Programmer-Transparent Efficient Parallelism with Skeletons

Paul Metzger

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2021

Abstract

Parallel and heterogeneous systems are ubiquitous. Unfortunately, both require significant complexity at the software level to the detriment of programmer productivity. To produce correct and efficient code programmers not only have to manage synchronisation and communication but also be aware of low-level hardware details. It is foreseeable that the problem is becoming worse because systems are increasingly parallel and heterogeneous.

Building on earlier work, this thesis further investigates the contribution which algorithmic skeletons can make towards solving this problem. Skeletons are high-level abstractions for typical parallel computations. They hide low-level hardware details from programmers and, in addition, encode information about the computations that they implement, which runtime systems and library developers can use for automatic optimisations. We present two novel case studies in this respect.

First, we provide scheduling flexibility on heterogeneous CPU + GPU systems in a programmer transparent way similar to the freedom OS schedulers have on CPUs. Thanks to the high-level nature of skeletons we automatically switch between CPU and GPU implementations of kernels and use semantic information encoded in skeletons to find execution time points at which switches can occur. In more detail, kernel iteration spaces are processed in slices and migration is considered on a slice-by-slice basis. We show that slice sizes choices that introduce negligible overheads can be learned by predictive models. We show that in a simple deployment scenario mid-kernel migration achieves speedups of up to 1.30x and 1.08x on average. Our mechanism introduces negligible overheads of 2.34% if a kernel does not actually migrate.

Second, we propose skeletons to simplify the programming of parallel hard real-time systems. We combine information encoded in task farms with real-time systems user code analysis to automatically choose thread counts and an optimisation parameter related to farm internal communication. Both parameters are chosen so that real-time deadlines are met with minimum resource usage. We show that our approach achieves 1.22x speedup over unoptimised code, selects the best parameter settings in 83% of cases, and never chooses parameters that cause deadline misses.

Lay Summary

For decades computer programs have benefited from processor improvements without requiring modification. The interworkings between programs and processors was carefully engineered so that technological improvements in processor designs benefited programs without requiring changes or the awareness of programmers. This has changed about 15 years ago when processors could no longer be improved in this way. As a result, programmers must deal with significantly more complexity since then. To reduce the costs of software development and maintenance this new complexity must be hidden from ordinary application programmers. One way to achieve this is to package complex processor specific code, which has been produced by specialists, into reusable and configurable components that provide commonly needed functionality. Such components are so called *skeletons* and have been the subject of research since the late 80s. This thesis presents two novel additions to the body of knowledge on skeletons.

Firstly, we show that techniques for the orchestration of a new fine-grained resource management ability can be packaged into skeletons. Without intervention by application programmers, our skeleton makes resource choices, manages data movement, and even automates internal configuration decisions, which are difficult to make manually, through machine learning. Programs with this capability execute up to 1.30x faster (1.08x on average) and our skeleton reduces program size by at least 88%, which benefits development and maintenance costs.

Secondly, we show that skeletons can simplify the programming of so called hard real-time systems, which have to process inputs within a set time and violations of this requirement are unacceptable. For example, robotics applications in which deadline misses might lead to injuries. In the context of these systems, we show that skeletons can automate configuration choices that would have to be made manually by application programmers otherwise. Our skeleton chooses in 83% of cases the best configurations and never makes choices that would cause inputs to be processed late.

Acknowledgements

I would like to thank my supervisors Murray Cole, Christian Fensch, and Volker Seeker for their advice and for teaching me how to work as a researcher. Thanks to Volker for helping me with the implementation of two benchmark applications for Chapter 3 and for the discussion on how we could better generate OpenMP implementations in the future (see Section 5.3.1).

Thanks to Chris Cummins and Hugh Leather for discussions at the early stages of the work in Chapter 3 and during the annual reviews. Thanks to Enrico Bini and Marco Aldinucci of the University of Turin for the collaboration on the work in Chapter 4, and also thanks to XMOS and Paul Neil from XMOS for their donation of the evaluation platform of that chapter.

Finally, I also would like to thank my parents, grandparents, partner, and friends for their support during my PhD.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text or by acknowledgment, and that this work has not been submitted for any other degree or professional qualification except as specified. Parts of this thesis have been published in the following papers:

Paul Metzger, Murray Cole, Christian Fensch, Marco Aldinucci, Enrico Bini, "Enforcing Deadlines for Skeleton-based Parallel Programming", In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020

Paul Metzger, Volker Seeker, Christian Fensch, Murray Cole, "Device Hopping: Transparent Mid-Kernel Runtime Switching for Heterogeneous Systems", In: *Transactions on Architecture and Code Optimization (TACO), 2021.*

(*Paul Metzger*)

Table of Contents

1 Introduction **11**

1.1 The Problem: Dealing with Hardware Complexity in Software 11

1.2 A Solution: Abstraction with Algorithmic Skeletons 14

1.3 Contributions 15

1.4 Thesis Outline 16

1.5 Summary 17

2 Background and Related Work **19**

2.1 Algorithmic Skeletons 21

2.1.1 Recent Skeleton Works 24

2.2 Alternative Parallel Programming Frameworks and Abstractions . . . 27

2.2.1 Other High-Level Abstractions 27

2.2.2 Mid-Level Abstractions 28

2.2.3 Low-Level Abstractions 30

2.3 Thread and Process Migration 31

2.3.1 Migration Mechanisms for CPUs 31

2.3.2 Applications of Migration on CPUs 31

2.4 Heterogeneous Systems 32

2.4.1 General Purpose Graphics Processing Units 32

2.4.2 Related Work 36

2.5 Hard Real-Time Systems 40

2.5.1 Terminology 41

2.5.2 Task Farms for Real-Time Systems 41

2.5.3 Related Work 42

2.6 Critical Reflections on Related Work 45

2.6.1 Mid-Kernel Migration 45

2.6.2 Hard Real-Time Skeletons 46

2.7 Summary 47

3 Transparent Kernel Migration **49**

3.1 Introduction 49

3.2 Motivation for Mid-Kernel Migration 52

3.2.1 Better Performance with Mid-Kernel Migration 53

3.2.2 Simplified Scheduling Decisions 53

3.3 Mid-Kernel Migration 54

3.3.1 Iteration Space Slicing, Runtime Switching, and Slicing Aware Data Transfers 54

3.3.2 Migration Strategies 57

3.3.3 Interference Reduction and Earlier Aborts 58

3.3.4 Device Setup Cost Reduction 59

3.4 Choosing Slice Sizes 60

3.4.1 Target Slice Sizes 60

3.4.2 Application Kernel Features for the Slice Size Predictors . . . 61

3.4.3 Training the Slice Size Predictors 62

3.4.4 Deploying the Slice Size Predictors 63

3.4.5 Choosing Slice Sizes for Sparse Matrix Vector Multiplication 64

3.5 Our High-Level Programming Model 64

3.6 An Idealised Performance Model 67

3.6.1 Simplifying Assumptions 67

3.6.2 Our Baseline Comparator System 67

3.6.3 The Scheduler 68

3.6.4 Components of the Model 68

3.6.5 Speedup with Migration over the Perfect Non-Migrating Scheduler 69

3.6.6 Application Kernel and Device Independent Maximum Speedup 71

3.6.7 Speedups with Different k 71

3.6.8 Speedups with Different δ 71

3.7 Evaluation 72

3.7.1 Experimental Setup 72

3.7.2 Experimental Method 73

3.7.3 Speedups Over the Perfect Non-Migrating Scheduler 76

3.7.4 Overheads in the Absence of Migration 79

3.7.5 Code Size Reduction with Parallel_For 80
3.8 Summary 81

4 Autotuning Parallel Hard Real-Time Systems 83
4.1 Introduction 83
4.2 The Case for Job Batching and Self-Adaptation 85
4.2.1 Reduced Core Count via Job Batching 85
4.2.2 Improved Ease of Programming Through Self-Adaptation . . 86
4.3 System Model 87
4.3.1 Jobs, Job Releases, and Deadlines 87
4.3.2 Cores and Batch Size 88
4.3.3 Execution Time 88
4.4 Our Analytical Framework: Analysis of Batch Scheduling 92
4.4.1 Worker Core Count vs. Task Period 92
4.4.2 Job Batch Size vs. Task Deadline 94
4.5 The Peso Library 97
4.5.1 API Concepts 97
4.5.2 Implementation and Internal Communication Overheads . . . 99
4.6 Experimental Setup 99
4.6.1 Evaluation Platform and Methodology 99
4.6.2 Predictability and The Memory System 100
4.6.3 Worst-Case Execution Times 100
4.6.4 Benchmarks 102
4.7 Evaluation 102
4.7.1 Experimental Validation of our Analytical Framework 102
4.7.2 Fewer Cores with Batching and The Effect of Input Sizes . . . 104
4.7.3 Abstraction Layer Overheads 105
4.8 Conclusion 106

5 Conclusion 109
5.1 Contributions 109
5.2 Reflections on Related work 110
5.3 Limitations and Future Work 112
5.3.1 Transparent Kernel Migration 112
5.3.2 Autotuning Parallel Hard Real-Time Systems 118

5.4 Final Remarks 120

Bibliography **123**

Chapter 1

Introduction

"...[T]he Mind makes the particular Ideas, received from particular Objects, to become general; which is done by considering them as they are in the Mind such appearances, separate from all other Existences, and the Circumstances of real Existence, as Time, Place, and any other concomitant Ideas. This is called ***Abstraction****, whereby Ideas, taken from particular Beings, become general Representatives of all of the same Kind; and their Names, general Names, applicable to whatever exists, conformable to such abstract Ideas."* [Loc53]

John Locke, An Essay Concerning Human Understanding

1.1 The Problem: Dealing with Hardware Complexity in Software

For decades, improvements in chip manufacturing processes have steadily increased transistor counts and led to more sophisticated processor designs with each new product generation [Moo98, HP17, EBA+11]. Microarchitectural advancements like out-of-order execution, caches, and deep pipelines were either not visible to programmers by nature or were carefully hidden from programmers behind the ISA. About 15 years ago it became clear that this trend could not continue because of growing thermal issues and difficulties to further improve performance by purely smarter execution of a single instruction stream [EBA+11, HP17]. From the perspective of programmers, a paradigm shift occurred from simple seemingly sequential execution to complex explicit parallel execution on multicores and a heterogeneous set of accelerators. For example, a modern system on chip for smartphones is comprised of, among others,

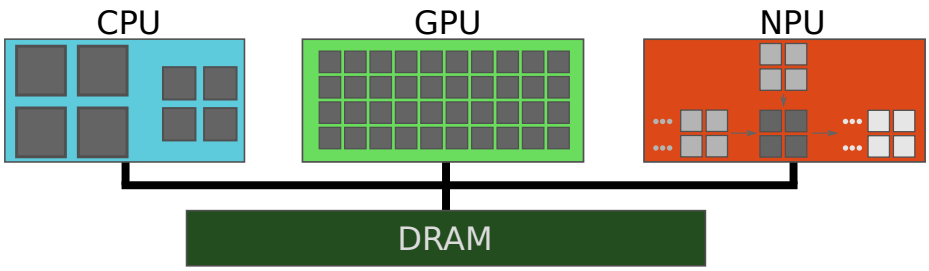


Figure 1.1: Illustration of the heterogeneous set of processing devices in a modern smartphone based on the Huawei Kirin SoC (reproduced from [LTXZ19]). The smartphone is comprised of a CPU, a GPU, and a Neural Processing Unit (NPU) for machine learning tasks. The CPU has two types of cores, energy demanding high performance cores and more energy efficient cores.

two types of multicores, a GPU, and an accelerator for machine learning tasks (see Figure 1.1) [LTXZ19].

As illustrated in Figure 1.2, both multi-core processors and accelerators require software developers to deal with significant complexity [SGG19, LNOM08]. To fully utilise multi-core processors programmers must first identify suitable sources of parallelism and then distribute work over all cores. This includes having to deal with issues such as synchronisation, load balancing, and dead- and live-locks. Additionally, in heterogeneous systems programmers must be aware of low-level details of the underlying architecture to achieve desirable performance. Among other things, programmers must find code sections that are suitable for available accelerators and then provide code that orchestrates the execution on them. It is foreseeable that the problem of increasing programming complexity is becoming worse because over the past decade computing systems have become more and more parallel and heterogeneous.

Architectures that expose parallelism to programmers have existed for a long time. The IBM System 370 of the 1970s, the INMOS Transputer of the 1980s, and the Sony Cell BE of the early 2000s are just a few famous examples [CSG99, GHF+06]. One of the reasons why such systems have not become mainstream for a long time is likely the difficulty of parallel programming. Processors expose parallelism to programmers now but not because of a breakthrough in parallel programming or auto-parallelisation but because a pure focus on single thread performance could not be maintained any longer, as mentioned above. Therefore, to enable mainstream programmers to utilise heterogeneous multicore systems further research on better parallel programming models that hide and manage the complexity of the underlying hardware is crucial.

```
1  __global__ void sum(int *input, int *results, unsigned int n) {
2    // Handle to thread block group
3    cg::thread_block cta = cg::this_thread_block();
4    int *sdata = SharedMemory<T>();
5
6    // Perform first level of reduction
7    unsigned int tid = threadIdx.x;
8    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
9    int mySum = (i < n) ? input[i] : 0;
10   if (i + BLOCK_SIZE < n) mySum += input[i + BLOCK_SIZE];
11   sdata[tid] = mySum;
12   cg::sync(cta);
13
14   // Do reduction in shared mem
15   for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
16     if (tid < s) sdata[tid] = mySum = mySum + sdata[tid + s];
17     cg::sync(cta);
18   }
19
20   cg::thread_block_tile<32> tile32 = cg::tiled_partition<32>(cta);
21   if (cta.thread_rank() < 32) {
22     // Fetch final intermediate sum from 2nd warp
23     if (BLOCK_SIZE >=  64) mySum += sdata[tid + 32];
24     // Reduce final warp using shuffle
25     for (int offset = tile32.size()/2; offset > 0; offset /= 2)
26       mySum += tile32.shfl_down(mySum, offset);
27   }
28   // Write result for this block to global mem
29   if (cta.thread_rank() == 0) results[blockIdx.x] = mySum;
30 }
```

Figure 1.2: Illustration of the complexity introduced by parallelism and low-level programming on GPUs with a simple summation. The equivalent sequential code for a CPU is 7.2x smaller and comprises only five lines of code. Additionally, programmers must be aware of more concepts on the GPU such as scratchpad memory, warps, and synchronisation. The sole purpose of this example is to illustrate the additional complexity on the GPU and so we do not discuss the code in more detail. The code has been taken from the CUDA SDK code samples with minor modifications [nVib].

```
1  ...
2  int result = reduction_skeleton(input, input_size, +);
3  ...
```

Figure 1.3: Illustration of how skeletons are used with a reduction skeleton, which is used to implement a summation. Reductions combine all elements of an input buffer with reduction operators such as +, *, min(), and max(). The skeleton in this example is the function `reduction_skeleton`. The skeleton is parameterised with a pointer to the input buffer `input`, the size of that input buffer, and the reduction operation `+`.

1.2 A Solution: Abstraction with Algorithmic Skeletons

Algorithmic Skeletons are abstract high-level programming constructs for typical parallel computations [Col04]. Examples of typical parallel computations include reductions, stencil computations, and pipelines [MRR12]. Skeletons are reusable implementations of these computations that are, for example, offered to programmers as functions or classes, which are then parameterised with application specific code by the programmer. For example, Figure 1.3 shows an implementation of the summation of Section 1.1 with a reduction skeleton.

Skeletons are a solution to the problem described in the previous section in two ways. Firstly, due to their high-level nature they hide low-level details concerned with parallelism and concerned with details of the target architecture from software developers. For example, the skeleton in Figure 1.3 hides the parallelism and GPU specific details of its implementation in Figure 1.2. Furthermore, the code in Figure 1.2 is only one of many possible implementations. Skeletons can have multiple implementations for different target architectures, thanks to their high-level nature. Secondly, information encoded in skeletons about the computations they implement can be used by runtime systems and specialised library developers for optimisations. For example, the reduction skeleton encodes information about the access pattern in the input buffer and the way input elements can be combined. The input buffer is accessed with no stride and two input elements can be combined independent of the other elements. This information can be used to parallelise the reduction and coalesce memory accesses on GPUs, as is done in Figure 1.2. We exploit these characteristics in the contributions of this thesis (see below). Section 2.1 provides a more detailed introduction to skeletons.

1.3 Contributions

Section 1.1 described how technological changes at the hardware level introduced complexity at the software level. The previous section argued for skeletons as a way to hide this complexity and, at the same time, use them for efficient implementations. This thesis demonstrates these benefits in two, for skeletons, novel contexts. Firstly, we present a parallel_for skeleton capable of mid-kernel migration in heterogeneous systems. Secondly, we present a self-tuning task farm skeleton for parallel hard real-time systems. Both case studies are discussed in more detail below.

Transparent Mid-Kernel Migration with Parallel_For In this case study we are the first to provide fine-grained migration flexibility on heterogeneous systems in a programmer transparent way. We allow computational kernels to migrate and switch the underlying language runtime, for example, from CUDA to OpenMP and vice versa while they execute. We call this mid-kernel migration. In contrast, state-of-the-art systems can schedule kernels only on a kernel-by-kernel basis [RVKP19, PS16].

Conventional runtime systems and operating systems can freely migrate computations on multi-cores. Adding similar flexibility to heterogeneous systems is intricate and requires intimate knowledge of the applications in question. We automate this with the parallel_for skeleton and a novel extension to it for the description of memory access patterns. Access pattern descriptions are used for a new optimisation that allows for an efficient implementation of the migration mechanism. We also show that choices for a tuning parameter of this mechanism, which has to be set on a kernel-by-kernel basis, can be learned by predictive models and hidden behind a skeleton interface.

We show with analytical models that mid-kernel migration can outperform kernel-by-kernel scheduling by up to 1.33x in our simple deployment scenario and confirm this experimentally. In addition, we show that our skeleton interface reduces code size by at least 88% compared to a hand implementation of mid-kernel migration. This work is presented in Chapter 3.

Self-Tuning Skeletons for Hard Real-Time Systems In the second case study, we are the first to present a timing-predictable and fully self-tuning skeleton for hard real-time systems. This is a first step towards a set of composable self-tuning hard real-time skeletons.

Real-time systems programmers must determine the minimum degree of paral-

lelism required to meet real-time timing requirements and for optimal resource usage. This is even more challenging than on general purpose systems, because for hard real-time system any violations of timing requirements are unacceptable. Additionally, communication overheads can be reduced with careful use of knowledge about the target microarchitecture and application. Using the concept of task farms and knowledge about the independence of tasks encoded in the task farm skeleton we address this challenge. We automatically choose the minimum degree of parallelism that is required to process inputs in time and, additionally, we demonstrate that structural information encoded in skeletons can be used for optimisation in hard real-time systems. For this we automatically choose a tuning parameter for an inter-thread communication optimisation. Both self-tuning choices are made with carefully constructed models of the response time of our task farm. The complexity of this optimisation and thread management are hidden behind the skeleton.

We validate our models and implementation by showing that they never choose parameters that would cause inputs to be processed too late. We also show that our models choose either optimal or close to optimal parameters. This work is presented in Chapter 4.

1.4 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 provides necessary background knowledge and discusses related work. Algorithmic skeletons and alternative abstractions are introduced at the start. The chapter then discusses thread migration on CPUs because Chapter 3 brings similar flexibility to heterogeneous systems, which are discussed subsequently with a focus on CPU + GPU systems. Lastly, hard real-time systems and the real-time systems terminology, which are necessary for Chapter 4, are introduced.

Chapter 3 presents our work on programmer transparent migration for heterogeneous systems. We enable computational kernels to migrate between heterogeneous devices mid-execution without programmer intervention. We use semantic information encoded in skeletons to find points at which the state of a kernel is stable and can be migrated to another device. Furthermore, the complexity of this is hidden thanks to the high-level nature of skeletons. Please note that this chapter does not consider

real-time systems. We present our real-time systems work in the following chapter.

Chapter 4 argues for the use of skeletons to program parallel hard real-time systems. In a case study with the task farm skeleton, we show that information encoded in this skeleton can be used to autotune applications. In more detail, we automatically make complex static scheduling decisions and choose the degree of parallelism without programmer intervention. Programmers only have to provide sequential code because the complexity of the parallel execution is again hidden behind a skeleton.

Chapter 5 concludes the thesis and discusses limitations and future work.

1.5 Summary

This chapter outlined the problems of increasing hardware complexity that is visible at the software level. A potential remedy is already ubiquitous in computer science: abstraction. Algorithmic skeletons have been shown to simplify parallel programming by abstracting away low-level implementation details. This thesis presents two novel case studies in which the high-level nature of skeletons simplifies the programming of parallel systems. In more detail, we are the first to show that skeletons can provide similar scheduling flexibility on heterogeneous systems to that which OS schedulers already have on CPUs, and this in a way that is transparent to programmers. We are also the first to present a fully self-tuning and timing-predictable hard real-time skeleton.

Chapter 2

Background and Related Work

Many abstractions for parallel programming have been devised over the past decades. They fall along a spectrum from high-level to low-level abstractions. High-level abstractions make parallelism implicit and, therefore, like with skeletons, anything parallelism related is hidden from application developers. These abstractions give the most freedom to libraries and runtime systems in terms of implementation choices. With mid-level abstractions some details of the parallel execution are hidden from programmers. However, crucial performance relevant decisions are baked into the source code. As a result, applications cannot adapt dynamically to changes in the execution environment and compilers have limited freedom for autotuning. Also, high-level application information as it is encoded in, for example, skeletons is not available. Low-level frameworks leave all implementation details to the programmer and provide only a thin abstraction layer over the hardware.

In this chapter we first introduce skeletons and, following this, other common parallel programming models of different abstraction levels are discussed. Next, we discuss thread migration on CPUs in Section 2.3 and heterogeneous systems in Section 2.4. These sections are relevant to our work on fine-grained migration on heterogeneous systems in Chapter 3. Lastly, to provide the necessary background knowledge for Chapter 4, Section 2.5 introduces real-time systems and the corresponding terminology. Recent work on skeletons and other related work are discussed at the end of these subsections after an introduction to the relevant background material. Finally, we conclude this chapter with critical reflections on previous work that is most relevant to the main chapters of this thesis.

```
1  sum = reduce(map(abs, in_buf), +, 0)
```

(a) High-level implementation of *sum of absolute values* with *map* and *reduce* skeletons.

```
1  int  sum = 0;
2  for (size_t i = 0; i < IN_BUF_SIZE / 16; ++i) {
3    __m512i temp = _mm512_load_si512(in_buf);      // Load 16 ints
4    temp         = _mm512_abs_epi32(temp);         // Apply abs()
5    sum         += _mm512_reduce_add_epi32(temp); // Partial reduce
6    in_buf      += 16;}
```

(b) Low-level single threaded SIMD implementation with C of the computation in (a).

```
1  int sum = 0; int partial_sum = 0;
2  #pragma omp parallel firstprivate(partial_sum, in_buf) shared(sum)
3  {
4    #pragma omp for schedule(static)
5    for (size_t i = 0; i < IN_BUF_SIZE / 16; ++i) {
6      __m512i temp = _mm512_load_si512(in_buf + 16 * i);
7      temp         = _mm512_abs_epi32(temp);
8      partial_sum += _mm512_reduce_add_epi32(temp);
9    }
10   #pragma omp atomic
11   sum += partial_sum;}
```

(c) Low-level multithreaded SIMD implementation with C and OpenMP of the computation in (a). Partial sums are computed in each thread and then combined with atomic operations.

Figure 2.1: Illustration of how skeletons are used and the flexibility they provide to runtime systems and library developers. (a) shows the skeleton implementation of a kernel that computes a sum of absolute values. (b) and (c) show potential corresponding low-level implementations that a library could provide and from which a runtime system could choose with a high-level programming model like skeletons. Alternatively, the reduction component in (b) and (c) could be implemented with an OpenMP reduction. Compilers can also generate vectorised code, either automatically or guided by programmers, for example, with the SIMD pragma of OpenMP. SIMD operations are shown in (b) and (c) to illustrate how low-level operations can be hidden from programmers behind a skeleton interface in (a), which simplifies programming. Whether the CPU implementations in (b) and (c) or a GPU implementation similar to the one in Figure 1.2 is best depends on the relative performance of CPU and GPU and where the input buffer resides. Additionally, the single threaded implementation in (a) is likely better than the multithreaded implementation in (b) for small input buffers. A runtime system could take these factors into account and choose the best implementation from a set of implementations.

2.1 Algorithmic Skeletons

Algorithmic skeletons are high-level parallel programming constructs, which implement common patterns of parallel computation and communication [Col04]. To build applications, skeletons are composed and parameterised with application specific code. Skeletons are an attractive programming model for two reasons. Programmers

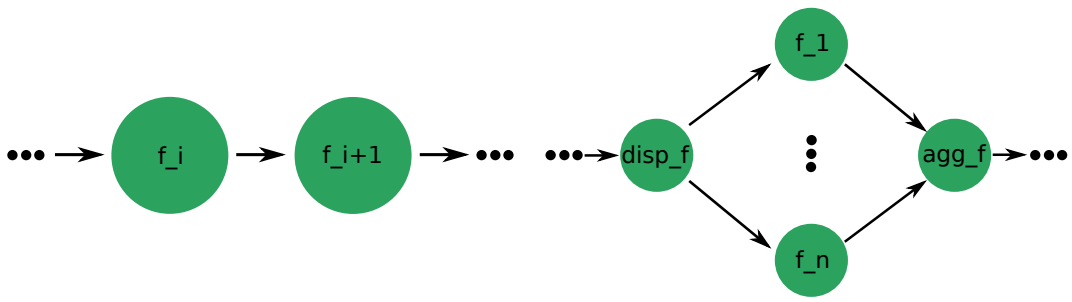
Table 2.1: Examples of typical data parallel skeletons. The first three skeletons compute new results by applying the application specific user function `f` to input data. Simple data parallel skeletons like these are sometimes combined with skeletons like `scatter` and `zip`, that do not compute new results but rearrange data.

Skeleton function	Description	Sequential pseudo-code
`map(f, in, out)`	Applies the user function `f` to each element in the input buffer. In- and output buffer can be the same for in-place updates.	`for i in 1, in.size:` `  out[i] = f(in[i])`
`parallelFor(` `  f, r, ...)`	Calls the user function `f` $r.end - r.start$ times, with the iteration count and other parameters passed to `parallelFor`.	`for i in r.start, r.end:` `  f(i, ...)`

`reduce(in, f,`
`init)`
	Folds elements in the input buffer with the user function. Typical user functions are `+`, `-`, `max`, and `min`.	`result = init`
		`for i in 1, in.size:`
		`  f(result, in[i])`
		`return result`

`scatter(`
`  out, in, ind)`
	The skeleton reorganises data. Elements in `in` are placed in `out` at the indices in `ind`.	`result = []`
		`for i in 1, indices.size:`
		`  out[ind[i]] = in[i]`

`zip(in1, in2)`	Creates a list of tuples. The ith tuple contains `in1[i]` and `in2[i]`.	`result = []`
		`assert in1.size == in2.size`
		`for i in 1, in1.size:`
		`  result[i] =`
		`  (in1[i], in2[i])`
		`return result`



(a) Illustration of the pipeline skeleton. (b) Illustration of the task farm skeleton.

Figure 2.2: Examples of task parallel skeletons based on the task farm and pipeline skeletons of FastFlow [ADKT17]. Both skeletons process a continuous stream of input data. Circles are functions that a compiler or runtime system can choose to execute in parallel and arrows indicate the data flow between them. Pipelines apply user functions consecutively (here `f_i` and `f_i+1`) to input data. Each user function is called a pipeline stage, and stages can be executed in parallel. Task farms distribute input data via the `dispatcher` (here `disp_f`) to concurrent worker functions (here `f_1` to `f_n`), which are provided by the programmer. Task farms can also run multiple instances of the same worker function in parallel (not shown), which is helpful if the time required to process an input is longer than the time between arrivals of new inputs. We exploit this in Chapter 4. An `aggregator` function (here `agg_f`) collects the results and passes them on to the rest of the program.

```
1  read_input():
2    // Read input data e.g. via I/O
3    ...
4    send_to_next_stage(data)
5
6  process():
7    data ← receive_from_prev_stage()
8    // Process the data
9    ...
10   send_to_next_stage(data)
11 write_result():
12   data ← receive_from_prev_stage()
13   // Handle results e.g. write to I/O
14   ...
15
16 main()
17   p ← init empty pipeline
18   p.add_stage(read_input)
19   p.add_stage(process)
20   p.add_stage(write_result)
21   p.run()
```

Figure 2.3: Pseudocode illustration of pipeline skeletons, which are task parallel, based on the API of FastFlow [ADKT17]. This skeleton instance is composed of three pipeline stages. Again, a runtime system could choose whether to run them in parallel.

only specify the computations to be performed but not how they are implemented, which gives libraries and runtime systems the freedom to choose among different implementations and target devices, as has been demonstrated previously [SRD17, DLK13, LFC13]. At the same time, they encode high-level semantic information about memory access patterns and other properties, such as the independence of tasks, which can be exploited by library programmers and runtime systems for efficient implementations. Figure 2.1 illustrates how skeletons enable implementation choices and abstract complexity with a simple example. Figure 2.1a shows an implementation of a computational kernel using two skeletons and figures 2.1b and 2.1c show corresponding hand implementations. These are significantly more complex and leave less freedom to runtime systems and compilers. For example, if the implementation in Figure 2.1b was given by the programmer, runtime systems or compilers could not choose to run the kernel on a GPU.

Skeletons can be categorised into data parallel and task parallel skeletons. The distinction is based on the broader concepts of *data parallelism* and *task parallelism*. Hennessy et al. and Culler et al. define these as follows:

- "Data-level parallelism (DLP) arises because there are many data items that can be operated on at the same time." [HP17].

- "In addition to data parallelism, applications often exhibit function parallelism as well: entirely different calculations can be performed concurrently on either the same or different data. Function parallelism is often referred to as control parallelism or task parallelism, ..." [CSG99].

Therefore, data parallel skeletons typically process the same input data set in parallel like the `map` and `reduce` skeletons in Figure 2.1. Table 2.1 provides further examples of data parallel skeletons. Task parallel skeletons are concerned with the parallel execution of different user functions. Figure 2.2 illustrates typical task parallel skeletons, and Figure 2.3 their use with the pipeline skeleton. The task farm skeleton, which is illustrated in Figure 2.2, can be used in both ways. Task farms can exploit task parallelism with different user functions as is shown in Figure 2.2 or data parallelism by applying the same worker function in parallel with multiple workers to, for example, different parts of the same input data set.

2.1.1 Recent Skeleton Works

Skeletons have been researched for the past thirty years [GVL10]. We discuss below recent examples, which exploit skeleton information for efficient programmer transparent implementations.

Copperhead generates CUDA code based on data parallel skeletons that are already part of Python and other, for Python, new skeletons [CGK11]. To improve performance programmers can either manually choose among different CUDA implementations of nested skeletons or use an autotuner. As many skeleton frameworks do, the Copperhead compiler uses semantic information encoded in skeletons to generate efficient code. In more detail, the compiler attempts to fuse skeletons to improve locality. For this, skeleton information is used to determine whether successive skeletons require barrier synchronisation between them and to determine the size of temporary buffers. Lastly, the authors show that using skeletons improves code size and therefore readability and maintainability significantly compared to hand implementations in CUDA.

The Distributed Multiloop Language (DMLL) consists of a collection of data parallel skeletons that can be nested [BLR+16]. DMLL targets NUMA systems, clusters with multiple GPUs, and single nodes. The DMLL compiler and runtime system use skeleton knowledge for efficient implementations in two ways. The compiler can rewrite combinations of skeletons with other skeletons to optimise for different targets and to better distribute data in NUMA systems and clusters. For example, the dimension over which operations on matrices are parallelised can be changed this way. Besides that, the runtime system uses skeleton specific knowledge about memory access patterns to improve data locality. To achieve this, the runtime system schedules sub-computations on NUMA and cluster nodes that store the corresponding input data. DMLL is built on top of Delite which is a skeleton-based framework for the implementation of DSLs [SBL+14].

FastFlow is a C++ library for CPUs, GPUs, and clusters with task and data parallel skeletons [ADKT17, ASD+12, ACD+12]. A distinguishing characteristic is that the library and programming interface are specialised for streaming applications, in which a steady stream of new inputs is processed. To process inputs, the code of task farm worker threads and pipeline stages (see below), which are the main abstractions of FastFlow, are invoked continuously. Internally, FastFlow uses queues to implement communication over shared memory between, for example, worker threads and pipeline stages. In addition, abstractions for typical communication patterns in

distributed system are offered [ACD+12]. To ease parallel programming, FastFlow can automatically adjust its resource usage at runtime to meet quality of service targets [DSTD16]. For example, the runtime system can adjust dynamically the number of cores an application uses. Furthermore, almost the complete Parsec benchmark suite has been implemented with FastFlow [DSDMT+17]. With these implementations, the authors show that the skeletons of FastFlow improve code size significantly compared to implementations with lower-level abstractions.

Lift offers programmer transparent performance portability for heterogeneous systems [SRD17, SFLD15]. The Lift language includes data parallel skeletons and skeleton-like functions for typical data layout operations. Skeletons are available at two different abstraction levels. The first level are the typical device independent high-level skeletons, and the second are OpenCL specific low-level skeletons. The second level, for example, offers a map skeleton that maps OpenCL workgroups to buffer elements. To achieve performance portability the Lift compiler generates OpenCL code optimised for the target device from high-level functional skeleton code. For this, semantic information encoded in skeletons is exploited in two ways. The compiler uses repeatedly rewrite rules that substitute high-level skeletons with other skeletons from both abstraction levels. Next, efficient OpenCL code is generated from an intermediate presentation that uses only the low-level skeletons. RISE and ELEVATE are continuations of Lift [KS21, HLK+20]. RISE is, like Lift, a functional programming language with data parallel skeletons and skeletons for data reorganisation, which serves as an intermediate language for the implementation of DSLs. One of the shortcomings of Lift is that it relies on a fixed set of rewrite rules and does not let programmers influence the compiler when it searches the space of possible rewritings to find good implementations of input kernels. ELEVATE, in comparison, lets programmers add new rewrite rules by composing a set of basic rules, and gives programmers more control over how the compiler applies them.

Musket is a C++ extension with data parallel skeletons, which generates code for clusters with multi-GPU nodes [WK20, RWK19]. A source-to-source translator generates standard C++, MPI, CUDA, and OpenACC code. The extensions to C++ are centred around distributed data structures for arrays and matrices, data parallel skeletons, and communication skeletons. An optimiser in the translator automatically fuses skeletons, adapts the implementation of user functions depending on the contexts in which they are called, and automatically distributes data over cluster nodes. The authors emphasise ease of use, which informed the decision to extend C++. C++ is

already known among the target user group, and many mistakes can be caught at compile time or beforehand because C++ is statically typed. In line with this, plug-ins for a common IDE, which implement static syntax checks and code completion, are provided. Musket is a continuation of Muesli, which is a C++ skeleton library for cluster systems [KS02, EK16]. Muesli offers data and task parallel skeletons and distributed data structures, which are very similar to the data structures of Musket. A distinguishing feature of Muesli is support for *currying*. Currying modifies functions by binding some of the functions' arguments to values, which allows programmers to specialise functions before they are passed to skeletons. Muesli has backends for MPI, OpenMP, and CUDA. Also, a Java implementation exists, which, additionally, supports OpenCL.

Partans provides a high-level API for stencil computations and automatically generates efficient multi-GPU implementations [LFC13]. Iterative stencil computations have to regularly copy data between memories in distributed memory systems. Typically, costly data transfers can be traded against redundant computations. Partans not only generates code for these transfers but also finds good trade-offs between transfer frequency and redundant computations without programmer intervention.

SkePU started as a purely macro-based C-library, and has evolved to a C++ library and source-to-source translator with data and task parallel skeletons for which it offers implementations in OpenMP, MPI, CUDA, StarPU, and OpenCL [ELK17, ÖEK19]. Some of its many features are discussed below. The runtime system automatically manages memory coherence between main-memory and DRAM of dedicated GPUs with an MSI-like protocol [DK16]. At the programming interface level, this is achieved with container classes, which, besides the standard C++ container methods, offer further methods for memory coherence. These methods are then used by the skeleton implementations to ensure coherence in a programmer transparent way. In addition, this allows for lazy data movement. A simple implementation might copy results from dedicated GPUs to main memory after each skeleton execution on the GPUs. Instead SkePU only copies data when needed. In more detail, this means that data is not copied to main memory between successive skeleton executions on the GPU. The coherence mechanism works in a fine-grained way at the level of application defined access ranges. If a skeleton reads only the first n elements of a buffer on a GPU, for instance, then the mechanism ensures that only these n elements are up-to-date, and no data transfers are performed for the remaining elements. Additionally, the runtime system can also schedule kernels on heterogeneous systems on a skeleton-by-skeleton basis, including the ability to spread the execution of a skeleton across multiple hetero-

geneous devices [ÖEK19]. Skeletons are executed lazily, which means that they are not executed right away but only when their execution cannot further be deferred [EK18]. Because the runtime system has thus an overview over multiple skeletons and their relationship it can apply optimisations by, for example, fusing them. The source-to-source translator of SkePU supports target device specific user functions [EK20]. In addition to generic user functions for skeletons, programmers can provide functions for specific target architectures or devices. This is helpful if users want to manually optimise their code beyond what SkePU or a C++ compiler can do.

Spark is a continuation of the widely known MapReduce framework for cluster computing [DG08, ZXW+16, ZCF+10]. With MapReduce programmers express their computation with single or multiple successive map and reduce computations. The execution on a cluster is orchestrated transparently to the programmer. One of the design issues of MapReduce, that Spark overcomes, is that the results of each reduction invocation are written to disk, which incurs overheads. To overcome this limitation, skeleton calls can be arbitrarily chained without disk writes. Spark also offers a wider variety of patterns, which are either similar to map and reduce or variations of it.

2.2 Alternative Parallel Programming Frameworks and Abstractions

This section discusses examples of alternative high-, mid-, and low-level parallel programming models and abstractions.

2.2.1 Other High-Level Abstractions

Domain Specific Languages (DSLs) are programming interfaces specialised to applications of a particular field. They can either be stand-alone languages with their respective compilers or be embedded into existing languages. DSLs are very similar to skeletons in some respects because they also encode semantic information, that is typically not available with generalist languages, through high-level constructs specific to their domain. These high-level constructs also abstract low-level implementation details and provide freedom to runtime systems and compilers. In contrast, some skeletons are applicable to multiple domains. For example, the map skeleton is applicable to a wide range of applications including financial analysis, computer vision, computer graphics, and text analytics [DG08, DSDMT+17]. However, other skeletons, such as

stencil skeletons, are applicable to only a few domains. Additionally, skeletons can also be specialised to domains, such as FastFlow's skeletons, which are specialised to streaming applications [ADKT17].

A good example of a DSLs is Halide [RKBA+13], which is a language for image processing pipelines. Programmers provide high-level implementations of kernels, and optionally a separate specification of how they should be implemented. For example, programmers can specify whether kernels should be fused or tiled. Alternatively, if this is not specified then the compiler has the freedom to choose from a wide variety of implementations. In contrast to the work presented in this thesis, Halide is only applicable to a narrow set of applications and does not allow for mid-kernel migration or provide real-time guarantees.

Specialised libraries with simple high-level functions provide performance-tuned implementations of common and potentially time intensive operations. Examples include libraries for linear algebra and computer vision like OpenCV [WWX+16]. The boundaries between DSLs and simple libraries are blurry. However, the programming interface of such libraries is simpler. In general, the interface is not reminiscent of a new language, and it is not implemented with dedicated compilers.

Efficient implementations can be chosen at compile- and runtime, and application programmers are not concerned with implementation details. However, library developers have to provide new implementations for new target platforms and application developers rely on the availability of libraries for their kernels and target systems. Another major issue is that some optimisations such as kernel fusion, which would optimise across multiple function calls, cannot always be applied [RKBA+13]. Again, skeletons and, therefore, the work presented in this thesis are of wider applicability than such libraries.

2.2.2 Mid-Level Abstractions

Bulk Synchronous Programming (BSP) based programs are executed in so called supersteps, which themselves consist of two steps: first computation and asynchronous communication and then barrier synchronisation [Val90]. While this frees programmers from having to implement synchronisation, BSP does not hide other aspects of parallelism such as communication, task granularity, and load balancing.

Pregel is a C++ DSL for graph processing, which requires programs to have a BSP

structure [MAB+10]. In each BSP superstep, vertices perform computations based on their own value and data sent by other vertices in the preceding superstep. Pregel uses the structure that it imposes on applications for efficient fault tolerance. Corrupted data is recomputed based on checkpoints taken between supersteps and recorded messages.

Message passing allows concurrently running threads and processes to communicate and synchronise via messages. Efficient implementations of the messaging infrastructure are abstracted from programmers and left to the compiler and runtime system. However, programmers are exposed to all other details of parallel execution and related performance optimisations. Examples of this paradigm are the Message Passing Interface (MPI) for distributed systems and the POSIX message passing interface for shared memory systems [SGG19].

Task-based programming lets programmers express applications as collections of interdependent tasks. Tasks that are independent at any one time during the execution can be executed in parallel. Some parallelism related issues such as synchronisation between tasks, thread management, and task to device mappings are abstracted away [PS16]. However, task-based programming models still put a bigger burden on application developers because fewer parallelism related implementation details are hidden than with skeletons. For example, a map skeleton does not require programmers to choose the granularity at which computations are assigned to threads internally. However, a manual implementation of a map (see Section 2.1) with tasks would require programmers to decompose the map into tasks and to carefully choose a granularity for the decomposition. Tasks also do not provide any additional information about the code within the tasks to the compiler or runtime system that could be used for efficient implementations. In fact, programmers have to provide multiple task implementations if they want to give the runtime system the freedom to schedule them on different devices [ATNW09, Uni20]. Additionally, applications must be manually decomposed into tasks.

OpenMP, OmpSs, and StarPU are well known implementation of this programming paradigm [ATNW09, Boa20, Bar20, PBAL13]. OpenMP is a mature programming language extension whose task pragma annotations are inspired by OmpSs, which is a research implementation. StarPU and OmpSs can schedule tasks on different devices but require programmers to provide multiple implementations as mentioned above.

Because these systems are scheduled on a task-by-task basis it is either not pos-

sible to migrate kernels between heterogeneous devices if they are implemented as a single task or programmers must manually decompose kernels, which is error prone and time intensive. As discussed in Chapter 3, we provide more scheduling flexibility without requiring programmers to manually decompose kernels or to provide multiple implementations.

2.2.3 Low-Level Abstractions

Threads are sequential instruction streams, which by default share the same address space with other threads that belong to the same process. To implement synchronisation and computation, threading libraries typically offer low-level primitives such as mutexes, semaphores, and simple message queues. These are known to be error prone and difficult to use [SL05, HM93, RCKH09]. Threads convey no information about communication and synchronisation patterns that compilers or runtime systems could use to assist programmers. A widely used thread-based API is *Pthreads* [Ker10].

Single Instruction Multiple Data (SIMD) Intrinsics are functions, which map to SIMD instructions of the underlying hardware. SIMD instructions apply the same operation in parallel to multiple input data elements [HP17]. Intrinsics are close to the actual assembly instructions because programmers work manually with loads and stores and have to be aware of the SIMD register width of the target microarchitecture. Figure 2.1b uses SIMD intrinsics in lines three to five.

Single Instruction Multiple Threads (SIMT) is an abstraction for GPU compute kernels in which, from the point of view of programmers, a single function is executed in parallel by multiple threads but with different input data [HP17, LNOM08]. SIMT is similar to SIMD programming (see above) because the same operations are applied in parallel to different input data. However, among other things, programmers can implement different per thread execution paths like in non-SIMD programs. Popular implementations of SIMT include the CUDA programming framework for GPUs and OpenCL. In practice, implementations of this paradigm encode information about the independence of groups of threads. Besides that, this programming model provides no further information about the computations in the threads that libraries or compilers could use.

2.3 Thread and Process Migration

Operating system schedulers have been the subject of extensive research since the early days of commercial computing [CMDD62]. The ability to migrate threads in mid-execution from one core to another has been a crucial building block to implement a range of scheduling policies. We first discuss the mechanism for migration on CPUs and why it cannot be used for migration between CPUs and GPUs. Following this, we discuss some of the existing policies, which exploit migration, to motivate our work on providing similar programmer transparent flexibility for heterogeneous CPU + GPU systems.

2.3.1 Migration Mechanisms for CPUs

To be able to migrate processes and threads, operating systems have a mechanism to preempt them. When a process is preempted its architectural state, which comprises the program counter and all other registers, is saved to a per process data structure in the operating system, the so called *Process Control Block (PCB)* [SGG19]. To resume execution on another core the OS copies the register values from the PCB and sets the program counter with the saved value.

Migration is possible in this way on mainstream CPUs because all cores have the same instruction set architecture (ISA) and the same, or in principle similar, microarchitectures. In more detail, if a process is resumed on another core the previously saved values can simply be copied to the same registers on the new core, even if the microarchitectures between cores differ as in big.Little systems [CKC12]. However, the ISAs and microarchitectures of CPUs and GPUs are vastly different, and the architectural state recorded on one cannot directly be mapped to the other, let alone in an efficient way.

2.3.2 Applications of Migration on CPUs

Operating systems can preempt threads if others with a higher priority, such as real-time threads, require their CPU cores [SGG19]. The preempted threads can be resumed on other, for example, weaker cores like in big.Little systems. Threads are also migrated for load balancing. For example, Linux and FreeBSD periodically invoke a load balancing function, which, if a load imbalance exists, migrates threads from cores with higher load to cores with lower load [BCL+18]. Additionally, underutilised cores

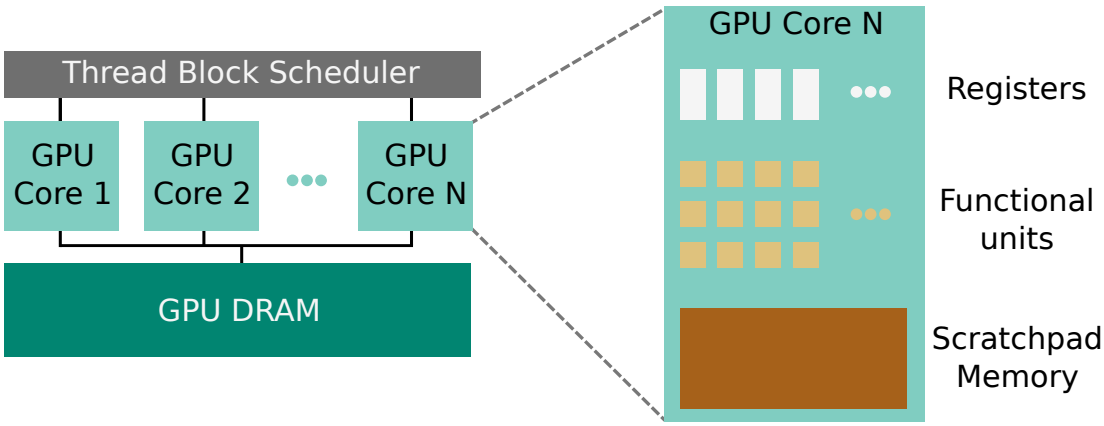


Figure 2.4: High-level illustration of GPU microarchitectures based on nVidia GPUs [LNOM08, HP17]. A GPU consists of multiple cores, which are called streaming multiprocessors (SMs) in CUDA terminology. Each SM is composed of registers, multiple functional units, which operate in lockstep, and scratchpad memory. The block scheduler breaks the iteration space of the kernel down into blocks and schedules them onto the SMs. (This figure is a simplification of the figures by Lindholm et al. [LNOM08].)

can steal threads from other cores to balance load. Lastly, the Energy Aware Scheduler by ARM for the Linux kernel can migrate threads between heterogeneous cores if this reduces energy consumption [kdc20].

2.4 Heterogeneous Systems

Heterogeneous systems are composed of a mix of different processing units like CPUs and GPUs. Some example systems are discussed in Section 1. This section discusses general purpose graphics processing units (GPGPUs) because we use CPU + GPU systems in Chapter 3. We use CUDA terminology in this chapter [LNOM08, HP17].

2.4.1 General Purpose Graphics Processing Units

Figure 2.4 provides a high-level overview of typical GPU microarchitectures. A GPU consists of multiple independent streaming multiprocessors (SMs) which are akin to multi-lane vector processors [LNOM08, HP17]. A thread block scheduler divides the iteration spaces of kernels into blocks and schedules them on the SMs. Each SM consists of several functional units [1] that execute in lockstep. Per SM schedulers in

[1] Note, these functional units are also called CUDA cores.

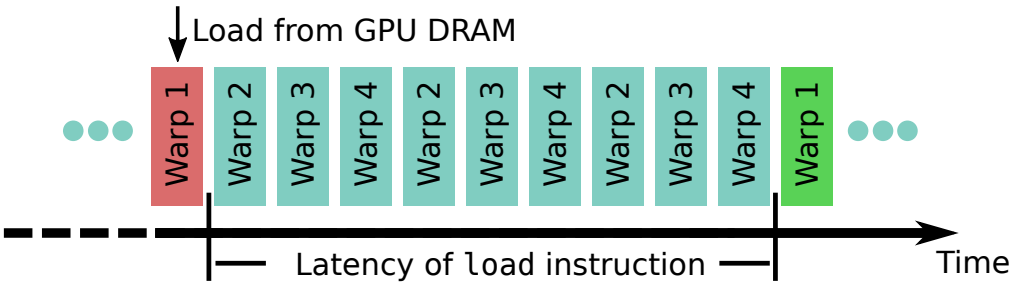


Figure 2.5: Illustration of how GPUs hide the costs of high-latency operations with fine-grained multithreading. In this example, the warp scheduler schedules other warps on the functional units while Warp 1 waits for a long latency load from GPU DRAM to finish.

turn divide blocks into smaller groups of GPU-threads, so called warps, which are then executed in parallel on the functional units (not shown). GPUs use fine-grained multithreading to hide the cost of high-latency instructions such as loads and stores. For example, while a warp waits for a load from GPU DRAM per-SM-schedulers can run other warps on the functional units of an SM, as illustrated in Figure 2.5. Therefore, it is important for GPUs to have multiple concurrent warps per SM to choose from. The ratio of the theoretical maximum number of concurrent warps that the GPU could schedule and the actual number of warps into which the iteration space of a kernel can be divided and which can be run concurrently is called occupancy [LR11]. In general, the higher the occupancy the better because the warp schedulers have more opportunity to overlap high-latency operations with other operations.

2.4.1.1 Programming Frameworks for GPUs

GPUs can be programmed with CUDA or OpenCL, which are implementations of the low-level SIMT programming paradigm discussed in Section 2.2. nVidia offers CUDA for nVidia GPUs and OpenCL is a cross-vendor framework, which supports several device types, including CPUs, GPUs, and FPGAs. From a programmer perspective both are in principle the same and because CUDA is more concise this chapter uses only CUDA. Programmers write kernels, which are executed on the GPU, in a C-like language, and so called host code, which is executed on the CPU to orchestrates data transfers and the execution of kernels on the GPU. Figure 2.6 illustrates this with a simple vector addition kernel. As shown in Figure 2.6b kernels are launched with a special function call. To achieve high performance programmers must produce very low-level CUDA code, which exploits the general microarchitectural principles discussed above. In addition, programmers can use features of particular microarchi-

```
1 __global__ void vector_addition(float* in1, float* in2, float* out) {
2   const unsigned int global_tid = blockIdx.x * blockDim.x + threadIdx.x;
3   out[global_tid] = in1[global_tid] + in2[global_tid];
4 }
```

(a) Kernel code that is executed on the GPU.

```
1 ...
2 // Copy data to the GPU
3 cudaMemcpy(in1_device, in1_host, VEC_SIZE_IN_BYTES, cudaMemcpyHostToDevice);
4 cudaMemcpy(in2_device, in2_host, VEC_SIZE_IN_BYTES, cudaMemcpyHostToDevice);
5
6 // Execute the kernel
7 vector_addition<<<VEC_SIZE / BLOCK_SIZE, BLOCK_SIZE>>>(
8   in1_device,
9   in2_device,
10  out_device);
11
12 // Wait for the execution to finish
13 cudaDeviceSynchronize();
14
15 // Copy results to main memory
16 cudaMemcpy(out_host, out_device, VEC_SIZE_IN_BYTES, cudaMemcpyDeviceToHost);
17 ...
```

(b) Host code that is executed by the CPU.

Figure 2.6: Illustration of how GPUs are programmed with CUDA with a simple vector addition kernel. The kernel function in (a) is called in (b) in line seven. The two parameters within `<<<...>>>` specify the iteration space of the kernel. The parameter on the right-hand side sets the size of the blocks, and the parameter on the left-hand side sets how many blocks are executed. For simplicity, we assume that VEC_SIZE is a multiple of the block size. From a high-level perspective, CUDA launches a separate thread for each point in the iteration space, which each execute the function in (a) once. Error checking code is omitted for brevity.

tectures through intrinsics [nVi21].

Other alternative languages and libraries with a much more high-level programming interface such as DSLs, skeletons, and specialised libraries exist [nVia, RKBA+13, SRD17]. These often use OpenCL or CUDA internally or directly generate assembly

Table 2.2: Typical DRAM and PCI-E bandwidths in a current system. PCI-E data transfers can have significant costs because the PCI-E bandwidth is considerably smaller than the bandwidth of main memory and GPU DRAM. We assume a system with dual channel main memory.

Component	Bandwidth (GB/s)
Main memory (DDR4-2400)	38.4 [HP17]
GPU DRAM (nVidia Titan Z)	672 [nVic]
PCI-E 3.0 with 16 lanes	16 [PS14]

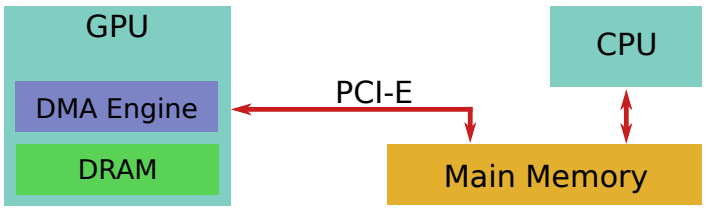


Figure 2.7: Illustration of competing memory accesses by the CPU and PCI-E transfers.

code for the GPUs.

2.4.1.2 Data-Transfer Costs and Interference

Dedicated GPUs require input data and results to be transferred between host main memory and GPU DRAM typically via PCI-E. Transfers have a high overhead because PCI-E bandwidth is significantly smaller than the DRAM bandwidth on the host and GPU, as illustrated in Table 2.2. Therefore, a simple implementation of a migration mechanism, which would copy all input data when a kernel migrates would in some cases incur unacceptably high costs because input data that has already been processed would be copied. For example, if the vector addition kernel of Figure 2.6 would be migrated from CPU to GPU after half the iteration space has been processed on the CPU, such a simple mechanism would transfer half of the input data unnecessarily.

Data transfers between main memory and the GPU can also interfere with other parts of the system. This subsection discusses only interference with execution on the CPU because this kind of interference is most relevant to the work presented in Chapter 3. Data transfers from and to the GPU are typically performed with DMA transfers, as illustrated in Figure 2.7, which can slow down execution on the CPU for various reasons. First of all, the CPU cannot use the full DRAM bandwidth if a DMA transfer is in progress because both kinds of accesses, DMA and accesses by the CPU,

have to be served by main memory. In addition, DMA transfers can cause cache line invalidation or flushes to ensure memory consistency, which further interferes with execution on the CPU [PH05]. Lastly, depending on the transfer mode, DMA and CPU accesses are either interleaved or the CPU is barred from accessing main memory during a transfer [TB15].

2.4.1.3 No Interrupt Support

Execution on the CPU can be interrupted, for example, by a timer to hand control over to the operating system. This involves a context switch, which saves the state of the preempted thread and so allows it to resume execution on another core. GPU kernels cannot receive interrupts and do not support context switches as CPUs do. Additionally, already launched GPU kernels and data transfers cannot be aborted by host code.

2.4.1.4 Integrated GPUs

Integrated GPUs share DRAM with the CPU and, therefore, do not require costly PCI-E data transfers. As with dedicated GPUs, integrated GPUs also do not support interrupts and CPU-style preemption, which could allow migration and, therefore, also benefit from our work. However, in this thesis we only use dedicated GPUs and leave integrated GPUs for future work. Because dedicated GPUs introduce additional complexity with PCI-E data transfers they are a more challenging for our work than integrated GPUs.

2.4.2 Related Work

Preliminary work on mid-kernel migration is presented in Section 2.4.2.1. Section 2.4.2.2 discusses work that schedules kernels on heterogeneous systems without mid-kernel migration. Section 2.4.2.3 discusses the use of kernel slicing and data transfer chunking, which are our core migration mechanisms, in other contexts. Section 2.4.2.4 discusses work that executes kernels on multiple heterogeneous devices at the same time with slicing but is not concerned with migration. Section 2.4.2.5 presents work that predicts device affinities for unseen kernels. Our system would require such a predictor in a real deployment.

2.4.2.1 Preliminary Work on Mid-Kernel Migration

To improve the utilisation of heterogeneous systems, Lösch et al. present a framework that allows fine-grained migration between heterogeneous devices and a novel high-level scheduling algorithm, which exploits this capability [LP20]. The programming interface is based on an abstract class, which requires programmers to implement methods for data transfers and kernel launches. Programmers provide a separate implementation of this abstract class for each device. The presented scheduling algorithm aims to minimise the makespan of applications, which are implemented with a task graph. Section 2.6.1 discusses weaknesses of this work and why we consider it preliminary.

2.4.2.2 Kernel-by-Kernel Scheduling

The following runtime systems migrate applications in heterogeneous systems but are limited to coarse grained *kernel-by-kernel* scheduling decisions.

Rinnegan has a task-based programming model and makes scheduling decisions on a task-by-task basis (tasks contain kernels) [PS16]. For each task, programmers must provide multiple implementations for potential execution on each device in the target system. Kernel-by-kernel scheduled runtime systems have two choices when a new kernel arrives and its fast device is not available: (1) wait for an unpredictable amount of time until the fast device becomes available, or (2) launch on the slow device without the ability to migrate later on. The perfect decision requires knowledge about the execution time of the kernel on each device and when the fast device will become available. This information is typically not available. Rinnegan requires a profiling phase and incorporates models to estimate these times. In contrast to Rinnegan, our system does not require multiple implementations of the same kernel for each target device. We can generate kernels for each device including data transfers for discrete GPUs with the information encoded in our programming model.

HTrOP uses the polyhedral model to generate OpenCL code for loops from sequential C and C++ code [RVKP19]. These loop nests are dynamically scheduled to accelerators or multicore CPUs depending on availability and input data size. Additionally, the runtime system can choose to execute the original sequential computation. Like all kernel-by-kernel scheduled systems, HTrOP has to decide whether a kernel should wait for an unavailable but better suited device or launch on an earlier available but slower device. HTrOP considers input data location, in- and output data size, and if an accelerator has previously been used to make this decision together with user set

weights for each factor. Kernel generation and migration are user transparent. However, because OpenCL kernels are generated from sequential code users can add only some performance optimisations for the fast device.

StarPU and OmpSs also use task-based parallel programming models and can dynamically schedule tasks to heterogeneous devices [ATNW09, PBAL15]. Programmers must provide multiple implementations of a task for the runtime system to be able to consider different devices. OmpSs supports task stealing on heterogeneous systems. For example, multicore CPUs can steal tasks from GPU task queue, assuming an implementation for both devices is provided.

SkePU is a skeleton library with the ability to schedule data parallel skeletons automatically to heterogeneous compute devices [DLK13]. In contrast to some task-based parallel programming models, programmers do not have to provide multiple implementations for the same skeleton. However, programmers can choose to provide device or input specific implementations to further improve performance [EK20].

Diamos et al. propose a system that dynamically constructs a dependency graph of kernels and schedules them to heterogeneous compute devices in an out-of-order execution fashion [DY08]. Again, this requires programmers to provide multiple device specific implementations of each kernel.

ConSerner automatically generates data transfer code for devices with separate memory from C code [GSB14]. The authors pair ConSerner in their evaluation with a scheduler that is capable of kernel-by-kernel migration.

CheCL checkpoints OpenCL applications between OpenCL kernels and can migrate and restart applications at checkpoints [TKS+11].

2.4.2.3 Kernel Slicing and Data Transfer Chunking

Slicing and data transfer chunking, which are our migration mechanisms in Chapter 3, have been used without migration in real-time systems as mechanisms to preempt GPU kernels and their data transfers [KLK+11, BK12, ZTL15].

RGEM makes data transfers preemptable by dividing them into chunks [KLK+11]. Dedicated GPUs have their own DRAM and all data needed by a kernel must be transferred from main memory to GPU DRAM before kernel launch (see Section 2.4.1). In commodity systems data transfers take place over PCI-E and typically via a DMA transfer, as explained in Section 2.4.1.2. However, once the DMA engine has started a data transfer it cannot be signalled to abort the transfer. This poses a problem for real-time systems in which jobs must be processed within a set time (see Section 2.5).

A job performing a data transfer can block another job from transferring its data, which may mean that the blocked job misses its deadline. To fix this, RGEM replaces a single data transfer with multiple successive transfers in software. This creates potential preemption points before and after each new transfer. The complexity of chunking is hidden behind a CUDA-based programming model.

PKM combines data transfer chunking with kernel slicing to further increase the granularity with which GPU jobs can be preempted [BK12]. The data transfer chunking strategy is the same strategy RGEM uses (see above) except that kernel executions and memory transfers may be overlapped. PKM divides application-level kernels into subkernels to add potential preemption points before and after each subkernel. The programming model is inspired by CUDA and OpenCL. In contrast to our work, PKM does not support slicing-aware data transfers (see Section 3.3.1) and it requires a profiling phase to choose appropriate slice sizes. We present predictive models to choose slice sizes and do not require a costly profiling phase.

GPES also combines kernel slicing and data transfer chunking [ZTL15]. The implementation of kernel slicing is different than ours and the one of PKM (see above). GPES inserts if-statements that check if the IDs of thread blocks are within the range of the current slice and exits blocks if this is not the case. Slicing and chunking are hidden behind the standard CUDA API. This implementation of slicing does not require additional computations to compute the, from the perspective of the original non-sliced kernel, correct block IDs. However, the additional checks introduce overheads too and each slice launches some blocks unnecessarily. The authors do not state how slice sizes are chosen and data transfers are not slicing aware.

Chen et al. use slicing to alleviate thrashing in the programmer transparent caches of GPUs [CHZ+18]. GPUs are more prone to cache thrashing than CPUs because the per thread cache size is significantly lower. Chen et al. use slicing as a mechanism to control the number of threads that execute on a GPU at any one time in order to reduce cache contention.

2.4.2.4 Simultaneous Use of CPU and GPU without Migration

Several works distribute the execution of an application kernel over multiple heterogeneous devices at the same time and some of these works also use slicing. However, these works are not concerned with migration. They also do not investigate the overheads that kernel partitioning mechanisms such as slicing introduce if a kernel executes on only one device at a time.

Cho et al. execute kernels in slices to parallelise the execution of a slice with the preprocessing of the next slice for split computation on the CPU and GPU [CNP+18]. Intra-block load imbalance leads to inefficient use of GPUs because a block only finishes and frees its multithreading slot for a new block when all of its threads have finished. Cho et al. reorder threads to reduce imbalance in blocks. Blocks with high expected execution times are scheduled on the CPU and the others on the GPU. The overheads of reordering and the profiling phase are hidden through slicing. A slice is profiled and reordered while its predecessor slice executes on the CPU and GPU.

FluidiCL slices on the CPU to keep track of which parts of the iteration space have already been processed on the CPU [PG14]. Pandit et al. process thread blocks [2] on CPUs from the highest to the lowest block ID and vice versa on GPUs. Bookkeeping data on GPUs, which keeps track of the block IDs run on CPUs is updated after each CPU slice. Execution finishes once all blocks have been executed.

A range of other works executes kernels on the CPU and GPU simultaneously without slicing [LHK09, GO11, SVZ+14, KBS+14]. We do not discuss these in detail as they are not related to kernel migration with slicing.

2.4.2.5 Device Affinity Predictors

Grewe et al. determine device affinities for CPU + GPU systems with a decision tree [GWO13]. The decision tree uses kernel features such as the communication to computation ratio and the number of coalesced memory accesses. The presented model correctly predicts the affinity of 97% of the NAS-PB benchmarks on two systems.

Taylor et al. use a hierarchy of support vector machines with polynomial kernels to determine device mappings for different metrics [TMW17]. The authors report 100% accuracy for the correct kernel to device mappings in a system with a big.Little CPU and GPU and with execution time as target metric.

2.5 Hard Real-Time Systems

Chapter 4 demonstrates how parallel hard real-time systems can be autotuned in a programmer transparent way with skeletons. Therefore, this section introduces hard and soft real-time systems and the terminology required for Chapter 4. Because Chapter 4

[2] We use the term *thread block* instead of the OpenCL term *work group* here to be consistent with the rest of the chapter.

is a case study with the task farm skeleton we also discuss this skeleton in the context of hard real-time systems.

Real-Time systems require inputs to be processed within set times and are typically divided into *soft real-time systems* and *hard real-time systems* [But11]. Failure to process inputs within the time limits, so called deadline misses, are unacceptable in *hard* real-time systems. Examples, for such systems are robotics applications or, in general, control systems in which deadline misses might lead to injuries. In contrast, in soft real-time systems deadline misses only lead to reduced quality of service.

2.5.1 Terminology

Tasks make up real-time programs and correspond to program code, which may be executed repeatedly [LL73, But11]. Tasks can be *aperiodic*, *periodic*, and *sporadic*. Periodic tasks are invoked at regular intervals, the so called *period*. Similarly, sporadic tasks have a minimum time between invocations, and *aperiodic* tasks are instantiated unpredictably. *Jobs* are instantiations of real-time tasks. The *release time* of a job is the time at which a particular job is instantiated [But11]. A *relative deadline* is the time within which a job must finish starting from its release time [But11]. In diagrams, job releases are depicted with an upwards facing arrow and job completions with a downward facing arrow. *Worst-Case Execution Times (WCETs)* are the longest possible execution times of code fragments or functions [HP17]. WCETs are particular important for hard real-time systems where system designers must ensure that a job never exceeds its deadline. WCETs are determined either empirically or with a hardware model and static code analysis [But11]. The *slack time* is the time between job completion and deadline [But11].

2.5.2 Task Farms for Real-Time Systems

Task[3] farms are composed of a set of workers that run in parallel and apply a function to a stream of inputs, as explained at the start of this chapter [ADKT17]. Inputs are generated by a producer and the results of workers are sent to a consumer. Figure 2.8 illustrates real-time applications that lend themselves to farms with a set of jobs, released every period T. Each job j is composed of a producer phase P_j that generates input data, a worker phase W_j that processes the data, and a consumer phase C_j that

[3]Note that the term *task* has here a different meaning than in the broader real-time systems context (see Section 2.5.1).

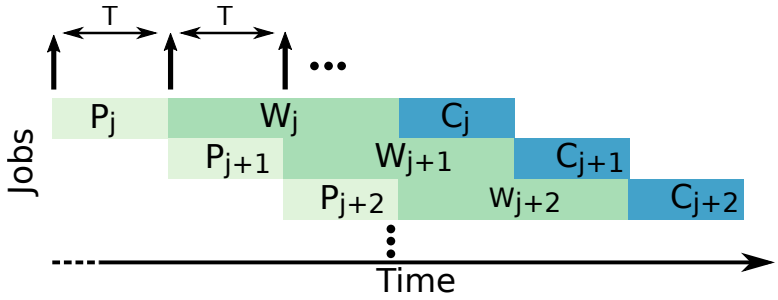


Figure 2.8: Illustration of hard real-time computations that can be implemented with task farms. Such computations have three generic phases: a producer (P), a worker (W), and a consumer phase (C).

consumes the results. Here the phases P_j can be mapped to the same thread as their execution does not overlap. The same is true for the C_j phases. However, W_j and W_{j+1} cannot be mapped to the same thread as their execution overlaps.

Synchronisation and farm internal communication between producer, workers, and consumer as well as the parallel execution of workers is implemented by a skeleton library or compiler. The sole task of application developers is to provide sequential worker, producer, and consumer functions. Internal communication is implemented with the so-called *dispatcher* and *aggregator*. They are hidden from application developers behind the farm API. The dispatcher schedules jobs on workers and the aggregator informs the consumer when new worker generated results are available.

2.5.3 Related Work

Section 2.5.3.1 discusses preliminary work on skeletons for real-time systems. Section 2.5.3.2 and Section 2.5.3.3 discuss work on task-based programming models and OpenMP for real-time systems. Commonly used mature programming models for real-time systems are introduced in Section 2.5.3.4. Finally, Section 2.5.3.5 discusses work that is similar to our job batching technique, which we use in Chapter 4 to demonstrate that skeletons can hide complex efficient implementations in the context of hard real-time systems.

2.5.3.1 Preliminary Work on Skeletons for Real-Time Systems

Stegmeier et al. and Ungerer et al. present work on skeletons for parallel real-time systems [UBF+16, SFJU]. The framework is a set of composable skeletons which exploits parallelism within job instances [SFJU, UBF+16]. To aid programmers, the

authors present a tool that makes core count recommendations based on UML models and estimated WCETs [UBF+16, JFGU14]. For the evaluation, the authors use a simulator instead of real hardware [UBF+16]. Section 2.6.2 discusses why we consider this work preliminary.

2.5.3.2 Task-Based Programming Models and Directed Acyclic Graphs

Task-based programming models are used for real-time and non-real-time systems (see Section 2.2.2). Recent publicly available (academic) work try to adapt the task pragmas of OpenMP, which is well known in non-real-time contexts, to real-time systems.

Pinho et al. use an implementation of OmpSs with real-time extensions [PNY+15]. In contrast to the original implementation of OmpSs, the authors extract the complete task graph at compile time in order to compute a static schedule in advance. To achieve this, they have to make pessimistic assumptions about the number of tasks and the dependencies between tasks to ensure correct execution for all possible situations that might arise at runtime. Firstly, programmers have to provide an upper bound for loop iterations. To ensure correctness in all situations, their compiler has to assume that the number of iterations in a loop with a dynamic iteration count is always equal to that upper bound. As a consequence, the static task graphs may contain more tasks than are actually executed at runtime. Secondly, dependencies between tasks whose presence depends on dynamic information are assumed to be always present. This means some opportunities for parallel execution may not be used because the task graphs may contain dependencies that are actually not present at runtime.

Sun et al. investigate worst-case response times of complex OpenMP task-graphs, which contain nested parallelism, tied tasks[4], and branches that cannot be modelled by standard directed acyclic graphs [SGW+17, SGSC19, SGL+20]. The authors present techniques to compute the worst-case response times (WCRTs) of such graphs and scheduling algorithms that improve WCRTs (see Section 2.5).

Serrano et al. analyse the response times of OpenMP tasks in the context of heterogeneous real-time systems [SQ18], and Wang et al. present a benchmark suite of OpenMP applications for real-time systems [WGS+17].

These works are motivated by a wealth of works that model parallel real-time applications as directed acyclic graphs, which lend themselves to an implementation with

[4]Tied tasks have to execute only on a single thread [SGW+17]. For example, tied tasks cannot resume execution on another thread after they have been preempted. All OpenMP tasks are tied tasks by default.

task-based programming models [LALG13, SFL+14, BMSSW13, BBMS+12].

2.5.3.3 Programming Models Based on the OpenMP `for` Pragma

Ferry et al. and Li et al. present first steps towards parallel_for based programming models for real-time systems. Both present RT-OpenMP, which is an implementation of the OpenMP `for` pragma that uses the theoretical results mentioned above [FLM+13, LLF+14]. Li et al. also present another implementation based on the unmodified OpenMP implementation of GCC and a patched Linux kernel that supports real-time scheduling algorithms [LLF+14].

Both works are motivated by theoretical work on the *synchronous task model*, which is an alternative application model to task graphs [SLA+12] (see above). In this model, applications are composed of sequential and parallel phases. All threads in a parallel phase start at the same time and a phase ends with a barrier. The number of threads in a phase and the execution time of each thread can be arbitrarily chosen.

2.5.3.4 Mature Real-Time APIs

We review parallelism related constructs that are offered by the commonly used real-time APIs of POSIX, Java, and ADA [Hun19, Ope18, Ada16]. These APIs require programmers to work with threads, which are a low-level abstraction, and set periods and deadlines at the level of threads. Synchronisation and communication are implemented with mutexes, monitors, queues, remote procedure calls and/or critical sections. Except for queues, programmers have to implement mutual exclusion manually. Failing to do so correctly leads to subtle bugs that are hard to reproduce and fix. Queues are on a higher abstraction level but are often used to send pointers to data structures. Programmers have then to implement mutually exclusive access across concurrently executing threads again themselves.

These constructs are on a lower abstraction level than skeletons. None of the currently used low-level constructs allow compilers or runtime systems to automatically set the degree of parallelism or hide and tune resource efficient implementations such as batching that are informed by the structure of a real-time task.

2.5.3.5 Job-Batching

Previous works also group computations to reduce communication costs and map jobs to multicore processors [KB06, BBW11, TBG+17, SGW+19, Sar87]. For example,

Sarkar and Kianzad et al. group sub-computations modelled as task graphs so that they are scheduled on the same core to reduce communication costs [Sar87]. However, the main contribution of the work presented in Chapter 4 is not batching. We use job batching just as an example for how structural information encoded in skeletons can be used for performance optimisations for real-time systems and how the complexity of such optimisations can be hidden behind skeletons. Besides that, the work mentioned above groups computations of a single task graph instance. As a reminder, periodic real-time systems instantiate real-time tasks once per period. In contrast, we group sub-computations of successive instantiations of a real-time task, which is an intricate challenge because task instances must be delayed without missing their deadlines. The complexity of this is hidden behind our task farm skeleton.

2.6 Critical Reflections on Related Work

In the light of our goal to simplify parallel programming and enable efficient implementations with structural information encoded in skeletons, as discussed in Chapter 1, we discuss here weaknesses of the closest related work that has been presented in the previous sections. In addition, other subordinate weaknesses are also discussed. We address all of them with our work in chapters 3 and 4.

2.6.1 Mid-Kernel Migration

Section 2.4.2.1 discussed preliminary work on mid-kernel migration by Lösch et al. [LP20]. In comparison to this thesis, the authors focus more on high-level scheduling and less on the migration mechanism and abstractions to hide the additional complexity. Therefore, the presented work has multiple limitations that make it preliminary when it comes to the mechanism and abstraction. Lösch et al. expose significant complexity to application developers. Programmers have to provide multiple implementations of each kernel for each device, and, in addition, kernels have to be manually modified so that they are still correct in the presence of migration. Furthermore, parameters of the migration mechanism have to be manually chosen by the programmer. However, manual choices are, in some cases, impractical because they would require a manual search through a very large parameter space.

The presented evaluation has two major weaknesses. Average speedups are determined based on the end-to-end execution times of several hundred randomly generated

task-sets with up to 32 kernel instances. This means migration time points are uncontrolled and the impact of slowed down kernels could be hidden by kernels that benefit from migration. Besides that, the authors do not use a benchmark suite to evaluate kernel migration and their implementation.

Section 2.4.2.2 discussed kernel-by-kernel scheduled programming models. With such programming models, application kernels could manually be made migratable by decomposing them into multiple sub-kernels. However, this would expose to the programmer the complexity of the decomposition and the decomposition granularity would need to be chosen manually, in the worst-case with a time consuming brute force search.

2.6.2 Hard Real-Time Skeletons

Section 2.5.3.1 presented work on real-time system skeletons by Stegmeier et al. and Ungerer et al. [UBF+16, SFJU]. We consider this work preliminary for several reasons.

The presented skeletons are not self-tuning. This complicates parallel programming because programmers have to choose parameters such as the degree of parallelism by hand, which is complicated on general purpose systems but even harder on hard real-time systems where programmers must ensure that no deadline misses occur. To aid programmers the authors present an autotuner that recommends how many cores should be assigned to each skeleton to meet the real-time demands of the application [UBF+16]. However, the tool only provides estimates because they are based on approximated WCETs and UML diagrams. Programmers who take the recommended core counts without manual analysis risk deadline misses, which are, as mentioned above, unacceptable in hard real-time systems.

The authors also do not demonstrate how information encoded in skeletons can be used for efficient implementations in the context of real-time systems, as demonstrated before in other fields [DG08, MCF18, LFC13, DSTD16]. This is an important property of skeletons and should be demonstrated to make a convincing case for hard real-time skeletons.

Independent of these programmability related criticisms, the presented skeletons are unfit for *hard* real-time systems because they are not fully timing analysable. As mentioned above Stegmeier and Ungerer work only with approximated WCETs of their skeletons. For example, the authors add a blanket 100,000 cycles to the WCET of a code section for each skeleton invocation and another 10,000 cycles for each thread

used by a skeleton [JFGU14]. In the worst-case, deadline misses occur if these WCET buffers are too small. In the best case, they waste computational resources. Lastly, the authors use a simulator for the evaluation and not real hardware [UBF+16].

Section 2.5.3.3 discussed work on OpenMP and its `for` pragma in the context of real-time systems. This pragma is very close to skeletons, however Ferry et al. and Li et al. are not concerned with real-time skeletons but want to adapt the OpenMP API for real-time systems [FLM+13, LLF+14]. In contrast, our vision is to develop a composable set of real-time skeletons that encodes more structural information, which can help runtime systems and compilers, than the current OpenMP and task-based approaches. Lastly, no detailed WCET analysis of the OpenMP runtime system is presented, which would be necessary for hard real-time systems.

2.7 Summary

This chapter discussed background material and related work. We first discussed skeletons in detail and, following this, alternative programming models at different abstraction levels. Next, we explained how existing operating systems implement thread migration on CPUs and why migration between CPUs and GPUs cannot be implemented in the same way. We introduced heterogeneous systems, with a focus on GPGPUs, and work related to our work on dynamic scheduling on heterogeneous systems. Following this, we discussed real-time systems and related work on skeleton programming for these systems. Lastly, we concluded this chapter with critical reflections on existing works and opportunities for improvements in the context of our goal to ease parallel programming.

Chapter 3

Transparent Kernel Migration

This chapter presents novel migration techniques for heterogeneous systems that allow finer-grained scheduling decisions than competitor systems. The migration mechanism is informed by the parallel_for skeleton and the complexity of the mechanism is hidden behind it. This chapter considers general purpose systems. Our work on real-time systems is presented in Chapter 4.

3.1 Introduction

Resources available to an application can vary unpredictably over its execution. For example, multiprogramming is used across the computing spectrum to improve utilisation: high-end data centre machines are multiprogrammed with demand-driven services and other jobs that reuse resources at times of low activity [LCG+15, KMHK12, VPK+15], while mixed criticality embedded systems mix timing critical workloads with less critical ones, again to improve utilisation and energy consumption [dNLR09, Ves07, BYA+19]. Energy and thermal constraints exacerbate the problem. An evolving workload mix can cause changes to the best allocation of the available power budget to silicon, or the appropriate exploitation of hardware characteristics, as in big.Little systems [CKC12]. In response, on CPU only systems, the OS is able to migrate multithreaded applications between cores. For example, dynamic scheduling strategies redeploy cores as they become available [CKB13, SGS14, RZLA], and dynamically review the application to device mapping [CHCF15]. Power capped systems control the active core count and so their power consumption with thread migration [CHCR11].

It would be natural and desirable for a similarly flexible migration capability to

be available on CPU + GPU platforms. However, migrating running applications between devices in such systems is challenging. Current runtime systems for GPUs make kernel-to-device scheduling decisions only at coarse-grained kernel launches and cannot perform *mid-kernel migration* [RVKP19, PS16]. Because of this, perfect scheduling decisions on whether to launch on an earlier available slow device or wait for a fast device require unattainable knowledge of the future. If the best performance after migration also requires a change of language runtime (e.g. switching from OpenMP on the CPU to CUDA on the GPU, or vice-versa) the challenge is even greater.

In this chapter we investigate a mechanism which enables these flexibilities for CPU + GPU systems, including the ability to switch the underlying language runtime. Specifically, we investigate a mechanism which allows schedulers to migrate applications in *mid-kernel execution* from CPU to GPU, and vice versa. In principle, to exploit mid-kernel migration, each application could be written with multiple embedded variants, data transfer code to switch between these, and heuristics to decide under which dynamic circumstances to do so. However, this would be very challenging to application developers, and in fact, could reduce maintainability by solidifying these decisions amongst true application-level code. We show that mid-kernel migration can be hidden behind the parallel_for skeleton, and that decisions on scheduling granularities, migration strategies, and device utilisation can be handled efficiently and transparently, without burdening the application programmer. In more detail, our migration mechanism subdivides iteration spaces into slices as prior work has done [CHZ+18, PG14, CNP+18, ZTL15], and considers migration on a slice-by-slice basis. Slices provide stable states at which points the context of an application can switch devices and runtimes. To choose slice sizes we use off-line trained predictive models. Data transfers in systems with distinct per-device memory can have a significant cost, and so we also transfer data in a slicing aware way, to avoid unnecessary transfers if a kernel migrates. Our mechanism provides the key technological basis for transparent migration and runtime adaption of applications in CPU + GPU systems.

We evaluate mid-kernel migration with the First Come, First Served (FCFS) scheduling policy, using a simple scenario to allow us to focus on the cost and contribution of the migration mechanism. We show analytically that mid-kernel migration removes the need for unattainable knowledge of the future for scheduling decisions. In more detail, we show that FCFS with mid-kernel migration can achieve a theoretical maximum speedup of 1.33x over a perfect knowledge FCFS schedule without mid-kernel migration, under ideal conditions. Crucially, FCFS with mid-kernel migration never

performs worse. We confirm these results experimentally with nine benchmarks on a CPU + GPU system and show that mid-kernel migration with our simple policy achieves speedups of up to 1.30x over kernel-by-kernel scheduled systems even if they benefit from a perfect schedule. We also demonstrate that if a kernel never migrates the overheads of slicing are negligible, and lastly, that our parallel_for reduces the code complexity significantly if compared to a manual implementation of the migration capability. In summary, this chapter makes the following contributions:

1. We describe a mechanism that enables mid-kernel execution migration between CPU and GPU in both directions, including the possibility of switching language runtimes dynamically. In contrast, current systems cannot migrate kernels between CPUs and GPUs once they have been launched [PS16, RVKP19].

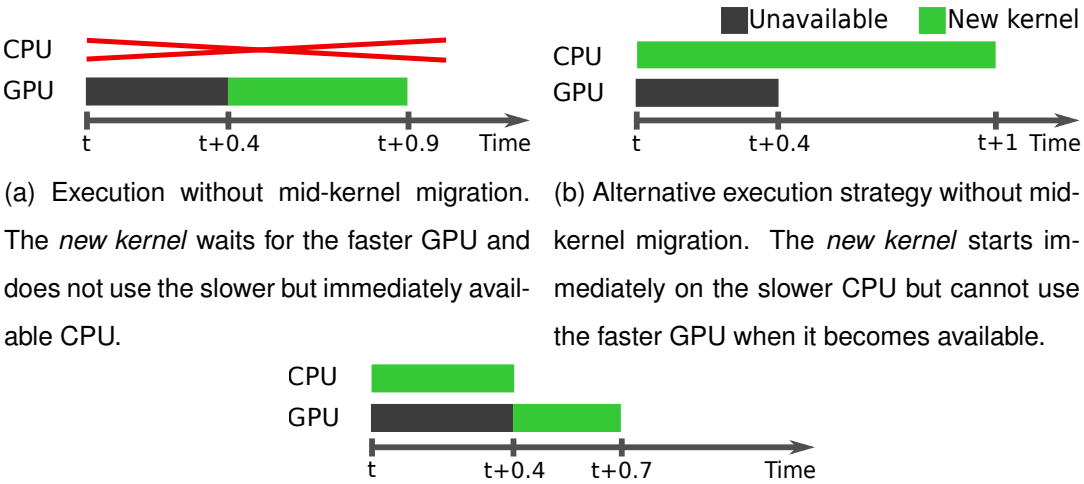
2. We show that the complexity of the mechanism and its efficient implementation can be hidden from programmers behind a skeleton-like high-level programming model based on the widely known parallel_for construct.

3. We show that slice size choices that introduce acceptable overheads can be learned by predictive models.

4. We present an analytically derived maximum for the speedups with migration over current kernel-by-kernel scheduled systems in a simple deployment scenario.

5. We provide a detailed evaluation that demonstrates, among other things, the general performance benefits of mid-kernel migration and exposes the performance behaviour of the mechanism in edge cases.

The remainder of this chapter is structured as follows: Section 3.2 provides a motivating example. Section 3.3 introduces the migration mechanism and techniques for an efficient implementation. Section 3.4 discusses the predictive slice size models. Section 3.5 discusses a high-level programming model based on parallel_for that hides the complexity of the migration mechanism and the slice size predictors. Section 3.6 discusses the comparator for our analytical models and experimental evaluation, and the maximum theoretical speedup over it that can be achieved with mid-kernel migration for our deployment scenario. Section 3.7 presents experimental results. Section 3.8 concludes this chapter.



(a) Execution without mid-kernel migration. The *new kernel* waits for the faster GPU and does not use the slower but immediately available CPU.

(b) Alternative execution strategy without mid-kernel migration. The *new kernel* starts immediately on the slower CPU but cannot use the faster GPU when it becomes available.

(c) Execution with mid-kernel migration. The *new kernel* starts on the CPU and finishes execution on the faster GPU.

Figure 3.1: Illustration of the typical performance improvement opportunities missed without mid-kernel migration. (3.1a) and (3.1b) illustrate two possible execution strategies of a *new kernel* without migration, and (3.1c) its execution with migration. When the new kernel arrives at time t the GPU, on which it would execute twice as fast as on the CPU, is unavailable for 0.4 time units. Without mid-kernel migration the *new kernel* either waits for the GPU (3.1a) or starts immediately on the slower CPU (3.1b). In the former case, the new kernel finishes after 0.9 time units, after it waited for 0.4 time units for the GPU and then executed for 0.5 time units on it. In the latter case, the kernel launches immediately on the CPU and executes for one time unit. With migration the kernel executes for 0.4 time units on the CPU and migrates to the GPU when it becomes available to finish the remaining work in 0.3 time units (3.1c). With migration the *new kernel* finishes after 0.7 time units, and, therefore, is 1.29x faster than the schedule that results in the shortest combined waiting and execution time without migration, which is illustrated in (3.1a). We leave scheduling policies that exploit simultaneous execution on the CPU and GPU of a kernel for future work (see 5.3.1).

3.2 Motivation for Mid-Kernel Migration

This section illustrates the potential performance benefits of mid-kernel migration, and how a scheduler can take advantage of it in a simple scenario. We make mid-kernel migration and its implementation programmer transparent with our parallel_for, which is introduced in Section 3.5.

3.2.1 Better Performance with Mid-Kernel Migration

In current systems, kernels are only scheduled at launch time as discussed in Section 3.1 and illustrated with an example in Figure 3.1 [ATNW09, RVKP19, PS16] (see Section 3.6.2 for a more detailed discussion of current systems). Opportunities for performance improvements are therefore missed if devices are temporarily unavailable. On the one hand, kernels cannot make progress while they wait for their fast device[1]. On the other hand, if launched on an alternative slow device, kernels cannot migrate when faster devices become available. We fix this with mid-kernel migration. As shown in Figure 3.1c, kernels make progress on earlier available devices instead of waiting, and schedulers can migrate them to faster ones when they become available.

3.2.2 Simplified Scheduling Decisions

Without mid-kernel migration, perfect decisions on whether to launch kernels on their fast or slow device require generally unattainable knowledge of the future. To determine which kernel launch decision leads to the shortest combination of waiting and execution time, schedulers of such systems need to know when the fast device of a kernel will become available, and how long a new kernel would take to complete on each device. We simplify this decision with mid-kernel migration so that schedulers do not require such knowledge about the future. With mid-kernel migration, schedulers can launch kernels on the earliest available device and migrate them if a faster device becomes available. This enables kernels which would not otherwise have been migrated to make progress on another device in the meantime. Similarly, kernels can utilise faster devices when they become available, which is advantageous when the best decision for current systems and schedulers is to execute on their slow device.

Section 3.6 builds idealised models to show that this policy never performs worse than the current kernel-by-kernel scheduled systems. However, in practice, management code, interference during the migration, and on-the-fly device setup cause slowdowns in some cases. We develop techniques and strategies which address these problems in order to leave an overall performance win.

[1] We distinguish between the *fast* and *slow device* from the perspective of a kernel. In the absence of interference and contention kernels have a lower execution time on their *fast device* than on their *slow device*. Whether the CPU or the GPU is the slow device depends on the kernel and the target system.

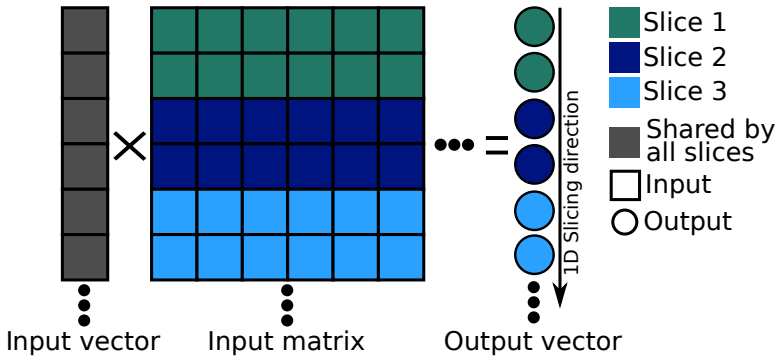


Figure 3.2: Slicing and slicing aware data transfers with matrix-vector multiplication. The output vector is computed slice-by-slice. Only the matrix rows needed for a slice are transferred. The input vector is shared by all slices and so not affected by slicing. In this example the iteration space is one-dimensional, and its size is equal to the length of the output vector. Additionally, each output element could be computed in multiple slices with a 2D iteration space.

3.3 Mid-Kernel Migration

This section presents a migration mechanism for systems with a CPU and a dedicated GPU, and techniques required for its efficient implementation. We implement the mechanism on top of OpenCL, CUDA, and OpenMP. Specifically, in our implementation, kernels switch between CUDA on GPUs and OpenCL or OpenMP on CPUs. The mechanism is enabled through our parallel_for based programming model by which the programmer guarantees that iteration space points can be processed independent of each other (see Section 3.5). Moreover, the programming model makes the mechanism and its optimisations programmer transparent by virtue of its high-level nature.

3.3.1 Iteration Space Slicing, Runtime Switching, and Slicing Aware Data Transfers

To enable migration, kernel iteration spaces are processed in slices as illustrated in Figure 3.2 and migration is considered on a slice-by-slice basis. For clarity, in the remainder of this chapter we distinguish between *application-level kernels* and *slice kernels*. The former are the simple intuitive CUDA and OpenCL kernels or OpenMP loop nests which would have been written in a conventional coding of the application and which are now generated implicitly by our system. With our mechanism, these are each sliced into a sequence of finer grained kernels, as illustrated in Figure 3.3,

and for which we now reserve the term *slice kernel*. We slice in all dimensions of the iteration space with the same slice size. More complex policies can be developed. For example, slices in 2D iteration spaces are squares except for the slices at the end of rows or columns, which might be rectangular. Optionally, kernels can execute without slicing on their fast device if preemption on this device never occurs. In some cases, offsets need to be added to the thread and block IDs in the generated internal kernel code as illustrated in Figure 3.4.

It might be necessary or desirable to change the underlying language runtime when an application kernel migrates. For example, CUDA kernels cannot execute on CPUs and so the runtime system must be switched when an application kernel migrates from a GPU where it used CUDA to a CPU. Between slices the state of a kernel, including

```
1  ...
2  // Execute the kernel in slices
3  while not all slices have been executed:
4    offset_into_iteration_space ← get iteration space offset
5    slice_size                  ← get slice size
6    target_dev                  ← get device
7    // Launch slice kernel
8    launch_slice_kernel(target_dev, offset_into_iteration_space, slice_size)
9  ...
```

(a) A pseudocode-based illustration of slicing.

```
1  ...
2  launch_application_kernel(target_device, iteration_space_size)
3  ...
```

(b) The equivalent unsliced pseudocode.

Figure 3.3: High-level illustration of how slicing is implemented internally with pseudocode. We do not require programmers to hand implement sliced kernels but automate this with our parallel_for instead (see Section 3.5). Implementation details are omitted for clarity. For example, we have omitted code for chunked data transfers, code for the abortion of slices (see Sections 3.3.2 and 3.3.3), and code for edge cases such as iteration space sizes that are not a multiple of the slice size. Additionally, our actual CUDA implementation executes data transfers and manipulates pointers into the in- and output buffers inside the loop before and after the kernel launch.

```
1  parallel_for pf(0, problem_size, [&]DEVICE_HOPPER_FUNCTION_PARAMETER() {
2    int iteration = GET_ITERATION();
3    ...
4  });
5
```

(a) High-level application developer implementation with our parallel_for skeleton (see Section 3.5).

```
1  __global__ void cuda_kernel(size_t _batch_offset_x) {
2    int iteration = ((blockDim.x * blockIdx.x + threadIdx.x) +
3                     _batch_offset_x * blockDim.x);
4    ...
5  }
6
```

(b) Generated CUDA kernel with automatically inserted offsets (see `_batch_offset_x`).

Figure 3.4: Illustration of automatically added offsets to compute the correct iteration count in each slice kernel. The CUDA constructs `blockDim.x`, `blockIdx.x`, and `threadIdx.x` return only the respective values within a slice kernel and are not aware of the complete iteration space of the application kernel. The source-to-source translator that inserts the required offsets in the generated code is introduced in Section 3.5.

which iteration space portion has already been processed, is stable and known. This way, after migration the application kernel execution can be resumed on the target device with a different implementation and runtime. This does not require any changes to the underlying CUDA, OpenCL, and OpenMP runtimes because slicing can be implemented as a layer on top of them. After migration, execution is simply resumed by calling the kernel launch functions and possibly data transfer calls of the new target device to execute the next slice kernel[2].

Data is transferred to dedicated GPUs in a slicing aware way to avoid unnecessary transfers which might negate the benefits of migration. Without this, the entire input data set needs to be transferred before the computation of the first slice kernel. If an application kernel migrates later on, some of the input data would have been transferred unnecessarily. Therefore, only the input data required for the current slice kernel must be transferred. In Figure 3.2 slices and the parts of the input matrix they require are

[2]In the case of OpenMP a loop nest that implements the slice kernels is executed.

colour-coded. In some cases, such as sparse matrix multiplication, this is non-trivial because some of the data required by a slice kernel is dependent on other input data, but this is addressed in our implementation.

3.3.2 Migration Strategies

The migration strategy depends on the device in use, whether the kernel is idempotent[3], and the current execution step of the slice in execution. In addition, partial results computed on different devices must be merged. In some situations, slice kernels on the GPU are *aborted*. To abort a slice kernel, the execution jumps out of the current sequence of data transfers and kernel launches and restarts the slice kernel on the CPU.

Migration from GPU to CPU In this migration scenario, the current slice kernel is in most cases aborted on the GPU and restarted on the CPU. GPU slice kernels are composed of multiple data transfers and a CUDA kernel launch, and can be aborted between these substeps, even though the individual steps cannot be aborted once launched. Therefore, if the application kernel migrates from a GPU to a CPU the current slice kernel is restarted on the CPU and aborted at the end of the current substep on the GPU. Non-idempotent slice kernels are not aborted on the GPU if any results of the slice kernel have already been transferred back from the GPU to the host. Unlike our implementation, migration could be implemented without aborts but would be less efficient because data transfers of the non-aborted slice kernel on the GPU would interfere with execution on the CPU.

Migration from CPU to GPU In this scenario, either the *current* slice kernel on the CPU is restarted on the GPU or the *next* slice kernel is started on the GPU and the *current* one finishes on the CPU. On CPUs slice kernels correspond to either a single OpenCL kernel without data transfers, which cannot be safely aborted, or to an OpenMP loop nest, which will not be aborted in our implementation. We choose not to abort OpenMP loops in order to avoid the performance penalty that the corresponding code would introduce even if the abortion is not activated. Therefore, if the application kernel migrates from a CPU to a GPU and the kernel is idempotent the *current*

[3]Our parallel_for allows programmers to set whether the user code, that is passed to it, is idempotent [dKSJ12]. The user code is idempotent if it can be re-executed multiple times for an iteration point and produces the same correct outputs with each re-execution. The results of the parallel_for are still correct if this optional tuning parameter is not set despite the user code being idempotent, but potential performance might be lost.

slice on the CPU is started again on the GPU. In this case we do not wait for the old instance of the slice on the CPU to finish if the application kernel finishes earlier on the GPU. If the kernel is not idempotent the *next* slice kernel is started immediately on the GPU and executes in parallel with the *current* already launched slice, which runs on the CPU.

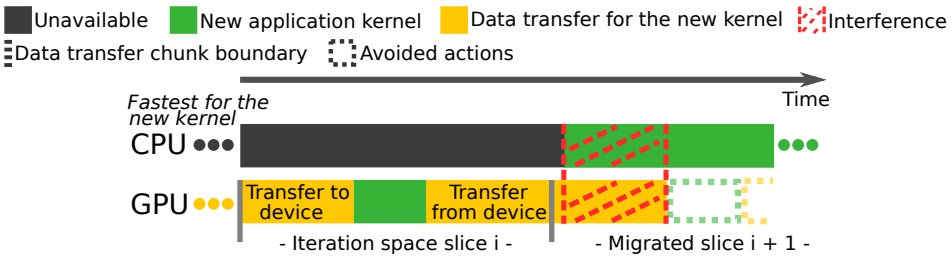
Merging results Intermediate results successively computed on more than one device must be merged. We use two strategies depending on the access patterns in the output buffers. Both access pattern and whether a buffer is an in- or output buffer are specified through our programming model (see Section 3.5 for a discussion of the access pattern attributes).

Merging Strategy 1: If the slice kernels write to distinct subsections of an output buffer, then results computed on the GPU for such a buffer are transferred into their subsection in host main memory after each slice kernel.

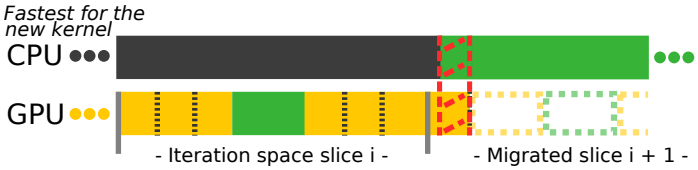
Merging Strategy 2: If a buffer is not accessed in this way, then incremental buffer updates in host main memory, as described above, are not possible. Instead, intermediate results computed on the CPU are transferred to the GPU before the first GPU slice kernel executes. With this strategy, GPU results are only transferred to host main memory when the application kernel migrates to the CPU or once the application kernel has completed on the GPU.

3.3.3 Interference Reduction and Earlier Aborts

Migration from GPU to CPU involves the abortion of the current slice kernel on the GPU (see Section 3.3.2). To be able to abort GPU slices earlier, data transfers are broken down into chunks as shown in Figure 3.5. Each data transfer chunk corresponds to a new internal API call and so transfers can be aborted between them. Application kernels cannot be aborted at arbitrary points because, as above, OpenCL and CUDA do not allow already issued data transfers and kernels to be aborted. Because of this, execution on the CPU and GPU are overlapped until the application kernel can be aborted, as indicated in Figure 3.5 by the red dashed lines. This overlap must be minimised for two reasons. Firstly, data transfers interfere with the execution on the CPU and therefore degrade performance. Secondly, reaching the next point at which the slice on the GPU can be aborted can take longer than the remaining application kernel execution time on the CPU.



(a) Migration without data transfer chunking. The slice kernel on the GPU cannot be aborted until the data transfer is finished. The execution on the CPU is slowed down by interference caused by the data transfer.



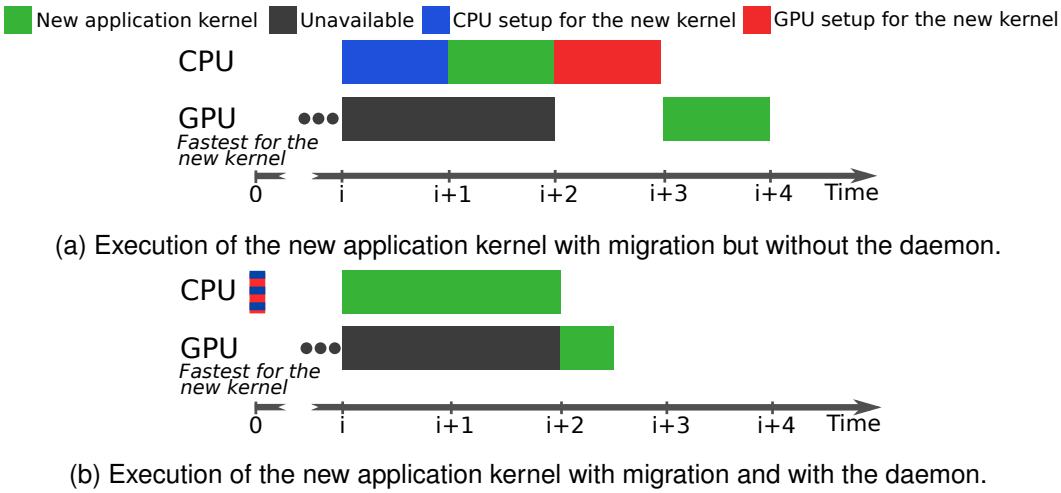
(b) Migration with data transfer chunking. The slice kernel on the GPU is aborted between data transfer chunks. This reduces the interference with the execution on the CPU.

Figure 3.5: Migration without (3.5a) and with (3.5b) chunked data transfers.

We use a one-off brute force search to determine chunk sizes that introduce no more than an implementation-set maximum slowdown over execution without chunking. In a full deployment, this search would take place only once, transparently to programmers, "at the factory" or when the runtime system is installed. The search is not repeated for each application kernel instance because the chunking overheads are application kernel independent. This is the case because per buffer data transfer code is the same across kernels. In our implementation, instead of a single call to `cudaMemcpy`, `cudaMemcpy` is called once for each chunk in a loop, and the overheads are caused by the loop, additional function calls, and pointer arithmetic.

3.3.4 Device Setup Cost Reduction

OpenCL and CUDA must set up devices before their use. Applications that execute without migration can hide this cost when an application kernel is waiting for a device other than the CPU. With migration this is not possible if waiting for the GPU, because the CPU, where the setup must take place, is already occupied by the new application kernel, as shown in Figure 3.6a. To fix this we introduce a daemon that performs the setup once when it starts, on behalf of any applications that run afterwards, as illustrated in Figure 3.6b. In this way, applications do not have to spend time in setup



(a) Execution of the new application kernel with migration but without the daemon.

(b) Execution of the new application kernel with migration and with the daemon.

Figure 3.6: Execution without and with a daemon that sets up OpenCL and/or CUDA in advance. With the daemon the new application kernel does not spend time in setup code when it migrates. The daemon performs the setup once when it starts as indicated by the striped bar at time zero in (3.6b). Data transfers are not shown for simplicity.

code during migrations. To implement this, applications are executed by the daemon to give them access to the preinitialised OpenCL and CUDA handles.

3.4 Choosing Slice Sizes

This section discusses our machine learning models that predict kernel instance-specific slice sizes. Since slicing is hidden entirely behind our parallel_for, slice size choices are also handled transparently to the programmer. As discussed in Section 3.3.1 if the slow device of an application kernel is available earlier than its fast device, then the kernel executes on its slow device in slices to allow for mid-kernel migration once the fast device becomes available. The models presented in this chapter are a function $f(v) = s$, that based on a vector of application kernel features v predicts a slice size s. We build separate models for the CPU and GPU because of their strong microarchitectural differences. We also exploit prior work on slicing in the context of caching (see Section 3.4.5) [CHZ+18, KHL+19].

3.4.1 Target Slice Sizes

The target slice sizes are a compromise between slicing overheads and resource wastage. On the one hand, *the larger* the slice sizes the smaller the overheads because fewer

Table 3.1: Features used to predict slice sizes. “Device” indicates whether the features are used to predict slice sizes for the CPU or GPU. All static features are extracted at compile time and are adjusted at runtime. Dynamic features are extracted at runtime.

Feature	Device	Extraction Type
Bytes transferred to the GPU per CUDA thread	GPU	Dynamic
Comparison operations	CPU	Static
Floating point and integer compute operations	CPU & GPU	Static
Memory accesses	CPU	Static

slices are executed, which in turn means code that implements slicing and introduces overheads is executed less often (see Section 3.3.1). These overheads cannot be amortised if a kernel never migrates, and lead to slowdowns in these situations. On the other hand, *the smaller* the slice sizes, the quicker an application kernel migrates once a faster device becomes available because migration is considered more frequently. Additionally, less interference is caused on the fast device by the residual execution on the slow device after migration (see Section 3.3.2 and Section 3.3.3). This is so because the remaining slice on the slow device finishes earlier with smaller slice sizes, or because a point at which the slice can be aborted is reached earlier, since the data transfers between two such points are smaller. Similarly, the smaller the slice sizes, the less work is thrown away when a slice is aborted. Therefore, the target slice sizes are the smallest slice sizes that introduce an acceptable slowdown if an application kernel never migrates. As noted above, we do not consider real-time systems in this chapter. Therefore, the acceptable overheads introduced by the predicted slice sizes do not have a hard upper bound.

3.4.2 Application Kernel Features for the Slice Size Predictors

Table 3.1 lists the features used by the slice size predictors. To extract static source code features, we build a feature extractor with Clang that traverses the abstract syntax tree of the kernel code. The extractor is based on source code used by Cummins et al. and Grewe et al. [CPWL17, GWO13, CGW20]. Static features are adjusted at runtime once loop bounds are known by multiplying counts for operations inside the loops with the iteration counts of the corresponding loops. Loop bounds are determined based on the parameter values that are passed to the kernel if the parameter values

can be directly inserted into the bounds, or the kernel source code if the bounds are hard-coded. As heuristics, terms in the loop condition that cannot be evaluated before the kernel execution are ignored and loops whose bounds cannot be determined are set to an iteration count of one. Operations in if-branches without a corresponding else-branch in loops are not counted as they are typically not taken, for example a branch that is only taken when a sought value has been found. To generate training data, we determine loop bounds manually if they cannot be determined automatically. The only dynamic feature is *"Bytes transferred to the GPU per thread"*. This feature is computed based on the amount of data to be transferred to the GPU and the number of CUDA threads with which an application kernel implemented in CUDA would be launched.

The chosen features are indicative of the work required for each iteration space point. This enables the models to predict slice sizes that introduce acceptably low overheads if an application kernel never migrates. This is so because the target slice size for a set overhead and the average execution time per iteration space point correlate. The less work is required per iteration space point, the larger the slice sizes that introduce only a set overhead. The reason is that the per slice execution time of the slicing code that causes the overheads is independent of the slice size (see Section 3.3.1). For example, for an average execution time per iteration space point of 1ms and 1ms per slice for the slicing code, the smallest slice size that introduces a slowdown of no more than 1.01x is 100.

Other features that we explored include global, local, and private memory accesses, bytes transferred to and from the GPU[4], and compute operations per memory access. We did not consider high-level kernel features like memory access patterns that can be encoded in skeletons or that can be detected via compiler analysis and leave this for future work.

3.4.3 Training the Slice Size Predictors

We use linear regression for the CPU model, and a random forest regressor with 50 decision trees with a depth of two for the GPU model [Bre01]. All hyperparameters except the tree depth and the tree count are the defaults of the popular Scikit-Learn machine learning library [PVG+11]. We reduce the tree count to reduce the runtime costs of predictions with the random forest. For slice sizes on the CPU, we use linear

[4]Note, we use bytes transferred *per CUDA thread*

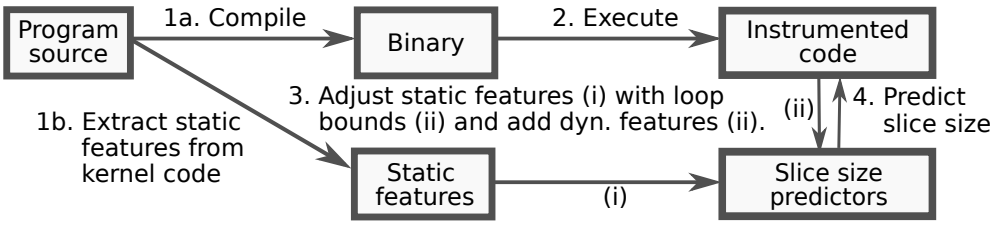


Figure 3.7: Overview over the steps required to predict slice sizes (see Section 3.4.4 for a detailed discussion). The working steps are performed in order of the numbering.

regression because of its low runtime overheads and the strong linear correlations between CPU target slice sizes and input features. For GPU slice sizes we use random forests because they performed best of all models explored. These other GPU models were support vector machines, decision trees, and linear regression.

To train the models we extract features from the kernels in the training set, as described in Section 3.4.2, and determine target slice sizes through brute force search. The training and the required brute force search need to be done only once *"at the factory"* before the system is deployed, as in previous work [WO09, GWO13]. If the workload type in a real deployment changes the models can be retrained with applications representative of the new workload and updated. For practical reasons we do not test each point in the slice size parameter space of each kernel, but step through the parameter space with a step size that we set by hand for each kernel instance. During the brute force search, we increase the slice size until the overheads are between 2% and 4% (see Section 4.1). We use a maximum allowed overhead of 2% for slicing on the GPU except for a small number of benchmark instances for which slice sizes with only 2% overhead cannot be found. We increase the maximum allowed overhead to up to 2.75% in these cases. For the same reason we always use a maximum of 4% for slicing on the CPU. We take the logs of the features for both models and the log of the training slice sizes for the CPU model to strengthen the linear correlations between features and target slice sizes. Otherwise, the correlations are weakened by heavy tails. Finally, for the CPU model we standardise the features and slice sizes to place their means at zero and normalise them to their standard deviation.

3.4.4 Deploying the Slice Size Predictors

Figure 3.7 illustrates when features are extracted and predictions are made. Static features are extracted at compile time (1) (see Section 3.4.2). At runtime (2) the size of the buffers allocated on the GPU, the iteration space size, and application kernel

parameters that determine loop bounds are recorded. Next, the loop bounds are used to adjust the static features (3) as described in Section 3.4.2. Finally, the predictive models predict a slice size (4). Because we use the log of the slice sizes and standardise the slice sizes afterwards to train the CPU model (see Section 3.4.3), we apply the inverse of both to the raw predictions to compute the final CPU slice sizes. Finally, to avoid a load imbalance on the CPU we round the predicted slice size down to the next multiple of $number\_of\_cores * ocl\_block\_size$. This way the same number of blocks is executed by each core.

3.4.5 Choosing Slice Sizes for Sparse Matrix Vector Multiplication

The slice sizes for Sparse Matrix Vector Multiplication (SPMV) are handled differently, because SPMV is usually implemented as a specialised library, and so we assume that the runtime system knows when SPMV is executed. We use this knowledge to repurpose previous work that uses slicing without migration [CHZ+18, KHL+19]. This work exploits caching effects in SPMV of which our predictors are not aware with smaller and so for our purposes better slice sizes (see Section 3.4.1). Chen et al. report that SPMV benefits from these caching effects [CHZ+18]. With this knowledge we use a static analysis-based slice size heuristic that is heavily inspired by the heuristic presented by Kim et al. [KHL+19]. In contrast to Kim et al., we consider the L2 cache instead of the L1 because on our GPU normal memory accesses do not go through the L1 cache. Besides that, we do not change the block size but use the original block size of the benchmark.

3.5 Our High-Level Programming Model

The complexity of mid-kernel migration and the slice size prediction is hidden behind a high-level programming model based on parallel_for in the style of OpenMP, Kokkos, Raja, and SYCL [Boa20, ETS14, BBH+19, Gro20]. In comparison to existing parallel_for implementations, we require programmers to provide additional memory access pattern attributes for each buffer. In return we hide the complexity of an efficient implementation of slicing and the slice size prediction. In more detail, these attributes are required to generate code for slicing aware data transfers (see Section 3.3.1) and to merge partial results (see Section 3.3.2).

Our parallel_for executes in parallel multiple instances of a function parameter that

```
1 ...
2 // Create parallel_for instance. Code similar to this is required by existing parallel_for
      implementations.
3 parallel_for pf(0, output_vect_size, [=]DEVICE_HOPPER_FUNCTION_PARAMETER() {
4  int iteration = get_iteration();
5  int result = 0;
6  for (unsigned int col = 0; col < input_vector_size; ++col)
7   result += in_vector[col] * in_matrix[iteration * input_vector_size + col];
8  out_vector[iteration] = result;
9 });
10 // Specify memory accesses. Our programming model requires these in addition to the function
      parameter.
11 int results_per_batch       = pf.batch_size;
12 int matrix_inputs_per_batch = pf.batch_size * MATRIX_ROW_SIZE;
13 pf.add_buffer_access_patterns(
14  buf(in_vector,  direction::in,  pattern::all_or_any),
15  buf(in_matrix,  direction::in,  pattern::successive_subsections(
      matrix_inputs_per_batch)),
16  buf(out_vector, direction::out, pattern::successive_subsections(
      results_per_batch)));
17 // Set optional tuning parameters and run.
18 pf.opt_set_simple_indices(true).opt_set_is_idempotent(true).run();
19 ...
```

Figure 3.8: Illustration of the parallel_for based programming model with a simple implementation of matrix vector multiplication (see Section 3.5 for a detailed explanation). Existing parallel_for implementations require code similar to lines three to nine. The only additional code that our model requires are access pattern attributes for each in- and output buffer in lines 11 to 16. The method calls in 18 are optional tuning parameters except for run(), which executes the parallel_for.

each correspond to an iteration space point. As with any parallel_for, programmers must ensure that the function parameter does not assume any order in which the iteration space points are executed. Our API is implemented with a library and C++ source-to-source translator that generates CUDA, OpenCL, and OpenMP kernels, as well as code that implements slicing and chunked data transfers (see Section 3.3).

Figure 3.8 uses an example user implementation of matrix vector multiplication to demonstrate the parallel_for. The first two arguments of the parallel_for constructor are the start and end of the iteration space, and the third is the function parameter

(lines three to nine). Iteration points are grouped into successive *batches*. On GPUs batches correspond to OpenCL workgroups and CUDA blocks. Slices are composed of at least one batch. The current iteration and the batch size can be retrieved (line 4, and lines 11 and 12) with a respective function and data field. All buffers that are used by the function parameter are registered with attributes that describe the access direction and access pattern (lines 14 to 16). The access pattern attribute is `all_or_any` if each batch accesses either all elements of a buffer or the access pattern does not fit the attributes discussed below (line 14). The attribute is `successive_subsections` (lines 15 and 16) if successive batches access only successive contiguous subsections of a buffer. With this attribute programmers can simply specify how many buffer elements each batch accesses. Optional tuning parameters indicate how indices are used (first function call in line 18), or if the function parameter is idempotent, which informs the migration strategy (see Section 3.3.2). The `opt_set_simple_indices(true)` function call indicates that the `get_iteration()` and `get_batch_iteration()` indices are only used for memory accesses. `get_batch_iteration()` returns the current batch ID. As an optimisation, buffers allocated on the GPU are only large enough to contain data for a single slice if this tuning parameter is set, and if the buffers have the access pattern attributes `successive_subsections` or `continuous_subsections` (see below for an explanation of the latter).

We offer further API calls for more complex applications. For example, for applications with indirect memory access patterns or applications in which batches access overlapping buffer subsections an attribute called `continuous_subsections` can be parametrised with two function parameters that compute the start and end indices of the subsections based on the batch IDs. These function parameters can access other buffers for indirect memory accesses. To optimise execution on the GPU, address space qualifiers can be added to variables and buffers. Finally, the parallel_for can be specialised to a reduction with a method call.

One typical use-mode for our API is to code kernels from scratch. However, pre-existing OpenCL and CUDA kernels can be ported to it in a simple process. Original kernel code can be used as the function parameter for the parallel_for with minor modifications, like replacing CUDA barrier operations with our barrier function. Code for manual management of device buffers and data transfers is replaced with the memory access attributes for each buffer.

3.6 An Idealised Performance Model

This section shows analytically that mid-kernel migration outperforms kernel-by-kernel scheduling, which is typical for current systems. For this, we create idealised models of both and derive a maximum of 1.33x for the speedups that can be achieved by adding migration in our deployment scenario (see below), irrespective of the kernels and devices involved. Finally, insights into how these speedups change with system characteristics and migration time points are provided. In Section 3.7.3, application specific speedups based on this model serve as essentially unattainable idealised upper bounds, allowing us to evaluate the quality of our practical slicing implementation and its overheads. Our modelled scenario is composed of a new application kernel, and its *fast* and *slow* devices (see Section 3.2.1 for an explanation of the terms *fast* and *slow device*). The new kernel arrives while its fast device is temporary unavailable. This simplified scenario allows us to focus on the evaluation of the migration mechanism. The execution times on the fast and slow devices, and the waiting time for the fast device are normalised to the execution time on the fast device.

3.6.1 Simplifying Assumptions

The models make simplifying assumptions regarding the absence of some practical issues, including the ones discussed in Section 3.3. We assume that kernels can migrate and resume execution after a migration instantaneously. However, in an actual implementation, kernels can migrate only between slices and have to wait for input data to be transferred to the new target device. We also do not model interference caused by other kernels that might execute on the system. In more detail, GPU management code that is executed on the CPU can cause interference with kernels on the CPU. Similarly data transfers of other kernels or residual data transfers of the same kernel can also cause interference as discussed in Section 3.3.

3.6.2 Our Baseline Comparator System

In this section and our experimental evaluation in Section 3.7 mid-kernel migration is compared against the best possible implementation of the non-migrating kernel-by-kernel scheduling, which is typical of current systems [RVKP19, PS16]. These systems require unattainable knowledge for perfect scheduling decisions (see also Section 3.2). When a kernel arrives, and its *fast device* is occupied by another kernel or is otherwise

not available, current systems have two options: (1) wait for an unpredictable amount of time for its *fast device* or (2) launch earlier on an alternative but *slower device* without being able to migrate when a better one becomes available. The wrong decision can lead to serious slowdowns. A perfect scheduler for such systems would need to know how long the new kernel will run on its slow and fast devices, and when in the future its fast device will become available, information which is not always known.

As a comparator, we define a theoretical perfect scheduler which has this practically unattainable knowledge and, therefore, call it the *Perfect Non-Migrating Scheduler* (PNS). However, the PNS is incapable of mid-kernel migration and, therefore, limited to kernel-by-kernel scheduling decision. Because the PNS has knowledge about the future, actual systems presented in previous work can only be approximations of it [ATNW09, RVKP19, PS16]. Therefore, it is a harder reference point than any of these systems. Other than the PNS our scheduler does not rely on unattainable knowledge. Kernels are simply started on their slow device if the fast device is not available and are migrated to the fast device as soon as it becomes available (see Section 3.2.2).

3.6.3 The Scheduler

We schedule kernels with the First Come, First Served (FCFS) policy, as in previous work [RVKP19, PS16]. In more detail, in this section and the evaluation we compare *FCFS with mid-kernel migration* with *FCFS without mid-kernel migration* but with perfect knowledge about the future, the latter is implemented by the PNS.

3.6.4 Components of the Model

We will model idealised implementations of the PNS and a system with migration in order to derive the maximum speedup of the latter over the former. The models are idealised because they assume the absence of these practical issues: interference during migration, device setup costs, slicing overheads, and the fact that kernels cannot be migrated instantaneously (see Section 3.3 and Section 3.4.1 for a discussion of all of these). The models use the following components:

- k denotes the ratio of how much faster the new application kernel executes on its fast device than on its slow device.
- The normalised execution time on the slow device of a kernel is also k (now

as a number of time units) because it is, as above, the execution time on the slow device normalised to the execution time on the fast device. Because of this equality we use k for both in the rest of the chapter. In fact, the formula to compute the theoretical maximum speedup of a kernel through migration exploits this equality.

- Time $k-1$ marks a crucial transition point for the PNS. Recall that (because of normalisation) the new application will execute in *one* time unit on its fast device. Therefore, when the new kernel arrives, if PNS knows that the fast device will become available before time $k-1$, it is preferable to wait and execute it there. This will cause the new kernel to finish earlier than executing it immediately on the slow device. In contrast, if PNS knows that the fast device will only become available after time $k-1$, then it is preferable to execute it immediately on the slow device. At $k-1$ either decision results in the same execution time.

- δ is the difference between $k-1$ and the point in time at which the fast device becomes available.

Additionally, the components C_{fast}, C_{slow}, C_{PNS}, and C_{mig} denote the execution time of the new kernel on its fast device, its slow device, with the PNS, and with our migration mechanism respectively, and are used to derive the models. W_{fast} (waiting time) is the time starting from the arrival of the new kernel after which its fast device becomes available and is also referred to as waiting time for the fast device.

3.6.5 Speedup with Migration over the Perfect Non-Migrating Scheduler

The maximum speedups for the two choices available to the PNS are modelled separately. Our first model is a specialisation of Amdahl's Law [HP17] and our second model is related to it. Figures 3.1a and 3.1b depict the choices of the PNS and Figure 3.9 execution with migration.

PNS Choice 1) Immediately launch the new application kernel on its slow device

We distinguish two cases for the value of δ.

a) $\delta < 0$: This case is not possible. The choice made by the Perfect Scheduler implies that the execution time of the new application kernel on its slow device C_{slow}

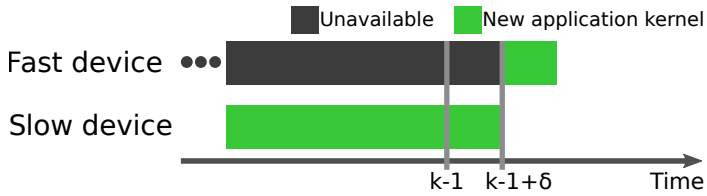


Figure 3.9: Execution with migration at time $k-1+\delta$. In contrast, the execution with the PNS is as shown in Figure 3.1a for negative δ and as in Figure 3.1b for positive delta.

is lower than the combined waiting and execution time with its fast device:

$$C_{slow} < W_{fast} + C_{fast} \tag{3.1}$$

However, if δ is negative then scheduling the application kernel on its fast device would result in a shorter combined execution and waiting time which stands in contradiction to the decision made by the PNS:

$$C_{slow} > W_{fast} + C_{fast} \tag{3.2}$$

$$\Leftrightarrow \quad k > k-1+\delta+1 \qquad ,k>1 \text{ and } \delta<0 \tag{3.3}$$

$$\Leftrightarrow \quad k > k+\delta \tag{3.4}$$

b) $\delta \geq 0$: In this case the execution time C_{PNS} with the PNS and C_{mig} with migration are

$$C_{PNS} = k \qquad ,k>1 \tag{3.5}$$

$$C_{mig} = k-1+\delta+\frac{1-\delta}{k} \qquad ,k>1 \text{ and } \delta \geq 0. \tag{3.6}$$

The speedup $S_1(k,\delta)$ of migration over the PNS derived from this is

$$S_1(k,\delta) = \frac{C_{PNS}}{C_{mig}} \tag{3.7}$$

$$\Leftrightarrow \quad S_1(k,\delta) = \frac{k}{k-1+\delta+\frac{1-\delta}{k}}. \tag{3.8}$$

PNS Choice 2) Wait for the fast device

Again, we distinguish two cases for the value of δ.

a) $\delta > 0$: This case is not possible. The choice made by the PNS implies

$$W_{fast} + C_{fast} < C_{slow}. \tag{3.9}$$

However, if δ is positive then scheduling the new application kernel on its slow device would have resulted in a shorter execution time which stands in contradiction to the decision made by the PNS:

$$C_{slow} < W_{fast} + C_{fast} \tag{3.10}$$

$$\Leftrightarrow \quad k < k - 1 + \delta + 1 \qquad ,k > 1 \text{ and } \delta > 0 \tag{3.11}$$

$$\Leftrightarrow \quad k < k + \delta \tag{3.12}$$

b) $\delta \leq 0$: The execution times with the PNS and migration are

$$C_{PNS} = k + \delta \qquad ,k > 1 \text{ and } \delta \leq 0 \tag{3.13}$$

$$C_{mig} = k - 1 + \delta + \frac{1-\delta}{k} \qquad ,k > 1 \text{ and } \delta \leq 0. \tag{3.14}$$

The speedup $S_2(k,\delta)$ derived from this is

$$S_2(k,\delta) = \frac{k+\delta}{k-1+\delta+\frac{1-\delta}{k}}. \tag{3.15}$$

3.6.6 Application Kernel and Device Independent Maximum Speedup

The maximum speedup over the PNS is 1.33x. Both speedups derived in the previous subsection are maximal at $k = 2$ and $\delta = 0$.

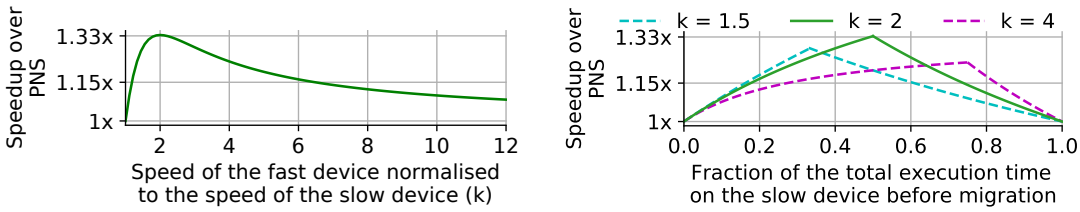
$$S_1(2,0) = S_2(2,0) = 1^1/_3 \tag{3.16}$$

3.6.7 Speedups with Different k

The maximum speedup for a particular application kernel depends on the performance difference between both devices, which is k. Figure 3.10a shows the maximum speedup over the PNS for different k. The speedup decreases for $k > 2$ because the larger the performance difference between the devices, the less progress can be made on an alternative slow device before the migration. For $k < 2$ the speedup decreases because the closer the performance of both devices, the less advantage can be gained through migration compared to full execution on the slow device.

3.6.8 Speedups with Different δ

The amount of work done on the slow device before the migration determines the speedup over the PNS. For example, if an application kernel migrates right after it



(a) Maximum speedup with migration over the PNS for different speedups of the application kernel on its fast device over execution on its slow device. This is a limit study of the speedup and so the best migration time point is used, which means δ is set to zero.

(b) Speedups for different migration time points, i.e. for different δ. See Section 3.6.4 for an explanation of k.

Figure 3.10: Speedups over the PNS with different relative device speeds (a) and migration time points (b).

Table 3.2: Details of the evaluation machine. The GPU uses the Kepler microarchitecture. DVFS and Turbo-Boost are deactivated.

CPU Model	Cores	Sockets	Hyperthreading	GPU Model
Intel Core i7-4770	4	1	Off	nVidia GTX Titan

launched or with virtually no work left, no significant benefit can be gained with migration over just waiting for its fast device or finishing the execution on its slow device. Figure 3.10b shows speedups over the PNS for different migration time points expressed as the fraction of the total execution time on the slow device. For k not equal to 2 the maximum is lower. The maximum moves to the right for $k > 2$ and to the left for $k < 2$.

3.7 Evaluation

3.7.1 Experimental Setup

Table 3.2 lists details of the evaluation platform. We use the Intel C++ Compiler (ICC) 19.0.5.281 with `-O3`, NVCC 10.2, version 455.23.05 of the nVidia GPU driver, Linux Kernel 5.3.18, and version 18.1 of the Intel CPU OpenCL runtime and compiler. Thirty samples are taken for each data point and the mean speedups are reported if not stated

otherwise. Error bars show the standard error of the mean and are in some cases barely visible.

3.7.2 Experimental Method

The goal of the experiments is to measure speedups obtained through mid-kernel migration over an ideal implementation of current systems that provide migration flexibility for heterogeneous systems in a programmer transparent way. These systems make scheduling decisions on a kernel-by-kernel basis [RVKP19, PS16] (see also Section 3.6.2). We leave comparisons against systems with non-transparent but more fine-grained migration flexibility for future work (see Section 5.3.1).

The kernel-by-kernel approach assumes perfect scheduling decisions but cannot migrate application kernels once they are launched, in other words it is an implementation of the Perfect Non-Migrating Scheduler (PNS) introduced in Section 3.6.2. For the evaluation we use the scenario described in Section 3.6, in which the fast device[5] of the new kernel is initially unavailable. Speedups for different migration time points are measured because speedups change in response to how long the fast device is unavailable, as discussed in Section 3.6.8. Additionally, the geometric mean of the per migration point speedups are reported to determine if migration benefits overall performance. For a fair comparison, all sample points are equally spread out. The runtime costs of the slice size models are included in the measurements.

We do not use iteration space slicing on the fast device as discussed in Section 3.3.1 because in our evaluation scenario kernels never migrate from their fast to their slow device. We determine an application independent data transfer chunk size that introduces a maximum slowdown of 0.5% for parallel_for and 1.5% for reductions as discussed in Section 3.3.3. For practical reasons, we double the tested chunk size with each search step starting with a chunk size of 1MB. The chunk size is 64MB for standard parallel_for and 16MB if the parallel_for is specialised to a reduction (see Section 3.5).

Results are reported with nine benchmarks from the SHOC and Rodinia benchmark suites (see Table 3.3) [CBM+09, DMM+10]. We use all benchmarks of SHOC and Rodinia that consist of a single kernel that is not invoked repeatedly and are therefore relevant to mid-kernel migration. Current runtime systems are not applicable to

[5] We determine device affinities experimentally. In a real deployment this would be replaced with the predictive models of prior work [GWO13, TMW17] or the programmer would set the fast device as is the case with OpenCL and implicitly with CUDA.

Table 3.3: Application kernel instances. The fast devices of the kernels, and the relative speeds of the fast and slow devices, which are indicated by k are determined experimentally (see Section 3.6.4, and Section 3.7.2 for an explanation of k). Slice counts are computed based on the predicted slice sizes. The final column shows the prediction accuracy, which is the training slice size divided by the predicted slice size. Varying accuracies might reflect the sensitivity of the benchmarks to the slice sizes. We cannot report the accuracy for two benchmark instances because we could not generate training data with them as explained at the end of Section 3.7.2.

Suite	Bench.	Input size	Fast dev.	CPU impl.	Slices	k	Pred. Acc.
Rodinia	B+Tree Find K	Small	GPU	OpenCL	507	2.74	442.81
		Medium	GPU	OpenCL	1014	3.05	442.81
		Large	GPU	OpenCL	2264	2.95	659.32
	B+Tree Find Range	Small	GPU	OpenCL	949	2.45	920.45
		Medium	GPU	OpenCL	1756	2.84	920.45
		Large	GPU	OpenCL	3161	2.54	920.45
	Nearest Neighbour	Small	*CPU*	OpenCL	53	2.35	20.89
		Medium	*CPU*	OpenCL	103	2.34	41.78
		Large	*CPU*	OpenCL	230	2.37	41.78
SHOC	MD5-Hash	Small	GPU	OpenCL	53	12.31	-
		Medium	GPU	OpenCL	115	12.94	0.71
		Large	GPU	OpenCL	245	13.02	0.64
	FFT	Small	*CPU*	OpenCL	36	1.32	57.3
		Medium	*CPU*	OpenCL	72	1.32	57.3
		Large	*CPU*	OpenCL	108	1.31	57.3
	GEMM	Small	GPU	OpenCL	64	7.19	1
		Medium	GPU	OpenCL	121	5.54	16
		Large	GPU	OpenCL	144	8.44	1
	Inverse FFT	Small	*CPU*	OpenCL	36	1.32	57.3
		Medium	*CPU*	OpenCL	72	1.31	57.3
		Large	*CPU*	OpenCL	108	1.31	57.3
	Reduct.	Small	*CPU*	*OpenMP*	3	3.19	0.02
		Medium	*CPU*	*OpenMP*	6	3.20	0.02
		Large	*CPU*	*OpenMP*	12	3.21	-
	SPMV	Small	*CPU*	OpenCL	15	3.47	0.17
		Medium	*CPU*	OpenCL	22	3.36	0.13
		Large	*CPU*	OpenCL	30	3.23	0.1

applications that consist of such single kernels because they make migration decisions only on a kernel-by-kernel basis as discussed in Section 3.6.2. For measurements with migration capable implementations all benchmarks are implemented with our parallel_for (see Section 3.5). To demonstrate the applicability of our API to kernels with indirect memory accesses we also implement SPMV with parallel_for in our experiments. However, as explained in Section 3.4.5 we assume that our runtime system knows when it executes SPMV and can use this knowledge to choose better slice sizes as this is what would happen in a realistic library deployment. We measure execution time spent in a region of interest that includes all code sections of the benchmark (kernel execution, data transfers, migration management, etc.) except the generation or reading in of input data. As shown in Table 3.3 both CPU and GPU are about equally often represented as the fast device, and the benchmarks cover a wide range of values for the performance difference k between the devices, which is introduced in Section 3.6.4.

We use CUDA on the GPU, and OpenCL for all CPU implementations except reduction for which we use OpenMP. We experimentally determined that CUDA performs better or equally well than OpenCL on the GPU, and that OpenCL outperforms OpenMP on the CPU in the majority of cases. OpenMP improves performance only marginally otherwise, except for reduction for which OpenMP performs significantly better. Reductions are a distinct computational pattern and so the best implementation can be identified before the system is deployed with one-off costs. Additionally, our runtime system knows when reductions are executed through our programming model (see Section 3.5). Because SHOC uses only CUDA and OpenCL, we ported the "reduction" benchmark to OpenMP with the `reduction` clause.

We replace the outdated standard problem sizes of the benchmarks with three larger ones that roughly require these execution times on the slow device: 200-250ms (small), 400-500ms (medium), and 800-1000ms (large). The execution times on the fast devices range from 17ms to 784ms.

To train the slice size predictors we use leave-one-out cross-validation, which means we train the predictors separately for each benchmark, using only the other benchmark kernels as the training set. This way the predictors choose a slice size for a kernel they have not seen before.

Measurements with the PNS implementation, with which we compare migration, do not include CUDA and OpenCL setup times. The migration-capable implementations benefit from setup taking place ahead of time in the daemon. To focus on

migration, we factor this out by giving the PNS the same benefit.

Deliberate slicing can improve performance even if an application kernel never migrates, for orthogonal reasons such as better use of the caches [KJKD13, CHZ+18]. To avoid unfairly disadvantaging the PNS implementation through these effects we allow it to use slicing, but without migration, if this improves its performance. We determine experimentally for each benchmark instance if this is the case.

For the training data we brute-force slice sizes as described in Section 3.4.3 with all nine benchmarks on the CPU and GPU irrespective of which one is the fast and slow device. We use the OpenCL implementation of reduction instead of the OpenMP implementation to create training data for the CPU model. We use our small, medium, and large input sizes except for two cases. We do not use the large input of reduction because the Intel OpenCL implementation does not support buffers larger than 4GB. Instead, we use a slightly smaller input. We do not use the small input of MD5Hash because a slice size with acceptable overheads, which we set to 1.04x slowdown or less (see Section 3.4.3), cannot be found on its fast device. This is because the execution time of this benchmark instance is very low with less than 20ms and so a larger share is spent in fixed overheads than in the other benchmark applications. We use a larger input instead to generate training data.

3.7.3 Speedups Over the Perfect Non-Migrating Scheduler

Figure 3.11 shows speedups with migration over the implementation of the PNS (see Section 3.6.2 and Section 3.7.2). As discussed in Section 3.6.2, the PNS includes a perfect scheduler and is, therefore, at least as good as any possible implementation of current mid-kernel migration incapable systems. Mid-kernel migration outperforms the PNS in all cases, as shown by the geometric means of the speedups. The maximum and average speedups are 1.30x and 1.08x. Figure 3.11 supports contribution (5) of Section 3.1 in two ways. Firstly, it shows that mid-kernel migration outperforms current systems, which are represented by the PNS, for all benchmark instances as indicated by the geometric mean speedups. Secondly, the figure provides detailed insights into the performance behaviour of mid-kernel migration at different migration time points during the execution of the kernels.

The remainder of this subsection first explains Figure 3.11 by describing the structure of one of its subgraphs, and then the experimental results in general. The subgraph in the top left corner shows speedups with mid-kernel migration over the PNS with the

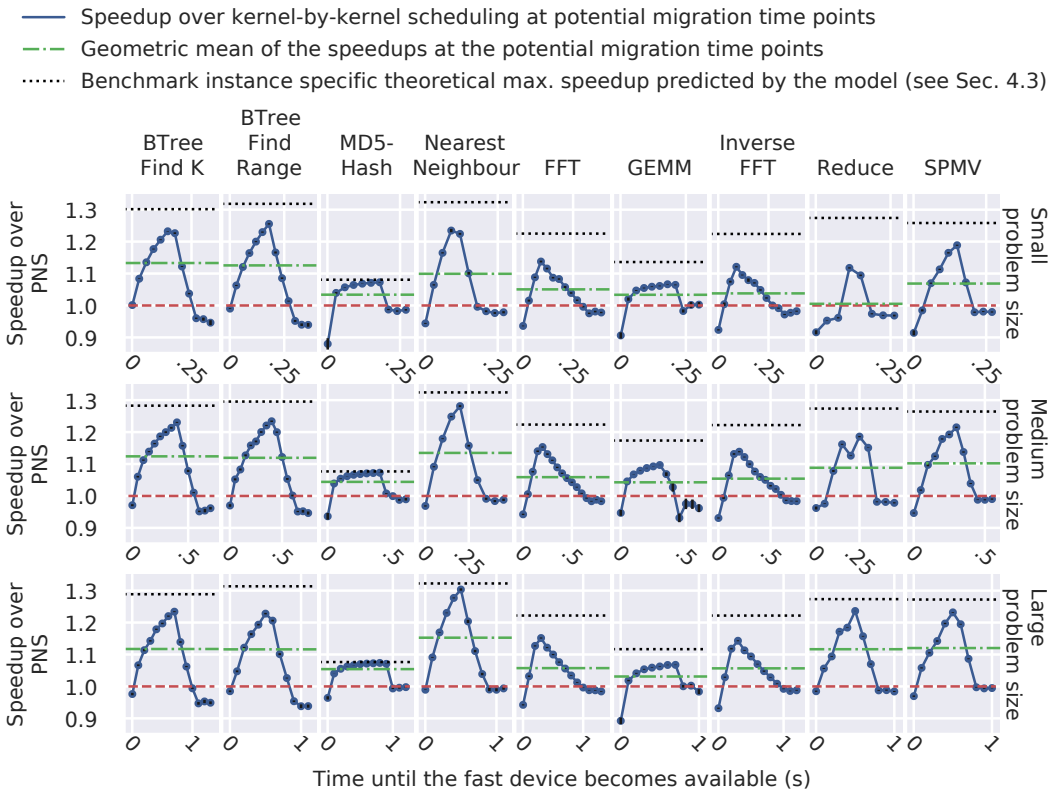


Figure 3.11: Speedups with mid-kernel migration (blue dots) and their geometric means (green lines). We present speedups over current systems, which are represented by the PNS. As discussed in Section 3.6.2, the PNS is better than any possible real implementation. Mid-kernel migration outperforms current systems in all cases as indicated by the geo. means. Insights into the performance behaviour at different migration time points are also provided. Each value on the x-axes corresponds to a distinct potential migration time point. The green lines are the geo. means of the per migration point speedups, which are the blue dots. The red lines indicate the break-even speedup of 1x. The theoretical maximum (black dotted lines) is benchmark instance specific because it depends on k, which is the speedup of a benchmark on its fast over execution on its slow device as discussed in Section 3.6. At the arrival of a new benchmark its fast device remains unavailable for the time on the x-axis, so the benchmark executes for this time on its slow device before migrating. Migration causes slowdowns when a kernel migrates soon after its start, as shown on the left-hand sides of the subgraphs but speeds up the cases in the middle sections. Migration benefits performance of all kernels because the speedups outweigh the slowdowns as indicated by the geo. means of the speedups above 1x.

B+Tree Find K benchmark and its small problem size. The blue dots show speedups at distinct potential migration time points. For example, at the third blue dot the fast device of the kernel is unavailable for 60ms starting from the arrival of the benchmark. In this case, the PNS decides to wait for the fast device. In contrast, with mid-kernel migration the benchmark makes progress on its slow device for the first 60ms instead of waiting, and then migrates to its fast device. In this case, mid-kernel migration is 1.14x faster than the PNS. The speedup closest to the theoretical maximum (indicated by the black line), occurs with a migration at time 150ms. This maximum is instance specific because it depends on k, which is the kernel specific speedup on its fast over its slow device. For this benchmark k is 2.74 as detailed in Table 3.3. The green line shows the geometric mean of the speedups at the potential migration time points (blue dots). Migration statistically benefits overall performance because the geometric mean is above 1x, as indicated by the red line. The points on the x-axis are *potential* migration points because migration does not happen at all of them as discussed below.

At the left-hand end of each subgraph, slowdowns are visible for small waiting times for the fast device. In these corner cases the new application kernel has spent little or no time on its slow device before it migrates and so the PNS chooses to wait for the fast device. Thus, little or no advantage can be gained from the ability to use the slow device before the fast one is available. In the worst-case, interference during the migration caused by the slice kernel, which has started on the slow device, negates all benefits of mid-kernel migration and causes a slowdown. MD5Hash, and GEMM experience the worst slowdowns in this area. However, they also have the shortest execution times of all benchmarks on their fast devices.

As expected, migration outperforms the PNS in the middle section of the subgraphs because application kernels can use the slow device first and then the fast device as discussed in Section 3.3. On the left-hand side of the middle section of each subgraph the PNS decides to wait for the fast device. If the time until the fast device becomes available, which is the value on the x-axis, is high enough the PNS decides not to use the fast device and launches the kernel immediately on the slow device (see Section 3.6.5).

The maximum speedups in these areas are up to 96% (and 74% on average) of the theoretical maximum speedups (see Section 3.6) that assume that kernels can migrate at any point in time, execution on the slow device does not cause interference, and no work is thrown away due to slice aborts (see Section 3.3). Reduction with the small input has the largest gap between the measured maximum speedups and the theoretical maximum. This is because the predicted slice size divides the iteration space into

very few slices (see Table 3.3), this in turn means that significant amounts of work are thrown away when the kernel migrates because the slice that is at that time on its slow device is aborted (see Section 3.3.3).

Towards the right-hand end of each subgraph the new application kernel does not migrate because its fast device is unavailable for longer than the execution time of the new application kernel on its slow device. The PNS chooses to execute the new kernel on its slow device in these cases but has no slicing overheads. Most of these tail points are just under 1x and the geometric mean of the slowdowns on the tails is 2.34%. This is as expected because the training slice sizes for the predictors have overheads of up to 2% to 4% (see Section 3.4.3)

The geometric means in Figure 3.11 exclude the final three points from the tail towards the right-hand end of each figure. This is because the means are intended to capture the trade-off between those areas of the subgraph in which our technique generates a speedup, and those areas in which it generates a slowdown. Since the right-hand end tail is arbitrarily extendable (i.e. the fast device could be unavailable for an arbitrary amount of time), and since it inevitably converges to be close to one, its impact would eventually swamp the mean and also converge it to one. This would leave us with no useful information about the areas of practical interest. The maximum per kernel instance geometric mean speedup is 1.15x with the Nearest Neighbour kernel and the large problem size. The same has been applied to the overall average stated above for the same reason. The rightmost tail points are above or below their preceding points in some cases. We confirmed experimentally that this is not the general trend of the subsequent migration time points. Subsequent points are closer to the other two tail points shown. In summary, mid-kernel migration outperforms the PNS in accordance with the model of Section 3.6 by up to 1.30x and 1.08x on average.

3.7.4 Overheads in the Absence of Migration

This section shows that our implementation of the migration mechanism introduces only small slowdowns of 2.44% on average if an application kernel never migrates from the slow device to the fast device. Time spent in additional code required for migration, like slicing code, which is continuously executed on the slow device introduces overheads that cannot be amortised in these situations (see Sections 3.3 and 3.4.1). Figure 3.12 shows execution times on only the slow devices with the migration capable implementations normalised to execution times without the migration mechanism. The

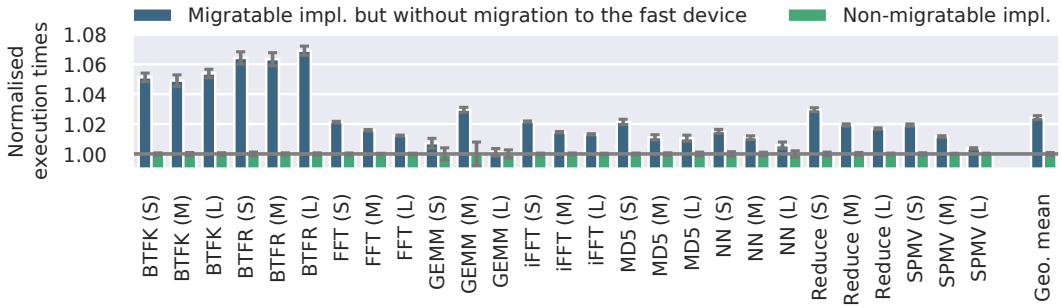


Figure 3.12: This figure shows that the overheads of slicing if a kernel never migrates are small. Blue bars are execution times only on the slow devices but with our migration mechanism and slice size prediction, and green bars are execution times without slicing normalised to the latter (lower is better). All green bars have a height of one. The letters in brackets indicate the input sizes **s**mall, **m**edium, and **l**arge (see Section 3.7.2). **BT**ree **F**ind **K**, **BT**ree **F**ind **R**ange, Inverse FFT, MD5Hash and **N**earest **N**eighbour are abbreviated to BTFK, BTFR, iFFT, MD5, and NN respectively. We collected 100 samples for each data point.

slice size choices by the predictors also govern the slowdowns because they determine how often the slicing code is executed. Therefore, the results show that slicing can be implemented with low overheads and that predictors can learn slice size choices that introduce acceptable overheads in cases where a kernel never migrates.

3.7.5 Code Size Reduction with Parallel_For

Our parallel_for-based programming model hides the complexity of mid-kernel migration. Table 3.4 demonstrates that the significant complexity that a manual implementation of mid-kernel migration would introduce can be hidden with our parallel_for. The parallel_for based code is at least an order of magnitude smaller for three reasons. Firstly, it implements code required for an efficient implementation of migration, which is discussed in Section 3.3, like slicing aware data transfers and chunked transfers. Secondly, programmers do not have to provide multiple versions of the same code for the target devices in CUDA, OpenCL, and OpenMP. Compared to a manual implementation our parallel_for reduces the code size by at least 88%.

Table 3.4: This table shows that mid-kernel migration would introduce significant complexity if implemented by hand, and that our parallel_for removes this complexity from application code. Lines of code (LOC) for code that is only concerned with migration are provided in brackets. The total LOC for the hand-implementation (not in brackets) includes this code. The LOC do not include comments, includes, defines, and blank lines. The final column presents the code size reduction.

Benchmark	LOC of the hand implementation	LOC with our parallel_for	Reduction
BTree Find K	731 (371)	55	92%
BTree Find Range	824 (382)	74	91%
Nearest Neighbour	634 (402)	29	95%
FFT	559 (392)	30	95%
Inverse FFT	559 (387)	30	95%
GEMM	633 (371)	67	89%
MD5Hash	598 (459)	74	88%
Reduction	589 (382)	6	99%
SPMV	820 (481)	45	95%

3.8 Summary

Operating Systems do not yet have the same fine-grained migration capabilities for heterogeneous systems that they have for CPUs. We provide more flexibility in this respect, including the ability to switch between the underlying CUDA, OpenCL, and OpenMP runtimes. The complexity of this mechanism, management code, and execution strategies, that are required for an efficient implementation, is hidden behind the parallel_for skeleton. The semantics of this skeleton guarantee to the runtime system the independence of its sub-computations. This information is key for the proposed mechanism.

Mid-kernel migration has two benefits: firstly, kernels can utilise better devices when they become available mid-kernel, and secondly, in our evaluation scenario, perfect scheduling decisions that require unattainable knowledge can be replaced with a better scheduling policy that does not require such knowledge. We show analytically that mid-kernel migration outperforms current systems by up to 1.33x, and that an idealised implementation of the new scheduling policy never performs worse than current systems even if they make perfect scheduling decisions. Our experimental results show

that mid-kernel migration performs better than current systems by up to 1.30x, 1.08x on average, and introduces an average slowdown of less than 2.44% if kernels never migrate.

Chapter 4

Autotuning Parallel Hard Real-Time Systems

This chapter presents the first fully self-tuning and timing predictable skeleton for hard real-time systems (see Section 2.5 for an explanation of hard real-time systems). Please note that this chapter does not consider GPUs. We presented our work on heterogeneous systems in Chapter 3.

4.1 Introduction

High throughput applications with timing constraints, such as autonomous vehicles, drive the development of parallel real-time systems [UBG+13, YAY+18, AUT14, TSV+20, dD19]. To program these, programmers have to resort to low-level programming models such as message passing and threads [Hun19, Ope18, Ada16, MSH11, XMO15], (see Section 2.5.3.4). These are considered error prone, non-portable and inefficient in terms of programmer productivity [Gor04, Lee06, LPSZ08, SL05]. As previous work points out, new high-level programming models for real-time systems are therefore needed [HGL12]. Task graphs, which have attracted considerable attention from the theoretical real-time scheduling community (see Section 2.5.3.2), are a very general application model that can capture any possible interaction among threads [HJGG19] (see Section 2.5.3.2). In contrast, we propose to constrain the application structure to a set of composable skeletons to improve programmability, resource usage and timing analysis at the price of often expendable generality w.r.t. task graphs. Individual skeletons may cover separate portions of an application. Therefore, we envision a framework of different real-time skeletons that can be composed to imple-

ment full applications as is the case for mainstream parallel systems [Col04, DG08, ADK+11, EK10].

As a first step towards such a framework we conduct a case study with the *job farm* skeleton [1] that is applicable to a wide range of applications [ADK+11, DSDMT+17, UBF+16]. Job farms lend themselves to programs that process streams of input data such as signal processing, graphics, and networking applications [KCLL+05, KRD+03, BBM+12, BLC16]. Thies et al. report 51 applications that use farms with a median of eight farm instances per application [TA10]. These applications include a "Ground Moving Target Indicator", "2D Inverse Discrete Cosine Transform", and "Fast Fourier Transform".

As explained earlier in this thesis, structural information encoded in skeletons can be used to automatically tune applications. As an example of this, we introduce *job batching* for shared memory parallel real-time systems. Batching reduces the overheads that come with parallelism and so decreases the required core counts. Alternatively, it allows the use of less powerful and so cheaper hardware, or adding additional workload without increasing resources. We show that batching is viable in the context of real-time systems and that it can be implemented transparently to developers with the farm skeleton. To further ease programming, we devise an analytical framework for the computation of farm internal parameters, which would have to be carefully chosen by hand otherwise. Using this framework, we implement Peso [2], a deterministic[3] and self-tuning *farm* skeleton library for the hard real-time XMOS xCore-200 microcontroller.

We show experimentally that parameter choices that Peso makes are the same or are very close to the best parameter choices that we determine through brute force searches and never cause deadline misses. Batching reduces the minimum task period that can be sustained by a given application and core count by 22.38%. Therefore, it can improve the throughput by the same percentage or reduce the core count. Finally, we show that the overheads introduced by Peso over hand-coded solutions are negligible.

[1] We remark that this skeleton is normally called "task farm" in the parallel computing community. However, to not overload the term "task" with conflicting interpretations from real-time and non-real-time contexts, we use the term *job farm*, which, we believe, better represents our intended interpretation in real-time systems. See Section 2.5.1 for an explanation of real-time *jobs*.

[2] Available at: rtas2020.paulmetzger.info

[3] The WCETs of farm internal code and an upper limit for the time between input arrival and the availability of corresponding results can be determined. This implicitly also means that if the farm is executed multiple times with the same set of inputs, the assignment of input to farm worker is always the same.

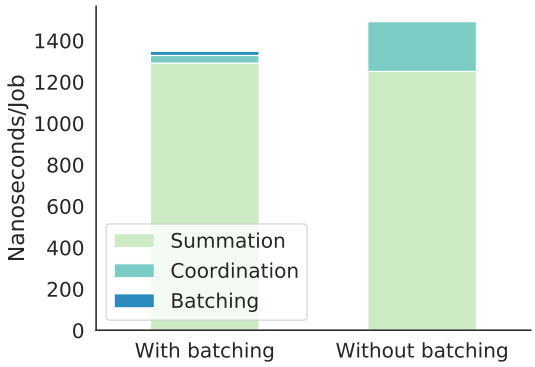


Figure 4.1: Breakdown of the per job execution time in a farm worker with and without batching for a simple example application that sums up 30 integers per job (less is better). 10 jobs are aggregated to a batch on the left-hand side.

The remainder of this chapter is structured as follows: Section 4.2 motivates and describes job batching. Section 4.3 describes our system model. Section 4.4 presents our analytical framework for farm parameters. Section 4.5 introduces our Peso library. Sections 4.6 and 4.7 present the experimental setup and results. Finally, Section 4.8 summarises this chapter.

4.2 The Case for Job Batching and Self-Adaptation

This section motivates batching with a simple example and argues for a self-adaptive implementation to further ease programming.

4.2.1 Reduced Core Count via Job Batching

Passing a *job* (see Section 2.5.1) through a job farm incurs bookkeeping overheads as some of the execution time is spent by coordination between producer, worker, and consumer threads. In the case of our evaluation platform and library, this means passing a pointer to the in- and output data of a job from the producer to a worker, and from a worker to the consumer. Note, the costs of this do not depend on the in- and output size of a job. These coordination costs can be substantial for tasks with short running rapidly arriving jobs. We propose *job batching* to reduce these overheads.

In a simple farm (with no batching), jobs are executed immediately when their input data and the necessary computing resources are available. Job batching exploits the slack time to the deadline to reduce communication overheads. With job batching,

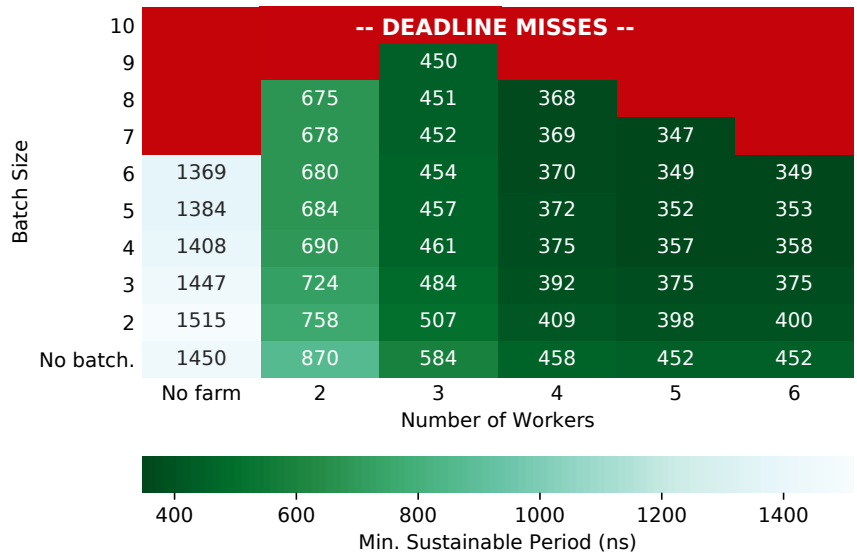


Figure 4.2: Measured minimum supported task periods with increasing batch size and worker count on our evaluation platform (see Section 4.6.1). Each job computes the sum of 30 integers and the relative deadlines are 15μs.

jobs are halted and aggregated to be then dispatched and processed in batches. This way communication costs are spent only per batch and not per job. Reduced overheads in turn allow tasks to run on cheaper hardware with fewer cores.

As a preliminary investigation, we break down the execution time that each job of a simple example task spends in a worker with and without batching. Figure 4.1 shows this breakdown for a benchmark. Job batching reduces the communication costs by 10× here and introduces a small overhead that comes from additional instructions that implement batching. The number of instructions executed during each summation is slightly higher when batching is used as the instructions generated by the compiler are slightly different.

Multiple instances of this simple computation need to be run in parallel if new input data arrives rapidly and a single core cannot meet the target period (see next section). In this example job batching allows for 12% lower task periods than without batching. Section 4.7.2 presents a quantitative study of the benefits of batching.

4.2.2 Improved Ease of Programming Through Self-Adaptation

The number of jobs in batches and worker counts have to be carefully chosen to avoid deadline misses. Choosing these parameters is non-trivial for developers as the batch size and worker count parameter space is difficult to navigate. For example, Figure 4.2

shows the minimum supported periods for all possible parameter combinations with the same simple task used in Figure 4.1 on our evaluation platform. As can be seen if this task has a period of 400ns then four workers and batch sizes larger than two are best. Given a task and hardware platform it is not obvious what the best parameter choice is if data like the one in this heatmap is not available. Without our analytical framework, this data can only be attained through a time consuming and so often impractical exhaustive search.

4.3 System Model

The relation between the number of workers, batch size, and characteristics of the farm workers is established in this section. The presented system model allows the implementation of a farm skeleton to automatically pick the minimal number of workers required to meet the periods and deadlines of applications.

4.3.1 Jobs, Job Releases, and Deadlines

The workload to be executed by the job farm is modelled by a periodic *task* that releases a sequence of jobs.[4] The k-th job is released at time

$$r_k = (k-1)\,T, \qquad k = 1,2,\ldots$$

with T being the *period* of job releases. When a job is released, it processes its input data and generates the corresponding output data. All jobs have a *deadline* D relative to the release instant. This means that the k-th job cannot finish later than

$$d_k = r_k + D = (k-1)\,T + D.$$

We do not set any constraint on the deadline (neither implicit nor constrained deadline model). Hence, we assume to have an *arbitrary* deadline.

The *response time* R_k is the time taken by the k-th job to complete starting from its release at r_k. Hence, no job misses any deadline if

$$\forall k = 1,2,\ldots, \quad r_k + R_k \leq d_k$$

which is equivalent to

$$\forall k = 1,2,\ldots \quad R_k \leq D. \tag{4.1}$$

[4]Thanks to Enrico Bini for contributing most of Section 4.3.1 in the course of our collaboration on this chapter.

The computation of the response time R_k depends on several scheduling decisions and is investigated in Section 4.4.

4.3.2 Cores and Batch Size

To minimise the communication costs incurred through parallelisation, jobs are grouped in *batches* of size b. An entire batch of b jobs is then executed on the same core. The number of available cores that process batches is denoted by m. Usually the term "worker" is used to denote a thread that executes the worker function of the jobs that are assigned to it, while a "core" is a physical piece of hardware capable of executing instructions. However, from a scheduling point of view the distinction between these two notions vanishes, in our case, because we assume a static one-to-one assignment from workers to cores. Hence, we use the terms *worker cores* and *workers* interchangeably.

For the analysis we assume that code executing on one core cannot influence the WCET of code on another. We discuss when this holds for our evaluation platform in Section 4.6.3. Situations in which this does not hold are handled by our implementation (see Section 4.6.3).

4.3.3 Execution Time

Introducing job batching requires a deeper understanding of the job execution time, which goes beyond a single worst-case execution time (WCET). For this reason, we split the job execution time into time intervals, which map to the execution phases of Figure 4.3. Most terms stand for time spent in a worker and are denoted with $C_{\mathsf{W}\ldots}$.

- C_{D} is the execution time spent in the dispatcher (see Sections 2.5.2 and 4.5.2).
- C_{com} is the latency of farm internal communication that is required for the coordination between dispatcher, workers, and aggregator. Therefore, this is application independent. C_{com} does not include execution time that is required to tear down a communication channel on the sender side and set up a channel on the receiver side because these instructions do not contribute to the latency. More specifically, this is the communication delay between dispatcher and workers, and workers and aggregator (see Sections 2.5.2 and 4.5.2).
- C_{Wc} is the execution time spent by a worker in farm internal communication.

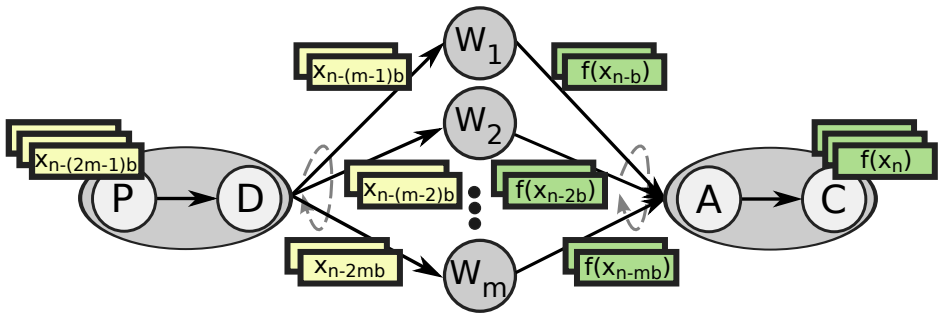


Figure 4.3: Illustration of a farm implementation. The producer (P) generates jobs that are scheduled in batches of size b over m workers (W) by a dispatcher (D). An aggregator (A) receives results and sends them to the consumer (C). Grey ellipses indicate that producer and dispatcher, and aggregator and consumer share a core each. Workers execute on their private cores. Light and dark green boxes are in- and output data. Drawn through and perforated arrows indicate communication channels and round robin scheduling.

Unlike C_{com}, it includes the code necessary to set up and tear down the communication channels. This corresponds to the *coordination* time in Figure 4.1.

- C_{Wsetup} is the execution time of code that sets up the execution of a batch and so is executed once per batch. Hence, in the special case when $b = 1$ (no job batching) this term is $C_{\mathsf{Wsetup}} = 0$.

- C_{WonceJ} is the execution time of code that implements batching and so is executed once per job. For the same reason as above, if $b = 1$ then $C_{\mathsf{WonceJ}} = 0$. The sum of C_{Wsetup} and C_{WonceJ} corresponds to the time spent in *batching* in Figure 4.1.

- C_{Wuser} is the execution time for the user provided worker function. This function is executed once per job. This corresponds to the time spent in the *summation* in Figure 4.1.

- C_{A} is execution time that is spent in the aggregator (see Sections 2.5.2 and 4.5.2), once per batch. Note that no reordering is necessary in the aggregator, because inputs are dealt out in a round-robin fashion, and likewise results are collected from workers in a round-robin fashion, as illustrated in Figure 4.3.

- C_{C} is the execution time to unbatch the results of a job. This takes place before results are used by the consumer and is necessary to hide batching from

application developers. Since unbatching happens sequentially

$$C_{\mathsf{C}} \leq T \tag{4.2}$$

must hold. Otherwise, the unbatching phase is overloaded. As above for C_{Wsetup}, if $b = 1$ then $C_{\mathsf{C}} = 0$.

The concrete execution times that we used for our experiments are listed in Table 4.1. These execution times may be summed up depending on whether they are executed once per job or once per batch. To highlight these two portions of time, we define the following quantities:

- $C_{\mathsf{WonceB}} = C_{\mathsf{Wc}} + C_{\mathsf{Wsetup}}$ subsumes execution time that is spent once per batch.
- $C_{\mathsf{WfullJ}} = C_{\mathsf{WonceJ}} + C_{\mathsf{Wuser}}$ subsumes execution time of code that is executed once per job.

Finally, we also set

- $C_{\mathsf{WfullB}} = C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b$, which is the execution time required to process an entire batch.
- $C_{\mathsf{O}} = C_{\mathsf{A}} + 2C_{\mathsf{com}} + C_{\mathsf{D}}$, which are the overheads of code that implements the parallel execution. C_{com} is multiplied by two to account for the communication between dispatcher and a worker, and a worker and the aggregator. This is the only term that subsumes execution time spent outside the workers.

From the perspective of a farm, jobs arrive when the input data associated with a job is ready to be processed. Therefore, we do not introduce a term for any external input data preparation.

Two example schedules of the same sequence of jobs with and without batching are illustrated in Figure 4.4. As shown in Figure 4.4b, thanks to the savings of communication cost, one worker core less is required if three consecutive jobs are grouped in a batch, at the price of an increase of response time. If the deadline D of a task allows for larger response times as in Figure 4.4b then batching reduces the number of cores required for a task. Communication blocks correspond to C_{com} (see above). The next section is dedicated to the formalisation of this qualitative argument.

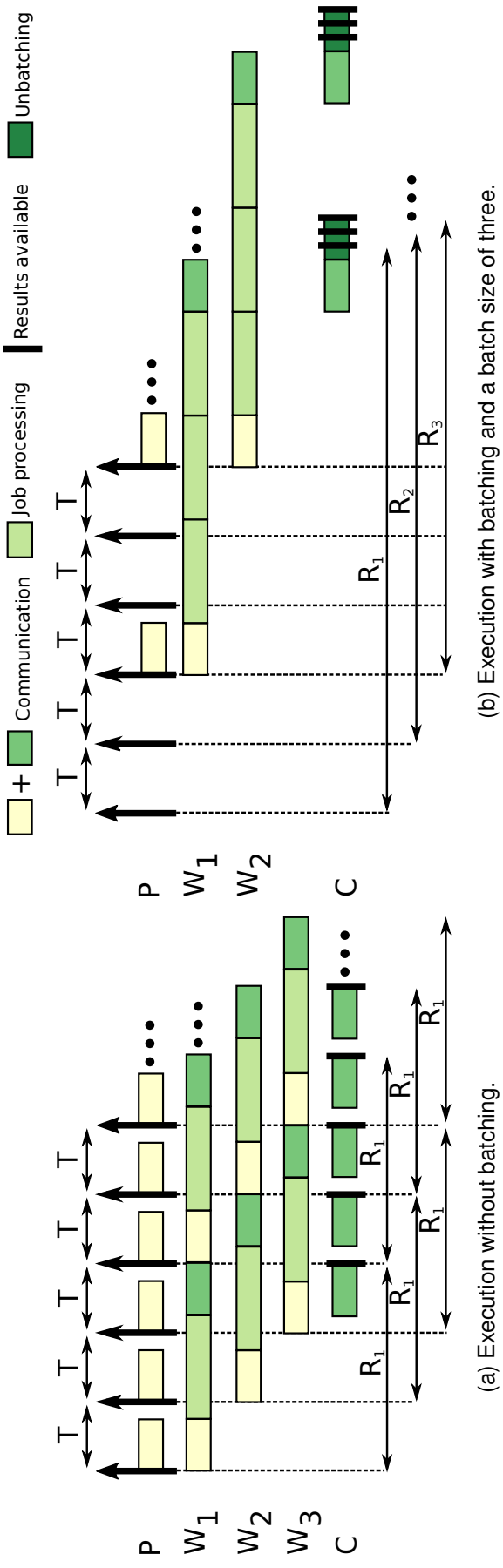


Figure 4.4: Illustration that shows that batching reduces the required worker core count at the expense of longer job response times R_k. (a) and (b) illustrate the execution of a job farm with three and two workers (W_i), a producer (P), a consumer (C), and with the same period in both subfigures. Time flows from left to right. (a) does not use batching, but (b) does with a batch size of three. The farm in (b) needs fewer worker cores for the same task due to batching. The response times are higher in (b) than in (a) because jobs have to wait for other jobs in the same batch. Some of the arrows in (b) that indicate the arrival of a new job are not followed by communication because batches are only dispatched after enough jobs have been accumulated. The index k of R_k in (b) indicates the relative position of a job in its batch.

4.4 Our Analytical Framework: Analysis of Batch Scheduling

This section presents an analytical framework that allows farm skeleton implementations such as Peso to automatically choose the number of worker cores and the batch size. Firstly, we compute the minimum worker core count m as a function of the batch size b and demonstrate that the required worker core count m decreases with the batch size b. As the intuition suggests, we show that the job response times increase linearly with the batch size b. Hence, the job deadline sets a natural upper limit on the batch size b. The maximum feasible batch size b_{max} is then the value that:

- maximises throughput if the core count is given, or
- minimises resource usage if the job period T is given.

4.4.1 Worker Core Count vs. Task Period

We establish here the relationship between the task period T, which determines the required throughput, and the worker core count m, which determines the available throughput. Clearly, a shorter task period T needs more worker cores m and vice versa.

The first step is to define the minimum sustainable period.

Definition 1. Given m worker cores and a batch size of b, we define the *minimum sustainable period* $T_{\mathsf{min}}(b,m)$ of a task as the minimum period which does not cause overload and so deadline misses.

For an arbitrarily small period T the job farm will be overloaded at some point, while it will never be overloaded for an arbitrarily large T.

As a first step, we compute the minimum period $T_{\mathsf{min}}(b,1)$ that can be sustained if only a *single* worker is used. A batch of b jobs is ready to be processed every $b \times T$ time units. The time required by a worker to process such a batch is C_{WfullB}. Hence, with only one worker no overload occurs as long as

$$b \times T \geq C_{\mathsf{WfullB}} \qquad b = 2,3,\ldots \tag{4.3}$$

which means that

$$\begin{aligned} T_{\mathsf{min}}(b,1) &= \frac{C_{\mathsf{WfullB}}}{b}, \qquad b = 2,3,\ldots \\ &= \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{b} \\ &= \frac{C_{\mathsf{Wc}} + C_{\mathsf{Wsetup}} + (C_{\mathsf{Wuser}} + C_{\mathsf{WonceJ}})b}{b}. \end{aligned} \tag{4.4}$$

Note that Equation (4.3) is only concerned with hypothetical job farms with a single worker and so does not stand in contradiction with Figure 4.4, which uses multiple workers.

Following the same arguments that we used to derive (4.4), if m worker cores are available (instead of only 1) a lower period can be sustained. Therefore, we construct

$$T_{\mathsf{min}}(b,m) = \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{bm}, \qquad b = 2,3,\ldots, \tag{4.5}$$

by multiplying the denominator of (4.4) by m.

$T_{\mathsf{min}}(1,m)$, which is the minimum sustainable period without batching ($b = 1$) and with m worker cores cannot be determined by setting $b = 1$ in (4.5). $T_{\mathsf{min}}(b,m)$ in (4.5) accounts for all batching overheads which are clearly not present if jobs are not batched. Hence, we have

$$T_{\mathsf{min}}(1,m) = \frac{C_{\mathsf{Wc}} + C_{\mathsf{Wuser}}}{m} \tag{4.6}$$

that is based on the same reasoning as (4.5) except that C_{Wsetup} and C_{WonceJ} are set to 0 and b is set to 1 because jobs are not processed in batches.

To show that batching can reduce the minimum sustainable period we compute the factor by which batching improves the minimum sustainable period over an implementation without batching. More specifically, we compute the ratio between $T_{\mathsf{min}}(b,m)$ of (4.5) and $T_{\mathsf{min}}(1,m)$ of (4.6)

$$\begin{aligned} \frac{T_{\mathsf{min}}(b,m)}{T_{\mathsf{min}}(1,m)} &= \frac{\frac{C_{\mathsf{WonceB}}}{b} + C_{\mathsf{WonceJ}} + C_{\mathsf{Wuser}}}{C_{\mathsf{Wc}} + C_{\mathsf{Wuser}}} \\ \Longrightarrow \lim_{b\to\infty} \frac{T_{\mathsf{min}}(b,m)}{T_{\mathsf{min}}(1,m)} &= \frac{C_{\mathsf{WonceJ}} + C_{\mathsf{Wuser}}}{C_{\mathsf{Wc}} + C_{\mathsf{Wuser}}} \\ \Longrightarrow \lim_{b\to\infty} \frac{T_{\mathsf{min}}(b,m)}{T_{\mathsf{min}}(1,m)} &\begin{cases} < 1, \text{ if } C_{\mathsf{Wc}} > C_{\mathsf{WonceJ}} \\ \geq 1, \text{ if } C_{\mathsf{Wc}} \leq C_{\mathsf{WonceJ}}. \end{cases} \end{aligned} \tag{4.7}$$

Equation (4.7) shows two things. Firstly, batching improves the minimum sustainable period because $\frac{C_{\mathsf{WonceB}}}{b}$ approaches zero if b approaches infinity. Secondly, we can assert that batching reduces the minimum sustainable period $T_{\mathsf{min}}(b,m)$, if this factor

is lower than 1 i.e. if the time required for batching C_{WonceJ} is lower than the time required for communication C_{Wc}.

Finally, we address a different although related problem: given an application period T, what is the minimum number of worker cores m that can match the demanded workload? If the following inequality holds strictly

$$T \geq T_{\mathsf{min}}(b,m) \tag{4.8}$$

the difference between the task period T and minimum period that the system supports $T_{\mathsf{min}}(b,m)$ can be used to choose a lower core count than the maximum. From (4.5) and (4.8) it follows that

$$T \geq \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{mb}$$

and

$$m \geq \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{Tb}.$$

Since m must be an integer it must be

$$m \geq m_{\mathsf{min}}(b,T) = \left\lceil \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{Tb} \right\rceil = \left\lceil \frac{C_{\mathsf{WonceB}}}{Tb} + \frac{C_{\mathsf{WfullJ}}}{T} \right\rceil. \tag{4.9}$$

As can be seen in (4.9), the minimum number of required worker cores $m_{\mathsf{min}}(b,T)$ decreases with the batch size b. However, the batch size b cannot be chosen arbitrarily high to minimise the number of worker cores as it has a natural limit due to the task deadline D, as shown next.

4.4.2 Job Batch Size vs. Task Deadline

This section shows that the batch size cannot be chosen arbitrarily high. This is the case because, as the size of batches grows, the time needed for a job to pass through a job farm grows as well, which eventually leads to deadline violations.

Figure 4.4 illustrates how batching affects the response time. As expected, the response time with batching (of Figure 4.4b) is higher than the response time without batching (of Figure 4.4a). Other factors that can increase response times are the same in both figures. As illustrated in Figure 4.4b the response time increases with the batch size for multiple reasons:

- jobs are not immediately dispatched to workers but are halted until the batches to which they belong are full,

- jobs have to wait until the other jobs in the same batch are processed by a worker,
- jobs have to wait until other jobs are unbatched.

We start by computing the response time $R_k(b)$ of the k-th job, assuming a batch size b, which is

$$\underset{k=1,\ldots,b}{R_k(b)} = \underbrace{(b-k)T}_{\substack{\text{Batch}\\ \text{aggregation}}} + \underbrace{bC_{\mathsf{WfullJ}}}_{\substack{\text{Batch}\\ \text{processing}}} + \underbrace{(k-1)C_{\mathsf{C}} + C_{\mathsf{C}}}_{\text{Unbatching}} + C_{\mathsf{O}}. \tag{4.10}$$

Job response times are composed of several components that are discussed in detail below:

- $(b-k)T$ is the time spent to batch b jobs. The first job in a batch (with $k=1$) experiences the longest delay $(b-1)T$, while the last one (with $k=b$) completes a batch and experiences no aggregation delay.
- bC_{WfullJ} is the batch processing time. As discussed in depth earlier, the completion of jobs is not communicated to the aggregator until all b jobs in the same batch are processed. Therefore, all jobs in a batch experience a delay of bC_{WfullJ} which is the time required to process an entire batch.
- The time to unbatch the result of the k-th job is due to (i) the waiting time for the earlier jobs in the same batch $C_{\mathsf{C}}(k-1)$ to be unbatched, plus (ii) the time to unbatch the k-th job itself which is C_{C}. Hence, altogether the time needed to unbatch the result of the k-th job is kC_{C}, which is the sum of two terms: time spent by the k-th job waiting for earlier jobs to be unbatched, and the time needed to unbatch the k-th job.
- The term C_{O} represents time spent in farm internal communication, dispatcher, and aggregator (see Section 4.3.3).

For clarity, some of the terms used in (4.10) are not explicitly shown in Figure 4.4b. The *batch aggregation time* in (4.10) is the time between the arrival of a job and the subsequent communication between the producer and worker in Figure 4.4b. The *batch processing* time corresponds to the light green "Job processing" boxes, and the time required for *unbatching* is directly shown in the figure. The term C_{O} corresponds to the yellow and green communication blocks.

Next, we determine the impact of the deadline constraint of (4.1) on the batch size b. Since the *task response time* R is

$$R = \max_{k}\{R_k\} = \max_{k=1,\ldots,b}\{R_k\} \tag{4.11}$$

then the deadline constraint of (4.1) trivially becomes $R \leq D$.

From (4.10), the job response time R_k can be written as:

$$\begin{aligned} \underset{k=1,\ldots,b}{R_k(b)} &= (b-k)T + bC_{\mathsf{WfullJ}} + kC_{\mathsf{C}} + C_{\mathsf{O}} \\ &= bT + bC_{\mathsf{WfullJ}} + C_{\mathsf{O}} - k(T - C_{\mathsf{C}}). \end{aligned} \tag{4.12}$$

From the constraint of (4.2), it follows that $T - C_{\mathsf{C}} \geq 0$. Unsurprisingly, (4.12) is maximal for $k = 1$ since the first job in a batch has the longest response time. By setting $k = 1$ in (4.12), we find the response time R of the task that is

$$R = \max_{k=1,\ldots,b}\{R_k\} = (b-1)T + bC_{\mathsf{WfullJ}} + C_{\mathsf{O}} + C_{\mathsf{C}}. \tag{4.13}$$

Finally, from the deadline constraint of

$$R \leq D$$

we can find the constraint on the batch size b, that is

$$\begin{aligned} R &= (b-1)T + bC_{\mathsf{WfullJ}} + C_{\mathsf{O}} + C_{\mathsf{C}} \leq D \\ &= b(T + C_{\mathsf{WfullJ}}) - T + C_{\mathsf{O}} + C_{\mathsf{C}} \leq D \end{aligned}$$

which allows us to find the maximum batch size $b_{\mathsf{max}}(T,D)$

$$b_{\mathsf{max}}(T,D) = \left\lfloor \frac{D + T - C_{\mathsf{O}} - C_{\mathsf{C}}}{T + C_{\mathsf{WfullJ}}} \right\rfloor. \tag{4.14}$$

We observe that the upper bound for $b_{\mathsf{max}}(T,D)$ is

$$b_{\mathsf{max}}(T,D) \leq \left\lfloor \frac{D}{T} \right\rfloor + 1$$

which states the natural fact that the number of jobs in a batch cannot exceed the maximum number of pending jobs.

Based on Equation (4.14) we determine when a task benefits from batching. Naturally, batching is beneficial if

$$b_{\mathsf{max}}(T,D) \geq 2.$$

This means that batching is only applicable if it is possible to aggregate two or more tasks. This equation is equivalent to

$$\frac{D+T-C_{\mathsf{O}}-C_{\mathsf{C}}}{T+C_{\mathsf{WonceJ}}+C_{\mathsf{Wuser}}} \geq 2.$$
$$C_{\mathsf{Wuser}} \leq \frac{D-T-C_{\mathsf{O}}-C_{\mathsf{C}}-2C_{\mathsf{WonceJ}}}{2}. \tag{4.15}$$

A necessary condition for batching is $D > T$ since jobs cannot be aggregated otherwise.

For example, by replacing the constants in Equation (4.15) with values for our evaluation platform (see Table 4.1) we get

$$C_{\mathsf{Wuser}} \leq \frac{D-T-980\,\mathsf{ns}}{2}.$$

In this case a task with period $T = 1\,\mu\mathsf{s}$ and deadline $D = 5\,\mu\mathsf{s}$ benefits from batching as long as $C_{\mathsf{Wuser}} \leq 1.51\mu\mathsf{s}$.

Assuming that the maximum batch size b_{max} is used (there is no reason to use a smaller batch size), we can find the minimum core count $m_{\mathsf{min}}(b_{\mathsf{max}}(T,D),T)$. In fact, as apparent from (4.9), it is always best to use the largest possible batch size. We find the minimum core count by setting $b = b_{\mathsf{max}}(T,D)$ in (4.9):

$$m_{\mathsf{min}}(b_{\mathsf{max}}(T,D),T) = \left\lceil \frac{C_{\mathsf{WonceB}}}{T\left\lfloor \frac{D+T-C_{\mathsf{O}}-C_{\mathsf{C}}}{T+C_{\mathsf{WfullJ}}} \right\rfloor} + \frac{C_{\mathsf{WfullJ}}}{T} \right\rceil. \tag{4.16}$$

The values of the terms of the right-hand sides of (4.14) and (4.16) are either task properties or can be determined with WCET analysis. These equations can thus be used to compute the minimum number of worker cores and the maximum batch size at compile time as demonstrated by our library.

4.5 The Peso Library

We present a macro-based farm library for real-time systems that is statically scheduled to guarantee predictable WCETs. Peso uses the equations presented in Section 4.4. In this section, we illustrate the farm API of Peso and discuss its implementation.

4.5.1 API Concepts

Figure 4.5 shows the implementation of an example application with Peso. The macros `PRODUCER`, `CONSUMER`, `WORKER_FUNCTION`, `PREPARE_FARM`, and `CONFIGURE_FARM` are

```
1  typedef struct per_job_data {
2    int per_job_input_vector[PER_JOB_INPUT_SIZE];
3    int result;
4  } per_job_data_t;
5  PREPARE_FARM(per_job_data_t, PERIOD_NS)
6
7  PRODUCER(producer_func, //Producer name
8    while (1) {
9        /*Wait for input data to be ready*/
10       submit_data();})
11
12 WORKER_FUNCTION(worker_func, //Worker function name
13   data_t,
14   unsigned int result = 0;
15   for (int i = 0; i < PER_JOB_INPUT_SIZE; ++i) {
16       result += access_data()-> per_job_input_vector[i];
17   }
18   access_data()->result = result;)
19
20 CONSUMER(consumer_func, //Consumer name
21   data_t,
22   printf("Result:%d", receive_data()->result);)
23
24 CONFIGURE_FARM(farm, //Farm name
25   producer_func, worker_func, consumer_func)
26
27 void main() {
28   start_farm();
29 }
```

Figure 4.5: Illustration of the API of Peso. Each job computes the sum of `PER_JOB_INPUT_SIZE` integers. Multiple summations are executed in parallel but each one is performed sequentially.

provided by the library. The first parameters are unique names that are required to instantiate the farm and the second is the implementation.

Peso provides functions for data accesses that hide the internal storage scheme. The producer uses `submit_data()` in line ten to send input data to the job farm. Either the producer prepares the input data in `per_job_input_vector` or an external process that then signals the producer when the data is ready. The worker function accesses input

data and stores results via `access_data()` in lines 16 and 18. Finally, the consumer receives results via `receive_data()`. Lines one to four specify the in- and output data of each job. Note that an instance of `per_job_data` is accessed by only a single worker but multiple instances are processed in parallel.

Finally, line five in Figure 4.5, and the code in lines 24 to 29 instantiate the task farm. `CONFIGURE_FARM` generates the `start_farm` function, that is called in line 28, based on the farm name that is passed to via its first parameter.

4.5.2 Implementation and Internal Communication Overheads

Figure 4.3 provides an overview over the architecture of Peso. The farm is composed of two logical entities that are hidden from programmers by the farm API: the dispatcher, and the aggregator. The dispatcher deals out batches to workers and the aggregator collects the corresponding results. Both serve workers in a round robin fashion to achieve predictability, for instance, the aggregator does not collect the n-th result of worker $i+1$ before it has collected the n-th result of worker i. Peso stores the in- and output data of the workers consecutively in a buffer. The dispatcher sends a pointer into this buffer to a worker when it deals out a job or a batch. Only the pointer to the inputs of the first job of a batch are sent. Data locations for other jobs in the same batch are computed based on this pointer. Time spent in sending these pointers makes up the internal communication overheads that we reduce through batching in this implementation.

4.6 Experimental Setup

4.6.1 Evaluation Platform and Methodology

We use the XMOS xCore-200 microcontroller which is designed for hard real-time systems and has two clusters of eight logical cores (see below) [XMO18]. The microcontroller is programmed with the xC programming language, which is akin to C but offers, among other things, programming language constructs for threads and so called *channels* for communication and synchronisation purposes [XMO15]. We use the XCC compiler version 14.4.4 with the default `-O2` optimisation flag. No OS is on the device and no thread scheduler is required in the context of our experiments as all threads execute on dedicated cores. We perform brute force parameter space searches to determine optimal parameter choices for comparison with those computed by our

analytical framework (see Section 4.7.1.1 and 4.7.1.2). These are based on experiments and measurements with timers provided by the target platform.

We take five samples per data point. This is true for the brute force parameter searches as well. Each sample yields the same result except in a small number of cases in Section 4.7.3 where small variations in intercore communication cost (of up to 1ns) occur. Note that our WCET costs allow for this. Because of this invariability we do not show error margins.

4.6.2 Predictability and The Memory System

To achieve predictability the xCore-200 microcontroller uses interleaved multithreading [XMO18, CSG99]. In more detail, instructions of eight logical cores are issued to a shared five stage pipeline in a round robin fashion. A core is serviced every five cycles if five or fewer cores are used or every six, seven, or eight cycles if more are active. Consequently, the device must be programmed with multiple threads to achieve best performance. The memory system is designed so that all memory access instructions complete in five cycles. Therefore, at a device clock frequency of 500Mhz the WCET of memory accesses is: 10ns, 12ns, 14ns or 16ns depending on how many cores are active. The best-case execution time is always 10ns.

The device does not have data caches and translation lookaside buffers which can cause interference in conventional hardware [May, XMO18]. Each core has private registers, an instruction buffer, and access to shared SRAM. Memory accesses have exclusive access to the shared SRAM as all cores share the memory pipeline stage. The WCET for memory requests is not affected by reordering or bank conflicts.

Cores can directly communicate with each other over so called channels. These are set up and torn down with dedicated instructions in the ISA [May].

4.6.3 Worst-Case Execution Times

XMOS provides a static code analysis tool for WCETs [XMO13] that is similar to the OTAWA Eclipse plugin [BCRS10]. The tool provides WCETs on the level of code blocks, and single instructions. The WCET of a code section can be computed by summing up the WCETs of the relevant instructions.

We determine the WCETs on our evaluation hardware through static code analysis and measurements. The WCET of intercore communication C_{com} is determined through measurements under stress. To put maximum stress on the core-interconnect

Table 4.1: WCETs in cycles of the job farm components (see Section 4.3.3) for the benchmarks (see Section 4.6.4) and the used input sizes. Each cycle is 2ns. Presented WCETs are determined with fewer than six cores (see Section 4.6.3). Only C_{Wuser} changes with the benchmark and the input size.

App.	**DMV**		**RED**		**SMV**	
Input size	**5**	**10**	**15**	**30**	**10**	**15**
C_{D}	75	75	75	75	75	75
C_{com}	65	65	65	65	65	65
C_{Wc}	125	125	125	125	125	125
C_{Wsetup}	5	5	5	5	5	5
C_{WonceJ}	40	40	40	40	40	40
C_{Wuser}	*630*	*4030*	*415*	*790*	*1455*	*3435*
C_{A}	115	115	115	115	115	115
C_{C}	90	90	90	90	90	90

we execute dummy threads on all cores that use up all available interconnect communication channels and do nothing but constantly send data back and forth. The maximum of 10000 measurements is then used for C_{com}. Note, the WCET and best-case execution time of intercore communication differs. All other WCETs are determined through static analysis.

The compiler generated instructions of the worker function and the farm internal worker code can slightly vary with the batch size. With rare instruction sequences the hardware cannot refill the instruction cache transparently and has to stall the pipeline for a single cycle to fetch instructions [May]. This can cause the statically determined execution times to vary with the batch size. Based on the maximum number of such pipeline stalls that we could observe for a set of instructions we manually increase the generated WCETs that we used for our evaluation to allow for these stalls. The so adjusted WCETs are presented in Table 4.1. These stalls cause the slight increases in Figure 4.2 when six instead of five workers are used.

Per instruction WCETs increase with each additional core if more than five cores are used (see Section 4.6.2). Peso considers this when it chooses the worker count and handles this further complexity for application developers.

4.6.4 Benchmarks

The benchmarks are: dense matrix vector multiplication (DMV), sparse matrix vector multiplication (SMV), and reduction (RED). They are widely used across various domains such as computer vision and machine learning [Cor17]. SMV uses the compressed sparse row format. RED computes the sum of a set number of integers. Each worker executes the computational kernel of an application sequentially. However, multiple kernel instances are executed in parallel. We use two input vector sizes with each application. Table 4.1 shows the relevant WCETs.

4.7 Evaluation

This section experimentally validates the analytical framework presented in Section 4.4, and shows that the overheads of Peso over hand-crafted code are small. As mentioned in Section 4.6.2, to achieve predictability our evaluation platform uses an atypical core design as opposed to the cores in conventional systems (see Section 4.6.2). For clarity, we refer to the logical cores of our evaluation system as cores.

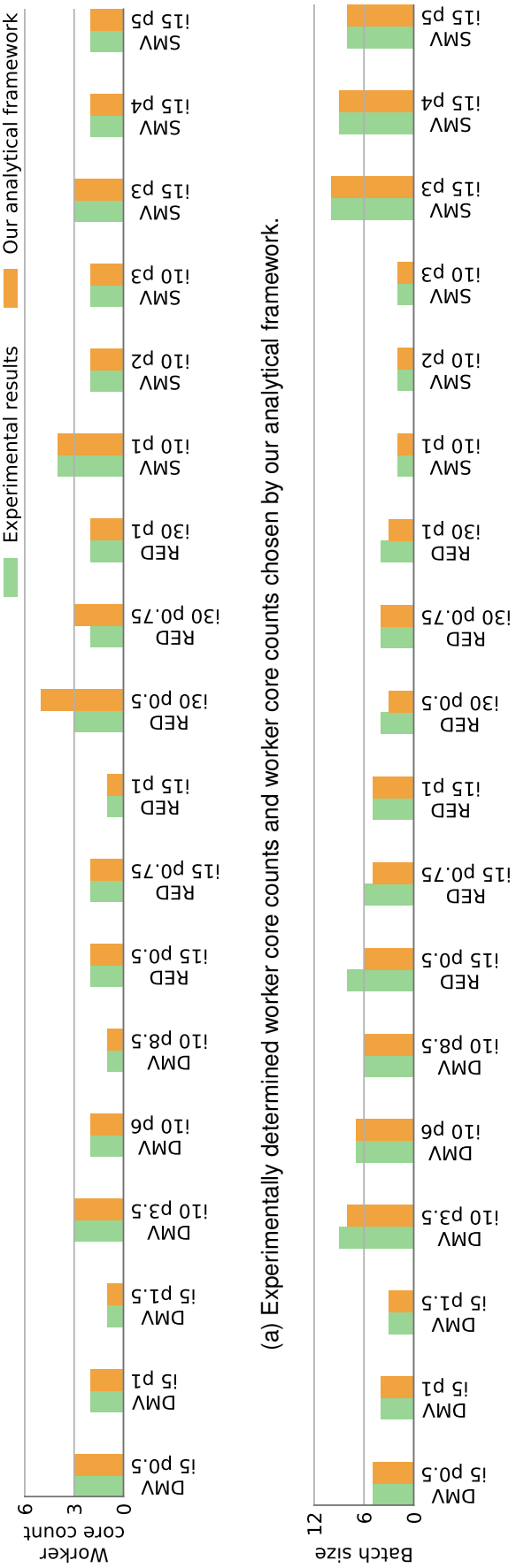
4.7.1 Experimental Validation of our Analytical Framework

4.7.1.1 Worker Core Counts

We validate the worker count choices of our framework (that is $m_{\mathsf{min}}(b_{\mathsf{max}}, T)$) as computed from (4.16) against, the best possible worker counts that we determine with an exhaustive search (see Section 4.6.1). For this we decrease the number of workers until deadline misses occur.

Figure 4.6a shows that our framework chooses the best worker count in all cases except two and crucially, never makes choices that cause deadline misses. Possible explanations for these two cases are the pessimism added to WCETs by potential pipeline stalls, and intercore communication that can be faster than its WCET as discussed in Section 4.6. The pessimism added by stalls is highest in these two cases.

Required worker core counts decrease with higher periods, and increase with larger input sizes. Increasing periods mean less pressure on the farm and so allow for fewer worker cores. Larger input sizes cause higher batch processing times (see Section 4.4) and so require more workers to match the task period.



(a) Experimentally determined worker core counts and worker core counts chosen by our analytical framework.

(b) Experimentally determined batch sizes and batch sizes computed by our analytical framework.

Figure 4.6: Job farm parameter comparisons. Benchmarks: Reduction (*RED*), dense and sparse matrix-vector multiplication (*DMV* and *SMV*). Periods in μs are prefixed with a p and input sizes are prefixed with an i.

4.7.1.2 Batch Sizes

We validate the batch sizes computed by our framework by comparing them with the best ones, which we determine through a brute force search (see Section 4.6.1). For this we hard code the worker count to the maximum and increase the batch sizes until we measure deadline misses.

Figure 4.6b shows that our framework chooses the best or close to best batch sizes and never ones that cause deadline misses. Again, differences between the two batch sizes can be explained with the pessimism inherent to the WCETs of the user code and intercore communication (see Section 4.7.1.1).

The batch sizes decrease with increasing periods (see DMV) and with larger job input sizes (see RED). Larger periods and input sizes cause longer job aggregation and batch processing times (see Section 4.4) respectively and so only allow for smaller batch sizes with the same relative deadlines.

Most deviations occur with the reduction benchmark. This computational pattern might be less vulnerable to the instruction fetch issue explained in Section 4.6.3, which adds pessimism to the WCETs. If this issue does not occur the actual execution time of the benchmark and farm internal code is lower than its WCET, which in turn gives our oracle the opportunity to choose larger batch sizes and lower worker core counts than our models, without causing deadline misses.

4.7.2 Fewer Cores with Batching and The Effect of Input Sizes

To quantify the impact of batching, Figure 4.7 shows by how much batching reduces the minimum sustainable periods $T_{\mathsf{min}}(1,m)$ for a given number m of cores over implementations without batching (see Equation (4.6)). A lower minimum period means that tasks that previously needed additional cores because their periods are too low for a given core count can now be scheduled. We use a variant of Peso without batching for the baseline measurements. To experimentally determine the minimum periods of both versions, we decrease the periods until the farm is overloaded and jobs miss their deadlines. The number of worker cores is hard coded to the maximum for this.

Batching lowers the minimum periods $T_{\mathsf{min}}(b,m)$ (and so enables higher throughput) by up to 45.36%, 16.6% on average, and never degrades the minimum period. However, the baselines of RED with input size 15 and DMV with input size 10 are affected by the pipeline stall issue discussed in Section 4.6.3. Therefore, in the interest of a fair comparison, omitting these gives a maximum and average lowering of the

Figure 4.7: Reduction in the minimum sustainable periods with batching over Peso without batching (higher is better). Benchmarks: Reduction (*RED*), dense and sparse matrix-vector multiplication (*DMV and SMV*). Numbers after the benchmark name indicate the number of input elements per job and batch sizes prefixed by *i* and *b*.

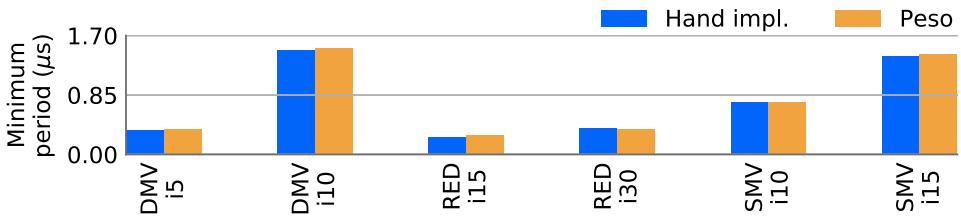


Figure 4.8: Minimum sustainable period with and without Peso (lower is better). Benchmarks: Reduction (*RED*), dense and sparse matrix-vector multiplication (*DMV and SMV*). Numbers after the benchmark names indicate the number of input elements per job prefixed by *i*.

minimum period of 22.38% and 12.54% respectively.

The improvements of the minimum period decrease with increasing input sizes. For example, RED benefits more with an input size of 15 elements than with 30 elements. This is expected since larger input sizes mean higher application code WCETs. The execution time share spent in communication decreases with higher application code WCETs and so maximum achievable improvements through batching decrease.

4.7.3 Abstraction Layer Overheads

This section evaluates the overheads of Peso in terms of the minimum sustainable period. Figure 4.8 compares the minimum periods of Peso-based implementations

with the ones of carefully hand-crafted code that does not use the abstractions of Peso and thus parallelism and batching have to be implemented in the application code. To measure minimum periods, we decreased the periods until the implementations are overloaded and miss deadlines. The worker count is set to the maximum for this.

The minimum, average, and maximum difference between the minimum period of the hand and Peso implementations are 8ns, 18.6ns, and 24ns. The overheads introduced by the abstraction layer of Peso over hand implementations translates to an on average 3.37% higher minimum period. The maximum overhead is 8.66% and the minimum 1.06%. The minimum period of the hand implementation of the reduction application with an input size of 30 is slightly higher than the period of the Peso-based implementation due to the stall issue discussed in Section 4.6.3.

The overheads come from hiding job batching from programmers through the abstraction layer of Peso. To make batching application developer transparent the consumer calls `access_data()` for each job even though all results of a batch of jobs are completed at the same time. Removing this abstraction and its associated costs lowers the minimum period but exposes application developers to much more complexity.

4.8 Conclusion

We argue for skeletons to program parallel hard real-time systems as they ease programming by abstracting implementation details. Structural information encoded in skeletons also allows for tight analysis and efficient scheduling.

We conduct a case study with the farm skeleton. We present an analytical framework that combines knowledge about this skeleton with predictable hardware to automatically choose the minimum core count. Based on this, we also develop an efficient execution strategy that reduces parallelism related overheads.

We demonstrate experimentally that in most cases our framework chooses the best or close to best parameters, and never makes choices that cause deadline misses. Our skeleton informed execution strategy improves minimum sustainable periods by up to 22.38% and so reduces required core counts. Lastly, compared to carefully hand-crafted code, the overheads of our farm are negligible.

The presented programming model is very restrictive and so can implement only simple applications. For example, all workers must execute the same function, we only support periodic real-time systems, and we offer only a single skeleton. As a result, current systems can already schedule the simple applications that we support. Our aim

with this work is to make a first step towards a set of real-time skeletons that together can implement complex applications. While current compiler analysis could determine dependencies in the simple benchmarks that we presented in this initial work and then schedule them, this is likely not possible in more complex situations. In these situations skeletons are beneficial in two ways. Firstly, they provide structural information, in our case this is the knowledge that farm inputs can be processed independently, and, secondly, their high-level nature provides considerable implementation flexibility. An example of this in non-real-time contexts is Lift, where the high-level nature of skeletons and information encoded in them is used to dramatically change the structure of computational kernels [SRD17]. Until now Lift and its extensions require a skeleton programming interface and it has not yet been shown that solely compiler analysis of sequential code or other programming models can be used instead.

Chapter 5

Conclusion

This thesis presents two novel case studies in how skeletons improve parallel programming through abstraction. Chapter 3 enables migration of computational kernels between heterogeneous devices and hides the complexity of this behind a skeleton programming interface. Chapter 4 argues for the use of skeletons to program hard real-time systems, and presents a timing predictable skeleton, which hides complex scheduling decisions from programmers.

The remainder of this chapter is structured as follows. Section 5.1 provides an overview over the contributions of this thesis. Section 5.2 reflects on how we address weaknesses of previous work. Section 5.3 discusses limitations of the work presented in this thesis and opportunities for future work. Finally, Section 5.4 concludes this thesis with final remarks.

5.1 Contributions

This thesis makes the following contributions:

- We present a skeleton framework, which enables runtime systems to migrate computations between CPUs and GPUs in a programmer transparent way. Adding this flexibility to computational kernels by hand is in principle possible but would introduce unacceptable complexity into the application code. We hide this complexity with the parallel_for skeleton and novel attributes, which describe the memory access patterns of kernels. To enable migration, kernels are internally subdivided into smaller sub-kernels, which each cover a subsection of the original iteration space, so called *slices* and *slice kernels*. Migration is possible between the invocation of slice kernels because the high-level state of

the application-level kernel is known and stable. The semantics of the parallel_for skeleton guarantees to runtime systems that this subdivision of the iteration space is a legal transformation. To further hide complexity from application programmers we use predictive models to choose the slice size of the subkernels. We show with a simple but realistic evaluation scenario that mid-kernel migration improves performance by up to 1.30x and 1.08x on average. Furthermore, compared to a hand-implementation of the migration mechanism, our skeleton reduces the code size by at least 88%.

- We demonstrate that skeletons can simplify the programming of hard real-time systems in a case study with the task farm skeleton. The skeleton implementation uses knowledge about the independence of tasks and execution time analysis, which is available for hard real-time systems, to automate difficult scheduling decisions. We automatically choose the minimum degree of parallelism that is required to meet the computational demands of an application and a parameter of an optimisation technique. Parameter choices and the implementation of the optimisation are made transparent to programmers with the high-level task farm abstraction. We show that our skeleton implementation chooses the optimal parameters with an accuracy of 83% and never parameters that would cause deadline misses. We also show that the skeleton abstraction layer introduces only small overheads by showing that it increases the minimum sustainable period by only 3.37% on average.

5.2 Reflections on Related work

This section discusses how we address weaknesses in previous work that is closest to the two main chapters of this thesis (see Section 2.6). In more detail, we discuss work on migration in heterogeneous systems by Lösch et al., and preliminary work on real-time skeletons by Ungerer et al. and Steigmeier et al..

As discussed in Section 2.4.2.1, Lösch et al. present a runtime system for fine-grained migration in heterogeneous systems and a novel scheduling strategy [LP20]. In comparison to our programming model, Lösch et al. require programmers to manually provide implementations for all devices and adapt kernels manually to support migration. In addition, Lösch et al. also do not include slicing-aware data transfers and chunked data transfers. From experience with our system we know that, if these

features are missing, unnecessary data transfers can cause significant slowdowns over alternative kernel-by-kernel scheduled systems. This is especially true for data intensive kernels that spend most of their time in data transfers when they are scheduled on dedicated GPUs. The authors also require programmers to choose parameters of their mechanism manually. In comparison, we automatically choose slice sizes with predictive models and chunk sizes with a one-off brute force search "at the factory".

In our evaluation we use all nine applicable benchmark kernels from two standard benchmark suites. In contrast, the authors use five kernels that are not from standard suites. Over half of them are compute intensive which could leave slowdowns that the simple data transfer scheme might cause unrevealed. In addition, the large multiprogrammed experiments that the authors use do not expose the behaviour of the proposed mechanism in edge cases (see Section 2.6.1). However, we investigate the benefits and overheads of slicing in detail and, therefore, present fine-grained results for single kernels and a range of controlled migration time points.

Lastly, our implementation also benefits from smart management of OpenCL and CUDA, and from more sophisticated migration strategies. We avoid OpenCL and CUDA setup costs during migrations with our daemon. Our implementation can restart slices, overlap the current and next slice on both devices, and overlap cleanup of the old device and execution on the new device.

Stegmeier and Ungerer et al. present work on skeletons for parallel real-time systems [UBF+16, SFJU]. We consider this work preliminary because it uses estimated WCETs, which makes it unsuitable for *hard* real-time systems. We make several contributions that go beyond this work.

First of all, our skeleton is fully timing analysable, and, therefore, suitable for hard real-time systems. As mentioned above, Stegmeier and Ungerer work only with approximated WCETs of their skeletons [JFGU14]. The authors add several thousand cycles to their WCETs for each skeleton call and each additional thread (see Section 2.6.2). We are the first to show that detailed models of the WCETs of skeletons can be created without approximations.

Our skeleton is fully self-tuning. The authors provide a tool for core count recommendations [UBF+16]. However, the recommended core counts can be underestimates that could lead to deadline misses, because they are based on estimated WCETs (see Section 2.6.2). In contrast, we model the required core count with a detailed model and do not use approximated WCETs. Therefore, our autotuning mechanism for core

counts is fully automatic and always chooses safe core counts. In addition, the autotuner of Ungerer et al. uses a genetic algorithm [FJO+16]. Unfortunately, the authors do not discuss how long it takes for this algorithm to make decisions and how close they are to the optimum. Because we choose core counts based on an analytical model our autotuner should be considerably faster than the authors' genetic algorithm.

In the context of hard real-time systems, we are the first to demonstrate that structural information encoded in skeletons can be used for performance improvements. We demonstrate this with job batching, which reduces communication overheads with information encoded in our skeleton.

We also use a different kind of parallelism. Our skeleton exploits inter-job parallelism because it executes sequential code of multiple real-time task instances in parallel. In contrast, the authors exploit intra-job parallelism [SFJU, UBF+16].

Finally, the authors use a simulator for their evaluation, and we use off-the-shelf hardware [UBF+16].

5.3 Limitations and Future Work

Sections 5.3.1 and 5.3.2 discuss future work and limitations of chapters 3 and 4 respectively. The ideas discussed in each section below are roughly ordered from high-level research ideas to more concrete iterative improvements that could be added in the context of a real deployment.

5.3.1 Transparent Kernel Migration

Chapter 3 has shown how the semantics of the parallel_for skeleton enables our mid-kernel migration mechanism and that the complexity of the machanism can be hidden behind this skeleton. This section discusses limitations and new research directions.

Programming Model The benefits of even higher level programming models for our migration mechanism could be investigated. A higher level of abstraction could give compilers more freedom to automatically generate highly optimised implementations for all devices in the system. In addition, with another programming model, programmers might not need to manually add optimisations, which improve performance for some devices, as intended, but might actually degrade performance on other devices. In more detail, with our current interface, programmers can manually add optimisa-

tions for one of the available devices, for instance, for the fast device. For example, programmers can use Structure-of-Arrays (SoA) data layouts to optimise for execution on GPUs or Arrays-of-Structures for CPUs [SLH12]. However, SoA layouts likely degrade performance on CPUs. As explained above, better compilers and a programming model on a higher abstraction level might solve this problem.

Data parallel skeletons such as the ones used by RISE and Lift encode implicitly memory access patterns [HLK+20, SRD17]. Using a similar set of skeletons would simplify programming even more because programmers would not need to explicitly provide information about access patterns, as is the case with our current programming interface. RISE aims to be an intermediate language for DSLs. This in turn means that DSLs that are implemented with RISE-like skeletons would automatically benefit from mid-kernel migration.

Alternatively, a more iterative extension would be to keep our current interface and add the ability to automatically undo device specific optimisation for one device if a kernel is executed on another. Some work in this direction exists. For example, Grewe et al. automatically optimise kernels, that are extracted from OpenMP loop nests, for execution on GPUs with loop interchange and memory load reordering among others [GWO13]. Sung et al. automatically transform data layouts from CPU to GPU friendly layouts [SLH12].

Migration from or to Multiple Devices This chapter only considers migration between CPU and GPU in either direction. However, if the fast device becomes available and the slow device remains available one might want to continue execution on both devices at the same time, as has been done in previous work [GO11]. Vice-versa, in scenarios in which new applications arrive unpredictably or scenarios in which hardware availability changes unpredictably one would want, in some situations, to migrate to just a single device. This could be added to a real deployment to further improve performance.

Further Scheduling Policies Future work could investigate further policies that make use of the new migration flexibility. For example, systems with priority scheduling will benefit. Lower priority kernels could be migrated instead of just being preempted when kernels with higher priorities require their current device. Also, schedulers that co-schedule computations in interference-aware ways could benefit from mid-kernel migration [CYG+17, DK13]. In this context, the best task-to-resource assignments

might change as applications come and go. For example, if a new kernel has a strong preference for a device, it might be beneficial for overall system performance to migrate other kernels, which would cause interference, from that device to other devices.

Hardware Extensions More control over data transfers and the hardware scheduler on the GPU could help to further reduce the performance costs of the migration mechanism. One source of overheads are chunked data transfers. What is on a conceptual level a single data transfer is divided into multiple API calls to allow a data transfer to be aborted between them. Alternatively, GPGPU low-level APIs could allow users to abort data transfers while they are in progress. In addition, thread block schedulers could be modified to support migration without slicing. A potentially simple modification could allow the host processor to signal the thread block scheduler to not schedule any further thread blocks. The scheduler could then return a bit mask of the already scheduled thread groups, which have finished or will finish on the GPU. The host could then schedule the remaining thread blocks on the CPU or another device. However, with this scheme efficient data transfers would be a challenge. Because the runtime system does not know which thread blocks will be executed on the GPU it must transfer all input data to the GPU before kernel launch. If a kernel migrates then later on, some input data will have been transferred unnecessarily. With slicing, in contrast, only the input required for the next slice is transferred to the GPU.

Better Translation to OpenMP A more sophisticated translation mechanism is needed for translation from our parallel_for to OpenMP than the current one, which translates only to CUDA and OpenCL except for reductions. An obstacle is that barriers within batches do not have a one-to-one translation to OpenMP constructs. Figure 5.1 shows with a simple example how barriers within a batch could be translated to OpenMP with careful management of loops, that internally implement batch iterations. This is a sketch for a potential translation strategy and likely does not cover all corner cases. This idea has been developed in discussions with Volker Seeker as mentioned in the acknowledgements.

Evaluation The evaluation could be extended in several directions:

- The goal of the work presented in this chapter is to provide programmer transparent migration flexibility for heterogeneous systems. Therefore, we chose a

```
1 parallel_for pf(start, end, [&]DEVICE_HOPPER_LAMBDA() {
2   # This will be held in scratchpad memory on GPUs because of the attribute.
3   __attribute__((device_hopper_batch_shared)) int local_mem[BATCH_SIZE];
4
5   # Copy the weights to the scratchpad memory.
6   int local_id  = get_batch_iteration();
7   local_mem[local_id] = weights[local_id];
8   device_hopper::batch_barrier();
9
10  ... # Code that uses local_mem
11 });
```

(a) Parallel_for based code with GPU optimisations. The implementation uses scratchpad memory because `local_mem` is annotated with `__attribute__((device_hopper_batch_shared))` (see line three).

```
1 #pragma omp parallel for
2 for (int batch = start / BATCH_SIZE; batch < end / BATCH_SIZE; ++batch) {
3   int local_mem[BATCH_SIZE];
4
5   # Copy the weights to 'local_mem'.
6   # This loop implements the semantics of 'device_hopper::batch_barrier()'
7   for (int local_id = 0; local_id < BATCH_SIZE; ++local_id) {
8     int iteration = batch * BATCH_SIZE + local_id;
9     local_mem[iteration] = weights[iteration];
10  }
11
12  for (int local_id = 0; local_id < BATCH_SIZE; ++local_id) {
13    ... # Code that uses local_mem
14  }
15 }
```

(b) OpenMP translation of the code in Figure 5.1a.

Figure 5.1: Sketch for a potential translation strategy from GPU optimised parallel_for code to OpenMP. (b) is an OpenMP translation of the code in (a). The high-level idea is that barriers split loops over the batch elements in generated CPU implementations. This way, all iterations preceding a barrier are executed before any subsequent code. In this example, the first loop in (b) at line seven performs all writes to the `local_mem` array before any later code is executed, as intended by the barrier in (a).

baseline with the migration flexibility that current systems offer in a programmer transparent way, which is kernel-by-kernel migration. Future work could compare our system with others that provide fine-grained migration flexibility for heterogeneous system but in a non-transparent way. For example, kernel slicing implemented on top of a task-graph based programming model. This would provide insights into how the slicing overheads of our implementation compare to other ways to schedule slices.

- We only use one machine for the evaluation. Repeating the experiments on further machines would provide evidence that the migration mechanism and machine learning models will behave similarly on other systems.

- We co-execute benchmarks only with a dummy application to simulate unavailable devices. More realistic co-executing applications may cause interference. For example, data transfers between main memory and a dedicated GPU will likely interfere with the execution of another kernel on the CPU. Previous work has shown that streaming access patterns, like the ones of DMA transfers, can have significant impact on co-executing applications [MM07]. Our results do not take this interference into account because the dummy application does require data transfers and it does not perform any work while it blocks a device.

Slicing Overheads Our results indicate that the overheads of the mechanism are higher the smaller the problem size of a kernel. In the worst-case, the benchmarks will experience overall slowdowns for small enough inputs. A heuristic needs to be devised that, for a kernel instance and the input size of it, decides if slicing should be used.

Irrespective of the input size, slicing introduces overheads in all cases, and these are not amortised if kernels do not migrate. These overheads are likely caused by additional instructions that are executed between slices and the associated cache line evictions and cache misses. To further reduce the overheads, the exact sources need to be determined with more fine-grained measurements including performance counters. A potential solution to the additional cache misses is to pin cache lines containing instructions and data for the bookkeeping required for slicing into one of the cache levels. A similar feature exists on some ARM CPUs [ARM21]. Another potential solution is to prefetch these cache lines when a slice is about to finish.

The Daemon The daemon might not be practical in a real deployment because of security and stability issues. Kernels have access to the memory of each other, and an exception in one kernel can crash other kernels executed by the daemon at the same time. Future work could investigate how to provide isolation between daemon clients (i.e. applications that use the daemon simultaneously) so that they cannot access the memory of each other. On the CPU, this can potentially be achieved with an additional field in the page table entries that indicate by which daemon client a page is owned. This is similar to address-space identifiers or also called *ASIDs*, which are optional TLB entry fields that associate entries with processes [SGG19]. The daemon would run code belonging to different clients in different threads and these fields would contain the thread IDs. A special purpose register would contain the ID of the current thread and would be populated by the OS on a context switch. The TLBs could then check the thread ID on a TLB hit and the OS could do the same on a TLB miss to detect access violations. However, this adds complexity in several places. In addition to the hardware extension described above, OS support is required, and the daemon needs to handle access violations.

Data Transfer Chunk Sizes It may be possible to reduce data transfer chunk sizes without increasing the overheads of chunking. Data transfers via memory mapped I/O are more efficient than DMA transfers for small data sizes of a few MBs or less [FAN+13]. However, the current implementation uses the high-level CUDA function `cudaMemcpy`, which is free to always use DMA transfers.

Kernel-level Scheduler Our prototype implementation is purely in user space. Future work could extend existing schedulers in kernel space to use the new scheduling flexibility or add new scheduling policies. The kernel could use signals to communication scheduling decisions to a user space component, which implements slicing. The overheads of slicing should be unaffected because signals are asynchronous and no significant changes on the current code that is executed between slice would be required. The overheads would be affected if this is implemented in an inefficient way via polling and a system call. However, such an inefficient implementation is not required, as this can be implemented with signals.

5.3.2 Autotuning Parallel Hard Real-Time Systems

Chapter 4 provides evidence that skeletons can ease the programming of parallel hard real-time systems. As mentioned in the chapter introduction, this is a first step towards a composable set of hard real-time skeletons and opens up scope for future work.

More Sophisticated Programming Models Future work could investigate more sophisticated and more high-level programming models, which hide more of the underlying hardware. For example, hard real-time support for Lift- and RISE-like data parallel programming models could be investigated. Compilers for such hard real-time skeletons might also support heterogeneous systems, which the existing non-real-time implementations of Lift and RISE already support. Additionally, they might also support mid-kernel migration because of the high-level nature of the programming models of Lift and RISE (see the second paragraph in Section 5.3.1 for a discussion of related ideas in non-real-time contexts). Adding hard real-time requirements to heterogeneous systems with manually implemented fine-grained migration would only add further complexity to systems and codebases that are already challenging. The proposed work could significantly simplify their programming and maintenance.

This work would be motivated by existing heterogeneous real-time systems such as the Tesla chip for self-driving cars, which incorporates CPUs, a GPU, and a machine learning accelerator [TSV+20]. In addition, mid-kernel migration for heterogeneous hard real-time systems would allow for schedules which are otherwise not possible. More precisely, this would allow some task sets, which could previously not be scheduled on a given system, to be scheduled because the system could make better use of the available resources.

A Less Restrictive System Model Our system model is too restrictive for some applications because it assumes that new jobs arrive periodically. Future work could extend the presented models to so called *sporadic* and *aperiodic tasks* whose jobs do not arrive periodically [1]. To accommodate such tasks the task farm could retain batches to fill them just until the oldest job in a batch can still be processed in time.

More Extensive Evaluation Our evaluation has some limitations:

[1] This idea is the result of private conversations with attendants of the RTAS 2020 conference and Enrico Bini of the University of Turin.

- We use only one system for the evaluation. Further evaluation systems with different architectures would strengthen our evidence that skeletons can simplify the programming of hard real-time systems without prohibitive overheads or deadline misses.

- We use a set of important isolated kernels in the evaluation. To make an even more convincing case for hard real-time skeletons full applications and benchmark suits for embedded systems could be implemented with skeletons [2]. Further skeletons, which can be combined to implement full applications, are needed for this. A potentially fruitful next step is to develop a real-time pipeline skeleton that can be combined with our task farm. It is likely possible to implement some StreamIt applications and image processing pipelines with a combination of both skeletons. Another source of further suitable benchmarks are the EEMBC benchmark suites [TA10].

Task Farms with Specialised Workers All workers of our task farm execute the same function. However, Thies et al. identified applications that require task farms that can execute specialised workers [TA10]. A simple example is an application in which every second job should go to workers of type A and all other jobs to workers of type B. Workers of type A and B execute different user functions in this example. This would require more sophisticated scheduling logic in the dispatcher (see Section 2.5.2), which in turn would affect farm internal WCETs.

Use the Full XMOS Processor The XMOS microcontroller consists of two CPUs that each have eight logical cores. In our evaluation we use one CPU tile and all cores of it. Future work could investigate the use of both at the same time. A complication is that both CPUs have separate main memories but can communicate via messages. There are two potential strategies to use both CPUs. Firstly, a new farm variant that can manage workers on both CPUs and hides the communication between the CPUs from the programmer. Secondly, use the current farm variant whose instances are confined to one CPU with another skeleton, for example a pipeline, to implement communication between both CPUs. Two instances of this simpler farm could then process different parts of a more complex application on both CPUs and communicate with each other via another skeleton.

[2] https://www.eembc.org/

Limited Applicability of Batching On shared memory systems like our evaluation platform batching saves the costs of copying two pointers. These costs are only substantial if new jobs arrive rapidly, as in our evaluation, with periods of less than $10\mu s$. However, batching serves as an example for optimisations that can be informed by information encoded in skeletons and that can be hidden behind skeletons. In addition, batching can benefit applications with larger periods on other systems with higher inter-thread communication costs.

Ease of Programming The Peso library uses macros for the task farm, which do not fit naturally in the xC programming language (see Section 4.6.1). A compiler implementation with new language constructs or a pragma-based implementation might further simplify programming.

5.4 Final Remarks

The past two decades have seen a proliferation of parallelism in mainstream systems across the computing spectrum [HP19, LNOM08, LTXZ19]. Data centre servers, desktop PCs, and mobile devices now include multithreaded CPUs with SIMD instructions, GPGPUs, and, in some cases, machine learning accelerators or FPGAs. The complexity of these is not hidden behind a single ISA but must be managed in software. However, current mainstream programming languages and libraries often do not fully shield software developers from this complexity, which hurts programmer productivity and code maintainability.

Skeletons abstract away code for the management of parallel hardware, so that application developers do not have to be concerned with this. Instead, parameterisable and reusable low-level implementations are either provided by specialised programmers or are automatically generated by sophisticated compilers [SRD17, ELK17]. In addition, high-level abstractions give more freedom to runtime systems in the choice of tuning parameters and scheduling decisions that would otherwise be baked into application code.

We demonstrate the benefits of skeletons in two novel contexts: OS scheduling for heterogeneous systems, and hard real-time systems. We show that significant implementation complexity can be hidden behind a skeleton interface and that complex scheduling decisions can be automated. In this context, we would like to mention again that in the chapter on migration our skeleton interface reduces code sizes by at least by

88% compared to hand implementations.

As discussed above, this thesis provides a basis for future work in multiple directions. In particular, we hope we will see further research on how high-level programming models can simplify the programming of modern real-time systems and enable scheduling flexibility in programmer transparent ways.

Bibliography

[ACD+12] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting Distributed Systems in FastFlow. In *European Conference on Parallel Processing*, pages 47–56. Springer, 2012.

[Ada16] Ada Conformity Assessment Authority. *Ada Reference Manual*, 2016. ch. Real Time Systems.

[ADK+11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-cores with FastFlow. In *Int'l European Conf. on Parallel and Distributed Computing*, pages 170–181. Springer, 2011.

[ADKT17] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *FastFlow: High-Level and Efficient Streaming on Multi-Core*. John Wiley & Sons, Inc., 2017.

[ARM21] ARM. Register 9, cache lockdown register. `https://developer.arm.com/documentation/ddi0184/b/programmer-s-model/cp15-register-map-summary/register-9--cache-lockdown-register`, 2021. Last accessed: 12/05/2021.

[ASD+12] Marco Aldinucci, Concetto Spampinato, Maurizio Drocco, Massimo Torquati, and Simone Palazzo. A Parallel Edge Preserving Algorithm for Salt and Pepper Image Denoising. In *Int'l Conf. on Image Processing Theory, Tools and Applications*, pages 97–104. IEEE, 2012.

[ATNW09] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling

on Heterogeneous Multicore Architectures. In *European Conf. on Parallel Processing*, pages 863–874. Springer, 2009.

[AUT14] AUTOSAR. *AUTOSAR Guide to Multi-Core Systems*, March 2014.

[Bar20] Barcelon Supercomputing Center (BSC). *OmpSs-2 Specification*, December 2020.

[BBH+19] David Alexander Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas R.W. Scogland. RAJA: Portable Performance for Large-Scale Scientific Applications. In *Int'l. Workshop on Performance, Portability and Productivity in HPC*, pages 71–81. IEEE, 2019.

[BBM+12] Siegfried Benkner, Enes Bajrovic, Erich Marth, Martin Sandrieser, Raymond Namyst, and Samuel Thibault. High-level Support for Pipeline Parallelism on Many-Core Architectures. In *European Conf. on Parallel and Distributed Computing*, pages 614–625. Springer, 2012.

[BBMS+12] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A Generalized Parallel Task Model for Recurrent Real-Time Processes. In *Real-Time Systems Symp.*, pages 63–72. IEEE, 2012.

[BBW11] Giorgio Buttazzo, Enrico Bini, and Yifan Wu. Partitioning Real-Time Applications Over Multicore Reservations. *Transactions on Industrial Informatics*, 7(2):302–315, May 2011. Published by the IEEE.

[BCL+18] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *USENIX Annual Technical Conference*, pages 85–96, 2018.

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Int'l Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[BK12] Can Basaran and Kyoung-Don Kang. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *Euromicro Conf. on Real-Time Systems*, pages 287–296. IEEE, 2012.

[BLC16] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. *The International Journal of High Performance Computing Applications*, 31(5):391–404, 2016. Published by SAGE.

[BLR+16] Kevin J. Brown, Hyouk Joong Lee, Tiark Romp, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *Int'l Symp. on Code Generation and Optimization*, pages 194–205. ACM, 2016.

[BMSSW13] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility Analysis in the Sporadic DAG Task Model. In *Euromicro Conference on Real-Time Systems*, pages 225–233. Euromicro, CPS, 2013.

[Boa20] OpenMP Architecture Review Board. OpenMP Application Programming Interface, November 2020. Version 5.1.

[Bre01] Leo Breiman. Random Forests. *Machine learning*, 45:5–32, 2001.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Science+Business Media, 2011. Third Edition, see Section 1.2.1 for a discussion of hard and soft real-time systems, page 9, see Section 2.2.1 for a discussion of some of the real-time systems terminology, pages 27–28, see Section 12.5.1 for two WCET analysis tools, page 453.

[BYA+19] Ashikahmed Bhuiyan, Kecheng Yang, Samsil Arefin, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Mixed-Criticality Multicore Scheduling of Real-Time Gang Task Systems. In *Real-Time Systems Symp.*, pages 469–480. IEEE, 2019.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark

Suite for Heterogeneous Computing. In *Int'l. Symp. on Workload Characterization*, pages 44–54. IEEE, 2009.

[CGK11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *Symp. on Principles and Practice of Parallel Programming*, pages 47–56. ACM, 2011.

[CGW20] Chris Cummins, Dominik Grewe, and Zheng Wang. Feature Extractor. `https://github.com/ChrisCummins/phd/blob/master/research/grewe_2013_cgo/feature_extractor_binary.cc`, 2020. Last accessed: 02/06/2021.

[CHCF15] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. Lira: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems. In *Int'l. Workshop on Runtime and Operating Systems for Supercomputers*, pages 1–8. ACM, 2015.

[CHCR11] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *Int'l. Symp. on Microarchitecture*, pages 175–185. IEEE, 2011.

[CHZ+18] Yanhao Chen, Ari B. Hayes, Chi Zhang, Timothy Salmon, and Eddy Z. Zhang. Locality-Aware Software Throttling for Sparse Matrix Operation on GPUs. In *USENIX Annual Technical Conf.*, pages 413–425. USENIX, 2018.

[CKB13] Timothy Creech, Aparna Kotha, and Rajeev Barua. Efficient Multiprogramming for Multicores with SCAF. In *Int'l. Symp. on Microarchitecture*, pages 334–345. ACM, 2013.

[CKC12] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big.LITTLE™ Technology. *Samsung White Paper*, 2012.

[CMDD62] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An Experimental Time-Sharing System. In *Spring Joint Computer Conference*, pages 335–344. ACM, 1962.

[CNP+18] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R. Gross. On-The-Fly Workload Partitioning for Integrated

CPU/GPU Architectures. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, pages 1–13. ACM, 2018.

[Col04] Murray Cole. Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. Published by Elsevier.

[Cor17] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms In MATLAB® Second, Completely Revised*, volume 118. Springer, 2017. See Chapter 2 and Section 12.7.4 for uses of matrix multiplication, pages 17–61 and page 405.

[CPWL17] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end Deep Learning of Optimization Heuristics. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 219–232. IEEE, 2017.

[CSG99] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999. See Section 1.5 for an overview over the history of parallel computers, including the IBM 370 and the Transputer, pages 66–68. See Section 3.1.1 for the quoted text on task parallelism, page 124. See Section 11.7.1 for a discussion of interleaved multithreading, pages 902–904.

[CYG+17] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Int'l. Conf on Architectural Support for Programming Languages and Operating Systems*, pages 17–32. ACM, 2017.

[dD19] Benoît Dupont de Dinechin. Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore. In *Design Automation Conference*, pages 1–4. ACM, 2019.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. Published by the ACM.

[DK13] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013. Published by the ACM.

[DK16] Usman Dastgeer and Christoph Kessler. Smart Containers and Skeleton Programming for GPU-Based Systems. *International Journal of Parallel Programming*, 44(3):506–530, 2016.

[dKSJ12] Marc de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static Analysis and Compiler Design for Idempotent Processing. In *Conf. on Programming Language Design and Implementation*, pages 475–486. ACM, 2012.

[DLK13] Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive Implementation Selection in the SkePU Skeleton Programming Library. In *Int'l Workshop on Advanced Parallel Processing Technologies*, pages 170–183. Springer, 2013.

[DMM+10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In *Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[dNLR09] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Real-Time Systems Symp.*, pages 291–300. IEEE, 2009.

[DSDMT+17] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite. *Transactions on Architecture and Code Optimization*, 14(4):33:1–33:26, 2017. Published by the ACM.

[DSTD16] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. A Reconfiguration Algorithm for Power-Aware Parallel Applications. *Transactions on Architecture and Code Optimization*, 13(4):43:1–43:25, 2016. Published by the ACM.

[DY08] Gregory Diamos and Sudhakar Yalamanchili. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In *Int'l. Symp. on High Perf. Distributed Computing*, pages 197–200. ACM, 2008.

[EBA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multi-core Scaling. In *Int'l Symp. on Computer Architecture*, pages 365–376. ACM, 2011.

[EK10] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. In *Int'l. Workshop on High-level Parallel Programming and Applications*, pages 5–14. ACM, 2010.

[EK16] Steffen Ernsting and Herbert Kuchen. Data Parallel Algorithmic Skeletons with Accelerator Support. *Int'l Journal of Parallel Programming*, 45(2):283–299, 2016. Published by Springer.

[EK18] August Ernstsson and Christoph Kessler. Extending Smart Containers for Data Locality-Aware Skeleton Programming. *Concurrency and Computation: Practice and Experience*, 31(5):1–13, 2018. Published by John Wiley & Sons, Ltd.

[EK20] August Ernstsson and Christoph Kessler. Multi-Variant User Functions for Platform-Aware Skeleton Programming. *Advances in Parallel Computing*, 36:475–484, 2020. Published by IOS Press, source: `https://www.ida.liu.se/~chrke55/papers/APC-36-APC200074.pdf`, last accessed: 28/05/2021.

[ELK17] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *Int'l Journal of Parallel Programming*, 46(1):62–80, 2017. Published by Springer.

[ETS14] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Published by Elsevier.

[FAN+13] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data Transfer Matters for GPU Computing. In *Int'l Conf. on Parallel and Distributed Systems*, pages 275–282. IEEE, 2013.

[FJO+16] Martin Frieb, Ralf Jahr, Haluk Ozaktas, Andreas Hugl, Hans Regler, and Theo Ungerer. A Parallelization Approach for Hard Real-Time Systems and Its Application on Two Industrial Programs. *Int'l Journal of Parallel Programming*, 44(6):1296–1336, 2016. Published by Springer.

[FLM+13] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A Real-Time Scheduling Service for Parallel Tasks. In *Real-Time and Embedded Technology and Applications Symp.*, pages 261–272. IEEE, 2013.

[GHF+06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *Micro*, 26(2):10–24, 2006. Published by the IEEE.

[GO11] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Int'l. Conf. on Compiler Construction*, pages 286–305. Springer, 2011.

[Gor04] Sergei Gorlatch. Send-Receive Considered Harmful: Myths and Realities of Message Passing. *Transactions on Programming Languages and Systems*, 26(1):47–56, 2004. Published by the ACM.

[Gro20] Khronos SYCL™ Working Group. *SYCL™ Specification, SYCL™ Integrates OpenCL™ Devices with Modern C++*. Khronos Group, April 2020. Version 1.2.1.

[GSB14] Ramy Gad, Tim Süß, and André Brinkmann. Compiler Driven Automatic Kernel Context Migration for Heterogeneous Computing. In *Int'l. Conf. on Distributed Computing Systems*, pages 389–398. IEEE, 2014.

[GVL10] Horacio González-Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[GWO13] Dominik Grewe, Zheng Wang, and Michael F.P. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Int'l. Symp. on Code Generation and Optimization*, pages 1–10. IEEE, 2013.

[HGL12] Huang-Ming Huang, Christopher Gill, and Chenyang Lu. MCFlow: A Real-Time Multi-Core Aware Middleware for Dependent Task Graphs. In *Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 104–113. IEEE, 2012.

[HJGG19] Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo. Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores. *Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019. Published by the IEEE.

[HLK+20] Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. Achieving High-Performance the Functional Way. *Programming Languages*, 4(ICFP):92:1–92:29, 2020. Published by the ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Int'l Symp. on Computer Arch.*, pages 289–300, 1993.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017. see Section 1.2 for the quoted text on data-level parallelism, page 10, see Section 1.4 for a discussion of Moore's Law, page 19, see Section 1.9 for a discussion of Amdahl's Law, pages 49–50, see Chapter 3 for comments on the increasing difficulty to further exploit ILP in the early 2000s, which motivated multithreaded and multicore architectures, pages 168–266, see Chapter 4 or a discussion of SIMD instructions, vector processors,

and GPGPUs, pages 282–357, see the subsection on SDRAM in Section 2.2, page 89, see Appendix E.1 for a discussion of worst-case execution times, no page numbers.

[HP19] John L. Hennessy and David A. Patterson. A New Golden Age for Computer Architecture. *Communications of the ACM*, 62(2):48–60, 2019. Published by the ACM.

[Hun19] James J. Hunt. *Realtime and Embedded Specification for Java Version 2.0 Draft 72*. TimeSys and aicas GmbH, March 2019.

[JFGU14] Ralf Jahr, Martin Frieb, Mike Gerdes, and Theo Ungerer. Model-based Parallelization and Optimization of an Industrial Control Code. In *Dagstuhl-Workshops*, pages 97–106, 2014.

[KB06] Vida Kianzad and Shuvra S. Bhattacharyya. Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors. *Transactions on Parallel and Distributed Systems*, 17(7):667–680, 2006. Published by the IEEE.

[KBS+14] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, pages 151–162. ACM, 2014.

[KCLL+05] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming. In *Conf. on Programming Language Design and Implementation*, pages 224–236. ACM, 2005.

[kdc20] The kernel development community. 4. Energy-Aware task placement. `https://www.kernel.org/doc/html/v5.10/scheduler/sched-energy.html`, December 2020. Last accessed: 10/05/2021.

[Ker10] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

[KHL+19] Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, and Hwansoo Han. Compiler-Assisted GPU Thread Throttling for Reduced

Cache Contention. In *Int'l. Conf. on Parallel Processing*, pages 1–10. ACM, 2019.

[KJKD13] Onur Kayıran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, pages 157–166. IEEE, 2013.

[KLK+11] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan (Raj) Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *Real-Time Systems Symp.*, pages 57–66. IEEE, 2011.

[KMHK12] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring Interference Between Live Datacenter Applications. In *Int'l. Conf. on High Perf. Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[KRD+03] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable Stream Processors. *IEEE Computer*, 36(8):54–62, 2003. Published by the IEEE.

[KS02] Herbert Kuchen and Jörg Striegnitz. Higher-Order Functions and Partial Applications for a C++ Skeleton Library. In *Joint ACM-ISCOPE Conference on Java Grande*, pages 122–130. ACM, 2002.

[KS21] Thomas Kœhler and Michel Steuwer. Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs. In *Int'l Symp. on Code Generation and Optimization*, pages 27–38. IEEE, 2021.

[LALG13] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of Global EDF for Parallel Tasks. In *Euromicro Conference on Real-Time Systems*, pages 3–13. Euromicro, CPS, 2013.

[LCG+15] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Int'l. Symp. on Computer Architecture*, pages 450–462. ACM, 2015.

[Lee06] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, 2006. Published by the IEEE.

[LFC13] Thibaut Lutz, Christian Fensch, and Murray Cole. PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. *Transactions on Architecture and Code Optimization*, 9(4):59:1–59:24, 2013. Published by the ACM.

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Int'l. Symp. on Microarchitecture*, pages 45–55. ACM, 2009.

[LL73] Chung Laung Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973. Published by the ACM.

[LLF+14] Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Global EDF Scheduling for Parallel Real-Time Tasks. *Real-Time Systems*, 51(4):395–439, 2014. Published by Springer.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro*, 28(2):39–55, 2008. Published by the IEEE.

[Loc53] John Locke. *An Essay Concerning Human Understanding. In Four Books.* Robert Taylor (Berwick), 1753. 15th edition, page 135.

[LP20] Achim Lösch and Marco Platzner. MigHEFT: DAG-based Scheduling of Migratable Tasks on Heterogeneous Compute Nodes. In *Int'l. Parallel and Distributed Processing Symp. Workshops*, pages 6–16. IEEE, 2020.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 329–339. ACM, 2008.

[LR11] Justin Luitjens and Steven Rennich. CUDA Warps and Occupancy – GPU Computing Webinar, December 2011. presentation slides by nVidia, see slide 6 for an explanation of occupancy.

[LTXZ19] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A Scalable Architecture for Neural Network Computing. In *Hot Chips Symp.*, pages 1–44. IEEE, 2019. `https://old.hotchips.org/hc31/HC31_1.11_Huawei.Davinci.HengLiao_v4.0.pdf`, last accessed: 13/05/2021.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Int'l Conf on Management of Data*, pages 135–146. ACM, 2010.

[May] David May. *xCORE-200: The XMOS XS2 Architecture*. XMOS, April.

[MCF18] Paul Metzger, Murray Cole, and Christian Fensch. NUMA Optimizations for Algorithmic Skeletons. In *Int'l European Conference on Parallel and Distributed Computing*, pages 590–602. Springer, 2018.

[MM07] Thomas Moscibroda and Onur Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security Symp.*, pages 256–264, 2007.

[Moo98] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. Published by the IEEE.

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[MSH11] John W. McCormick, Frank Singhoff, and Jérôme Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, 2011.

[nVia] nVidia. cuBLAS. `https://developer.nvidia.com/cublas`. Last accessed: 10/05/2021.

[nVib] nVidia. CUDA Samples Version 11. https://github.com/NVIDIA/cuda-samples/blob/v11.0/Samples/reduction/reduction_kernel.cu, see function *reduce4*, last accessed 26/04/2021.

[nVic] nVidia. *nVidia Titan RTX Product Overview*, May.

[nVi21] nVidia. B.18. Warp Vote Functions. https://docs.nvidia.com/cuda/archive/11.2.2/cuda-c-programming-guide/index.html#warp-vote-functions, March 2021. Last accessed: 10/05/2021.

[ÖEK19] Tomas Öhberg, August Ernstsson, and Christoph Kessler. Hybrid CPU–GPU Execution Support in the Skeleton Programming Framework SkePU. *The Journal of Supercomputing*, 76(7):5038–5056, 2019.

[Ope18] The Open Group and IEEE. *The Open Group Base Specifications Issue 7, 2018 Edition*, 2018. ch. 2.8 Realtime.

[PBAL13] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Self-adaptive OmpSs Tasks in Heterogeneous Environments. In *Int'l Symp. on Parallel and Distributed Processing*, pages 138–149. IEEE, 2013.

[PBAL15] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. SSMART: Smart Scheduling of Multi-Architecture Tasks on Heterogeneous Systems. In *Workshop on Accelerator Programming using Directives*, pages 1–11. ACM, 2015.

[PG14] Prasanna Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Int'l. Symp. on Code Generation and Optimization*, pages 273–283. ACM, 2014.

[PH05] David. A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. third edition, 2005.

[PNY+15] Luís Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, and Michal Mardiak. P-SOCRATES: A Parallel Software Framework for Time-Critical Many-

Core Systems. *Microprocessors and Microsystems*, 39(8):1190–1203, 2015. Published by Elsevier.

[PS14] PCI-SIG. PCI Express® 3.0 Frequently Asked Questions. `https://web.archive.org/web/20140201172536/http://www.pcisig.com/news_room/faqs/pcie3.0_faq/`, 2014. Last accessed: 10/05/2021.

[PS16] Sankaralingam Panneerselvam and Michael Swift. Rinnegan: Efficient Resource Use in Heterogeneous Architectures. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, pages 373–386. ACM, 2016.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. No publisher but hosted by the ACM.

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *European Conf. on Computer Systems*, pages 275–288. ACM, 2009.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013. Published by the ACM.

[RVKP19] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. Transparent Acceleration for Heterogeneous Platforms With Compilation to OpenCL. *Transactions on Architecture and Code Optimization*, 16(2):1–26, 2019. Published by the ACM.

[RWK19] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. Musket: A Domain-Specific Language for High-Level Parallel Programming with Algorithmic Skeletons. In *Symp. on Applied Computing*, pages 1534–1543. ACM, 2019.

[RZLA] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: A System for Flexible Parallel Execution. *ACM SIGPLAN Notices*, 47(6):133–144.

[Sar87] Vivek Sarkar. Partitioning and Scheduling Parallel Programs for Execution Multiprocessors. 1987. PhD thesis at Stanford University.

[SBL+14] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *Transactions on Embedded Computing Systems*, 13(4s):134:1–134:25, 2014. Published by the ACM.

[SFJU] Alexander Stegmeier, Martin Frieb, Ralf Jahr, and Theo Ungerer. Algorithmic Skeletons for Parallelization of Embedded Real-Time Systems. In *Workshop on High-Performance and Real-time Embedded Systems*, pages 1–12.

[SFL+14] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Parallel Real-Time Scheduling of DAGs. *Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014. Published by the IEEE.

[SFLD15] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. *ACM SIGPLAN Notices*, 50(9):205–217, 2015. Published by the ACM.

[SGG19] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts Global Edition*. John Wiley & Sons, 10th edition, 2019. See Section 9.3.2.1 for a discussion of ASIDs, 396.

[SGL+20] Jinghao Sun, Nan Guan, Feng Li, Huimin Gao, Chang Shi, and Wang Yi. Real-Time Scheduling and Analysis of OpenMP DAG Tasks Supporting Nested Parallelism. *Transactions on Computers*, 69(9):1335–1348, 2020. Published by the IEEE.

[SGS14] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *Conf. on Programming Language Design and Implementation*, pages 169–180. ACM, 2014.

[SGSC19] Jinghao Sun, Nan Guan, Jingchang Sun, and Yaoyao Chi. Calculating Response-Time Bounds for OpenMP Task Systems with Conditional Branches. In *Real-Time and Embedded Technology and Applications Symp.*, pages 169–181. IEEE, 2019.

[SGW+17] Jinghao Sun, Nan Guan, Yang Wang, Qingqiang He, and Wang Yi. Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks. In *Real-Time Systems Symp.*, pages 92–103. IEEE, 2017.

[SGW+19] Jinghao Sun, Nan Guan, Xiaoqing Wang, Chenhan Jin, and Yaoyao Chi. Real-Time Scheduling and Analysis of Synchronous OpenMP Task Systems with Tied Tasks. In *Design Automation Conference*, pages 1–6. ACM, 2019.

[SL05] Herb Sutter and James Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, 2005. Published by the ACM.

[SLA+12] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-Core Real-Time Scheduling for Generalized Parallel Task Models. *Real-Time Systems*, 49(4):404–435, 2012. Published by Springer.

[SLH12] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W. Hwu. DL: A Data Layout Transformation System for Heterogeneous Computing. In *Innovative Parallel Computing*, pages 1–11. IEEE, 2012.

[SQ18] Maria A. Serrano and Eduardo Quinones. Response-Time Analysis of DAG Tasks Supporting Heterogeneous Computing. In *Design Automation Conf.*, pages 1–6. ACM, 2018.

[SRD17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *Int'l Symp. on Code Generation and Optimization*, pages 74–85. IEEE, 2017.

[SVZ+14] Jie Shen, Ana Lucia Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In *Int'l. Conf. on Supercomputing*, pages 241–250. ACM, 2014.

[TA10] William Thies and Saman Amarasinghe. An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 365–376. IEEE, 2010.

[TB15] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems Global Edition*. Pearson, 2015. Fourth Edition.

[TBG+17] Qi Tang, Twan Basten, Marc Geilen, Sander Stuijk, and Ji-Bo Wei. Mapping of Synchronous Dataflow Graphs on MPSoCs Based on Parallelism Enhancement. *Journal of Parallel and Distributed Computing*, 101:79–91, 2017. Published by Elsevier.

[TKS+11] Hiroyuki Takizawa, Kentaro Koyama, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In *Int'l. Parallel & Distributed Processing Symp.*, pages 864–876. IEEE, 2011.

[TMW17] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems. In *ACM SIGPLAN Notices*, pages 11–20. ACM, 2017.

[TSV+20] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev. Compute Solution for Tesla's Full Self-Driving Computer. *IEEE Micro*, 40(2):25–35, 2020. Published by the IEEE.

[UBF+16] Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume

Abella, Carles Hernández, Francisco Cazorla, Sascha Uhrig, Mathias Rohde, and Arthur Pyka. Parallelizing Industrial Hard Real-Time Applications for the parMERASA Multicore. *Transactions on Embedded Computing Systems*, 15(3):53:1–53:27, 2016. Published by the ACM.

[UBG+13] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. G. Zaykov, Z. Petrov, B Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA – Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *Euromicro Conference on Digital System Design*, pages 363–370. IEEE, 2013.

[Uni20] Université de Bordeaux, CNRS (LaBRI UMR 5800), Inria. *StarPU Handbook for StarPU 1.3.7*, 10 2020. Last accessed: 11/05/2021.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990. Published by the ACM.

[Ves07] Steve Vestal. Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. In *Int'l. Real-Time Systems Symp.*, pages 239–243. IEEE, 2007.

[VPK+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *European Conf. on Computer Systems*, pages 1–17. ACM, 2015.

[WGS+17] Yang Wang, Nan Guan, Jinghao Sun, Mingsong Lv, Qingqiang He, Tianzhang He, and Wang Yi. Benchmarking OpenMP Programs for Real-Time Scheduling. In *Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2017.

[WK20] Fabian Wrede and Herbert Kuchen. Towards High-Performance Code Generation for Multi-GPU Clusters Based on a Domain-Specific Language for Algorithmic Skeletons. *Int'l Journal of Parallel Programming*, 48:713–728, 2020. Published by Springer.

[WO09] Zheng Wang and Michael F.P. O'Boyle. Mapping Parallelism to Multicores: A Machine Learning Based Approach. In *Symp. on Principles and Practice of Parallel Programming*, pages 75–84. ACM, 2009.

[WWX+16] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. BLASX: A High Performance Level-3 BLAS Library for Heterogeneous Multi-GPU Computing. In *Int'l Conf. on Supercomputing*, pages 1–11. ACM, 2016.

[XMO13] XMOS Ltd. *XMOS Timing Analyzer Manual Revision B*, May 2013.

[XMO15] XMOS Ltd. *XMOS Programming Guide*, September 2015. Sections 1.2 and 2.2.

[XMO18] XMOS Ltd. *XE216-512-TQ128 Datasheet*, September 2018.

[YAY+18] Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Shige Wang. Making OpenVX Really "Real Time". In *Real-Time Systems Symp.*, pages 80–93. IEEE, 2018.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Workshop on Hot Topics in Cloud Computing*, pages 1–7. USENIX, 2010.

[ZTL15] Husheng Zhou, Guangmo Tong, and Cong Liu. GPES: A Preemptive Execution System for GPGPU Computing. In *Real-Time and Embedded Technology and Applications Symp.*, pages 87–97. IEEE, 2015.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65, 2016. Published by the ACM.