4.7)

\[ \delta_{\text{sum}}(\tau, \mu) \] The total density of the $([\mu] - 1)$ highest-density tasks.

\[ \delta_{\text{sum}}(\tau, \mu) = \sum_{k=1}^{[\mu]-1} \delta_k \] (4.8)

where tasks in $\tau$ are ordered such that $\delta_1 \geq \delta_2 \geq \ldots \geq \delta_N$.

**Theorem 4.** A task system $\tau = \{\tau_i : (e_i, d_i, p_i)\}_{i=1}^N$ is schedulable upon a platform comprised of $m$ unit-capacity processors, provided that:

\[ LOAD(\tau) \leq \max \{\mu(\tau)\delta_{\text{sum}}(\tau, \mu(\tau)), ([\mu(\tau)] - 1)\delta_{\text{sum}}(\tau, [\mu(\tau)] - 1)\} \] (4.9)

An extensive description and analysis of the test in Theorem 4 appears in [BB09].

In [FBB06] it is shown that it is sufficient to evaluate the maximum in the expression for $LOAD(\tau)$ only at points $\{d_i + kp_i | k \in \mathbb{N}, \tau_i \in \tau\}$ that are smaller than the least common multiple of all task periods $\text{LCM}_{\tau_i \in \tau}(p_i)$. The test validates that the maximum processor load is not higher than an expression that includes the number of processors $m$ and the density $\delta_i$ of the tasks in $\tau$. The test in Theorem 4 subsumes and extends the test in Theorem 3.

In [FBB06] it was shown that for the special case where for all tasks $d_i \geq p_i$, $LOAD(\tau) = \sum_{\tau_i \in \tau} u_i$ (where $u_i = e_i / p_i$ is the utilization). We show the proof for this theorem, starting with the following lemma:
Lemma 5. For any task \( \tau_i : (e_i, d_i, p_i) \) such that \( d_i \geq p_i \),

\[
\max_{t > 0} \left( \frac{1}{t} DBF (\tau_i, t) \right) = \lim_{t \to \infty} \left( \frac{1}{t} DBF (\tau_i, t) \right) = u_i.
\] (4.10)

Proof. For a given \( t \), let \( 0 \leq r < p_i \) be the value such that \( \left\lfloor \frac{t - d_i}{p_i} \right\rfloor = \frac{t - d_i - r}{p_i} \). Then,

\[
\frac{1}{t} DBF (\tau_i, t) = \frac{1}{t} \left( \left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1 \right) e_i = \frac{1}{t} \left( \frac{t - d_i - r}{p_i} + 1 \right) e_i = \left( 1 - \frac{d_i - p_i + r}{t} \right) e_i = \left( 1 - \frac{d_i - p_i}{t} \right) u_i.
\]

Since \( d_i \geq p_i \), \( \frac{1}{t} DBF (\tau_i, t) \) is bounded by a monotonically increasing function with respect to \( t \) that converges to \( u_i \) when \( t \to \infty \). For \( \frac{1}{t} DBF (\tau_i, t) \) itself, \( \lim_{t \to \infty} \left( \frac{1}{t} DBF (\tau_i, t) \right) = u_i \).

Hence, \( \max_{t > 0} \left( \frac{1}{t} DBF (\tau_i, t) \right) = u_i. \)

Theorem 6. For any task system \( \tau = \{ \tau_i : (e_i, d_i, p_i) \}_{i=1}^{N} \) such that \( \forall i : d_i \geq p_i \), the following is true:

\[
LOAD (\tau) = \sum_{\tau_i \in \tau} u_i.
\] (4.11)

Proof. In Lemma 5 we proved that for any \( \tau_i \) such that \( d_i \geq p_i \),

\[
\max_{t > 0} \left( \frac{1}{t} DBF (\tau_i, t) \right) = \lim_{t \to \infty} \left( \frac{1}{t} DBF (\tau_i, t) \right) = u_i.
\]

Consequently,

\[
LOAD (\tau) = \max_{t > 0} \left( \frac{1}{t} \sum_{\tau_i \in \tau} DBF (\tau_i, t) \right) = \sum_{\tau_i \in \tau} \left( \lim_{t \to \infty} \frac{1}{t} DBF (\tau_i, t) \right) = \sum_{\tau_i \in \tau} u_i.
\]

The schedulability test under these assumptions is much simpler:

Corollary 7. A task system \( \tau = \{ \tau_i : (e_i, d_i, p_i) \}_{i=1}^{N} \) such that \( \forall i : d_i \geq p_i \) is schedulable upon a platform comprised of \( m \) unit-capacity processors, provided that:

\[
\sum_{\tau_i \in \tau} u_i \leq \max \{ \mu(\tau) \delta_{\text{sum}} (\tau, \mu (\tau)), (\left\lfloor \mu (\tau) \right\rfloor - 1) \delta_{\text{sum}} (\tau, \left\lfloor \mu (\tau) \right\rfloor - 1) \}.
\] (4.12)
By definition, the task that corresponds to a data stream \( s_i : (q_i, t_i, op_i, l_i) \) has parameters \( d_i = l_i \) and \( p_i = t_i \). Therefore, in stream sets where for every stream \( s_i \), \( l_i \geq t_i \), the corresponding task set satisfies the condition in Corollary 7. Therefore, the schedulability test in Corollary 7 can be used for such stream sets.

We have shown an improved schedulability test for stream sets \( S = \{ s_i : (q_i, t_i, op_i, l_i) \}_{i=1}^{N} \) where for every \( s_i \in S, l_i \geq t_i \).

### 4.3 Rectangle Method for CPU/GPU Stream Distribution

In the previous sections we described the following principles for work distribution and scheduling to process a set of real-time streams on a CPU/GPU system:

- **Partitioned scheduling**: each data stream is assigned to a CU (CPU or GPU).
- **Batch processing on GPU**: packets from streams assigned to the GPU are periodically collected from the input buffers and processed in a 4-stage pipeline. The collection period is the same for all streams and defined by the system. From the point of view of the execution engine, each such collection of packets forms a batch of jobs to be executed by the hardware scheduler on the GPU.
- **GPU schedulability test based on performance model**: the validity of the schedule is tested by checking whether the collection period exceeds the expected processing time in any of the pipeline stages. The estimated latency of each stage is computed using an accurate performance model of the GPU and interconnect.
- **Classic multiprocessor scheduling on CPU**: existing multiprocessor scheduling and schedulability testing methods can be used to process streams on a multi-core CPU.

Using these principles, we provide a heuristic solution for the real-time stream processing problem. To complete our solution, it is necessary to define a policy for partitioning the streams between the CPU and the GPU. Each partition must be shown to be schedulable on its target CU. The problem addressed in this section is defined as follows:

**Problem 8.** For a set of real-time streams \( S = \{ s_i : (q_i, t_i, op_i, l_i) \}_{i=1}^{N} \) and a system with a CPU and a GPU, find a partition of \( S \) into two sets – one to be processed on the CPU and the other on the GPU – such that each set is schedulable on its target CU.

Unfortunately, even with two partitions, the search space is exponential in the number of streams, which makes exhaustive search impractical. We thus developed a heuristic that reduces the number of tested partitions to polynomial in the number of streams.

#### 4.3.1 Considerations in Workload Distribution Between a CPU and a GPU

A GPU poses several harsh constraints on its workload if the jobs are to be schedulable. It is therefore more efficient to first prune the jobs which are not schedulable on the GPU. Most importantly, invoking the GPU incurs overhead, which, coupled with the latency of data transfers
to and from the GPU’s memory, determine a lower bound on job processing time. Therefore, any job whose deadline is below this lower bound cannot be processed on the GPU and should be removed from its partition.

We next consider the job with the shortest deadline in the batch of jobs in the GPU’s partition. Because the batch completes as a whole, this job effectively determines the deadline for all the jobs in the batch. We will denote by $d_{\text{low}}$ the threshold for the shortest deadline of jobs assigned to the GPU’s partition. By lowering $d_{\text{low}}$ we restrict the maximum execution time of a batch, thus effectively decreasing the number of jobs that can be assigned to the GPU. By increasing $d_{\text{low}}$, we increase this number, with the penalty of having more short deadline jobs assigned to the CPU. It makes sense to limit the range of values of $d_{\text{low}}$ to one of the streams’ latency limits $l_i$, zero (0), or infinity ($\infty$). Once the value for $d_{\text{low}}$ is set, it is possible to determine the job collection period $T$ for GPU-assigned streams (Section 4.1), which is used for computing the job sizes.

A simple heuristic is therefore to test all partitions induced by $d_{\text{low}}$, where all jobs with the deadlines above $d_{\text{low}}$ are assigned to the GPU’s partition, and all the rest allocated to the CPU. This heuristic exhaustively tests all the created partitions for different values of $d_{\text{low}}$. It may fail, however, to find a schedulable partition for inputs with too many short-deadline jobs, even if one exists. For such a workload, decreasing $d_{\text{low}}$ would result in too tight a deadline for the GPU, while increasing $d_{\text{low}}$ would overwhelm the CPU with too many short-deadline jobs. A possible solution is to move a few longer-deadline jobs to the CPU’s partition, thereby decreasing the load on the GPU and allowing it to assist the CPU in handling shorter-deadline jobs. The longer-deadline jobs impose no extra burden on CPU scheduling, as short-deadline jobs would do. Therefore, a natural extension to a single threshold heuristic is to add an upper bound $d_{\text{high}}$ to limit the deadlines for jobs assigned to the GPU’s partition.

Both heuristics, however, ignore the amount of work per job. Consequently, the GPU may be assigned a highly unbalanced workload, which would seriously decrease its throughput (as explained in section 4.1), thus rendering the partition not schedulable. A reasonable approach to better load balancing is to remove work-intensive jobs from the GPU’s partition. The increased load on the CPU can be compensated for by moving shorter jobs to the GPU, whose overall throughput would thus increase. Deadlines should be taken into account when selecting the work-intensive jobs to move to the CPU in order to limit the deadline pressure on it.

Good workload distribution is, therefore, a tradeoff between setting a higher threshold on the shortest deadline and removing work-intensive jobs from the GPU’s partition. Therefore, we characterize each stream $s_i : (q_i, t_i, l_i, o_i)$ as $\langle r_i, d_i \rangle$, where $r_i = \frac{\lambda_i q_i}{l_i}$ is the computation rate – a measure of the job size, and $d_i = l_i$ is the relative deadline. Our workload distribution algorithms define the distribution using these properties.

The job size of stream $s_i$ in a batch of jobs created for stream set $S = \{s_i\}_{i=1}^n$ is defined as $W_i = \lambda_i \left\lceil \frac{T}{t_i} \right\rceil q_i$, where $T$ is the job collection period. However, we did not include $T$ in the expression for $r_i$ since $T$ is not a property of a stream. Note that if the number of computations per byte is the same for all streams ($\forall i : \lambda_i = \lambda$), then we can omit $\lambda_i$ from the expression for $r_i$ and define $r_i = \frac{q_i}{t_i}$, the stream’s data rate.
4.3.2 **RECTANGLE Method**

The **RECTANGLE** stream partitioning method assigns all streams inside a contiguous area in rate-deadline space to the GPU, and other streams to the CPU. In this method, this area is bounded by a rectangle, defined by four coordinates: \([d_{\text{low}}, d_{\text{high}}, r_{\text{low}}, r_{\text{high}}]\). For example, Figure 4.5 shows a partition of 10,000 streams with normally distributed rates and deadlines. Each dot represents a stream. All streams inside the black rectangle are assigned to the GPU, while the others are assigned to the CPU. We see that a stream is assigned to the GPU if its rate is in the range \([0 \text{bps}, 1.8 \text{Mbps}]\) and its deadline is in the range \([27 \text{ms}, 81 \text{ms}]\).

The number of rectangles in an \(n \times n\) space is \(\left(\frac{n(n-1)}{2}\right)^2 = O(n^4)\). We reduce the number of tested partitions to \(O(n^2)\), first by setting the lower rate bound \(r_{\text{low}}\) to 0 (since slower streams facilitate load balancing), and second, by increasing only the upper rate bound \(r_{\text{high}}\) and decreasing the upper deadline bound \(d_{\text{high}}\), for each value of \(d_{\text{low}}\). The full algorithm is shown in Listing 4.2.

Every partition returned by the algorithm produces deadline-compliant CPU and GPU schedules that were validated using the performance models. Following existing methods for real-time scheduling, it assumes perfect runtime prediction by the performance model [RM00, ER10, Bar06]. However, to compensate for imprecision, a safety margin is taken, which ensures 100% precision at the expense of slight degradation in schedulable throughput. Furthermore, with a
minor change to the algorithm, out of all the schedulable partitions (rectangles), we can choose the one for which the performance model predicts the largest safety (error) margin.

To reduce the number of tested partitions by a factor of $O(n)$, the method makes use of the following property of schedulability:

If a set of streams is schedulable on a CU, its subsets are schedulable as well. Similarly, if it is not schedulable, then no containing set is schedulable.

It is enough to test $O(n)$ partitions in order to find a schedulable assignment, if one exists, for the given lower deadline bound (with $r_{low}$ fixed to 0).

Performing a full schedulability test on each candidate partition is time consuming, as it includes execution simulation for multiple jobs on the GPU. To reduce the schedulability testing time, we first try to detect non-schedulable partitions using a light and less accurate test. The test simply checks whether the total processing demand on the CPU or the GPU is higher than their maximum processing throughput. If this test detects that any of the CUs would be overloaded, then a full test is no longer necessary and the algorithm can move on to the next partition. Normally, since the algorithm stops as soon as the first valid solution is found, most of the examined partitions are pruned at this test. To further speed up the search for partitions that passed the previous test, GPU schedulability is tested only if the CPU schedulability test failed. The latter test is usually much quicker because the CPU performance model and schedulability function are simpler and the GPU normally can handle many more streams than the CPU.

With $O(n^2)$ partitions tested, the search time for stream sets with thousands of streams can still be substantial, and it may be impractical to run it periodically to adjust for changes in the workload. However, by coarsening the step size in the rectangle borders, it is possible to control the search time at the expense of solution quality; for simplicity, this option was not included in the algorithm. Since some partitions are skipped, a valid partition or a better valid partition may not be found even though they exist. With coarsening, the stream set can also be mapped onto a grid defined by the step size, such that each cell will contain a small set of streams. This grid enables efficient extraction of streams inside or outside a rectangle from the full stream set and computation of their total load, which provide additional speedup. The RECTANGLE method can, therefore, be implemented in systems where the stream set is occasionally updated.

### 4.3.3 GPU Performance Optimizations

#### GPU load balancing

A GPU attains the best performance for massively parallel SPMD programs, with all the threads having the same execution path and runtime. Multi-stream processing, however, may deviate from this pattern. The difference in the rates of different streams results in greatly varying job sizes in the same thread block and degraded performance. Figure 4.6 shows that kernels that processed jobs with a wide spectrum of sizes took 48% more time for the kernel to process than equal-size jobs with the same total amount of work.
Threads that execute different jobs may follow different execution paths (diverge), resulting in serialized execution. Thus, limited fine-grain parallelism per job may require assigning several jobs to the same warp, where thread divergence and load imbalance might affect performance. We found, however, very little divergence for data stream processing, due to the periodicity of data stream functions. But load imbalance between the threads of a warp is detrimental to performance since the warp execution time is dominated by the longest job.

The distribution of jobs among warps and thread blocks is crucial to kernel performance. We use the following simple algorithm to increase GPU utilization.

1. Create warps with jobs of similar size;

2. Partition warps to $M$ bins such that total job size in each bin is about the same\(^1\), where $M$ is the number of SMs;

3. Assign the warps in each of the $M$ bins to thread blocks, such that warps with similar size are assigned to the same block.

Since the thread blocks are scheduled by the hardware, not all thread blocks created from a single bin can be guaranteed to execute on the same SM. Our approach attempts to combine good latency hiding, by creating thread blocks that process jobs of similar size, with balanced overall load on the SMs, by creating equally-sized bins. Our experience shows that the hardware scheduler can leverage these benefits to find an optimal schedule for the thread blocks. Experimentally, this approach provides better throughput than creating thread blocks with equal total job size and with maximally-similar job size.

Existing dynamic job scheduling methods for GPU, e.g. [CT09, CVKG10], use software load balancing mechanisms, such as job queues. The required synchronization incurs large overhead, especially for short jobs. These methods can be used in addition to ours since our method is offline.

\(^1\)This is a version of the PARTITION problem, which is NP-Complete. We use a popular greedy algorithm that provides $4/3$-approximation to the optimal solution.
Selecting kernel execution parameters

The total number of thread blocks plays an important role in GPU performance. Figure 4.7 shows the throughput of an AES-CBC encryption kernel as a function of the number of thread blocks on a GPU with 30 SMs. We see that choosing the number of thread blocks as a multiple of the number of SMs maximizes utilization of the cores, whereas other choices cause up to 42% performance loss. Our threads-per-block choice satisfies two conditions: (1) the number of threads per block is maximized to allow better SM utilization; (2) the total number of thread blocks is a multiple of the number of SMs to balance the load between them.

Data transfers

As mentioned in Section 4.1.2, incoming data for GPU-assigned streams is automatically stored contiguously in memory and periodically transferred to the GPU in a single data-transfer operation, which is more efficient than doing multiple small data transfers. This is done by pre-allocating a large chunk of GPU memory where contiguous blocks of memory (256 B) are manually allocated on-demand to store incoming data from the streams.

4.3.4 Real-Time Stream Processing Framework

We have implemented a framework for processing real-time data streams that we used to validate and evaluate the RECTANGLE method, and later, additional methods presented in this work. The framework includes an evaluation infrastructure, an implementation of the RECTANGLE method and the CPU and GPU performance models, and an execution engine. The entire framework was implemented in C, and a simulation-only version was also implemented in MATLAB. Since our framework uses an offline scheduler, the simulated schedules exactly match the schedules produced at runtime.

For evaluation, the framework includes an implementation of the AES-CBC 128-bit encryption algorithm as a stream processing operator. The AES 128-bit symmetric block cipher [Nat01] is a standard algorithm for encryption, considered safe, and widely used for secure
connections and classified information transfers (e.g., SSL). Several modes of operation exist for block ciphers. Cipher Block Chaining (CBC) is the most common – it enforces sequential encryption of data blocks by passing the encryption result of one block as the input for the encryption of the following block. Therefore, the AES-CBC stream processing operator requires storing the encryption result of the last block as the state for the next invocation of the operator on the stream.

The CPU implementation uses the Crypto++ 5.6.0 open-source cryptographic library with machine-dependent assembly-level optimizations and support for SSE2 instructions. For the GPU, we developed a CUDA-based multiple stream execution of AES. The implementation uses a parallel version of AES, in which every block is processed by four threads concurrently. Streaming data is simulated by an input generator that periodically generates random input for each data stream according to its preset rate.

### 4.4 Evaluation

In this section we evaluate our framework on a workstation with a multi-core Intel CPU and an NVIDIA GPU on thousands of streams.

#### 4.4.1 Experimental Platform

Our platform was an Intel Core2 Quad 2.33GHz CPU and an NVIDIA GeForce GTX 285 GPU card. The GPU has thirty 8-way SIMD cores and is connected to the North Bridge of an Intel P45 chipset on the main board by a 3GB/s PCIe bus. During the simulation we dedicate one CPU core to the input generator and another to control the GPU execution. Data is processed on two CPU cores and the GPU as an accelerator.

We compare the RECTANGLE method with two baseline techniques: MinRate, which attempts to optimize the scheduling of streams with respect to their rates, and MaxDeadline, which does the same with respect to the deadlines. In MinRate streams are sorted by rate in non-decreasing order \( \{s_i : (r_i, d_i)\} \). The first \( k \) streams (those with the highest rates) are assigned to the CPU and all others to the GPU, where \( k \) satisfies \( \sum_{i=1}^{k} r_i \leq \tau_{CPU}, \sum_{i=1}^{k+1} r_i \geq \tau_{CPU} \), and \( \tau_{CPU} \) is CPU throughput. In MaxDeadline streams are sorted by deadline in non-decreasing order, so that the CPU is assigned the streams with the shortest deadlines up to its capacity \( \tau_{CPU} \), and the GPU processes the rest.

To examine the maximal effective throughput of each method, we do not use safety margins for the execution time predictions in the experiments. As a result, the system may fail to process all the data on time for some workloads that pass the schedulability test. We measure the fraction of data that was processed on time, and use these measurements to determine the maximum throughput of each method. The accuracy of the performance model is evaluated in Section 4.4.4, and recommended safety margins for this system are determined there.
### 4.4.2 Setting Parameters

System throughput was measured as the highest total processing rate with all the data processed on time. We use the following notations:

- \( N \)  Number of streams
- \( R_{\text{tot}} \)  Total stream rate (Gbps)
- \( \mu_R, \mu_D \)  Average stream rate (Mbps), deadline (ms)
- \( \sigma_R, \sigma_D \)  Standard deviation of rate (Mbps), deadline (ms)
- \( \Delta_R, \Delta_D \)  Rate and deadline distribution functions

In the experiments we used Exponential (Exp) and Normal (Norm) distribution functions for rate and deadline. Such workloads are common in data streaming applications. To avoid generating extremely high or low values, we limited generated values to \([0.1\mu, 30\mu]\) for the exponential distribution, and to \([0.002\mu, 100\mu]\) for the normal distribution, where \(\mu\) denotes the average distribution value. Figure 4.5 displays an example of a randomly generated workload with the following parameters:

\[ N = 10K, \Delta_R = \Delta_D = \text{Norm}, \mu_R = 1Mbps, \sigma_R = 0.5Mbps, \mu_D = 50ms, \sigma_D = 15ms. \]

### 4.4.3 Results

We tested the efficiency of the RECTANGLE method on workloads with different distributions of stream rates and deadlines in three experiments. Each workload consisted of \( N = 12,000 \) streams. Figure 4.8 shows results of experiments where the average deadline is long \((\mu_D = 500ms)\), the standard deviation of the rate is high, and the distribution is exponential. In this experiment, the throughput of the RECTANGLE method was 30\% higher than that of MaxDeadline and similar to that of MinRate. MaxDeadline suffered from insufficient load balancing on the GPU, caused by simultaneous processing of streams with a wide range of rates. MinRate was effective because all deadlines were long, resulting in GPU processing of big batches.
Workload $N = 12K, \Delta_R = \text{Exp}, \Delta_D = \text{Exp}, \mu_D = 50 \text{ ms}$

Figure 4.9: MaxDeadline overloads the CPU with low-deadline streams. MinRate suffers from high batch scheduling overhead because low-deadline streams are processed on the GPU.

Figure 4.9 presents the performance of processing input with shorter average deadlines ($\mu_D = 50 \text{ ms}$). The workloads in this experiment are more time constrained, causing lower throughput in all three scheduling methods. MaxDeadline failed to produce a valid schedule for any value of the total rate larger than 2 Gbps, because the CPU was assigned more short-deadline streams than it could handle. For both MaxDeadline and MinRate, GPU throughput was limited by $r_{tot} \leq 5 \text{ Gbps}$. For MaxDeadline, the reason is inefficient load balancing due to high-rate streams, whereas for MinRate, the limiting factor is the batch scheduling overhead created by short-deadline streams. In comparison, the RECTANGLE method provides up to 7 Gbps throughput by finding the correct balance between rate and deadline.

Figure 4.10 presents results of experiments using normal distribution for rates and deadlines. Here we allow generation of values within a higher range, as described in Section 4.4.2. However, values are closer to $\mu$ on average due to lower variance. The shorter deadlines greatly reduce the throughput of MinRate, which makes no effort to avoid assigning short-deadline streams to the GPU. In contrast, MaxDeadline assigns these streams to the CPU, and benefits from a lower variance of stream rate than in the previous experiments. However, for low values of total rate, MaxDeadline assigns to the CPU more short-deadline streams than it can handle.

The number of processed data streams defines the amount of parallelism in the workload. We tested the efficiency of our method for $4K$ streams, where the GPU is underutilized because there are not enough jobs that can execute in parallel. Figure 4.11 shows that the RECTANGLE method outperformed the baseline methods by finding a balance between the rates and deadlines. Assignment of high-rate streams to the CPU increases the number of parallel jobs on the accelerator. Controlling the deadline increases batch size and reduces batch scheduling overhead.

For our last experiment we created workloads with exponentially distributed rates and deadlines while increasing the number of streams. Figure 4.12 shows that the high overhead of batch scheduling – a result of processing short-deadline streams on the accelerator – reduced the throughput of MinRate to below the measured range.
Workload $N = 12K$, $\Delta_R = Norm$, $\Delta_D = Norm$, $\mu_D = 50\,ms$, $\sigma_D = 15\,ms$

Figure 4.10: Both baseline methods overload the CPU on low rates. MinRate fails for all rates.

Workload $N = 4K$, $\Delta_R = Norm$, $\Delta_D = Norm$, $\mu_D = 50\,ms$, $\sigma_D = 15\,ms$

Figure 4.11: Low level of parallelism (4000 streams). The average rate of each stream is higher than in previous experiments. The highest throughput is achieved by the RECTANGLE method, which balances rate (increases the number of parallel jobs on the accelerator) and deadline (reduces batch scheduling overhead).

Workload $\Delta = Exp$, $\mu_R = 300\,Kbps$, $\sigma_R = 180\,Kbps$, $\Delta_D = Exp$, $\mu_D = 100\,ms$, $\sigma_D = 30\,ms$

Figure 4.12: Increasing the number of streams. The RECTANGLE method finds a schedulable solution for 17% more streams than MaxDeadline. MinRate chokes on the overhead of batch scheduling.
Table 4.1: Performance model estimation error

<table>
<thead>
<tr>
<th></th>
<th>COPY</th>
<th>KERNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AVG</td>
<td>MAX</td>
</tr>
<tr>
<td>CONST</td>
<td>3.3%</td>
<td>4.8%</td>
</tr>
<tr>
<td>EXP</td>
<td>3.0%</td>
<td>4.7%</td>
</tr>
</tbody>
</table>

Workload $N = 12K$, $\Delta_R = \text{Exp}$, $\Delta_D = \text{Exp}$, $\mu_D = 500$ ms (Fig. 4.10)

Figure 4.13: Safety margin for kernel time estimation. No misses with 20% margin.

4.4.4 Performance Model

The estimation error of our GPU performance model is shown in Table 4.1. Jobs of different sizes were randomly generated according to exponential and constant distributions. For each batch we calculated the estimated kernel time and data copying time, and measured its running time on the GPU. Then, we calculated the average and maximum precision errors. Figure 4.13 shows that with a 20% safety margin no deadlines are missed.

4.4.5 Time Breakdown

We analyzed the CPU and GPU states during processing. The results show that CPU scheduling overhead is less than 3% on all tested workloads. On average, this overhead is 1.4% for streams with constant rate distribution and 2.8% for streams with exponential rate distribution. The CPUs reach full occupancy with constant rates; idling is less than 1%. With exponential distribution, the assignment puts more load on the GPU and does not fully occupy the CPUs. Therefore, the CPUs have idle periods for low values of total rate. In the GPU, we see that processing time is not linear in the load. This is most apparent for the constant rate, where execution time grows more slowly than load.

Figure 4.14 shows a breakdown of GPU relative execution time for three pipeline stages: copy input to accelerator, kernel execution, and copy output to main memory. We see that in the case of constant rate distribution, the running time of the kernel is similar to the total copying time, about 50% of the total time. For exponentially distributed streams, the share of kernel execution time is significantly larger (72%), because batches with higher variance of job size do not fully utilize the GPU.

61
4.4.6 How Far Are We From the Optimum?

Figure 4.15 presents AES-CBC performance results on a single GPU for workloads consisting of batches with equal-size jobs. Since scheduling is trivial in this case, the results can be considered optimal for the given total aggregated rate. The columns show the throughput results for the following execution modes:

1. A GPU kernel executes one large batch of jobs.
2. A kernel executes a stream of relatively large batches.
3. A GPU executes a stream of relatively large batches, including memory copies from/to the CPU memory.
4. A CPU executes a stream of jobs.

According to these results, the performance of dynamic data processing on the GPU is 5% lower than for static processing. This is a result of additional accesses to the off-chip memory required for streaming data. Dynamic data is stored in memory as a chain of data blocks, and threads must perform memory accesses to follow the pointers between these blocks. This overhead does not exist for static data, as jobs are simply stored sequentially. Interestingly, there is an order of magnitude difference in throughput between the GPU and a CPU core for this algorithm.

The throughput of a system with a GPU and two CPU cores for real-life workloads is compared to a system with three CPU cores in Figure 4.16 (one of the four cores is used for input generation). In the experiments, both rates and deadlines of $12K$ streams were generated using the constant, normal and exponential distributions, with $50\text{ms}$ average deadline. The chart shows that our system achieves up to 4-fold speedup over a CPU-only system even for streams with $50\text{ms}$ deadlines.

4.5 Related Work

The problem of hard real-time stream processing can be mapped to the domain of real-time periodic task scheduling. A series of studies dealt with different aspects of task scheduling and
Figure 4.15: Optimal results for single GPU

Figure 4.16: Throughput of GPU+2CPUs is 4 times higher than that of 3CPUs
load balancing in CPU-GPU systems. Joselli et al. [JZC+08, JZC+09] proposed automatic task scheduling for CPU and GPU, based on load sampling. These methods are not suitable for hard real-time tasks, as there is no guarantee of task completion times. In specialized applications, such as FFT [OEMM08] or the matrix multiplication library [OKKY07], application-specific mechanisms for task scheduling and load balancing were developed.

Dynamic load balancing methods for single and multiple GPU systems using task queues and work-stealing techniques were developed [CVKG10, CT09, TPO10, ATNW11]. In [ATNW11], a set of workers on the accelerator execute tasks taken from local EDF (earliest deadline first) queues. These approaches cannot be used in our case as it is very hard to guarantee hard real-time schedulability in such a dynamic environment. An important work of Kerr et al. [KDY10] describes Ocelot, an infrastructure for modeling the GPU. Modeling can be used in our work to create a precise performance model without benchmarking.

Kuo and Hai [KH10] presented an EDF-based algorithm to schedule real-time tasks in a heterogeneous system. Their work is based on a system with a CPU and an on-chip DSP. This algorithm is not compatible with our system model because of the high latency of CPU to GPU communication. The problem of optimal scheduling of hard real-time tasks on a multiprocessor is computationally hard [Ram02]. Our work borrows some partitioning ideas from [RM00].

4.6 Conclusions

We have shown the first hard-deadline data stream processing framework on a heterogeneous system with CPU cores and a GPU accelerator. We have developed the RECTANGLE method, an efficient polynomial-time scheduling approach that uses simple geometric principles to find schedulable stream assignments. RECTANGLE method throughput was shown to be stable for different workloads and is higher than that of the compared baseline methods in all of the experiments. It is especially preferable for workloads with shorter deadlines and more streams. In Chapter 5 we extend the RECTANGLE method for systems with multiple CPUs and GPUs.
4.7 Rectangle Method Algorithm Pseudo-code

Input: Workload $W = \{s_i : (r_i, d_i)\}$, $n$ - length of $W$
Output: A schedulable Rectangle distribution of $W$:

$(W_{CPU}, W_{GPU})$ if successful, or $\emptyset$ otherwise.

$\text{sorted\_deadline} = \text{SortByDeadline}(W)$
$\text{sorted\_rate} = \text{SortByRate}(W)$
$\text{total\_load} = \text{SumRate}(W)$

/* indexes to sorted\_deadline and sorted\_rate: */
* $[d_{low}, d_{high}]$ - deadline range of $W_{gpu}$
* $[r_{low}, r_{high}]$ - rate range of $W_{gpu}$ */

$r_{low} = 1$ /* minimum rate is fixed at 0 */

for $d_{low} = 1...n$ /* array indexes start from 1 */

$r_{high} = 0$ /* initial high rate - minimum */
$d_{high} = n$ /* initial high deadline - maximum */
$\text{min\_deadline} = \text{GetDeadline}(\text{sorted\_deadline}[d_{low}])$

/* while loop: increase $r_{high}$ and/or decrease $d_{high}$ */

while $r_{high} <= n$ AND $d_{high} >= d_{low}$

/* schedulability test based on load vs. throughput */
* this is a light and efficient opportunistic test */

$\text{gpu\_load} = \text{LoadInsideRect}(d_{low}, d_{high}, r_{low}, r_{high})$
$\text{cpu\_load} = \text{total\_load} - \text{gpu\_load}$

if $\text{cpu\_load} > \text{MaxThroughput}('CPU')$

$r_{high} = r_{high} + 1$ /* reduce load on CPU */
continue /* skip to next configuration */

end

if $\text{gpu\_load} > \text{MaxThroughput}('GPU')$

$d_{high} = d_{high} - 1$ /* reduce load on GPU */
continue /* skip to next configuration */

end

/* schedulability test based on performance model */
* this is a more computationally heavy and accurate test than the load vs. throughput test */

$\text{cpu\_streams} = \text{StreamsOutRect}(d_{low}, d_{high}, r_{low}, r_{high})$
if $\text{IsSchedCPU}(\text{cpu\_streams}) == \text{false}$

$r_{high} = r_{high} + 1$
continue

end

$\text{gpu\_streams} = \text{StreamsInRect}(d_{low}, d_{high}, r_{low}, r_{high})$
if $\text{IsSchedGPU}(\text{gpu\_streams}, \text{min\_deadline}) == \text{false}$

$d_{high} = d_{high} - 1$
continue
else
/* partition found */

return $(\text{cpu\_streams}, \text{gpu\_streams})$
end
end

if d_high == n
/* the CPU cannot handle this or higher d_low, abort */
break
end
end

return ∅
Chapter 5

Stream Processing in a Multi-GPU System

Current data processing systems employ multiple accelerators on each machine. Placing multiple GPUs on a single board saves power, floor space and maintenance costs, and allows performance scaling with low latency and good load balancing. The RECTANGLE method does not naturally support multiple GPUs. It is designed to offload a small portion of the workload to a CPU in order to maximize the efficiency of a GPU. In contrast, with multiple GPUs, it is not enough to maximize the throughput of any one GPU by optimizing its share of the workload. Rather, it must be decided how to partition the workload between the GPUs and which portion of each partition to offload to the CPUs, in attempt to maximize the total throughput. Additionally, the temporal analysis of data transfers in a multi-GPU environment is significantly more complicated than in single-GPU systems due to contention and interference of simultaneous data transfer operations on the buses of the interconnect. Scheduling input and output data transfers between the main memory and the GPUs in a way that guarantees their completion on time is essential for optimizing the workload distribution between the GPUs. We address this important problem in Chapter 6.

On the basis of the RECTANGLE method, we developed several methods for data stream distribution between multiple CPU and GPU devices. The methods present different compromises between the number of tested partitions and the achievable throughput. The methods are illustrated in Figure 5.1 and summarized in Table 5.1.

5.1 Virtual GPU

For multiple GPUs, we apply the RECTANGLE method with a virtual GPU configuration. This virtual GPU represents the unification of the compute and memory resources of all the GPUs in the system under a single GPU device. For example, the unification of two GPUs with 30 SMs each, 1 GByte of on-board memory, and a 3 GByte/sec PCIe connection would be represented by a virtual GPU with 60 SMs, 2 GBytes of on-board memory, and a 6 GByte/sec PCIe connection.
Virtual GPU Split Rectangle Joined Rectangles Loose Rectangles

Figure 5.1: Illustration of workload distribution methods for multiple GPUs in the rate-deadline space

<table>
<thead>
<tr>
<th>Multi-GPU method</th>
<th>Properties (incremental)</th>
<th>Tested partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual GPU</td>
<td>Simple extension</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Split Rectangle</td>
<td>Reduced batching overhead or improved load balancing</td>
<td>$O(n^2g \log(n))$</td>
</tr>
<tr>
<td>Joined Rectangles</td>
<td>Reduced batching overhead and improved load balancing</td>
<td>$O\left(\left(\frac{n}{g+1}\right)g \log(n)\right)$</td>
</tr>
<tr>
<td>Loose Rectangles</td>
<td>Increased utilization for deadline-constrained CPU</td>
<td>$O\left(\left(\frac{n}{2g}\right)g \log(n)\right)$</td>
</tr>
</tbody>
</table>

$n$ – number of streams  
$g$ – number of GPUs

Table 5.1: Summary of workload distribution methods for multiple GPUs

Figure 5.2: Workload distribution to multiple GPUs in the Virtual GPU method. Streams are partitioned between the SMs of a virtual GPU. This maps each stream to an SM that corresponds to a real GPU.
Given some workload, if the Rectangle method produced a valid CPU/GPU partition, then it found the GPU partition schedulable on the virtual GPU. Figure 5.2 illustrates a part of the schedulability test where streams are partitioned between SMs (mapped to SMs). If the workload is found schedulable, then the streams are partitioned between the CPUs and (non-virtual) GPUs. Each stream in the virtual-GPU partition is assigned to the GPU that corresponds to the SM it is mapped to. The complexity of tested workload partitions is the same as in the Rectangle method, $O(n^2)$.

When jobs are actually processed on a GPU, the produced stream-to-SM mapping is not enforced since work on the GPU is scheduled by a hardware scheduler. The estimated execution time could thus be shorter than the actual time, and this error could result in the workload being falsely characterized as schedulable. The performance model avoids underestimation of execution times by using a scheduling policy that trivially assigns the same number of randomly chosen streams to each SM. In contrast, the hardware scheduler of a GPU dynamically schedules thread blocks for execution when resources become available, and although the scheduling policies of GPUs are not disclosed, we assume that they are superior to the simple scheduling policy of the performance model.

We evaluate the performance of this method in Section 5.5.

5.2 Split Rectangle

When mapping streams to SMs, the Virtual GPU method does not attempt to prioritize the SMs of one GPU over those of another. As a result, the streams with deadlines that are equal to or close to $d_{\text{min}}$ are distributed among all GPUs. Effectively, on every GPU, the minimum stream deadline is approximately $d_{\text{min}}$. Since the batch collection period is $\frac{1}{4}d_{\text{min}}$, batch processing on the GPUs is synchronous.

This does not have to be the case. In fact, it might be beneficial to use different batch collection periods for different GPUs. For example, suppose that a workload $S$ with a minimum stream deadline $d_{\text{min}} = 8\ ms$ is partitioned between GPU1 and GPU2. Using the Virtual GPU method, the collection period for both GPUs is $8/4 = 2\ ms$. However, if all streams with deadlines in $[8\ ms, 20\ ms]$ are assigned to GPU1 and all streams with deadlines higher than $20\ ms$ to GPU2, then the batch collection period will be $2\ ms$ for GPU1, and $5\ ms$ for GPU2. The throughput of GPU2 might then be higher in the second case. Similarly, it may be beneficial to split the rectangle between the GPUs according to rate ranges instead of deadline ranges.

This method uses the Rectangle method with a modified GPU schedulability function. Whenever schedulability on the GPU is tested for a set of streams bounded in a rectangle, the function attempts to split the rectangle into vertical (deadline range) or horizontal (rate range) stripes, and assigns each stripe to a different GPU. If a split is found in which all stripes are schedulable on their respective GPUs, either by rate or by deadline, then the given rectangle is considered to be schedulable. For each of the $O(n^2)$ calls to the schedulability function by the Rectangle method, the modified function makes $O(g\ log\ (n))$ schedulability tests to find the
best split. Hence, the number of tested partitions is $O \left( n^2 g \log(n) \right)$.

### 5.3 Joined Rectangles

In the Split Rectangle method, the workload is distributed between GPUs in a way that optimizes throughput by one parameter: rate or deadline. Here, we select the optimal rate bound for every vertical strip (range of deadline values). This approach attempts to maximize the utilization of each GPU for the input range, while taking advantage of the per-GPU minimum deadline. An exhaustive search of a schedulable partition can be done efficiently by selecting $g + 1$ deadline range bounds, and then, for every range $i$, using binary search to find the maximum $r_{high}$ that defines a strip schedulable on GPU $i$. However, the search complexity in this case is significantly higher than in the previous methods, and the GPU-assigned workload is still bound to a contiguous deadline range. The number of tested partitions is $O \left( \frac{n^g}{g+1} g \log(n) \right)$.

### 5.4 Loose Rectangles

In workloads for which the CPU cannot process all the short-deadline streams, the Rectangle method assigns some long-deadline streams to the CPU in exchange for reducing the shortest deadline for the GPU. With multiple GPUs, it makes sense to assign all the short-deadline streams that were not assigned to a CPU to one GPU, while the other GPUs benefit from longer minimum deadlines in their working sets. Here, we do not enforce a contiguous deadline range on the area processed by the GPUs. Each GPU is assigned a rectangle, without overlapping. A search of a schedulable partition can be done in a manner similar to the Joined Rectangles method, but with $2g$ range bounds. This method achieves the best overall throughput, but its search complexity is high. The number of tested partitions is $O \left( \frac{n^g}{2g} g \log(n) \right)$.

### 5.5 Evaluation of Virtual GPU

In this section we evaluate our framework using modern hardware on thousands of streams.

#### 5.5.1 Experimental Platform

Our platform was an Intel Core2 Quad Q8200 2.33Ghz CPU and two NVIDIA GeForce GTX 285 GPUs with CUDA 4.1. Each GPU has thirty 8-way SIMD cores and is connected to the North Bridge of an Intel P45 chipset on the main board by a 3GByte/sec PCIe bus. We used our real-time stream processing framework as a testing platform. The workload is based on the AES-CBC data stream encryption application. During the simulation we dedicate one CPU to the input generator and another to control the GPU execution. Data is processed on two CPU cores, using the GPUs as accelerators.

We also developed a tool that estimates the system throughput for a given type of workload on an arbitrary configuration of compute units. We use this tool to (1) compare the performance
of our method to simpler heuristics that we consider as baseline in Section 5.5.3, and (2) estimate how system performance scales for up to 8 GPUs in Section 5.5.4.

5.5.2 System Performance

We tested the efficiency of our method on workloads with different distributions of stream rates and deadlines in five experiments using one and two GPUs. Each workload consisted of 12,000 streams. In each workload we used the constant, normal or exponential distributions to randomly generate the rates and the deadlines. Such workloads are common in data streaming applications, and test the framework for robustness to large variance and asymmetric distribution. To avoid generating extremely high or low values, we limited generated figures to $[0.1\mu, 30\mu]$ for the exponential distribution, where $\mu$ denotes the average distribution value, and to $[0.002\mu, 100\mu]$ for the normal distribution. In each experiment, we measured the maximum throughput (sum of stream rates) for the given workload specification. The average stream rate can be calculated by dividing the maximum throughput by 12,000. For brevity, we will use the term throughput instead of maximum throughput.

Figure 5.3 shows the throughput of the framework with 2 CPUs and one or two GPUs, on 5 different workloads. The overall throughput with two GPUs was up to 87% higher than with one GPU, and the GPU-only throughput (the total rate processed by the GPUs) increased by up to 106%. An increase of more than 100% in GPU-only throughput may occur, together with an increase of less than 100% in CPU-only throughput, when the total speedup is not higher than 100%.

In the first experiment, equal rates and deadlines were assigned to all streams. The framework achieved the highest throughput for this workload since there is no load imbalance for GPU jobs and a relatively large batch collection period can be used. In the second experiment, the framework achieved 97% of the maximum throughput for a realistic workload of streams with normally distributed rates and constant 40ms deadlines. In workloads with exponential distribution of rates (experiments 3 and 5), stream rates were as high as 150 Mbit/sec. The gain in throughput from an additional GPU for such workloads is smaller because the number of streams with very high rate that can be offloaded to the CPUs is limited. Such streams can create great load-balancing problems for the GPU. In the fourth experiment, the framework finds a provably deadline-compliant schedule for a workload that contains streams with sub-millisecond deadlines, with throughput as high as 94% of the maximum.

5.5.3 Comparison to Baseline Methods

We compared our method with two baseline techniques: MinRate and MaxDeadline. MinRate attempts to optimize the scheduling of streams with respect to their rates, and MaxDeadline does the same with respect to the deadlines. In MinRate, streams are sorted by rate, and the streams with the highest rates are assigned to the CPUs, up to their capacity, and the rest are assigned to the GPUs. In MaxDeadline, streams are sorted by deadline, and the
streams with the shortest deadlines are assigned to the CPUs, up to their capacity, and the rest are assigned to the GPUs. Between GPUs, streams are equally distributed in random order.

We estimated the system throughput of each method by applying a performance estimation tool that we developed. The tool simulates the distribution of a workload to a set of compute units and estimates the resulting throughput. Its input is a set of CU specifications (CPUs and GPUs), the desired workload distribution method, and a specification of the workload. The tool generates a large pool of stream configurations following the given workload specification, \( \text{POOL} = \{ s_i : \langle \text{deadline}_i, \text{rate}_i \rangle \} \), and finds the maximum \( N \) for which the workload \( \{ s_1, \ldots, s_N \} \) has a schedulable distribution (using the method) between the CUs. Internally, the CU specifications are passed to the performance model, which is used in order to determine whether a distribution is schedulable. The tool uses benchmark results and specification data of GPUs to make the performance model as precise as possible. Note that the tool uses a workload specification format in which the number of streams is not specified. Therefore, its throughput results may not match the results in Figure 5.3.

We ran a series of five experiments with different distributions of stream rates and deadlines for each of the compared methods. Figure 5.4 shows the throughput of our method (Virtual GPU), MinRate, and MaxDeadline. The table below the graph describes the settings used for generating the streams.

Our method provided higher throughput in all of the experiments. For the const workload, all the methods provided similar throughput since the deadlines and rates of the streams were the same. MinRate was never better than MaxDeadline. Since this method ignores deadlines, it assigned streams with very short deadlines to the GPUs, thus imposing a very short and inefficient processing cycle. The performance of MaxDeadline was better, but the schedules it produces suffer from load-balancing problems of varying severity. The performance of Virtual GPU was higher by up to 51% than that of MaxDeadline.
5.5.4 Scaling to More GPUs

We tested how system throughput scales with up to 8 GPUs by applying the performance estimation tool that was described in Section 5.5.3. We ran a series of five experiments, the settings for which are shown in the table in Figure 5.4. We used configurations of two CPU cores and between 1 and 8 GPUs, with the CPU and GPU specifications that were described in Section 5.5.1. Since the number of CPUs is kept constant, the purpose of these experiments is to show how the throughput of the GPUs scales with the number of GPUs rather than with the total throughput.

Figure 5.5 shows the throughput of the system for experiment const. The throughput scales perfectly in the number of GPUs because every GPU processes the same number of streams as the first. Figure 5.6a shows the throughput of the system for exp. A sample of 200 such streams is illustrated in Figure 5.6b. The system does not scale perfectly, with x4.3 speedup in GPU-only throughput for 8 GPUs. This might be surprising at first, since streams are drawn from a pool with randomly generated values. Each additional GPU would apparently achieve similar utilization with a similar number of additional streams. However, as shown in Figure 5.6c, the minimum deadline of streams on the GPUs \( d_{\text{min}} \) gets shorter as the number of GPUs is increased. As a result, the batch collection period \( T = \frac{1}{4}d_{\text{min}} \) gets shorter, and the batching overhead increases (as explained in Section 4.1). The reason for the decrease in \( d_{\text{min}} \) lies in the limited capacity of the CPU. As more streams with short deadlines are drawn from the pool...
Figure 5.5: Perfect scaling (gray line): all streams have a 20.5ms deadline and 3.1Mbit/sec rate and assigned to the CPU by the Rectangle method, streams with slightly higher deadlines that were previously assigned to the CPU get reassigned to the GPU. If the number of CPUs scaled with the number of GPUs, then we would expect to get perfect scaling of the total throughput. In this experiment, the batch collection period gets as low as 0.4 ms.

The minimum and maximum scaling observed in all of the experiments is shown in Figures 5.6a and 5.5, respectively.

### 5.5.5 Transfer Rates

Table 5.2 presents the maximum transfer rates between the main memory (Host) and the GPUs (Devices), and compares them to the maximum kernel performance for AES-CBC. One can see that the bandwidth of two GPUs is in the order of twice the bandwidth of a single GPU. This is in line with the bandwidth in the hardware model. Namely, each GPU gets half of the 16 PCI-Express lanes. These lanes connect the GPUs to the North Bridge, which is directly connected to the memory. The GeForce series GPU cards have one DMA engine, and cannot work in full-duplex. Hence, the rate of data transfer to the device and back is half the average uni-directional rate. Data transfers are a bottleneck of the system for AES-CBC when the kernel runs at maximum performance.

Figure 5.7 presents data transfer rates for a range of chunk sizes using one and two GPUs. For chunks of size 4 KiB or smaller the transfer time ranges around 20 µs, and the transfer rate is very low accordingly (up to 0.22 GByte/sec). For larger chunk sizes, the rate increases with size, reaching 95% of the maximum with chunks of 2 MiB (per GPU).

As opposed to always assuming maximum rate, we use a transfer-rate lookup table in our performance model to predict data-transfer time. This is especially important when deadlines are short. Since such deadlines impose short batch processing cycles, data is transferred in small chunks, and the transfer rates are lower than the maximum.

### 5.5.6 Accuracy of Performance Model

The estimation error of our GPU performance model for the workloads described in Section 5.5.2 is shown in Table 5.3. A negative error means that the measured execution time was
Figure 5.6: GPU speedup is sub-linear due to decreasing minimum stream deadline. Deadlines and rates were generated using the \textit{truncated exponential} distribution.

\begin{table}[h]
\begin{center}
\begin{tabular}{|l|c|c|c|c|}
\hline
 & D-to-H & H-to-D & H-to-D-to-H & GPU kernel \\
\hline
1 GPU & 2.77 & 3.355 & 1.517 & 1.875 \\
2 GPUs & 5.31 & 5.81 & 2.775 & 3.75 \\
Speedup & x1.92 & x1.73 & x1.83 & x2 \\
\hline
\end{tabular}
\end{center}
\caption{Maximum transfer rates and kernel throughput in GByte/sec ($G = 10^9$)}
\end{table}

Transfer rate on two GTX285 GPUs with P45 chipset

Figure 5.7: Data transfer rate to the GPU and back is lower than maximum kernel throughput for AES-CBC
higher than the estimation. For example, the measured execution time of the kernel was never higher than the estimated time by more than 2%. In the framework, we leave a safety margin of 10% to account for performance model errors. As the figures show, the performance model never estimated any time by less than 94% of the measured time. When the kernel runtime is very short (less than 2 ms) and the batch is composed of jobs with a wide range of sizes (experiment 5), the estimation error is large. This can be explained by: (1) the inaccurate estimation of small constant factors, which results in a large relative error when the runtime is short, and (2) the deliberately simplistic scheduling policy in the performance model (as described in Section 4.1.2). Interestingly, there was a 17% error in the prediction of small data transfers with one GPU despite using a lookup table for the expected transfer rate. Apparently, the actual time of the data transfers was longer than expected since it was calculated including a CUDA device synchronization command, while in the benchmark data transfer times were measured more accurately using events.

5.6 Summary

The use of multi-GPU systems for real-time data stream processing presents new challenges in work distribution and data transfer scheduling. This chapter presented four new methods for data stream distribution between multiple CPUs and GPUs. The methods present different compromises between the number of tested partitions and the achievable throughput. For single-GPU systems, the RECTANGLE method presented in Chapter 4 aims to maximize the system throughput by offloading streams that reduce GPU efficiency to the CPU. In contrast, with multiple CUs (CPUs and GPUs), the cumulative throughput of the GPUs (and the system throughput in general) needs to be maximized, so it is necessary to decide how the streams will be distributed between the GPUs. The GPU throughput can be optimized, through stream distribution, by reducing load imbalance in the workload of each GPU, and by allowing longer processing periods on the GPU, which results in more efficient data transfer and reduced kernel invocation overhead.

When using GPUs in asynchronous processing pipelines, it is difficult to schedule the CPU-GPU data transfers in a way that does not violate the timing constraints. Estimation of the time to complete a data transfer is complicated by possible contention in the interconnect caused by other asynchronous data transfers that happen to run at the same time, and made more difficult by the complex architecture of the interconnect. We address this important problem in Chapter 6.

The first method we presented, VIRTUAL GPU, solves the data transfer scheduling problem

<table>
<thead>
<tr>
<th></th>
<th>Kernel</th>
<th>H-to-D copy</th>
<th>D-to-H copy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIN</td>
<td>MAX</td>
<td>MIN</td>
</tr>
<tr>
<td>1 GPU</td>
<td>-2%</td>
<td>15%</td>
<td>-2%</td>
</tr>
<tr>
<td>2 GPUs</td>
<td>-2%</td>
<td>21%</td>
<td>-6%</td>
</tr>
</tbody>
</table>

Table 5.3: Performance model estimation error
by enforcing a synchronous operation mode for the GPUs, hence avoiding unexpected interference between data transfers to (or from) different GPUs. The method makes use of synchronous data transfers to all the GPUs, which causes bus contention but also achieves a higher data transfer rate and predictable transfer times. We evaluated the method using a real system constructed of a four-core CPU and two GPUs and ran AES-CBC encryption to process different streams of data with a variety of deadlines. We performed extensive experiments on thousands of streams with different workloads. We used a software tool to estimate the performance of the algorithm on systems configurations with up to 8 GPUs.

Our current framework allows up to 87% more data to be processed per time unit than the one in our previous work by taking advantage of a second GPU, and up to a 106% increase in GPU-only throughput. We also show that by adding two GPUs to a quad-core machine, we get a 7-fold throughput improvement over highly optimized CPU-only execution.
Chapter 6

Data Transfer Scheduling on the Interconnect

6.1 Introduction

In this chapter we present the design of our scheduler and execution engine for data transfers. Scheduling the data transfers is necessary to guarantee deadline-compliant execution in the multi-GPU data stream scheduling methods presented in the previous chapter. The scheduler is based on a new approach where the basic unit of work sent for execution (a job) is a batch of data-block transfer operations. Movement of data between memories and compute units is an integral part of data stream processing. In an RTSPS, data transfers and computations must be carefully scheduled to maintain logical correctness and guarantee that the processing latency bound of each stream is not exceeded. In the previous chapter, we showed that a method for scheduling CPU-GPU data transfers would enable advanced work distribution techniques in multi-GPU RTSPSs. Studying the problem of scheduling data transfers in real-time systems is important for a number of reasons:

- **The interconnect can be a bottleneck for performance.** Figure 6.1a compares the growth of computational throughput to the growth of CPU-GPU bandwidth for compute nodes with NVIDIA Tesla-series GPUs\(^1\). The figure shows that the compute power is growing much faster than the communication bandwidth. The gap increases substantially for multiple GPUs. As shown in Figure 6.1b, the ratio between aggregate computation and data transfer rates grows with the number of GPUs\(^2\). For example, when scaling from four to eight GPUs in a dual-socket system, the CPU-GPU bandwidth remains the same because the system multiplexes GPU pairs on the same PCIe bus, while the compute throughput doubles. The development of data transfer scheduling methods that make efficient use of the available bandwidth is, therefore, crucial for high performance.

---

\(^1\) 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.

\(^2\) Using a dual-socket system for four GPUs or more.
• **Data transfer time is an integral part of the processing latency.** Data transfers are essential for getting the data to and from the compute units for processing; hence, the transfer time is included in the streams’ processing latency and must be taken into account when checking that timing constraints are met.

• **Heterogeneous nodes have become standard in high-performance systems.** Heterogeneous compute nodes that consist of an interconnected set of CPUs, massively-parallel accelerators, and high-speed network interfaces have become standard in high-performance systems. It has been noted that collaboration between different CUs provides benefits in performance and energy consumption for applications in various domains, including numerical methods, image and video processing, data mining, database systems, and computational science [MV15]. Despite this trend, the state-of-art in real-time scheduling theory for heterogeneous systems is not fully developed [Rar14]. The existing research in this field leans heavily towards scheduling the execution of tasks on compute units (e.g., [Rar14, Fun04, AR14, NWB11, TRC11]) and much less attention has been given to scheduling of data transfers on the system interconnect. Due to the centrality of data transfer for CU collaboration, we find data transfer scheduling an important topic of study.

• **New hardware models are required for effective scheduling of modern interconnects.** In multiprocessor real-time systems with a distributed topology, the interconnect serves as a shared resource that is managed by a data transfer scheduler. Traditionally,
interconnect schedulers in real-time systems assume simple topologies, such as a fully connected network or a single shared bus, where data transfers are performed one at a time to avoid contention [DAN+11, RAEP07, LS86]. Sinnen and Sousa [SS05] studied the problem of communication contention in task scheduling. They considered the problem of scheduling a program that is defined as a directed acyclic graph (DAG), where a vertex represents a computational task and an edge represents communication between tasks, on a network of point-to-point buses and shared buses. The communication scheduling is performed at the packet level, and it is assumed that a bus transmits one packet at a time. Contention is considered at the endpoint (limited number of connections) and network (occupied buses). In CPU-GPU systems, the packet scheduling algorithms are embedded into the interconnect and fixed, which excludes the use of the algorithms proposed in [SS05]. The scheduling problem in CPU-GPU systems is of application-level data transfers, which can execute concurrently on the same bus, splitting its bandwidth; hence, the model proposed in [SS05] is not appropriate in this case. Hsiu et al. [HLK09] considered real-time task scheduling in a multi-layer bus system. A multi-layer bus system consists of heterogeneous processing elements that communicate with one another via multilayer buses. Such a bus includes several layers and each layer can execute one communication task at a time. This model also does not fit because it does not reflect dynamic bandwidth redistribution.

• This model does not fit the interconnect of a system with multiple CUs, 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 route changes in accordance with bus contention. A new hardware model for the interconnect is therefore required for effective scheduling of data transfers in RTSPs.

Recall that the processing model of data streams on a GPU is a synchronous pipeline that includes data aggregation in main memory, data transfer to local GPU memory, kernel execution, and transfer of results to main memory. From the point of view of the scheduler, within every period of the pipeline, a chunk of data needs to be transferred between the main memory and the GPU. In this chapter we consider a more general problem, where periodic data transfer tasks may be required not only on a CPU-GPU route but between any pair of endpoints on the interconnect. As we show later, the extended problem can be used to support complex data stream processing operations that include splitting packets between multiple CUs and communication between CUs.
6.2 Problem Definition

6.2.1 System Model

We consider an RTSPS with multiple compute units and a heterogeneous interconnect. Here, we use an extended data stream model that supports complex processing operations. The next three subsections provide a formal definition of the data stream model, detail the system architecture, and describe the execution model.

Extended Real-Time Data Stream Model

Recall that a data stream is a periodic source of data packets that need to be processed within a given latency bound from their arrival (see Section 3.2). A data stream \( s_i \) is characterized by the tuple \((p_i, t_i, op_i, l_i)\), whose parameters are as follows:

- \( p_i \) – packet size;
- \( t_i \) – time period between two consecutive packet arrivals;
- \( op_i \) – processing operator applied to the packets (in order of their arrival);
- \( l_i \) – latency bound on processing a packet.

In the extended model, the packet processing operator \( op_i \) is implemented as a chain of alternating data-transfer and computation steps \( (procstep_k)_{k=1}^{n_i} \) that are executed sequentially. For example, the processing chain that is illustrated in Figure 6.2 consists of three steps:

1. Split input data in two and transfer each part to a different GPU.
2. Run a kernel on each GPU to process the data.
3. Transfer results from the GPUs to an output buffer in CPU memory.

The computations and data transfers are mapped to specific compute units and data transfer routes on the interconnect.

A data transfer step is composed of one or more data-block transfers between distant memories in the system, and is characterized by \( \{(x_k, src_k, dst_k)\}_{k=1}^{n_i} \), where \( x_k \) denotes the size of the data block and \( (src_k, dst_k) \) denote the communication endpoints. A computational step runs computation on one or more processors, and is characterized by the set \( \{(proc_k, kern_k)\}_{k=1}^{m_i} \), where \( proc_k \) is the processor identifier (e.g., a CPU core or a GPU) and \( kern_k \) is the computation to be executed.
The system processes multiple data streams using a centralized (not distributed) runtime engine that executes data transfer operations over the interconnect and launches kernels on the compute devices according to a predefined schedule. A data transfer operation cannot be aborted, but can include any part of a data block that needs to be transferred.

In order to make efficient use of the resources, it (1) processes each stream in a synchronous pipeline that executes data transfer and computation steps in different stages (Figure 6.3), and (2) executes multiple operations on the same resource in parallel. The cycle time in the processing pipeline of each data stream is set to $T_i = l_i / n_i$, where $l_i$ is the stream’s relative deadline and $n_i$ is the number of processing steps. Consequently, the processing pipelines of different streams may be asynchronous.

The pipeline execution mode for each stream eliminates scheduling dependencies between processing steps by assigning a time window for each operation, which simplifies the scheduling problem. Furthermore, since each operation requires either computational resources or the interconnect, data transfers and kernels can be scheduled independently. This chapter focuses on the data transfer scheduling problem.

The data transfer requirement from the system is a set of periodic data transfer tasks, where each task represents a data transfer step of a data stream. Following previous studies on scheduling data transfers [LS86, DM01, SS05], we will use the term message to describe a data transfer task.

**Definition 9.** 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.

For every data-transfer step $\{(x_k, src_k, dst_k)\}_{k=1}^{n}$ of every stream $s_i$ processed by the system, the message set includes a message $m_{i,k} = (x_k, src_k, dst_k, T_i, 0)$. 

---

Figure 6.3: Asynchronous processing pipelines for two data streams
Heterogeneous Interconnect

The system interconnect consists of a network of controllers, memory nodes, and endpoint devices that are linked by point-to-point buses, and it is used to transfer data between memories in the system.

Endpoint devices act as nodes in the network. Each endpoint device directly loads and/or stores data in local memories, which reside either inside the device or externally as memory nodes. An endpoint device may include one or more DMA controllers that operate data transfers to and/or from it. A data transfer operation copies a chunk of data between the local memories of two endpoints on a fixed route by using a corresponding DMA controller. Each route of data transfer is associated with a specific 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.

The buses act as edges in the network. Each bus is composed of either one data transfer channel that can pass data in one direction or two such channels that can pass data in opposite directions. The bus has a certain bandwidth for each direction, which denotes the maximum total data transfer rate that can pass through it in this direction. The bandwidth of buses may vary due to the combination of different communication technologies in various parts of the interconnect. During the execution of a data transfer operation, the data exceeds through all the buses on the route from the source to the destination at a certain transfer rate. The rates of the data transfers that are in progress are limited by the bandwidths of the buses such that the bandwidth of at least one bus on each route is saturated (the saturated bus passes the maximum data rate). When a data transfer begins or ends, the bandwidths are redistributed instantly and automatically. Direct accesses of endpoint devices to local memories are assumed to be isolated from DMA transfers by either static reservation of bandwidth in buses on the route or by not executing them at the same time.

The bandwidth bottleneck of a data transfer does not always have to be in the same bus, as shown in the following example. Consider the architecture of the Tyan FT72B7015 system with four GPUs illustrated in Figure 6.4. The following table lists the (theoretical) bandwidth of the communication domains and their bus types in this system:

<table>
<thead>
<tr>
<th>Communication domain</th>
<th>Bus type</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory bus</td>
<td>DDR3</td>
<td>32</td>
</tr>
<tr>
<td>processor interconnect</td>
<td>QPI</td>
<td>9.6</td>
</tr>
<tr>
<td>PCIExpress bus</td>
<td>PCIe 2.0</td>
<td>8</td>
</tr>
</tbody>
</table>

One can see that QPI has higher bandwidth than PCIe 2.0. Therefore, the bandwidth from CPU1 to GPU1 is limited by PCI Express. However, the bottleneck for aggregate bandwidth is not necessarily in the bus with the smallest bandwidth. In the example, QPI is the bottleneck for the aggregate bandwidth from the main memory to GPUs 1 and 2 since each GPU uses a separate PCIe port.
The controllers act as intermediate nodes that direct communication in the network. A controller is connected to two or more buses and forwards data that passes through it. Due to architectural limitations, controllers may limit the transfer rate of data that passes through them. Specifically, the rate may be limited when the controller transfers data in multiple directions at the same time, such as during bi-directional data transfers.

In some systems, data transfers between certain pairs of endpoints need to be staged in another memory. For example, in some multi-GPU compute nodes data transfers, between two GPUs that reside in different PCIe communication domains require staging in CPU memory. If different DMA controllers are used for the two routes (e.g., GPU → CPU and CPU → GPU), the two data transfers can overlap by using both DMA controllers at the same time.

Since some factors that limit the transfer rate are undocumented, we differentiate between theoretical and effective bandwidth. Theoretical bandwidth is the maximum rate at which data can be transferred over a communications network according to hardware specification. Effective bandwidth is the maximum measured rate, which is typically lower than the theoretical bandwidth. We use the effective bandwidth for the edge weights instead of theoretical; for worst-case analysis, a lower bound on the effective bandwidth is used. Another relevant term is aggregate bandwidth, which is the total rate of data transfer over a number of routes.

**Implications on task scheduling** Note that although data transfer speeds may change in time, time-dependent scheduling is not applicable to scheduling data transfers on the interconnect, because time-dependent scheduling assumes that the execution time of a job depends only on the time it started executing, while the data transfer speed also depends on which data transfers were active at that time.
6.2.2 Problem Statement

For an RTSPS with a heterogeneous interconnect whose data transfer requirements are given by a set of messages $M = \{m_1, m_2, \ldots, m_n\}$, find a valid data transfer schedule.

6.3 Data Transfer Scheduling Method: Overview

The problem of scheduling periodic data transfers on the interconnect has a lot in common with the problem of scheduling periodic tasks on a uniprocessor, for which there exist optimal scheduling algorithms and efficient schedule validity tests (see Section 2.2). In fact, in hard real-time systems a similar model of the interconnect is conventionally used, where data transfer requests are handled by a shared bus that serves only one request at a time. However, in modern compute nodes, transferring data on multiple routes in parallel is essential to make effective use of the interconnect, so executing data transfers one at a time would be inefficient. To address this problem, we developed a scheduling method that executes data transfers on different routes in parallel. Executing data transfers in parallel makes the scheduling problem more difficult, because concurrent data transfers may influence each other’s transfer times due to contention on the buses. The transfer rates depend on the interconnect topology and changes in the load on the buses due to events such as the start of a new data transfer or completion of a running data transfer.

Our method implements a compromise between scheduling complexity and execution efficiency. At its core, it uses a new execution model for data transfers in real-time systems where a basic operation includes a number of data blocks that need to be transferred and possibly their subsequent compute kernels; we call this a batch operation. Adding compute kernels is useful for scheduling computations and data transfers together, but we did not use it for scheduling periodic data transfers. High utilization of the interconnect can be achieved by transferring data blocks on different routes in one batch. Batch operations are executed one at a time and use the resource exclusively. Thus, the execution time of a batch operation depends only on the operation it contains and their execution schedule.

In Section 6.4 we describe the Batch method, a new method that optimizes the total execution time of a batch using a model of the interconnect and information about kernel execution times, and accurately predicts the execution time of such an operation offline. Heterogeneous interconnects combine several communication technologies that have unique features and undocumented characteristics, which makes it difficult to predict the data rates of each data transfer. In Section 6.4.3 we present an algorithm for computing the data transfer rates of multiple concurrent data transfers using a topology graph of the network that can be generated automatically using a set of benchmarks.

By default, the bandwidth of a bus is distributed fairly between data transfers that traverse it. However, the total transfer time of a batch can be optimized by controlling the distribution of bus bandwidths. In Section 6.5 we present an optimal method for controlling the bandwidth distribution at an intersection point of two communication technologies. This method can speed
up data transfer by up to 14.6%. Controlling the bandwidth distribution is supported by many platforms at the hardware level, but it is not revealed to the user and is currently not being used for optimization. Our study shows initial results in this is research direction.

In order to make effective use of the new execution model for periodic data transfers, it is necessary to combine individual data block transfer operations into batch operations. We define the term *batch message* – a periodic batch data transfer task – and present a new method that transforms the original message set into a set of batch messages.

**Definition 10.** 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, hence, an implicit-deadline periodic task characterized by the tuple $(a,c,p)$, where $c = WCET((x_k, src_k, dst_k)_{k=1}^n)$ is the worst-case total transfer time of the data blocks in the set. Note that every message is also a batch message.

The data blocks to be transferred are released at different time points and with different deadlines, so they cannot be simply merged. It is necessary to group them into batches and schedule the batch operations in a way that yields high utilization but meets timing constraints. Our algorithm starts with the initial set of messages and iteratively combines batch messages from the set in a way that maximizes the gain in efficiency (due to higher aggregate bandwidth), until no further efficiency gain is expected (Section 6.6).

After the message set is transformed into a batch message set, the batch messages are scheduled using the non-preemptive EDF algorithm; thus, the schedule can be checked using an existing validity test. In our experiments, the scheduler improved the overall performance of two realistic real-time systems by factors of between 31% and 74% (Section 6.7).

Our conclusions for this chapter are summarized in Section 6.8 and selected MATLAB implementations of the algorithms discussed in this chapter appear in Section 6.9. A data flow diagram of the method is depicted in Figure 6.5.

### 6.4 Batch Method for Data Transfer and Computations

In order to allow better utilization of the communication buses (and so, the entire system) while maintaining the ability to bound the execution time, we introduce a new run-time interface to the application (API). The new interface shifts from asynchronous command streams, as defined by CUDA and OpenCL, to batch operations that “bind” segments of multiple command streams together. It allows the system to optimize the execution, taking into account all the resources needed by the batch, and, in particular, to move resources from one stream to another in order to minimize the total execution time. From the point of view of the programmer, batch operations are executed atomically in issue order, but internally the run-time schedules the command stream segments concurrently, aiming to minimize the execution time.

Using the new interface, we will show that the run-time can provide two main services to the application: better resource utilization and accurate prediction of execution times for future
Figure 6.5: Data flow diagram of the data transfer scheduling method
batch operations. Such information can be used by the application to better schedule the batch operations in a way that meets system requirements (such as deadlines). It can also be used to dynamically control the execution, hence reducing the gap between the worst-case and actual execution times, and further improving the utilization of the system and its power consumption. Our new model is also compatible with the task model of classic scheduling algorithms, such as earliest-deadline-first (EDF) and rate-monotonic (RM).

6.4.1 Sharing Resources in Real-time Systems

The common methods for resource sharing in real-time systems include resource partitioning and time division. The resource partitioning method calls for the resource to be divided into a number of smaller portions, and to assign each of them to a task. For example, if the bandwidth of a bus is 8 GB/s, it can be distributed among four tasks so that each task is assigned 2 GB/s. The time division method calls for the time domain to be split into time slots that will be assigned to the tasks according to a given priority scheme. For example, the 8 GB/s bus can be assigned for a period of time to task A and later to task B. Each of these methods allows the system to make the communication time deterministic at the expense of underutilization of resources such as the bus bandwidth, because a resource that has been assigned to a task but was not fully utilized will not be used by the other tasks. Underutilization of the resources could increase power consumption and prevent the system from meeting real-time deadlines. For efficient utilization of shared resources in real-time systems, the resource-to-task assignment needs to be adjustable in both time and bandwidth.

The current interfaces for CUDA and OpenCL do not provide mechanisms for real-time execution. The interfaces do not define methods to specify deadlines for commands or command streams, or methods for predicting their execution times. Applications that schedule the program execution in heterogeneous architectures usually do a worst-case static analysis of the execution patterns in order to guarantee that the deadlines are met. The analysis is based on static resource sharing methods, such as static resource allocation or time division, and assumes that the worst case execution time (WCET) is known ahead of time for each job (computation or data transfer). Such a system can be suboptimal in terms of power and utilization, but also may cause the system to miss deadlines, as the following example indicates.

Consider a real-time production inspection system that has a CPU and four GPUs, as illustrated in Figure 6.6. Every period of 50 ms, four images arrive to main memory, are copied to GPU memory, scanned for defects, and, for each image, a boolean value is returned to main memory. This value denotes whether defects were found. The image sizes are 32 MB, 128 MB, 128 MB, and 32 MB, and they are processed on GPUs 1-4, respectively, one image per GPU. The system is required to produce the result in main memory within 50 ms, i.e., before the next period starts. The throughput of a GPU for scanning an image is 10 GB/s (value chosen arbitrarily). The result transfer time is in the order of microseconds; for simplicity, we ignore it in the following time analysis.

Using the resource allocation method (bandwidth allocation) for sharing resources, each
task is statically assigned a portion of the communication bandwidth. In this system, the bandwidth bottleneck (to all four GPUs) is 8 GB/s on the CPU-IOH link. Assuming equal bandwidth distribution, each stream is assigned 2 GB/s. Figure 6.7a shows a timeline of the expected execution for this method. The processing time is computed as follows:

\[ t_{\text{bandw}} = \frac{128 \text{ MB}}{2 \text{ GB/s}} + \frac{128 \text{ MB}}{10 \text{ GB/s}} = 76.8 \text{ ms} . \]

The resulting time is longer than the 50 ms latency bound, so this approach does not guarantee that the latency constraints are satisfied.

Using time division, the full bandwidth is allocated to tasks for periods of time. The GPUs are allocated separately; hence, kernel executions may overlap. The order in which the system allocates bandwidth to the images is not strictly defined by the method; Figure 6.7b shows a timeline for the scenario with the shortest expected processing time. This time exceeds the latency bound. The method does not utilize the bandwidth of the 8 GB/s CPU-IOH bus, but only of a 6 GB/s bus.

Note that the inability of the system to move allocated but unused resources from one stream to another caused the system to miss the deadline. In the next section we present the batch allocation method, which aims to ease this problem.

### 6.4.2 Interface

We propose the use of the following API for batch operations:
batchOp(dsts, srcs, sizes, kernels, n)

h = batchOpAsync(dsts, srcs, sizes, kernels, n)

b = batchOpQuery(h)
batchOpSynchronize(h)

t = predictBatchLatency(dsts, srcs, sizes, kernels, n)

Listing 6.1: API for batch operations

The API has three types of functions:

• Execution - `batchOp()` and `batchOpAsync()` execute a batch operation that contains `n` command streams. The function `batchOp()` is synchronous, while `batchOpAsync()` is asynchronous and returns a handler object `h` that the application uses to monitor the execution progress. Each command stream is either a data-block transfer, a kernel call, or a data-block transfer to a GPU followed by a kernel call that uses the transferred data on the target GPU.

• Latency prediction - `predictBatchLatency()` returns an execution time prediction for a batch operation. Here, the sources and destinations represent the memory modules, rather than specific addresses, such as GPU0 or MEM0 (local to CPU0).

• Progress monitoring - `batchOpQuery(h)` returns a boolean value indicating the completion status of an asynchronous batch operation, while `batchOpSynchronize(h)` blocks until the operation completes.

Using the API, the application can schedule data transfer operations and kernel calls, providing the run-time with the necessary information to use the interconnect and GPUs efficiently. In this work, we use a restricted model of a command stream: a stream is composed of a data transfer operation followed by a kernel call (both optional). This model does not limit the scheduling options for the application, as might seem to be the case. We discuss this issue in greater detail in Section 6.4.6.

6.4.3 Implementation

This section describes the implementation of the batch method for data transfer and kernel execution. The latency prediction algorithm is presented here and described in detail in the next section.

Execution

Calls to `batchOp()` and `batchOpAsync()` are processed in FIFO order. The execution algorithm schedules the command streams to execute in parallel with the goal of minimizing the execution time. For data transfers, it configures multiple DMA controllers to concurrently transfer data blocks that belong to different streams. Execution of the component operations in parallel is subject to the following constraints:
1. **Order dependencies.** Command streams define execution order between data transfers and kernels in them.

2. **Compute unit sharing.** Kernel executions on the same compute unit must not overlap.

3. **DMA controller sharing.** Data transfers that use the same DMA controller must not overlap.

4. **Staging in CPU memory.** In cases where a data transfer between two GPUs requires staging in CPU memory, the GPU→CPU transfer must precede the CPU→GPU transfer.

**Runtime Scheduling and Latency Prediction**

We propose a heuristic algorithm that reduces bandwidth contention by overlapping data transfers with kernel execution and optimizes the aggregate bandwidth; a MATLAB implementation of the algorithm is shown in Section 6.9.1. This algorithm also computes the total execution time of the operation; i.e., it predicts the execution latency. Due to order dependencies, it is not enough to optimize the schedule of the data transfers alone, because kernels may unnecessarily prolong execution if scheduled too late.

To improve the overlap between kernel executions and data transfers, the algorithm aligns the completion times of the command streams (Figure 6.8); namely, it computes when to start executing each command stream, such that the execution of all streams completes at the same time. Note how this approach reduces overlap and bandwidth contention between the data transfers (in comparison to Figure 6.7a, for example), thus allowing them to complete faster. To improve the aggregate bandwidth, the algorithm scans through various combinations of data transfers that can be executed in parallel on different DMA controllers to find the combination that yields the highest aggregate bandwidth.

The event times are computed iteratively by simulating the execution. Since our execution method aligns the kernel completion times, the event times are computed from the batch completion time backwards. For simplicity of discussion, we consider the moment when the batch execution completes as $t = 0$, and count the time backwards.

Computing the start and completion times of kernels is trivial since they all start at time zero (on the reversed timeline) and execute in parallel for fixed amounts of time, given by the user. For data transfers, such computation is more complex, since it needs to take contention into account. The transfer rates of concurrent data transfers do not change as long as no transfer starts or completes, so they are constant between any two events.
The ability to predict execution times can be used not only for scheduling the program’s execution, but also for monitoring its progress and for debugging. Next, we present an algorithm for computing the transfer rates of concurrent data transfers.

Model and Algorithm for Bandwidth Distribution in Concurrent Data Transfers

We represent the interconnect topology as a weighted directed graph \( G = (V, E, b) \), where a vertex \( v \in V \) represents a component of the network, such as main memory, a CPU, a GPU, or a routing component (controller, I/O hub, switch, etc.). A directed edge \( e_{ij} \in E \) represents a unidirectional link from component \( n_i \) to component \( n_j \), and the positive weight \( b(e_{ij}) \) represents its bandwidth.

The topology graph can be used to detect the bandwidth bottleneck along a route between two components and across multiple routes. For example, Figure 6.10 illustrates the network topology graph of a Tyan FT72B7015 system with four GPUs. In this graph, the M0→GPU0 and M0→GPU1 routes both traverse the CPU0→IOH0 edge; hence the aggregate bandwidth on these two routes is 9.6 GB/s. The aggregate bandwidth for given source-destination pairs can be computed by solving the multi-source multi-sink maximum flow problem on the graph. The rate of a data transfer is limited by the minimum edge weight \( b(e) \) on its route in the graph. It may further be limited by concurrent data transfers that consume some of the bandwidth of the buses on its route. Section 6.9.2 shows a MATLAB representation of a system with a similar topology.

PCI Express controllers apply an arbitration mechanism to resolve contention between data transfers. The default arbitration mechanism sends data packets waiting on the input ports to the output port in round-robin fashion, which effectively results in fair bandwidth distribution between the data transfers. However, the bandwidth is not necessarily distributed equally. For example, consider three concurrent data transfers: M0→GPU0, GPU1→GPU0, and M0→GPU2. The bandwidth of the bus closest to GPU0 is equally distributed between the first two transfers, 4 GB/s each. Therefore, the 9.6 GB/s bandwidth of the CPU-IOH bus is not equally distributed between M0→GPU0 and M0→GPU2, but the first gets 4 GB/s and the second the remaining 5.6 GB/s.

The edge weights may need to be further lowered in order to compensate for low-level data packet delays that result from chipset limitations. When data is transmitted in both directions, the aggregate bandwidth may be lower than the sum of the unidirectional bandwidths (due to chipset limitations). We suggest representing a bus with reduced aggregate bandwidth by adding auxiliary nodes and edges to the graph, as illustrated in Figure 6.9. Let \( A \) and \( B \) be two components connected by a bus, and let \( x, y, \) and \( z \) be the bandwidth of \( A\rightarrow B, B\rightarrow A \) and \( A\leftarrow B \), respectively. We add two auxiliary nodes and connect them with weighted edges as shown in the figure. One can see that the bi-directional throughput is limited by \( z \) because the communication in both directions passes through the middle edge, and the throughput on the \( A\rightarrow B \) and \( B\rightarrow A \) routes is limited by \( x \) and \( y \), respectively.

The algorithm shown in Listing 6.2 computes the expected data rates of each data transfer.

93
operation in a case where multiple data transfers are in progress. The algorithm iteratively finds the tightest bandwidth bottleneck for any data transfer and derives the final throughput for it. It then subtracts the allocated throughput from the edges on the route traversed by this data transfer and redistributes the remaining bandwidth between the remaining streams. A MATLAB implementation of the algorithm is shown in Section 6.9.3.

```matlab
// Input:
// G - topology graph (V,E,b)
// D - set of source->destination data transfer routes
// Output: Data rates T(d) for each data transfer d in D
// Auxiliary functions:
// n(e,D) - number of data transfers in D traversing e
// P(d) - the set of edges traversed by data transfer d
While D is not empty
    Remove from E all edges that are not in {P(d) | d in D}
    Find an edge e in E such that b(e)/n(e,D) is minimal
    For each d in D such that e in P(d)
        T(d) := b(e) - b(e)/n(e,D)
    For each e' in P(d)
        b(e) := b(e') - b(e)/n(e,D)
    Remove from D all d such that e in P(d)
Return T
```

Listing 6.2: Transfer rate computation for concurrent data transfers

Configuration of the DMA controller and later synchronization with it adds some latency; this time is typically short in comparison to the transmission time, but it can be significant for small data transfers. This time is added to the computed transfer time to result in the total execution time for the batch operation. We model this time as an expression of the form
\[ t_{\text{setup}} = c_1 + nc_2, \] where \( n \) is the number of transferred data blocks; \( c_1 \) and \( c_2 \) can be determined experimentally. (A similar model was also used in [Bal10].)

### 6.4.4 Performance of the Batch Method for the Example in Section 6.4.1

Our example in Section 6.4.1 indicates that using the existing resource sharing mechanisms may lead to sub-optimal utilization of the shared resources and to a failure of the system (deadline miss). In this section, we return to this example and show that our method solves these problems for that case.

The batch method re-allocates the bandwidth during the execution in a predictable manner. Thus, the application is able to utilize the full 8 GB/s bandwidth, while still being able to validate that the execution completes within the given latency constraints. The application binds the CPU→GPU data block transfers and kernel calls into a batch operation, followed by another batch operation for the return values. The run-time executes the command streams concurrently, aiming for a minimum total execution time. Figure 6.8 shows a timeline of the expected execution of a batch operation consisting of the data transfers and kernel calls for the four images. Due to the higher bandwidth, the operation completes after 44.8 ms, which is consistent with the given latency constraints. Thus the schedule is valid.

The example shows that our proposed method achieves shorter execution times than the existing methods by using the interconnect more efficiently.

### 6.4.5 Evaluation

To evaluate the batch method, we consider two applications running on two multi-GPU systems. We compare the batch method with two other bandwidth distribution methods: bandwidth allocation (resource partitioning) and time division.

For each system, we provide the system specification, find and analyze the effective bandwidth, and compare the performance using each of the methods in test case applications.

#### Baseline Configurations and Their Effective Bandwidth

In this section we describe two existing systems and use the techniques described in Section 6.4.3 to analyze their effective bandwidth.

**Nehalem multi-GPU system**

This system is a Tyan FT72B7015 server featuring the following components:

- Two 4-core Intel Xeon 5620 CPUs at 2.4 GHz, based on the Nehalem micro-architecture.
- An Intel 5520/ICH10R chipset with a QPI processor interconnect at 4.8 GT/s (9.6 GB/s) and two I/O hubs, each with two PCIe 2.0 ports.
- A total of 24 GB RAM in two modules.
- Four NVIDIA Tesla C2050 GPUs, each with 3 GB of GDDR5 memory.

---

95
The system runs Ubuntu 10.04 x64 Linux with CUDA SDK 5.0.

**Finding and analyzing the effective bandwidth.** The batch method uses the effective aggregate bandwidth to compute the completion times of data transfers. In Section 6.4.3, we presented a method for computing the aggregate bandwidth from a topology graph with edge weights that represent bus bandwidth. We obtained the edge weights using a series of benchmarks. To find the effective bandwidth of a bus, we transferred 256 MiB ($2^{28}$ bytes) data blocks between CPU0 and the GPUs in host-to-device (H2D), device-to-host (D2H), and bi-directional (Bi) modes on all the routes that traverse this bus, and measured the aggregate throughput for each direction (Figure 6.13a). Using these measurements, we determined the effective bus bandwidths in Figure 6.12a. For simplicity, we did not include the bi-directional bandwidth measurements in the figure.

The results show that the bus bandwidth is asymmetric; the effective H2D PCIe bandwidth is between 25% and 50% higher than the D2H bandwidth. We also see that the effective bandwidth to the remote GPUs is 20% lower. Since the QPI bus has higher bandwidth than the PCIe bus, this indicates that the latency of the extra hop over QPI translates into throughput degradation. The H2D aggregate bandwidth scales up for two GPUs by 57% for local GPUs (saturates the QPI bus), and by 33% for remote GPUs. In contrast, the D2H bandwidth for two GPUs does not scale. For four GPUs, the H2D bandwidth does not scale further, but the aggregate D2H bandwidth lines up with the bandwidth of the local GPUs. We ascribe the reduced scaling to chipset limitations, except where the QPI bus was saturated. Since the GPUs only have one DMA engine, they are not able to get more bandwidth from bi-directional transfer, yet we see that for the local GPUs a bi-directional transfer is faster than an H2D transfer followed by a D2H.

**Sandy Bridge multi-GPU system**

This system features the following components:

- Two 6-core Intel Xeon E5-2667 CPUs at 2.9 GHz based on the Sandy Bridge micro-architecture;
- An Intel C602 chipset with QPI at 8 GT/s (16 GB/s) and two PCIe 3.0 ports in each CPU;
- 64 GB of RAM in two modules;
- Two NVIDIA Tesla K10 cards, each with two GPU modules, connected by a PCIe switch, that include a GPU and 4 GB of GDDR5 memory.

The system runs Red Hat Enterprise 6.2 Linux with CUDA SDK 5.0.

**Finding and analyzing the effective bandwidth.** We determined the effective bus bandwidths shown in Figure 6.12b using a series of benchmarks, as described for the Nehalem system. Figure 6.13b shows the benchmark results for the Sandy Bridge system.
Figure 6.11: Measured CPU to GPU transfer latency

The Sandy Bridge system has higher bandwidth than the Nehalem system as it uses PCIe 3.0. Moreover, the bandwidth in Sandy Bridge is symmetric and is similar for local and remote GPUs, unlike in the Nehalem system. The aggregate bandwidth to the GPU pairs is only slightly higher than for the individual GPUs; this is expected, as the GPUs share a PCIe bus. However, moving further to four GPUs, the bandwidth scales almost perfectly. For a single GPU, bi-directional transfers increase the bandwidth by 32\%-72\% over uni-directional transfers, while for all four GPUs, the increase in bandwidth is only 10\%.

Figure 6.11 illustrates the latency of CPU $\rightarrow$ GPU data block transfers as a function of block size. The graph shows a minimum latency of 0.016 ms, which we identify as the setup time. The shape of the graph is consistent with our model, which divides the transfer time to constant setup time and transmission time that scales linearly with the block size.

**Application Based Performance Evaluation**

The evaluation in this section is based on two applications, one that represents a solution for a wide range of algorithms in areas such as signal processing and image processing, and another that is motivated by an application we developed as part of the Metro 450 consortium for inspection of 450 mm wafers.

**Domain decomposition.** In many scientific GPGPU applications that work on large datasets, the data domain is decomposed into parts that fit in GPU memory. The application we examine uses two GPUs, and follows a common pattern in multi-GPU programming:

1: split input data (3.69 GB) in two and scatter between GPUs
2: repeat N=100 times
3: for each GPU in parallel: compute boundary points (1 ms)
4: -- synchronization barrier --
5: for each GPU in parallel:
6: get boundary points from other GPU (295.2 MB)
7: compute internal points (23 ms)
Figure 6.12: Topology graph showing effective bandwidth in GB/s

Figure 6.13: Bus bandwidth benchmark. H=CPU (host), D=GPU (device)
In this application, the data transfer time is not negligible in comparison to the computation time, so increasing the data transfer efficiency could yield higher application performance. The execution times of the application using each of the bandwidth distribution methods for the data transfers are shown in Table 6.1. On the Sandy Bridge system, the batch method yielded 7.9 times shorter execution time than the bandwidth allocation method and twice shorter execution time than the time division method. On the Nehalem system, it was 28% faster than bandwidth allocation and 15% faster than time division.

The lower performance of the bandwidth allocation method can be attributed to presenting an abstraction of a less powerful interconnect to the scheduler. The method splits the bandwidth among the data transfer paths such that the bandwidth assigned to a path is guaranteed even under worst-case contention with other paths. As a result, in cases with less contention, the bandwidth of the “idle” paths is considered lost, while in practice it is used for data transfers on the “active” paths. For example, data transfers on the CPU→GPU1 and GPU2→GPU1 paths use the PCIe bus leading to GPU1, which is the bandwidth bottleneck for such transfers. When only one of the paths is being used, it gets the full bandwidth of the PCIe bus. However, the bandwidth allocation method assigns only half of the bandwidth of a PCIe bus to this path.

The time allocation method also does not fully utilize the available bandwidth, because doing so is impossible without executing several data transfers concurrently. In contrast, our results show that the batch method efficiently uses the available bandwidth.

**Wafer inspection.** In a real-time wafer inspection system, an image of a wafer arrives to main memory every 25 ms, and is sent to the GPUs for defect detection. The defects on the wafer are distributed non-uniformly, so in order to balance the kernel execution times, the image is split between GPUs 0-3 as follows: 40%, 10%, 10%, and 40%. The four GPUs process the image at 10 GB/s.

The system is required to produce a defect report in main memory every 60 ms, and its size equals that of the image. This requirement limits the resolution of wafer images processed by the system, yet the maximal resolution depends on the bandwidth distribution method. To calculate the maximum resolution for each of the methods, we take into account that the system uses the non-preemptive EDF scheduler, and check the validity of the schedule using the test described by Jeffay et al. [JSM91]. We use image size as a measure of the resolution.

Table 6.1 shows the maximum image sizes using each of the bandwidth distribution methods. The batch method achieves the highest average throughput by multiplexing the data transfers. On the Nehalem system, it supports 39% higher image resolution than that of the bandwidth allocation method because, when the short data transfers complete, it assigns some of the released bandwidth to the long transfers. It also achieves 37% higher image resolution than the time division method by utilizing the aggregate bandwidth of multiple routes.

On Sandy Bridge, it supports 22% higher image resolution than that of the bandwidth allo-
Table 6.1: Performance comparison of the different methods

<table>
<thead>
<tr>
<th></th>
<th>Domain decomposition (execution time)</th>
<th>Wafer inspection (image size – larger is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nehalem</td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td>Batch method</td>
<td>20.03 sec</td>
<td>3.18 sec</td>
</tr>
<tr>
<td>B/W allocation</td>
<td>25.59 sec</td>
<td>25.19 sec</td>
</tr>
<tr>
<td>Time division</td>
<td>23.10 sec</td>
<td>6.43 sec</td>
</tr>
</tbody>
</table>

cation method because, when the short data transfers complete, it assigns some of the released bandwidth to the long transfers. It also achieves 39% higher image resolution than the time division method by utilizing the aggregate bandwidth of multiple routes.

**Execution Time Prediction**

A fundamental part of the batch method is the execution timeline computation algorithm described in Section 6.4.3. This algorithm is used both for scheduling the execution of batch operations and predicting their latency. Hence, precision is essential. The precision of the algorithm is evaluated by:

1. Computing an execution timeline for selected batch operations.
2. Executing the operations and recording the execution time of each command stream.
3. Comparing the execution times for each command stream and for the batch in total.

We ran two experiments on each system. On Nehalem, we executed the following batch operations:

<table>
<thead>
<tr>
<th>Stream</th>
<th>Experiment 1 (Batch #1)</th>
<th>Experiment 2 (Batch #2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>256 MiB H2D (GPU0) + kernel (10 ms)</td>
<td>256 MiB D2H (GPU0)</td>
</tr>
<tr>
<td>S2</td>
<td>128 MiB H2D (GPU1) + kernel (10 ms)</td>
<td>128 MiB D2H (GPU1)</td>
</tr>
</tbody>
</table>

On Sandy Bridge, we executed similar experiments, but with four GPUs; the additional block sizes were 64 MiB and 32 MiB. Figure 6.14 displays the results of the computations and measurements side by side, and shows the prediction error of the total execution time. The prediction error was no more that 3.6% for all the tested cases. The table in that figure shows that the computed execution times of individual command streams are also similar to the recorded values.

We have shown that our algorithm correctly estimates command stream execution times for concurrent batch operations with different data transfer sizes. Thus, it can be used for efficiently scheduling the execution of command streams and predicting the execution time of batch operations.

**Throughput Computation vs. Measurement**

In Section 6.4.3, we presented a method for computing the aggregate throughput using a topology graph of the system. The topology graph represents bus bandwidth as edge weights, al-
Following the computation of end-to-end throughput by finding the bottleneck. However, the computed throughput may differ from the actual throughput due to chipset limitations. We evaluated the relative error of our algorithm by comparing the computed values with throughput measurements in a series of experiments with concurrent data transfers on the Sandy Bridge system.

For the experiments in Figure 6.13b the error was 0% for H2D and D2H, and 0%–(−4) % for Bi. The error in these experiments was low, because the edge weights in the topology graph were set to match these measurements as much as possible; the weights could not fully match the Bi bandwidth measurements, and an error of 4% was observed. In some other combinations of data transfers, however, especially including bi-directional data transfers, larger errors were observed.

The maximum error for H2D and D2H transfers was much smaller: 0.7% for H2D and 5.4% for D2H. The maximum error of 26% was observed in the case where data was transferred on the routes CPU0-GPU0 and CPU0-GPU3 in both directions. Using the technique for representing limited bi-directional bandwidth illustrated in Figure 6.9, we were able to improve the accuracy for these cases, but some limitations of the chipset could not be modeled.

### 6.4.6 Discussion

We showed in Section 6.4.5 that our method is efficient in executing workloads of data transfers and computations. The experimental results show that it achieves up to 7.9 times better results than the bandwidth allocation method (execution time) and up to 39% better results than the time division method (higher image resolution) in realistic applications. In Section 6.4.5, we also showed that the batch execution time is predictable and that our latency prediction algorithm computes a close estimation of that execution time. Sufficient safety margins on bandwidth and setup time should be used to compensate for prediction errors where strict latency
guarantees are required.

Our method executes batch operations that are a collection of command stream segments containing data block transfers and kernel calls. We have limited each command stream segment to contain only a data transfer followed by a kernel call (both optional), forcing the application to divide longer streams into multiple batch operations. Nonetheless, fine-grained scheduling can be achieved using two techniques: (1) dividing large data blocks into smaller chunks (also suggested in other works [KL11, BK12]), and (2) scheduling kernels in parallel with batch operations, without using the API. When using the second technique, the application needs to make sure that the kernel and the batch operation do not use the same GPU for computations. Using the techniques mentioned above, the application can make fine-grained scheduling decisions for data transfers and kernel calls. Our method uses execution time computations assuming no external factors that influence the execution time. Therefore, during the execution of a batch operation, the interconnect must be used exclusively by the execution algorithm. To guarantee this, all data transfers that use the interconnect must be called using the batch method’s interface. This requirement restricts the use of some memory access modes that are sometimes used in GPGPU applications, including direct read/write of remote memories between the CPUs and the GPUs, and peer memory access by GPUs. The application is allowed to run kernels in parallel with executing batch operations, provided that the target GPU is not used by the batch operation; a small execution time margin needs to be added to the time prediction in this case to compensate for the extra communication.

In Section 6.4.3 we presented an algorithm for computing throughput using a topology graph of the system with edge weights that represent bus bandwidth. The precision evaluation in Section 6.4.5 shows that this method is effective for data transfers from the CPU to the GPUs and in the opposite direction, but not for bi-directional data transfers. Our experiments show that for bi-directional transfers, the results are not consistent with topology graph analysis. In applications with workloads where the throughput computation algorithm is not efficient, we recommend using a lookup table that contains the throughput values for any combination of data transfer routes that may be used in the application. The throughput values can be found using benchmarks. Since the batch operation execution uses the interconnect exclusively, the benchmark results are expected to be consistent with the throughput values during the execution.

6.5 Controlling Bandwidth Distribution to Reduce Execution Times

The batch method achieves high throughput by sending data on multiple routes in the interconnect concurrently, thus saturating the bandwidth of buses shared by more than one route. By default, the bandwidth of a bus is distributed fairly between the data transfers that traverse it. Here, we present a technique for reducing the execution times of batch operations by manually controlling bandwidth distribution.

Consider the example shown in Figure 6.15. Two data blocks with different sizes are transferred in contention for the shared 8 GB/s bus. With the fair bandwidth distribution policy (default), each block gets 4 GB/s until the smaller block completes, and then the remainder of
the larger block is transferred at 6 GB/s. The data blocks can be transferred faster with different bandwidth distribution of the 8 GB/s bus. In the optimal case, the larger block would get 5.33 GB/s and the smaller would get 2.67 GB/s; and the two transfers would complete in 22.5 ms, which is 10% faster than using the default distribution. This example demonstrates that shorter execution times can be achieved by optimizing the bandwidth distribution.

PCI Express provides several mechanisms for controlling bandwidth distribution. One of these mechanisms is to set the priority field in data packets, so that during contention controllers give higher priority to packets with a higher priority field value. An application can control bandwidth distribution by assigning higher priority to packets that belong to specific data transfers; however, this requires an interface between the application and the DMA controller that initiates the packets. Currently, GPU vendors do not expose such an interface, so we conduct a theoretical analysis of the benefits of optimizing the bandwidth distribution.

### 6.5.1 Problem Definition

We consider the configuration illustrated in Figure 6.16. Two data blocks of sizes $D_1$ and $D_2$ are transferred on routes that pass through a common bus (shared) and then split to different buses (private), starting at time $t = 0$. This configuration is relevant to many systems, because often the bottleneck is an intersection point of two bus types with different bandwidth. We denote the bandwidth of the shared bus by $W$, and the bandwidth of each of the private buses by $w$. Hereafter, we assume without loss of generality that $D_1 \leq D_2$.

For the case where $W \geq 2w$, each data block gets the full bandwidth of the private bus; hence there is no need for an alternative bandwidth distribution. Another special case is when $W \leq w$, where the average data rate equals $W$ regardless of the bandwidth distribution policy. Therefore, we further consider only cases where $w < W < 2w$.

To characterize the bandwidth distribution between the transfers, we define the bandwidth...
Figure 6.17: Example: influence of bandwidth distribution factor on transfer time.
Parameters: data block sizes $D1 = 1.5$, $D2 = 2$ and bus bandwidths $w = 1$, $W = 1.5$.

distribution factor denoted by $b$ as the fraction of shared bus bandwidth that goes to the first data transfer. Thus, during contention, block $D_1$ is transferred with data rate $bW$ and block $D_2$ with data rate $(1 - b)W$. Since the data rate of a block cannot exceed the bandwidth of the private bus, the bandwidth distribution factor is limited as follows:

$$W - w \leq bW \leq w .$$

(6.1)

Figure 6.17 illustrates the influence of the bandwidth distribution factor on the total transfer time for the setup [$D1 = 1.5$, $D2 = 2$, $w = 1$, $W = 1.5$].

The following problem is considered:

Problem 11. Given setup parameters $D_1$, $D_2$, $W$, and $w$, find the bandwidth distribution factor $b$ that minimizes the total transfer time.

6.5.2 Formula for the Optimal Bandwidth Distribution Factor

In this section, we present a formula for the optimal bandwidth distribution factor. We start by developing the formula for the total transfer time as a function of $b$.

There are two periods during the data transfer: contention and remainder. The contention ends as soon as (at least) one of the data transfers completes, i.e., at the earliest of $\frac{D_1}{bW}$ and $\frac{D_2}{(1-b)W}$, and the remaining time equals the time it takes to transfer the remaining data with speed $w$. We denote the total transfer time, contention time, and remainder time by $T$, $T_c$, and $T_r$ ($T = T_c + T_r$); the formulas for their computation are as follows:

$$T_c = \min \left( \frac{D_1}{bW}, \frac{D_2}{(1-b)W} \right)$$

(6.2)

$$T_r = \frac{D_1 + D_2 - T_c \cdot W}{w}$$

(6.3)

$$T = \frac{D_1 + D_2}{w} - T_c \cdot \left( \frac{W}{w} - 1 \right) .$$

(6.4)

The aggregate throughput during contention is higher than during the remainder; hence the total transfer time will be shortest when the contention time is longest. The formula for $T$ indeed
shows that $T$ is at its minimum when $T_c$ is at its maximum, because the expression $(W_w - 1)$ is always positive. We now find the value of $b$ that maximizes $T_c$. $T_c$ is the minimum of two monotonic expressions: $\frac{D_1}{bW}$ and $\frac{D_2}{(1-b)W}$; hence it reaches its maximum when their values are closest (see Figure 6.18). Since the first expression is decreasing and the second is increasing, $T_c$ is at its maximum at either an intersection point or the minimum value of $b$, i.e., $\frac{W-w}{W}$. The transfer time will clearly be shortest when the two data transfers complete at the same time, if possible, or when $D2$ gets the maximum rate (and still completes after $D1$). We show this analytically. The two expressions intersect for the following value of $b$:

$$\frac{D_1}{b_x W} = \frac{D_2}{(1-b_x)W} \Rightarrow b_x = \frac{D_1}{D_1 + D_2}.$$  

If $b_x$ comes before the range of valid values for $b$, then the smallest valid value, $b_{\text{min}} = \frac{W-w}{W}$, results in the shortest execution time. Let $b_{\text{opt}}$ be the bandwidth distribution factor that results in the shortest execution time. Since $b_{\text{opt}}$ has passed the intersection point ($b_{\text{opt}} = b_{\text{min}} > b_x$), the first expression is smaller than the second. It follows that for $b = \frac{W-w}{W}$, $T_c = \min \left( \frac{D_1}{bW}, \frac{D_2}{(1-b)W} \right) = \frac{D_1}{bW} = \frac{D_1}{W-w}$. On the other hand, if $b_{\text{min}} \leq b_x$, then the execution time is shortest at the intersection point $b_{\text{opt}} = b_x = \frac{D_1}{D_1+D_2}$. In summary, the optimal bandwidth distribution factor is computed as follows:

$$b_{\text{opt}} = \begin{cases} \frac{W-w}{W}, & W-w > \frac{D_1}{D_1+D_2} \\ \frac{D_1}{D_1+D_2}, & \text{otherwise} \end{cases}$$  \hspace{1cm} (6.5)

We substitute $b = b_{\text{opt}}$ into the formula for contention time to derive the maximum contention time:

$$T_{c}^{\text{max}} = \min \left( \frac{D_1 + D_2}{W}, \frac{D_1}{W-w} \right).$$

Next, we substitute $T_{c}^{\text{max}} = T_c$ into the formula for $T$ and use the rule $\min(x, y) = -\max(-x, -y)$; thus, we arrive at the final form of the formula for minimum total transfer time:

$$T_{\text{min}}^{\text{max}} = \max \left( \frac{D_1 + D_2}{W}, \frac{D_2}{w} \right). \hspace{1cm} (6.6)$$

### 6.5.3 Evaluation of Bandwidth Distribution Method

In this section, we evaluate whether bandwidth distribution optimization yields better transfer time than the default fair distribution. In the previous section, we showed that longer contention time results in shorter total transfer time and derived the formula for the transfer time as a
The bandwidth distribution factor in fair distribution is $b_{fair} = 0.5$, yielding the following total transfer time:

$$T_{fair} = 2D_1 + \frac{D_2 - D_1}{w}$$ \hspace{1cm} (6.7)

The speedup achieved using the optimal bandwidth distribution instead is given by the following expression (after simplification):

$$X = \frac{T_{fair} - T_{min}}{T_{fair}} = \min \left( \frac{(D_2 - D_1) \cdot (W - w)}{2D_1 w + (D_2 - D_1)W}, \frac{2w - W}{2D_1 w + (D_2 - D_1)W} \right).$$ \hspace{1cm} (6.8)

Since the speedup depends on the problem parameters, an initial estimate of the benefit from bandwidth distribution can be obtained by finding the ratio between $D_1$ and $D_2$ that gives the highest speedup. For convenience, we denote the two expressions inside the min function in the formula for $X$ by (1) and (2).

We observe that, with respect to $D_2$, (1) is monotonically increasing and (2) is monotonically decreasing; thus, the minimum of (1) and (2) is greatest at the point of their intersection, if it exists. We find the value of $D_2$ in this point:

$$D_2^{max} = \frac{D_1 \cdot W}{W - w}.$$

We observe that for this value of $D_2$, the optimal bandwidth distribution factor is $W/w$ (Equation 6.5) and the two data block transfers complete at the same time. The value of $X$ in this
The analysis shows that the maximum speedup that can be achieved by optimizing the bandwidth distribution in the configuration considered in this section is 14.6%. This speedup is achieved for $\frac{W_w}{w} = \frac{p}{2}$ and $\frac{D_2}{D_1} = \frac{w}{W_w}$, using a bandwidth distribution factor 0.2929.

6.5.4 Conclusions

By default, bus bandwidth in modern compute node interconnects is distributed fairly between the data transfers that pass through it. We have shown that using a custom bandwidth distribution policy makes it possible to reduce the total transfer time of data blocks on different paths. This approach could speed up the batch method and yield higher performance of the stream processing system due to better utilization of the interconnect. Mechanisms for controlling bandwidth distribution in buses are supported by communication standards such as PCI Express, but some chipset and DMA controller manufacturers do not implement or expose this functionality, which is available only in low-level interfaces. Making these capabilities available at the level of task schedulers could enable more efficient use of interconnect resources.

We examined the problem of total transfer time minimization for two data blocks that pass through a shared bus and continue through different buses with smaller bandwidth. This scenario is common in high-performance compute nodes, where bandwidth bottlenecks often occur at a connection point between communication domains (such as QPI and PCIe). We developed a formula for computing the bus bandwidth distribution factor that minimizes the total transfer time, and also developed a formula to compute the expected transfer time in this case. The effectiveness of using an optimal bandwidth distribution was evaluated by computing the speedup in comparison to the default fair bandwidth distribution. We developed a formula for the speedup factor, which depends on the ratio between data block sizes and the ratio between bus band-
Having an upper bound evaluation of the expected speedup is useful to estimate whether the effort required to make use of these capabilities is worthwhile. We found that the speedup is limited to 14.6%.

### 6.6 Task Set Optimization

In this section, we will use the term message for brevity when referring to any batch message.

#### 6.6.1 Data Transfers to a Set of Batch Messages

As discussed in Section 6.4.3, 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 batch messages, where each batch message describes a data transfer operation, the *elementary* message set. Note that other batch 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 12.** An elementary message set $E = \{ m_i : (x_i, src_i, dst_i, p_i, a_i) \}_{i=1}^n$ describes the data transfer requirements of the system. For each data transfer operation there is one corresponding batch message in $E$ that represents its data transfer requirements 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. By construction, a schedule of an elementary message set is valid if and only if it satisfies the system’s requirements and timing 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 they are compatible with the original timing constraints. For example, the following two message sets describe the same resource requirement:

$$E = \left\{ \left(10\text{MB}, \text{CPU}0, \text{GPU}1, 15\text{ms}, 0\right)\right. , \left(10\text{MB}, \text{GPU}1, \text{CPU}0, 15\text{ms}, 0\right), \left(15\text{MB}, \text{CPU}0, \text{GPU}2, 20\text{ms}, 0\right) \right\}$$

$$M = \left\{ \left(\left\{ \left(10\text{MB}, \text{CPU}0, \text{GPU}1\right), \left(10\text{MB}, \text{GPU}1, \text{CPU}0\right) \right\}, 15\text{ms}, 0\right), \left(15\text{MB}, \text{CPU}0, \text{GPU}2, 20\text{ms}, 0\right) \right\} .$$

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 $E$. 
6.6.2 Message Set Transformation Algorithm

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.

Listing 6.3 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. Hence, it increases the aggregate bandwidth and shortens 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 ($\sum u_i$). The sum utilization is the fraction of time that the resource is required over an infinite period of time.

```
function R = reduceUtilization(E)
    % E - Elementary message set
    % R - Optimized batch 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
```

Listing 6.3: Message set transformation algorithm

6.6.3 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 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.

**Synchronous Combining**

Consider the two messages illustrated in Figure 6.19a. The numbers in squares indicate the block sizes in MB and the vertical marks indicate their arrival times. The synchronous method adjusts the scheduling bounds of the message with the longer period \((m_1)\) to match the bounds of the other message \((m_2)\), and then combines the two messages into one. The bounds are adjusted by splitting \(m_1\)’s data block into smaller blocks and splitting the period respectively. As illustrated in Figure 6.19b, 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 13.** We define *synchronous combining* as an operation that receives two messages,

\[
m_1 = \{(x_k, src_k, dst_k)\}_{k=1}^{n_1}, 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

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

**Asynchronous Combining**

Figure 6.20a 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 6.20b). 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.

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 \( \overline{a_2} = a_2 - \left\lceil (a_2 - a_1)/p_2 \right\rceil p_2 \).

**Definition 14.** We define *asynchronous combining* as an operation that receives two messages,

\[
\begin{align*}
\mathcal{m}_1 &= (\{ (x_k, src_k, dst_k) \}_{k=1}^{n_1}, p_1, a_1) \\
\mathcal{m}_2 &= (\{ (x_k, src_k, dst_k) \}_{k=n_1+1}^{n_1+n_2}, p_2, a_2),
\end{align*}
\]

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

\[
\mathcal{m}' = \left( \{ (x_k/n, src_k, dst_k) \}_{k=1}^{n_1} \cup \{ (x_k, src_k, dst_k) \}_{k=n_1+1}^{n_1+n_2} \right)_{p_2, \overline{a_2}, \overline{a_2}},
\]

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

### 6.6.4 Message Set Scheduling and Validity Test

The optimized message set defines the execution and timing requirements of the batch data transfer jobs for the scheduler. The interconnect serves every batch job exclusively, so it is modeled as a uniprocessor. The batch messages are scheduled as non-preemptive implicit-deadline periodic tasks with specified release times; the preemptive model is not suitable in this case because (offline) preemption involves DMA configuration and synchronization overhead. Our scheduler uses the non-preemptive EDF algorithm, which is optimal for such tasks [GMR95].

Jeffay et al. showed that the problem of testing the validity of the schedule is NP-Hard [JSM91].
and presented a sufficient polynomial time validity test that is used by our scheduler [JAM88]. A MATLAB implementation of the validity test algorithm appears in Section 6.9.4.

6.7 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.

6.7.1 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 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. Then, we applied the algorithm in Listing 6.3 to find an 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.

6.7.2 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 [VSS11, VSSM12]. 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.

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 the system.

**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 6.21.

**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 7500 MB/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 are described in Figure 6.22a.
AES-CBC 128-bit encryption of multiple streams

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

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

(b) System 1

c) System 2

Figure 6.22: Throughput increase due to more efficient communication scheduling

Results

The two charts in Figure 6.22 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 average, and in System 2 the speedup was 31%.

The scheduler automatically detected that overlapping transfers in opposite directions for different GPUs gives the highest bandwidth; hence, it outperformed the baseline scheduler, which overlaps data transfers in the same direction. The measured speedups are in line with the respective bandwidth ratios. For example, in System 2, the aggregate bandwidth in opposite directions (11.6 GB/s) is 37% higher than the average bandwidth in the same direction (8.5 GB/s). Overlapping all four transfers was not possible since bi-directional data transfer is not supported by these GPUs.
6.8 Conclusions

In recent years, the compute power of CUs has grown 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 heterogeneous compute nodes focused mainly on the problem of scheduling the computation. In this chapter, we focused on the communication scheduling problem.

Current state-of-the-art resource sharing methods for real-time systems logically partition resources that can serve multiple jobs at the same time into independent resources with smaller and fixed capacity that work on one job at a time. This model, where concurrent jobs are executed in isolation, has limited efficiency for complex resources such as the system interconnect, where execution of a data transfer (a job) on some path $A \rightarrow B$ can decrease the available bandwidth (capacity) for concurrent data transfer on some intersecting path $C \rightarrow D$. The abstraction of isolation in such cases is commonly achieved by allocating resources in excess, such as assuming worst-case bus contention in the interconnect example. This chapter presented a new approach where the jobs can share the full capacity of the resource without rigid allocation, and predictability is achieved using a detailed performance model.

We presented a new method for scheduling the data transfer requirements of an RTSPS with multiple CUs. To address the tradeoff between scheduling complexity and resource utilization efficiency, the method employs a two-level scheduling hierarchy. The top level schedules compound data transfer jobs, i.e., batch operations, to execute one at a time, while the bottom level schedules individual data transfers within a batch operation. This hierarchy achieves both effective scheduling and timing analysis at the task level, along with efficient utilization of the interconnect at the level of individual data transfer operations.

The Batch method was introduced as an efficient implementation of the bottom level scheduler. The method makes use of a detailed model of the interconnect to minimize and predict the execution time of batch operations, which can include data transfers and kernel executions. It makes use of multiple DMA engines and scheduling order optimizations to saturate the available bandwidth and ultimately reduce the makespan. Since a batch operation uses the interconnect exclusively, the batch method can effectively simulate the distribution of data rates during execution using a performance model and accurately predict the execution time of the operation. This prediction is later used to describe the compatible job at the top level scheduler. The batch method achieved up to 7.9x better results than the bandwidth allocation method (execution time) and up to 39% better results than the time division method (higher image resolution) in realistic applications.

To further reduce the execution time of batch operations, we considered optimization of the bandwidth distribution on buses that limit the data transfer rates. The default bus bandwidth distribution policy in case of contention is fair distribution. We have shown that by optimizing the bandwidth distribution at a point of contention, the transfer time of two data blocks can be reduced by up to 14.6%, and we developed a method for optimal bandwidth distribution. Limited support in hardware and software make it difficult to utilize this capability because,
although low-level mechanisms to control the bandwidth distribution exist in some communication standards, such as PCIe packet priorities, many hardware components do not currently implement them. Nor is this capability exposed at higher level interfaces. Investigating the practical application of this method is left for future research.

Data transfers in a RTSPS can be described by a set of periodic data transfer tasks with deadlines (messages). We presented a comprehensive scheduling method for these tasks that makes efficient use of the interconnect. The method redefines the data transfer requirements as a set of periodic batch tasks (batch messages), schedules the batch tasks, and uses the batch method to schedule the data transfers within each batch job. The transformation from the original set of data transfer tasks to the set of batch tasks is performed using a new algorithm that combines periodic data transfer tasks into periodic batch tasks. The algorithm is capable of combining tasks even with different periods. The combined task is constructed such that its scheduling constraints are at least as strict as the original constraints. The algorithm iteratively aims to optimize the total utilization, which is the fraction of time the resource is required by the tasks. Combining the tasks can reduce their utilization by achieving higher bandwidth, but the overhead may be larger than the gain. The algorithm greedily combines pairs of tasks with the greatest utilization gain, until no further gain is achieved. The optimized task set is scheduled using non-preemptive EDF. Our experiments show that our scheduler improved the overall performance of two realistic real-time systems by factors of between 31% and 74%.
6.9 MATLAB Implementation

6.9.1 Batch Run-Time Scheduling Algorithm

```matlab
function [makespan, schedule] = batchmethod(batch)

% BATCHMETHOD Generates an execution schedule for a batch job and computes
% its makespan (execution time).
% Computes a schedule for executing a batch job that contains a number of
% execution streams. Each execution stream is composed of a data transfer
% followed by a kernel execution (both optional). The returned schedule
% specifies the start and end time of each operation. In addition, the
% function returns the makespan of the execution, i.e., the total
% execution time.
% The format of the input and output arguments is as follows:
% INPUT:
% batch = [ (dst1,src1,size1,kernel1),
%           (dst2,src2,size2,kernel2),
%           ...
%           (dstN,srcN,sizeN,kernelN)]
% OUTPUT:
% schedule = [ (copy begin, copy end, kernel begin, kernel end), %1
%               (copy begin, copy end, kernel begin, kernel end), %2
%               ...
%               (copy begin, copy end, kernel begin, kernel end), %N]
% makespan = a non-negative real number

global interconnect

if isempty(batch)
    makespan = 0;
    schedule = [];
    return;
end

% Since the stream completions are aligned, the event times are computed
% from completion to start. We define t=0 as the batch completion time
% and count the time from this time backwards, i.e., an event that
% happened l time unit before the completion time happens at t=l. The
% times are reversed to normal in the end, after computing the makespan.

% Variables and notations
n = size(batch,1); % Number of execution streams in batch

% Schedule table. Event times are on the reversed schedule.
% (kernel begin, kernel end, copy begin, copy end)
reverseSchedule = inf * ones(n,4); %zeros(n,4);

% Get list of DMA engines and re-enumerate as 1,2,3...
dmaIds = arrayfun(@(i) interconnect.getDmaId( ...
    batch(i,2),batch(i,1)), 1:size(batch,1));
```
uniqueDmas = unique(dmaIds);
dmaIds = arrayfun(@(x) find(uniqueDmas == x), dmaIds);
nDma = length(uniqueDmas);  % Number of DMA engines

%%% Schedule kernels
reverseSchedule = scheduleKernels(reverseSchedule, batch);

%%% Schedule data transfers

% find first time point (on the reversed schedule) where a data
% transfer can be scheduled

t = min(reverseSchedule(:,2));

dmaInUse = zeros(nDma,1);  % list of DMAs in use
bexecuting = false(n,1);  % bit vector for data transfers in execution
nComplete = 0;  % number of streams completed

while(nComplete < n)
    % find potential ready data transfers on idle DMAs

    ready = find(...
        reverseSchedule(:,2) <= t & reverseSchedule(:,4) > t & ...
      dmaInUse(dmaIds(:)) == 0 );

    % if not all transfers finished, but there are no data transfers
    % in progress and no new data transfers can be started then
    % fast-forward to the next data transfer release time
    if isempty(ready) && all(bexecuting == 0)
        t = min(reverseSchedule(reverseSchedule(:,2) > t,2));
        continue;
    end

    % schedule new data transfers on idle DMAs

    if ~isempty(ready)
        % Try all possible combinations of transfers on DMA engines
        % to find the combination that results in the highest
        % aggregate throughput

        [~,toScheduleFilter,~] = ...
        unique(strcat(batch(ready,1),batch(ready,2)));
        ready(~toScheduleFilter) = [];
        reschedDmas = unique(dmaIds(ready));
        dmaQueues = arrayfun(@(x) ready(dmaIds(ready) == x), ...
            reschedDmas, 'uniformoutput', false);

        % split to DMA queues and limit to one data transfer per route
        [~,-,toScheduleFilter,~] = ...
        unique(strcat(batch(ready,1),batch(ready,2)));
        ready(~toScheduleFilter) = [];
        reschedDmas = unique(dmaIds(ready));
        dmaQueues = arrayfun(@(x) ready(dmaIds(ready) == x), ...
            reschedDmas, 'uniformoutput', false);

        % generate permutations
        vQueueLen = arrayfun(@(x) length(x{:}), dmaQueues);
        permutations = 1:vQueueLen(1);
        for ii = 2:size(vQueueLen,1)
            permutations = combvec(permutations,1:vQueueLen(ii));
        end

end
% find permutation with highest throughput
maxThroughput = 0;
selectedTrans = [];
prevExecuting = find(bexecuting)';
for iPerm = 1:size(permutations,2)
    p = permutations(:,iPerm);
    newRun = arrayfun(@(x) dmaQueues{x}(p(x)), ... 
        1:size(dmaQueues,2))';
    allRun = [newRun; prevExecuting];
    T = computethroughput(allRun,batch);
    if sum(T) > maxThroughput
        maxThroughput = sum(T);
        selectedTrans = newRun;
    end
end

% update list of transfers in progress, DMAs in use, and
% start time for new transfers
bexecuting(selectedTrans) = 1;
dmaInUse(dmaIds(selectedTrans)) = 1;
reverseSchedule(selectedTrans,3) = t;
end

duration = next_event_time - t;

% update remaining data to transfer in all executing streams
EPS = 1e-10; % account for roundoff error
sizes = sizes - duration * throughput;
sizes(sizes < EPS) = 0;
for ii = 1:length(executing)
    batch{executing(ii),3} = sizes(ii);
end

% update state for completed data transfers
completed = executing(sizes <= 0);
reverseSchedule(completed,4) = next_event_time;
dmaInUse(dmaIds(completed)) = 0;
bexecuting(completed) = 0;
nComplete = nComplete + length(completed);

% update time variable
t = next_event_time;
end

%% Add setup time and reverse schedule back to normal
% makespan is the time difference between first and last events
setup = setupTime(batch);
makespan = t + setup;
% reverse schedule
schedule = makespan - reverseSchedule(:,end:-1:1);
end

function reverseSchedule = scheduleKernels(reverseSchedule, batch)
% update times of commands streams without kernels
reverseSchedule(strcmp(batch(:,4),''), 1:2) = 0;

% group kernels in per-GPU queues
uniqueKernelExecutingGpus = unique(batch(~strcmp(batch(:,4),''),1));
kernelQueues = arrayfun(@(x) find(strcmp(batch(:,1),x)), ...
uniqueKernelExecutingGpus, 'uniformoutput', false);

% schedule kernels from queues
nKerQueues = size(kernelQueues,1);
kQExecutionTimes = zeros(nKerQueues,1); % kernel completion times
for iKerQueue = 1:nKerQueues % kernel completion times
    t = 0;
    for iQueuePos = 1:size(kernelQueues(iKerQueue),1)
        iKernel = kernelQueues(iKerQueue)(iQueuePos);
        kerTime = kerneltime(batch(iKernel,4));
        reverseSchedule(iKernel,1) = t; % start
        reverseSchedule(iKernel,2) = t + kerTime; % stop
        t = t + kerTime;
    end
    kerQExecutionTimes(iKerQueue) = t;
end

function x = computethroughput(streams, batch)
global interconnect
if isempty(streams)
    x = [];
    return;
end
endpoints = [batch(streams,2), batch(streams,1)];
dmaSelected = 1:size(endpoints,1);
x = interconnect.bandwidthdistribution(endpoints(dmaSelected,:));

Listing 6.4: Batch run-time scheduling (MATLAB)
6.9.2 Interconnect Model

classdef InterconnectModel < handle
  properties
  SystemName;

  % Topology graph properties
  V; % vertices
  nV; % number of vertices
  E; % edges matrix 1 in (i,j) denotes edge i->j
  C; % edge capacities
  Paths; % paths between endpoints
  PathEdges; % traversed edges between endpoints (sparse bitmap)

  % System properties
  SetupTime; % DMA configuration time in ms
  SetupTimeErr; % Possible deviation in setup time in ms
  GpuDmaCount; % Number of DMA engines per GPU

  % Cache
  BwDistLookup; % cache for bw distribution lookups
end

methods
  function obj = InterconnectModel(name)
    if nargin > 0
      obj.SystemName = name;
    end
    switch lower(obj.SystemName)
      case 'sandybridge'
        obj.loadsandybridge();
      end
      % other models
  end

  function loadsandybridge(obj)
    obj.SystemName = 'Sandy_Bridge';

    % define graph
    obj.V = {'c0', 'c1', 'ioh0', 'iohl1', 'g0', 'g1', 'g2', 'g3'};
    obj.nV = length(obj.V);
    obj.E = zeros(obj.nV, obj.nV);
    obj.C = zeros(obj.nV, obj.nV);
    obj.Paths = cell(obj.nV, obj.nV);
    obj.PathEdges = cell(obj.nV, obj.nV);

    % set edges
    obj.setedge('c0','ioh0',12.5,12.1);
    obj.setedge('ioh0','g0',11.8,11.8);
    obj.setedge('ioh0','g1',11.7,11.8);

end
obj.setedge(‘c0’, ‘c1’, 12.0, 12.6);
obj.setedge(‘c1’, ‘ioh1’, 12.0, 12.6);
obj.setedge(‘ioh1’, ‘g2’, 11.7, 11.1);
obj.setedge(‘ioh1’, ‘g3’, 11.7, 11.0);

%% set paths
obj.setpath(‘c0’, ‘g0’, (‘c0’, ‘ioh0’, ‘g0’));
obj.setpath(‘c0’, ‘g1’, (‘c0’, ‘ioh0’, ‘g1’));
obj.setpath(‘c0’, ‘g2’, (‘c0’, ‘c1’, ‘ioh1’, ‘g2’));
obj.setpath(‘c0’, ‘g3’, (‘c0’, ‘c1’, ‘ioh1’, ‘g3’));

%% setup
obj.GpuDmaCount = 2;
obj.SetupTime = 0.010;
obj.SetupTimeErr = 0.003;
obj.BwDistLookup = {};
end

function setedge(obj, vFrom, vTo, c, cBack)
  % cBack is optional and if present means that the edge is
  % bi-directional with capacity cBack
  i = obj.c2i(vFrom);
  j = obj.c2i(vTo);
  obj.E(i,j) = 1;
  obj.C(i,j) = c;
  if nargin == 5
    obj.E(j,i) = 1;
    obj.C(j,i) = cBack;
  end
end

function setpath(obj, vFrom, vTo, path)
  i = obj.c2i(vFrom);
  j = obj.c2i(vTo);
  indexPath = cellfun(@(x) obj.c2i(x), path);
  obj.Paths(i,j) = {indexPath};
  obj.Paths(j,i) = {indexPath(end:-1:1)};
  edgesTo = zeros(size(obj.E));
  edgesFrom = zeros(size(obj.E));
  for ii = 1:length(indexPath)-1
    edgesTo(indexPath(ii),indexPath(ii+1)) = 1;
    edgesFrom(indexPath(ii+1),indexPath(ii)) = 1;
  end
  obj.PathEdges(i,j) = { sparse(edgesTo) };
  obj.PathEdges(j,i) = { sparse(edgesFrom) };
end

function iVertex = c2i(obj, v)
  iVertex = find(strcmp(v, obj.V));
end
Listing 6.5: Interconnect model (MATLAB)
6.9.3 Computation of Data Rates

```matlab
function T = bandwidthdistribution(obj, transferRoutes)
% BANDWIDTHDISTRIBUTION Computes the expected data rates of
% each data transfer operation in a case where multiple data
% transfers are in progress.
%
% INPUT:
% % dataStreams = [ src0 dst0; % endpoints of concurrent
% src1 dst1; % data transfers
% ...
% ];
%
% OUTPUT:
% % T = [ rate0; % expected data rate
% rate1;
% ...
% ];
%
% optimization: perform lookup in cache
hashkey = {transferRoutes{:}}; % hashkey example: c0c0g0g3
if isfield(obj.BwDistLookup,hashkey)
    T = obj.BwDistLookup.(hashkey);
    return;
end
%
% initialization
T = zeros(size(transferRoutes,1),1);
bMarkFinished = zeros(size(transferRoutes,1),1);
remainingC = obj.C; % remaining edge capacities (matrix)
%
% limit to one stream per DMA controller
dmaIds = arrayfun(@(i) obj.getDmaId( ...
    transferRoutes{i,1},transferRoutes{i,2}), ...
    1:size(transferRoutes,1));
uniqueDmaIds = unique(dmaIds);
notUsingDmaMask = ones(size(transferRoutes,1),1);
notUsingDmaMask(isUsingDma) = 0;
bMarkFinished = bMarkFinished | notUsingDmaMask;
%
% remove unused edges by marking used ones and removing
% unmarked
countUsedEdges = zeros(obj.nV, obj.nV);
for iRoute = 1:size(transferRoutes,1)
    if bMarkFinished(iRoute), continue, end
    src = obj.c2i(transferRoutes(iRoute,1));
    dst = obj.c2i(transferRoutes(iRoute,2));
```
countUsedEdges = countUsedEdges + obj.PathEdges(src,dst);
end

while sum(bMarkFinished) < size(transferRoutes,1)
    remainingC(countUsedEdges == 0) = 0;
end

% Find bottleneck edge, store rate for each data transfer
% that pass through it in ratePerRoute (fair distribution)
proportionalBandwidth = remainingC ./ countUsedEdges;
[ratePerRoute, I] = min(proportionalBandwidth(:));
[v1, v2] = ind2sub(size(proportionalBandwidth),I);

% update
% 1.set bandwidthPerStream as the rate of all the streams
% 2.for each such stream, remove their rate from all the
% edges that they traverse and decrease the edge count.
for iRoute = 1:size(transferRoutes,1)
    if bMarkFinished(iRoute), continue, end
    src = obj.c2i(transferRoutes{iRoute,1});
    dst = obj.c2i(transferRoutes{iRoute,2});
    pathEdges = obj.PathEdges(src,dst);
    isCrossingEdgeV1V2 = full(pathEdges(v1,v2));
    if isCrossingEdgeV1V2
        T(iRoute) = ratePerRoute;
        traversedPos = find(pathEdges);
        remainingC(traversedPos) = ... 
        remainingC(traversedPos) = ratePerRoute;
        countUsedEdges(traversedPos) = ...
        countUsedEdges(traversedPos) = 1;
        bMarkFinished(iRoute) = 1;
    end
end
end

% store results in cache for future queries
obj.BwDistLookup.(hashkey) = T;
end

Listing 6.6: Computation of data rates (MATLAB)
6.9.4 Non-Preemptive EDF Schedulability Test

function b = isNonPreemptiveEdfSchedulable(taskset)
    %isNonPreemptiveEdfSchedulable A non-preemptive EDF schedulability test for
    % periodic implicit-deadline task sets. A task set is a matrix of type:
    % [a1, c1, p1; a2, c2, p2; ...]; where a is the offset, c is the WCET and p
    % is the invocation period.
    % The schedulability test is described in "On optimal, non-preemptive
    % set is considered schedulable if two conditions are satisfied:
    % 1. utilization(messageSet) <= 1.0
    % 2. Let the messages be sorted in non-decreasing order by period.
    % forall i,L; 1<i<n, pl<L<pi; L >= ci + Sum[j=1..i-1](floor((L-1)/pj)*cj)

    n = size(taskset,1);
    c = taskset(:,2);
    p = taskset(:,3);

    % Condition 1
    u = sum(c./p);
    b = ( u <= 1.0 );
    if b == 0
        return
    end

    % Condition 2
    % Intuitively, f(L) (the DBF function) is the processor demand for the
    % interval [0, L-1] when all tasks are released at time zero. For all
    % L, pl < L < pn. We can compute f(L) in time O(pn) as follows.
    % Initialize an array of integers A of size Pn to zero. For each task
    % Tk, 1 <= k <= n, add Ck to location m of array A for all m that are
    % multiples of Pk. At the completion of this process the sum of the
    % first L locations of A will be f(L).
    %
    % sort messages by period non-decreasingly
    [p, IDX] = sort(p);
    c = c(IDX);

    % compute DBF function for 0 <= t <= pn
    STEP = p(1)/100;
    demandArr = zeros(length(0:STEP:p(end)),1);
    for iMsg = 1:n
        incPositions = floor((p(iMsg):p(iMsg):p(end)) / STEP);
        demandArr(incPositions) = demandArr(incPositions) + c(iMsg);
    end
    DBF = cumsum(demandArr);
    b = all((0:STEP:p(end)) >= DBF);
end

Listing 6.7: Non-preemptive EDF schedulability test (MATLAB)
Chapter 7

Extended Multiprocessor Model for Efficient Utilization of Parallel Resources

7.1  Introduction

In recent years, computer architecture has become more parallel and heterogeneous. Many systems now contain a variety of components, some of which have a complex internal structure, such as multicore CPUs, many-core accelerators (GPUs), distributed memories, processor interconnects, and high-speed communication networks. Scheduling such systems to make efficient use of the compute and memory resources is a challenging task that greatly affects system performance.

In scheduling, a parallel resource is commonly modeled as a set of independent processing units, each of which executes tasks one at a time [ERAL95]. This model has been extensively studied in the literature for various scheduling problems (see survey in [CPW99]). Many real parallel resources, however, cannot be decomposed into independent serial components because of the inherent dependency between the processing units, due to sharing a power or thermal envelope, for example. To conform to the model, lowered and guaranteed speeds are used and opportunities to achieve higher performance are not exposed to the scheduler.

In previous chapters we have shown how detailed performance models of GPUs and the system interconnect can be used for more efficient scheduling and higher performance in an RTSPS. While different in nature, a GPU and an interconnect have some similar performance properties; specifically, both resources serve multiple concurrent processing streams that affect each other’s progress rates. This property was analyzed in detail in previous chapters: in GPUs, the number of jobs executed concurrently on an SM influences the progress rate of each job and the aggregate progress rate (Section 4.1.2), and in the interconnect concurrent data transfers influence each other’s progress due to bandwidth contention, affecting the aggregate bandwidth.

This property is in fact common to many parallel resources in computer systems, also in-
cluding multicore CPUs. In applications that schedule compute jobs on multicore CPUs, each core is considered as an independent processing unit with a fixed processing speed. In practice, however, the processing speed of each core depends on the other cores in the CPU package. Core frequency scaling technologies such as Turbo Boost are one cause of this dependency; Turbo Boost opportunistically increases the cores’ operating frequency (speed) when the power, current, and temperature of the chip are below certain limits. Schedulers that model the cores’ performance using the base CPU frequency overlook up to 100% of additional frequency (e.g., in Core i7-4550U CPU). Effectively, higher frequencies are achieved when fewer cores are active, so the speed of each core depends on the state of the other cores (active or idle). For example, the maximum core frequencies for different numbers of active cores in a Core i7-820QM CPU are shown in Figure 7.1.

Turbo Boost has implications on the expected speedup and improvement in energy consumption that results from parallelizing sequential codes. Rewriting sequential codes as multithreaded is known to achieve speedup and reduced energy consumption on a given multicore system. However, writing and validating parallel codes requires significant programming effort and expertise. Thus, it is very important to have a preliminary estimate of the expected benefits. Such estimates are commonly computed using Amdahl’s law [Amd67] and its extension for energy [WL08], which provide easy-to-compute formulas for upper bounds for the corresponding parameters. These formulas are based on several simplifying assumptions, such as the processors being identical and their speeds fixed, which do not hold with Turbo Boost. This observation encouraged us to revisit Amdahl’s law for multicore processors.

We present a detailed model of resource performance that reveals new opportunities for efficient scheduling. Using this model, we define a new class of scheduling problems that can be used to boost performance in various parallel and heterogeneous systems. We demonstrate the usefulness of these problems on a new Turbo Boost aware method that efficiently schedules compute tasks on a CPU by solving an instance makespan (total execution time) minimization problem from the new class. While other parallel resources could achieve better speedup from a detailed model, we have chosen to analyze Turbo Boost because its relatively simple performance model allows us to focus on the new scheduling challenges that arise when taking inter-processor performance dependencies into account. To the best of our knowledge, the problem of scheduling multiprocessors where the speed of each processor is influenced by the state of other processors (e.g., active or idle) has not been studied.

The main contributions of this chapter are as follows:

- We propose an extension of Graham’s $\alpha|\beta|\gamma$ classification of scheduling problems [GLLRK79] with a new class of problems, where processor speeds depend on the execution state of the multiprocessor. These new problems model many hardware resources more accurately and enable higher utilization.

- We characterize the computational throughput of CPUs with Turbo Boost and define a minimum-makespan scheduling problem from the new class that can be applied to reduce execution time in a number of applications. For this problem, we present a new efficient
scheduling algorithm that fits the work distribution to the core frequencies in Section 7.3 and prove its optimality for three main sub-classes of the problem in Section 7.4. Our evaluation using a realistic web-log analysis application and a variety of synthetic workloads has shown that the new scheduling method was more efficient than three commonly used schedulers that use the classic multiprocessor model (Section 7.5).

• We simulate various CPU models using their specification data and show that different CPUs and workloads need to be scheduled differently.

• We show analytically and by experimental evaluation that, from a software design perspective, Turbo Boost aggravates the speedup limitations obtained by Amdahl’s law by making parallelization of sequential codes less profitable. This is in contrast to previous studies that showed that from a processor design perspective frequency scaling mechanisms mitigate these limitations by providing higher performance for the same energy budget [AGS05, CJA+09]. We extend the model used in Amdahl’s law to account for frequency changes. Based on this model, we propose refined formulas for the maximum expected speedup and energy scaling, and validate them experimentally.

7.2 New Class of Scheduling Problems

Graham et al. [GLLRK79] presented a common classification scheme for scheduling problems. We describe and extend this scheme with a new class of scheduling problems that includes and

---

Table 7.1: Classic machine type classification

<table>
<thead>
<tr>
<th>$\alpha_1$</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦</td>
<td>single machine</td>
<td>uniprocessor</td>
</tr>
<tr>
<td>$P$</td>
<td>identical parallel machines</td>
<td>$m$ processors with speed $s = 1$</td>
</tr>
<tr>
<td>$Q$</td>
<td>uniform parallel machines</td>
<td>$m$ processors with speeds $s_i$</td>
</tr>
<tr>
<td>$R$</td>
<td>unrelated parallel machines</td>
<td>$m$ processors with per-job speeds $s_{i,j}$</td>
</tr>
<tr>
<td>$O$</td>
<td>an open shop</td>
<td>every job is processed on each of $m$ processors</td>
</tr>
<tr>
<td>$F$</td>
<td>a flow shop</td>
<td>every job on every processor in same order</td>
</tr>
<tr>
<td>$J$</td>
<td>a job shop</td>
<td>every job on every processor with total ordering</td>
</tr>
</tbody>
</table>
extends the problem we presented previously. In Graham’s scheme, problems are classified by the 3-field notation \(\alpha/\beta/\gamma\), where \(\alpha\) describes the machine environment, \(\beta\) describes the job characteristics, and \(\gamma\) describes the objective criterion. Our extension mostly applies to the machine environment.

### 7.2.1 Machine Environment (\(\alpha\))

We use the term processor as a synonym for machine in Graham’s work, the term multiprocessor to refer to the set of processors, and the term \(m\)-processor as an abbreviation for a multiprocessor with \(m\) processors.

The machine environment is described by the first field \(\alpha := \alpha_1\alpha_2\), where \(\alpha_1\) describes the machine type from the list in Table 7.1, and \(\alpha_2\) describes the number of processors. If \(\alpha_2 = \circ\) (empty symbol) then the number of processors is considered a variable of the problem, otherwise it is constant and equal to \(\alpha_2\). For example, machine environment \(\alpha = P\) describes an \(m\)-processor with identical processors \((m\) is a variable). Machine types \(P\), \(Q\), and \(R\) describe multiprocessors with different levels of processor speed uniformity, while machine types \(O\), \(F\), and \(J\) differ in processing order restrictions.

This work extends machine types \(P\), \(Q\), and \(R\), as it focuses on processor speeds. \(R\) is the most generic machine type, and it contains the other two. For machines of this type, the speed of each processor is specified separately for every job. However, the current model assumes this speed to be independent of the state of the other processors, hence limiting its generality. To extend it, we define the concept of processor execution state, as follows. The execution state of a processor is either idle or executing job of type \(\lambda\), for some job type \(\lambda\) from a set of possible types \(\Lambda\). Two jobs are considered to be from the same type if they affect processor speeds in the same manner. We denote the execution state of processor \(i\) by \(x_i\), as follows:

\[
x_i = \varepsilon: \text{ idle}
\]

\[
x_i = \lambda: \text{ executing job of type } \lambda
\]

The execution state of the multiprocessor is a vector of the execution states of its processors and is denoted by

\[
X = (x_i)_{i=1}^{m}.
\]  \hfill (7.1)

We propose an extension to the machine environment that enables characterization of speed dependencies between the processors. The extended machine environment is described by \(\alpha := \alpha_1\alpha_2\alpha_3\), where \(\alpha_3\) is defined as follows:

\(\alpha_3 \in \{\circ, x\}\),

\[
\alpha_3 = \circ: \text{ independent processors}; \text{ the processors’ speeds do not depend on the execution states of other processors (classic model)}.
\]

\[
\alpha_3 = x: \text{ dependent processors}; \text{ the processors’ speeds depend on the execution states of other processors}.
\]
Table 7.2: New machine environment types

<table>
<thead>
<tr>
<th>$\alpha_1$</th>
<th>$\alpha_2$</th>
<th>$\alpha_3$</th>
<th>Speed characterization</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$\cdot x$</td>
<td>${s(a)</td>
<td>a \in M}$</td>
</tr>
<tr>
<td>$Q$</td>
<td>$\cdot x$</td>
<td>${s_{i,A}</td>
<td>A \subseteq M, i \in A}$</td>
</tr>
<tr>
<td>$R$</td>
<td>$\cdot x$</td>
<td>${s_{i,X}</td>
<td>i \in M, X \subseteq (\Lambda \cup {\varepsilon})^m}$</td>
</tr>
</tbody>
</table>

$M = \{1, \ldots, m\}$ - set of processors
$\alpha$ - number of active processors
$A$ - set of active processors
$X$ - execution state

We denote the speed of processor $i$ when the execution state is $X = (x_k)_{k=1}^m$ by $s_{i,X}$. The processor speed specification for an $m$-processor is hence described by:

$$(s_{i,X} | i \in \{1, \ldots, m\}, X = (x_k)_{k=1}^m, x_k \in \Lambda \cup \{\varepsilon\}) \quad (7.2)$$

For machine types where the performance dependencies are simpler, shorter descriptions are used. The characterization of the processor speeds for the new machine environments is listed in Table 7.2.

The new machine environments $Px$, $Qx$, and $Rx$ are described as follows:

$\alpha = Px$ All the processors have the same speed at every time point, and the speed is specified for every number of active processors.

$\alpha = Qx$ The speed of each processor is specified for every set of active processors.

$\alpha = Rx$ The speed of each processor is specified for each execution state.

The environments provide three levels of flexibility in expressing the dependency between the execution state of the processors and their speeds, allowing more accurate characterization of performance while keeping the basic processor speed properties of $\alpha_1 \in \{P, Q, R\}$.

### 7.2.2 Job Characteristics ($\beta$)

In the extended model, the **processing requirement** of each job $j$ is given by $p_j$ (job size). When the job is being executed, its **execution progress** increases at a rate equal to the processor’s speed; the job is completed when its execution progress reaches $p_j$. In previous studies, the processing time of job $j$ on processor $i$ was characterized by $p_{ij}$ [GLLRK79]. In the extended model, this processing time is not fixed; hence the processor speed and the job size are defined separately.

Let $X(t)$ denote the execution state of the multiprocessor at time $t$, and let $\text{proc}(j, t)$ denote the identifier of the processor that executed job $j$ at time $t$, or $\varepsilon$ if none. Then, the progress rate of job $j$ at time $t$ is as follows:

$$s(j, t) = \begin{cases} 
\text{proc}(j, t), & \text{proc}(j, t) \neq \varepsilon \\
0, & \text{otherwise}
\end{cases} \quad (7.3)$$

133
where \( s_{\text{proc}(j,t),X(t)} \) denotes the parameter \( s_{i,X} \) in the processor speed specification with \( i = \text{proc}(j,t) \) and \( X = X(t) \). The execution progress of job \( j \) at time \( \tau \) is

\[
P_j(\tau) = \int_{-\infty}^{\tau} s(j,t)dt \tag{7.4}
\]

and the job’s completion time is \( \tau_j^{\text{end}} = \min(\tau \mid P_j(\tau) = p_j) \).

Other common job characteristics of the classic model include job availability, precedence constraints, preemption, and scheduling model (e.g., offline/online). For example, \( \beta = r_j, \text{prec} \) describes jobs with release dates and precedence constraints. An extended description of the job characteristics is provided in [CPW99].

### 7.2.3 Optimality Criteria (\( \gamma \))

The third parameter \( \gamma \) defines the optimality criteria, i.e., it is a characterization of the best valid schedule. Some of the most common optimality criteria are to minimize the sum or the maximum of the job completion time \( C_i \), lateness \( L_i = C_i - d_i \), tardiness \( T_i = \max(0, C_i - d_i) \), or unit penalty \( U_i = [\text{if } C_i \leq d_i \text{ then } 0 \text{ else } 1] \). For example, \( \gamma = C_{\text{max}} \) aims to minimize the makespan (\( \max_i C_i \)).

### 7.2.4 Summary

We extended the machine environment in the classification of scheduling problems to include multiprocessors with speed dependencies among the processors. Each original parallel machine type now also has an extended counterpart. The new model provides a more accurate view of resource performance and enables scheduling algorithms to achieve better utilization. The extension of the machine environment has doubled the number of scheduling problems considered in the classification, because every original multiprocessor scheduling problem now also has an extended counterpart. For example, the basic problem of minimizing the makespan of preemptive jobs on a multiprocessor of type \( P \), denoted by \( P|mtn|C_{\text{max}} \), has a counterpart problem where the processors’ speed depends on the number of active processors, denoted \( P_x|mtn|C_{\text{max}} \). As a case study, we use an instance of this problem to improve scheduling of compute jobs on a multicore CPU with Turbo Boost in Section 7.3. Alongside its utilization efficiency benefits, the extended model adds complexity to the scheduling problems. As our case study shows, even scheduling problems that have simple and efficient solutions under the classic model are difficult to solve under the extended model. We believe that finding efficient scheduling algorithms for the new problems would enable more efficient utilization of resources in modern computing systems, whose architecture becomes increasingly parallel and complex.
7.3 Scheduling with Turbo Boost

Turbo Boost technology enables higher CPU performance through increased core frequency. Higher frequency can shorten execution times and reduce energy consumption. However, increased frequency comes at the cost of higher power demand and increased heat generation, which must stay below certain limits to avoid damage to the chip. When the measured environmental parameters are below rated limits, Turbo Boost uses the headroom to increase CPU frequency above the base frequency. It dynamically increases and decreases the frequency in the range between the base and maximum Turbo frequencies to keep the parameters within the limits.

The increased core frequency in Turbo Boost can be utilized in task scheduling to optimize application goals. However, efficient task scheduling with Turbo Boost is more challenging than on processors with fixed frequency due to the following:

1. Execution time prediction: task execution times are hard to predict because processing speeds are volatile and depend on external factors.

2. Scheduling: even with an exact model of core frequency, existing scheduling methods cannot be applied as they assume fixed or controllable processor speeds.

We start with an overview of the Turbo Boost technology and related works, define a scheduling problem that uses an extended resource model, and describe a scheduling method that solves this problem.

7.3.1 Turbo Boost Overview and Related Works

Turbo Boost is a dynamic frequency and voltage scaling (DVFS) technology implemented in many Intel processors, ranging from ultra-low power embedded CPUs to high-performance manycore CPUs [Int14b, Int13], and similar technologies such as AMD’s Turbo Core exist in processors from other vendors. The frequency is the same for all active cores and is set using the following parameters [Inta, Intb]: number of active cores, estimated current consumption,
estimated processor prior and present power consumption, processor temperature, and type of workload.

The increments occur in fixed steps of 133MHz for Nehalem microarchitecture and 100MHz for newer microarchitectures (Sandy Bridge through Broadwell). The scaling mechanism is automatic, but it is enabled and disabled on demand by the operating system. The maximum Turbo Boost frequency is dictated by the number of active cores and specified in the processor documentation. For example, the Core i7-4790T Haswell quad-core processor has a base frequency of 2.7GHz and maximum Turbo frequencies of 3.9, 3.8, 3.6, and 3.3GHz, which correspond to 1, 2, 3, and 4 active cores, respectively [Inta].

The effect of Turbo Boost on system performance depends on application compute and memory requirements [CJA+09, WSYI14]. Lo and Kozyrakis [LK14] suggested optimizing Turbo Boost performance for a set of efficiency metrics by enabling/disabling Turbo Boost dynamically at the OS level. Our work takes a different approach – performance is improved by optimizing the task schedule.

Wakisaka et al. [WSYI14] studied the problem of minimizing the makespan of a task graph with communication costs on a processor with Turbo Boost. As this problem is NP-Hard, the authors suggested a heuristic algorithm that is based on List Scheduling. Gawiejnowicz [Gaw08] studied the problem of Time-Dependent scheduling, where processor speeds may change during the execution. However, dependencies between processors were not considered there.

Zhong et al. [ZRL11, ZRL12] studied the problem of resource allocation and mapping (data partitioning) among and within multiprocessors where performance dependencies between processors exist. In this chapter, we study the problem of scheduling jobs on such multiprocessors, which also considers the time dimension.

7.3.2 Model and Scheduling Problem Definition

We formulate a scheduling problem for minimizing the makespan of a set of preemptive compute tasks on a CPU with Turbo Boost. For simplicity, we assume that the core frequency changes are immediate, and use the preemptive unrelated jobs model, i.e., there are no order dependencies between the tasks, their sizes are given in CPU cycles, and the execution of every task can be interrupted and later resumed on any processor without time penalty, but a task must not be executed on more than one core at a time. These assumptions hold for many compute-intensive tasks in High-Performance Computing (HPC).

To model the CPU performance we first characterize the Turbo Boost frequencies. Figure 7.3 shows the Turbo Boost maximum core frequency and total computational throughput for a Core i7-4790T quad-core CPU (throughput = |active_cores| × frequency).

We observe two properties of the throughput function that hold for all the processors that support Turbo Boost:

1. Diminishing returns\(^1\): as more cores are active, the frequency of each core diminishes

\(^{1}\)In economics, the principle that when the investment increases the average return decreases is called the law of diminishing returns.
2. Increasing: the total throughput is greater when more cores are active (Fig. 7.3b).

We denote the speed of any active processor when \( i \) active processors by \( s(i) \), and the total throughput of the multiprocessor by \( S_i \), i.e., \( S_i = i \cdot s(i) \). The observations above are expressed in the constraints given in the following problem definition.

**Problem 15.** Minimize the makespan of a set of preemptive jobs \( J = (p_i)_{i=1}^n \) on an execution-dependent identical \( m \)-processor whose speed is given by \( (s(i))_{i=1}^m \), given that

(A1) for \( 0 \leq i \leq m - 1 \), \( S_i \leq S_{i+1} \), and

(A2) for \( 0 \leq i \leq m - 1 \), \( s_i \geq s_{i+1} \),

where \( S_0 = 0 \) and \( S_i = i \cdot s(i) \) for \( i > 0 \).

This problem is as an instance of the \( Pmtn | C_{max} \) scheduling problem in the classification described in Section 7.2.

### 7.3.3 Existing Scheduling Techniques for Identical Machines

For the problem of scheduling independent tasks on a multiprocessor in the traditional model, a simple technique for constructing an optimal schedule is known (McNaughton, 1959 [McN59]). For a set of preemptive jobs \( J = (p_i)_{i=1}^n \) that execute on \( m \) processors, the makespan of the schedule is given by the formula:

\[
M = \max(p_1, p_2, \ldots, p_n, \frac{1}{m} \sum_{i=1}^n p_i)
\]

The schedule is constructed by successively filling the processors until the processing time reaches \( M \), preempting the jobs if necessary, as illustrated in Figure 7.4. The jobs can be scheduled in any order. The validity of this schedule is straightforward; all the jobs can be scheduled within \( M \) time units since \( M \geq \sum_{i=1}^n p_i \), and no job is scheduled on more than one machine at the same time since \( M \geq \max_i p_i \). This schedule is optimal because its makespan is at one of the following lower bounds: (1) the execution time of the largest job or (2) the time required to execute all the jobs at full multiprocessor throughput. For job sets that are not amenable to equal work distribution among the processors due to job size constraints, i.e., \( \max_i p_i > \frac{1}{m} \sum_{i=1}^n p_i \), the method packs the work into as few processors as possible, such that each processor is assigned at most \( \max_i p_i \) work.

Other commonly used scheduling techniques for such workloads include global and partitioned queuing schemes, time slicing, and work stealing. Lawler & Labetoulle described a linear-programming based polynomial solution to a more general scheduling problem [LL78].

### 7.3.4 Scheduling Challenges of the New Model

McNaughton’s technique computes the schedule event times based on equal and fixed processor speeds. In the new multiprocessor model, this schedule cannot be applied due to the discrepancy between scheduled and real processing times. (The duration of a CPU cycle is the reciprocal \((\frac{1}{\tau})\) of the core frequency for that cycle.)
However, even if the event times in McNaughton’s schedule are corrected to match actual processing times, the schedule may not be optimal. Consider the schedule produced using McNaughton’s technique with corrected event times illustrated in Figure 7.5a. The schedule includes four jobs, where \( p_1 \) and \( p_2 \) are twice the size of \( p_3 \) and \( p_4 \). Under the \( P_x \) model, the makespan of this schedule is \( p_1/s(3) \), as the speed of each of the three active processors is \( s(3) \). Alternatively, \( p_4 \) could be placed on the idle processor, as shown in Figure 7.5b. Although the sizes of the jobs remained the same, their execution times were affected by changes in the processors’ speeds. In this case, the execution would consist of two periods – the first with four active processors and the second with only two – and the makespan would be \( p_3/s(4) + (p_1 - p_3)/s(2) \). On a Core i7-820QM processor with \( (s(i))_{i=1}^{m} = (3.1, 2.8, 1.9, 1.9) \) GHz, which satisfies the problem constraints, the alternative schedule has 16% shorter makespan.

### 7.3.5 Frequency-Aware Scheduling Method (FAS)

We present a new scheduling method for the problem at hand. The method constructs a schedule as follows.

#### Problem Classification and Algorithm Selection

FAS selects a scheduling algorithm based on problem properties. It classifies the input problem into one of four categories by testing the two conditions described in Table 7.3, and selects the scheduling algorithm based on the category.

#### Construct a Speed-oblivious Work Plan

Normally, job execution times are given as input to the scheduling algorithm and only the job processing requirement (number of CPU cycles) is specified. The job execution times can only be computed given a schedule, because the processing time depends on the number of active
Table 7.3: Problem classification

<table>
<thead>
<tr>
<th>Category</th>
<th>Condition 1</th>
<th>Condition 2</th>
</tr>
</thead>
</table>
| Equal Distribute          | \[
\max_i p_i \leq \frac{1}{m} \sum p_i
\]               | \[
\forall i: \frac{1}{s_{i+1}} \cdot \frac{1}{s_i} \leq \frac{1}{s_{i+2}} \cdot \frac{1}{s_{i+1}}
\]               |
| Pack                      | Yes                                                          | any                                                          |
| Load Balance              | No                                                           | \[
\forall i: \frac{1}{s_i} >
\]                                                          |
| Non-monotonic             |                                                              | otherwise                                                     |

![Figure 7.6: Computation of schedule event times](image)

processors as dictated by that schedule. Therefore, our method schedules the job processing requirements rather than their execution time, hence creating a work plan that does not depend on the processor speeds. This schedule would be valid on a unit-speed multiprocessor, but for a different processor the work axis must be transformed into time by scaling each work unit (CPU cycle) by its duration (inverse of processor speed and frequency). We use the terms initial schedule and work plan interchangeably to denote the schedule constructed in this step (Figure 7.6a).

**Compute Schedule Event Times**

The final schedule is constructed by replacing the times of the job start and completion events in the work plan with the actual times according to processor speeds. For every work unit \(i\), the processor speed is determined from the number of active processors in the work plan for that work unit, and its duration is computed as the inverse of the speed, i.e., if \(i\) processors were active then the duration is \(1/s(i)\). The time of each event is computed as the cumulative duration of the preceding work units. Figure 7.6b illustrates the computation of event times by scaling the work plan timeline.

**Scheduling Job Sets That Are Amenable to Equal Distribution**

The total throughput of the multiprocessor is maximal when all the processors are active, as expressed in condition (A1) in the problem definition. Therefore, if the jobs can be scheduled such that the processors are assigned equal shares of the work, without executing any job on more than one processor at a time, then such a schedule would have minimal makespan. A set of jobs \(J = (p_i)_{i=1}^n\) can be
equally distributed between the processors iff

$$\max_i p_i \leq \frac{1}{m} \sum_{i=1}^{n} p_i \quad \text{(Condition 1).} \quad (7.5)$$

Note that the condition does not depend on the processor speeds \((s(i))_{i=1}^{m}\).

We ascribe problems that satisfy this condition to the ‘Equal Distribution’ category, and use McNaughton’s technique to construct the work plan. Figure 7.7a illustrates a work plan for a problem amenable to equal distribution.

**Scheduling Job Sets That Are Not Amenable to Equal Distribution**

For workloads that cannot be equally distributed between the processors, finding the optimal work distribution depends on processor speeds \((s(i))_{i=1}^{m}\). FAS optimizes the work distribution to achieve minimal makespan for these speeds. For two types of multiprocessors, where the processor speeds have certain monotonic qualities, FAS constructs an optimal schedule. For other multiprocessors, it fits the work distribution to the processor speeds using linear programming, achieving near-optimal results for most workloads.

We categorize the problem into one of three categories using the processor speed properties specified next, and describe the schedule construction algorithm in each case. Aspects of validity and optimality of the method are discussed in Section 7.4.

**Pack:** \(\forall i : \frac{1}{s_{i+1}} - \frac{1}{s_i} \leq \frac{1}{s_{i+2}} - \frac{1}{s_{i+1}}\)

On multiprocessors that satisfy this above condition, our scheduling method maximizes the number of processors that have load \(\max_i p_i\). McNaughton’s technique generates a schedule of this form, so we use it to generate the work plan. Figure 7.7b illustrates such a work plan.

**Load Balance:** \(\forall i : \frac{1}{s_{i+1}} - \frac{1}{s_i} > \frac{1}{s_{i+2}} - \frac{1}{s_{i+1}}\)

On multiprocessors that satisfy this condition, our scheduling method aims to achieve maximum load balancing without violating job size constraints. The algorithm repeatedly extracts the biggest job and assigns it to a new processor until Condition 1 is true for the remaining jobs on the remaining processors. The remaining jobs, if any, are then equally distributed among the remaining processors using McNaughton’s technique. Figure 7.7c illustrates such a work plan.

**Non-Monotonic:**

Problems that do not fall into one of the previous categories belong to this most general type. FAS defines a linear programming (LP) problem with \(m \cdot n\) variables, where variable \(y_{i,j}\) denotes the amount of work from job \(j\) assigned to processor \(i\). We denote the total work assigned to processor \(i\) by \(w_i = \sum_j y_{i,j}\), and define the minimization function to be the makespan:

$$f = w_1 \cdot \frac{1}{s(1)} - \sum_{i=2}^{m} w_i \cdot \left( \frac{1}{s(i-1)} - \frac{1}{s(i)} \right). \quad (7.6)$$
The LP problem is defined as follows:

\[
\forall i, j : 0 \leq y_{i,j} \leq p_j \quad \text{lower and upper bound} \quad (7.7)
\]
\[
\forall j : \sum_i y_{i,j} = p_j \quad \text{total work on job } j \text{ is } p_j \quad (7.8)
\]
\[
\forall i < m : w_i \geq w_{i+1} \quad \text{sort } (w_i) \text{ in descending order} \quad (7.9)
\]
\[
\forall i : w_i \geq \sum_{i'=i}^m y_{i',j} \quad \text{job size limitations (not sufficient)} \quad (7.10)
\]

These constraints do not completely enforce job size limitations, and a returned solution may result in a work distribution without a matching valid schedule (infeasible). In Section 7.4 we describe a necessary and sufficient constraint to test the feasibility of a work distribution, and show that satisfying constraint 7.10 above is necessary for a feasible work distribution. To generate a valid solution, the linear program is iteratively solved and the feasibility of the work distribution is checked, until a feasible solution is found. After each iteration that resulted in an invalid work distribution, an additional inequality is added that forces the LP solver to assign more work to the first processor that violates the job size constraints. This method was found very efficient in finding optimal or near-optimal work distributions. Once a feasible work distribution \((w_i)_{i=1}^m\) is found, the schedule is constructed by first sorting the jobs by size in decreasing order, and then successively filling the processors until the processing time reaches \(w_i\), preempting the jobs if necessary. For completeness, if the method fails to find a feasible solution using LP, it uses a work plan generated using McNaughton’s technique.

### 7.4 Validity and Optimality

In this section, we prove that FAS is valid, and that it is optimal for the problems in categories Equal Distribution, Pack, and Load Balance. To prove validity we must show that in the pro-
duced schedule the processing requirement of all the jobs has been supplied and that no job is
scheduled on more than one processor at the same time. The optimality proof must show that
no valid schedule for the problem exists that has a shorter makespan.

We now define several concepts and a theorem, which will later be used in the discussion
on validity and optimality.

Compact Schedule and Work Distribution

A compact schedule $S$ is a schedule where each processor executes its work share continuously
from time $t = 0$ without inserted delay, and the work share of any processor $i$ is greater or equal
to that of processor $i + 1$ (Figure 7.8). FAS constructs the schedule by ‘filling’ processors with
work, one after the other, and never increases the work share of a processor; hence it always
produces a compact schedule.

The work distribution of a compact schedule is characterized by $(w_1, \ldots, w_m)$, where $w_i$
is the work share (total execution progress) assigned to processor $i$. From the definition of a
compact schedule, it follows that $w_1 \geq \ldots \geq w_m$.

The amount of work completed when $i$ processors were active is hence $(w_i - w_{i+1})$ for
$1 \leq i \leq m - 1$, and $w_m$ for $i = m$. For simplicity, we define $w_{m+1} := 0$ to make the formula
uniform for every $i$. The makespan of the compact schedule can be computed from the work
distribution as follows:

$$\text{makespan}(S) = \sum_{i=1}^{m} \frac{w_i - w_{i+1}}{s(i)} . \quad (7.11)$$

The efficiency of schedules can hence be evaluated by their work distribution alone.

Job Size Constraints

FAS optimizes the work distribution and then constructs a matching schedule. For a given work
distribution, a matching valid schedule may not exist due to job size limitations. It must there-
fore be determined whether a valid schedule exists for a given work distribution, i.e., whether
the work distribution is feasible. The following theorem defines a necessary and sufficient set
of constraints for a feasible work distribution.
Theorem 16. Let $J = (p_i)_{i=1}^n$ be a job set, $p_1 \geq \ldots \geq p_n$, and let $W = (w_1, \ldots, w_m)$ be a work distribution. Then, there exists a valid schedule for $J$ with work-distribution $W$ iff

$$\sum_{k=1}^n p_k = \sum_{k=1}^m w_m, \text{ and for every } 1 \leq i \leq m \text{ and } 1 \leq j \leq n,$$

(7.12)

if $w_i < p_j$ then $\sum_{k=1}^j p_k \leq \sum_{k=1}^i w_k$.

(7.13)

Proof. (→) Suppose a valid schedule for $J$ exists with work distribution $W$. Since $W$ is a valid work distribution for $J$, $\sum_{k=1}^n p_k = \sum_{k=1}^m w_m$. Let $w_i < p_j$ for some $i,j$. Since a job cannot be processed on more than one processor at the same time, the progress of each job $1, \ldots, j$ by time $w_i$ was at most $w_i$ and at least $(p_k - w_i)$ work remained to be executed later in processors $1, \ldots, i - 1$. Furthermore, if $j < i$ then only $j$ processors could be used to complete the remaining for job $1, \ldots, j$. Then,

$$\sum_{k=1}^j (p_k - w_i) \leq \sum_{k=1}^{\min(i,j)} (w_k - w_i)$$

$\Rightarrow$

$$\sum_{k=1}^j p_k \leq \sum_{k=1}^{\min(i,j)} w_k + w_i(\min(i,j) - j) \leq \sum_{k=1}^i w_k.$$ 

(←) We describe a method for constructing a valid schedule for $J$ that has work distribution $W$. The schedule is constructed by successively filling the processors with work from $p_1, \ldots, p_n$, in order, until the assigned work to the current processor $i$ reaches $w_i$, preempting the jobs if necessary. Since $\sum_{k=1}^n p_k = \sum_{k=1}^m w_m$, all the jobs have been fully scheduled. We now prove that no job is scheduled on more than one processor at a time (overlaps). Suppose, in contradiction, that some job $j$ overlaps on processors $i$ and $i + 1$. Since it overlaps, it executed on processor $i$ from its starting time $t$ until $w_i$ and on processor $i + 1$ from time $0$ until after time $t$, and therefore $p_j > w_i$. From the supposition, $\sum_{k=1}^j p_k \leq \sum_{k=1}^i w_k$, but then jobs $1, \ldots, j$ fit in $w_1, \ldots, w_i$ and job $j$ would not be "spilled" to processor $i + 1$ – a contradiction. Therefore, no job overlaps in the constructed schedule. We have shown that the schedule supplies the execution demand of all the jobs and that no job was scheduled on more than one processor at a time; hence it is valid and a valid schedule for $J$ with work distribution $W$ exists. $\square$

Theorem 16 contains non-linear constraints, so it is not suitable for an LP problem formulation. The following theorem defines a set of linear constraints that are necessary for a feasible work distribution and were used in the LP problem formulation (constraint 7.10):

Theorem 17. Let $J = (p_i)_{i=1}^n$ be a job set where $p_1 \geq \ldots \geq p_n$, and let $S$ be a valid schedule of $J$ with work distribution $(w_1, \ldots, w_m)$. Then,

$$\text{for } 1 \leq i \leq \min(m, n), \sum_{k=1}^i p_k \leq \sum_{k=1}^i w_k.$$ 

(7.14)
Proof. Since a job cannot be executed on more than one processor at the same time, by the time processor $i$ completed execution, the progress of each of $p_1, \ldots, p_i$ was at most $w_i$. Therefore, a remainder of at least $(p_k - w_i)$ was later executed on processors $1, \ldots, i - 1$: $\sum_{k=1}^{i} (p_k - w_i) \leq \sum_{k=1}^{i} (w_k - w_i)$. Consequently, $\sum_{k=1}^{i} p_k \leq \sum_{k=1}^{i} w_k$.

7.4.1 Validity

To prove the method’s validity, we first prove that the initial schedule is valid for the classic model and then that the final schedule is valid for the extended model.

Initial Schedule / Work Plan

For problems in categories Equal Distribution and Pack, our method uses McNaughton’s technique to produce the initial schedule; therefore it is valid [McN59]. For problems in the Load Balance category, each job is either assigned to a separate processor or scheduled using McNaughton’s technique on the remaining processors, so its processing requirement is supplied and it cannot be assigned to more than one processor at a time. For Non-Monotonic problems, the feasibility of the work distribution that is returned by the LP solver and used to construct the schedule is validated using the constraints in Theorem 16; hence the schedule is valid. If a solution is not found using LP, it is constructed using McNaughton’s technique, so it is valid in this case too. We have thus shown that the initial schedule produced by our method is valid.

Final Schedule

The final schedule is built as a concatenation of periods that correspond to work units in the work plan. By construction, in each period, the execution of jobs that appear in the work unit’s job assignment progresses by the specified amount. Therefore, the execution progress during each such period is the same as the progress of the corresponding period in the initial schedule on a unit-speed multiprocessor. Consequently, the execution progress of each job at the end of the execution is the same for the initial and final schedules on their corresponding systems. Since the initial schedule is valid, it is guaranteed that each job is executed to completion in the final schedule. Also, since the job assignments in the final schedule are taken from the initial schedule, and the initial schedule is valid, no job is assigned to more than one processor at the same time. This concludes the validity proof.

7.4.2 Optimality

We prove that our method is optimal for the problems in categories Equal Distribution, Pack, and Load Balance.

Proof Scheme

We show that the work distribution of the schedule produced by our method guarantees that this schedule has the shortest makespan of all valid schedules. For Equal Distribution, the
Optimality is straightforward as maximum throughput is achieved during the execution.

Compact Schedule Transformation

We exemplify how to transform any schedule to a compact form (Figures 7.9a, 7.9b). The vertical lines in the diagram denote time units and the numbers at the top denote the number of active processors. The transformation procedure is as follows:

1. Sort the time unit periods and the job assignments in them by the number of active processors in descending order.

2. In each period, reassign the jobs to the first \( k \) processors, where \( k \) is the number of active processors.

The compact schedule retains the validity and optimality properties of the original schedule, and its makespan is the same or shorter. We use the compact schedule as a uniform form for comparing makespans. In step 1, sorting the periods does not change the order of commands in each job, but only their time allocation. In step 2, migrating jobs between the processors does not change the execution time because a preemptive job model is used.

Work Redistribution

To understand how changes in work distribution affect the execution time, we examine the effect of local variations in the work distribution on it. A local variation here means that a certain amount of work \( w_{kl} \) is moved from processor \( k \) to processor \( l \) (\( k \neq l \)). The original work distribution is denoted by \( W = (w_1, \ldots, w_m) \) and the modified by \( W' = (w'_1, \ldots, w'_m) \), where \( w'_k = w_k - w_{kl}, \quad w'_l = w_l + w_{kl}, \quad \text{and} \quad w'_i = w_i \) for \( i \neq k, l \). The discussion here is limited to cases where the modification does not cause inversion in the order of \( w'_1, \ldots, w'_m \) in sorted order, i.e., \( w'_1 \geq \ldots \geq w'_m \).

Let \( t \) and \( t' \) denote the makespans of \( W \) and \( W' \), respectively. Then, the execution time difference is

\[
\Delta t = t - t' = \sum_{i=1}^{m} \frac{w_i - w_{i+1}}{s(i)} - \sum_{i=1}^{m} \frac{w'_i - w'_{i+1}}{s(i)}.
\]  

(7.15)

145
Two cases are considered separately: (1) $k = 1$, and (2) $1 < k < l$, depicted in Figure 7.10.

In the first case (Figure 7.10a), all the $w_i, w_{i+1}, w'_l, \text{ and } w'_{i+1}$ terms in the formula, except where $i \in \{1, 2, l, l+1\}$, cancel each other out. The formula is reduced to the following expression:

$$\Delta t_{1l} = w_{kl} \cdot \left( -\frac{1}{s(1)} + \left( \frac{1}{s(l)} - \frac{1}{s(l-1)} \right) \right).$$

(7.16)

The following theorem proves that this time difference is non-positive, i.e., moving work from the processor with the biggest work share to another processor does not increase the execution time. Conversely, moving work to that processor does not decrease the execution time.

**Theorem 18.** For any $1 < i \leq m$,

$$\frac{1}{s(1)} \geq \frac{1}{s(i)} - \frac{1}{s(i-1)}.$$

(7.17)

**Proof.** Condition A1 in the problem definition states that $S_1 \leq S_2 \leq \ldots \leq S_m$. The proof layout is as follows:

$$S_{i-1} \leq S_i \Rightarrow \frac{1}{S_{i-1}} \geq \frac{1}{S_i} \Rightarrow \frac{i}{S_{i-1}} \geq \frac{i}{S_i} \Rightarrow \frac{1}{S_{i-1}} \geq \frac{i}{S_i} - \frac{i-1}{S_{i-1}}$$

$$S_1 \leq S_{i-1} \Rightarrow \frac{1}{S_1} \geq \frac{i}{S_i} - \frac{i-1}{S_{i-1}} \Rightarrow \frac{1}{S_1} \geq \frac{1}{s(1)} - \frac{1}{s(i-1)}.$$

This concludes the theorem proof. 

In the second case (Figure 7.10b), where $1 < k < l$, the formula for $\Delta t_{kl}$ is reduced to the following expression:

$$\Delta t_{kl} = w_{kl} \cdot \left( -\left( \frac{1}{s(k)} - \frac{1}{s(k-1)} \right) + \left( \frac{1}{s(l)} - \frac{1}{s(l-1)} \right) \right).$$

(7.18)

In this case, it is not possible to determine whether the execution time increases or decreases due to the modified work distribution without further inspection of the processor speeds $(s(i))_{i=1}^m$.

Condition 2 in the problem classification examines whether the series $\frac{1}{s_{i+1}} - \frac{1}{s_i}$ for $1 \leq i \leq m-1$ is increasing, decreasing, or non-monotonic. This condition distinguishes between problems in categories 2, 3, and 4. For problems in Category 2, the series is increasing, so
\( \Delta t_{kl} \geq 0 \), i.e., the modification increases the execution time. Conversely, for problems in Category 3, the series is decreasing, so the modification decreases the execution time. For problems in Category 4, the execution time decreases or increases depending on whether \( \frac{1}{s_{(k)}} - \frac{1}{s_{(k-1)}} \geq \frac{1}{s_{(l)}} - \frac{1}{s_{(l-1)}} \) is true or not, respectively. Moving work from processor \( l \) to \( k \) (in the other direction) has the opposite increase or decrease effect on the execution time in each of the described cases.

The effect of work distribution modification on execution time is summarized in Table 7.4.

### Optimality for Problems in Category 2

Our scheduling algorithm for problems in Category 2 iteratively assigns \( \max_i p_i \) work to processors, until the remaining work is less than that, and this remainder, if it exists, is assigned to the next unused processor. The work distribution of the schedule is characterized as follows.

There exists \( 1 \leq j \leq m \) such that:

- \( w_1 = \ldots = w_{j-1} = \max_i p_i \) (maximum load)
- \( w_j = \sum_i p_i - (j-1) \max_i p_i \) (remainder)
- \( w_{j+1} = \ldots = w_{m+1} = 0 \) (we remind that \( w_{m+1} := 0 \))

Consider a case where our algorithm produced a schedule \( S \) with work distribution \( (w_1, \ldots, w_m) \) for job set \( J = (p_i)_{i=1}^n \), and let \( S' \) be some valid schedule for \( J \) with work distribution \( (w'_1, \ldots, w'_m) \). We prove that the makespan of \( S \) is shorter than or equal to that of \( S' \) by showing that the work distribution of \( S' \) can be changed to that of \( S \) without increasing the execution time using local modifications that bring the work distribution closer to the goal. The next modification is chosen by inspecting the difference between the work distributions:

- If \( w'_1 \neq w_1 \) then it must be that \( w'_1 > w_1 \), or otherwise the largest job is necessarily executed on more than one processor at a time. We move the extra \( w'_1 - w_1 \) work from processor 1 to the other processors. This does not increase the execution time, as proven in the redistribution analysis shown previously.

- If the two work distributions differ only in the distribution of the remainder, i.e., \( w'_i = w_i \) for \( 1 \leq i \leq j-1 \), then \( w_j = \sum_{i=j}^m w'_i \) because \( w_{j+1} = \ldots = w_m = 0 \). We transfer the entire load of processors \( j+1, \ldots, m \) to \( w'_j \), i.e., \( w'_{j+1}, \ldots, w'_m \rightarrow w'_j \), starting from
Each of the steps above brings the work distribution closer to that of our schedule without increasing the execution time. Therefore, the schedule of $S$ is shorter than or equal to that of $S'$, proving the optimality of our method for problems in Category 2.

**Optimality for Problems in Category 3**

We show that our method is optimal for problems in Category 3. Let the jobs be sorted in descending order of size $p_1 \geq \ldots \geq p_n$, and let $k$ be the smallest index such that jobs $p_k, \ldots, p_n$ can be distributed equally between processors $k, \ldots, m$, i.e., $p_k \leq \frac{1}{m-k+1} \sum_{j=k}^{n} p_j$. Our scheduling algorithm for problems in Category 3 assigns jobs $p_1, \ldots, p_{k-1}$ to processors $1, \ldots, k-1$, one job per processor, and distributes the remaining jobs equally between processors $k, \ldots, m$.

The work distribution of the schedule is $(w_1, \ldots, w_m)$, where $w_i = p_i$ for $1 \leq i \leq k-1$ and $w_i = \frac{1}{m-k+1} \sum_{j=k}^{n} p_j$ for $k \leq i \leq m$.

**Theorem 19.** Let $S$ be a schedule produced by FAS with work distribution $(w_1, \ldots, w_m)$ for job set $J = (p_i)_{i=1}^{n}$, and let $S'$ be some valid schedule for $J$ with work distribution $(w'_1, \ldots, w'_m)$. Then,

$$
\text{for } 1 \leq i \leq k-1, \sum_{j=1}^{i} w'_j \geq \sum_{j=1}^{i} w_j.
$$

(7.19)

**Proof.** Since a job cannot be executed on more than one processor at the same time, by the time processor $i$ completed execution, the progress of each $p_k \in \{p_1, \ldots, p_i\}$ was at most $w'_i$. Therefore, a remainder of at least $(p_k - w'_i)$ was later executed on processors $1, \ldots, i-1$:

$$
\sum_{k=1}^{i} (p_k - w'_i) \leq \sum_{k=1}^{i} (w'_k - w_i).
$$

Consequently, $\sum_{k=1}^{i} p_k \leq \sum_{k=1}^{i} w'_k$. Since $p_i = w_i$ for $1 \leq i \leq k-1$, the claim follows. \qed

We show that the makespan of $S$ is shorter than or equal to that of $S'$ using a series of local modifications that do not increase the execution time. The next modification is chosen by inspecting the difference between the work distributions:

- If for some $1 \leq i \leq k-1$, $w'_i \neq w_i$, then let $j$ be the first index such that $w'_j \neq w_j$. Then, from Theorem 19, $w'_j > w_j$. We move $w'_j - w_j$ work from processor $j$ to processors with
higher index. This does not increase the execution time, as proven in the redistribution analysis.

- Otherwise, let \( j \) be the last index such that \( w'_j \neq w_j \) (\( j > k \)). Then \( w'_j < w_j \), or otherwise, since \( w'_k \geq \ldots \geq w'_j \), the total load is greater than \( \sum_{j=1}^{n} p_j \). Since the total work is equal in the two work distributions, there is \( w_j - w'_j \) more work assigned to processors \( j+1, \ldots, k \) in the modified work distribution. We move this extra work to processor \( k \). The work redistribution analysis shows that this does not increase the execution time.

Each of the steps above brings the work distribution closer to that of our schedule without increasing the execution time. Therefore, the schedule of \( S \) is shorter than or equal to that of \( S' \), which proves that our method is optimal for problems in Category 3.

### 7.5 Evaluation

This section evaluates the efficiency of FAS and compares it with three other scheduling algorithms. The workloads for the comparison were chosen to represent typical HPC applications.

#### 7.5.1 Methodology

We evaluated the efficiency of FAS for CPUs with Turbo Boost by running realistic and synthetic experiments on one physical CPU, and by simulating it with configurations of multiple Intel Core i7 and Xeon series CPUs.

**Platform**

The evaluation platform was a Supermicro workstation with two eight-core Xeon E5-2690 CPUs, each with 2MB/20MB of L2/L3 cache, 128GB DDR3 RAM, and two Kingston 240GB SSD drives in RAID5 configuration, running Linux kernel 3.13.0, and located in an air-conditioned server room. The base frequency of the CPUs is 2.9GHz, and the maximum Turbo frequencies in GHz for 1-8 active cores are 3.8, 3.6, 3.6, 3.4, 3.4, 3.3, 3.3, 3.3, respectively. We disabled hyper-threading, CPU power limitations, and the CPU in Socket #1, and configured the OS frequency scaling governor to run the CPUs at maximum frequency. Eight worker threads attached to the eight cores executed jobs from a job set according to a given schedule, and logged the start and completion time of each job.

**Baseline Schedulers**

Each job set was executed using FAS and the following three baseline scheduling methods. The total execution time of the job set was used as a measure of scheduler efficiency (shorter is better).

- **McNaughton’s technique** was described in Section 7.3.3. This offline scheduler splits jobs into parts that run on different cores at different times. To prevent them from executing on two
cores at the same time, jobs must not begin execution before their scheduled time. The method assumes fixed core speeds, so for a fair comparison with FAS, the lowest Turbo frequency in the specification (all cores are active) was chosen (3.3GHz), rather than the base frequency (2.9GHz).

**Global-Queue** is a commonly used online task scheduling method. In this method, all the jobs are initially placed into a queue, in arbitrary order. Whenever a core becomes idle and the queue is not empty, it removes a job from the top of the queue and assigns it to an idle core.

**Linux CFS** [Mau10] is the default Linux process scheduler. It is an online scheduler that splits the processor time fairly between the running processes (a.k.a. time-slicing). When evaluating this scheduler, we executed each job in a new thread, instead of using the worker threads. The thread scheduling was done by the OS, but the threads were synchronized on an execution barrier before and after execution to eliminate process creation overhead.

### 7.5.2 Accuracy of the Performance Model

Prior to evaluating the scheduling methods, we first test if our performance model, on which our method is based, is accurate for simple compute jobs. We defined a job set with eight jobs of sizes (8, 7, 6, 5, 4, 3, 2, 1), executing blocks of integer MUL and DIV commands proportional to their size, measured the execution time on each core, and computed the expected execution times using our model. As shown in Figure 7.11, the prediction error did not exceed 0.3%, which confirms that our model is accurate for this kind of applications. The average error in the experiments described below was 0.5%.

### 7.5.3 Synthetic Benchmarks

We examine the performance of each of the schedulers for different types of workloads using synthetic workloads.

**Large Job and Many Small Jobs**

This benchmark tests the schedulers’ performance on workloads that cannot be equally distributed between the processors. The workload consisted of job sets with job sizes [1,...,1,80]
with 0, 1, 2, ... jobs of size 1. Since the large job cannot be executed in parallel, the load cannot be equally distributed between the eight cores (80 each) with fewer than 560 small jobs. Each job executed Quicksort from the standard C library on 4KB of integer data, in a loop.

Figure 7.12 clearly shows the advantage of FAS over the other methods in adapting to different workloads. FAS achieves stable and moderate increase in execution time as the total load increases, which implies near-optimal performance. McNaughton’s technique fails for workloads with few smaller jobs, taking up to 12% longer than FAS, as it packs the work into few cores. The execution time flattens when two cores have been filled, as the benefit of load balancing is reduced. Global Queue and Linux CFS both assign the large job and a fair share of small jobs to one of the cores, causing great load imbalance and a performance hit of x6.8 per job, compared to FAS. For Linux CFS there is also a sharp leap in execution time when the memory footprint outgrows the L2 cache; other schedulers process the jobs on each core serially, so their memory footprint is much smaller.

Equal Distribution

This benchmark tests the schedulers’ performance for workloads where work can be equally distributed between the processors using job preemption. The job set consisted of nine (9) unit-size jobs that execute AES-CBC processing. Figure 7.13 shows our time measurement results. FAS and McNaughton’s technique perfectly balanced the load between all the cores using job preemption, while Global Queue assigned two of the jobs to the same core and was 67% slower. Note that while core #1 was assigned two jobs, its execution time was shorter than double that of the other cores due to the higher Turbo frequency during the second period. Linux CFS sliced the jobs’ execution time unevenly between the cores and had 13% higher execution time.
7.5.4 Web-log analysis

We evaluated the performance using a realistic web-log analysis application implemented as a MapReduce program. The application is composed of two steps:

1. Data preparation (Map): the application reads records of accesses to a website from a list of Apache Web Log files, extracts the client IPs from them, finds their countries of origin, and sorts them by country.

2. Data processing (Reduce): for each country, the application reads the IP addresses for this country, and uses each IP as a key to decrypt a new 128B data block that it reads from a file located on the SSD drive, using AES-CBC 128-bit decryption from the Crypto++ 5.6.2 library.

In this experiment, we only evaluated the data processing step of the application (step 2).

We have used the GeoWeb Apache Log Generator tool [Jaf13] to generate realistic Apache web logs. The logs were then preprocessed as described in step 1, and a text file for each country was created containing the IP addresses of accesses that originated from that country. For each file, we defined a job to process the IPs listed in it (20 jobs in total). From these jobs we generated 20 job sets, such that job set $i = 1..20$ was composed of the largest $i$ jobs (top countries by number of accesses).

Figure 7.14a shows the execution times of the job sets using each of the schedulers. FAS achieved the shortest execution times in all of the experiments (within a 1% margin). The results emphasize the shortcomings of the baseline methods. McNaughton’s technique packs the jobs into fewer processors, while balancing the load between the cores is preferred for the tested CPU model. Global Queue assigns the jobs in their order of appearance (random), which causes instability in the work distributions. In Linux CFS, some cores have to process shorter jobs in addition to longer ones, which results in extra load on the most heavily loaded cores. The speedup of FAS over the baseline schedulers (McNaughton, Global Queue, and Linux CFS) was MAX [12%,15%,13%] and AVG [7%,4%,4%] (Figure 7.14b).

7.5.5 Simulation of Performance on Other CPUs

In Section 7.3.5 we have shown that, theoretically, different work distribution (WD) should be used with different CPUs, depending on their Turbo frequencies. In this section, we extend this
Figure 7.14: Web-log analysis application
statement to real CPUs by simulating them using their Turbo frequencies specification. The simulation was based on our performance model, which was shown to be highly accurate in previous experiments.

**Work distribution on Core i7-820QM**

We found that the quad-core Core i7-820QM CPU for the mobile market best shows the importance of fitting the WD to the Turbo frequencies. Here, we compare FAS to McNaughton’s technique, which packs the work into few active cores, and Global Queue, which balances the load.

For this CPU, the following two workloads: [6,1,1,1,1,1] and [6,1...{x12}] (12 jobs of size 1) need to be scheduled with completely different work distributions. Figure 7.15a shows that for the first workload, packing the work into only two active cores is most efficient, with 13% speedup over the balanced schedule. On the other hand, Figure 7.15b shows that for the second workload Global Queue (load balancing) outperforms McNaughton’s method (packing). Interestingly, FAS found an even better WD, which is neither fully load balanced nor fully packed, with 20% speedup over McNaughton’s technique and 4% over Global Queue.

**Breakdown by Series and Number of Cores**

We collected and analyzed the frequency specifications of Intel’s Core i7 and Xeon series CPUs (413 CPU models released by January 2015), and classified them as having a *Packed* or *Balanced* preferred WD. The classification was made based on the WD created for each CPU by FAS for the job set \([10,1...{x5 \cdot (m - 1)}]\), where \(m\) is the number of CPU cores. A Packed WD denotes that at least 66.7% of the work was done by cores \(1::m/2\). CPUs with only two cores and CPUs with uniform Turbo frequencies were omitted from the evaluation.

Figure 7.16 shows the preferred WD by series and number of CPU cores. For the Core i7 series, each of the two WDs was preferred by close to 50% of the CPUs, while for Xeon Balanced was preferred in 68% of the CPUs. Further analysis shows that a Packed WD was preferred by most CPUs with 4 cores, one 6-core i7, and 31% of the 8-core Xeons. CPUs with more than 8 cores always preferred Balanced WD. In total, 63% of the CPUs preferred a Balanced WD.
7.6 Extending Amdahl’s Law for Multicores with Turbo Boost

7.6.1 Introduction

Multithreading is known to provide speedup and reduced energy consumption. However, writing and validating parallel codes requires significant programming effort and expertise. Thus, given a sequential code, it is important to have a preliminary estimate of the expected benefits of its parallelization. Such estimates are commonly computed using Amdahl’s law \([\text{Amd67}]\) and its extension for energy \([\text{WL08}]\), which provide easy-to-compute formulas for upper bounds for the corresponding parameters. These formulas are based on several simplifying assumptions, such as the processors being identical and their speeds fixed, and they do not account for external factors that may affect the performance of the cores.

Most modern processors are limited by maximum power and thermal constraints. Therefore, in multicore processors the nominal frequency is set according to the power consumption and temperature when all the cores are active. However, when some of the cores are idle, the spare power budget can be utilized to increase the frequency of the active cores; most modern multicore processors implement such a mechanism. For example, Intel CPUs with Turbo Boost technology \([\text{Int08}]\) are equipped with a frequency governor that aims to maximize core frequencies such that the power consumption, current consumption, and temperature stay below certain limits; AMD processors use a similar technology called Turbo Core. Hereafter, we refer to these mechanisms as Turbo Boost. These technological advances encouraged us to revisit Amdahl’s law for multicore processors.

Using analysis and experimental evaluation, we show that, from a software design perspective, Turbo Boost aggravates the speedup limitations obtained by Amdahl’s law by making parallelization of sequential codes less profitable. This is in contrast to previous studies showing that from a processor design perspective, frequency scaling mechanisms mitigate these limitations by providing higher performance for the same energy budget \([\text{AGS05, CJA}^+\text{09}]\). We extend the model used in Amdahl’s law to account for frequency changes. Using this extended model, we propose refined formulas for the maximum expected speedup and energy scaling, and validate them experimentally.
7.6.2 Background

Amdahl’s Law

Amdahl’s law states that having multiple processors to accelerate a program has limited effect on the speedup unless the processing rate of its sequential component is increased by nearly the same magnitude. Gene Amdahl illustrated this “law” using the following simplified model. The program has two parts: a parallel part of proportionate size $f$, which is equally split without scheduling overhead between $N$ identical processors running in parallel, and a sequential part of size $1 - f$ that does not overlap the parallel part. The speedup in this case is computed using the following formula:

$$speedup_{Amdahl} = \frac{1}{(1 - f) + \frac{f}{N}}.$$  

(7.20)

Using this model, Amdahl advocated for the continuation of using single-processor machines (in 1967) [Amd67, Amd13]; the formula was, in fact, derived from his original paper by others.

Four decades later, when the industry turned to multicore processors, a series of works extended Amdahl’s law to take into account additional characteristics of these processors (see the survey in [ABAKAFA13]). The assumptions in Amdahl’s law were refined for multicores in two main directions:

- **Architecture**: use of various chip designs, including symmetric, asymmetric, dynamic, distributed, and heterogeneous multicores, and dynamic frequency scaling.

- **Shared resources and scalability**: cache contention, limited memory latency/bandwidth, communication between cores, synchronization, context switching, and computation scaling.

The extended models provided more accurate estimation of performance and energy consumption, and showed architectural tradeoffs in multicore design. Hill and Marty [HM08] compared different multicore designs under a fixed resource budget (e.g., chip area) and provided important insights for chip architects. Their designs consist of a number of basic cores and at most one more powerful core. These models generally cannot represent dynamic voltage and frequency scaling (DVFS), where the power of more than one core can be increased or decreased dynamically. Optimizing time and energy in multicores using DVFS was examined in [CM08, CM10, LnD10, AGS05, GO15]. Turbo Boost implements many such optimizations in an embedded automatic power management mechanism.

This paper considers a different aspect of Amdahl’s law – estimating the improvement ratio in time and energy between parallel and serial codes on an existing multicore processor with Turbo Boost. Unlike previous works, in our model the sequential program runs on one core of the multicore processor.

Energy Scaling

The energy consumed by a program is equal to $P \times T$, where $P$ is the average power consumption and $T$ is the execution time. Woo and Lee [WL08] extended Amdahl’s law for energy
efficiency. In their augmented model, each processor consumes one unit of power when it is in active state and a fraction $\pi$ when it is in idle state ($0 \leq \pi \leq 1$). The energy consumption of the parallel program is given by the following formula:

$$E = (1 - f) \cdot (1 + (N - 1)\pi) + \frac{f}{N} \cdot N$$

$$= 1 + (N - 1)\pi(1 - f) . \quad (7.21)$$

The energy consumption of the sequential program, which is also executed on the multicore processor, is equivalent to the energy consumption of a parallel program with $f = 0$. Therefore, the energy scaling factor is as follows:

$$E^+ = \frac{1 + (N - 1)\pi}{1 + (N - 1)\pi(1 - f)} . \quad (7.22)$$

Normally, program parallelization reduces energy consumption, as the numerator is greater than the denominator.

### 7.6.3 Extended Amdahl’s Law

#### System Model

A compute-intensive sequential program is executed on a symmetric multiprocessor with $N$ cores and Turbo Boost. An alternative implementation is considered, where a part of proportionate size $f$ is split equally between the $N$ cores without overhead; we call this implementation the parallel program. Our aim is to compute the maximum expected speedup and energy improvement of the parallel program over the sequential one. Our model extends Amdahl’s law to account for multiprocessors where the performance of each processor depends on the number of active (non-idle) processors. Concretely, we focus on multicore CPUs with Turbo Boost, which operate at higher frequency when fewer cores are active. For example, Tables 7.5 and 7.6 show that the maximum CPU frequency scales inversely with the number of active cores in two Xeon series CPUs. We denote the effective speed (performance) of every active core when $n$ cores are active by $s(n)$. Note that $s(N)$ is higher than the base frequency. This is partly because the chip’s temperature depends on other factors besides the number of active cores.

#### Extended Speedup Model

In the extended model, the core speeds are not fixed. For simplicity, we normalize all the speeds by $s(1)$. Thus, when only one core is active, its normalized speed is 1, and when $N$ cores are active, the normalized speed of each core is $s(N)/s(1)$. We modify the speedup formula to match the extended model by introducing a correction factor for the parallel part:

$$\text{speedup}_{\text{multicore}} = \frac{1}{(1 - f) + \frac{f}{N} \cdot \frac{s(1)}{s(N)}} . \quad (7.23)$$
Table 7.5: Frequency specification: 8-core Intel Xeon E5-2690

<table>
<thead>
<tr>
<th>Active cores</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency [GHz]</td>
<td>3.8</td>
<td>3.6</td>
<td>3.6</td>
<td>3.4</td>
<td>3.4</td>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Table 7.6: Frequency specification: 12-core Intel Xeon E5-2658 v3

<table>
<thead>
<tr>
<th>Active cores</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 – 12</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency [GHz]</td>
<td>2.9</td>
<td>2.9</td>
<td>2.7</td>
<td>2.6</td>
<td>2.5</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Normally, the factor $s(1)/s(N)$ is greater than 1 because higher frequency is achieved when some cores are idle. Therefore, the above formula provides a tighter upper bound on the speedup than does Amdahl’s law.

### Extended Energy Scaling Model

The energy model of Woo and Lee [WL08] assumes that active cores always have the same power consumption. However, in multicore with frequency scaling mechanisms, active cores consume more power when some cores are idle, because they operate at higher frequency. We augment the model we used for the speedup formula with information about power to compute the energy improvement factor.

Let $P(n)$ denote the average power consumption of the CPU when $n$ cores are active ($1 \leq n \leq N$). Then, the energy consumption during a period of length $t$ where $n$ cores are active is $P(n) \cdot t$. The energy scaling factor is hence described by the following formula:

$$
\frac{1 \cdot P(1)}{(1 - f) \cdot P(1) + \frac{f}{N} \cdot \frac{s(1)}{s(N)} \cdot P(N)}
$$

We rewrite this formula in a slightly more compact form:

$$
E^+_{\text{refined}} = \frac{1}{(1 - f) + \frac{f}{N} \cdot \left( \frac{P(N)}{s(N)} \cdot \frac{P(1)}{s(1)} \right)} .
$$

(7.24)

### 7.6.4 Evaluation

In this section, we evaluate the accuracy of the traditional and refined formulas for speedup and energy scaling. We start with an experimental evaluation on two Intel multicore CPUs with Turbo Boost. Then, we perform a theoretical analysis to identify how different problem parameters will affect accuracy, assuming our model is correct, and examine the potential for improvement in a series of Intel Xeon CPU models, based on their specification.

### Experimental Setup

We used two evaluation platforms that were located in an air-conditioned server room.
• A Sandy Bridge platform composed of two eight-core Intel Xeon E5-2690 CPUs with 128GB RAM that runs Linux kernel 3.13.0.

• A Haswell platform composed of two twelve-core Intel Xeon E5-2658 v3 CPUs with 64GB RAM that runs Linux kernel 4.0.9.

We disabled hyperthreading, CPU power limitations, and the CPU in Socket 1, and configured the OS frequency scaling governor to run the CPUs at maximum frequency. The base frequency and the maximum Turbo frequencies for the two processor types are shown in Tables 7.5 and 7.6.

A worker thread was attached to each core and executed a given function in a loop, for a given number of iterations. Each thread measured its execution time, and thread 0 also measured the energy consumed by the cores and by the entire processor package, using processor counters MSR_PKG_ENERGY_STATUS and MSR_PP0_ENERGY_STATUS. The process was run with maximum priority.

We measured the accuracy of speedup and energy improvement formulas in the following three configurations:

INT-SB: a synthetic sequence of integer MUL and DIV operations on the Sandy Bridge platform.

AES-SB: an AES-CBC 128-bit encryption function from the Crypto++ 5.6.2 library on the Sandy Bridge platform.

AES-HW: the AES-CBC function on the Haswell platform.

For each configuration, a program with a configurable parallel portion \( f \in [0, 1] \) and a sequential portion \( (1 - f) \) was executed, and its execution time and energy consumption were measured. The experiments were run with Turbo Boost enabled and again with it disabled.

**Speedup Measurements**

The time measurement results are listed in Table 7.7. For each value of \( f \), we computed the speedup by dividing the execution time of the sequential program \( (f = 0) \) by that of the parallel program, and used it to compute the error of each formula.

Figure 7.17 shows the estimation errors of the traditional and refined speedup formulas for the AES-HW configuration, with Turbo Boost enabled. The traditional formula overestimated the speedup by 16% for \( f = 1 \), while the refined formula accurately estimated the speedup with a maximum error of 0.3% and as low as 0.04% for \( f = 1 \). These results clearly show that the refined formula significantly improves the accuracy of the speedup estimation, indicating that the upper-bound on speedup given by the refined formula is achievable for real-world applications. For INT-SB, the error similarly increased, up to 15.1% for \( f = 1 \), while the refined formula was very accurate with 0.05% maximum error. The function here was designed for maximum utilization of the ALU; hence these results verify that the core frequencies are as expected. For AES-SB, the results were similar; the maximum error of the original formula was 16.2%, while for the refined formula it was 0.9%.
Table 7.7: Execution time [sec]

<table>
<thead>
<tr>
<th>Xeon E5-2690</th>
<th>Xeon E5-2658 v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>INT</td>
</tr>
<tr>
<td>0</td>
<td>15.8 20.7</td>
</tr>
<tr>
<td>0.2</td>
<td>13.1 17.1</td>
</tr>
<tr>
<td>0.4</td>
<td>10.4 13.4</td>
</tr>
<tr>
<td>0.6</td>
<td>7.7 9.8</td>
</tr>
<tr>
<td>0.8</td>
<td>5.0 6.2</td>
</tr>
<tr>
<td>1</td>
<td>2.3 2.6</td>
</tr>
</tbody>
</table>

Table 7.8: Package energy consumption [J]

<table>
<thead>
<tr>
<th>Xeon E5-2690</th>
<th>Xeon E5-2658 v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>INT</td>
</tr>
<tr>
<td>0</td>
<td>712.8 639.6</td>
</tr>
<tr>
<td>0.2</td>
<td>608.7 548.0</td>
</tr>
<tr>
<td>0.4</td>
<td>509.4 457.5</td>
</tr>
<tr>
<td>0.6</td>
<td>407.3 366.3</td>
</tr>
<tr>
<td>0.8</td>
<td>307.9 274.6</td>
</tr>
<tr>
<td>1</td>
<td>209.4 182.8</td>
</tr>
</tbody>
</table>

Figure 7.17: Speedup estimation accuracy for AES-HW configuration

Figure 7.18: Energy scaling estimation accuracy for AES-HW config.
Energy Consumption Measurements

Figure 7.18 shows the estimation errors of the traditional and refined energy scaling factor formulas for the AES-HW configuration, with Turbo Boost enabled. The refined formula significantly improved the accuracy, though its error was not as low as for the speedup. Notably, the maximum error of the original speedup and energy improvement formulas is the same at 16%. We show that this result is expected in our theoretical analysis in Section 7.6.4. As for speedup, the results indicate that the estimated improvement in energy consumption is achievable in real-world applications. For INT-SB, the trend was similar. The original formula had errors 5.0% and 15.1% for \( f = 0.8 \) and \( f = 1 \) in the original formula, while the refined formula had a maximum error of 0.8%, for \( f = 0.6 \), and an error of 0.03% for \( f = 1 \). For AES-SB, the errors were higher for both formulas; for the original formula they were 10% and 16.2%, for \( f = 0.8 \) and \( f = 1 \), while for the refined formula they were 2.1% and 0.9%, respectively.

The refined formula uses the average power consumption parameters \( P(1) \) and \( P(N) \). We measured \( P(n) \) for every \( 1 \leq n \leq N \) active cores as follows: a fixed-size workload was split equally between the \( n \) cores, and the power was computed by dividing the CPU package energy consumption by the execution time. On the Haswell platform with Turbo Boost enabled, for the AES-CBC workload, the power scaling was close to linear, \( P_{pkg}^{T}(n) \approx 3.7n + 37.9 \) [W], but the curve was not smooth. With Turbo Boost disabled, it was also close to linear and the curve was smoother; \( P_{pkg}^{D}(n) \approx 3.2n + 35.7 \) [W].

Accuracy With Turbo Boost Disabled

Our refined formulas are based on the assumption that Amdahl’s law accurately models multi-cores with fixed frequency; hence, we expect the original formulas to be accurate with Turbo Boost disabled. The results of our experiments confirm this assumption. For INT-SB, the maximum error of the original speedup function was only 0.1%. For AES-HW, the error was 1.2% for all \( f = 0.2 - 1.0 \). Interestingly, for AES-SB, the error for \( f = 0.2 - 0.8 \) was at most 0.4%, but for \( f = 1 \) it was 4.4%. Our temperature measurements indicate that this was not caused by thermal issues. Note that in all the experiments, the parallel part is executed by all the threads, so any delay caused by using all the cores (such as frequency throttling) should have an effect for any \( f > 0 \). We ascribe this error to microarchitectural issues.

For energy scaling, the formula based on Woo and Lee’s model was also quite accurate. For INT-SB, the maximum error was 0.3%; for AES-HW it was 1.3%; and for AES-SB it was again 4.4%. Note that Woo and Lee’s formula yields different estimates when Turbo Boost is enabled and when it is disabled because the idle state power parameter \( \pi \) is different in the two cases, which is why there is a difference in the estimation error.

Theoretical Analysis

Assuming that our extended model is correct, the error of the original formula is \( \text{abs}(\frac{o - r}{o}) \) where \( o \) and \( r \) are the results of the original and refined formulas, respectively. After rearranging the
terms, the expected speedup error is

\[ \text{err}_{\text{speedup}} = \frac{1}{N \cdot \left( \frac{1}{f} - 1 \right)} + 1 \cdot \left( \frac{s(1)}{s(N)} - 1 \right). \]  

(7.25)

The formula shows that the factors that can increase the error include increasing the parallel part \( f \), changing to a CPU with a larger range of Turbo frequencies, or changing to a CPU with fewer cores. We used this formula to find that the average improvement potential for 36 Haswell-EP CPU models with 4-18 cores is 15.5%, and the maximum is 38%.

To compute the energy improvement error, we first need to bring the two formulas to use the same representation of power. \( \pi \) is defined as the fraction of the power of an active core consumed by an idle core; consequently, it can be represented as follows:

\[ \pi = \frac{N}{N-1} \cdot \frac{P(1)}{P(N)} - \frac{1}{N-1}. \]

The expected error in energy improvement is

\[ \text{err}_{\text{energy}} = \frac{\left( \frac{s(1)}{s(N)} - 1 \right) f}{N \cdot \frac{P(1)}{P(N)} \cdot (1 - f) + f}. \]  

(7.26)

The error can increase by the factors mentioned for speedup, and additionally by changing to a CPU where idle cores consume less power.

For \( f = 1 \) we get \( \text{err}_{\text{speedup}} = \text{err}_{\text{energy}} = \frac{s(1)}{s(N)} - 1. \) This explains why we got the same maximum error for speedup and energy in our experiments; e.g., for AES-HW, this error is \( \frac{2}{2^3} - 1 = 16\% \), as we observed.

### 7.7 Discussion and Future Work

Current schedulers of parallel resources are mostly based on the independent processing-unit model. This model is fairly accurate for sparsely distributed systems such as computer clusters, while in tightly packed systems such as multicore CPUs there are significant performance dependencies between the processing units. This chapter lays the foundation for characterization of such dependencies and efficient task scheduling in these systems.

We extended Graham’s \( \alpha|\beta|\gamma \) classification of scheduling problems [GLLRK79] to a new class of problems, which include an extended resource model. We believe that the extended model can be used to improve the utilization of various systems, including multicore and many-core processors, low-power systems, and communication networks. For example, the extended model can be used to achieve higher utilization in multicore CPUs with Turbo Boost, where the frequency is scaled in accordance to the number of active cores.

We characterized the performance of Intel CPUs with Turbo Boost and presented FAS, a new scheduler that adapts the work distribution to the CPU frequencies to achieve minimal total execution time. FAS performed as well as or better than McNaughton’s technique, Global Queue, and Linux CFS schedulers in all of the realistic and synthetic workloads we tested on a platform with a Xeon E5-2690 CPU, and the analysis revealed weaknesses of these
methods for certain types of workloads. In a web-log analysis application, FAS generated up to 12% shorter schedules than the best baseline scheduler. On synthetic workloads with one large job and multiple smaller jobs, FAS balanced the load effectively, while with Global Queue and Linux CFS schedulers each job took 6.8 times longer to complete. Even for a simple workload with equally-sized jobs, FAS’s schedule was 13% shorter than that of Linux CFS and 67% shorter than that of the Global Scheduler. Simulation of different CPUs based on their Turbo frequency specification showed that the work distribution approach (packed or balanced) needs to be adapted for the specific CPU and workload. Our theoretical analysis showed that for certain CPU models, an optimal schedule for $n$ jobs can be constructed in $O(n)$ time, while for others this task is more complex and the current solution may not work as a periodic online scheduler. Classification of CPUs by their scheduling properties can assist in the choice of a hardware configuration and the development of fast heuristic scheduling methods.

Turbo Boost and similar frequency scaling technologies such as Turbo Core have changed the scaling curve of parallel programs. One might expect that without any parallelization overhead, the speedup would be linear, but with Turbo Boost the performance scales sub-linearly. While the use of multiple core processors has become ubiquitous in modern computer systems, much of the legacy software remains sequential due to the significant cost and effort required for its parallelization. Therefore, simple tools for initial evaluation of the speedup and energy savings that could be gained from the parallelization are needed.

We have shown that the speedup limitations obtained under Amdahl’s law are greater on multicore processors with Turbo Boost, where the use of multiple cores is accompanied by lower frequency. Previous works [AGS05, CJA+09] conversely claimed that Turbo Boost mitigates these limitations. The difference in the conclusions results from a different choice of reference points. The prior works showed that multicore processors with scaling mechanisms can complete the program in shorter time than their fixed-performance counterparts that have the same power consumption. This work, on the other hand, evaluated the speedup factor achieved by parallelizing a sequential program such that it makes use of multiple CPU cores.

We presented a refined formula for Amdahl’s law that computes a tight upper bound on the speedup for CPUs with Turbo Boost while having a minimal impact on the simplicity of the model. In our experiments with AES-CBC 128-bit encryption and a synthetic compute-intensive function on an eight- and twelve-core Intel Xeon series CPUs with Turbo boost, the estimated speedup using Amdahl’s law was up to 16% too high. In contrast, our refined formula had an estimation error of less than 1%. The changes in core frequency with Turbo Boost affect the power consumption of the CPU, so the extension of Amdahl’s law for energy scaling also needs to be revised. We presented a refined formula for the energy improvement factor that can be achieved from parallelization on a CPU with Turbo Boost and showed that it significantly reduces the estimation error. Using theoretical analysis, we showed that increasing the parallel part, changing to a CPU with a larger range of Turbo frequencies, or changing to a CPU with fewer cores can increase the necessary correction factor. For 36 Haswell-EP CPU models with 4-18 cores, we found the average correction factor to be 15.5%, and the maximum 38%. For energy, changing to a CPU with more power-efficient idle cores can also increase the correction.
factor. The analysis also shows that with a fully parallelizable program, the correction factor for speedup and energy is the same.

Normally, the frequency of a core is highest when all other cores are idle; however, this is not guaranteed by the CPU specification. Theoretically, it is possible that for some CPU and workload, the sequential program would run only at base frequency (e.g., 2.2GHz), while the parallel program would run at a Turbo frequency (e.g., 2.5GHz). In this case, the speedup would be higher than the upper bound given by Amdahl’s law. For example, suppose the sequential program starts running at maximum Turbo frequency and shortly after that, the core overheats and its frequency drops to base; in the parallel program, however, each core runs at a lower Turbo frequency and does not overheat.
Chapter 8

Discussion and Conclusion

Computing platforms that combine latency-oriented and throughput-oriented processors, such as multicore CPUs and discrete GPUs, have the potential to provide high throughput and short response times. However, their use in real-time data stream processing systems (RTSPSs) is complicated by the need to simultaneously satisfy both the high aggregate throughput of all the streams and the strict processing latency bounds of each individual stream. In such a platform, each type of compute unit provides different qualities that are essential to satisfy the challenging requirements. Therefore, its effective use in an RTSPS can only be achieved through collaborative resource management and scheduling.

The goal of this dissertation was to develop an approach to efficient processing of real-time data streams on a computing platform that consists of CPUs and GPUs.

One of the main challenges we encountered was to achieve efficient utilization of the GPU’s rich computational resources for multiple small data-processing tasks that have different computational demands and deadlines. While GPUs are designed to effectively handle massively-parallel tasks that can be partitioned into independent work units, they execute these work units as a batch, which limits their ability to complete certain work units before others and complicates the timing analysis. This observation led us to a new approach for executing tasks with limited internal parallelism on the GPU – to execute batches of tasks concurrently; previous studies on GPU task scheduling only considered tasks that occupy the entire GPU [ACOTN10, EWA13, KL11, RCS+11].

By handling multiple data streams at the same time on the GPU, it is possible to achieve high utilization of the GPU resources. However, the batch-based method gives rise to two main problems: uneven work distribution between the processing units and high latency. The first problem is due to combined execution of multiple tasks with different sizes and it strongly affects the GPU throughput and the predictability of the kernel execution time. The second problem arises because small batches do not contain sufficient parallelism to saturate the GPU, and small DMA transactions result in low throughput; consequently, processing streams with short latency bounds (deadlines) on the GPU could result in deadline misses.

CPUs are more robust to variance in task sizes than GPUs and can complete a task faster, but they have smaller computational capacity. Therefore, we reversed the normal relation between
the CPU and GPU, and used the CPU as the offload device. In Chapter 4, we developed the Rectangle method for CPU-GPU work distribution. The method assigns the streams that negatively affect GPU performance, such as those with unusually high computational demands or short latency-bounds, to the CPU, and all other streams to the GPU. To find a valid distribution of the streams, it scans through the CPU/GPU partitions of the stream set in a 2D space where the axes are stream rate and deadline, in search of a partition that satisfies the computational requirements and time restrictions of all the streams. Our analysis showed that static assignment of data streams to the compute units is preferable to dynamic work distribution, due to the high communication cost involved in state migration between the CPU and GPU memories. Our evaluation of the Rectangle method for stream sets with different numbers of streams and various distributions of rates and deadlines showed that collaborative CPU-GPU work distribution is essential to provide reasonable performance for computationally demanding workloads that include short-deadline streams. The Rectangle method yielded up to 30% higher throughput in comparison to simpler collaborative stream distribution methods.

A key problem in real-time systems is computing the worst-case execution time (WCET) of the tasks. Unfortunately, modern GPUs do not lend themselves to formal timing analysis because their internal scheduling mechanisms and device drivers are not fully revealed by the vendors. Therefore, the research in computing the WCET of GPU kernels has focused empirical evaluation and statistical methods [BSBT14, BD13] to estimate execution time. The WCET analysis in Chapter 4 focused on kernels that process multiple data-stream processing tasks. To further simplify the problem, we introduced two additional assumptions: (1) the same processing function is applied to all the streams, and (2) the length of the task is proportional to the size of the data being processed; these assumptions hold for many applications that process multiple streams, such as video monitoring for multiple cameras and a security layer for a VoIP server. We developed an empirical performance model for processing a batch of tasks with arbitrary sizes on the GPU. Our benchmarks show that the GPU performance scales sub-linearly with the number of tasks; i.e., increasing the number of tasks decreases the average time per task, but it increases the total execution time. This performance model did not match any of the models previously used in the real-time scheduling literature.

Multiple CPUs and GPUs can be used to meet the processing needs of an RTSPS. In this setup, the stream partitioning problem is more complex – it is necessary to decide not only whether to assign each stream to a CPU or a GPU, but also to select the specific CPU or GPU to which it will be assigned. The additional partitioning options enable better fitting of the stream distribution to the capabilities of each compute unit. For example, it is possible to improve load balancing on the GPUs by assigning small, medium, and large tasks to different GPUs. Similarly, it is possible to sacrifice the performance of one GPU that would process all the streams with short deadlines, to allow the others to benefit from greater efficiency. In Chapter 5, we provided four methods for work partitioning in multi-GPU systems that build upon the Rectangle method and present different compromises between algorithm complexity and the achievable throughput. For a system with $n$ GPUs, the methods select $n$ non-overlapping rectangles in the rate-deadline space that define a partition of the streams. The streams that
fall into each of them are assigned to the corresponding GPU, and the rest of the streams are assigned to the CPUs. As in the Rectangle method, these methods search for a partition that satisfies the computational requirements and time restrictions. To decrease the search complexity, the methods reduce the search space by applying different constraints on the way streams can be partitioned. Using the Virtual GPU method, our stream processing framework allows up to 87% more data to be processed per time unit than the Rectangle method by taking advantage of a second GPU, and up to a 106% increase in GPU-only throughput.

Another problem of scaling to multiple GPUs is that of real-time scheduling of data transfers. In systems with one CPU and one GPU, the data transfers take non-intersecting paths, and have marginal effect on each other. With multiple GPUs, however, the data transfers have intersecting paths on the system interconnect, and slow each other down. In Chapter 6, we analyzed the interconnect bandwidth in multi-GPU systems and identified the following trends:

1. The computational power of GPUs grows faster than the data transfer bandwidth.
2. As the number of GPUs in the system increases, the bandwidth per GPU decreases.

We therefore expect that limited bandwidth will continue to be a major problem in multi-GPU systems. Emerging inter-GPU communication technologies such as NVLink [Fol14] will likely offset the computation/communication ratio, but we expect the gap will continue to grow. Our experiments show that running additional data transfers concurrently can increase the aggregate data transfer rate, but decrease the transfer rate of each individual data transfer; this performance behavior is reminiscent of the sub-linear throughput scaling observed in GPUs. From a scheduling point of view, the execution time of a data transfer (e.g., CPU1 → GPU1) depends on whether it shares bandwidth with other data transfers (e.g., CPU1 → GPU2) during all or even some of its execution. This presents a new and very challenging scheduling problem. In order to handle it, we presented a new method for scheduling data transfers in RTSPSs with multiple CUs. The method employs a two-level scheduling hierarchy. The top level schedules compound data transfer jobs, i.e., batch operations, to execute one at a time, while the bottom level schedules individual data transfers within a batch operation. This hierarchy achieves both effective scheduling and timing analysis at the task level, along with efficient utilization of the interconnect at the level of individual data transfer operations. Our scheduler achieved up to 7.9 times better results than the bandwidth allocation method (execution time) and up to 39% better results than the time division method (higher image resolution) in realistic applications.

In Chapter 7, we investigated sub-linear performance scaling in multicore CPUs. The operating frequency of modern CPUs is limited by power consumption and thermal considerations. Most multicore CPUs have a frequency scaling mechanism such as Intel’s Turbo Boost that opportunistically increases the operating frequency of the CPU when some of the cores are idle and does not consume their power and thermal budget. For example, the frequency of a Core i7-4790T quad-core CPU when all of its cores are active is limited by 3.3GHz, but when three of its cores are idle it can reach frequencies of up to 3.9GHz. Again, executing multiple tasks (relatively) increases the execution time of each task and decreases the average time per
task. We presented a new offline scheduler that minimizes the total execution time of tasks on a multicore CPU with Turbo Boost. The scheduler adapts the work distribution to the Turbo frequency specification of the target CPU and achieves near-optimal performance. Using our characterization of core speeds in multicore with Turbo Boost, we extended Amdahl’s law, which serves as a guideline to assess the speedup and energy improvement of parallel programs over sequential ones. Amdahl’s law assumes that the performance of the processing units is fixed, which is not the case for the cores in a CPU with Turbo Boost. Our refined formulas for speedup and energy savings provide tight bounds, improving the estimation accuracy by up to 16%.

Our observations regarding performance scaling in modern computer resources encouraged us to develop a new abstract multiprocessor model to be used in theoretical scheduling problems, where processor speeds depend on the execution state of the multiprocessor. In Chapter 7, we extended Graham’s $\alpha|\beta|\gamma$ classification of scheduling problems [GLLRK79] to a new class of problems where the computing platform is characterized by the new model. These new problems model many hardware resources more accurately and enable higher utilization.

**Future Work**

**Task Scheduling**

As computing platforms become more heterogeneous and complex, effective resource sharing becomes increasingly important. We have shown that the models traditionally used in task scheduling problems inadequately represent modern computing platforms and limit their effective utilization. As a result, even scheduling algorithms that are optimal under the traditional model may be inefficient on real hardware. Our analysis of modern hardware resources revealed performance dependencies between components that have traditionally been considered independent, and we presented a new model that represents these dependencies and enables higher utilization. Therefore, many of the scheduling problems need to be reconsidered with the new model. In Section 7.2 we defined a new class of important theoretical task scheduling problems that use the new model. Studying these problems is a promising new research direction. Our analysis of one such problem, makespan minimization in multicore CPUs with Turbo Boost, showed that many of the traditional techniques for scheduling and timing analysis are inadequate under the new model, and new approaches to these problems need to be developed.

**Applications**

In this dissertation, we considered a specific model for a real-time stream processing system (RTSPS) that was defined in Chapter 3. Some RTSPSs have different processing requirements, hardware, and mode of operation, which would require reexamination of the proposed methods and the underlying theory. These differences may include the following:

- **Multi-stream processing operations.** In our model, each stream is processed independently of other streams, while in some stream processing operations, such as *intersect*...
and join, the processing operation is applied on multiple streams. Other characteristics of the processing requirements of a data stream may also differ.

- **Packet arrival and processing requirements.** We have used the sporadic model to describe the real-time data streams. A different model may be more appropriate in systems with aperiodic data arrival and soft, varying, or unknown deadlines.

- **Hardware topology.** We considered a computational platform with a distributed topology, while an integrated design may be more suitable for low-power and embedded systems.

**Real-time Communication Scheduling**

The traditional communication scheduling methods that are used in real-time systems are not efficient on many modern interconnects, because they are based on models that consider simple topologies and avoid bus contention issues. We presented a new model that allows the scheduling algorithm to achieve more efficient utilization of the bandwidth, and a new scheduler for real-time data transfers that uses it. While our scheduler significantly increased the throughput relative to previous methods, further analysis is required to develop optimal scheduling algorithms and communication protocols under the new model.

**Making Use of New Features and Technologies**

Recent technological advances in GPUs and interconnection technologies provide new opportunities to improve data stream scheduling:

- Modern GPUs can execute multiple kernels concurrently. Using this feature it is possible to execute smaller batches of work asynchronously on the same GPU. The challenges of this direction include adapting the performance model, partitioning the streams into smaller batches, and scheduling them (taking their mutual influence into consideration, just as we did with data transfers).

- Using persistent kernels, it would be interesting to consider a pull-based work distribution mode, where each GPU gets work from a pool of tasks shared by CPUs and GPUs.

- NVLink is expected to influence data transfer scheduling. First, the bandwidth bottlenecks in the interconnect are likely to shift. Second, it connects GPUs in different PCIe Express tree branches, hence creating alternative paths for communication between endpoints. While our interconnect model effectively adapts to different topologies, the change may require modifications to the data transfer scheduling methods.

- Virtualization support in GPUs may enable finer resource management capabilities for the scheduler, which could be used to achieve higher utilization and more accurate execution time prediction.
Bibliography


Alexis Mather. Tech brief: AMD FirePro SDI-Link and AMD DirectGMA technology, 2011.


Mike Mantor and Mike Houston. AMD Graphics Core Next. *AMD Fusion Developer Summit*, 2011.


NVIDIA Corporation. NVIDIA Tegra K1 whitepaper. 2014.

NVIDIA Corporation. CUDA C programming guide (v7.0). (March), 2015.


PCI-SIG. PCI Express 3.0 base specification. 2010.


הוחלה ב阚שוד מוגדו עיני ירחו וגרנוק געיית תجزيرة ומימונה. ה阚שוד ו阚שוד
מגורשי פושט במודל הביצועים המוקדש, אור במדיום היהjuvenניה התنجاح הקInSeconds(peer entities. ה阚שוד ו阚שוד
אוליגרמות ה阚שוד על ה阚שוד של האפסטאוליג של עבר שולש מחוז איברה וה阚שוד של העם. בנ.EXIT
שעוני האוליגרמות ה阚שוד שערית ביתהלקוף. ה阚שוד ואיבריו מחיית המעמד, הרחוב וה阚שוד
אוצרות וה阚שוד לשינר בריכת אחר עיבוד מירבי ליבת הם
ה阚שוד של ה阚שוד של שערית בית הלקוף ה阚שוד לשינרה במקבלי תכנית מדרית מבצע מירבה ליבת. ביצועים שערית, הנסהוות של
גבל ועלון פצך של שערית מתקהל בתוכנית מחובך מבית הלקוף מודרני ליבת. הנסיון של
יוד-ב%61 ה阚שוד של חדש השבโชค אודיות אלקוד המלקוי.

האבחנות שלל הביצועים של שענדיב מירבי ליבת מโดרניר עוזרו אחר לעת הפיחות.FileWriter.
מוערת מדורב מתבגרת שערית בית הלקוף. במדיום וה阚שוד הוא הליך במדים של关税 מתבגרת למברית
ה阚שוד וה阚שוד. הרחוב הוא שערית העם המוקדש של גמבית שעל השל חמור של עתיות שערית
במודל החזק שערית. עתיות של מוקדש משאבי חמור ברובו מודיקת אור ואופישים שלגלים מלב
יוד שלום.
RTSPS and its applications in modern multimedia systems, with a focus on GPU and CPU cooperation. Effective scheduling and load balancing strategies are essential for maximizing the performance of multimedia applications. The thesis presents a new algorithm for load balancing between CPU and GPU, demonstrating significant improvements in processing efficiency.

The main contribution of this work is the development of a novel scheduling algorithm that efficiently manages the workload distribution between CPU and GPU. This algorithm considers the characteristics of both processors, including their computational capabilities and communication latencies, to achieve optimal performance.

The evaluation of the proposed algorithm shows that it outperforms existing methods by up to 30% in terms of processing efficiency, particularly in scenarios with high computational demands. The algorithm is designed to be adaptable to various multimedia applications, including video processing and virtual reality, where efficient resource utilization is critical.

In conclusion, this thesis provides a significant contribution to the field of multimedia processing, particularly in the context of GPU and CPU cooperation. The developed algorithm offers a practical solution for improving the performance of multimedia applications, thereby enhancing user experience and system efficiency.
A summary of the digital information that has been growing year by year and is growing in a way that we estimate it. The information includes text, but also between them, we need to analyze that it is required to be processed. Due to the large volume of data and its processing, we tend to relate to the information as a zero. There is a need to work with the digital data as it arrives, and not keep it in the form of storage, but to use it in the future. The calculation systems that are used with the arrival of the information, and no other, can be related to the growth of the GPS phones and computers, which are able to send video, sound, and coordinate locations at the same time, and for the interactions that increase with the user in the Internet environment, which makes it possible for the user to perform almost every action on the site. In addition to the need to analyze the digital information as it arrives, there are limitations in saving the data, in some applications there are limitations on the processing time. In them, interactive applications (for example, to limit the conversation of the phone video ( ), interactive applications of time and location (in medical devices, in creation, in thermometers and so on ( ), applications that are bound to the page in the needs of the users for the purpose of the execution of the applications, as well as ( ) ( ( ( ( ( GPS ). (RTSPS ).

The goal of the work was to develop a way to handle the digital information in real-time on platforms that are heterogeneous and complex. Each flow of information is accompanied by a continuous stream of data packages that are received in a continuous and random manner, so that the time is minimal. GPU and CPU on the processor are used. The common public that enters them is handled by the processor and the network, which is also handled by the processor and the network. In addition, intertwined packages are created. They are used for processing separate flow of data, there is a need to perform a deep analysis of the processing time and verify whether the system is ready to handle all limitations in the time of the construction.

The purpose of this work is to develop an understanding of the digital information in real-time on platforms that are heterogeneous and complex. Each flow of information is accompanied by a continuous stream of data packages that are received in a continuous and random manner, so that the time is minimal. GPU and CPU on the processor are used. The common public that enters them is handled by the processor and the network, which is also handled by the processor and the network. In addition, intertwined packages are created. They are used for processing separate flow of data, there is a need to perform a deep analysis of the processing time and verify whether the system is ready to handle all limitations in the time of the construction.

The purpose of this work is to develop a way to handle the digital information in real-time on platforms that are heterogeneous and complex. Each flow of information is accompanied by a continuous stream of data packages that are received in a continuous and random manner, so that the time is minimal. GPU and CPU on the processor are used. The common public that enters them is handled by the processor and the network, which is also handled by the processor and the network. In addition, intertwined packages are created. They are used for processing separate flow of data, there is a need to perform a deep analysis of the processing time and verify whether the system is ready to handle all limitations in the time of the construction.

The purpose of this work is to develop a way to handle the digital information in real-time on platforms that are heterogeneous and complex. Each flow of information is accompanied by a continuous stream of data packages that are received in a continuous and random manner, so that the time is minimal. GPU and CPU on the processor are used. The common public that enters them is handled by the processor and the network, which is also handled by the processor and the network. In addition, intertwined packages are created. They are used for processing separate flow of data, there is a need to perform a deep analysis of the processing time and verify whether the system is ready to handle all limitations in the time of the construction.

The purpose of this work is to develop a way to handle the digital information in real-time on platforms that are heterogeneous and complex. Each flow of information is accompanied by a continuous stream of data packages that are received in a continuous and random manner, so that the time is minimal. GPU and CPU on the processor are used. The common public that enters them is handled by the processor and the network, which is also handled by the processor and the network. In addition, intertwined packages are created. They are used for processing separate flow of data, there is a need to perform a deep analysis of the processing time and verify whether the system is ready to handle all limitations in the time of the construction.


בעיבוד זרמי מידעי בודק אמת במטריקות מערכות

GPU

היבר על מחקר

לשם מולי חלפי של הרצאות לקבוצת המhoa

דוקטור לפיליפט

אורי גורן

וגן לוסט הפסניכן — מכון מדגולי ליישראלי

טכניון - מכון טכנולוגי לישראל

ועבר联系电话

2015 דצמבר
עיבוד ורמי מידעי בודקי אמת במערכות מבוססות GPU

אורי ורנר