Algorithms and Framework for Energy Efficient Parallel Stream Computing on Many-Core Architectures

Nicolas Melot

Linköping Studies in Science and Technology

Dissertations. No. 1813

Algorithms and Framework for Energy Efficient Parallel Stream Computing on Many-Core Architectures

by

Nicolas Melot

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2016

Copyright © 2016 Nicolas Melot

ISBN 978-91-7685-623-9

ISSN 0345-7524

Printed by LiU Tryck 2016

Information and errata list: <http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-132308>

Abstract

The rise of many-core processor architectures in the market answers to a constantly growing need of processing power to solve more and more challenging problems such as the ones in computing for big data. Fast computation is more and more limited by the very high power required and the management of the considerable heat produced. Many programming models compete to take profit of many-core architectures to improve both execution speed and energy consumption, each with their advantages and drawbacks. The work described in this thesis is based on the dataflow computing approach and investigates the benefits of a carefully pipelined execution of streaming applications, focusing in particular on off- and on-chip memory accesses. As case study, we implement classic and on-chip pipelined versions of mergesort for Intel SCC and Xeon. We see how the benefits of the on-chip pipelining technique are bounded by the underlying architecture, and we explore the problem of fine tuning streaming applications for many-core architectures to optimize for energy given a throughput budget. We propose a novel methodology to compute schedules optimized for energy efficiency given a fixed throughput target. We introduce *Drake*, derived from Schedeval, a tool that generates pipelined applications for Many-Core architectures and allows the performance testing in time or energy of their static schedule. We show that streaming applications based on Drake compete with specialized implementations and we use Schedeval to demonstrate performance differences between schedules that are otherwise considered as equivalent by a simple model.

This work has been supported in parts by CUGS (the Graduate School in Computer Science, Sweden), Vetenskapsrådet, SeRC and EU FP7 EXCESS.

Acknowledgements

I would like to thank all members of PELAB for the inspiring and stimulating working environment as well as the passionate discussions around coffee or tea. In particular, my thanks to Kristian Sandahl for his efforts at maintaining a strong group culture. Warm thanks to Christoph Kessler, for all fruitful discussions and ideas when my imagination came short, for his support and patience when results came late and for entrusting me with opportunities and responsibilities that taught me valuable experiences. Many thanks to Jörg Keller for sharing his ideas in details about my work, and for always carefully reviewing and challenging any hypothesis I proposed, as it repeatedly resulted in strengthening good ideas and discarding bad ones. I would like to thank all students who participated in the work, providing a precious help to progress. I am grateful to Intel for providing the Single Chip Cloud computer research prototype and their efforts to help in the numerous moments where nothing worked. Many thanks to the National Supercomputer Center (NSC) for providing powerful computing means to run my experiments, TUS for always providing a quick and precious technical assistance with university resources and peculiar machines I employed in my work. Thanks to the numerous anonymous reviewers for giving me a honest, although painful at times, feedback about my work. I appreciated the scientific community I have been part of for its open, modest, and very smart culture where only arguments' soundness matters. This thesis would not have met its current quality without the careful, patient and repetitive proof reading assistance from Christoph Kessler for the scientific content and Anne Moe for the popular science paragraph in Swedish. My thanks to all administrative staff for having made my settlement in Linköping much easier and helped me in all administrative tasks here at IDA, and even sometimes helped me coping with French administration requirements. A particular thought to Åsa Kärrman, who started her employment at the same time as I did at IDA 6 years ago, but who will stay here longer than I do.

Populärvetenskaplig sammanfattning

Utvecklingen inom halvledarteknik har gjort det möjligt att förbättra processorer genom att fler och mindre transistorer används för vanligt förekommande funktioner så att processorerna skall kunna köras snabbare. En annan populär väg att få snabbare processorer är att köra dem med högre klock-frekvens. Precis som styrkan i en fors pressar vattnet genom en vattenkvarn, använder CMOS teknologi elektrisk spänning för att trycka elektroner genom transistorer när de växlar från 0 till 1 eller tillbaka, dock med följden att några elektroner tappas samtidigt. Ju snabbare transistorerna växlar, desto högre spänning behöver elektronerna för att flöda genom transistorn i tid, men samtidigt innebär högre spänning att ännu fler elektroner tappas från transistorn. Detta ökar energikostnaden till den gräns att transistorer inte kan växla snabbare idag.

Vi behöver dock fortfarande snabbare datorer. Istället för att ha snabba processorer kan vi integrera fler processorkärnor på ett chipp och därigenom fördela arbetet i ett datorprogram på flera kärnor och följaktligen köra programmet snabbare. Processorer kan, på så sätt, arbeta mera energieffektivt. Problemet med parallella program är att de är svårare att utveckla än sekventiella program (program som inte kan delas upp mellan flera kärnor) och utvecklingen blir svårare ju fler kärnor som skall dela på arbetet. Kärnorna behöver dessutom dela data, till exempel i ett delat minne, för att kunna arbeta tillsammans; de använder ett snabbt nätverk på chipet för att skicka data mellan varandra. Om man integrerar för många kärnor upptar det snabba nätverket en allt större del av chipet som då inte kan användas för att arbeta snabbare. Istället för att alltid dela alla data i ett delat minne kan man låta kärnorna skicka data explicit till en eller flera andra kärnor endast där behov uppstår. På så sätt behöver inte processorerna så stort och snabbt nätverk och framtidens processorer kan köra snabbare. Tyvärr blir program för den typen av processorer också svårare att utveckla.

Strömprogrammering har studerats sedan mer än 50 år, ursprungligen i ett annat syfte än högprestandaberäkningar, men det passar också bra för kärnor som utbyter data. Strömprogram består av aktörer (mindre delar av ett program) som skickar data mellan varandra. I den här avhandlingen studerar vi hur man kan utveckla och optimera strömprogram för massivt parallella processorer. Vi fokuserar på energieffektiva tekniker för att bestämma hur många kärnor en aktör ska tilldelas (tilldelning), vilka kärnor som ska köra den (mappning) och med vilken klockfrekvens (frekvensskalning). Om detta görs för alla aktörer i programmet kan programmet köras så snabbt som det behövs istället för så snabbt som möjligt. Till exempel, om vi vill optimera en videoläsningssapp så att den visar minst 30 bildrutor varje sekund, så behöver appen inte visa fler och därmed förbruka mer energi från batteriet än nödvändigt. Med den experimentella processorn SCC (Single Chip Cloud computer) från Intels forskningsprogram Tera-Scale Computings studerar vi hur man kan använda kärnor och nätverk för att utveckla effektiva program. Vi introducerar Crown Scheduling, en teknik som gör exekveringen av strömprogram mer energieffektiv. Till exempel, kan den välja lägre klockfrekvens för vissa aktörer eller stänga av oanvända kärnor men fortfarande uppfylla appens prestandakrav. Vi redovisar också detaljer om Drake, ett programmeringsramverk som använder en Crown Scheduler för att generera effektiva strömprogram för massivt parallella processorer.

Contents

1	Introduction	1
2	Background	11
2.1	Introduction	11
2.2	Streaming Computation	11
2.2.1	Semantics	12
2.2.2	Properties of Computation over Streams	13
2.3	Processor Architectures Overview	15
2.3.1	Platform model	15
2.3.2	Single Chip Cloud computer (SCC)	17
2.3.3	Intel Knights Corner	19
2.3.4	Intel Knights Landing	19
2.3.5	Sony-Toshiba-IBM Cell	20
2.3.6	Tilera Tile, Tile-Gx and TilePro	20
2.3.7	Adapteva Epiphany	21
2.3.8	Kalray MPPA	22
2.3.9	Intel Xeon	23
2.3.10	Other Parallel Architectures	23
2.4	Programming for High Performance on Many-Core Architec- tures	24
2.4.1	Pthreads	24
2.4.2	OpenMP	24
2.4.3	CUDA	24
2.4.4	OpenCL	25
2.4.5	Streamit	25
2.4.6	CAL Actor Language	27
2.5	Conclusion	27
3	Investigations on SCC Capabilities	28
3.1	Introduction	28
3.2	Memory Bandwidth	28
3.3	Communication Latency and Bandwidth	32
3.4	Mergesort	32
3.5	Conclusion	42

4	On-Chip Pipelining on SCC	45
4.1	Introduction	45
4.2	A Hybrid Parallel Mergesort Algorithm	46
4.2.1	Hybrid Parallel Sorting on the SCC, Overview	47
4.2.2	Optimizing Task Mapping by ILP	48
4.2.3	Phase 1: On-Chip Pipelined Merge	52
4.2.4	Phase 2: Parallel Non-Pipelined Merge	54
4.3	Experimental Evaluation	55
4.4	Conclusion	57
5	Scheduling	58
5.1	Introduction	59
5.2	Crown Scheduling	62
5.3	Phase-Separated Energy-Efficient Crown Scheduling	65
5.3.1	Crown-optimal Crown Resource Allocation	66
5.3.2	Crown-optimal Task Mapping	67
5.3.3	Heuristic Task Mapping with Load Balancing	68
5.3.4	Crown-optimal Voltage/Frequency Scaling of Schedules	70
5.3.5	Height Heuristic for Voltage/Frequency Scaling	71
5.3.6	Binary Search Allocation Heuristic	72
5.3.7	Simulated Annealing	74
5.4	Integrated Energy-Efficient Crown Scheduling	75
5.5	Crown Scheduling Extensions	76
5.5.1	Energy model	77
5.5.2	Crown Configuration	78
5.5.3	Dynamic Crown Rescaling	80
5.5.4	Core Consolidation	81
5.5.5	Island-aware Crown Scheduler	83
5.6	Restriction-Free Optimal Scheduler	86
5.6.1	ILP formulation	86
5.6.2	Cost of Crown Scheduling Structure	88
5.7	Experimental Evaluation	92
5.7.1	Crown Scheduling with no Idle Energy	93
5.7.2	Crown vs Non-Crown	108
5.7.3	Core Consolidation	111
5.7.4	Island-Aware Crown Scheduling	116
5.8	Related Work	122
5.9	Conclusion	130
6	Energy Evaluation of Streaming Applications	133
6.1	Introduction	133
6.2	Overview	134
6.2.1	Drake Streaming Application	135
6.2.2	Memory Management	143
6.2.3	Platform Plugin	146
6.3	Evaluation	150

6.3.1	Drake Overhead	151
6.3.2	Frequency Scaling	153
6.3.3	Computation Speed with Drake on SCC	157
6.3.4	Energy Consumption with Drake on SCC	160
6.3.5	Performance Test with Mergesort on Intel IA64	163
6.4	Related Work	168
6.5	Conclusion	171
7	Software	173
7.1	Pelib	174
7.1.1	Terminal Front-End	175
7.1.2	C Data Structures	176
7.2	Crown Schedulers	178
7.3	Freja	182
7.4	Mimer	184
8	Conclusion	189
	Appendices	192
A	Pelib	193
A.1	Options for pelib-convert	193
A.2	Architecture	197
A.2.1	Algebra	202
A.2.2	Taskgraph	203
A.2.3	Platform	204
A.2.4	Schedule	204
A.3	Plugins	204
A.3.1	Parsing and Output Plugins	204
A.3.2	Process Plugins	205
A.3.3	Solve Plugins	205
A.3.4	Scheduling Plugins	206
A.4	Define C Data Structures for Pelib	208
B	Crown Scheduler Software Architecture	216
B.1	Crown Modular	218
B.2	Crown ILP Integrated	218
B.3	Crown Binary	218
B.4	Crown Composite	219
B.5	Crown Configuration	219
C	Freja	220
C.1	Experiment Scenario	220
C.2	Preparing a Set of Experiments	222
C.2.1	Preparing Environment Settings	223
C.2.2	Preparing Experiment Runs	224

C.3	Running Experiments	224
C.4	Manual Error Handling	228
C.5	Plotting	230
C.6	Additional Options	233
D	Mimer	236
D.1	Energy Evaluation	236
D.2	Data Analysis	237
D.3	Mimer Experiment	237
	Appendix	193

Chapter 1

Introduction

Constant efforts have been made through the years to steadily improve the performance of computer systems. From the earliest forms of automated data processing systems such as Hollerith's census machine [58], computers' hardware was developed to make it more reliable, faster, and less energy consuming. Major improvements include the replacement of mechanical parts with faster and more reliable electronic equivalent (e.g. vacuum tubes and more recently, solid-state drives) and miniaturization with transistors and later integrated circuits. Until recently, microprocessors gained in performance through a fast paced growth in the number of integrated transistors, following the self-fulfilling Moore's law [33]. Today, integrated circuits with a structure size in the order of magnitude as small as a dozen of nanometers face many challenges, and industries recently struggle to follow Moore's law when producing new processors.

Until recently, further improvements of computers' processing power came from the increase of frequency so processors accomplish more work in the same unit of time, with no need to adapt programs already existing. However, as power consumption is often modeled as f^α where $\alpha \approx 3$, it becomes harder and harder to supply systems with enough power to run them at higher speeds. Because of the speedup being, at best, linear with frequency, the total energy consumed by an application is the product between its work and f^2 . In other words, the acceleration of processors already running at a high frequency results in poor reductions of execution time and a great increase of energy consumption; this is called the power wall. Because of the power wall, the frequency of processors stagnates since about 2005 [30] and this technique does not allow for improvements anymore.

Parallel processing bypasses the power wall limit by multiplying the number of processing units (cores), therefore multiplying the output produced in a unit of time by the same factor for the same frequency. The power model above is multiplied by the number of cores employed, that still allows for a linear increase of power consumption for a linear performance improve-

ment through parallelism and a constant frequency. However, compilers and programmers face many challenges to design efficient parallel programs, distributing workload equally among cores, and using the complex memory structure (Fig. 1.1) efficiently. Inefficiencies related to the intensive use of a slow main memory is often denoted as the *von Neumann Bottleneck*. Compilers reduce this penalty through prefetch operations, although all latencies cannot be hidden. Complex, multilevel cache hierarchies implement memories closer to processing units with higher bandwidth (see Fig. 1.1), allowing for much shorter average latencies of memory operations. However, algorithms must take this memory structure into account and more complexity is transferred to the compiler or the programmer's responsibility. Algorithms with irregular memory access patterns such as Breadth-First Search cannot use this complex memory hierarchy to improve their performance.

Parallel algorithms proved to be more challenging to design than sequential ones. It is challenging despite the efforts developed to maintain the common assumption that all data is always available to all parts of a program through memory operations. Processors' memory hierarchies have gained in complexity to maintain performance brought by the cache hierarchy and a coherent view of the distant common shared memory space throughout the execution of a parallel program. This introduces inefficiencies: if one core modifies a value that needs to be kept coherent and that is stored in another core's cache, then both cores' caches must be stalled until all are at least aware of the update that took place. Numerous improvements for efficient cache coherency have been developed; for instance, Grahn and Stenström [50] suggest an adaptive cache coherency mechanism that switches block of a cache between *write invalidate* and *write update* policies, depending on the observed behavior of the application, and so reduce cache misses by 71% and bandwidth requirement by 26%. Stenström et al. [134] acknowledges latencies due to cache misses and invalidation and explores 4 optimization techniques for applications running on a Cache-Coherent Non Uniform Memory Access (CC-NUMA) architecture. Martin et al. [99] concludes that despite its cost, the performance of cache coherency protocols can scale with the number of cores. However, they admit that their analysis does not take into account the cost of on-chip networks necessary to implement cache coherency. The use of network links is expensive in energy and the energy drawn by links grows with their length [119]. Also, the end of Dennard Scaling¹ makes smaller transistors from refined lithography processes to consume relatively more energy. However, the energy spent in networks does not contribute to the processing power of an architecture and therefore, their area and energy consumption must scale to be employed in massively parallel architectures.

Classic networks such as crossbars, buses, hypercubes, fat trees or meshes

¹The Dennard scaling [32] is often cited to denote that the power density of transistors remains constant while their size in new manufactured chips decrease. In other words, the power drawn by a transistor scales with its size.

(a) AMD processor without its fan and its main memory. (b) Wafer of the Intel Haswell architecture, Intel Free Press (<http://www.flickr.com/people/54450095@N05>)).

(c) Schematic view of memory sizes in a 16 GB main memory and Haswell 5960X L3 cache. (d) Schematic view of memory sizes in Haswell 5960X L3 and L2 caches. (e) Schematic view of memory sizes in Haswell 5960X L2 and L1 caches and registers (letter R).

(f) Schematic view of memory bandwidth evaluation (thickness of red lines between memories) in a 8 cores Haswell 5960X (representation for 2 cores).

Figure 1.1: Picture of a typical computer system, processor die and schematic view of a typical memory hierarchy [71, 130].

fail to provide a good cost/latency compromise to implement scalable cache-coherency on-chip networks: crossbars implement $O(1)$ link traversal for communications between any pair in p cores, but it requires (p^2) links. Buses are very efficient for broadcast but their energy consumption $O(p^{3/2})$ [14] is too high, they cannot operate at high frequency and they do not allow several pairs of cores to communicate in parallel [51]. Hypercubes and fat trees provide a scalable latency of $O(\log p)$ but their cost in number of links, $O(p \log p)$, grows too fast. Finally, meshes are cheap ($O(p)$ links), predictable due to uniform links' length [119] but their $O(\sqrt{p})$ latency does not allow fast broadcasting. As cache coherency protocols rely heavily on broadcast [35], effective cache coherency over meshes requires on knowledge about the sharing pattern (producer-consumer for instance) of the memory being shared [23] to limit broadcast operations. Therefore, for the same programming complexity, low-latency cache coherent architectures are costly in silicon area and energy to operate.

Current research investigates on the energy consumption challenge described above by trading latency for throughput to consume less energy. This allows for simpler on-chip networks and the scaling of the number of cores embedded. The challenge posed by these architectures is an ever increasing complexity due to numerous features. Due to the high number of cores, programmers can no longer manage individual threads manually as they would do with legacy libraries such as pthreads. Data accesses must be laid out so that communications between caches, as well as the distance between them, should be reduced to lower energy consumption and communication latencies. In some cases, programmers can no longer assume data to be directly accessible at any time through a simple memory operation. Instead, they must explicitly initiate communications by sending and waiting for messages.

While this message passing approach may seem harder for programmers, a lot of research has been performed on parallel programming based on message passing instead of shared memory. Stream programming is a special form of message passing model that consists in modeling a program as several concurrent tasks, each reading a small subset of input data, process it and forward intermediate results to another processing task while more data is read from input and processed. Kahn [77] as well as Lee and Messerschmitt [90] demonstrate the verifiability of correctness for streaming algorithms, even in real-time context, making streaming algorithms suitable for safety critical systems. Streaming algorithms involve consecutive tasks that may run concurrently using *pipeline parallelism*. If the pipeline of a streaming algorithm is implemented so that only the cores at the beginning and at the end of a pipeline need to operate on the distant main memory and other communications are forwarded within the processor chip directly from core to core (*on-chip pipelining*), the algorithms are much more efficient, as shown e.g. by Keller et al. [81] through a on-chip pipelined implementation of mergesort for the Cell B/E processor. In this context, the use of caches

is much more predictable, which enables the development of fast real-time systems. If tasks of a pipeline are sequential and do not directly share memories, then this approach also waives the need to maintain coherent caches across a multi-core processor as well as complex synchronization strategies; the saving of transistors allows the implementation of more cores into the same chip.

The work described in this thesis takes profit of opportunities brought by many-core architectures to elaborate on active research on stream processing. We investigate challenges to implement fast and energy efficient on-chip streaming algorithms for many-core architectures. An important challenge tackled in this thesis consists in the layout of streaming tasks into processors in order to optimize energy efficiency under throughput constraints.

Thesis outline

Chapter 2 provides the necessary background about many-core architectures and stream processing. Chapter 3 details a thorough benchmarking of the Intel Single Chip Cloud computer (SCC) architecture. Chapter 4 describes on-chip pipelining experiments with the SCC. Chapter 5 describes *Crown Scheduling*, a novel methodology to optimize the energy consumption of task schedules across multi-processors under makespan constraints. Chapter 6 introduces *Drake*, a tool designed to test the quality of static schedules for streaming applications under throughput constraints and compare properties such as energy consumption. In Chapter 7, we give a brief introduction to all software developed to perform the work described in thesis, with more details in appendices. Finally, Chapter 8 gives final remarks and concludes this thesis.

Contributions

The contributions in this thesis include the benchmarking of a many-core architecture, the Intel Single Chip Cloud computer, the design and evaluation of an on-chip pipelined implementation of streaming mergesort, and Crown Scheduling, a novel scheduling technique that optimizes energy of streaming applications under a throughput constraint.

- We scrutinize the on-chip network of the SCC and investigate the behavior of its main memory controllers. In particular, we study the bandwidth to main memory available to cores for read, write and combined read and write operations in linear, strided, mixed linear and strided and random memory addresses access patterns. We find that the SCC cores fail to saturate the memory bandwidth available and that memory controllers are able to efficiently compensate for cache misses for regular memory accesses.

- We use four implementations of a classic mergesort algorithm for the SCC in order to study its behavior: a shared memory algorithm, with enabled or disabled caches and a message passing scheme implemented on shared off-chip memory or using the on-chip network. We observe that the use of the on-chip network to transfer large amounts of data does not yield much time penalty over merging data directly over cached shared memory without transfers.
- We implement a complete sorting algorithm adapted to the SCC, including an on-chip pipelined phase. We find that the on-chip pipelined phase brings a significant acceleration over classic mergesort implementations.
- We formulate *Crown Scheduling*, a novel technique to schedule moldable streaming tasks for many-core architecture under a throughput constraint. We solve the problems of allocation, mapping and frequency scaling of streaming tasks, either separately or all together using an Integer Linear Programming (ILP) formulation and heuristics. We find that the integrated variants require a longer time to execute, but they produce better schedules. Also, our heuristics produce solutions of a quality very close to the ones produced by an ILP solver. We extend the technique to target platforms of arbitrary number of cores through *Crown configuration* and we evaluate the efficiency of several configuration strategies. We investigate further energy saving by scheduling tasks to a reduced set of cores and switch off the unused ones. We adapt Crown Scheduling to take architecture constraints such as voltage and frequency islands so that the solutions computed can be used in practice. We describe *Crown rescaling* to adapt dynamically a schedule computed statically, in case of tasks running slower or faster than expected so that to save even more energy. Finally, we provide a mathematical argument to demonstrate that, for realistic target platform and energy consumption models, our integrated Crown Scheduler produce solutions estimated to make a stream application consume at worst 3.7 times more energy than with an optimal schedule.
- We introduce Schedeval, a tool to evaluate schedules of communicating streaming tasks under throughput constraints for massively parallel architectures and Drake, a C programming framework derived that builds optimized and retargetable streaming applications. We show that the overhead of Drake can be hidden by running several streaming tasks on the same core and that a mergesort implementation based on Drake competes with a specialized implementation for the SCC. We use Drake to demonstrate the energy performance difference between two schedules that are otherwise considered as equivalent by a simple energy model. We show that on Intel Xeon platform, our mergesort implementation for Drake is competitive with other implementations

using state-of-the-art frameworks OpenMP and Intel TBB, both in sorting time and energy consumption.

List of publications

The work described in this thesis has been published in the articles listed below. For each paper, we detail the contribution of each author and we use the first person (*I*) to denote Nicolas Melot, the author of this PhD thesis. We further give the chapter that gives all details about the work achieved in the paper.

Kenan Avdic, Nicolas Melot, Jörg Keller and Christoph Kessler: *Parallel sorting on Intel Single-Chip Cloud computer*. A4MMC, ISCA-2011, San Jose, USA, 2011 [7] (Ch. 3).

Christoph Kessler and I supervised Kenan Avdic, then master student working on his master thesis at Linköping University. We suggested mergesort variants to implement and Kenan Avdic wrote them and managed the experimental work. Jörg Keller participated in interpreting intermediate results and deciding on the next appropriate experimental step. I wrote the final paper, with the help of Kenan Avdic to check the accuracy of the content and Christoph Kessler, and of Jörg Keller for writing style.

Nicolas Melot, Kenan Avdic, Christoph Kessler, Jörg Keller: *Investigation of Main Memory Bandwidth on Intel Single-Chip Cloud computer*. Intel MARC3 Symposium 2011, Ettlingen, Germany, 2011 [104] (Ch. 3).

Christoph Kessler and Jörg Keller suggested the experimentation. I supervised Kenan Avdic to implement and run it and I wrote the paper.

Nicolas Melot, Christoph Kessler, Kenan Avdic, Patrick Cichowski and Jörg Keller: *Engineering parallel sorting for the Intel SCC*. Procedia Computer Science, Vol.9(0), pp.1890–1899, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012 [105] (Ch. 4).

Christoph Kessler and myself imagined the two task placement strategies that co-optimize communication load and load-balancing. I adapted software from Christoph Kessler to read the ILP-computed mappings into our sorting implementation of [8]. Christoph Kessler, Patrick Cichowski, Jörg Keller and myself discussed an alternative third phase for merging; Patrick Cichowski implemented it.

Christoph Kessler, Nicolas Melot, Patrick Eitschberger and Jörg Keller: *Crown Scheduling: Energy-Efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Malleable Streaming Tasks*. 23th

International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS-2013), Karlsruhe, Germany, Sept. 9-11, 2013 [83] (Ch. 5).

Christoph Kessler imagined the recursive decomposition scheme of Crown Scheduling and suggested an ILP model to implement it in phase separated and integrated manners. Patrick Eitschberger wrote the task set generator that we used in the experimental section and helped me writing the software we used to run the separated phases together. I fixed details on all ILP models suggested by Christoph Kessler and managed all experiments we describe. Christoph Kessler and Jörg Keller wrote the article.

Nicolas Melot, Christoph Kessler, Jörg Keller and Patrick Eitschberger: *Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Many-Core Systems*. ACM Transactions on Architecture and Code Optimization, Vol.11, pp.62:1–62:24, 2015 [108] (Ch. 5).

We reused the basic technique and experimental approach of [83]. In this work, I designed the optimal allocation, LTLG and Height phase-separated heuristics for Crown Scheduling as well as the binary search and simulated annealing integrated heuristics and their time complexity analysis. Christoph Kessler suggested the additional concrete task set we used in our extended experimental section and I imported the task set derived from the Streamit benchmark suite. I designed the earliest version of Mimer [106] in order to manage experiments and analyze the data produced. I wrote the final paper with the help of Christoph Kessler, Patrick Eitschberger and Jörg Keller, mainly for proof-reading.

Nicolas Melot, Johan Janzén and Christoph Kessler: *Mimer and Schedeval: Tools for Comparing Static Schedulers for Streaming Applications on Manycore Architectures*. 8th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), 2015, Beijing, China, Sept. 1st, 2015 [106] (Ch. 6).

I supervised Johan Janzén, then master student at Linköping University working on his master thesis. Johan Janzén adapted into Schedeval a more general implementation of on-chip pipelined stream programs for the SCC than in [8], that I initially developed, to add frequency scaling and power measurement for the SCC and that resulted in *Schedeval* (I later refactored *Schedeval* into *Drake* to decouple

platform-specific and streaming framework routines). Johan Janzén used *Mimer* to conduct experiments under my supervision. Christoph Kessler helped discussing the results. Johan Janzén wrote the article part on Schedeval and the experimental section and I wrote the remaining parts with writing advice from Christoph Kessler.

Nicolas Melot, Christoph Kessler and Jörg Keller: *Improving Energy-Efficiency of Static Schedules by Core Consolidation and Switching Off Unused Cores*. Proceedings of International Conference on Parallel Computing (ParCo), Edinburgh, UK, Sept., 2015 [107] (Ch. 5).

I received the suggestion of Core Consolidation in a visit to Denis Trystram’s group at INRIA in Grenoble. Christoph Kessler, Jörg keller and myself discussed the ILP model details. I implemented it and used *Mimer* [106] to manage experiment. Jörg Keller wrote the article.

Nicolas Melot and Christoph Kessler: *Voltage Island-Aware Energy-Efficient Scheduling of Parallel Streaming Tasks on Many-Core Processors*. Accepted for presentation at the 9th Nordic Workshop on Multi-Core Computing (MCC2016), Trondheim, Norway, 2016 (Ch. 5) and submitted at 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 6th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM), Stockholm, Sweden, 2017 (Ch. 5).

I suggested the experiment and Christoph Kessler and myself discussed the details of the extension of Crown Scheduling Integrated ILP model. I ran the experiment with *Mimer* [106] and I wrote the article with writing advice from Christoph Kessler.

Nicolas Melot, Christoph Kessler, Patrick Eitschberger and Jörg Keller: *Co-optimizing Core Allocation, Mapping and DVFS in Streaming Programs with Moldable Tasks for Energy Efficient Execution on Manycore Architectures*. Submitted at 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS), Orlando, Florida USA, 2017 (Ch. 5).

Christoph Kessler, Jörg Keller and myself discussed the details of the restriction free integrated ILP scheduler. Patrick Eitschberger and I implemented the model. I designed the proof that demonstrates the approximation ratio of optimal Crown Scheduling. I conducted experiments with *Mimer* [106] and I wrote the article with help from Patrick Eitschberger for the section describing our ILP formulation for an unrestricted scheduler.

Chapter 2

Background

2.1 Introduction

This introductory chapter gives elements to show why programming for performances is a challenging activity and introduces *Stream programming* as an approach to ease the design of high-performance systems. In this chapter, we give the necessary background to the rest of this thesis. We begin with an introduction to streaming computation in Sec. 2.2, then we review a few relevant processor architectures commonly used in the high-performance research community in Sec. 2.3. The chapter ends by a brief introduction to the main programming languages designed for high performance that are available today (Sec. 2.4). Finally, Sec 2.5 concludes this chapter.

2.2 Streaming Computation

Streaming computation can be described as the computation and output of data depending on input data that the computation system cannot entirely capture and store beforehand. Instead, the system captures only a small subset of this input data at a time to perform computation upon it and replaces it immediately by more input data. Many industrial applications naturally involve this situation: for example, scientific computing and big data often involve the processing of an amount of data far greater than the memory of computers used to process it. Similarly, the duration of a telephonic conversation is unknown until the user hangs the phone; a telephony system must sample a user's voice, process it and transmit it immediately so that the user at the other end of the communication channel can interact and form a two-sided conversation. In this situation, input data can be seen as an infinite size input until the conversation effectively ends. Concrete examples range from telephony and multimedia encoding and decoding, to the processing of large data such as from social networks and scientific facilities such as the CERN's Large Hadron Collider. When

the high-speed memory of a computer cannot hold the entire input data set, streaming computation is a preferred method to accelerate the process by limiting accesses to the slower memories. Because of these many advantages, streaming is a well researched area whose roots date as early as in the 1960 decade [135].

This section attempts to give a brief overview of stream computing. Section 2.2.1 gives a brief formal introduction to the semantics of stream processing.

2.2.1 Semantics

Stephens [135] observes in a survey of stream processing, that while lots of attention is given to systems to compute on streams, little care is given to study streaming as an abstract computing machine. The survey stresses the difference between *Stream Processing Systems* (SPS) and *Stream Transformers* (ST).

SPS are systems composed of *modules* (otherwise called *agents*, *filters*, *nodes*, *computing stations* or *processes*) that run in parallel and whose only means of communication with each other is a set of *channels* between them. *Sources* input data to the system and *sinks* output computed data from the system. Channels convey *streams* between modules and a stream can be defined as a finite or infinite ordered list of elements $a_1, a_2, \dots, a_n$ where all elements, including the empty element Λ (noted $\emptyset$ by Broy and Dendorfer [16] and $\langle \rangle$ by Stephens [135]), belong to a set A . Elements can be assigned to a discrete time value from a time set $T = \mathbb{N} = \{0, 1, 2, 3, \dots\}$, in a function $a : T \rightarrow A$. Kahn [77] denotes such a list or function as A^ω and Broy and Dendorfer [16] decompose it further into $A^\omega = A^* \cup A^\infty$ where A^* is the set of *finite* sequences of elements in A , and A^∞ is the set of *infinite* sequences over A . It is interesting to observe that the first references to streams were intended to model histories of loop variables and employed in the verification of operating systems [16, 135]. Consequently, channels and streams are sometimes also referred to as *histories*.

Stream Transformers (ST) [135] define operations on streams. Stephens [135] describes them as an abstract system taking n input streams and produces m output streams ($n, m \geq 1$), that could be represented by the function

$$\Phi : [T \rightarrow A]^n \rightarrow [T \rightarrow A]^m$$

Stephens [135] stresses the differences between a ST and a SPS, where the latter is a set of communicating processes that implement a ST, and thus is a special case of ST. However, it is not clear why a ST should be restricted to output streams over the same set A of its input streams. Primitive operations over streams remind of basic functional programming constructs, such as:

- F: to any sequence $x \in D^\omega$, associate its leftmost (first) element.

- R: to any sequence $x \in D^\omega$, the sequence at the right of (after) its leftmost element.
- A: to an element $d \in D$ and a sequence $x \in D^\omega$, associate the sequence starting with d followed by x .

We refer to the survey by Stephens [135] and to Broy and Dendorfer [16] for more primitive operations over streams. An important property of a ST is that it is a function that maps an input stream to an output stream; in other words, every element in the output stream is a function of *all* elements of input streams consumed so far instead of just one corresponding element in each input stream. Thies et al. [138] gives 6 properties characteristic of a streaming application: large streams of data, independent stream filters, a stable computational pattern, occasional modifications of the stream structure, occasional out-of-stream communications and high performance.

Stephens [135] classifies SPS along three dimensions of two elements each, that defines if a SPS is synchronous (S) or asynchronous (A), deterministic (D) or non-deterministic (N) and if its channels are unidirectional (U) or bidirectional (B). A synchronous system controls the rates in which its nodes fire and consume tokens transported on channels. A deterministic SPS denotes a SPS whose nodes run a deterministic working function, that is, the behavior of the process is determined by the function's code only. Finally, the channels of a unidirectional SPS can only carry data tokens in one direction. Communications in the opposite direction must be performed through other channels. For example, a synchronous, deterministic SPS with unidirectional channels is denoted a SDU-SPS. Stephens [135] further divides the dataflow model of computation into *data driven* where modules start to compute data upon data availability (eager evaluation), and *demand driven* where modules request data on the input line as they need to compute output (lazy evaluation).

Thies et al. [138] argues that general purpose languages such as C/C++ are oblivious to challenges brought by recent parallel architectures with independent, loosely synchronized clocks, cores communicating with each other and where data locality is a more important problem than for sequential machines. Indeed, Stephens [135] claims that dataflow research has little impact on the traditional “von Neumann computing” because practical implementations on conventional architectures lead to inefficiencies. Specialized architectures for SPS have been developed to avoid the von Neumann bottleneck, but Stephens [135] further invokes the “lack of generally accepted clear and straightforward semantics” to further explain the relatively low impact of the dataflow approach over von Neumann's.

2.2.2 Properties of Computation over Streams

As Stephens [135] explains, studies over streaming takes root in the study of histories of loop variables. Kahn [77] studies the dataflow approach and

proposes streaming semantics aiming at simplifying the analysis of such programs to prove some of its properties. The paper gives a simple example of a streaming program for which he proves that all its processes run forever, a particular process produces an alternating sequence of 0's and 1's forever and that if any process stops, then all processes stop. However, Kahn [77] makes a number of assumptions in his analysis:

1. Channels are the only means of communication between two processes.
2. Channels convey information in a finite time.
3. No two processes can send information through the same channel.
4. Processes can hold an infinite amount of information before consuming it.
5. At any given time, a process is either computing or waiting for information on *one* of its inputs.
6. Nothing can prevent a process from writing on any of its output channels.
7. All processes implement a *continuous* mapping of their input stream elements to their output streams.

Because of Assumption 7, a complete dataflow program can be seen as the aggregation of all functions of its processes. Hence the dataflow program implements a larger but still continuous function. This allows for a top-down design of a data-flow system, where the overall function can be later split and distributed to processes, without affecting the global system. Kahn [77] explicitly allows processes to run in parallel and mentions that “this model exhibits some form of parallelism”. A sequential machine can simulate such a model, provided that the scheduler guarantees that all processes eventually receive computing time if required.

The major weakness of Kahn networks lies in assumption 4, which states that processes can hold an infinite amount of information before it is consumed. No real architectures can provide such a guarantee. If any communication buffer is full, then either the process writing on the corresponding channel is stalled (contradicting assumption 6), or information is lost. Lee and Messerschmitt [90] propose Synchronous Data Flow (SDF) to annotate arcs of data flow graphs with firing and consuming rates (called *sample rates*) to process infinite streams. An arc is annotated with a firing rate r if the producer node connected to this arc produces r tokens of information each time it is run. In SDF, all nodes cannot run in any arbitrary order, therefore some form of scheduling is required. Lee and Messerschmitt [90] propose to compute static schedules, arguing that a dynamic strategy is likely to provoke overhead. All SDF graphs cannot be scheduled statically; in particular, Lee and Messerschmitt [90] give an example of a simple graph

with an *inconsistent sample rate*, that requires at least one unbounded communication buffer; clearly, no real architecture can hold such a buffer. Lee and Messerschmitt [90] give examples of graphs with consistent sample rates that do not admit a valid schedule. In this case, Lee and Messerschmitt [90] argue that a class of scheduling algorithms are proved to find a schedule for any graph that admits a valid schedule, and fail if such graph does not admit any valid schedule; they provide polynomial scheduling algorithm for uniprocessors that satisfies this property. Falk et al. [39] proposes the opposite approach: they dynamically adjust the size of communication buffers so that no deadlock can happen due to a full memory.

2.3 Processor Architectures Overview

This section reviews several many-core architectures developed in the recent years and often used in many-core research. Most of these architectures aim at scalability, that is it should be possible to increase the number of cores and obtain a proportional increase in performance; most of them include several dozens of cores. As discussed in the introduction, this often implies a relaxed cache coherency mechanism. They also aim at an increase of computation speed (floating points operations per seconds) and energy efficiency (floating point operations per watt).

2.3.1 Platform model

A model provides a common language to describe existing and non-existing execution platforms. It facilitates comparisons between platforms and allows algorithms to reason about them. Such a model should capture various platform characteristics. These include the number of cores available or the degree of dependence between processing cores. For example, the cores within a GPU are very dependent on each other because of performance penalties related to branches in the code they execute collectively. Other important characteristics include network topologies as well as their links' bandwidth and latency, cores' micro-architecture or DVFS capabilities. The modeling of computer platforms can take shape in an idealized, easy to reason with but unrealistic model such as PRAM [80], more realistic but more difficult model to reason with such as LogP or LogGP and very realistic and difficult to handle models such as PDL [124] and XPDL [84]. While PRAM and LogP are used to analyze parallel execution time of algorithms, PDL and XPDL provide a very detailed model designed as part of a retargetable optimization framework. XPDL includes CPUs, cores and their instruction set, main memory, GPUs and accelerators with their own memory and software available.

Using a model allows performance analysis of algorithms designed for processing architectures. We focus on analysis on energy usage. Analytical energy models decompose power consumption in static and dynamic

power [110]. Dynamic power accounts for the power spent switching logic gates, and static power models the power lost by gates through power leakage, regardless of their activity. Mudge [110] models the total power consumption as

$$P = A \cdot C \cdot V^2 \cdot f + \tau \cdot A \cdot V \cdot I_{short} \cdot f + V \cdot I_{leak}$$

where A, C, τ, I_{short} and I_{leak} are architecture-dependent constants, f is the frequency the processor runs at in Hertz and V is the power supply in Volts. The two first terms model *dynamic power*: the first term models the power consumed to switch a gate and the second describes the energy loss when switching momentarily short-circuits the supply voltage to the ground. The last term is *static power*, the power loss of gates through leakage regardless of their state. As Mudge [110] states, the maximal frequency is roughly proportional to voltage, hence the equation above can be reduced to

$$P = O((1 - \zeta)f^3 + \zeta \cdot f) \quad (2.1)$$

where the dynamic power is $O(f^3)$ and the static power is f and $\zeta \in [0; 1]$ models the part of static and dynamic power in the total energy. Mudge [110] claims that dynamic power accounts for most of total power consumption. Although Borkar [15] acknowledges that most power consumption was due to dynamic power until the publication time of writing his article, the static power part began to grow faster than the dynamic power as the size of transistors continued to decrease. Equation 2.1 demonstrates the importance of the frequency in the total power consumption. In principle, a small decrease of frequency results in high energy savings. However, Snowdon et al. [131] shows that energy savings through frequency scaling are dependent on applications and that a lower frequency does not always translate to a lower power consumption. For example, the execution time of memory intensive applications might not be affected much by their frequency as they spend most time waiting for memory operations to complete.

Frequency and voltage scaling is restricted to numerous constraints that make any switching decision challenging. It must be performed on voltage or frequency *islands*, i.e, groups of cores whose voltage or frequency, respectively, cannot differ from one another. This constraint is necessary because portions of a chip running at a different voltage or frequency must be isolated from each other; this consumes additional silicon to implement the voltage regulator necessary for each island, and complicates floor planning and clock propagation problems as the clock must be isolated within an island. A voltage island often includes several frequency islands; see the description of the SCC in Sec. 2.3.2 for an example. As shown on Fig. 2.1a, switching frequency or voltage have costs in delay and energy, and switching frequency takes shorter time than switching voltage [101]. This complicates the problem as switching to a voltage or frequency level may cost more switching energy than saving energy through a lower voltage and/or frequency. Also,

(a) Switching delays hinders energy saving due to voltage or frequency scaling. Light gray areas denote transition delays trying to follow the schedule. Dark gray areas show periods where the running frequency matches the frequency scheduled. Too short periods of expected low frequency may not allow actual energy savings or may make impossible any actual switching.

(b) Example of frequency over time and maximal frequency given the voltage level at the same time, and voltage over time and the minimal voltage given the frequency at the same time.

Figure 2.1: Constraints in voltage and frequency scaling.

voltage and frequency are dependent on each other: the maximal frequency is linear in voltage to ensure reliable gate switching [110] and reducing voltage as much as possible to run at a given frequency can yield large energy saving (Eq. 2.1). As shown in Fig. 2.1b, the voltage must be increased before frequency, if necessary. On the contrary, the frequency must be lowered before the voltage is decreased.

2.3.2 Single Chip Cloud computer (SCC)

The SCC is the second generation processor issued from the Intel Tera-scale research program [61, 66, 67, 68, 100]. It provides 48 independent IA32 cores, organized in 24 tiles. Figure 2.2a provides a global view of the organization of the chip. Each tile embeds a pair of cores, and tiles are linked together through a 6×4 on-chip mesh network. A tile is represented in Fig. 2.2b: each tile comprises two cores with their cache and a message

Figure 2.2: Organization of SCC.

passing buffer (MPB) of 16 KiB (8 KiB for each core); it supports core-to-core communications through memory operations on local and distant MPBs thanks to the on-chip network. The cores are IA32 (P54C [64]) cores running at 533 or 800 MHz; each of them has its own individual L1 and L2 caches of size 32 KiB (16 KiB code + 16 KiB data) and 256 KiB, respectively. Since these cores have been designed before Intel introduced MMX, they provide no SIMD instructions. The mesh network can work at up to 2 GHz. Each of its links is 16 bytes wide and exhibits a 4 network cycles latency, including the routing and conveying activity. In the default setting, a group of six tiles share an off-chip DDR3 memory controller that the cores access through the mesh network. The overall system admits a maximum of 64 GiB of main memory accessible through 4 DDR3 memory controllers evenly distributed around the mesh. Each core is attributed a private domain in this main memory whose size depends on the total memory available (682 MiB in the system used here). Furthermore, a common small part of the main memory (32 MiB) is shared between all cores; this small amount of shared memory may be increased to several hundred megabytes. Note that private memory is cached on cores' L2 cache but cache support for shared memory is not activated by default in Intel framework RCCE. When activated, no automatic management for L2 cache coherency among all cores is offered to the programmer. This functionality must be provided through a software implementation.

The SCC can be programmed in two ways: a baremetal version for OS development, and using Linux. In the latter setting, the cores run an individual Linux kernel on top of which any Linux program can be loaded. Also, Intel provides the RCCE library which contains MPI-like routines to synchronize cores and allows them to communicate data to each other, as well as the management of voltage and frequency scaling. Because of its architecture consisting of a network that links all cores and each core running a Linux kernel, programming on the SCC is very similar to programming parallel algorithms for clusters (for instance) on top of a MPI library.

2.3.3 Intel Knights Corner

The first Xeon Phi generation *Knights Corner* [70] is a PCI-express accelerator with up to 61 Intel-64 bit independent cores based on the P54C [64], compatible with 32-bit programs. The cores do not provide SSE or AVX vector instructions, but another 512 bit wide vector instruction set. Additionally, all cores provide support for 4 hardware threads, aggregating up to 244 hardware threads through the chip. Each core has an individual 64 KiB L1 cache (32 KiB data + 32 KiB code) and 256 KiB L2 cache. Coherency between cores is maintained through a 8 rings network between caches. The chip embeds 8 DDR5 memory controllers with two channels per controllers, providing a maximum bandwidth of 352 GiB per second.

The Knights Corner is designed to facilitate the porting of applications to exploit its parallel resources. Because there is a cache coherency protocol, it behaves like a Symmetric Multi-Processing machine. Each core runs a lightweight version of Linux, which provides a large programming flexibility and a familiar execution environment. The chip allows for dynamic voltage and frequency scaling, and for several sleep states for individual cores and for the whole board. Voltage and frequency scaling as well as the selection of an idle state are not directly accessible to the programmer; it is only possible to select and parametrize a scaling algorithm.

2.3.4 Intel Knights Landing

Intel released *Knights Landing* [132], the second generation of Xeon Phi processor, in mid 2016. It provides numerous changes over the first generation *Knights Corner*, including the cores as well as the on-chip interconnect and memories. The Knights Landing chip includes 72 out-of-order cores based on Silvermont architecture [72] instead of P54C. Each core embeds 4 hardware threads and 2 Vector Processing Units (VPU) [102]. Each VPU embeds 512 bits AVX, SSE, AVX1, AVX2 and EMU instructions. All cores are grouped in 36 tiles of 2 cores and 4 VPUs each, as well as a shared 1MB L2 cache kept consistent across the chip and a *Caching Home Agent* (CHA), i.e., a cache coherent NUMA directory. Cache coherency takes place with a MESIF protocol (F stands for Forward), using each tile's CHA as a distributed tag directory as well as the on-chip 2D mesh network that links tiles to forward data between caches, using a deterministic XY routing scheme. The chip includes 2 DDR4 controllers for a total of 6 channels and up to 384 GB at 90 GB/sec. Finally, the chip includes two 16x and one 4x PCI-express Gen3, one 4-lanes DMI controller and two 25 GB/sec *Omni-Path* on-die ports designed to link several Knight Landing packages together to a cluster, for example through Infiniband.

One major improvement brought to Knights Landing is the Multi-Channel DRAM (MCDRAM) integrated in the chip's package, but not in the chip, via 3D stacking of silicon dies. It offers 16GB with 400GB/sec bandwidth, accessible through dedicated controllers all around the chip and that can

be used in three modes. *Cache mode* provides a package-level shared L3 cache with 64 bits lines that covers the whole off-package main memory. The *Flat mode* gives the programmer full explicit control of the MCDRAM memory with Non-Uniform Memory Access (NUMA) in the address space as off-package DDR memory. The programmer can use a dedicated high-level allocation routine `hbw_malloc(size)` to use it explicitly, taking the on-chip mesh network into account to optimize accesses. Finally, the *Hybrid mode* partitions MCDRAM into 4 or 8 GB L3 cache as in Cache mode and 8 or 12GB memory as in Flat mode. Thanks to the cache mode, the Knights Landing processor offers binary compatibility with Xeon and can even stand as a system's main processor initialized by UEFI or a Bios to load and boot an operating system. As any Xeon-compatible operating system can run on a Knights Landing-based system, it can be programmed with any language or software environment. However, frequency scaling is only possible at the chip level, making impossible the fine tuning of voltage and frequency scaling techniques described in this thesis.

2.3.5 Sony-Toshiba-IBM Cell

The Cell Broadband Engine [2, 76] was conjointly designed by Sony, Toshiba and IBM for media processing and scientific computing and released in 2005. It is composed of 1 *Power Processor Element* (PPE) and 8 *Synergistic Processor Elements* (SPE). The PPE is a 64-bit Power PC core with 2 hardware threads issuing 2 instructions per cycle. It has a private 32 KiB L1 cache (16 KiB code + 16 KiB data) and a 512 KiB L2 cache and provides a 128-bit wide vector instruction set and can issue 2 instructions per cycle. The 8 SPEs are designed to accelerate multimedia processing. All its instructions are 128-bit wide SIMD instructions. SPEs have no cache but a 256 KiB explicit *local store* memory. A 4 rings on-chip network links the PPE, SPEs and the memory controller together and provide a bandwidth of 307.2 GiB per second. The Cell BE implements two channels to a Rambus XDR memory systems of bandwidth 12.8 GiB per second each, or a total of 25 GiB per second memory bandwidth.

IBM provides a software development kit with C and C++ compilers and libraries for both the PPE and the SPEs. The compilers do not automatically determine if some code must run on the PPE or the SPE, making it particularly challenging to program for this processor.

2.3.6 Tilera Tile, Tile-Gx and TilePro

Tilera distributes commercial processors based on TileGx and TilePro Tilera architectures. Figure 2.3 gives an overview of an older Tile64 architecture, but both TileGx and TilePro have a similar structure. The Tile-Gx variant includes between 9 and 72 64-bit cores with SIMD extensions. Each core has an individual L1 cache of 32 KiB for data and 32 KiB for code, an individual 256 KiB L2 cache and a coherent 18 MiB L3 cache. Cores are

Figure 2.3: Schematic view of Tiler Tile64 architecture and one of its tiles

linked by 5 independent on-chip networks aggregating 110 TB per second of bandwidth and a latency of 1 clock cycle per hop. The Tile-Gx includes 4 DDR3 memory controllers that can address a total of 1 TB. No information is given on any voltage and frequency scaling capability, or any sleeping state for cores. The TilePro variant embeds 36 or 64 independent VLIW 32-bit cores. The cores are connected through an on-chip network of 12 TB per second bandwidth. The chip embeds 2.8 MB of on-chip cache.

In order to make their architecture scalable, Tiler implements a *Distributed Shared Cache* that does not implement a complex cache coherency scheme. If two cores manipulate the same memory address, then only one of their associated cache keeps the value and both cores read and write to this copy. The physical distance between both cores affects the overall performance. Choi et al. [22] investigate this issue and claim to improve 5 out of the 11 benchmarks in the SPLASH benchmark suite [142], by 32% to 77%.

The TileGx architecture allows programming in C/C++ and TilePro additionally supports Linux 2.6 to run separately on individual tiles or as SMP on multiple tiles.

2.3.7 Adapteva Epiphany

The Epiphany architecture [1] is a coprocessor that standard microprocessors can offload their computation to. Adapteva claims that it can scale to up to 4096 cores organized in individual tiles. Each tile embeds an individual

²CC-BY-SA MovGP0 <http://en.wikipedia.org/wiki/File:Tile64.svg>

²CC-BY-SA MovGP0 <http://commons.wikimedia.org/wiki/File:Tile64-Tile.svg>

RISC (Reduced Instruction Set Computer) 32-bit 1 GHz processor optimized for floating-points operations. Each core is paired with a local memory of 32 KiB (although the web page mentions up to 128 KiB per core). The local memories are not caches and are not managed automatically by the hardware; instead, local memories are part of the memory address space and memory operations to these memories must be performed explicitly. On-chip memories can be used to implement core-to-core communications.

Each tile is connected to the on-chip network with a 4 directions (North, East, South and West) router. The network is designed to allow the direct composition of an Epiphany IP with other components of a chip, such as a host core, a memory or IO controller or even another instance of epiphany architecture. Communication is conveyed through the *eMesh*, a on-chip network composed of 3 mesh interconnects called *cMesh*, *rMesh* and *xMesh*. The *cMesh* interconnect conveys on-chip write operations with a maximal throughput of 8 bytes per cycle, that is 0.5 TiB per second at 1GHz. *rMesh* is used for all read operations within and outside the chip and offers 1 read operation per 8 clock cycles. Finally, *xMesh* carries write operations outside the chip (to main, off-chip memory, host CPU, I/O or other composed Epiphany IP) with a bandwidth of 8 GiB per second. The Epiphany documentation hints that the *cMesh* network has lower latency and higher bandwidth, and therefore tasks communicating a lot with each other should be placed on the same chip. Routers guide data packets in a XY fashion with a latency of 1.5 cycles per hop. Epiphany implements hardware-aided multicast communication operations. The Epiphany can be programmed in C/C++ and allows approaches such as Single Instruction Multiple Data (SIMD), Simple Program Multiple Data (SPMD), Multiple Instructions Multiple Data (MIMD), shared memory multithreading, message passing and several variants of dataflow programming. Consult the paper by Olofsson et al. [112] for more details about Epiphany.

2.3.8 Kalray MPPA

Kalray’s Multi-Purpose Processor Architecture (MPPA [78]) implements from 64 to 1024 cores. The MPPA-256 variant is structured in 16 clusters of 16 identical cores each. The cores implement a VLIW instruction set with a single, double precision floating point unit and an individual L1 instruction and data cache. Each cluster provides a shared memory island, and a DMA (Direct Memory Access) controller as well as Dynamic Voltage and Frequency Scaling (DVFS) and DPS (Dynamic Power Switch). Each cluster is managed by an additional and individual core with its own FPU and MMU. Clusters are linked together with a 2D torus on-chip network with a bandwidth of 3.2 GiB per second between two adjacent clusters and a “low and predictable latency”. The chip provides 2 DDR3 64-bit memory controllers of up to 12.8 GiB per second bandwidth. The MPPA can be programmed using C, C++ or Fortran and provides a dataflow programming

framework.

2.3.9 Intel Xeon

There is a wide variety of Xeon processors commercialized by Intel and commonly employed in clusters. In contrast to many processors reviewed in this section, Xeon processors embed a few (typically no more than 16), but powerful cores. Cores implement elaborate instruction sets, including 512 bits wide SIMD instructions and atomic instructions such as Test-and-Set or Compare-and-Swap. Each core has typically a private 64 KiB L1 cache (32 KiB data + 32 KiB code) and a 256 KiB L2 cache, although some models share L2 caches among a subset of the cores. Many Xeon chips embed a large L3 cache (up to 45 MiB) shared among all cores.

All Xeon processors maintain a coherent view of the shared memory with the MESI (Modified, Exclusive, Shared, Invalid) cache coherency protocol. Newest versions of Xeon cores also provide Quick Path Interconnect (QPI [65]) that aims at having caches to fetch up-to-date shared data directly from the cache holding the newest value instead of the main memory.

2.3.10 Other Parallel Architectures

Other architectures for low power, fast computing are being developed with radically different strategies to enhance programmability for performance. Among these are Replica [56] and XMT which emulate a PRAM architecture, providing many execution threads and uniform, constant and predictable latencies for read and write memory accesses. For instance, Replica provides 4, 16 or 64 VLIW cores linked by a 2D on-chip network. The chip can run in 2 modes: the *PRAM* mode enables 512 threads per core and unit-cost memory accesses (in time units) and *NUMA* mode, where the cost depends on the distance between cores. There is a high time cost to switch from a mode to another. The PRAM mode performs well on algorithms with an irregular access to memory, but does not perform well with regular accesses compared to other architectures. The Replica can be programmed with C with keyword extensions to declare symbols shared between cores and switch from a mode to another.

General Purpose Graphic Processing Units (GPGPU or simply “GPU”) are very commonly used for high performance multimedia computing and more recently for scientific computing. GPUs are typically packaged in an accelerator board, they have their own main memory and they cannot access the system’s main memory. A master core pushes the data to be processed onto the GPU board and triggers the execution of a kernel on GPU, then it fetches the data processed; this data transfer is slow and can typically not be issued at the GPU’s initiative. A typical GPU embeds several hundreds of tiny cores (192 cores and 2048 threads on Kepler GK110) running an identical program in strictly synchronized steps, on different data (SIMD). If the program includes a branch (if-then-else construct) then cores that do

not execute a code block must wait for other cores to finish running it before all cores can run in parallel the rest of the program.

2.4 Programming for High Performance on Many-Core Architectures

This section briefly reviews most common ways to program many-core architectures for high performance as well as challenges. Each of the approaches and languages described here represent a commonly used or researched programming approach, but there are other industrial languages to implement similar features.

2.4.1 Pthreads

Pthreads is a collection of C routines to create and schedule new threads in the system userspace. With pthreads, the programmer must manage almost everything including thread spawning, termination and various forms of synchronization. Programming with pthreads is difficult, error-prone and is very invasive in the design of an implementation. However, it provides a fine grain control on the parallel execution of a program.

2.4.2 OpenMP

OpenMP [113] extends regular C or C++ with parallelization-related annotations. These annotations make easy the parallelization of loops or the synchronization of shared variables. In contrast to pthreads, it is much less invasive in the source code and is therefore preferred to port legacy applications for parallelism. Furthermore, the compiler can use annotations to take decisions automatically and generate complex strategies to manage threads it spawns to implement load-balancing for example. However, OpenMP assumes a shared memory architecture, which limits its scope to SMP processors. Furthermore, a deep knowledge of the target architecture is still required by the programmer to fine-tune a program for performance. This limits the performance portability.

2.4.3 CUDA

Cuda extends C++ to allow the programming of nVidia CUDA-enabled GP-GPU. CUDA's additional constructs allow a programmer to write programs executed by a GPU (a kernel), send and receive data to and from the GPU as well as starting a kernel and waiting for it to terminate. nVidia provides a specialized compiler able to produce binaries from a mixed CPU/GPU source code. A CUDA kernel translates into a sequence of SIMT instructions run by the GPU.

2.4.4 OpenCL

OpenCL is a portable C++ extension to compete with CUDA and provides a programming language for non-nVidia GPGPUs as well as other parallel architectures such as a multicore Xeon processor. In contrast with CUDA, OpenCL can exploit simultaneously both programmable GPUs and CPUs.

2.4.5 Streamit

Thies et al. [138] explores computing on data flows for high performance. They argue that classic general purpose programming languages are not well suited to parallel architectures. These languages provide a common abstraction for single processor systems, but they fail to model interactions between several processors, in particular the data flow between computation nodes. Thies et al. [138] propose *Streamit*, a programming language where data flows between computation nodes form a central concept. In addition to the models by Kahn [77] and Lee and Messerschmitt [90], they include “high performance expectations” as one salient characteristic of computing over streams.

Streamit provides 3 basic blocks to build a streaming application (Fig. 2.4): *filter*, *pipeline* and *split-join*. The basic unit of computation is the *filter* (Fig. 2.4a). Its program is defined in its function *work*, that is its finest-grain execution step in the steady state. Filters implement a deterministic work function that consumes and fires data at a fixed rate, triggered by the arrival of new data to compute. Filters cannot read from any of its output links or write to its input links. The function *work* takes its input from *pop* operations on FIFO queues, and write its outputs with *push* operations to FIFO queues. *pop* removes an element from the end of a channel, *peek(i)* returns a copy of the i^{th} element from the end of the channel. *push* adds an element to the beginning of a channel. A *pipeline* (Fig. 2.4b) builds a linear sequence of streams between an ordered collection of filters or other pipelines. The rates and types of channels are implicitly deduced from what the filters declare. A *split-join* (Fig. 2.4c) allows the building of loops in the streaming program. It is composed of a *body stream*, a *loop stream*, a *joiner*, a *splitter* as well as an *init path*. The body stream is the main part of the loop and its flow goes frontward. The loop stream links the end of the body stream to its start through one or several filters. The splitter takes the data produced by the body stream and distributes it between the loop output and the loop stream. A *duplicate* splitter duplicates all data on its input and forwards all copies to each of its consumer streams. A *round robin* splitter distributes an element from its input stream to one of its consumer streams, the next element to the next consumer stream and so on. Consumer streams can be weighted to bias the distribution of data to consumer streams. Finally, a *null* splitter has no input stream and is valid only if all its consumer streams begin with a filter with no input channel. Similarly, the joiner feeds the body stream with data from both the loop input stream

Figure 2.4: The three basic blocks of a Streamit program

and the loop stream. A joiner can be *round robin* like a splitter, or *null* if all its producer streams have no output channel. The init path provides initial values to substitute with the loop stream until it begins to produce any data.

All these basic blocks can be used alone or composed either with another block or as part of another block. A block can be used alone only if it consumes no element and produces no element. Instead, input data is read from source filters and written to sink filters. Streamit proposes other features such as low-latency communication of small control messages between filters to reconfigure them. Instead of using regular channels, Thies et al. [138] argues that dedicated channels for controls allow the compiler to run optimizations taking their specific characteristics into account. Messages can be sent from filter to filter (unicast) or from a filter to many or all other filters (multicast). For the latter, Streamit proposes *portals* that forward messages to any filter having subscribed to the portal. Additionally, Streamit allows filters to trigger the dynamic reinitialization of basic blocks. The composite feature of a basic block makes easy the reinitialization of a subset of a streaming application. Finally, the language allows the definition of latency constraints between two filters.

Streamit exposes to the programmer a well-defined structure for writing streaming applications. It is restrictive but Thies et al. [138] argues that it allows for better compiler analysis and optimization. They argue that other stream programming solutions allow for more programming flexibility but, as for GOTO statements in general purpose languages, this flexibility impedes the compiler analysis capabilities, hindering its capacity to produce efficient code. A Streamit program implements a data-driven SDU-SPS.

2.4.6 CAL Actor Language

The CAL Actor Language [38] (CAL, a recursive acronym) structures a parallel program as a set of independent actors sending communication through channels, following the dataflow approach described in Sec. 2.2. A CAL actor contains one or several actions to run upon data arrival on one or several of its input channels. If several actions can be performed upon data arrival, the actor can omit the rule governing which action to run; in this case, an action is chosen after an undefined rule. A CAL actor can store internal variables whose values persist across the actor's executions, therefore implementing a stateful actor. There is no hierarchy in CAL actor composition to structure a streaming application as for Streamit. Channels must have exactly one producer actor and one consumer actor and actors cannot write to their input channels or read from their output channel. A CAL streaming application is therefore a data-driven ANU-SPS.

2.5 Conclusion

This chapter covers 3 dimensions of stream programming for performance. We show that stream programming is a well researched topic dating from the early 1960, and whose advantages cover more than the mere high-performance. It was initially investigated to explore the correctness of a program for safety-critical systems and this property is still in use today in industries. Stream programming enables *pipeline parallelism*, a property initially seen as a side-effect that makes it suitable for high-performance computing. Massively parallel architectures that emerged in the recent years implement a larger number of simpler cores than conventional general-purpose multi-core architectures available. These architectures implement a more complex but more scalable on-chip networks upon which memory and communication operations can be carried, and that is explicitly accessible to the programmer to implement communication or memory consistency. Explicit on-chip networks of massively parallel architectures and the pipeline parallelism in stream computation are a good match to implement on-chip pipelined algorithms. As such algorithm trade accesses to main memory for heavier use of the on-chip network, the main memory is accessed only at the source and sink of the corresponding dataflow graph, this technique can lower memory wall effects to bring more performance. There are many existing languages available to implement parallel computation; however, legacy approach such as pthread are very invasive and difficult to use to manage many cores. OpenMP and OpenCL are less invasive but they still require a deep knowledge of the underlying execution platform, which limits performance portability. CUDA is only available for GPU manufactured by nVidia, which limits severely its portability. Finally, stream languages such as Streamit or CAL provide a good abstractions for stream computing and are popular languages in the stream programming research community.

Chapter 3

Investigations on SCC Capabilities

3.1 Introduction

In this chapter¹, we use various benchmarks to analyze the Single-Chip Cloud computer (SCC) and investigate on characteristics of its network's and memory's bandwidth and latency. We implement several variants of parallel mergesort in order to test the architecture further with a actual algorithm and serve as a comparison basis to an implementation of an on-chip pipelined mergesort discussed in Chapter 4. We first analyze the influence of the core placement on memory access performance and communication latency and bandwidth, using a smaller part of the algorithm and a microbenchmark, respectively. Then we evaluate the performance of the overall merging algorithm implementations on SCC.

3.2 Memory Bandwidth

This work evaluates the performance of the SCC when accessing the off-chip memory. The focus of this study is not on testing the bare hardware. Instead, we are interested in the performance of applications that run on the Linux operating system and use the SCC as it is provided. We see that the per-core read memory bandwidth is largely independent of the number of cores accessing the memory simultaneously, but that the write memory access performance drops when more cores write simultaneously to

¹This chapter is based on the following publications: Kenan Avdic, Nicolas Melot, Jörg Keller and Christoph Kessler: *Parallel sorting on Intel Single-Chip Cloud computer*. A4MMC, ISCA-2011, San Jose, USA, 2011 [7] and Nicolas Melot, Kenan Avdic, Christoph Kessler, Jörg Keller: *Investigation of Main Memory Bandwidth on Intel Single-Chip Cloud computer*. Intel MARC3 Symposium 2011, Ettlingen, Germany, 2011 [104]

the memory. In addition, the global and per-core memory bandwidth, both writing and reading, depends strongly on the memory access pattern.

The goal of our experiments consists in the measurement of the bandwidth available to an application that runs on top of the Linux operating system in standard operating conditions (cores at 533 MHz, on-chip network at 800 MHz, memory controllers at 800 MHz). Furthermore, we are interested in how this bandwidth varies with the number of cores performing memory operations and the nature of the operations themselves, read or write. This is achieved by consecutively reading respectively writing the elements of a large array of integers, aligned by 32 bytes which is the size of a cache line. Thus, consecutive access to all integers (1-int-stride, 4-byte-stride) yields perfect spatial locality whereas 8-int-strided access (4 out of 32 bytes) to the data always results in a cache miss. Each participating core runs a process that executes a program as depicted in Fig. 3.1, where each array is located in the respective core’s private memory and through which the cores iterate exactly once.

While the 1-int-strided and 8-int-strided memory accesses stresses the bandwidth difference due to cache hits and cache misses, the *random* access pattern stresses the memory controllers’ throughput using a random access, making helpless its hardware optimizations that prefetch or parallelize or cache read or write accesses, such as using a plurality of open rows in the attached SDRAMs. To simulate random access, the array is accessed through a function $\pi(j)$ that is bijective in $\{0, \dots, SIZE - 1\}$, where j is the same index (strided 1 int or 8 ints) used to access the array in the strided access described above. In practice, we use $\pi(j) = (a \cdot j) \bmod SIZE$ for a large, odd constant a where $SIZE$ is a power of two and the size of the array to be read. The random access pattern also applies the 1-int-strided, 8-int-strided and mixed patterns described above to the index j .

Finally strided, mixed and random access make all the cores read or write at the same time, along the different access patterns they define. All these patterns also combine read and write operations, one half of processors performing reads, and the second half performing writes. This is denoted as the *combined* access pattern.

In this experiment, a varying number of cores synchronize, then iterate through the array to read or write as described above. Since every memory operation leads to a cache miss in the 8-int-strided access and random access reduces the memory controllers’ performance, such memory operations generate traffic and the time necessary to read the targeted amount of data allows the calculation of the actual bandwidth that is globally available to all cores. The amount of data to be read or written by each core is fixed to 200 MiB. 3 to 12 cores are used, as up to twelve cores share the same memory controller. Cores run at 533 MHz and 800 MHz in two different experiments, while the mesh network and memory controllers remain both at 800 MHz. The global bandwidth and the bandwidth per core are measured: the global bandwidth represents the bandwidth a memory controller

Listing 3.1: Pseudo-code of the microbenchmark for reading access. For writing, the order of the variables in the assignments is exchanged.

```

/* SIZE is a power of two
 * strictly bigger than L2 cache
 */
int array[SIZE];

void memaccess ( int stride )
{
    int i, j, tmp;

    for ( j = 0; j < SIZE; j += stride )
        tmp = array[j];
}

```

provides to all the cores. The bandwidth per core is the bandwidth a core gets when it shares the global bandwidth with all other running cores. Figures 3.1, 3.2 and 3.3 show the global and per core bandwidth measured in our experiments.

Figure 3.1 indicates that both read and write bandwidth are linearly growing with the number of cores. Since the SCC provides no hardware mechanism to manage and share the memory bandwidth served to cores, this shows that all cores together still fail to saturate the read memory bandwidth available. The random access pattern offers a much lower read throughput around 250 MiB/sec with 12 cores running at both 533 and 800 MHz. The write throughput for random stride 1 shows the same performance as write stride 1 (up to 105 and 120 MHz respectively at 533 and 800 MHz) and other write patterns do not exceed 20 MiB/sec nor about 7 MiB/sec for random stride 8 access pattern. This shows that memory controllers struggle to serve irregular main memory request patterns. The absolute numbers of read bandwidth per core in the 1-int-stride experiment are stable around 205 MiB/s and around 125 MiB/s with the 8-int-stride access pattern with cores running at 533 MHz and respectively 305 and 235 MiB/sec at 800 MHz, as shown in Fig. 3.2a. However, the bandwidth per core with the write accesses (Fig. 3.2b) drops with the number of cores from 10 MiB/sec with 3 cores to 9 MiB/sec using 12 cores at 533 MHz and from 11 MiB/sec to 10 MiB/sec at 800 MHz. The P54C's L1 cache no-allocate-on-write-miss behavior may explain this performance drop: as write cache misses do not lead to a cache line allocation, every consecutive write results in a write request addressed to the memory controller. In both cases, the low difference in performance of 1-int-stride and 8-int-stride access patterns shows that the high performance memory controllers are able to compensate efficiently the performance losses due to cache misses. However the mixed access pattern,

(a) Global main memory read bandwidth at 533 and 800 MHz.

(b) Global main memory write bandwidth at 533 and 800 MHz.

Figure 3.1: Measured global memory read and write bandwidth as a function of the number of cores involved, at 533 and 800 MHz.

with one half of the cores reading memory with a 1-int-stride and the second half with 8-int-stride, exhibits lower performance, which shows again the limited capabilities of memory controllers to serve irregular access patterns.

The bandwidth measured per core for the random access pattern reveals better performance with faster cores.

3.3 Communication Latency and Bandwidth

The on-chip network is also of interest, as a future pipelined mergesort implementation would make a heavy use of it. The evaluation method proposed here consists in investigating latency by making a core to send several blocks to another core, and monitor the average time these blocks take to be totally transferred. This test is repeated with growing block sizes and with pairs of cores at a growing distance. Figure 3.4 illustrates how the pairs are chosen to increase the distance to its maximum.

Figure 3.5 pinpoints that the communication speed is a linear function of the distance between cores, and the low standard deviation indicates that this speed is stable. However, when the block size reaches 256 KiB this linear behavior does not hold anymore and the standard deviation gets much higher. 256 KiB is the size of L2 cache and transferring such blocks may generate much more cache misses when storing received values, thus slowing communication drastically down.

3.4 Mergesort

Parallel sorting algorithms have been investigated since about half a century for all kinds of platforms, sorting being an important subroutine for other applications, and being a good model for non-numeric algorithms with a low computation-to-communication ratio. For an overview cf. [3] or [73]. Robison [120] discusses the problem of scaling the number of cores to parallelize quicksort. He argues that with a high number of cores, a performance bottleneck appear in the initial data partitioning phase, that cannot benefit from parallelization. Although a parallel partitioning phase is possible with sample sort, Robison [120] describes how memory bandwidth and a lack of TLB entries in many-core architectures limit the parallel performance of sample sort for big inputs. Finally, Robison [120] argues that mergesort benefits from a memory bandwidth and cache-friendly behavior, and that it shows no bottleneck so that it can benefit from a massive parallelism. Mergesort originated as an external sorting algorithm, with data held on the hard disks and merging done in main memory. A similar situation occurs with multiple cores on a chip, connected by an on-chip network, and the main memory being chip-external. Furthermore, mergesort has a simple control structure, is cache-friendly as the blocks to be merged are accessed consecutively from front to end, and is largely insensitive to distribution of

(a) Read memory bandwidth per core at 533 and 800 MHz.

(b) Write memory bandwidth per core at 533 and 800 MHz.

Figure 3.2: Measured per-core memory bandwidth as a function of the number of cores involved, for strided access patterns, at 533 and 800 MHz.

(a) Memory bandwidth per core with random access pattern at 533 MHz.

(b) Memory bandwidth per core with random pattern at 800 MHz.

Figure 3.3: Measured per-core memory access bandwidth as a function of the number of cores, for random access patterns, at 533 and 800 MHz.

Figure 3.4: Four different mappings of core pairs with a different and growing distance.

Figure 3.5: Average time to transfer 64, 128 and 256 KiB blocks as function of the distance between cores.

Figure 3.6: Data-flow graph of the mergesort algorithm.

values or occurrences of duplicates. For this reason, we chose mergesort for implementation on SCC.

The mergesort algorithm sorts values using a recursive method. The array to be sorted is recursively split into chunks that are trivial to sort then merged together to obtain larger sorted blocks. Figure 3.6 gives an example of the split/merge phases of the mergesort algorithm, where the split operation is actually trivial. The chunks issued from blocks split can be treated separately and independently. This separation enables parallelization of the algorithm thus leading to a *parallel mergesort algorithm*. When the block has been divided into as many chunks as there are cores available, the chunks are sorted sequentially and then merged pairwise on different cores. Experiments with larger problem sizes show that the merging phase dominates overall sorting time; for instance, when sorting 32 Mi integers on 32 cores with the fastest of our mergesort implementations, local sorting accounts for less than 10% of the overall time. For this reason, focus will be on the merging phase in the remainder of this chapter.

Algorithm implementations usually make a more or less heavy use of main memory to load data and to store intermediate or final results. Accesses to main memory represent a bottleneck in some algorithms' performance [81], despite the use of caches to reduce the penalty due to limited bandwidth to main memory. Caches are high-speed memories, close to processing units but are rather small and their effect is less visible when a program manipulates a larger amount of data. This leads to the design of other optimizations such as on-chip pipelining for multicore processors [81].

This section² investigates the actual memory access bandwidth limits of SCC from the perspective of applications that run on the Linux operating system and use the SCC as it is provided to them. As thus, the focus is not what the bare hardware is capable of, but what the system, i.e. the ensemble of hardware, operating system and programming system (compiler, communication library, etc) achieves. Our approach is to use microbenchmarking to create different sets of patterns to access the memory controllers. In Sec. 3.2, we show that the memory controllers can support all cores reading data from their private memory, but that the cores experience a significant

²This is an updated version of the paper by Melot et al. [104].

performance drop when writing to main memory. For both read and write accesses, the available bandwidth is strongly dependent on the memory access pattern.

The first variant of our parallel mergesort, namely *private memory merge-sort algorithm*, is an adaptation to the SCC's architecture. It uses cores' private memory to store integer blocks and shared memory to transfer these blocks from a core to another. Each core starts the algorithm by generating two blocks of size s/p of pseudo-random non-decreasing values, where s is the private memory's size and p is the number of cores. This size is chosen so that the processor that performs the last recursive merge operation can fit both input and output buffer in its private main memory. Then the algorithm enters a sequence of *rounds* as follows: A core merges its two blocks into a larger one and either sends it to another core or waits for a block to be received. All cores whose ID is lower than or equal to half the number of active cores in this round wait for a block to be received. All other cores send their blocks to a recipient core whose ID is $\lfloor \frac{id}{2} \rfloor$, through shared main memory and then turn idle. At the end of a round, the remaining active core number is divided by two. At the last round, only one core is still active and the algorithm ends when this core finishes the merging of both of its blocks. Figure 3.7 provides an illustration of this algorithm.

The communication process is shown in Fig. 3.8. Forwarding blocks uses shared memory for buffering: at every round, the shared memory is divided into buffers of size $2m/p$, where m is the total size of shared memory. This value maximizes the buffer size for block transfers because half of the active cores have to send a block in a round. Both sending and receiving cores can calculate to which shared main memory buffer they should read or write the block to be transferred, from their ID and active round number. A core sends its block to another core by writing a portion of this block in its dedicated buffer in shared main memory. When this shared memory buffer is full, the sending core sets a flag to the recipient core and waits for it to reply another flag stating it is ready to take more. Both the sender and the receiver can determine the amount of data to be sent and received from the total size of the input to be sorted and the number of the active round. Then the process restarts until the whole block has been transferred.

Flags and barriers used for synchronization are sent through the on-chip network. Also in the communication phase, each processor can calculate to what portion of main memory it should read or write. Therefore there is no two cores writing at the same address in shared memory. For each core writing in shared memory, exactly one other core reads the same address after both cores have synchronized using a flag and on-chip network. Finally, the shared memory used for this algorithm is not cached, thus there is no need any shared memory coherency protocol.

A variant of the private memory mergesort algorithm uses the on-die network mesh to communicate blocks between cores. The process *MPB private memory algorithm* is the same as the one described above but individual

Figure 3.7: Cores merge their blocks into one larger block and then send it to another core to merge it.

Figure 3.8: One iteration of the block communication process.

core's MPB memory buffer is used in place of shared memory as transfer buffer. This uses the high speed on-die network and reduces accesses to main memory to transfer data blocks from one core to another. The other two variants of parallel mergesort are similar to the one described above, except that they work directly on shared memory as storage for input and output blocks. Their implementation here actually relies on SCC's shared memory. Since they work directly on shared memory, they do not need any transfer phase in the algorithm. The two shared memory versions of mergesort differ in that one uses L1 and L2 cache and the second one does not use caches at all. Similarly to the private memory algorithm, each core can determine its own chunk in main memory thanks to its ID and the number of the round being run. Thus no coherency protocol is required here. In the case of cached shared memory, the cache is flushed before and after each round in order to make sure that no value cached in a round is considered valid and is read again from the cache in the next round; instead, it is forced to be actually loaded from main memory.

A significant difference between private memory and shared memory variants is that the private memory variant needs a communication phase between all rounds, which requires more or less accesses to main memory, while the shared memory variant does need any transfer phase. This difference gives a performance advantage to the shared memory variants over the private memory based algorithms.

The private memory mergesort algorithm requires the blocks to be copied to shared memory, and then to a different private memory. This doubles access to main memory and significantly slows down the private memory mergesort variant. Figures 3.9a and 3.9b show slightly better performance for the private memory variant compared to the cached shared memory variant but this does not scale with number of cores. As soon as more than 2 processors are used, the private memory gets the worst performances of all algorithms implemented. As expected, the private memory variant using the MPB is faster than its equivalent using shared memory to transfer blocks, as it requires less than half the number of main memory access when transferring a block from one core to another. This latter variant competes very well with the cached shared memory algorithm even though it still requires a communication phase between all rounds. The cached shared memory is the fastest of all algorithms studied here.

However the good performance of the MBP private memory algorithm shows that transferring data from core to core using MPBs and on-die network brings little penalty to the mergesort algorithms studied, and much less accesses to main memory. It demonstrates the high capacity of this on-die network, which encourages the implementation of a pipelined mergesort algorithm using this network.

Another setting alters the algorithms to generate blocks of constant, identical value among blocks and merges them using the same four techniques. This different pattern of data to be merge-sorted aims at showing

Figure 3.9: Performance of the four mergesort variants on SCC, sorting integer blocks with randomly generated data of different size and using different numbers of cores.

Figure 3.10: Performance of the four algorithms merging blocks of constant values, identical among different blocks, using 32 cores.

Figure 3.11: Runtime differences with increasing core count

the behavior of the algorithms in an extreme case of input. Constant, identical values in all blocks denote the case of uniform input as well as an input already sorted. The results of this latter setting are shown in Fig. 3.10. It shows better performance of the mergesort algorithms with an input of already sorted data. However, when compared to Fig. 3.9f, the ratio between times of algorithms merging random and constant input blocks is constant and can thus be neglected. Such absence of significant difference is a characteristic of mergesort algorithms, which are insensitive to distribution of values or occurrences of duplicates.

In the following, the length of the first merge round is compared in experiments with 2^k cores, where $k = 0, \dots, 5$, and initial block size of 2^{20} integers. This is done for the private memory variant, the uncached shared memory variant, and the cached shared memory variant, as described before. Each setting is run 20 times to reduce influences such as the operating

Figure 3.12: Execution times for round 1 of private memory mergesort algorithm, where nodes manage 1 million integers, depending on the distance from a core to its memory controller.

system’s scheduler. The averaged time is computed for each core, as well as the minimum, maximum, and average times over all cores. The results are depicted in Fig. 3.11.

The measurements show that the placement of cores on the die has a significant impact on performance while accessing main memory or when communicating to each other. When accessing to main memory, Fig. 3.12 demonstrates that the closer a core is to the relevant memory controller, the faster it is loading or writing data. It shows a linearly growing minimum time bound to complete round 1 of private memory mergesort algorithm. That round is quite an intensive main memory user (private and shared). In each setting, Fig. 3.11 shows that the runtime increases with the growing number of cores. This indicates contention and/or saturation at the memory controllers, as the code executed is the same in all cases. This effect is very prominent in the shared memory variants. Consequently, on-chip pipelining might help to relieve the memory controllers, especially in applications where the computation-to-communication ratio is low. Figure 3.11 also shows that the difference between the fastest and the slowest core in a round increases with growing core count, especially for the shared memory variants. This indicates that the distance of a core to the memory controller has a notable influence on the share of the controller’s bandwidth that the core gets. Yet, as the distribution of the times is not similar to the distribution of the distances, there must be some other influence as well, which however so far is unknown to us.

3.5 Conclusion

This chapter shows a significant performance penalty from the use of the SCC’s shared main memory, compared to using private memory or the on-chip network [7]. This work uses non-pipelined parallel mergesort as a case

study. The results also indicate that the distance to the memory controller has an influence on round-trip time. Further tests indicate that there is sufficient read bandwidth when accessing memory in a cache-friendly pattern, but that write bandwidth suffers [104]. Also, a comparison of different programming styles for the Intel SCC finds that using the on-chip message-passing buffers (MPB) provides performance advantages [25], and that using shared memory has performance problems either due to lack of caching or overhead from ensuring cache coherency by software [25].

Mergesort (and in particular its global merging phase) is an interesting case of a memory-access intensive algorithm. We have implemented four versions of mergesort on SCC, and evaluated their performance. Beyond the mergesort algorithms themselves, our experiments enlighten important factors that influence performance on SCC.

Our results motivate that on-chip pipelining would make sense also for SCC. Because the pipeline trades access to main memory for heavier use of the on-chip network, the main memory is accessed only when loading blocks to be sorted and when the final merged block is written back. Also, because off-chip memory access is more expensive than communication through the mesh network, it is expected that the on-chip pipelining technique brings significant speedup over any of the parallel mergesort implementations described here. Its performance may even be improved thanks to awareness of distance between active cores and distance to main memory controllers to generate better task-to-processor mappings.

The memory wall represents an important performance limiting issue still present in multicore processors, and implementations of parallel algorithms are still heavily penalized when accessing main memory frequently [81]. This work enlightens the available memory bandwidth on Intel's Single Chip Cloud Computer when several processors perform concurrent read and write operations. The measurements obtained here and the difficulty we experience to actually saturate the read memory bandwidth show that the cores embedded in the SCC cannot saturate all together the read memory bandwidth available for read access patterns that behave regularly. However, the measurements obtained from the write access patterns demonstrate a much smaller write bandwidth available. Also, we can note that the available bandwidth for both read and write strongly depends on the memory access pattern, as the low bandwidth on random access patterns indicates. Thus, there is no point in reducing the degree of parallelism in order to increase the available bandwidth for tasks requiring a high main memory bandwidth. The measurements shown in this chapter show a behavior possibly adapted to program restructuring techniques such as on-chip pipelining and our previous implementation of on-chip pipelined mergesort [81]. In this implementation, many tasks mapped to several cores fetch input data in parallel from main memory, and a unique task running on a unique core writes the final result back to main memory, therefore limiting expensive main memory accesses. However, the gap between the memory bandwidth

available and the limited capabilities of cores to saturate it shows that there is room to add more cores, run them at higher frequency or add SIMD ISA extensions. Without such improvements in the cores' processing speed and accordingly higher demands on memory bandwidth, our ongoing research on program restructuring techniques such as on-chip pipelining is, for SCC, limited to implementation studies leading to predictions of their theoretical speed-up potential, rather than demonstrating concrete speed-up on the current SCC platform. Such techniques could speed up memory-access intensive computations such as sorting [7, 81] on SCC-like future many-core architectures that are more memory bandwidth constrained.

Chapter 4

On-Chip Pipelining on SCC

4.1 Introduction

Engineering parallel algorithms for many-core architectures involves many considerations, e.g. load balancing, communication patterns, and memory access patterns. Especially for algorithms that mainly transport and modify large amounts of data, so-called streaming applications, the accesses to main memory may represent a performance bottleneck [81], despite the use of caches, because of the limited bandwidth to off-chip main memory. The goal for bandwidth optimization, combined with other design targets, leads to approaches such as on-chip pipelining for multicore processors [81].

In this chapter¹, we consider sorting large data sets as a simple and well-researched model for streaming applications. We investigate the implementation of sorting on the SCC as an algorithm engineering problem, and devise a combination of pipelined and non-pipelined mergesort as a good fit for that architecture. Preliminary performance measurements with a prototype implementation indicate the validity of our hypothesis and the applicability of on-chip pipelining.

The remainder of this chapter is structured as follows. In Section 4.2 we present our arguments for the choice of sorting algorithm used, together with an integer linear programming (ILP) approach to optimize the load balancing and latency reduction in memory accesses. In Section 4.3 we present preliminary performance results of our prototype implementation, and give an outlook to future work in Section 4.4.

¹This chapter was initially published as Nicolas Melot, Christoph Kessler, Kenan Avdic, Patrick Cichowski and Joerg Keller: *Engineering parallel sorting for the Intel SCC*. *Procedia Computer Science*, Vol.9(0), pp.1890–1899, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012. [105].

4.2 A Hybrid Parallel Mergesort Algorithm

In order to engineer a sorting algorithm for a given manycore system, one first has to consider whether the data set to be sorted fits into the on-chip memory (either local memories or caches) or whether the data is stored in the off-chip main memory. We consider the latter situation, especially as the SCC does not possess big local memory but only provides small on-chip caches and small buffers for core-to-core communication. While the main memory of SCC in principle can be used as a shared memory, there are two obstacles here. First, the system software only allows a fraction of the main memory to be declared as shared memory, the majority is used as private memory, where each private memory section can only be accessed by one user process. Second, that shared memory is, by default, not cached. Thus, as each core can only have one outstanding memory request, this results in a low bandwidth for shared memory accesses, especially as all cores will access the same memory controller (accesses to the private memory are distributed over the 4 available memory controllers). If caching is enabled for shared memory, there is no hardware coherency i.e. the application would have to take care of that. Therefore, we have concentrated on avoiding using shared memory, and work with message-passing like algorithms: although each core physically is able to access each part of the external memory, there is no way (at least not on application level) to re-label data in one private memory to belong to the private memory of a different core. Thus a core-to-core communication has to take place to transfer data in this case.

For sorting algorithms where each core uses its off-chip private memory and communicates with other cores over the on-chip interconnection network, we must consider that the network's bandwidth (64 GByte/s) is much higher than bandwidth to the off-chip memory, where the latter varies largely with the access pattern, cf. Chapter 3 or [104]. Furthermore, bandwidth to off-chip main memory is strongly dependent on the memory access pattern, with access to blocks of consecutive words providing the highest bandwidth. Thus, we arrive at a situation similar to parallel external memory algorithms. This moves our focus to merge sort which is a popular external sorting algorithm. It also has proven more efficient than other algorithms like e.g. bitonic sort used in CellSort on the Cell processor [44]. In order to relieve the external memory, pipelined mergesort has been introduced to forward results from a merge step directly to a another core where the follow-up merge step is to be executed; see e.g. [81] for results on the Cell processor.

On-chip pipelined mergesort organizes this merge tree computation in a pipeline, with all tasks running concurrently on the SCC's 48 cores. Producer and consumer tasks mapped to the same tile communicate through the tile's local memory, those mapped to different tiles through the mesh network. The leaf tasks start the computation by reading data from memory and they forward their intermediate merging results towards their consumer

tasks at the next level in the task graph. These tasks can then start the same process, and so on, until all the tasks in the pipeline are active. This schema restricts memory writes to the root node. As the merge operation is done blockwise, follow-up tasks can start as soon as leaf tasks have produced their first block of intermediate results.

As sending and receiving tasks can be mapped to different processors, the sending tasks can restart with new input data while the receiving one can process the data it has just received. Hence, parallelism is achieved through the overlapping execution of successive tasks. As there can be many more tasks than processors, a processor will generally handle several tasks. This introduces an optimization problem for mapping tasks to processors. One difficulty consists in keeping a balanced computational load among the cores, another one in the preference for keeping the distance between sender and receiver small, especially if the memory controller is involved, in order to reduce delays in data transmission. In [81], an ILP-based model for multi-criterion optimized mapping of tasks was presented. In this work, an adapted model is described in Section 4.2.2.

4.2.1 Hybrid Parallel Sorting on the SCC, Overview

Merging complexity (using ordinary sequential merging) is linear in the input size. Hence, each level of the merge tree involves the same computational load. In general, as the root node will allocate a core alone, an m -level mergetree will be mapped to m cores. The high number of 48 cores available on SCC however rules out this direct approach because of the vast size of the merge tree involved. Hence, on-chip pipelined mergesort needs to be adapted by using several smaller mergetrees concurrently. As each tree will allocate the same number of cores, both the number of the trees and the number of the cores per tree (which equals the number of levels per tree) should be a divisor of the number of the cores (48). Also, the number of the trees should be a power of two, to allow a balanced final merging phase, and the number of levels in the tree should be as large as possible, to get more work done in this phase, but less than 10, to avoid cores with more than 100 concurrent tasks. As a result, we arranged for 8 trees of 6 levels each. The guidelines above make this arrangement preferable over 4 trees with 12 levels each, and also over 16 trees with 3 levels each. Thus, our approach to parallel sorting on SCC as implemented in this work consists of three phases.

- In phase 0, all the leaf tasks of 8 merging trees mapped to 8 disjoint sets of 6 cores, as detailed in Section 4.2.2, fetch input data from files and locally sort their inputs using `qsort`.
- Phase 1 runs 8 on-chip-pipelined 6-level merge trees simultaneously. This phase produces 8 sorted subsequences from 8 root nodes. Note that the mapping of the tree leaves for phase 1 influences the runtime

of phase 0 as well via the number of cores that carry leaves, so this has to be considered when mapping the tasks to cores.

- Phase 2 merges these 8 subsequences together, using a non-pipelined mergesort where merge nodes are parallelized. We preferred this over using a parallel sample sort with all 48 processors, because of the coordination overhead involved in implementing the latter solution.

4.2.2 Optimizing Task Mapping by ILP

For mapping the overall pipelined task graph to the SCC resources, we have developed multiple variants of an integer linear programming (ILP) based method that either optimize (1) for the aggregated overall distance (in hops) between communicating tasks, weighted by their inter-task communication volumes, or (2) for the aggregated overall distance of tasks to their memory controller weighted by the tasks' memory access volumes (and possibly by the direction of the access, to prioritize write access). In addition, the model also balances the computational load per core, and tries to reduce the number of leaf tasks per core, to improve the runtime of phase 0. To consider all goals, we use a linear combination of the respective target functions, which is controlled by weight parameters.

The architecture model part of our ILP model can be configured by the number of tiles per row and column and by the number and distribution of memory controllers. Thus, besides the usage to provide optimal mappings for a concrete application such as mergesort, the model can also be used for computer architecture research [37]. For example, one can compute optimal positions for different numbers of memory controllers, given an application and the number and arrangement of cores for a hypothetical manycore chip.

In order to reduce model complexity by exploiting symmetries in the SCC architecture, we consider the problem of mapping a pipelined merge task *forest* (here, eight parallel 64-to-1 pipelined merge trees) to SCC and split it into four symmetric subproblem instances. Each subproblem is mapped to an SCC quadrant with its memory interface controller. Hence, only one instance, i.e., mapping two 64-to-1 merge trees (thus 126 tasks) to one quadrant consisting of 3×2 tiles of 2 cores each, actually needs to be solved. Two 6-level binary trees are easily modeled by a single 7-level tree where the now artificial root node 1 is assigned a zero workload. We further simplify the problem by mapping tasks to tiles only, not to individual cores.

We model the nodes of a pipelined b -ary merge tree (usually, $b = 2$) with k levels, which has thus $(b^k - 1)/(b - 1)$ merge tasks, as an index set $V = \{1, \dots, (b^k - 1)/(b - 1)\}$, where 1 denotes the root merger and where for $v \in V \setminus \{1\}$, $\text{parent}(v)$ denotes the merger task to which the task v forwards its output. Hence, the inner nodes (tasks) are $V_{\text{inner}} = \{1, \dots, (b^{k-1} - 1)/(b - 1)\}$ and $V_{\text{leaves}} = V - V_{\text{inner}}$ denotes the b^{k-1} leaves. For each leaf task v , we add an artificial *co-task* $v + b^{k-1}$ that models read memory accesses of v . For the root 1, we add a co-task 0 that models its

write memory accesses. The overall set of nodes comprising tasks and co-tasks is thus $V_{ext} = \{0, \dots, (2 \cdot b^k - b^{k-1} - 1)/(b - 1)\}$, the co-task set is $V_{ext} - V$. See Figure 4.1a for an example with $k = 3$ and $b = 2$.

A co-task is to be mapped on the memory controller closest to its corresponding leaf or root task, i.e., on the memory controller of the SCC quadrant where the corresponding task is mapped, while the tasks in V are to be mapped to SCC tiles. The (normalized) computational workload w_v of a merge task $v \in V$ at depth d is $1/b^d$ with $d \in [0; k - 1]$. The memory access volume of a co-task $v' \in V_{co}$ is defined by the computational workload of its corresponding task; a co-task has no computational workload.

We model a generic SCC-like architecture as a 2D mesh of tiles with arbitrary extent and arbitrary numbers and positions of memory controllers around the mesh. The SCC mesh is defined by extents $NRows$ and $NCols$, such that $Rows = \{1, \dots, NRows\}$ denotes the tiles' row indices and $Cols = \{1, \dots, NCols\}$ denotes the tiles' column indices.

For simplicity, the MICs are not modeled by extra tiles adjacent to specific boundary tiles but as part of those boundary tiles, as specified by the (generic) SCC architecture. The additional link to an extra tile would not influence the optimization. The binary parameter *secondMIC* configures if there are one or two MICs per (double) row, i.e., MICs on one or both sides of the SCC. See Fig. 4.1b for an example.

Figure 4.1: Examples of a merge tree and SCC-like architecture.

The mapping to be computed for all tasks and co-tasks is given by the

binary solution variables x where $x_{v,r,c} = 1$ iff node v is placed on tile (r, c) , for $v \in V_{ext}$, $r \in Rows$ and $c \in Cols$. In the mesh, we name each horizontal edge by its left endpoint $(r, c) \rightarrow (r, c + 1)$ and each vertical edge by its lower endpoint $(r, c) \rightarrow (r + 1, c)$, see Figure 4.1b for an illustration. In order to compute communication distances for tree edges, we use auxiliary binary variables $yh_{v,r,r'}$ and $yv_{v,c,c'}$ respectively, for $v \in V_{ext}$, $r, r' \in Rows$ and $c, c' \in Cols$. Here, $yh_{v,r,r'} = 1$ iff task v is placed in row r and task $\text{parent}(v)$ is placed in row r' . We also need to model to which quadrant (memory controller) a tile belongs, which is captured by the auxiliary binary variables $quad_{v,q}$ with $v \in V$ and $q \in Rows$ and a binary variable denoting whether MICs are attached on one or on both outer columns; the standard setting of SCC is one quadrant per half doublerow.

Solution variables $sumDistComm$, $sumDistMem$, $maxCompLoad$ and $maxLeaves$ denote various optimization goals, namely the weighted sum of communication distances, the weighted sum of memory access distances, the maximum computational load on any tile and the maximum number of leaves per tile, respectively. Then, the objective function of our ILP instance becomes

$$\begin{aligned} \text{minimize} \quad & \epsilon_1 \cdot maxCompLoad + \epsilon_2 \cdot sumDistMem \\ & + \epsilon_3 \cdot sumDistComm + \epsilon_4 \cdot maxLeaves, \end{aligned}$$

where $0 \leq \epsilon_i \leq 1$ and $\sum_i \epsilon_i = 1$. By choosing the tuning parameters appropriately between 0 and 1, priority can be given to the various optimization subgoals. We have the following constraints. $maxCompLoad$ is defined by the sum of computational work of all tasks mapped to any tile:

$$\forall r \in Rows, c \in Cols : maxCompLoad \geq \sum_{v \in V} w_v \cdot x_{v,r,c}$$

$maxLeaves$ is defined by the number of leaves mapped to any tile:

$$\forall r \in Rows, c \in Cols : maxLeaves \geq \sum_{v \in V_{leaves}} x_{v,r,c}$$

Each task node must be mapped to exactly one processor:

$$\forall v \in V_{ext} : \sum_{r \in Rows} \sum_{c \in Cols} x_{v,r,c} = 1$$

where the placement of the co-tasks is additionally constrained: The root's co-task 0 must be fixed to a memory controller in the root's quadrant, where we assume that memory controllers are placed in rows with odd index:

$$\forall r \in \{1, \dots, NRows/2\} : \sum_{c \in Cols} x_{0,2r,c} \leq 0$$

$$\forall r \in \{1, \dots, NRows/2\} : \sum_{c \in [2, NCols - secondMIC]} x_{0,2r-1,c} \leq 0$$

Likewise, a leaf's co-task must be fixed to the memory controller of the quadrant where the leaf is mapped:

$$\forall v \in V_{co}, r \in \{1, \dots, NRows/2\} : \sum_{c \in Cols} x_{v,2r,c} \leq 0$$

$$\forall v \in V_{co}, r \in \{1, \dots, NRows/2\} : \sum_{c \in \{2, \dots, NCols - secondMIC\}} x_{v,2r-1,c} \leq 0$$

The auxiliary quadrant variables are defined as follows:

$$\forall v \in V, mcr \in \{1, \dots, NRows/2\}, mcc \in \{0, 1\} :$$

$$\begin{aligned} quad[v, 2mcr - 1 + mcc] &\leq \sum_{\substack{r \in \{2mcr-1, \dots, 2mcr\}, \\ c \in \{(mcc \lfloor NCols/2 \rfloor + 1, \dots, mcr \lfloor NCols/2 \rfloor + NCols/2\}}}} x_{v,r,c} \\ \forall v \in V : \sum_{mcr \in [1, NRows/2]} \sum_{mcc \in \{0,1\}} quad[v, 2mcr - 1 + mcc] &\geq 1 \end{aligned}$$

Then, the following constraints force the co-tasks of leaves and the root on the same quadrant as their corresponding tasks:

$$\forall mcr \in \{1, \dots, NRows/2\}, mcc \in \{0, 1\}, v \in V_{leaves}, i \in \{1, \dots, b\} :$$

$$x_{v+b^{k-1}, 2mcr-1, mcc \cdot (NCols-1)+1} \leq quad_{v, 2mcr-1+mcc}$$

$$\forall mcr \in \{1, \dots, NRows/2\}, mcc \in \{0, 1\} :$$

$$x_{0, 2mcr-1, mcc \cdot (NCols-1)+1} \leq quad_{1, 2mcr-1+mcc}$$

The communication load on each mesh edge is defined by constraints such as

$$\forall u \in V_{inner}, i \in \{1, \dots, b\}, r1 \in Rows, r2 \in Rows :$$

$$yh_{u+b(u-1)+i+1, r1, r2} \geq \sum_{c \in Cols} x_{u+b(u-1)+i+1, r1, c} + \sum_{c \in Cols} x_{u, r2, c} - 1$$

$$\forall u \in V_{inner}, i \in \{1, \dots, b\}, r1 \in Rows, r2 \in Rows :$$

$$yh_{u+b(u-1)+i+1, r1, r2} \geq 0$$

for the row segments, and similarly for the column segments. The weighted sum of communication distances is defined as follows:

$$\begin{aligned} sumDistComm &= \sum_{\substack{u \in V_{inner}, i \in \{1, \dots, b\}, \\ r1 \in Rows, r2 \in Rows}} yh_{u+b(u-1)+i+1, r1, r2} \cdot 0.5|r2 - r1| \cdot w_u \\ &+ \sum_{\substack{u \in V_{inner}, i \in B, \\ c1 \in Cols, c2 \in Cols}} yv_{u+b(u-1)+i+1, c1, c2} \cdot 0.5|c2 - c1| \cdot w_u \end{aligned}$$

The weighted sum of memory access distances is defined by

$$sumDistMem = \sum_{\substack{v \in V_{leaves}, \\ r \in Rows, c \in Cols}} x_{v,r,c}(r+c-2)w_v + \sum_{\substack{r \in Rows, \\ c \in Cols}} x_{1,r,c}(r+c-2)$$

As an example, Fig. 4.2 depicts two mappings of two 6-level binary trees with roots 2 and 3 onto one quadrant of the SCC. Both mappings result from different settings of the weight parameters and are optimal with respect to load balancing. The level-wise mapping (Fig. 4.2a), incurs a small distance to memory for leaf and root tasks but increases core-to-core communication. The block mapping (Fig. 4.2b) puts each 4-level subtree containing leaves into one core. This minimizes the distances for core-to-core communication but increases the distance to memory for leaf and root tasks. Thus, if goal (2) has a stronger influence in the optimization, the level-wise mapping is to be preferred over the block-wise mapping. If writes to memory would get a higher weight than reads, because the write bandwidth to memory is smaller, the advantage would be even more on the side of the level-wise mapping. In the block-wise mapping however, the leaves are distributed over 8 cores, in contrast to only 2 cores for the level-wise mapping, so that the runtime for phase 0 is improved. Thus, depending on the detailed user preferences, different mappings can be chosen out of a number of pareto-optimal mappings.

ILP Solver Results Table 4.1 shows the runtime requirements of our ILP model for cases with a single MIC, using CPLEX 10.2 as ILP solver. We find that when considering one quadrant the problem can be solved to optimality within at most 2 minutes in almost all cases, while considering the whole SCC with 24 tiles leads to considerably higher complexity. However, using a newer ILP solver such as Gurobi might make even such configurations feasible. First tests with Gurobi (also running on a faster machine with more memory) show that, for mapping sufficiently large trees, significant improvements in optimization time are possible, e.g., a 250x speed-up for mapping a 5-level binary tree to a 4x6 mesh, and a 6-level tree mapping which was not feasible with CPLEX could be solved to optimality in 58 seconds with Gurobi.

4.2.3 Phase 1: On-Chip Pipelined Merge

The first phase of the on-chip pipelined mergesort implemented on SCC is quite similar to the implementation on Cell [81]. The focus here is given to 6 cores of one of the SCC's 4 quadrants, i.e., one octant. All 7 other octants of 6 cores execute in parallel the same actions.

All merging tasks of a 6-level binary merge tree are allocated to 6 cores, using an optimal task mapping computed by the ILP solver (see Section 4.2.2). The cores that are assigned some of the merge tree leaves locally sort each

Figure 4.2: Mappings of two 6-level binary trees onto one SCC quadrant.

Table 4.1: ILP solver times in seconds for mapping 4, 5 and 6 level binary merge trees to different SCC mesh sizes, using 1 MIC and accounting for maximum communication loads across any mesh edge, in seconds (CPLEX 10.2). A dash (“–”) means that no optimal result was obtained after 10 minutes, but an approximation solution is reported.

3x2 Quadrant, 1 MIC					6x4 Quadrant, 1 MIC		
k	#var	$\epsilon = 0.5$	$\epsilon = 0.1$	$\epsilon = 0.9$	k	#var	$\epsilon = 0.5$
4	374	0.5	0.1	1.0	4	1818	–
5	774	1.8	5.5	0.2	5	3786	93.6
6	1574	53.9	–	111.9	6	7722	–

input buffer stored in their private main memory (phase 0). Once all leaves' buffers are sorted, they are merged together as part of the on-chip pipelined merging procedure. Each merge task is assigned a buffer in the MPB to read data from, and some space in L2 cache to store the merge output before pushing it toward the parent task, with the exception of leaf tasks which take their input directly from the main memory, and of the root task that writes its output to main memory through the L2 cache. Every task in the merging tree executes the following sequence:

1. *Check and merge*

Checks both input buffers of the task from the core's MPB: if both contain data and if the output buffer in L2 cache has some free space, then the task merges as many elements as possible, i.e. the minimum of the smallest amount of elements available in both input buffers and the amount of elements that the output buffer can take up at the moment.

2. *Check and push*

Checks the input buffer of this task's parent task; if it can be filled with the sorted data available, then the parent's input buffer is filled and what is written to the parent is removed from the task memory. The root task writes directly to main memory without any prior verification.

The checks are performed thanks to a *header* preceding every buffer in the MPB. It allows tasks to keep track of the amount of data already sent and received, the amount of data available in this buffer or the amount of free space, as well as where to read input or write output. From this information and the size of the buffer, a task can calculate how much space is available to write, or how much data is available in this buffer. All tasks mapped to the same core time-share it in a Round-Robin fashion, where the time quantum for each task is the time necessary to perform either of the two steps enumerated above.

4.2.4 Phase 2: Parallel Non-Pipelined Merge

Phase 2 consists in merging the 8 subsequences into one final sorted sequence. For this, we use a parallel mergesort, where each merge node is implemented by several cores to increase performance. For example, in a first round, four merger tasks run concurrently, where each task is implemented by 4 cores. Every task splits one of the two corresponding input blocks into 4 chunks of equal size and separated by 3 pivots. Then they look up the positions of those 3 pivots in the other block. Thus there is no need to transfer the pivots between the cores, saving the overhead of communications. Now each core knows the positions of all pivots in both blocks; thus they can merge the respective chunks. From the positions of pivot i

in both blocks, it can also compute the position of the merged block in the sorted sequence. While this approach to parallelization does not guarantee balanced work between the cores, the overhead is low which leaves room for imbalance. In the second round, 2 merger tasks run concurrently, each implemented with 8 cores, and in the last round the root merger task is implemented by 16 cores. A higher parallelization lead to longer runtimes because the work imbalance gets too large.

Initially, we also considered a parallel sample sort in shared memory, where each of the 8 subsequences is split into 48 chunks according to 47 pivots, where the pivots are generated as averages of 47 pivots for each subsequence. Then each of the 48 cores could sequentially merge 8 corresponding chunks, one from each of the 8 subsequences. The concatenation of the 48 resulting sequences then forms the final sorted sequence. Details of the parallel sample sort can be found in [8]. While the degree of parallelism is higher with sample sort (48 cores instead of 16), its runtime still was longer than for the non-pipelined mergesort because of the coordination overhead involved that necessitated several synchronizations and scatter/gather communications. Therefore, we did not investigate that variant further.

4.3 Experimental Evaluation

We implemented a prototype of the hybrid sort algorithm for SCC described in Section 4.2. We ran it on the 48 cores, varying the total input size N from 2^{20} to 2^{25} integers (32 bits each). As our previous work on on-chip pipelined mergesort on Cell [81] exhibits different performance with different available buffer size per core, the sorting algorithm for SCC is run with allocated space in the MPB of 4096, 6144 and 8128 bytes. Each combination of input size and buffer size is run a hundred times to reduce interferences from the underlying operating system, using random data (uniformly distributed unsigned integers) as input. We use a variant of the level-wise mapping from Fig. 4.2a. The algorithm is monitored so that also sub-timings for the initial sequential sorting phase (phase 0) and the parallel sorting (phases 1 and 2) are available. The timing results are depicted in Table 4.2. We see that the runtime is growing linearly with the input data size, with a lower pace between 2^{22} and 2^{24} integers. Also, the difference in buffer size only has an influence for small data size, but is negligible for large data sizes.

The runtime distribution between the phases shows a tendency of phase 0 to grow faster than phase 1, as the runtime in phase 0 grows with $O(N \cdot \log N)$ because of the sorting, while the runtime in phase 1 grows with $O(N)$. In contrast, the runtimes for phase 2 tend to decrease their share on the total runtime from 2^{20} to 2^{25} integers, respectively from 5%, 14% and 81% to 14%, 25% and 61% (block-wise variant). This indicates that for a larger amount of data to merge (from 2^{23} integers in our measures), Parallel merge gets faster than an on-chip pipelined merge, despite the disadvantage of using shared memory.

Table 4.2: Runtime results for different data sizes with maximum buffer size, using both mapping shown in Fig.4.2 (values expressed in ms).

Data size	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
Total (layer)	481.4	960.8	1842.6	2938.2	5090.4	10308.6
Total (block)	415.6	817.8	1533	2279.8	3689.2	7321
Phase 0 (layer)	88	191.4	413.2	880.8	1875	3991
Phase 0 (block)	22.2	48.4	103.6	222.4	473.8	1003.4
Phase 1	57	114	226	451	902	1805
Phase 2	336.4	655.4	1203.4	1606.4	2313.4	4512.6

We compare the performance of phase 1 to the best variants of parallel merge shown in [7], that uses 4 cores to run a 4 levels tree and use 8 cores to run a 8 levels tree. As we run our on-chip pipelined merge phase (phase 1) with 8 merging trees, we focus on the runtime of one of them merging an input block 8 times smaller than the global input. Thus we compare phase 1 runtime merging a total of 2^{25} integers (2^{22} per tree) to variants shown in [7] merging 2^{22} integers. The classic merge implementation in [7] takes 3000 ms and 4 cores to run a 4 levels tree and 3200 ms and 8 cores to run a 8 levels tree. Our on-chip pipelined merge takes 1805 ms with 6 cores to run a 6 levels tree, yielding a speedup of 1.6 and 1.7, respectively and an efficiency of 1.1 and 2.26, respectively. Thus, our on-chip pipelined merge is more efficient than a classic parallel merge thanks to main memory access optimization despite the unfavorable case. Previous work [104] indicate that the SCC's cores struggle to saturate the main memory, due to old and slow cores coupled to modern, fast memory controllers. In other architectures, on-chip pipelining may yield even better speedup.

Table 4.3: Scaling the algorithm using one to four quadrants (values expressed in ms).

	1 quadrant	2 quadrants	4 quadrants
Phase 0 (block)	1003	473	222
Phase 1	1805	902	451
Phase 2	571.2	1221	1606
Total [ms]	3379.2	2596	2279
Speedup	1	1.3	1.5
Efficiency	1	0.6	0.4

We show in Table 4.3 the behavior of our algorithm when scaling it from 1 quadrant (twelve cores) to 4 quadrants (48 cores). All variants sort the same input size, that is equally distributed among all cores in all 1, 2, or 4 quadrants used. Since using 2 or 4 quadrants would result in smaller chunks distributed to merging trees, and phases 0 and 1 involve unsynchronized quadrants, it roughly divides the runtime by 2 or 4, respectively, compared

to using 1 quadrant. As shown in table 4.3, phase 2 does not scale perfectly. This can be explained by the growing number of input blocks with the number of quadrants, 2 to 8, that leads to more merge steps to be achieved by phase 2. In our measurements, using more quadrants always results in higher speedup, although this increasing speedup does not maintain the efficiency at a constant level.

We are not aware of performance results for other sorting algorithms on the SCC, besides our own implementation of non-pipelined mergesort [7]. Thus we compare with sorting algorithms implemented on the Cell processor [21]. The Cell processor, usually employed in pairs, provides 8 processors called SPEs running at 3.2 GHz, where the processors can issue up to 2 instructions per cycle, and the instructions work on 128-bit data (or four 32-bit integers). Thus, considering operations on 32-bit integers as a basic unit, an SPE performs 8 times as many operations per cycle compared to an SCC core. As the SPE operating frequency is 6 times higher than for the SCC cores running at 533 MHz, an SPE is about 48 times more powerful than a SCC core. As a Cell pair comprises 16 SPEs compared to 48 cores in an SCC, a Cell pair is roughly 16 times more powerful than a SCC. CellSort [44], which uses a local sort (corresponds to our phase 0) followed by a bitonic sort (corresponds to our phases 1 and 2), needs 565 ms to sort 32M integers with 16 SPEs, where the time for phase 0 is omitted. As we need about 6317.6 ms for the same data size, we are by a factor of about 1.43 more efficient, considering the factor 16 computed above, although the Cell BE may feature other advantages over the SCC's P54C that are hard to quantify, such as a newer and more efficient instruction set implementation.

4.4 Conclusion

We presented a hybrid sorting algorithm for the SCC processor as an algorithm engineering case study, where we had to combine different algorithmic techniques, knowledge about the processor's memory bandwidth, and an advanced ILP-based multi-objective mapping optimization method to derive an efficient implementation. Preliminary experimental results indicate that the resulting code yields good scaling properties and is competitive with simple off-chip parallel mergesort as well as other high performance parallel sorting algorithms. The algorithm should be portable to other upcoming mesh-based manycore architectures such as e.g. Tilera processors [139] and also to NUMA-based systems such as the AMD Interlagos manycore processor, where the coherent on-chip caches could take over the role of the message-passing buffers. The techniques presented to derive an efficient implementation should be adaptable to other streaming algorithms.

Chapter 5

Scheduling

Exploiting effectively massively parallel architectures is a major challenge that stream programming can help to face. In this chapter¹, we investigate the problem of generating energy-optimal code for a collection of streaming tasks that include parallelizable or moldable tasks on a generic manycore processor with dynamic discrete frequency scaling. Streaming task collections differ from classical task sets in that all tasks are running concurrently, so that cores typically run several tasks that are scheduled round-robin at user level in a data-driven way. A stream of data flows through the tasks and intermediate results may be forwarded to other tasks like in a pipelined task graph. In this chapter we consider *crown scheduling*, a novel technique for the combined optimization of resource allocation, mapping and discrete voltage/frequency scaling for moldable streaming task collections in order to optimize energy efficiency given a throughput constraint. We first present optimal off-line algorithms for separate and integrated crown scheduling based on integer linear programming (ILP). We make no restricting assumption about speedup behavior. We introduce the fast heuristic *Longest Task, Lowest Group* (LTLG) as a generalization of the Longest Processing Time (LPT) algorithm to achieve a load-balanced mapping of parallel tasks, and the *Height* heuristic for crown frequency scaling. We use them in feedback loop heuristics based on binary search and simulated annealing to optimize crown allocation.

Our experimental evaluation of the ILP models for a generic manycore

¹This chapter is based on the publications by Kessler et al. [83], Melot et al. [108] (DOI: 10.1145/2687653 <http://doi.acm.org/10.1145/2687653>) and Melot et al. [107], as well as the following submissions: Nicolas Melot and Christoph Kessler: *Voltage Island-Aware Energy-Efficient Scheduling of Parallel Streaming Tasks on Many-Core Processors*. Accepted for presentation at the 9th Nordic Workshop on Multi-Core Computing (MCC2016), Trondheim, Norway, 2016 and Nicolas Melot, Christoph Kessler, Patrick Eitschberger and Jörg Keller: *Co-optimizing Core Allocation, Mapping and DVFS in Streaming Programs with Moldable Tasks for Energy Efficient Execution on Manycore Architectures*. Submitted at 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS), Orlando, Florida USA, 2017

architecture shows that at least for small and medium sized streaming task collections even the integrated variant of crown scheduling can be solved to optimality by a state-of-the-art ILP solver within a few seconds. Our heuristics produce makespan and energy consumption close to optimality within the limits of the phase-separated crown scheduling technique and the crown structure. Their optimization time is longer than the one of the other algorithms we test, but our heuristics consistently produce better solutions.

We make the following contributions:

- We propose to extend Crown Scheduling heuristics [108] to support core consolidation.
- We reduce the restrictions on ILP-based integrated Crown Schedulers, by allowing arbitrary numbers of cores. In particular, we focus on a balanced binary decomposition, an extremely unbalanced binary decomposition and another binary decomposition based on the analysis of the tasks' efficiency functions.
- We give a ILP formulation to schedule moldable tasks with arbitrary parallel efficiency functions for many-core homogeneous architectures that admit a bounded, discrete frequency set but that does not assume any restriction on possible core allocation or mapping, i.e., it can compute optimal schedules given a sufficient optimization time.
- We show by a mathematical argument that our unrestricted scheduler produces schedules at best 3.7 times better than Crown Schedulers, under realistic execution platform description parameters, improving upon existing Crown Scheduling implementations [108] and therefore earlier techniques [116, 143].
- We use benchmarks based on random task collections, classic parallel algorithms and on the Streamit benchmark suite to experimentally demonstrate that our improved ILP-based Crown Schedulers produce schedules of a quality with no significant difference to solutions produced by our optimal scheduler and that our heuristics find in a short time near-optimal solutions.

5.1 Introduction

The increasing attention given to multi- and manycore architectures yields the need for programming models able to take easily profit of their power. Among many solutions to the general parallel programming problem, the streaming approach is particularly adapted to coarse architectures and gives opportunities toward the design of highly scalable parallel algorithms [48, 90]. Examples include image processing and video encoding/decoding pipelines, and other applications operating on large data volumes, e.g. streamed mergesort or map reduce. We consider the steady state of a software pipelined

streaming task application where all streaming tasks are active simultaneously. Each task repeatedly consumes some amount of input, does some computation, and produces output that is forwarded to another task (or to memory if it is a final result).

A streaming task application can be modeled by a static streaming task graph (or a Kahn Process Network [77]) where the nodes represent the tasks, that are not annotated with runtimes but with average computational rates. In the steady state, an edge between tasks u and v does not indicate a precedence constraint but denotes a communication of outputs from producer task u to become inputs of consumer task v , and is annotated with an average bandwidth requirement.

As the workloads of different tasks may vary around the given averages and as there are normally more tasks than cores on the underlying machine, dynamic round-robin scheduling enables several tasks to run concurrently on the same core. In this case, a scheduling point is assumed to occur after the production of a packet, and a round-robin non-preemptive user-level scheduler normally is sufficient to ensure that each task gets its share of processor time, provided the core has enough processing power, i.e. is run at a frequency high enough to handle the total load from all tasks mapped to this core.

Here we consider *streaming task collections*, i.e., we model the tasks' computational loads only. Our results can still be applied to streaming task graphs if we assume that the latencies of producer-consumer task communications can be hidden by pipelining with multi-buffering from one scheduling round to the next, and that on-chip network links are not oversubscribed².

We assume that our underlying machine consists of p identical processors, which can be frequency-scaled independently. We consider discrete frequency levels. We do not consider voltage scaling explicitly, but as most processors auto-scale the voltage to the minimum possible for a given frequency, this is covered as well. We allow for arbitrary core power models, which might comprise static and dynamic power consumption, and can be modeled analytically or based on measurements on real hardware. Beyond analytic power models like f^α , other models (e.g. derived from measurements) are also possible by replacing the corresponding terms in ILP formulations and heuristics.

A task instance produces a packet of output data from packets of input data. Its granularity depends both on the data packet size and on the complexity of the computation performed on it. In practice, we assume task instances (i.e., per packet) to take a time in the milliseconds range. This is coarse enough that the per-task user-level scheduling overhead, buffer management and frequency switching overhead can be considered reasonably low compared to task execution time. We allow for *modalable* (multithreaded, aka. parallelizable) tasks that can internally use a parallel algorithm involv-

²Scheduling tasks under bandwidth constraints and optimizing latency is a matter of future work.

ing multiple processors (possibly using communication and synchronization via shared memory or message passing) in order to achieve parallel speedup. We do not schedule tasks in a malleable way, i.e. a task cannot increase or decrease its number of allocated cores while running. Such task collections can still be scheduled by considering them as moldable tasks. We make no assumptions about the parallel efficiency functions of moldable tasks; these are parameters of our model. Moldable, partly moldable and sequential tasks might be mixed.

We are interested in statically (1) allocating processing resources to tasks and (2) mapping the tasks to processors in such a way that, after (3) suitable task-wise frequency scaling given a throughput constraint, the overall energy consumption during one full round of the round-robin scheduler is minimized. We refer to this combined optimization problem (1–3) shortly as *energy-efficient scheduling*. With this respect, we make the following contributions:

- We introduce the *crown structure* as a constraint on resource allocation, which reduces the complexity of allocation, mapping and discrete frequency scaling considerably, namely from p to $O(\log p)$ possible task sizes for allocation and from 2^p to $O(p)$ possible processor subsets for mapping. We show that this constraint makes the exact solution of the still considered NP-hard optimization problem feasible even for medium problem sizes. We argue that the resulting schedules are efficient and also flexible for dynamic rescaling to adapt to fluctuations in streaming task load at runtime, in order to save further energy e.g. in situations where a task may not be data-ready in a schedule round and has to be skipped. We present *crown scheduling*, a novel method for energy-efficiency scheduling under the crown structure constraint.
- We show how to apply discrete frequency scaling of the tasks in a given crown schedule to optimize its energy efficiency for a given throughput constraint.
- We give implementations of crown scheduling as Integer Linear Programming (ILP) models and evaluate them for a generic manycore architecture, showing that it can be solved to optimality (with regards to the crown structure) for small and most medium sized streaming task collections within a few seconds.
- We present efficient heuristics for crown allocation, mapping and discrete frequency scaling phases for large streaming task collections. We give a time complexity analysis for each of them and we show that the quality of their solution competes with optimal crown schedules for a much lower optimization time.

The remainder of this chapter is organized as follows: Section 5.2 introduces the general concepts and central notation. Section 5.3 considers

the separate optimization of crown resource allocation/mapping and subsequent frequency/voltage scaling, while Section 5.4 provides the integrated solution of crown resource allocation, mapping and scaling in the form of an integer linear programming model that can provide an optimal solution for small and medium sized problems. Section 5.5.3 addresses dynamic schedule rescaling. Section 5.7 presents experimental results. Section 5.8 discusses related work, and Section 5.9 concludes the chapter and proposes some future work.

5.2 Crown Scheduling

A crown³ is a recursive b -ary decomposition of the set of cores into subgroups down to singletons. These subgroups are to become the only possible mapping target for tasks. The base of a crown represents the full core set to run very parallel tasks. This inter-dependency decreases with the parallel degree of tasks and the increasing height in the crown. The crown structure aims to guarantee that cores belonging to a higher region of the crown (running less parallel tasks) can perform their work independently of cores belonging to another branch.

We consider a generic multicore architecture with p identical cores (processors). For simplicity of presentation, we assume in the following that $p = 2^L$ (see ⁴) is a power of 2 (see ⁵). All p cores can be individually and dynamically set to run at one frequency level from a finite set F .

The set of processors $P = \{P_1, \dots, P_p\}$ is hierarchically structured into a set H of $2p - 1$ *processor subgroups* by recursive binary partitioning as follows: The *root group* G_1 equals P ; it has the two child subgroups $G_2 = \{P_1, \dots, P_{p/2}\}$ and $G_3 = \{P_{p/2+1}, \dots, P_p\}$, four grandchildren groups $G_4 = \{P_1, \dots, P_{p/4}\}$ to $G_7 = \{P_{3p/4+1}, \dots, P_p\}$ and so on over all $L + 1$ tree levels, up to the *leaf groups* $G_p = \{P_1\}, \dots, G_{2p-1} = \{P_p\}$. Unless otherwise constrained, such grouping should also reflect the sharing of hardware resources across processors, such as on-chip memory shared by processor subgroups. Let C_m denote the set of all groups that contain processor P_m . For instance, $C_1 = \{G_{2^z} : z = 0, \dots, L\}$. Let $p_i = |G_i|$ denote the number of processors in processor group G_i . Where it is clear from the context, we also write i for G_i for brevity.

We consider a set $T = \{t_1, \dots, t_n\}$ of n moldable, partly moldable or sequential streaming tasks, where each task t_j performs *work* τ_j and has a *maximum width* $W_j \geq 1$ and an *efficiency function* $e_j(q) > 0$ for $1 \leq q \leq p$

³The name is inspired by the shape of a crown, whose triangular spikes symbolize a decreasing degree of connectivity, see Fig. 5.1a.

⁴We can define $p = b^L$ with $b > 1$ and $b \in \mathbb{N}$ for generality. However, our method yields more optimization opportunities as b decreases, for the same complexity. Therefore in this paper, we are only interested in the case where $b = 2$.

⁵See Sec. 5.5.2 for a generalization of Crown Scheduling with arbitrary recursive decompositions where p is any strictly positive integer.

Figure 5.1: A group hierarchy and a possible corresponding unscaled crown schedule for 8 processors.

that predicts the parallel efficiency (i.e., parallel speedup over q) with q processors. For *moldable tasks*, W_j is ∞ , i.e., unbounded; for *partly moldable tasks*, W_j can be any fixed value > 1 , and for *sequential tasks* $W_j = 1$. For all tasks t_j we assume that $e_j(1) = 1$, i.e., no parallelism overhead when running on a single processor⁶. Where clear from the context, we also write j as shorthand for t_j .

Resource allocation assigns each task t_j a *width* w_j with $1 \leq w_j \leq \min(W_j, p)$, for $1 \leq j \leq n$. As additional constraint we require for crown allocation that w_j be a power of 2, and thus could be mapped completely to one of the $2p - 1$ processor subgroups introduced above. A *crown mapping* is a mapping of each task t_j with assigned width w_j to one of the processor subgroups in $\{G_1, \dots, G_{2p-1}\}$ of a size matching w_j . For each processor group G_i , $1 \leq i \leq 2p - 1$, let $T^{(i)}$ denote the set of tasks $t_j \in T$ that are mapped to G_i . For each processor P_m , $1 \leq m \leq p$, let T_m denote the set of tasks $t_j \in T$ mapped to P_m , i.e., to any $G_i \in C_m$. Conversely, let R_j denote the group to which task t_j is mapped. *Crown scaling* attributes a running frequency level to each task. If task j is given frequency level $f_j \in F$, and mapped to group i , then all processors G_i must run at frequency f_j from when task j begins until task j finishes.

We define the execution time for tasks, groups and processors as functions of allocation, mapping and frequency settings given to all tasks. The execution time of a task j running on w processors at frequency f is defined as follows:

$$tt_j(w, f) = \frac{\tau_j}{f \cdot e_j(w) \cdot w} \quad (5.1)$$

⁶In principle, partial moldability might be also expressed by manipulating the e_j functions, but in this chapter we use the upper limits W_j for this purpose. — We may assume that the efficiency functions e_j are monotonically decreasing, although this assumption is not strictly necessary for the techniques of this chapter.

Where clear from the context, we write tt_j for the execution time of task j with given width w_j and frequency f_j . We conveniently assume here that the running time of task j is proportional to the frequency inverse $1/f$, but the model allows extensions to sample the tasks' speedup function of frequency. Similarly, we consider that tasks use all w cores through all their entire execution time⁷. The static runtime of a processor group i , denoted gt_i , is the sum of execution times of all tasks mapped to this group. If they all run at frequency f , then we write $gt_i(f)$ as:

$$gt_i(f) = \sum_{j \in T^{(i)}} tt_j(w_j, f) \quad (5.2)$$

We write for convenience $gt_i = \sum_{j \in T^{(i)}} tt_j(w_j, f_j)$ for the execution time of group i , with all tasks j running at the frequency f_j they have been assigned. A processor m runs as long as the sum of all execution time of all groups it belongs to, assuming frequency f :

$$pt_m(f) = \sum_{i \in C_m} gt_i(f) \quad (5.3)$$

When clear from the context, we also write $pt_m = \sum_{i \in C_m} gt_i$ for the execution time of processor m , where all tasks j that it executes run at the frequency f_j that they are assigned. When we need to compare task j , group i or processor m by unscaled execution time, i.e. with no bias due to any task frequency settings, for instance to compute task's running frequency, we use the constant frequency $1Hz$ with $tt_j(w_j, 1)$ (Eq. 5.1), $gt_i(1)$ (Eq. 5.2) and $pt_m(1)$ (Eq. 5.3), respectively.

For each processor P_m , we order the tasks in T_m in non-increasing order of width, e.g., by concatenating the elements of all $T^{(i)} \in C_m$ in increasing order of group index i . The relative order of the tasks within each processor group must be kept the same across all its processors, e.g., in increasing task index order. We call such a p -tuple of processor schedules a *crown schedule*.

A user-level, round-robin runtime scheduler running on each processor P_m , $1 \leq m \leq p$, uses a crown schedule for the (instances of) tasks in T_m to determine their order of execution. The scheduler works in *rounds*. Each round starts with a *conceptual* global barrier synchronization across all p processors⁸. Thereafter the tasks are executed round-robin as specified in the chosen crown schedule of the tasks in T_m . Tasks that are not data ready

⁷If not, then task j 's efficiency function should not be modeled after its work τ_j , the number of processors w and a parallelization overhead, but instead derive the parallel time function tt_j of each task j from analysis or measurements.

⁸There is no need for an explicit barrier to separate rounds. For the prediction of makespan, a conceptual barrier can be assumed: As the cores involved in a parallel task (e.g., the first full-width parallel task running on the root group at the beginning of a round) will task-internally communicate and synchronize with each other, the resulting waiting times that would occur if cores start the task at different times can conceptually be moved to the end of the preceding round, thus having the same effect as a zero-latency barrier separating the rounds.

are skipped. Then the next round follows. The stable state of a round in a crown-based streaming computation involves (almost) all tasks in T as they all have data to process. See Figure 5.1b for an illustration.

The processing of tasks in decreasing order of width is required to make sure that all processors participating in the same moldable task (instance) start and stop approximately at the same time, allowing for efficient inter-processor communications and avoiding idle times, except at the end of a round. Between tasks (even of different widths) within a round, scheduled idle time (“external fragmentation”) cannot occur because the processors of the subgroup assigned to the subsequent task automatically start quasi-simultaneously after finishing the previous task where they also participated; a moldable task is either data-ready or not, which means it is skipped either by all or none of the processors in the assigned processor group.

Crown scheduling computes crown allocation, mapping and frequency scaling for streaming tasks and minimizes energy consumption under a makespan constraint. We fix a target throughput for an application as the inverse of a time budget M to execute a round. Crown scheduling allows any analytic energy consumption model. Let $PC(f)$ be an arbitrary dynamic power function for the crown scheduler. For a concrete machine, it is sufficient to use results from power measurements of the target machine at the available frequencies [24]. In our experiments, we set $PC(f) = f^\alpha$ with $\alpha = 3$. The simple model of power and speedup as a function of frequency does not capture the detailed interactions between processors and main memory [131] depending on programs running. However it can give a rough estimation of speedup and power consumption.

We consider the energy spent at idle time as negligible, and also ignore the time and energy required to switch frequencies, as they are small against the time of a round. Equation 5.4 shows our energy consumption model.

$$energy = \sum_{j=1}^n tt_j \cdot w_j \cdot PC(f_j) \quad (5.4)$$

5.3 Phase-Separated Energy-Efficient Crown Scheduling

In this section we discuss separate optimization algorithms for each of the subproblems crown resource allocation, crown mapping and crown scaling. An integrated and more expensive approach is be discussed in Section 5.4. An overview of these problems is given in Figure 5.2.

We can reduce the overall energy usage by scheduling tasks to run at a frequency as low and uniform as possible through a pipeline stage [91]. Also, early phases of the phase-separated approach are unaware of their influence on the efficiency of subsequent phases. However the more idle time processors have between the target makespan and the execution of

Figure 5.2: Overview of phase-separated vs. integrated crown scheduling. The methods shown by solid boxes are detailed in this paper.

their last task, the lower frequency can run their task at and the lower energy a pipeline stage consumes. Therefore, the allocation and mapping phases aim at optimizing for unscaled execution time and load-balancing. The frequency scaling phase scales frequencies as low as possible under the makespan constraint.

In this section, we call *crown-optimal* an optimal solution of the allocation, mapping or frequency scaling phases in a separate manner. Because they are solved separately, they do not participate in an optimal solution to the general scheduling problem studied in this chapter.

5.3.1 Crown-optimal Crown Resource Allocation⁹

Let Q_z , where $z = 0, \dots, \log_2 p$, be the set of indexes of all the tasks which are assigned width $w_j = 2^z$. Then task t_j ($j \in Q_z$) has unscaled runtime $\hat{\tau}_j = tt_j(w_j, 1)$ assuming that its processors run at fixed frequency 1. The total execution time of all tasks running one after each other is

$$R_z = \sum_{j \in T} \frac{\tau_j}{e_j(2^z) \cdot 2^z}. \quad (5.5)$$

As most tasks run in parallel in the final schedule, the final execution time is likely to be much lower after load-balancing. An optimal resource allocation chooses the Q_z in a way to minimize

$$\sum_{z=0}^{\log p} R_z$$

i.e., we minimize the total runtime assuming that the processors run at a fixed frequency and running the tasks in parallel will further reduce the final execution time. To do this by linear optimization, we introduce binary variables $v_{j,z}$ with $v_{j,z} = 1$ iff task j gets width $w_j = 2^z$. Then Eq. 5.5 transforms to

$$R_z = \sum_{j \in T} v_{j,z} \frac{\tau_j}{e_j(2^z) \cdot 2^z}.$$

⁹ A programming error in the corresponding section by Melot et al. [108] made the scheduler using this phase to perform poorly. This section describes a technique that is corresponds better to the experiment that was run.

and we need the constraint

$$\forall j : \sum_z v_{j,z} = 1 .$$

It is simple to find an allocation that minimizes the total runtime, under the assumption that the total runtime is given by Eq. 5.5. Equation 5.5 is minimized iff all $\tau_j / (e_j(2^z) \cdot 2^z)$ are minimized. Therefore, we compute for each task j the speedup $g_j(q) = e_j(2^q) \cdot 2^q$ for each $q \in [0..L]$ and select one q that maximizes g_j , from all $L = \log p$ possible allocations for task j . Since we have n tasks, this takes at most $O(n \cdot \log p)$ time steps in total. Because all terms in the sum of Eq. 5.5 are independent and because we minimize each, the computed allocation is optimal. Because it computes an allocation such that each task runs individually for the shortest possible time, we refer to this algorithm as *fast allocation*.

5.3.2 Crown-optimal Task Mapping

For an optimal crown mapping given an allocation, we treat all widths separately, i.e., we solve $\log_2 p$ smaller optimization problems. For the tasks j of a given width $w_j = 2^z$, we try to distribute them over the p/w_j processor groups available such that the maximum load over any processor is minimized.

In order to formulate the mapping of the tasks of a given width 2^z as an integer linear program, we minimize a variable *maxload* (note that the target function here is simple) under some constraints. To express the constraints, we use binary variables $y_{j,i}$ where j runs (in consecutive numbering) over all tasks with width 2^z and i runs over all considered $p/2^z$ processor groups $G_{p/w_j}, \dots, G_{2p/w_j-1}$. Hence, $y_{j,i} = 1$ iff task j is mapped to processor group i . The constraints are as follows. Task j is mapped to exactly one processor group among those that have the right width w_j :

$$\forall j \text{ with width } w_j = 2^z : \sum_{i=p/w_j}^{2p/w_j-1} y_{j,i} = 1$$

We define L_i as the load mapped to processor i , that is the sum of the load of each group i is part of

$$\forall i \in 1..p : L_i = \sum_{q \in C_i} \sum_{j=1}^n \hat{\tau}_j \cdot y_{j,q}$$

And *maxload* as the maximal L_i , i.e., the makespan:

$$\forall i \in \{1, \dots, p\} : \text{maxload} \geq \max(L_i)$$

Figure 5.3: Mapping quality with or without load balancing

5.3.3 Heuristic Task Mapping with Load Balancing

We describe the *Longest Task, Lowest Group* (LTLG) mapping heuristic for the load balancing problem modeled in Sec. 5.2. We consider the *height* of a processor group i as shown in Eq. 5.6 and illustrated in Fig. 5.4a. The height of a group i is the maximum running time $\max_{m \in G_i} pt_m(f)$ of all its processors running at frequency f . We call for convenience $height_i$ the height of group i where all tasks j run at frequency f_j (Eq. 5.7). The LTLG heuristic discards all tasks of group 1 from the height of any group because group 1 cannot participate in any load imbalance ($height_{i=1} = 0$). As no frequency scaling is performed yet at the mapping stage (Fig. 5.2), LTLG computes the height of a group i for a constant frequency f ($height_i(f)$). Note that this is not a limitation of our LTLG heuristic as scaled tasks would only result in a different running time for each task to map.

$$height_i(f) = gt_i(f) + \sum_{i'=2}^{\lfloor \log_2 i \rfloor} gt_{\lfloor \frac{i}{2^{i'}} \rfloor}(f) + \max(height_{2i}(f), height_{2i+1}(f)) \quad (5.6)$$

$$height_i = gt_i + \sum_{i'=2}^{\lfloor \log_2 i \rfloor} gt_{\lfloor \frac{i}{2^{i'}} \rfloor} + \max(height_{2i}, height_{2i+1}) \quad (5.7)$$

LTLG is based on three principles to produce better load-balanced mappings:

1. The assignment of each task j to the group i of least $height_i$ among groups of size w_j limits load imbalance while building a schedule.
2. The insertion of tasks of highest parallel running time first (with w_j processors) lowers the risk of creating load imbalance when adding the last tasks to a schedule.
3. If tasks have the same parallel running time, inserting tasks of highest width first keeps lowest width tasks available to later fill in the schedule's holes.

(a) *Height* of a schedule for 8 processors and processor groups 3, 6 and 13 and area (gray hash) to be rescaled if the black hashed task is delayed or skipped.

(b) Four binary search scenarios: (1) Minimal efficiency increase does not prevent valid schedules (bottom, plain black squares), (2) search for highest valid value from the beginning (mid-dark green dots), (3) minimal efficiency increase prevents valid schedules then search for higher valid value (dark red diamonds) and (4) Binary search fails to find any allocation (top, light blue triangles).

Figure 5.4: Illustration of the *Height* of a schedule for eight processors and 4 possible scenarios of schedules' makespan evolution during binary search.

Figure 5.5: Running example of 7 tasks mapped with LTLG yielding a makespan of 7. Tasks are sorted and numbered after their parallel running time and degree as described by principles 2 and 3. Tasks 1 to 7 with their current allocation run for 5, 4, 4, 3, 2, 2 and 1 time units. The reader may build schedules without principles 2 and 3 (or using their complement instead) to experience their contribution.

$$\phi(i_1, i_2) = \begin{cases} \text{height}_{i_1} > \text{height}_{i_2} & \text{if } \text{height}_{i_1} \neq \text{height}_{i_2} \\ i_1 > i_2 & \text{otherwise.} \end{cases} \quad (5.8)$$

$$\phi(j_1, j_2) = \begin{cases} tt_{j_1}(w_{j_1}, 1) > tt_{j_2}(w_{j_2}, 1) & \text{if } tt_{j_1}(w_{j_1}, 1) \neq tt_{j_2}(w_{j_2}, 1) \\ w_{j_1} > w_{j_2} & \text{if } tt_{j_1}(w_{j_1}, 1) = tt_{j_2}(w_{j_2}, 1) \\ & \text{and } w_{j_1} \neq w_{j_2} \\ j_1 > j_2 & \text{otherwise.} \end{cases} \quad (5.9)$$

The algorithm maintains $\log_2 p$ priority queues, one per group size. The priority of groups is defined by their least height, or their group number if both groups have the same current height (Eq. 5.8). Since tasks of size $w_j = p$ do not create load imbalance and as the root group is the only group of size p , we do not use a priority queue for it. The algorithm first sorts the tasks using Eq. 5.9 to compare them, with both their parallel running time and their width. Considering tasks in this order, we assign task j to the highest priority group i of size w_j as given by the priority queue for size w_j . Then we recompute the runtime gt_i of group i and its height, as well as for all its children groups recursively. If the new height of group i is the maximum height among all children of group $i' = \lfloor i/2 \rfloor$, then the height of i' is also updated, and this is repeated for i' until the root group (excluded) is reached.

At each insertion of task j to a group i , we recompute the group's runtime gt_i . This takes constant time as we can add τ_j to an existing gt_i value, initialized to 0. Then, we need to update the height of all $O(p)$ descendants groups of i as well as all $O(\log p)$ ancestor groups of i . Each group update requires a constant time update of the group starting time for descendant groups and of the maximum child height for ancestor groups. Finally, for each height recomputed, the group list for the corresponding level must be sorted to reflect the new height. This can take $O(\log p)$ time by removing and inserting again the group in the list. Each task insertion takes therefore $O(p \log p + \log^2 p)$ time steps. As we insert n tasks, the time complexity of the LTLG heuristic is $O(n(p \log p + \log^2 p)) = O(np \log p)$.

5.3.4 Crown-optimal Voltage/Frequency Scaling of Schedules

We assume that when running all cores at maximum speed $F_{\max}$ all the time, the resulting makespan is $\hat{M} \leq M$. The gap $M - \hat{M}$ and any processor-specific idle times at the “upper end” of the crown can then be leveraged for voltage/frequency scaling to obtain better energy efficiency. We call this *scaling the schedule*.

For a given crown, energy efficiency can be optimized by each processor

running each of its tasks t_j requiring work τ_j at a frequency f_j chosen from a given set $F = \{F_1, \dots, F_s\}$ of s discrete (voltage and) frequency levels, such that it still meets the throughput constraint. We require

$$\forall m = 1, \dots, p : pt_m \leq M$$

and the overall energy usage as defined by Eq. 5.4 is minimized, subject to the additional constraint that all processors of a group must use the same frequency level for a task.

5.3.5 Height Heuristic for Voltage/Frequency Scaling

The *Height* frequency scaling heuristic reduces individual tasks' frequencies so that the final schedule satisfies the throughput constraint M and tries to minimize energy consumption. In general, energy consumption is lower when all tasks run with the same frequency compared to running them at different frequencies for performing the same work in the same time interval [91], therefore we decrease moderately the frequency of all tasks together instead of decreasing aggressively the frequency for some of the tasks while greatly increasing the frequency of other tasks in order to satisfy the makespan constraint M . Our heuristic is an implementation of the Best-Fit heuristic for the bin packing problem, where each group is a bin and each bin's free space is the difference between the group's height and the target makespan.

The algorithm begins by assigning all tasks the maximum frequency $F_{\max}$ and computing the initial height $height_i(F_{\max})$ of all groups i . All tasks j are queued after their decreasing running time tt_j at frequency f_j (initially $\forall j : f_j = F_{\max}$). We call $f_{j,cur}$ any frequency assigned to task j at any point in the algorithm. We call F_{next} the frequency variable that iterates over the set of frequencies F sorted decreasingly. For all tasks j in the task priority queue, if $f_{next} < f_{j,cur}$ and the time penalty resulting in running task j at frequency $f_{next} < f_{j,cur}$ is lower than, or equals the gap between the height $height_{i_j}$ of the group i_j (to which task j is mapped) and the target makespan M (that is, if $tt_j(w_j, f_{next}) - tt_j(w_j, f_{j,cur}) \leq M - height_{i_j}$), then we can assign task j the frequency f_{next} , update the height of group i_j as described in Sec. 5.3.3, and insert the newly scaled task into a new task priority queue for the next frequency iteration.

The initial frequency assignment to tasks takes $O(n)$ steps, the computation of the initial height for each group takes $O(n)$ for each $O(p)$ groups, hence $O(pn)$. Tasks and frequencies are sorted decreasingly in $O(n \log n)$ and $O(|F| \log |F|)$ time steps, respectively. In each nested loops iterations, taking the longest running time from the sorted task queue takes $O(1)$ time steps. Checking if running task j at frequency f_{next} does not make the schedule to cross the target makespan M takes $O(1)$ time steps if the height of group i_j is already computed. Updating the height of group i_j takes $O(p)$ time steps and adding the task in the new priority queue takes $O(\log n)$ time. Since we have $|F|$ frequency levels and n

tasks, we run $n \cdot |F|$ iterations. Therefore the Height heuristic runs in $O(n + pn + n \log n + |F| \log |F| + pn \cdot |F| \log n) = O(|F| \cdot pn \log n)$ time steps.

5.3.6 Binary Search Allocation Heuristic

The methods described above for the allocation, mapping and frequency scaling phases rely on the mapping phase capability to compute task distribution close to optimal and produce a schedule whose runtime is the average load that each core receives (Eq. 5.5). However, we target the minimization of energy consumption under makespan constraint instead of runtime. As parallelization usually induces additional processing cost due to a task j 's non-perfect efficiency function $e_j(q) \leq 1$, a too high parallelization consumes more energy. This suggests to give tasks as little parallelism as possible [11].

We set a minimal acceptable efficiency value $e_{\min}$ for allocations of processors to all tasks, in order to limit parallelization and reduce its cost. We run the *fast allocation*, with the additional constraint that no task j should be allocated a number of cores that yields $e_j < e_{\min}$. The constraint is implemented as a $O(1)$ condition to discard any allotment q for a task j if $e_j(q) < e_{\min}$ in the search for q that minimizes $tt_j(q, 1)$. Then, we use the fast heuristics described in Sections 5.3.3 and 5.3.5 to check if the limited parallelization still allows our heuristics to compute a valid schedule. If so, we increase the minimal efficiency threshold by Δ and compute a new schedule. Otherwise we decrease the minimal efficiency by Δ . Finally, we update Δ to its half. We implement a binary search on $e_{\min}$ to maximize the minimal efficiency in the search space $(0, 1)$, with an initial threshold 0.5 and $\Delta = 0.25$ (Fig. 5.4b). The binary search stops when the minimal efficiency is higher than any possible value for e_{j,w_j} ($W_j > 1$) for all tasks j (Fig. 5.4b case 1) or when Δ becomes lower than the minimal efficiency difference between two allocations, for any task.

The binary search sequentializes tasks as far as task mapping and frequency scaling allow the production of valid schedules. However, more sequential tasks take longer time to execute. This can result in the frequency scaling phase to increase the task's frequency, which can replace the parallelization overhead by a much higher frequency cost, due to the fast increase of $PC(f) = f^\alpha$. During the binary search, we keep track of the energy consumption of every schedule generated and we keep the best. The best initial solution is computed with fast allocation (Sec. 5.3.1) with no efficiency constraint ($e_{\min} = 0$). Before starting our binary search, we also try an allocation with a perfect efficiency constraint $e_{\min} = 1$, that is only possible with sequential or perfectly scalable tasks. If such an allotment results in a schedule running all tasks at the lowest possible frequency, then we select this solution and skip the search loop. Otherwise, we compare it to the initial best solution and begin the binary search iterations.

Figure 5.4b illustrates four possible search scenarios. In scenario 1, a

Figure 5.6: Running example of 3 tasks scaled with the Height heuristic for 2 processors of 3 frequency levels.

progressive increment makes the final schedule's makespan to approach M , but it never reaches or crosses it. All allocations are valid but the best one can be any of them (bottom curve, plain black squares). In scenario 2, the initial $e_{\min}$ does not allow subsequent heuristics to compute a valid schedule. $e_{\min}$ is drastically reduced by initial Δ , and the resulting schedule's makespan is much lower than the target. The binary search method continues to halve Δ and increasing and decreasing $e_{\min}$ by Δ until Δ is too small and the allocation that yields the best schedule is kept (mid-dark green rounds). In scenario 3, the binary search increments $e_{\min}$ so the final schedule approaches M as in scenario 1 but as $e_{\min}$ increases, the heuristics begins to fail at computing a valid schedule. Binary search behaves then as in scenario 2, until Δ is too small (dark red diamonds). Finally in scenario 4, the binary search may always decrease $e_{\min}$ but still fails to produce any allocation that the heuristics could use to compute any valid schedule (top light blue triangles). In this case, we use the initial best solution computed with the fast allocation strategy (Sec. 5.3.1) and rely on LTLG and Height heuristics to compute a valid schedule, if possible.

The binary search algorithm explores the search space $(0, 1)$ for Δ . Each round runs the LTLG ($O(np \log p)$) and Height ($O(|F| \cdot pn \log n)$) heuristics and run therefore in $O(np(\log p + |F| \cdot \log n))$ time steps. We define the stop condition as Δ being smaller than the efficiency gap between two allocations for all tasks, which we calculate before beginning the binary search. The minimal allocation gap takes time $O(n \log p)$ to compute (see Sec. 5.3.1 on allocation), where allocations that exhibit too low efficiency are pruned during search. If g is the minimal efficiency gap between two possible allocations for any task (a large value of g means that all tasks scale badly), then $\gamma = \lceil \log_2(\Delta/g) \rceil$ is the number of rounds in our binary search. We also compute twice a fast allocation (time $O(n \log p)$) as an initial solution and attempt with a perfect efficiency constraint. The overall complexity of our binary search is therefore $O(n \log p + 2(n \log p + np \log p + |F| \cdot pn \log n) + \lceil \log_2(\Delta/g) \rceil \cdot np(\log p + |F| \cdot \log n)) = O(\lceil \log_2(\Delta/g) \rceil \cdot np(\log p + |F| \cdot \log n))$.

5.3.7 Simulated Annealing

The binary search method described in Sec. 5.3.6 limits the allocation space exploration to solutions where all tasks share a common efficiency constraint. In particular, it does not permit any fine tuning through individual tasks' allotment adjustment. We use the fast LTLG mapping and height frequency scaling heuristics within a simulated-annealing meta-heuristic that optimizes the core allocation for each individual task. Our simulated annealing thus optimizes allocation for a task collection.

We define the *neighborhood* of an allocation as other allocations within a fixed *distance*. The distance between two allocations is a real in the interval $[0, 1]$, that defines a proportional amount of the n tasks, whose number of cores allotted differs between both allocations. The distance also defines the

proportional variation of number of cores, as a power of 2, that run the same task in both solutions. Searching in the neighborhood of an allocation within a distance 0, is searching in the set of all allocations whose exactly one task is allocated the next higher or lower power-of-2 number of cores, compared to the allocation of origin. Searching in the neighborhood of an allocation within a distance 1 is searching in all possible allocations. A search within the neighborhood of an allocation always excludes the allocation of origin.

We begin with a schedule computed with binary search (Sec. 5.3.6). The LTLG mapping and height frequency scaling heuristics compute a schedule for this allocation and we use the resulting energy consumption (Eq. 5.4, $PC(f) = f^\alpha, \alpha = 3$) as feedback to the simulated annealing allocation phase. In our experiment, we search a new solution within the neighborhood of the current allocation at a maximal distance 0.5 at each simulated annealing iteration. We allow at most 2 iterations before we decrease the simulated annealing temperature (starting at 8) by multiplying it by our cooling factor 0.6. For each temperature level, we accept at most 2 improving solutions. The simulated annealing stops when the simulated temperature reaches 0.9, or whenever it finds a correct solution where all tasks are sequential and run at the minimal frequency.

We have $\lceil \log(8/0.9)/\log(1/0.6) \rceil = 5$ iterations until the final temperature is reached. We allow at most 2 transformations per iteration, resulting in at most $10 = O(1)$ runs of the task mapping and frequency scaling phases. Because we use binary search as an initial solution, the overall time complexity is $O(np(\log p + |F| \cdot \log n + \lceil \log_2(\Delta/g) \rceil) + 10(pn(\log n + \log p)) = O(np(\log p + |F| \cdot \log n + \lceil \log_2(\Delta/g) \rceil))$.

5.4 Integrated Energy-Efficient Crown Scheduling

Separate crown allocation, crown mapping and a-posteriori scaling of the resulting crown schedule may lead to suboptimal results compared to co-optimizing resource allocation, mapping and discrete scaling from the beginning. See Fig. 5.7 for an example.

The integration of crown allocation, mapping and scaling is modeled as follows: We construct an integer linear program that uses $(2p - 1) \cdot s \cdot n$ binary variables $x_{i,k,j}$ where $x_{i,k,j} = 1$ denotes that task j has been assigned to group i and should run at frequency level F_k . We require that each task be allocated and mapped to exactly one group and frequency level:

$$\forall j = 1, \dots, n : \sum_{i=\max(p/W_j, 1)}^{2p-1} \sum_{k=1}^s x_{i,k,j} = 1 .$$

and forbid the mapping of a task with width limit W_j to an oversized group:

$$\forall j = 1, \dots, n : \sum_{i=1}^{\max(p/W_j, 1)-1} \sum_{k=1}^s x_{i,k,j} = 0 .$$

(a) Optimal phase-separated crown schedule

(b) Optimal integrated crown schedule

Figure 5.7: Phase-separated and integrated solutions; the lighter the tasks, the lower their running frequency. Optimal allocation used in the phase-separated approach restricts the quality of the final solution. Frequencies are represented by colors and darkness: dark red is high frequency (5) and lighter yellow represents low frequency (1).

Another constraint asserts that no processor is overloaded

$$\forall m = 1, \dots, p: \text{time}_m := \sum_{i \in C_m} \sum_{k=1}^s \sum_{j=1}^n x_{i,k,j} \cdot \frac{\tau_j}{p_i e_j(p_i) F_k} \leq M$$

Then we minimize the target energy function (as derived from Eq. 5.4)

$$\sum_{i=1}^{2p-1} \sum_{k=1}^s PC(F_k) \cdot \sum_{j=1}^n x_{i,k,j} \cdot \frac{\tau_j}{e_j(p_i)}$$

Note that the coefficients τ_j , p_i and $e_j(p_i)$ are constants in the linear program. Note also that the time and energy penalty for frequency switching is not modeled. One sees that $(2p-1) \cdot s \cdot n$ variables and $p+n$ constraints are needed to allocate, map and scale n tasks onto p cores with s frequency levels.

5.5 Crown Scheduling Extensions

In this section, we exploit the crown structure to design several extensions to Crown Scheduling. This includes an alternate energy model (Sec. 5.5.1) for a more accurate modeling of energy consumption, *crown configuration* (Sec. 5.5.2) that enables Crown Schedulers to compute schedules for non-power of 2 target architectures, *dynamic crown rescaling* (Sec. 5.5.3) to

dynamically adjust the frequency of a subset of tasks in an application with a low complexity and to improve energy consumption and robustness, *core consolidation* (Sec. 5.5.4) that, when taking idle energy into account, improves further energy consumption by switching cores off and offload tasks to other cores and *voltage and frequency island-aware* Crown Scheduler (Sec. 5.5.5) that computes schedules taking into account architectural voltage and frequency island constraints.

5.5.1 Energy model

In Sec. 5.2, we describe a simple energy model based on the power consumption of the target architecture as a function of frequency. However, this model does not take into account the energy drawn by cores when they are unused. In this section, we provide an alternative energy model that includes idle phases and switching off cores.

The power consumption of a processor core can be roughly split into dynamic power and static power. Dynamic power P_d is consumed because transistors are switching. This is influenced by the energy needed for a switch, which depends quadratically on the supply voltage (as long as this voltage is far enough from the threshold voltage), how often the switch occurs, i.e. the frequency f within set F of applicable discrete frequency levels, and how many transistors are switching, i.e. on the code executed [57]. There are more influence factors such as temperature, but for simplification we assume that minimum possible voltage for a given frequency (and vice versa) are linearly related, and that for compute intensive tasks, the instruction mix is such that we know the average number of transistors switching. Therefore, ignoring the constants, we get

$$P_d(f) = f^3 .$$

Static power is consumed because of leakage current due to imperfect realization of transistors and due to momentary shortcuts when switching both transistor parts of CMOS circuits. Static power is both dependent on frequency and on a device-specific constant κ [20]. For simplification, we express it as

$$P_s(f) = f + \kappa \cdot \min F .$$

For simplification, we assume a proportionality factor of 1 and get:

$$P_t(f) = \zeta \cdot P_d(f) + (1 - \zeta) \cdot P_s(f) ,$$

where $\zeta \in [0; 1]$ expresses the relative importance of dynamic and static power consumption.

The energy consumed by a processor core while processing a load can be computed as the product of power consumption and time, where the power consumption is fixed, or by summing over sub-intervals in which power consumption is fix. The total energy consumed by a multicore is the sum of the core energy consumptions.

We consider a processor set P and a task set T . If a processor core runs idle, it consumes power as well. However, as long as idle power is sufficiently lower than the power under load, it might be ignored. Moreover, consider the situation that n tasks $j \in T$ with workloads of τ_j cycles each are processed at frequencies f_j on p cores before a deadline M . Then the runtime tt_j of task j is given in Eq. 5.1, and the total energy can be computed as

$$E = \sum_{j \in T} tt_j \cdot P_t(f_j) + \sum_{m \in P} z_m \cdot P_{idle} , \quad (5.10)$$

where z_m is the sum of the lengths of all idle periods on core m and P_{idle} is the idle power that we model as the dynamic power at minimal frequency:

$$P_{idle} = P_t(F_{\min}).$$

We ignore the overhead in time and energy to scale frequencies between tasks, as we assume that the task workloads are much larger than the scaling time.

If the power under load is modified to

$$\tilde{P}_t(f) = \frac{P_t(f)}{\zeta} - \frac{P_{idle}}{1 - \zeta} ,$$

then the formula for the total energy changes to

$$E = \zeta \sum_{j \in T} tt_j \tilde{P}_t(f_j) + (1 - \zeta) \left(\sum_{j \in T} tt_j P_{idle} + \sum_{m \in P} z_m P_{idle} \right) \quad (5.11)$$

$$= \zeta \sum_{j \in T} tt_j \tilde{P}_t(f_j) + (1 - \zeta) p M P_{idle} , \quad (5.12)$$

because the sum of the task runtimes and the idle periods is the total runtime of all cores. The latter part is fixed for given p and M and hence not relevant when energy is to be optimized.

5.5.2 Crown Configuration

In this section, we give the details of a generalized Crown Scheduler for any integer number p of cores; we refer as *classic Crown Scheduling* the technique described by Melot et al. [108] in Sec 5.2, i.e., an integrated Crown Scheduler with a balanced binary core decomposition for $p = 2^L$ cores.

Let $P = \{1..p\}$ be the set of all cores and $G_1 = P$ be the root group. We recursively divide a group G_i into smaller subgroups. In each recursion step, we create a set of $b \geq 2$ subgroups that we call G_i^s , where $\forall g_1, g_2 \in G_i^s : g_1 \neq g_2$, we have $g_1 \cap g_2 = \emptyset$ and $\bigcup_{g \in G_i^s} g = G_i$. We can represent a recursive decomposition with a tree of processor groups, where the root is $G_1 = P$ and the children of a group are the disjoint subsets that form its subgroups.

Figure 5.8: The core partitioning of a classic Crown Scheduler and the corresponding decomposition and group matrices.

We call A the binary *decomposition matrix* that represents a decomposition of processor groups. $A_{i_1, i_2} = 1$ if group i_2 is a direct or indirect descendant of group i_1 and zero otherwise. If group $i_1 = 1$ is the root, then $A_{1, i_2} = 1$ for all $i_2 > 1$ because group i_2 is a direct or indirect descendant of group $i_1 = 1$ and $A_{i_1, 1} = 0$ for all i_1 because group $i_2 = 1$ is the descendant of no other group. Classic Crown Scheduling assumes $p = 2^L$ for any positive integer L , $b = 2$ for each of the $\log_2 p$ recursion steps, $2p - 1$ groups such that $|g_1| = |g_2| = |G_i|/2$ for all $g_1, g_2 \subset G_i^s$ with $g_1 \cap g_2 = \emptyset$. Its decomposition matrix is shown in Fig. 5.8b.

We can represent the set of cores that belong to each group with the binary *group matrix* $B_{i, q}$, where $B_{i, q} = 1$ if processor q belongs to group G_i and 0 otherwise. Likewise, the group matrix of a classic Crown Scheduler is shown in Fig. 5.8c. From group and decomposition matrices, it is trivial to compute the set of cores G_i for each group i as well as the set of groups C_m that contains core m . Therefore, the rest of our generalized Crown Scheduler does not differ from the classic one.

Based on this decomposition scheme, we provide 3 variants of Crown schedulers. We call *balanced binary* a Crown Scheduler with a balanced binary decomposition. The only difference between this variant and the classic Crown scheduler is the ability the binary crown scheduler to compute schedules for any number of cores. The balanced binary configuration consists of $2p - 1$ configuration steps organized in a binary tree. For the root node, we divide all p cores into 2 groups of $\lfloor p/2 \rfloor$ and $\lceil p/2 \rceil$ cores, respectively, and we recursively repeat this operation until we obtain groups of 1 core. We provide the *organ* binary crown scheduler, that splits recursively every

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) Decomposition matrix of an organ Crown Scheduler for 4 cores. (b) Group matrix of an organ Crown Scheduler for 4 cores.

Figure 5.9: Decomposition and group matrices of an organ Crown Scheduler.

processor group of p cores into one group of 1 core and another of $p - 1$ cores (Fig. 5.9). Finally, we provide the *taskgraph-configured* crown scheduler, that computes the desired size of groups after properties of the task collection to schedule.

For the taskgraph-configured Crown Scheduler, we begin by computing the set Q that contains, for each task, the preferred number of cores to run it. For a task j , the preferred number of cores $q_{j,best} \in \{1.. \min(p, W_j)\}$ is the value q_j that maximizes $e_j(q_j)/tt_j(q_j, 1)$, i.e., the number of cores that yields a high efficiency at low execution time for task j . The set Q (Eq. 5.13) holds all preferred group sizes for all tasks. This takes at most $n \cdot p$ times steps for n tasks and p processors. Then, we sort decreasingly the values in Q in $O(n \cdot \log n)$ time steps. Finally, we decompose all p cores into $2p - 1$ groups as follows: if $\max Q \geq \lfloor p/2 \rfloor$, then we create a group of size $s = \max Q$ and we remove s from Q ; otherwise, we create a group of size $s = \lfloor p/2 \rfloor$. We create another group with the remaining $p - s$ cores and we repeat the process recursively with both groups we created. Since we have p cores, we have at most $p < 2p - 1$ different preferred group sizes and since we can create at most $2p - 1$ groups¹⁰, this step takes at most $2p - 1$ time steps.

$$Q = \bigcup_{j \in T} \max_{q \in 1.. \min(p, W_j)} \frac{tt_j(q, 1)}{e_j(q)} \quad (5.13)$$

5.5.3 Dynamic Crown Rescaling

The above methods for crown resource allocation, mapping and scaling optimize for the steady state of the pipeline and in particular for the (“worst”) case that all tasks are data ready and execute in each round. In practice, however, there might be cases where individual tasks are not yet data ready

¹⁰We can obtain $2p - 1$ groups by decomposing every groups into exactly 2 subgroups. Any $b > 2$ at any decomposition step leads to a smaller final number of groups, regardless of the core distribution into subgroups. This can be proven for $p = 2^1$ cores and by induction as well as by proving that all decomposition step producing a constant amount b of direct subgroups, however randomly imbalanced at each recursion step, produce the same total amount of subgroups.

at the issue time and thus are to be skipped for the current round by the crown scheduler, or where tasks are delayed due to cache misses, branch mispredictions or unexpected I/O latency. If the skipped or delayed task j was mapped to a group (even a singleton group) G_i , for instance the black hashed task in Fig. 5.4a, we call β the additional execution time of group i due to variable delays or tasks skipped. If task j is skipped, then the additional execution time is the negative opposite of tt_j ($\beta = -tt_j < 0$).

If $\beta < 0$ we can save more energy or if $\beta > 0$, the schedule may not meet the target makespan M . We can rescale “down-crown” all $n' \leq n$ remaining tasks of the processors in G_i that follow j in the crown schedule (gray hashed area in Fig. 5.4a). Note that such rescaling must be consistent across all processors sharing a moldable task, and hence the concurrent search for candidates to rescale must be deterministic. The crown structure can be leveraged for this purpose. Note that if a task of the root group is skipped or delayed, then the entire schedule needs to be rescaled.

Dynamic rescaling of the current round is an on-line optimization problem that should be solved quickly such that its expected overhead remains small in relation to the saved amount of time. Hence, simple heuristics should be appropriate here. Let R_i be the set of tasks in group i already executed ($j \notin R_i$). We can apply the Height heuristic (Sec. 5.3.5) with the subgroup G_i running tasks in R_i as the root group and target the makespan

$$M' = M - tt_j - \beta - \sum_{x=1}^{\lfloor \log_2 i \rfloor} gt_{\lfloor i/2^x \rfloor} - \sum_{j' \in R_i} tt_{j'}$$

for $p' = |G_i|$ processors ($p' \leq p$) and $n' < n$ tasks, taking $O(|F|p'n' \log n')$ time steps.

5.5.4 Core Consolidation

Even if we specify the idle periods explicitly and optimize with Eq. (5.10), energy consumption remains the same for different placements of tasks as long as the frequencies are not changed. For example, both schedules in Fig. 5.10 consume the same amount of energy, as each task is run at the same frequency in both schedules. In the following, we will assume $P_{idle} = \eta P_t(f_{min})$, where f_{min} is the minimum possible operating frequency. This is a simplification, as even in idle mode, some transistors are used and thus $P_d > 0$, but it gives a good lower bound.

Another possible view on idle periods is that cores might enter a sleep-mode during such periods, and hence the energy consumption of idle phases might be neglected. However, the times to bring an idle core into a sleep-mode and to wake it up again are much larger than frequency scaling times, and hence cannot be ignored or is not even feasible in many applications. Yet, one possibility to take advantage of sleep-modes is core consolidation, where tasks are packed onto fewer cores, and unused cores are not switched

Figure 5.10: Different schedules for the same task set.

on at all. E.g. in the schedules of Fig. 5.10, one core could remain switched off in the right schedule, while all cores would be used in the left schedule. Thus, if idle power is counted and if core consolidation is possible, the energy consumption would be lower in the right case: Eq. (5.12) would change in the sense that p in the last term would be replaced by $p - u$, where u is the number of switched-off cores. The sections below give an extension of our integrated ILP Crown Scheduler (Sec. 5.4) and of our heuristics for core consolidation.

Optimal Core Consolidation

Sec. 5.5.1 gives a possible model taking the energy spent by idle cores into account. To explore whether core consolidation is helpful, we additionally use p binary variables u_m where $u_m = 0$ if and only if core m is not used at all.

Because of Eq. 5.1, the energy to be minimized, is

$$E = \sum_{j \in T, i \in G, k \in F} x_{i,k,j} \cdot \frac{\tau_j \cdot P_t(F_k)}{w_j \cdot F_k \cdot e_i(w_j)}. \quad (5.14)$$

Then the target function can be derived from Eq. (5.12) by adding idle power as in Eq. (5.14) but subtracting from p all unused cores:

$$E_{cons} = \sum_{j \in T, i \in G, k \in F} x_{i,k,j} \cdot \frac{\tau_j \cdot (P_t(F_k) - P_{idle})}{w_j \cdot F_k \cdot e_i(w_j)} + (p - \sum_m (1 - u_m)) \cdot P_{idle}. \quad (5.15)$$

In order to set the u_m variables correctly, we need further constraints. If a core m is used, then u_m must be forced to 1:

$$\forall m \in P, j \in T, k \in F : \sum_{i \in C_m} u_m \geq \sum_{i \in C_m} x_{j,i,k}$$

If any task i is mapped (at any frequency) to a processor group that uses m , then the sum is larger than zero, and hence the right-hand side is less than 1. Multiplying by $1/n$ ensures that the right-hand side can never be less than 0.

Forcing u_m to 0 if core m is not used, is not strictly necessary, as an optimal solution will set as many u_m as possible to 0 to minimize energy. Yet, to also have feasible non-optimal schedules (e.g. in case of time-out), one can set

$$\forall m \in P : u_m \leq \sum_{i \in C_m, j \in T, k \in F} x_{j,i,k} \quad (5.16)$$

Core Consolidation Heuristic

This section describes our approach to implement core consolidation in fast heuristics. We take as starting point the integrated Crown Scheduler heuristics based on *binary search* and *simulated annealing* described in Sec. 5.3.6 and 5.3.7.

Our approach for core consolidation heuristics consists in adding up to $p-1$ sequential mock-up tasks to the task collection and compute a schedule, not taking into account the contribution of mock-up tasks when computing the total energy consumption corresponding to the schedule. Each mock-up task simulates a work of $M \cdot \max F$ units, which forces schedulers to schedule them alone on a dedicated core. We begin by computing a schedule with no mock-up task, then we progressively add up to $p-1$ tasks and we keep the best quality schedule produced in the process, as defined by Eq. 5.12, where mock-up tasks are ignored to evaluate energy consumption.

We apply this principle to run $p-1$ times either the binary search or simulated annealing heuristics. For binary search, we use 0.5 as the initial minimal efficiency constraint value. Let $\Delta_1 = 0.25$ be the initial increment or decrement of the minimal efficiency constraint we use through the search. This yields a worst-case time complexity of $O(p \cdot \lceil \log_2(\Delta_1/g) \rceil \cdot np(\log p + |F| \cdot \log n))$. For simulated annealing, we use a initial temperature of 9, a final temperature of 0.9 and 0.6 as cooling factor. Finally, for the initial binary search solution, we use the same initial minimal efficiency constraint value (0.5) and search speed ($\Delta = 0.25$). As in Sec. 5.3.7, we define the distance between two neighbor solutions as a continuous value between 0 and 1, as the number of differences in tasks' individual core allocation, and the difference amplitude. The resulting time complexity is therefore $O(np^2(\log p + |F| \cdot \log n + \lceil \log_2(\Delta/g) \rceil))$ for our core-consolidated simulated annealing.

5.5.5 Island-aware Crown Scheduler

Existing target architectures often organize voltage and frequency scaling capabilities into *islands*, within which every cores always run at the same voltage or frequency level, respectively (see Sec. 2.3.1). In this section, we take profit of the recursive decomposition of Crown Scheduling to extend our

Figure 5.11: Platform with unbalanced island structure and the corresponding crown configuration for island-aware scheduling. Here $N = \{G_2, G_3\}$.

ILP-based integrated implementation with constraints that take voltage and frequency islands into account. As power consumption is mostly dependent on voltages (see Sec. 2.3.1), we consider voltage scaling only and we always use the fastest frequency available for a given voltage level. Therefore we focus on voltage islands only.

For a target architecture of p cores, we consider l sets of cores $H_1 \dots H_l$ to model islands, where $\bigcup_{r=1}^l H_r = P$. Note that islands do not necessarily include the same number of cores. Also, we have R_m the set of islands that contains processor m ; obviously, we have $\forall m \in 1..p : |R_m| = 1$, i.e., such a processor belongs to exactly one island. We define $M_i = \bigcup_{m \in G_i} R_m$ as the set of all islands any core in group i is a member. N is the set of groups for which the set of cores matches exactly the union of 1 or more frequency islands (Eq. 5.17); see Fig. 5.11b for an example. Finally, O_i is the set of descendant groups i according to the decomposition matrix A such as 5.8b or 5.9a (Eq. 5.18).

$$N = \bigcup_{i=1}^{|G|} \begin{cases} i & \text{if } \exists r \in 1..l : H_r = G_i \\ \emptyset & \text{otherwise} \end{cases} \quad (5.17)$$

$$O_i = \bigcup_{i' \in G : i' \neq i} \begin{cases} i' & \text{if } A_{i,i'} = 1 \\ \emptyset & \text{otherwise} \end{cases} \quad (5.18)$$

We define the binary variable $s_{i,f}$ and we force it to 1 if at least one task mapped to i or one of its subgroups is *not* set to f (Eq. 5.19)

$$s_{i \in N, f \in F} \geq \sum_{i' \in O_i} \sum_{j' \in T} \sum_{f' \in F : f' \neq f} \frac{x_{i',j,f'}}{n}, \quad (5.19)$$

and we make sure that for all group $i \in N$, the amount of frequency levels *not* assigned for any tasks in i or any of its subgroup, is exactly $|F| - 1$ (Eq. 5.20), i.e., the number of different frequency levels assigned for any

task in i or one of its subgroup is exactly 1; this is possible only if all tasks mapped to group i or one of its subgroups are assigned the same frequency level.

$$\forall i \in N : \sum_{f \in F} s_{i,f} = |F| - 1. \quad (5.20)$$

With Crown Consolidation (Sec. 5.5.4), we switch cores off to improve energy consumption. When a core is switched off, its voltage supply is cut to save the associated energy. However, this must be done at the voltage island level, therefore affecting all cores in the island. We add another constraint to u_m (see Sec. 5.5.4)

$$\forall r \in 1..l, m \in H_l, m' \in H_l : m \neq m' \wedge u_m = u_{m'}.$$

Let us consider a 4 cores target platform with 2 islands of 2 cores each, with the frequency set F is $\{l, h\}$, where $l < h$. We have a task graph of 3 sequential tasks of workload $\tau = h$ and deadline $M = 1$. Since one of the 4 cores is unused, constraint 5.16 forces our island-aware Crown Scheduler to switch off the other core in the same island; however, this requires to move the task it runs to one of the remaining 2 cores and they cannot fulfill the deadline constraint. In order to make our island-aware Crown Scheduler able to find a schedule for this problem instance, we need to waive constraint 5.16. This does not affect our scheduler's ability to switch cores off since, as discussed in Sec 5.5.4, it would try to switch off as many cores as possible to minimize energy consumption. However, in an island where at least one core runs at least one task, all unused cores cannot be switched off and therefore consume idle energy.

Note that a Crown Configuration is valid for a platform's island topology iff for each island r in the target platform and for each group G_i , all cores in G_i are also in M_i and all cores in M_i are also in G_i (i.e., G_i and M_i are equivalent), unless all cores in group G_i belong to a unique island (Eq. 5.21), i.e., group i is a subset of an island (groups 2 to 11 in Fig. 5.11b).

$$\forall i \in 1..|G| : G_i = \bigcup_{r \in M_i} H_r \vee |M_i| = 1 \quad (5.21)$$

In order to enforce a correct crown configuration, we modify the balanced binary configuration of Sec. 5.5.2 so that it creates exactly one group for each island of the target platform. For each recursive decomposition, we find the island that includes the first core we can allocate into the first subgroup; all cores of this island must be available to allocate a group. Let us consider that at iteration i , we have p_i cores to distribute into 2 subgroups and the first core we can allocation belongs to island H_r . We count iteratively $q = \sum_{j=r}^{r' \leq l} H_l$ such that $r \leq r' \leq l$, q is maximized and $q < P_i/2$. We allocate $|\bigcup_{j \in r..r'} H_j|$ cores to one subgroup, the remaining cores to the second group and we continue this process recursively for each subgroup until we create subgroups of 1 core.

5.6 Restriction-Free Optimal Scheduler

In this section, we describe an ILP formulation for an unrestricted, optimal scheduler to co-optimize core allocation, mapping and frequency scaling for streaming tasks under a makespan constraint with core consolidation. We first give our formulation, then we give a mathematical argument that shows that our scheduler produces solutions at best 3.7 times better than a crown scheduler, for a realistic execution platform model.

5.6.1 ILP formulation

The total energy consumption of a schedule with a given deadline can be achieved by summing up the energy consumption of all tasks and idle times. The energy consumption for one task is its average power consumption multiplied by its runtime, where the power depends on the frequency and the runtime depends on the frequency and on the number of cores used. The energy consumption for the idle times is the idle power consumption multiplied by the idle times. The optimal unrestricted scheduler is based on an integer linear program. To minimize the total energy consumption of a schedule the program uses $n \cdot p \cdot |F|$ binary variables $z_{j,q,k}$ with $z_{j,q,k} = 1$ if and only if task j is executed on q cores with frequency F_k . Then the runtime of a task j can be expressed by

$$r_j = \sum_{q,k} z_{j,q,k} \cdot \frac{\tau_j}{F_k \cdot e_j(q)}$$

and thus the energy to be minimized is

$$E = \sum_{j \in T, q \in \{1..p\}, k \in F} z_{j,q,k} \cdot \frac{\tau_j \cdot P(F_k)}{F_k \cdot e_t(q)} + t_{idle} \cdot P_{idle}.$$

In order to obtain feasible schedules, we formulate the following constraints: We start with some general restrictions to a task. A task can start at the earliest when the schedule begins and has to be finished before the deadline. Thus the start time s_j of a task j has to be higher than zero and its end time o_j can not be higher than the deadline:

$$\forall j \in T \quad s_j \geq 0,$$

$$\forall j \in T \quad o_j \leq M.$$

Within this range, the end time of a task depends on the start time plus the parallel (if necessary) workload. This can be achieved by

$$\forall j \in T \quad o_j = s_j + \sum_{q=1}^p \sum_{k \in F} z_{j,q,k} \cdot \frac{\tau_j}{F_k \cdot q \cdot e_t(q)},$$

where q represents the number of cores that are used to execute the task. Thus each task must use exactly as many cores, at one frequency, as indicated by the z -variable

$$\forall j \in T, k \in F \quad \sum_{m \in P} x_{j,m,k} = \sum_{q=1}^p q \cdot z_{j,q,k}$$

where $x_{j,m,k}$ are $n \cdot p \cdot |F|$ binary variables with $x_{j,m,k} = 1$ if and only if task j is running on core m with frequency F_k .

Additionally a task can just be executed on exactly one number of cores at a certain frequency:

$$\forall j \in T \quad \sum_{q=1}^p \sum_{k \in F} z_{j,q,k} = 1.$$

If we now focus on the possible ordering of two tasks, then we need some further constraints. To indicate whether a task j_1 precedes a task j_2 we use n^2 binary variables $y_{j_1,j_2} = 1$ if and only if j_1 precedes j_2 on all common cores. One task cannot precede itself:

$$\forall j \in T \quad y_{j,j} = 0.$$

Either task j_1 precedes task j_2 , or vice versa (or neither), but not both. This can be expressed by:

$$\forall j_1, j_2 \in T \quad y_{j_1,j_2} + y_{j_2,j_1} \leq 1.$$

If both tasks j_1 and j_2 use a common core (and $j_1 \neq j_2$), then j_1 precedes j_2 or vice versa:

$$\begin{aligned} \forall j_1, j_2 \in T : j_1 \neq j_2, \forall m \in P \\ y_{j_1,j_2} + y_{j_2,j_1} \geq -1 + \sum_{k \in F} (x_{j_2,m,k} + x_{j_1,m,k}), \end{aligned}$$

If tasks j_1 and j_2 both use a common core, and j_1 precedes j_2 , then j_2 has to be executed after j_1 :

$$\begin{aligned} \forall j_1, j_2 \in T \\ s_{j_2} \geq s_{j_1} + \sum_{q=1}^p \sum_{k \in F} z_{j_1,q,k} \cdot \frac{\tau_{j_1}}{(F_k \cdot q \cdot e_{j_1}(q))} - (1 - y_{j_1,j_2} \cdot M). \end{aligned}$$

Until now we did not take into account the idle energy. To consider the idle time for an active core m , we need to mark cores that do not run any task as unused. We set the binary variable u_m to 0 if core m is unused and to 1 otherwise:

$$\forall j \in T, m \in P, k \in F \quad u_m \geq x_{j,m,k}.$$

Then, we can compute the total idle time by deducting the sum of processing time for each task using as many cores allocated to run it in variable z , from the total time and number of processors available ($p \cdot M$). We further deduct unused cores for the complete duration of the pipeline stage. We obtain the total idle time for the whole schedule:

$$t_{idle} = p \cdot M - \sum_{j \in T, q \in 1..p, k \in F} z_{j,q,k} \frac{\tau_j}{k \cdot e_{j,q}} - M \cdot \sum_{m \in P} (1 - u_m)$$

and the total energy consumption results in:

$$E = \sum_{q=1}^p \sum_{j \in T, k \in f} z_{j,q,k} \cdot \frac{\tau_j \cdot P(F_k)}{F_k \cdot e_t(q)} + t_{idle} \cdot P_{idle}.$$

5.6.2 Cost of Crown Scheduling Structure

We now show that our unrestricted, optimal scheduler produces solutions at best 3.7 times better than Crown Schedulers, for a realistic execution platform model. Our argument involves an execution platform of $p > 2$ cores that can run at a low frequency f_{low} and a high frequency f_{high} ,^{see 11, 12} and a collection of 2 tasks j_1 and j_2 that, when running on $w_1 > \lceil p/2 \rceil$ and $w_2 = p - w_1$ cores, respectively, can be completed in M time steps at frequency f_{low} . We restrict j_1 to be able to run on at least w_1 cores and j_2 to run on w_2 cores or more. This can be achieved with very low efficiency values for j_1 and j_2 when running on less cores than the lower bound defined in the case study. Figure 5.12 shows 2 schedules for 32 cores: the optimal schedule computed by our unrestricted scheduler (Fig. 5.12a) and the worst-case schedule by a binary crown scheduler (Fig. 5.12b). With 3 or more tasks, we can add constraints as above to that a Crown Scheduler has no other alternative than scheduling all tasks to run on all p cores. If no such constraint is possible, then the quality difference is necessarily lower than our scenario because it would generate load imbalance and idle power consumption instead of dynamic power at highest frequency.

We begin with an informal description of the scenario. The optimal schedule performs all the necessary workload within the constraint M , always running at frequency f_{low} ; since there is no idle time, the workload balance is optimal and the frequency required is minimized. Also, running each task individually with more cores would results in a lower efficiency, therefore requiring a higher frequency to respect the deadline constraint. In contrast, the worst case schedule computed by a binary Crown Scheduler

¹¹Our scenario is still valid with bigger frequency sets, where there is no frequency level between f_{low} and f_{high} and $f_{high} - f_{low}$ is the smallest possible value for all f_{high} and f_{low} in F and $f_{high} \neq f_{low}$.

¹²For $1 \leq p \leq 2$, the crown structure includes all possible task allocation and mapping settings; therefore, our Crown Scheduler and an optimal, unrestricted scheduler have the same exploration space and would produce solutions of the same quality.

Figure 5.12: Best and worst-cases schedules for 32 cores and minimal frequency of 10 units. The color coding indicates clock frequency ($f_{low} = 10$ is yellow)

runs all tasks on p cores, which increases the workload due to decreasing parallel efficiency functions of task j_1 and j_2 . Because of the balanced binary decomposition, there is no groups of w_1 or exactly w_2 cores and the Crown Scheduler must schedule j_1 and j_2 to run on p cores instead. Because of the parallel efficiency loss j_1 and j_2 suffer from because they run on more cores, they cannot fit within the M makespan constraint and they are forced to run both at the higher frequency f_{high} . If f_{high} is higher than necessary, i.e., tasks j_1 and j_2 running on p cores at frequency f_{high} terminate at a time $m < M$, then this translates to an even higher energy consumption. Because every core participates to some computation, none can be switched off and all consume additional idle energy that corresponds to the idle time spent between time steps m and M . Note that the taskgraph-configured may configure the cores recursive decomposition so that a group of w_1 exists. This is only possible if

$$e_j^2(p) \cdot p < w_j \cdot e_j^2(w_j)$$

for either tasks j_1 and j_2 , i.e, if the parallel scalability of task j_1 or j_2 is so bad that the taskgraph-configured Crown Scheduler would favor a lower parallelization.

Let $0 < e_1(p) < e_1(w_1) < 1$ be the efficiency function for task j_1 , where $e_1(q)$ is very small for all $q < w_1$, and $0 < e_2(p) < e_2(w_2) < 1$ be the efficiency function for task j_2 , where $e_2(q)$ is very small for all $q < w_2$. Let M define the makespan constraint for the schedule to compute, and τ_1 and τ_2 be the workload of tasks j_1 and j_2 , respectively. We want τ_1 and τ_2 so that, given their efficiency functions, j_1 and j_2 finish at time M on w_1 and w_2 cores, respectively at frequency f_{low} , that is, for a generic task j :

$$\begin{aligned} tt_j(w_j, f_{low}) &= M, & \text{i.e.,} \\ \frac{\tau_j}{w_j \cdot e_j(w_j) \cdot f_{low}} &= M. \end{aligned} \quad (5.22)$$

Since we want to compute the workload, we can transform Eq. 5.22 into

$$\tau_j = M \cdot f_{low} \cdot w_j \cdot e_j(w_j). \quad (5.23)$$

We need f_{high} so that both tasks can run on p cores within the time limit M :

$$tt_1(w_1, f_{high}) + tt_2(w_2, f_{high}) \leq M. \quad (5.24)$$

We can transform Eq. 5.24 using Eq. 5.1 and 5.23 into

$$f_{high} \geq \frac{f_{low}}{p} \sum_{t=1}^2 \frac{r_t \cdot e_j(r_j)}{e_j(p)}. \quad (5.25)$$

However if f_{high} is too high, then the Crown Scheduler can schedule either j_1 or j_2 at frequency f_{high} and have enough time to run the second task at the low frequency f_{low} . In order to prevent that, we add the 2 constraints:

$$\begin{aligned} tt_1(p, f_{low}) + tt_2(p, f_{high}) &> M \quad \text{and} \\ tt_1(p, f_{high}) + tt_2(p, f_{low}) &> M, \end{aligned} \quad (5.26)$$

which translates to

$$f_{high} < \frac{e_1(p) \cdot f_{low} \cdot w_2 \cdot e_2(w_2)}{e_2(p)[p \cdot e_1(p) - w_1 \cdot e_1(w_1)]} \quad \text{and} \quad (5.27)$$

$$f_{high} < \frac{e_2(p) \cdot f_{low} \cdot w_1 \cdot e_1(w_1)}{e_1(p)[p \cdot e_2(p) - w_2 \cdot e_2(w_2)]} \quad (5.28)$$

where Eq. 5.27 is valid if

$$e_1(p) \geq e_1(w_1)/2 + e_1(w_1)/p$$

and has no effect otherwise, and Eq. 5.28 is valid if

$$e_2(p) \geq e_2(w_2)/2 + e_2(w_2)/p.$$

If both Eq. 5.27 and 5.28 are invalid, then f_{high} has no upper bound in our scenario. Given our energy model shown in Eq. 5.12, it is obvious that the energy consumption difference between the optimal schedule and the one computed by the Crown Scheduler is maximized when f_{high} is maximized. We will therefore use f_{high} as the highest possible value that still fits the constraints given by Eq. 5.27 and 5.28.

We now focus on the energy consumption difference between the optimal solution and the one computed by a Crown Scheduler. In both solutions, both tasks need to achieve $\tau_1 + \tau_2$ work together and both run at the same frequency level. For the optimal schedule, all p cores run for M time steps at frequency f_{low} , as defined by equations above. Since no core spends any time idle, the overall energy consumption of the optimal schedule is

$$E_{opt} = p \cdot M \cdot (\zeta \cdot f_{low}^\alpha + (1 - \zeta)(f_{low} + \kappa f_{low})). \quad (5.29)$$

For the Crown Schedule, we can reduce tasks j_1 and j_2 to a unique task h running at frequency f_{high} on p , since both j_1 and j_2 run at the same frequency f_{high} . Since h includes all the workload of tasks j_1 and j_2 including the efficiency penalty for running on p cores, then its workload is

$$\tau_h = \frac{\tau_1}{e_1(p)} + \frac{\tau_2}{e_2(p)},$$

its execution time on p processors is

$$tt_h = \frac{\tau_h}{p \cdot f_{high}}.$$

and its energy consumption due to processors' activity is

$$E_{crown}^{busy} = p \cdot tt_h(p, f_{high}) \cdot \zeta f_{high}^\alpha + (1 - \zeta)(f_{high} + \kappa f_{low}).$$

Also, the total idle energy is

$$E_{crown}^{idle} = \frac{\eta(1 - \zeta)[p \cdot M - p \cdot tt_h][\zeta f_{low}^\alpha + (1 - \zeta)(f_{low} + \kappa f_{low})]}{\zeta}$$

and the total energy consumption for the crown scheduler becomes

$$E_{crown} = E_{crown}^{busy} + E_{crown}^{idle}. \quad (5.30)$$

We can now compute the best quality improvement of our unrestricted scheduler over Crown Scheduling:

$$\frac{E_{crown}}{E_{opt}}. \quad (5.31)$$

In Fig. 5.12, we use $p = 32$ cores and energy model parameters $\alpha = 3$, $\zeta = 0.649$, $\kappa = 52.6392$ and $\eta = 0.018611$, derived from energy measurements performed on Intel Xscale architecture by Xu et al. [144]. If we take $f_{low} = 10$ then we have $13.202 \leq f_{high} < 18.549$ and we choose $f_{high} = 18.54$. If we consider a perfect task scalability, then we find that our optimal scheduler produces at best 2.79 times better solutions than a Crown Scheduler. Considering an efficiency loss for tasks j_1 and j_2 with an increasing amount of processors (we use $w_1 = p/2 + 1$ and $e_1(p/2 + 1) = 0.79167$, as well as $w_2 = p/2 - 1$ and $e_2(p/2 - 1) = 0.99167$, and $e_1(p) = e_2(p) = 0.7$), then we obtain a 3.69 difference factor. The factor decreases with ζ and η ; this is not surprising because as ζ tends to 0, the busy power difference factor become linear instead of quadratic or more. Also, when the weight of idle energy decreases, then the penalty of additional idle energy consumption decreases and the Crown Scheduler becomes more competitive. In the extreme case where α tends to 1, the difference factor also tends to 1. Finally, we can simplify the difference factor estimation in Eq. 5.31 by considering no static or idle energy consumption ($\zeta = 1$, $\eta = 0$) and a perfect parallel scalability

for tasks j_1 and j_2 ($\zeta = 1$ and $e_1(w_1) = e_2(w_2) = e_1(p) = e_2(p) = 1$), then we have

$$\begin{aligned}
 E_{crown}^{busy} &= p \cdot tt_h(p, f_{high}) \cdot f_{high}^\alpha \\
 &= p \cdot \frac{\tau_1}{p \cdot f_{high} \cdot e_1(p)} + \frac{\tau_2}{p \cdot f_{high} \cdot e_2(p)} \cdot f_{high}^\alpha \\
 &= \frac{M \cdot f_{low}}{f_{high}} \cdot (w_1 + w_2) \cdot f_{high}^\alpha \\
 &= \frac{M \cdot f_{low}}{f_{high}} \cdot p \cdot f_{high}^\alpha = p \cdot M \cdot f_{low} \cdot f_{high}^{\alpha-1},
 \end{aligned}$$

we reduce E_{crown}^{idle} to 0, and

$$E_{opt} = p \cdot M \cdot f_{low}^\alpha$$

and we can reduce Eq. 5.31 to

$$\frac{E_{crown}}{E_{opt}} = \frac{p \cdot M \cdot f_{low} \cdot f_{high}^{\alpha-1}}{p \cdot M \cdot f_{low}^\alpha} = \frac{f_{high}^{\alpha-1}}{f_{low}^{\alpha-1}}. \quad (5.32)$$

Using the energy model derived from Xu et al. [144], we obtain a factor 3.44. Comparing with the value of 3.69 obtained taking static and idle values into account, we see that α is by far the most dominant parameter to estimate the benefits of our technique over Crown Scheduling, and that the difference ratio admits no upper bound depending on the parameters.

5.7 Experimental Evaluation

In this section, we evaluate the optimization time, complexity and solution quality produced by our Crown Schedulers and their extensions described in Sec. 5.2.

For small and medium problems and machine sizes, we generate synthetic task collections as follows: we group synthetic task collections in different categories defined by the number of cores ($p \in \{1, 2, 4, 8, 16, 32\}$), the number of tasks (10, 20, 40 and 80 tasks), and tasks' maximum widths W_t : sequential ($W_t = 1$ for all t), low ($1 \leq W_t \leq p/2$), average ($p/4 \leq W_t \leq 3p/4$), high ($p/2 \leq W_t \leq p$) and random ($1 \leq W_t \leq p$). Tasks' maximum width are distributed uniformly.

We also provide task collections of classic streaming algorithms: parallel FFT, parallel-reduction and parallel mergesort. FFT is characterized by $2p - 1$ parallel tasks of a balanced binary tree. In level $l \in [0; \log_2 p]$ of the tree, there are 2^l data-parallel tasks of width $p/2^l$ and work $p/2^l$ so all tasks could run in constant time. The mergesort task collection is similar to FFT, but all its tasks are sequential. Parallel reduction involves $\log_2 p + 1$ tasks of maximum width 2^l and work 2^l for $l \in [1; \log_2 p + 1]$; they can also run in constant time.

Finally, we use the technique of Gordon et al. [48] to extract tasks' workload and parallel efficiency from the Streamit benchmark suite. We schedule all variants of the applications *audiobeam*, *beamformer*, *channelvocoder*, *fir*, *nokia*, *vocoder*, *BubbleSort*, *filterbank*, *perftest* and *tconvolve* from the Streamit compile source package¹³. We use their compiler to obtain for each task j the task's estimated workload τ_j , the task's parallel degree W_j (either 1 or $+\infty$) and the task's communication rate we call ψ_j . We use ψ_j to compute tasks j 's parallel efficiency $e_j(q)$ (Eq. 5.33).

$$e_j(q) = \begin{cases} 1 & \text{if } q = 1 \\ \tau_j / (\tau_j + q \cdot \psi_j) & \text{if } q > 1 \text{ and } q \leq W_j \\ 10^{-6} & \text{otherwise} \end{cases} \quad (5.33)$$

and we use Eq. 5.34 to compute a target makespan.

5.7.1 Crown Scheduling with no Idle Energy

We implemented the ILP models in AMPL [41], the LTLG mapping, the height frequency scaling, binary search and simulated annealing allocation heuristics in C++. We substitute the ILP-based mapping and frequency scaling phases of the phase-separated variant [83] with an optimal load-balanced ILP-formulated mapping phase, the LTLG mapping, and with the Height mapping heuristics, respectively. We obtain and compare 7 different crown schedulers:

1. ILP allocation, ILP load-balanced mapping, ILP frequency scaling (Fast, Bal.ILP, ILP)
2. ILP allocation, LTLG mapping heuristic, ILP frequency scaling (Fast, LTLG, ILP)
3. ILP allocation, ILP load-balanced mapping, Height heuristic frequency scaling (Fast, Bal.ILP, Height)
4. ILP allocation, LTLG mapping heuristic, Height frequency scaling heuristic (Fast, LTLG, Height)
5. Binary search with LTLG mapping and Height frequency scaling heuristics (Bin, LTLG, Height)
6. Simulated annealing allocation with initial binary search, LTLG mapping and Height frequency scaling heuristics (Bin, LTLG, Height Ann.)
7. ILP integrated (Integ.)

We also implemented an ILP formulation from Xu et al. [143], whose scheduler arranges tasks in levels of strips spanning across cores. Each level's height is defined by a *key* task, that is the first task mapped to its level. The rectangle of cores unused by the key task of a level is 2D-packed using some of the tasks not scheduled yet. Frequency scaling is performed through

¹³At the time of writing, the latest version was pushed to github on Oct. 4, 2016. See <http://groups.csail.mit.edu/cag/streamit/restricted/files.shtml> and the source package for information about these applications.

the rescaling of entire levels, that is the rescaling of all tasks mapped to this level, as a bin packing problem. Finally, we adapted and implemented the NLP (Non-Linear Programming) formulation from Pruhs et al. [116] in AMPL and their heuristic in C++. Their technique schedules sequential tasks only (and therefore it never parallelizes any task j regardless of its maximal parallel degree W_j) and scales tasks' frequency with a continuous and unbounded frequency domain. For the NLP implementation, we perform frequency scaling using our ILP-based method (Sec. 5.3.4). For the heuristic, we use best-fit to assign discrete and bounded frequency levels to tasks.

The target makespan of each synthetic task collection is the mean value between the runtime of an ideally load balanced task collection running at lowest frequency and at highest frequency

$$M_{syn} = \sum_{j=1}^n \frac{3 \cdot tt_j(1, 1)}{8 \min(W_j, p) F_{\min}} + \frac{tt_j(\min(W_j, p), 1)}{8 \cdot F_{\min}} + \frac{3 \cdot tt_j(1, 1)}{8 \min(W_j, p) F_{\max}} + \frac{tt_j(\min(W_j, p), 1)}{8 \cdot F_{\max}} \quad (5.34)$$

We use the same scheme to generate large problems and machine sizes, ranging from 500 to 2000 tasks and from 256 to 1024 processors for tasks of a random maximum width.

Because FFT, Mergesort and Parallel Reduction task collections yield perfectly balanced task work and maximum widths, they are easy to schedule with a loose target makespan as defined for the synthetic task collections. We use a more constraining target makespan for concrete task collections. The makespan for FFT is

$$M_{FFT} = \sum_{j=2}^n \frac{tt_j(1, F_{\min})}{2 \cdot p} + \sum_{j=2}^n \frac{tt_j(1, F_{\max})}{2 \cdot p} \quad (5.35)$$

that is, the makespan of a synthetic task collection, ignoring task 1. The makespan for parallel mergesort

$$M_{msort} = \frac{\max_j(\tau_j)}{\max F} \quad (5.36)$$

takes the root merging task and divide its work by the maximal frequency available. The makespan for parallel reduction task collections

$$M_{produce} = \sum_{j=1}^n \frac{\tau_j}{2 \cdot \min(p, W_j) \cdot F_{\min}} + \sum_{j=1}^n \frac{\tau_j}{2 \cdot \min(p, W_j) \cdot F_{\max}} \quad (5.37)$$

is the same as for synthetic collections, but we assume perfect scalability. All our synthetic and classic task collections have a minimal efficiency gap of

$g = 0.023438$, yielding a complexity factor of $\gamma = \lceil \log_2(0.5/0.023438) \rceil = 5$ for our binary search method.

We run both the ILP solver and heuristics on a quad-core i7 Sandy Bridge processor at 3 GHz and 8 GiB of main memory. We use Gurobi 5.10 and ILOG AMPL 10.100 with 5 minutes timeout to solve the ILP models and we compile our C++ implementations with GCC 4.6.3 and -O3 optimization switch, on Ubuntu 12.04. Gurobi exploits 8 hardware threads and our heuristics are implemented as sequential programs.

We measure the overall scheduling quality (energy consumption) of all 7 crown schedulers as well as schedulers from Xu et al. [143] and Pruhs et al. [116]. Figures 5.13a and 5.13b show mean values for 10, 20, 40 and 80 tasks and across all width classes $\{Random, Serial, Low, Average, High\}$ for the synthetic task collection. Figures 5.16 and 5.17b show the average energy consumption by task collection classes for the classic streaming algorithms and the Streamit benchmark, respectively. We can see that the integrated crown scheduler as well as scheduler heuristics based on binary search, LTLG, Height and simulated annealing combined produce consistently the best schedules. NLP and heuristics from Pruhs et al. [116] perform well on larger architectures and with sequential task sets (Figs. 5.13b and 5.16), but its solutions' quality drop when tasks' parallel degree increase (Figs. 5.13a, and 5.16). The level-packing solution from Xu et al. [143] performs mostly poorly, except for small amount of tasks, or for sequential tasks. An exception is parallel reduction (Fig. 5.16) for which it performs at least as good as our integrated, ILP-based crown scheduler. We scheduled a subset of our synthetic task collection set with random workload variations within 1% for each tasks and found no significant energy variations in schedules. Finally, we show for the classic streaming task collection and the Streamit benchmark suite, the schedulers' ability to compute a valid schedule in Figs. 5.15 and 5.17a. These figures show that Pruhs' techniques fail at scheduling parallel tasks under the makespan constraints we impose, while Xu's succeeds as much as our crown schedulers. For sequential task collections such as for mergesort, our crown scheduler has the same success rate as Pruhs', whereas Xu's approach fails more often.

We also compare the makespan produced by our LTLG mapping heuristic to our load-balanced ILP mapping formulations (variant 1 above) as well as our binary search and simulated annealing heuristics and our implementations of Xu's and Pruhs' techniques (Fig 5.14a). Similarly, we compare in Fig 5.14b the efficiency of the frequency scaling ILP formulation and Height heuristic to optimize energy through frequency scaling. In order to ensure fairness, we use our ILP formulations to compute both mappings the frequency scaling implementations we compare need to scale. Figure 5.14a shows that our LTLG heuristic is very competitive with our ILP load-balancing formulation. The same figure suggests that the initial allocation influences greatly the mapping makespan. Our ILP and LTLG heuristic use the fast allocation (Sec. 5.3.1), that is allocating as many processors as

(a) Energy prediction by task collection class (in multiples of joules).

(b) Energy prediction by number of tasks (in multiples of joules).

Figure 5.13: Projected energy per task collection class and number tasks.

(a) Mapping makespan by task collection class (in multiples of seconds).

(b) Frequency scaling energy by task collection class (in multiples of joules).

Figure 5.14: Efficiency of LTLG and Height heuristics for the synthetic task collection.

Figure 5.15: Scheduling success rate per task collection class.

Figure 5.16: Energy prediction by task collection class (in multiples of joules).

(a) Scheduling success rate per task collection class.

(b) Energy prediction by task collection class (in multiples of joules, logarithmic scale).

Figure 5.17: Scheduling success rate and projected energy for FFT, merge-sort and parallel-reduction

possible to tasks. In contrast, Pruhs' NLP formulation and heuristic schedule sequential tasks. As a consequence, Pruhs' makespan are much lower for tasks that cannot run on many processors (but can run on a few) while the situation is more balanced with highly parallel tasks. The binary search always finds another allocation that allows lower makespan than either our crown heuristic with fast allocation or Pruhs' solutions. Again, the approach from Xu et al. [143] yields much higher makespans. Figure 5.14b demonstrates the efficiency of our Height heuristic, compared to our ILP-based frequency scaling method.

We provide the measured overall optimization time for all scheduler variants for the synthetic task collection set by number of tasks and number of cores (Figs. 5.20a and 5.20b), as well as the optimization time for both mapping and frequency scaling phases (Figs. 5.18a, 5.19a, 5.18b and 5.19b). We compare the measured optimization time of our heuristics and their time complexity by number of tasks and number of cores in Figs. 5.21, 5.22, 5.23 and 5.24 for small, medium and large problems of the synthetic task set. We can see that Pruhs, Xu's and our crown integrated ILP and NLP formulations largely dominate the overall optimization time. Phase-separated, crown-optimal ILP crown schedulers yield a much lower optimization time and none of the heuristics is visible in Figs. 5.20a and 5.20b. The 6 figures in Fig. 5.19 demonstrate the LTLG and Height heuristics run much faster than their crown-optimal ILP counterparts, and that the phase-separated crown schedulers' optimization time is dominated by the mapping phase. Figures 5.21a, 5.21b, 5.23a and 5.23b demonstrate that the optimization time for heuristics are dominated by the simulated annealing and our binary search crown schedulers. The phase-separated LTLG and Height crown heuristic as well as Pruhs' heuristic perform much faster. This can be explained by their lack of an elaborate allocation phase that our binary search and simulated annealing implement as a feedback loop. However, such simplistic allocation strategy makes them to produce worse solution, or to fail at producing any valid one. Finally, Figs. 5.21a, 5.21b, 5.22a and 5.22b show that our heuristics's optimization time follow the optimization time predicted by our complexity analysis. However, this is not true for our simulated annealing and binary search with the large synthetic task collection and target architectures (Figs. 5.23a, 5.23b, 5.24a and 5.24b). This can be due to the search interruption when a valid schedule is found where all tasks run sequentially at the lowest frequency available.

The crown constraints exhibits no restrictions to map sequential tasks to processors, but it allows the mapping of some parallel tasks. Hence, the solution space of Pruhs' mapping technique is strictly included in our crown mapping's solution space. Also, since we implemented the same ILP frequency scaling method for both schedulers, the overall solution space of crown schedulers includes the one of our implementation of Pruhs'. Therefore, when both formulations can be solved to optimality, the integrated crown scheduler will always find a solution at least as good as Pruhs'. Sim-

(a) Mapping phase runtime by number of tasks.

(b) Frequency scaling phase runtime by number of tasks.

Figure 5.18: Mapping and frequency scaling phases by number of tasks.

(a) Mapping phase runtime by number of cores.

(b) Frequency scaling phase runtime by number of cores.

Figure 5.19: Mapping and frequency scaling phases by number of cores.

(a) Overall optimization time by number of tasks.

(b) Overall optimization time by number of cores.

Figure 5.20: Overall optimization time by number of tasks and cores.

(a) Overall optimization time per number of tasks.

(b) Overall optimization time per number of cores.

Figure 5.21: Measured optimization time and time complexity of schedulers

(a) Overall complexity per number of tasks (in number of steps).

(b) Overall complexity per number of cores (in number of steps).

Figure 5.22: Measured optimization time and time complexity of schedulers

(a) Overall optimization time by number of tasks.

(b) Overall optimization time by number of cores.

Figure 5.23: Overall scheduling time and energy consumption of the resulting crown schedules for large task collections and massively parallel architectures

(a) Overall complexity by number of tasks (in number of steps).

(b) Overall complexity by number of cores (in number of steps).

Figure 5.24: Overall scheduling time and energy consumption of the resulting crown schedules for large task collections and massively parallel architectures

ilarly, our LTLG heuristic for sequential tasks is reducible to LPT. Since we use LPT in our implementation of Pruhs’ heuristic, its solutions cannot be better than LTLG’s. Again, the Height heuristic is an implementation of Best-fit for the crown structure; therefore it has the same solution space as Pruhs’ frequency scaling method. Since our LTLG and Height heuristics are reducible to Pruhs’ mapping and frequency scaling methods, they can always find a solution at least as good.

We inspected schedules of the synthetic task sets produced by our best crown scheduler heuristic as well as the technique described by Pruhs et al. [116]. We assume that tasks in our synthetic task set run sequentially and at the lower frequency available for between 1 ms and 19 ms, that is, $8\mu\text{s}$ with 32 cores at frequency 5. We assume further that switching frequency takes $50\mu\text{s}$ [101] and $5 \cdot 10^{-6}$ energy units. Finally, we assume that frequency switching can take place asynchronously, while a task is running [101]. We found that all frequency transitions can be hidden when tasks are processed or when processors are idle. In the case of a task running for a too short time to hide a necessary frequency transition, no significant additional time and energy was found.

5.7.2 Crown vs Non-Crown

In this section, we compare the balanced binary integrated Crown Scheduler described in Sec. 5.2 to Crown Schedulers that can be configured as shown in Sec. 5.5.2 and to our optimal scheduler with no constraints at all (Sec. 5.6). We use the same task collection and target Makespan as described in Sec. 5.7.1. We run both our ILP-based solution and heuristics on a quad-core i7 Sandy Bridge processor at 3GHz with 8GB of main memory. We compile our C++ heuristics with GCC 4.8.2 and the -O3 optimization switch with Ubuntu Trusty (14.04 LTS) and we use Gurobi 6.0.0 and AMPL 10.100 with a 5 minutes timeout to solve scheduling instances with our ILP-based schedulers.

Figure 5.25 shows the ratio of problems for which Gurobi could solve the model within the 5 minutes time-out. We include our heuristics in the figures to show that they always find *some* solution in a reasonable time. We can see that our optimal scheduler fails to find optimal solutions within 5 minutes for most problem instances from our classic task sets and from the Streamit benchmark. This is expected since this model can solve the NP-complete scheduling problem to optimality. We see on Figs. 5.25b and 5.25c that the optimality rate seems stable over the number of cores, but its decreases significantly with the number of tasks to schedule. Fig. 5.25a shows the results for problems with the same number of cores or tasks, and suggests a strong dependency of the optimality rate observed over the task set pattern being scheduled. We further notice that the non-linear integer formulation by Pruhs et al. reaches timeout even more often. The non-linearity of their model is due to the continuous frequency they assume and because a non-

linear problem is much more difficult to solve than a linear one. From this observation, we can conclude that the assumption of continuous frequencies makes more difficult for a ILP or NLP solver to find good solutions, instead of simplifying the problem.

We show with Fig. 5.26a that all schedulers have a similar capacity to find *some* solutions for most problems. However, Fig. 5.26b confirms the observation of Melot et al. [108], that the technique from Pruhs et al. is efficient with sequential task sets but fails more often with very parallel ones. Similarly, the approach by [144] fails more often to find solutions for sequential task sets. Note that our heuristics fail to produce solutions for number of cores that are not powers of 2. Because of that, our heuristics fail to produce solutions in half of our test cases (Fig. 5.26b). Our generalized, ILP-based Crown Schedulers performs mostly as well as the solutions by Melot et al. [107], with the exception of the *organ* variant and the optimal scheduler that seems to fail more often with the FFT task sets. By looking into raw results, we observe that the optimal scheduler fails to find a solution within the 5 minutes timeout for the FFT variants with 63 tasks.

Figures 5.27 and 5.28 demonstrate the longer optimization times for the scheduler by Pruhs et al. and our optimal scheduler. Fig. 5.27a shows our two-phases implementation of the scheduler from Pruhs et al., as it occasionally reaches the 5 minutes timeout in both of its phases. For all schedulers, Fig. 5.27b hints a growing optimization time with number of cores. Figure 5.28b illustrates the same observation with the number of tasks, although we see that the optimal crown scheduler stagnates at the 5 minute time-out for 20 tasks and more. We can also see on Fig. 5.28b that our modified simulated annealing heuristic takes more time to find a solution than binary search. This is expected as it uses binary search to find an initial solution before running its simulated annealing process. However, we can see that even the classic crown scheduler can be faster than simulated annealing crown scheduler implementation.

We show in Fig. 5.29 the energy consumption of schedule for all schedulers we study. Except for our heuristics, we ignore all schedules produced by scheduler that couldn't solve their model to optimality for corresponding the input problem instance. First, the figure shows that the Crown Scheduler variant that ignores static and idle energy, as well as schedulers by Pruhs et al. and Xu et al. always produce solutions. Because they ignore idle energy, they admit no benefit in switching one or several cores off and they use more cores than consolidated crown schedulers. We see that our optimal scheduler consistently produces the best solutions in our experiments. However, every ILP-based Crown Scheduler variants seem to produce in all cases, schedules of no significant difference with optimal solutions. We can further observe that our consolidated heuristics generate schedules of near-optimal quality and that are always at least as good as schedulers by Pruhs et al. and Xu et al.. In most cases, simulated annealing does not bring any significant advantage over binary search, except for the random application

(a) Solver optimality rate as a function of application for the Streamit benchmark.

(b) Solver optimality rate as a function of number of cores for the Streamit task sets (legend in Fig. 5.25a).

(c) Solver optimality rate as a function of number of tasks for the synthetic task sets (legend in Fig. 5.25a).

Figure 5.25: Rate of problem instances where models are solved to optimality.

class of the synthetic task sets.

5.7.3 Core Consolidation

We measure the overall scheduling quality (energy consumption) using our integrated crown scheduler and its extension for core consolidation. We use the energy model as described by Eq. 5.15 with $\zeta = 0.649$, $\kappa = 52.64$ and $\eta = 0.5$. These parameters are derived from power values of an Intel Xscale processor [144]. We only left out a constant factor, because that does not show when comparing energy values of schedules, and we weakened the term κ , because a high value might favor switching off cores too much. We use the same task synthetic, concrete and Streamit-derived task collections as in Sec. 5.7.1 with the target Makespan multiplied by 3. With a looser throughput constraint, it is possible to increase the load of some core so that other core can be switched off, even if the frequency must be increased, as long as the resulting schedule consumes less energy, overall. We include the idle energy consumed by unused core in schedules produced by non-consolidated scheduler variants, whereas we consider unused cores as not consuming energy in schedules produced by core-consolidated schedulers.

We run the ILP solver on a quad-core i7 Sandy Bridge processor at 3GHz and 8GB of main memory. We use Gurobi 6.0.0 and ILOG AMPL 10.100 with 5 minutes timeout to solve both ILP models, on Ubuntu 14.04.

(a) Feasibility as a function of cores for the Streamit task sets.

(b) Feasibility as a function of application for the concrete task set.

Figure 5.26: Feasibility rate of Crown Schedulers, our optimal scheduler as well as Pruhs' and Xu's solutions.

(a) Optimization time as a function of number of tasks for the Streamit task sets (legend in Fig. 5.27b).

(b) Optimization time as a function of number of cores for the concrete task sets.

Figure 5.27: Optimization time of Crown Schedulers, our optimal scheduler as well as Pruhs' and Xu's solutions.

(a) Optimization time as a function of application for the synthetic task sets (legend in Fig. 5.29b).

(b) Optimization time as a function of number of tasks for the synthetic task sets (legend in Fig. 5.29b).

Figure 5.28: Optimization time of Crown Schedulers, our optimal scheduler as well as Pruhs' and Xu's solutions.

(a) Projected energy consumption for schedules as a function of application in the Streamit task sets (in multiple of joules, legend in Fig. 5.29b).

(b) Projected energy consumption for schedules as a function of application in the synthetic task sets (in multiple of joules).

Figure 5.29: Energy consumption of scheduling solutions.

Figures 5.30, 5.31 and 5.32 shows that our core consolidation technique benefits mainly to task collections whose tasks' parallel degree is low. This is not surprising as parallelization of tasks already balances load, so that rearranging the tasks is difficult [82]. The benefits increase with the number of tasks and the number of cores. The energy consumption of classic task collections is reduced by 12% on average for 63 tasks, 16% on average with mergesort and 63% on average with 32 cores. The energy consumption of the synthetic task collection is reduced by 41% on average for 80 tasks, 53% on average with sequential or tasks of low parallel degree and 63% on average with 32 cores. Fig. 5.33 indicates this results also apply to our Streamit Task collections. The energy consumption of the Streamit task collection is reduced by 74% on average for 127 tasks, 61% on average with the application *FIR* and 74% on average with 32 cores.

The more tasks are sequential, the more difficult it is for schedulers to balance the load among busy cores and the more our core consolidation technique can move tasks to scheduling "holes" of another busy processor, resulting in more energy saving opportunities. Also, the more cores are available to switch off, the more energy saving opportunities. Consequently, the looser is the target makespan, the more additional work a core can take and the more other cores can be switched off, resulting in greater differences in schedules quality; this happens in our experimental setup. In extreme cases, core consolidation reduces the energy consumption to 91% of the one by our crown scheduler without consolidation for the classic task collection, 81% for the Streamit task collection and 81% for the synthetic collection, while both schedulers could find an optimal solution within the 5 minutes timeout.

Finally, Fig. 5.34 shows that the core consolidation technique does not influence the optimization time of our crown scheduler.

5.7.4 Island-Aware Crown Scheduling

We measure the impact on energy efficiency of static schedules depending on the voltage and frequency island constraints imposed by the target platform. We target several platform variants of $p = 8$ and $p = 32$ cores, where cores are grouped in islands of 2^q cores where q varies between 0 and $\log_2 p$. We compute schedules for applications of concrete and Streamit task sets, using the same throughput constraints as explained in Sec. 5.7. We use AMPL [41] version 20160325 and Gurobi 6.51 with 15 cores Intel Xeon E5-2660 and 32 GiB 1600 MHz memory.

Figure 5.35 shows the resulting energy for the concrete and Streamit task sets. We see that, for the concrete task set, the solutions computed by our integrated Crown Scheduler decrease moderately with tighter constraints, but degrade suddenly when all cores are grouped in a single island. In contrast, the solutions generated for the Streamit task set seem largely unaffected. This shows the capacity of our Crown Scheduler to find good

(a) Energy consumption per number of tasks for the synthetic task collection (in multiple of joules).

(b) Energy consumption per number of tasks for the class task collection (in multiple of joules).

Figure 5.30: Our technique performs better on sequential task collections.

(a) Energy consumption per number of tasks for the synthetic task collection (in multiple of joules).

(b) Energy consumption per number of tasks for the classic task collection (in multiple of joules).

Figure 5.31: Our technique performs better on task collections with many tasks.

(a) Energy consumption per number of tasks for the synthetic task collection (in multiple of joules).

(b) Energy consumption per number of tasks for the classic task collection (in multiple of joules).

Figure 5.32: Our technique performs better on many cores.

(a) Energy consumption per number of tasks for the Streamit task collection (in multiple of joules).

(b) Energy consumption per number of tasks for the class task collection (in multiple of joules).

Figure 5.33: The Streamit task collection is mostly insensitive to our technique.

(a) Optimization time by number of cores for the classic task collection.

(b) Optimization time by number of cores for the classic task collection.

Figure 5.34: The additional packing constraint over our integrated crown scheduler does not affect significantly the optimization time.

scheduling alternatives in the presence of tighter, but not extreme, architecture constraints. For the concrete task set and for a 8 cores platform, there is a marginal energy consumption difference of 4% between frequency islands of 1 and 4 cores, but a difference of 20% between 4 cores per island and all cores in a unique island. For 32 cores, the difference is 4% between 1 and 16 cores per island and 23% between two and one island. For the same number of cores, the Streamit task sets show a 4% and 7% energy consumption improvement for 8 and 32 cores, respectively. For the Streamit task set, the difference is less than 1% between 8 cores in individual islands per core and all 8 cores in the same island, and 3% between 32 cores in individual islands and all 32 cores in a single island.

Figure 5.36 shows schedules with extreme energy consumption differences between individual islands for each core and a unique island composed of all cores, for a mergesort application on 32 cores. The best schedule consumes only about 30% of the energy of the worse schedule, according to our power consumption model. We see on the best schedule that only the root task needs to run at the highest frequency. However with the worst schedules, all other tasks are forced to run at the highest frequency. This is worsened by the impossibility to switch off unused cores, because they all are part of the same island.

Figure 5.37 shows the schedules for intermediate frequency island configurations. With 2 cores per island, most tasks can run at lowest frequency and many cores can be switched off. The best schedule in Fig. 5.36a consumes only 0.1% less energy than this one. In Fig. 5.37b, we show the case where there are only 2 islands of 16 cores. Here, all small tasks are packed on one half of the processor cores and run at the lowest frequency. Bigger tasks must run in the other half of processors at the highest possible frequency. This configuration yields a 30% energy consumption difference with the best schedule shown in Fig. 5.36a

We show in Fig. 5.38a the optimization time for our concrete and Streamit-derived task sets. We see that the optimizer does not run for the longest time in the extreme configuration cases where all cores have their own island, or when they are grouped in a single island; instead, it takes more time to optimize for platforms with 2 or $p/2$ cores per island. A possible explanation is that the solver finds quickly the best solutions for configuration with individual islands, allowing the early pruning of branches in the search tree. Similarly, the configuration with a unique island might lead the solver to prune a large portion of the solution space and find much faster an optimal solution.

5.8 Related Work

Larsson and Gustafsson [89] give their vision on the impact of voltage and frequency scaling features on multicore architectures. They observe that such a technique requires accurate power models in order to optimize ap-

(a) Projected energy consumption for the Concrete task set (in multiple of joules).

(b) Projected energy consumption for the Streamit task set (in multiple of joules).

Figure 5.35: Projected energy consumption for the Concrete and Streamit task sets for platforms of $p \in \{8, 32\}$ cores and islands of 2^q cores with $q \in 0..\log_2 p$ (constant multiple of Joule).

(a) Best schedule: switch as many cores as possible. Use frequency scaling to fit tasks together in the same core (32 cores, individual islands).

(b) Worst schedule: the root merging task must run at the highest frequency and forces all other tasks to also run at the maximum frequency. No core can be switched off (32 cores, 1 island).

Figure 5.36: Frequency scaling gives even more opportunities to stack mergesort tasks together and switch more cores off (10% energy saving).

(a) Second-best 6-level mergesort schedule for 32 cores (32 cores, 2 cores per island).

(b) Second-worst 6-level mergesort schedule for 32 cores: (32 cores, 2 islands of 16 cores).

Figure 5.37: Frequency scaling gives even more opportunities to stack mergesort tasks together and switch more cores off (10% energy saving).

(a) Optimization time function of island size for our concrete task set.

(b) Optimization time function of island size for our Streamit task set.

Figure 5.38: Optimization time for our concrete and Streamit task sets as a function of frequency island sizes.

plications and that algorithms must be aware of this model. However, they do not discuss scheduling or specialized programming languages such as Streamit that can decouple the application implementation from specific features of target platforms.

Energy-aware allocation, mapping and scheduling problems are being researched intensively in scheduling theory. For instance, Edmonds and Pruhs [34] consider an on-line scheduling algorithm for jobs arriving over time, minimize overall response time (the sum of distances between arrival and completion of each job). Jobs are parallelizable with arbitrary speedup curves, but frequency scaling or energy does not play a role. Chan et al. [19] use an analytic power model and continuous frequencies. Their algorithm does not statically know the work of a job, nor its speedup curve.

Energy efficient static scheduling of task collections with parallelizable tasks onto multiprocessors with frequency scaling has been considered by Li [92]. Their power model is defined as polynomially-reducible to f^α where $\alpha \geq 3$ and therefore comparable to the one we use in this paper. However, the number of processors allocated to each task is fixed and given as part of the problem scenario, and continuous (not discrete) frequency scaling is used.

Xu et al. [143] use a level-packing approach, which means that they might need as many conceptual barrier synchronizations as they have levels, i.e. up to the number of jobs. They use also discrete frequency levels, arbitrary power profiles of cores, moldable tasks with individual, arbitrary speedup curves and ILP. As their ILP approach needs many more variables than ours, they do not use a solver but only evaluate their heuristics.

Sanders and Speck [123] consider the related problem of energy-optimal allocation and mapping of n independent *continuously malleable* tasks with monotonic and concave speedup functions to m processors with *continuous* frequency scaling, given a deadline M and a continuous convex energy usage function E_j for each task j . Continuous malleability means that also a fractional number of processors (e.g., using an extra processor only during part of the execution time of a task) can be allocated for a task; this is not allowed in our task model where allocations, speedup and energy functions are discrete. They propose an almost-linear work algorithm for an optimal solution in the case of unlimited frequency scaling and an approximation algorithm for the case where frequencies must be chosen between a given minimum and maximum frequency. It is interesting to observe that the assumptions of continuous malleability and continuous frequency selection make the integrated problem much easier to solve. Pruhs et al. [116] solve the problem optimally for sequential tasks with continuous frequencies and power model f^α .

Related approaches for throughput or energy efficient (or multi-objective) mapping of complex pipelines have been developed mainly by the MPSoC community for HW/SW synthesis, e.g., by using genetic optimization heuristics [111]. Kessler et al. [82] elaborate on the relation between throughput

requirements of streaming applications and deadline-based scheduling of independent jobs. They also show that for discrete frequency levels, already the optimality criterion of Pruhs et al. [116] for sequential tasks does not hold anymore. Keller et al. [81] and Avdic et al. [7] have considered mapping of streaming task applications onto processors with fixed frequency with the goal of maximizing the throughput. First investigations of energy-efficient frequency scaling in such applications have been done by Cichowski et al. [24]. Zahedi and Lee [147] investigate the fair sharing of a computer resources to run independent tasks. Fairness is defined after a model derived from game theory to ensure each task receives enough resources to be computed fast without impacting the performance of other tasks. This work integrates better in a resource sharing context than optimizing energy usage of a particular task-based parallel application under makespan constraints as Crown scheduling does.

Static scheduling of task collections with parallelizable tasks for makespan optimization has been discussed e.g. by Blasewicz et al. [13]. Dynamic scheduling of task collections with parallelizable tasks for flow time optimization has been investigated by Gupta et al. [53]. In both settings, frequencies were fixed.

Our scheduling techniques rely on the models about both the application to schedule and the target architectures. We consider a set of tasks defined by a global deadline for all tasks and task-individual values such as workload, maximum width and efficiency functions. Rosen et al. [122] give a reliable method to evaluate the Worst Case Execution Time of an application running in parallel on a multiprocessor. They observe that the classic complexity analysis of an algorithm is not reliable on a multiprocessors due to delays introduced by TDMA-driven on-chip buses required by cache-coherency protocols. In their paper, they suggest a bus-aware analysis that provides a better complexity estimation, but that requires deeper knowledge about the target architecture. Huang and Wang [62] model the workload not only from the task but also from the input data. They consider a powerful stream emitter with no energy consumption constraints and energy-constrained terminal receiving the stream. This can be a streaming movie service and mobile phone terminal for example. Huang and Wang suggest to use the emitter's powerful compute capacity to profile the data streamed, to send stream metadata with the data and to let the stream receiver to adapt its own pipeline execution as a function of the predicted workload.

Our LTLG heuristic relates to NPTS (Non-Malleable Parallel Tasks) scheduling, where the number of cores for each task is known and fixed. Fan et al. [40] describe many previous research, including a 2-approximation [43] and a fully polynomial approximation [5]. Further approaches restrict parallel tasks to be mapped to contiguous cores [10] such as the NFDH (Next-Fit Decreasing Height) heuristic [26], FFDH (First-Fit Decreasing Height) [26] and BFDH (Best-Fit Decreasing height) [97]. However, they do not capture

the crown structure that further restricts the mapping of tasks to all cores within exactly one predefined group of cores. Zhou et al. [148] schedules dataflow actors with *intra*- and *iter*-actors parallelism (i.e. moldable tasks) for multi-core architectures, including ILP, simulated annealing and a heuristic based on List Scheduling. The LTLG heuristic is inspired from the LPT (Longest Processing Time) algorithm [49], which is a $4/3$ -approximation algorithm to optimize the makespan of a collection of sequential tasks for multiprocessors.

We consider a simple energy model to drive our scheduler’s decision and evaluation of the solutions’ final quality, based on dynamic, leakage and idle power with the possibility to scale voltage of cores and even to switch them off. Here, we use a convex power function for dynamic power consumption and constants for leakage and idle power. Although simple, this model is commonly used in scheduling work, such as by Conos et al. [27]. Another option involves the measurement of energy consumption running streaming applications on actual hardware with instrumentation code manually coded, using a compiler to generate measurement code or using hardware simulators or virtual machines [18]; this requires the development of a runtime system, which we discuss in Sec. 6. Lots of related work focuses on energy saving with voltage and frequency scaling. Huang and Wang [62] operates voltage scaling depending on the state of tasks’s input buffers. It dynamically accelerates a task whose buffer gets full and slows it down if its buffer is nearly empty. The power function of frequency or voltage is generally assumed as convex [93, 145]. Therefore, an optimal frequency scaling strategy consists in running all tasks at a constant frequency [93]. However, it is unlikely that target platforms can run at the frequency required. In principle, it is possible to approximate it by using the closest higher and lower frequencies available for a time proportional to their distance to the ideal frequency [36]. However in practice, this is not possible when the ideal frequency is too close to an existing one, because of the energy and time overhead induced by frequency switching. In our work, we assume that the execution time of a task is proportional to the frequency used to run it. We motivate this hypothesis with on-chip pipelining, where tasks only access on-chip memories that run at the same frequency as their associated core. This is not true in the general case, where processors can be delayed by memory operations through memory controllers that typically operate at a much lower frequency [45]. However the power function may not always be convex [27, 29]. Dabiri et al. [29] claim that it happens when taking the cost of voltage and frequency scaling into account and propose a scheduling solution that follows this observation. As Crown Scheduling is agnostic to the power function, it is still valid if the function is not convex. However, our model does not take scaling costs into account. De Matteis and Mencagli [31] also assume a linear speedup with frequency scaling and motivate it with CPU-bound computation; they provide a dynamic frequency scaling solution based on dynamically measured workload.

Jordans et al. [75] proposes a semi-automated flow to map throughput-constrained applications to a MPSoC. Their approach is based on Synchronous Data Flow and relies on taskgraph and platform descriptions to compute good mapping solutions. The tasks are annotated with minimal memory size requirements and communication token size to model communications, and a Worst Case Execution Time analysis to estimate its workload. They restrict the target platform to layouts where cores have exclusive accesses to computing resources or memory controllers in order to reduce uncertainties due to the sharing of resources. However, they show that the predicted and measured tasks's execution time is accurate and WCET is a relevant metric to compute schedules in this setting. The MPSoc Application Programming Studio (MAPS) provides a complete tool set to synthesize dataflow applications into MPSoC [18]. They only support sequential tasks and do not apply frequency scaling as our Crown Schedulers and Drake do, but they additionally provide support for heterogeneous target systems and a communication cost model as a linear function of the amount of data transferred. They optimize for energy under throughput constraints using a virtual machine to validate their technique. However, the problem is too complex to find integrated solutions for several applications combined together; their experimental section shows the performance of 3 applications (i.e. 32 tasks) scheduled separately and combined in a separate step. In contrast, our integrated scheduler finds solutions even for 80 tasks. Schor et al. [127] considers the mapping of multiple applications composed of sequential tasks with no frequency scaling to many-core systems. They compute a different schedule for each possible combination for active applications and make sure that an application that stays active before and after a combination transition runs on the same set of cores to prevent costs related to task migration.

Our core consolidation technique is motivated by the increasing cost of leakage current related to dynamic energy [27]. It relates to the work by Stuijk et al. [136] as we try to reduce the use of resources and still meet the deadline constraint. [109] schedules real-time tasks to a multiprocessor and optimize for latency under throughput constraints. However both works consider sequential tasks only and no frequency scaling. Gerards et al. [45] pinpoints that switching cores off is only beneficial if the core stays unused for a sufficiently long time so that more energy is saved than consumed by suspending and restoring a core. With our consolidation approach, we switch cores that we do not use at all, therefore our approach is correct even if we do not take suspend and restore overhead into account.

Few previous works consider voltage and frequency island constraints. Liu and Guo [96], Qi and Zhu [117] consider this constraint but they only support sequential tasks. Interestingly, the technique by Liu and Guo [95] resembles our LTLG mapping heuristic as they schedule tasks in non-increasing order of execution time to the least-loaded group first, but only for sequential tasks.

Additional work model other constraints such as maximal temperature or power consumption. With the increasing density of circuits, the very small and close transistors cannot dissipate heat fast enough and circuits can be damaged. Khdr et al. [86] and Sousa et al. [133] propose runtime adaptation systems that reconfigure actor-based application so that the underlying platform stays below a power and/or temperature budget. In [133], they dynamically adjust the quality or bitrate of video encoding to fit into constraints while still providing good processing quality. Rosales et al. [121] co-optimizes platform-oriented features such as number of cores, energy consumption or processing time together with video encoding-specific goals such as compression rate or image quality. Ukhov et al. [140] focus on the analysis of the steady-state of a periodic application. They calculate the *Steady-State Dynamic Temperature Profile* (SSDTP) to run a temperature-aware mapping technique in order to extend the lifetime of the embedded processor running the application.

Crown Scheduling provides a solution to compute static schedules. Although they can exhibit a low energy consumption, execution conditions such as input data or concurrent resource usage may influence the actual energy consumption. Dynamic schedulers can manage this situation, but they also induce a significant overhead. Schwarzer et al. [128] investigate Quasi-Static Scheduling for throughput optimization, in order to obtain the performance of a dynamic scheduler with a much lower overhead. Andrei et al. [6] describes a $O(1)$ time overhead online scheduler for voltage and frequency scaling for energy minimization with tasks' individual time constraint. Their technique is based on quasi-static scheduling and stores in look-up tables best setting for each task as a function of its actual starting time.

5.9 Conclusion

We have presented crown scheduling, a new technique for static resource allocation, mapping and discrete frequency scaling that supports data-driven scheduling of a set of moldable, partly moldable and sequential streaming tasks onto manycore processors in order to support energy-efficient execution of on-chip pipelined task graphs. Arbitrary power profiles for cores, analytic like f^α or from measurements, can be used.

We have presented heuristics and integer linear programming models for the various subproblems and also for an integrated approach that considers all subproblems together, and evaluated these with synthetic benchmarks. Our experimental results show that the complexity reduction imposed by the crown structure constraint, reducing the number of allocatable processor group sizes from p to $O(\log p)$ and of mappable processor groups from 2^p to $O(p)$, allows for the solution of even medium-sized instances of the integrated optimization problem within a few seconds, using a state-of-the-art integer linear programming solver. The crown structure also minimizes

the number of conceptual barrier synchronizations necessary, and thus external fragmentation of scheduled idle time, thus improving opportunities for energy savings by voltage/frequency scaling.

We presented the *Longest Task, Lowest Group* (LTLG) heuristic, a generalization of the LPT algorithm, to produce load-balanced mappings of moldable tasks, and the *Height* frequency scaling heuristic to minimize energy consumption under a throughput constraint. We demonstrated that using these heuristics lowers the overall optimization time of phase-separated crown scheduling, and showed that both LTLG and Height heuristics produce solutions of quality near to that generated by crown-optimal ILP formulations, with a much faster scheduling time. We also presented a method based on binary search to compute a good initial allocation and a simulated annealing meta-heuristic to further improve solutions toward the quality by an integrated optimal crown scheduler. Finally, we extend our schedulers with an improved energy model that takes static and idle power into account, crown configuration to allow the computation of optimized schedules for any number of cores, core consolidation to further the quality of the solution they generate and frequency islands-aware scheduler to take real architectural constraints into account. Our best heuristic runs in near-linear time in the number of cores and tasks with a factor γ depending on properties of the task collection. We observe that a good processor allocation improves the energy efficiency of schedules, but that a good allocation comes at a high price in optimization time. We generalize Crown Scheduling to compute solutions for any number of cores and with any arbitrary but predefined core decomposition. We investigate island aware scheduling and we show that, although our Crown Schedulers find good solutions for target architectures with large voltage islands, having all cores in a single island prevents significant energy savings. Finally, we give the details of an unrestricted, optimal scheduler and we show that it can produce up to 3.7 times better solutions than Crown Schedulers, under realistic assumptions over the underlying execution platform and parallel scalability of streaming tasks to schedule.

We experimentally show that Crown Schedulers and the heuristics we propose produce near-optimal solutions, using models of classic parallel algorithms, randomly generated task sets and tasks sets derived from the Streamit benchmark suite [48]. In particular, they produce better solutions than competitors [116, 143] and, except for our optimal scheduler, their optimization time is overall lower. In fact, the non-linear scheduler from Pruhs et al. [116] takes most often even more time than our optimal scheduler to find scheduling solutions.

Our technique could be used in a compiler, as we consider off-line scheduling. The inputs would be a description of the taskgraph with workload and efficiency parameters for tasks (and in future work even communication load for edges) and a description of the target architecture. When considering a different target architecture, the application needs to be re-optimized. We

have shortly described the increased flexibility for dynamic crown rescaling due to the crown structure constraints, which allows to easily adapt to fluctuations in streaming task load at runtime. This can be useful to handle applications whose workload can change dynamically, depending on the data of the application, e.g., video decoding [9], using quasi-static scheduling [6, 128]. We have also sketched ways to relax the restrictions of the crown shape, and showed that, in the worst case, an optimal crown schedule is off from an optimal general schedule by a factor of 3.7 for a realistic platform model.

In future work, frequency and sleeping state switching time need to be taken into account to better model architectural constraints. The energy cost of communication between tasks also needs to be taken up in the mapping problem. Our experimental result suggest that there is little gain to expect from a more advanced crown configuration than a binary decomposition. However we also show that for our Crown Scheduling heuristics, the final schedule quality is highly dependent on the decision made at allocation step, but that our feedback loop scheme to find good allocation is costly in time. We could apply machine learning techniques to the allocation phase so that our heuristics can produce near-optimal solutions for a much shorter optimization time. We believe that Crown Scheduling's recursive decomposition into independent branches makes it very well suited for additional extensions that takes target architectures' characteristics such as memory islands or shared caches into account. Examples include memory aware techniques to allocate bandwidth to branches of the crown, or map groups to sets of cores that share a common on-chip memory or a cache, and can be synchronized without message passing. We can schedule composite tasks, i.e., tasks that can be replaced by a set of tasks (for instance, a merge task can be replaced by a 1-level merge tree) and find more optimization opportunities.

Chapter 6

Energy Evaluation of Streaming Applications

Many studies involve the development of languages specialized in correctness [12] or performance [38, 138], all taking as fundamental abstraction communicating Turing machines, where parallelism and performance is either desired or a mere consequence of the abstraction. Stream programs are typically composed of sequential tasks that are easy to design; the overall performance of the final stream program depends greatly on the static or dynamic scheduling strategies applied when distributing tasks to actual processors. Today, there exists many stream languages or frameworks: Streamit [138], CAL [38], Flexstream [60], Fastflow [4] or Drake [106]. In this chapter¹, we introduce *Drake*², a C programming framework to generate efficient streaming programs.

6.1 Introduction

Drake is a plain C programming framework for efficient streaming programs on massively parallel architectures, that separates the code of streaming tasks and the one of target execution platform. It is derived from *Schedeval* [106], which was initially developed for the SCC [61] in order to test the efficiency of static schedules for streaming applications. Drake rely of efficient static schedulers such as Crown Schedulers [108] to produce fast code whose performance can be easily transfered from an execution

¹This chapter is based on the following publication: Nicolas Melot, Johan Janzén and Christoph Kessler: *Mimer and Schedeval: Tools for Comparing Static Schedulers for Streaming Applications on Manycore Architectures*. 8th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), 2015, Beijing, China, Sept. 1st, 2015 [106]

²Release under GPLv3 licence. Download at <https://www.ida.liu.se/labs/pelab/drake/>; contact its author to obtain it under any other licence.

platform to another. Drake focuses on energy efficiency of throughput-constrained streaming applications and supports parallel tasks, implements on-chip pipelined communications [63] as well as voltage and frequency scaling (if supported by the underlying target platform) to generate fast and energy-efficient stream programs. In this chapter, we explore the capacity of Drake to run efficiently a streaming application on many-core architectures. We assess the framework’s overhead with a simple Ping-Pong application and we measure its overall efficiency with mergesort and several mappings inspired from Chapter 4. We investigate the frequency scaling capabilities of Drake using a 12 cores, Intel Xeon-based execution platform and we find that the voltage and frequency can be adjusted to tasks running for a millisecond or more. We implement a Fast Fourier Transform (FFT) from the Streamit benchmark suite [48] for Drake, we deploy it on the SCC and we measure the energy consumed with several schedule variants. We observe that, although the analytical evaluators we use in Chapter 5 find no difference between them, actual measurements pinpoint schedule quality variations due to architectural details not taken into account in our analytical evaluators. Finally, we run mergesort on a 12 cores Intel IA system and compare the raw sorting time with several variants of parallel sorting implementations based on TBB [120] and we measure both the sorting time and the energy consumed. We see that our mergesort implementation for Drake runs in approximately the same as quicksort for TBB but not as well as mergesort for TBB, when the merging work is performed in parallel.

The rest of this chapter is structured as follow: in Sec. 6.2, we give a brief introduction of Drake and the structure of a Drake streaming program. Section 6.3.1 proposes a simple Ping-pong application and use it in various settings to test the overhead of Drake and its ability to hide communication latencies with the computation of other tasks. In Sec. 6.3.2, we devise a test streaming application to assess Drake’s ability to scale frequency of our test execution platform. Section 6.3.3 tests the competitiveness of Mergesort with Drake regarding sorting time on the SCC, compared to specialized implementations. Section 6.3.4 shows the impact of static schedules using both a theoretical model and Drake to measure energy consumption on the SCC. We show that our theoretical energy model that does not take voltage islands into account leads to suboptimal schedules, demonstrating the usefulness of technique explained in Sec. 5.5.5. Finally in Sec. 6.3.5, we compare several parallel sorting implementations using OpenMP [113] and Intel TBB [69] to our mergesort implementation for Drake and we compare their execution time and energy efficiency. Finally, Sec. 6.5 concludes this chapter and elaborates on possible future work.

6.2 Overview

Drake is a C/C++ programming framework that aims at facilitating the development of high-performance streaming applications and relies on static

schedulers to achieve performance portability from a target platform to another. Drake and static schedulers optimize streaming applications for energy under a throughput constraint instead of for time performance. We focus on the optimization of the steady state of a pipeline, where all tasks run once if they have data to operate on, or they are delayed to the next pipeline stage. We model the throughput constraint with a deadline within which every task in the steady state of the pipeline must be finished. If a core has no task to run anymore before the next stage, then it can enter a sleeping state to save energy. Drake supports several features such as tasks that can run on several cores in parallel, dynamic frequency scaling with a task granularity within a pipeline stage as well as stateful and uninterruptible filters. In order to ensure performance portability, Drake separates application and execution platform implementation details into different binary libraries. This allows applications to be compiled again and linked against the correct platform library to be ported from an architecture to another and preserve performance. An execution platform library includes many architecture-dependent details such as the conveying of messages from cores to cores, power or time management. In this section, we give an overview of the Drake framework, we show the structure of a Drake Streaming application and we give a short abstract description of a Drake backend for a target execution platform as well as how we implement the required features for Intel Xeon processors.

We show the general workflow of application building with Drake in Fig. 6.1. We can see that the process involves the application specific code provided by the programmer and the platform-specific part from the platform distributor. We describe the programmer-provided code in more details in Sec. 6.2.1, then we describe Drake’s communication and memory management in Sec. 6.2.2. Finally, the details of a platform-specific plugin are given in Sec. 6.2.3. Drake takes a streaming application and platform description and uses a static scheduler to generate an optimized static schedule and produces the corresponding C code. Then Drake compiles all the C code available using a regular C or C++ compiler and finally links all object files, the Drake framework library as well as the platform backend to produce the final executable.

6.2.1 Drake Streaming Application

Figure 6.1 shows the 3 components a streaming application programmer needs to provide to Drake to produce a final executable. The *Taskgraph* description gives necessary details for a scheduler to produce a good schedule. The Drake modules code contains the C or C++ source code of all tasks described in the taskgraph description. Finally, the *main program* contains the source C or C++ code of the program that uses Drake to achieve computation-intensive work.

Listings 6.1 and 6.2 and its graphic representation in Fig. 6.2 give an

Figure 6.1: Overall workflow of application development with Drake.

example of a 3 tasks mergesort application. Line 22 gives the time in milliseconds of a pipeline stage; if it is set to 0, then a core starts a new pipeline stage immediately after its last task in the schedule terminates, instead of entering some sleeping state until the deadline. Lines 23 to 29, 31 to 37 and 38 to 44 give the descriptions of 3 tasks. For instance, lines 31 to 37 describe a task of id *n1* and human-readable name *leaf_1*. The task's module is *presort* and its corresponding code can be found in file *presort.c* or *presort.cpp*. The fields *workload*, *max_width* and *efficiency* provide useful information to a scheduler. The task's estimated workload can be seen as an estimation of the task's execution time in milliseconds when running on 1 processor at frequency *1Hz*. The task's maximum width (*max_width*) models the maximum number of cores the task can take profit of through parallelism. Finally, the task's efficiency models the additional time required in synchronization to run the tasks with 2 or more cores, in a formula interpreted by `exprtk` [114].

Listing 6.1: Header and graph-wide features parts of the description of a 3-tasks pipelined mergesort application in GraphML format.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
5      http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
6    <!-- Created by igraph -->
7    <key id="g_name" for="graph" attr.name="name" attr.type="
8      string"/>
9    <key id="g_deadline" for="graph" attr.name="deadline" attr.
10     type="string"/>
11    <key id="v_name" for="node" attr.name="name" attr.type="
12     string"/>
13    <key id="v_module" for="node" attr.name="module" attr.type=
14     "string"/>
15    <key id="v_workload" for="node" attr.name="workload" attr.
16     type="double"/>
17    <key id="v_max_width" for="node" attr.name="max_width" attr
18     .type="string"/>
19    <key id="v_efficiency" for="node" attr.name="efficiency"
20     attr.type="string"/>
21    <key id="e_producer_name" for="edge" attr.name="producer_
22     name" attr.type="string"/>
23    <key id="e_consumer_name" for="edge" attr.name="consumer_
24     name" attr.type="string"/>
25    <key id="e_type" for="edge" attr.name="type" attr.type="
26     string"/>
27    <key id="e_producer_rate" for="edge" attr.name="producer_
28     rate" attr.type="double"/>
29    <key id="e_consumer_rate" for="edge" attr.name="consumer_
30     rate" attr.type="double"/>
31    <graph id="G" edgedefault="directed">
32      <data key="g_name">pipelined_merge</data>
33      <!-- <data key="g_deadline">exprtk:max ( tau ) / max ( F)</
34        data> -->
35      <data key="g_deadline">0</data>

```

Listing 6.2: Tasks description part of the description of a 3-tasks pipelined mergesort application in GraphML format.

```

23 <node id="n0">
24   <data key="v_name">root</data>
25   <data key="v_module">root</data>
26   <data key="v_workload">2</data>
27   <data key="v_max_width">1</data>
28   <data key="v_efficiency">exprtk: p == 1 ? 1 : p &lt;= W ?
    1 - 0.3 * ( ( p * p ) / ( W * W ) ) : 1e-06 </data>
29 </node>
30
31 <node id="n1">
32   <data key="v_name">leaf_1</data>
33   <data key="v_module">presort</data>
34   <data key="v_workload">400</data>
35   <data key="v_max_width">1</data>
36   <data key="v_efficiency">exprtk: p == 1 ? 1 : p &lt;= W ?
    1 - 0.3 * ( ( p * p ) / ( W * W ) ) : 1e-06 </data>
37 </node>
38 <node id="n2">
39   <data key="v_name">leaf_2</data>
40   <data key="v_module">presort</data>
41   <data key="v_workload">400</data>
42   <data key="v_max_width">1</data>
43   <data key="v_efficiency">exprtk: p == 1 ? 1 : p &lt;= W ?
    1 - 0.3 * ( ( p * p ) / ( W * W ) ) : 1e-06 </data>
44 </node>
45
46 <edge source="n1" target="n0">
47   <data key="e_consumer_name">left</data>
48   <data key="e_producer_name">output</data>
49   <data key="e_type">int</data>
50   <data key="e_producer_rate">1</data>
51   <data key="e_consumer_rate">1</data>
52 </edge>
53 <edge source="n2" target="n0">
54   <data key="e_consumer_name">right</data>
55   <data key="e_producer_name">output</data>
56   <data key="e_type">int</data>
57   <data key="e_producer_rate">1</data>
58   <data key="e_consumer_rate">1</data>
59 </edge>
60 </graph>
61 </graphml>

```

Lines 45 to 58 of Listing 6.2 model communication channels between tasks. The *consumer name* denotes the name of the channel as seen by the consumer task and *producer name* denotes the same channel as seen by the producer. A channel conveys types data defined by field *type* and parameters

Figure 6.2: Graphic representation of the streaming application described in Listings 6.1 and 6.2.

producer_rate and *consumer_rate* denote the production and consumption rates of data of producer and consumer tasks, respectively, in number of elements.

The module of a task denotes the source code the task runs as part of the pipeline. Listing 6.3 gives the 5 functions a module needs to implement to integrate into the Drake framework. Drake requires that every function takes the task it runs as first argument. Drake makes all cores to run `drake_init()` for all tasks mapped to this core, when Drake builds the pipeline and the function returns a non-zero value if initialization yields no error (initialization phase). It takes the additional pointer argument *aux* which the programmer can transmit to all tasks if required by the streaming application. This function is a good place to perform expensive, non-computation work such as memory allocation. When the pipeline begins, all tasks begin by running `drake_start()` so that they can be bootstrapped (starting phase); if `drake_start()` returns a non-zero value, its next execution is handled by the task's main function `drake_run()` (running phase). The task's main function should return 0 if it has to run again or non 0 if it should stop. In Listing 6.3, `drake_task_depleted(task)` returns non-zero if all the task's predecessors finished and if there is no more data to process from its input communication channels. When a task finishes, Drake runs `drake_kill()` as the final action from this task before the pipeline terminates and is destroyed (termination phase); it returns a non-zero value if no error happens. When all tasks are finished, Drake automatically destroys the pipeline and all cores run the function `drake_destroy()` for all tasks scheduled to them (destruction phase). This is the best place to deallocate memory allocated when the task is initialized.

As Drake assumes no shared memory, tasks are supposed to communicate only through communication channels. Drake performs little control over workarounds such as the use of global shared variables or functions imported through include directives. Using such workarounds might com-

promise platform portability and can occasionally result in linking errors. It is important to define all additional global variables or functions as *static* to prevent communications through shared variables or linking issues.

Listing 6.3: Stub of a Drake module C/C++ source code.

```

1  #include <drake.h>
2  int drake_init ( task_t *task, void* aux) {
3      return 1;
4  }
5  int drake_start ( task_t *task) {
6      return 1;
7  }
8  int drake_run ( task_t *task) {
9      size_t input_size, output_size;
10     link_t *input = pelib_array_read ( link_tp) ( task->pred ,
11         0);
12     link_t *output = pelib_array_read ( link_tp) ( task->succ ,
13         0);
14     int *read = pelib_cfifo_peekaddr ( int) ( input->buffer ,
15         0, &input_size, NULL);
16     int *write = pelib_cfifo_writeaddr ( int) ( output->buffer
17         , 0, &output_size, NULL);
18
19     // Consume input_size elements from read and produce
20     // output_size elements to write
21
22     pelib_cfifo_discard ( int) ( input->buffer, input_size);
23     pelib_cfifo_fill ( int) ( output->buffer, output_size);
24     return drake_task_depleted ( task);
25 }
26 int drake_kill ( task_t *task) {
27     return 1;
28 }
29 int drake_destroy ( task_t *task) {
30     return 1;
31 }

```

Listing 6.4 shows a minimal code stub to run a Drake streaming application within a larger host program. A program using Drake needs to initialize its execution platform (line 3). The platform initialization can take a pointer (here, a *NULL* pointer) whose usage is defined by the platform backend. In line 4, we initialize the Drake streaming application identified with its name. The name is a preprocessor symbol we choose to represent with *APPLICATION* in our example. Line 5 initializes the pipeline. At this stage, Drake allocates the necessary communication memory and calls `drake_init()` for all tasks and forwards a pointer as given in the second argument, to argument *aux* of Listing 6.3. Line 6 starts the pipeline and blocks until all tasks finish. At this stage, Drake runs `drake_start()` and `drake_run()` shown

in Listing 6.3, for all tasks until they finish and run `drake_kill()`. Finally, line 24 destroys the pipeline; Drake runs `drake_destroy()` for all tasks and deallocates all memory it used.

Listing 6.3 shows simple code to operate on input and output channels in function `drake_run()`. Input and output channels are accessible with `pelib_array_read(link_tp)(c, i)`. Drake defines channel through the C structure `link_tp`. Input channels are accessible with `c=task->pred` and output channels with `c=task->succ`. The function requires an additional parameter i (here, $i=0$) to obtain the i^{th} input or output channel. From an instance of queue, the programmer can get the address of the next element to read with `pelib_cfifo_peekaddr(int)(l, d, s, a)`. l is the queue to read from and can be obtained from channel `input` with `input->buffer`. d is the number of elements to skip, typically 0, s is a pointer to a `size_t` variable that contains the number of elements that can be read consecutively after the call to `pelib_cfifo_peekaddr` returns and if a is not NULL, then it is set to the number of elements still available in the queue after all the s elements already available have been discarded from the queue. `pelib_cfifo_writeaddr()` functions in a similar way to obtain the first address where s elements can be written in an output queue. When $t \leq s$ elements have been effectively read or written, `pelib_cfifo_discard(int)(l, t)` discards them from the input queue and `pelib_cfifo_fill(int)(l, t)` commits them in the output queue. After `drake_run()` returns, Drake manages input and output channels and requests or sends data, if needed.

Listing 6.4: Minimal code of a C/C++ program using Drake.

```

1  #include <drake.h>
2  int main ( size_t argc , char **argv ) {
3      drake_platform_t stream = drake_platform_init ( NULL );
4      drake_platform_stream_create ( stream , APPLICATION );
5      drake_platform_stream_init ( stream , NULL );
6      drake_platform_stream_run ( stream );
7      drake_platform_stream_destroy ( stream );
8      drake_platform_destroy ( stream );
9      return EXIT_SUCCESS;
10 }
```

The compilation process of Drake differs from the usual compilation and linking process, as every module is compiled once per task running it. This can result in several compilations of the same source file with different flags to let Drake identify the correct code for corresponding tasks. We automatize the process using GNU extensions of GNU Make and a pair of makefiles *Makefile* and *Makefile.in*. *Makefile* defines the actual compilation process whereas settings such as the Drake streaming application name, the platform description or the taskgraph description files are defined in *Makefile.in*. Listing 6.5 gives a stub of *Makefile.in*. Line 1 defines the list of Drake applications defined in the Makefile. Lines 3 to 16 define compilation settings

for application *merge* as denoted by the name of Drake variables defined, between underscores. Line 3 defines the taskgraph description file for the application; the “cc” suffix to “drake” indicates that sources of modules referred to in tasks descriptions should be stored in a filename ending with “c” and compiled with a C compiler. Line 4 has the same semantics as line 3, but it denotes modules source files to be in files ending with “cpp” and compiled with a C++ compiler. Lines 3 and 4 are mutually exclusive. Line 5 provides the platform description file, lines 6 and 7 are lists of additional C (respectively C++) filenames ending with “c” (respectively “cpp”) and required to the application. Line 8 defines the static scheduler to use to generate a static schedule for the streaming application. The remaining lines define additional libraries, linker flags and library packages (as managed by pkg-config software) required to link the final executable. Note that if `app.schedule.xml` already exists when `make` is invoked to build streaming application *app*, then Drake uses the schedule as it is to generate C code instead of using a scheduler to compute a new one.

Listing 6.5: GNU Makefile stub to define a Drake application.

```

1 drake_package = merge
2
3 drakecc_merge_graph = merge-1.graphml
4 drakecxx_merge_graph = #merge-1.graphml
5 drake_merge_platform = platform-12.dat
6 drakecc_merge_src =
7 drakecxx_merge_src = sort.cpp
8 drake_merge_scheduler = lpt
9 drake_merge_libs = -lm
10 drake_merge_ldflags =
11 drake_merge_pkg =

```

6.2.2 Memory Management

In this section, we describe some of internal mechanisms implemented in the Drake framework to orchestrate the execution of tasks implemented as described in Sec. 6.2.1. In particular, we focus on the basic message passing mechanism internal to Drake and how communication memory is allocated. In its current version, Drake manages *private* and *shared* memories. A memory is said to be *private* if only its owner core can perform read or write operations to it. In contrast, a memory is *shared* if also other cores than its owner core can read or write it in a direct or indirect manner, such as a DMA operation. Private memory is typically off-chip, big and costly whereas shared memory is on-chip, small and cheap, although its access cost depends on the distance between a core and the shared memory it operates on. Note that in its current version, Drake does not model memory islands, where cores included in a same island can perform direct read or write op-

erations to a common memory portion, while other cores must use indirect manners such as DMA operations.

Task-to-task communications take place thanks to *FIFO* queues for data and mailboxes for control. We implement our queue with a memory buffer of fixed size that both producer and consumer tasks can read from or write to, read and write pointers that indicates where data is ready to be read or written, respectively and a boolean value indicating if the last operation performed on the queue is a read or a write. This last boolean field marks the distinction between a full and an empty queue. Since Drake tasks cannot be interrupted, no race condition can occur. When data is read or written, read or write pointers are incremented according to the amount of data read or written and they are wrapped around if they reach the end of queue's memory buffer. When communication occurs, the producer calculates the amount of data it can write, then it writes through the queue's write pointer and updates it. The consumer performs the same operation by reading data through the queue's read pointer before updating it. If the producer and the consumer are mapped to the same core, then no synchronization is required.

If both producer and consumer tasks are mapped to different cores, then Drake allocates memory for the queue's buffer that both tasks can use to communicate using the platform backend's corresponding routine. Drake further allocates mailboxes to forward information from producer to consumer, about the producer's state, i.e., if the task is finished or not, and the amount of data the producer wrote or the consumer read at any given time. From the programmer's point of view, tasks always communicate through *FIFO* queues, but Drake manages actual communications using memory allocation and communication primitives provided by the platform backend. Figure 6.3 gives the communication scheme Drake employs between tasks mapped to different processors. Each processor receives a memory portion from the platform backend, that it can use to write messages to send and use another platform primitive to actually send it. Although Fig. 6.3 suggests that communication memory buffers are distinct, the platform backend can allocate them so that they actually point to the same address. In that case, the hardware automatically forwards the data from a core to another when it is read or written (in steps 4, 6, 14, 16 and 18) whereas pull and push requests and corresponding platform backend actions (steps 7, 8, 9, 10 11, 12, 19 and 20) do not perform anything. This depends on the features offered by the hardware and is defined accordingly in the platform backend source code.

Shared memory used for communication is always allocated in relation with a core. The Drake platform backend is responsible for allocating memory that this core can operate on with low time or energy penalty. Drake allocates shared memory for communication so that the receiver core has low penalty access to that memory. This is because receiver tasks often check on the data they may receive to check if they can work, but they push data only when necessary; therefore, the cost of checking input data must be

Step	Producer	Platform	Consumer
Init.	☐ Read ☐ Written ☐ State	Allocate and distribute memory to producer and consumer, that the platform backend can use to forward data.	☐ Read ☐ Written ☐ State
1	Trigger pull of amount of data read		
2	Read amount of data read by consumer so far		
3	Update read pointer		
4	Work, produce data		
5	Update write pointer		
6	Write amount of data written and producer state		
7	Trigger push of data		
8		Forward data	
9	Trigger push of amount of data written to queue		
10		Forward amount of data written	
11	Trigger push of producer's state		
12		Forward producer state	
13			Trigger pull of amount of data written
14			Read amount of data written
15			Update write pointer
16			Work, consume data
17			Update read pointer
18			Write amount of data read
19			Trigger push back of amount of data read
20		Forward amount of data read	

Figure 6.3: Communication process between two tasks mapped to different cores. Drake uses the platform backend to allocate memory the producer can read and write to, and do the same for the consumer. The platform backend is responsible to forward data between memory allocated between producers and consumers upon push or pull requests. Steps 4, 5, 16 and 17 are performed in the task's module source code and others are automatically managed by Drake.

cheaper than the cost of pushing output data. In the communication scheme Fig. 6.3 describes, the memory used to convey data, the write counter and producer's state are allocated so the receiver core has a cheap access to them (typically, they are allocated to the receiver core's local communication memory), whereas the read counter is allocated so that the producer's core can operate on it for a low penalty. If the producer or receiver task runs on two cores or more, then Drake allocates the necessary memory on the core of the parallel task for which the least number of channels of the same type (input or output channel) has already been allocated.

6.2.3 Platform Plugin

A platform plugin for Drake provides all necessary information for a static scheduler to produce effective schedules, and the necessary code to allow Drake to perform all tasks required: forwarding data from a core to another, setting voltage or frequency of a core, measuring power consumption or time, etc. In this section, we review all information and code required to implement a platform backend for Drake, and we describe how the existing backend for shared memory Intel IA architecture is implemented.

Listing 6.6 gives a platform description necessary to Drake to build code and use a scheduler to compute an efficient schedule. Scalar p (line 1) indicates the number of homogeneous cores available to Drake to run tasks. Set F (line 2) gives the frequency level independently admissible by each of the p cores in the platform. F_{unit} (Line 3) is the multiplier to each value in F to obtain the corresponding frequency in Hertz. Fin is the number of frequency islands in the platform (Line 4) and sets $Fi[i]$ (Line 5 for $i = 1$) give the set of core that composes frequency island i . Similarly, scalar Sin and sets $Si[i]$ describe the shared memory island structure of the platform. Note that, although Listing 6.6 uses an AMPL-based linear algebra language to model a platform, more advanced tools such as XPDL [85] can be used in principle³.

Listing 6.6: Platform description in AMPL input data format for Drake as static schedulers.

```

1 param p := 2;
2 set F := 2667000 2666000 2533000 2399000 2266000 2133000
   1999000 1866000 1733000 1599000 1466000 1333000 1199000;
3 param Funit := 1000;
4 param Fin := 1;
5 set Fi[1] := 1 2;
6 param Sin := 1;
7 set Si[1] := 1 2;
```

³This would require the implementation of an XPDL input library for PeLib [103].

Listing 6.7 gives the fundamental data structure and functions a Drake platform backend needs to implement. `drake_platform_init()` initializes the backend platform to be ready for use by Drake. It takes an optional memory pointer the programmer can use to tweak the platform settings. It returns a pointer to an instance of platform backend-defined `struct drake_platform`. When Drake no longer needs the platform, `drake_platform_destroy()` restores the platform's initial state and frees the associated memory. `drake_platform_shared_size()` and `drake_platform_shared_align()` give the size of a core's shared memory drake uses for communication and the memory alignment required for best performance. `drake_platform_private_size()` returns the size in byte of a core's private memory. `drake_platform_shared_malloc(s, c)` allocates a memory block of size s that core c can use with low penalty and returns a multiple of `drake_platform_shared_align()` as address for best performance. `drake_platform_shared_malloc_mailbox(s, c)` allocates a typically small, unaligned memory buffer of size s that core c can use with low penalty. `drake_platform_shared_free(a, c)` frees memory at address a of core c . `drake_platform_pull(a)` triggers data fetching corresponding to address a previously returned by `drake_platform_shared_malloc()` or `drake_platform_shared_malloc_mailbox()`, so that the required data is available at address a after this function returns. Conversely, `drake_platform_commit(a)` triggers the sending of data stored at address a to the corresponding recipient core's communication memory. `drake_platform_core_id()` returns the id of the core that runs the active thread while `drake_platform_size()` returns the number of cores available for this run. `drake_platform_max()` is the maximum number of cores the architecture could admit. These functions are not appropriate to obtain the core id or the number of cores available to a parallel task: for this usage, use `drake_core_id()` and `drake_task_width()` instead. These functions are defined by the schedule and should not be implemented in a platform backend. `drake_platform_barrier()` blocks execution until all threads reach it. This is for internal use by Drake only and it should not be used in a task's module implementation. `drake_platform_stream_create_explicit(s, i, d, f)`, `drake_platform_stream_init(s, a)`, `drake_platform_stream_run(p)` and `drake_platform_stream_destroy(p)` should make all cores to run functions `drake_stream_create_explicit(s, i, d, f)`, `drake_stream_init(s, a)`, `drake_stream_run(p)` and `drake_stream_destroy(p)`, respectively.

Our backend for Intel IA architecture reads the platform's topology information from the OS and spawns as many threads as number of hardware cores found, excluding virtual cores, and pins each of them to a core so that every core runs exactly one thread. It also uses Linux capabilities to set frequency scaling to be managed in userspace (*userspace* scaling governor), disables hyperthreading cores, disables Intel Turbo boost capability and sets up the interface to the Linux kernel to control processors's frequency and sleeping state, for use when a pipeline runs. Communication buffers are allocated so that both producer and consumer tasks mapped to different cores read and write directly from and to the same memory address. Conse-

quently, `drake_platform_pull()` and `drake_platform_commit()` do not perform anything.

Listing 6.7: API of Drake’s basic functionalities.

```

1  typedef struct drake_platform *drake_platform_t;
2  drake_platform_t drake_platform_init ( void* );
3  int drake_platform_destroy ( drake_platform_t );
4  size_t drake_platform_shared_size ( );
5  size_t drake_platform_shared_align ( );
6  size_t drake_platform_private_size ( );
7  volatile void* drake_platform_shared_malloc ( size_t size ,
        size_t core );
8  volatile void* drake_platform_shared_malloc_mailbox ( size_t
        size , size_t core );
9  void drake_platform_shared_free ( volatile void* addr , size_t
        core );
10 int drake_platform_pull ( volatile void* addr );
11 int drake_platform_commit ( volatile void* addr );
12 size_t drake_platform_core_id ( );
13 size_t drake_platform_core_size ( );
14 size_t drake_platform_core_max ( );
15 void drake_platform_barrier ( void* );
16 vint drake_platform_stream_create_explicit ( drake_platform_t
        stream , void ( *schedule_init ) ( drake_schedule_t* ) , void
        ( *schedule_destroy ) ( drake_schedule_t* ) , void* ( *task_
        function ) ( size_t id , task_status_t status ) );
17 int drake_platform_stream_init ( drake_platform_t stream ,
        void* arg );
18 int drake_platform_stream_run ( drake_platform_t );
19 int drake_platform_stream_destroy ( drake_platform_t );

```

Listing 6.8 shows the interface Drake uses to measure time and implement timed waiting operations. This is essential to Drake so it can manage the time of a pipeline stage. The platform backend defines `struct drake_time` as a time container. `struct drake_platform_time_alloc()` allocates a time data structure in private memory and `struct drake_platform_time_destroy()` frees it. `struct drake_platform_time_init(a, t)` initializes time structure at address *a* with time *t* in milliseconds. `struct drake_platform_time_get(a)` writes the current time in address *a*. `struct drake_platform_time_add(r, o1, o2)` and `struct drake_platform_time_subtract(r, o1, o2)` store in *r* the result of $o1 + o2$ and $o1 - o2$, respectively. `struct drake_platform_time_greater(o1, o2)` and `struct drake_platform_time_equals(o1, o2)` return a non-zero value if $o1 > o2$ and $o1 = o2$, respectively. Finally, `drake_platform_time_printf(f, t)` prints a human-readable digital representation of time *t* in millisecond on file *f*. Our backend of Intel IA relies on system call `clock_gettime` with clock `CLOCK_MONOTONIC` to measure time with a precision of up to a nanosecond.

Listing 6.8: Drake’s time management API.

```

1  typedef struct drake_time *drake_time_t;
2  drake_time_t drake_platform_time_alloc ( );
3  int drake_platform_time_init ( drake_time_t t, double ms);
4  void drake_platform_time_destroy ( drake_time_t time);
5  int drake_platform_time_get ( drake_time_t);
6  int drake_platform_time_subtract ( drake_time_t res, drake_
   time_t t1, drake_time_t t2);
7  int drake_platform_time_add ( drake_time_t res, drake_time_t
   t1, drake_time_t t2);
8  int drake_platform_time_greater ( drake_time_t t1, drake_time
   _t t2);
9  int drake_platform_time_equals ( drake_time_t t1, drake_time_
   t t2);
10 int drake_platform_sleep ( drake_time_t period);
11 FILE* drake_platform_time_printf ( FILE* stream, drake_time_t
   );

```

In Listing 6.9, we show optional routines Drake can use to manage power and frequency scaling. If the target platform doesn’t support this feature, then their implementation can be left empty. `drake_platform_get_frequency()` returns the current frequency for the core calling, expressed as the corresponding index in the sorted set F described in the platform description. `drake_platform_set_frequency()` sets the frequency of the calling core to the index specified as argument. Functions defined in lines 3 to 6 enable or disable an entire core or only deep sleeping states, for a given core id.

Listing 6.9: Power management API in Drake.

```

1  size_t drake_platform_get_frequency ( drake_platform_t);
2  int drake_platform_set_frequency ( int frequency);
3  void drake_platform_core_disable ( drake_platform_t pt, size_
   t core);
4  void drake_platform_core_enable ( drake_platform_t pt, size_t
   core);
5  void drake_platform_sleep_disable ( drake_platform_t pt, size
   _t core);
6  void drake_platform_sleep_enable ( drake_platform_t pt, size_
   t core);

```

Power measurement allows Drake to monitor and report the power consumed while running a streaming application. Its implementation is optional and only useful if the programmer needs to monitor the power consumption on the platform. The backend defines `struct` `drake_power` that contains all information required by the backend to coordinate all power measurement-related routines. `drake_platform_power_init(pt, s, m)` initializes the power monitoring system using platform data contained in `pt`. Once initialized, the backend can collect up to s power samples with an platform-defined time

period, of data defined by *measurement*: if bit `DRAKE_POWER_CHIP` is set, then Drake monitors the power consumption at a chip level, if `DRAKE_POWER_MEMORY_CONTROLLER` is set, then it adds the power consumed by memory controllers. Finally, if `DRAKE_POWER_CORE` is set, then Drake monitor the power consumption of each core, individually. Print routines defined in lines 5 to 8 print the data collected in memory returned by `drake_platform_power_init()` and given as second argument, to the output file given as first argument. `drake_platform_power_printf()` prints for all cores. `drake_platform_power_printf_line()` prints only one sample collected, as defined as third argument. Finally, `drake_platform_power_printf_cumulate()` and `drake_platform_power_printf_cumulate_line()` cumulate all values defined by parameter *metric* (defined the same way as *measurement* above) and print all samples, or only the one defined by argument 3, respectively. Although our backend for Intel currently reads values directly from Intel's PCM library [141], alternative power monitoring methods may be achieved with tools such as MeterPU [94], for instance, to read power from an actual watt meter instead of performance counters.

Listing 6.10: Drake's power monitoring API.

```

1  typedef struct drake_power *drake_power_t;
2  drake_power_t drake_platform_power_init ( drake_platform_t pt
      , size_t samples , int measurement);
3  void drake_platform_power_begin ( drake_power_t pwr);
4  size_t drake_platform_power_end ( drake_power_t);
5  FILE* drake_platform_power_printf ( FILE* stream , drake_power
      _t , char* separator);
6  FILE* drake_platform_power_printf_line ( FILE* stream , drake_
      power_t , size_t line , char* separator);
7  FILE* drake_platform_power_printf_cumulate ( FILE* stream ,
      drake_power_t , int metrics , char *separator);
8  FILE* drake_platform_power_printf_line_cumulate ( FILE*
      stream , drake_power_t , size_t line , int metrics , char *
      separator);
9  void drake_platform_power_destroy ( drake_power_t);

```

6.3 Evaluation

In this section, we evaluate the performance of Drake regarding computation speed and energy consumption. Section 6.3.1, gives a simple ping-pong application to evaluate the overhead of Drake over a hand-coded equivalent application, and shows how Drake hides this overhead with multiple tasks. In Sec. 6.3.3, we use Drake to run Mergesort on the SCC and compare its execution time with our specialized Mergesort implementations for the SCC, that we present in Sec. 3.4. In Sec. 6.3.4, we implement FFT for Drake and measure its power consumption on the SCC with various static

Figure 6.4: Topology and activity of the Ping Pong test application.

schedules. We measure the overhead of frequency scaling with Drake in Sec. 6.3.2. Finally, we measure and compare both execution time and energy consumption of Mergesort with Drake, to various Mergesort variants with Intel Thread Building Block (TBB), for an Intel Xeon processor.

6.3.1 Drake Overhead

We begin by measuring the overhead brought by Drake compared to programming communicating tasks with no additional framework. We devise a minimal program that we call *Ping Pong* and that consists in two tasks *Ping* and *Pong* (Fig. 6.4). Both tasks are initialized with the same constant c . Task *Ping* sends a random integer to task *Pong*. *Pong* receives it, adds c and sends the result back to *Ping*. *Ping* can check if the value that it received minus the one that it sent is c . We repeat this process for a fixed amount of time t and we count the number of iterations that could be run to obtain the average time of one iteration.

We schedule both tasks *Ping* and *Pong* to run at 800 MHz with 3 mappings (Fig. 6.5): both tasks running on the same core (*local*, Fig. 6.5a) and both tasks running on different cores in the same time (*tile*, Fig. 6.5b). We also schedule several pairs of *Ping* and *Pong* tasks with the *remote* setting to a unique pair of cores (Fig. 6.5c). Because *Ping* and *Pong* tasks perform very few calculations each time they run, the time for a round is mostly dedicated to the overhead of Drake. The variants *tile* and *remote* measure the core-to-core communications within the same tile and using the SCC's on-chip network, compared to *local*, where no core-to-core communication happens. Finally, we use the communication primitive provided by RCCE (see Sec. 2.3.2) as a comparison base of communication performance.

Figure 6.6 shows the average round trip time for all configuration described above. The roundtrip time measured is 4.2 microseconds for the local setting. As the local setting is equivalent to writing a value to the L1 cache and read it back, this time is unexpectedly long. However, because *Ping* and *Pong* tasks perform almost no calculation, this can be attributed

Figure 6.5: 3 different testing scenarios to test the overhead of Drake with the Ping Pong test application and one scenario to test the capacity of Drake to hide its own overhead.

Figure 6.6: Average round time measured for the Ping Pong application.

to the overhead of Drake when no inter-core communication happens and there are very few tasks to run. The roundtrip time for tasks mapped to different cores is much higher, but having them mapped to cores in the same tile or in neighbor tiles does not seem to affect the roundtrip time very much. The time difference between Drake and RCCE variants shows the overhead brought by Drake. Figure 6.6 shows that for *tile* and *remote*, 40% of tasks fired are not data-ready. When this happens, Drake checks other tasks in the schedule and tries again with the same task at the next pipeline round. In contrast, the RCCE implementation just polls on its local reception buffer. This different behavior upon missing data can explain this performance difference.

The overhead revealed by Fig. 6.6 can be hidden by more and heavier tasks to mask delays due to communications. We run several pairs of Ping and Pong tasks over remote cores as shown on Fig. 6.5d to hide commu-

nication delays and reduce the data miss rate. We also run multiple pairs mapped on the same core to study the effect of hiding latencies when no core-to-core communications happen. We run the Ping-Pong program for some time t and count the number of roundtrips r that all p pairs of Ping and Pong tasks could run in the time interval t . Yellow and green points in Fig. 6.7a shows the average roundtrip time of a Ping and Pong task pair for local and remote communications. For each setting we calculate the average roundtrip time as

$$\frac{t}{r \cdot p},$$

and we show on Fig. 6.7a the reference time of tasks polling using RCCE communication primitives as shown in Fig. 6.6. We can see that hiding communications with more independent tasks decreases the average execution time of tasks. This is supported by Fig. 6.7b, showing the drop in the number of tasks fired while data is not available. However, the roundtrip time gap between variants having tasks mapped to the same core and tasks mapped to different tiles remains about $1\mu s$. This gap can be attributed to the additional time required to send and receive synchronization messages as in steps 6 to 15 described in Fig. 6.3. We see that when communications are hidden, the average execution time of a task is lower than the one implemented using RCCE communication primitives. This suggests that our implementation using RCCE communication primitives and polling on a variable also suffers from communication latencies that could not be hidden in our experiment.

The remaining overhead difference could be attributed to the additional time required to fetch the input data from the MPB instead of the L1 cache. This suggests that performance could be improved by prefetching data and flags from the MPB to the L1 cache, in order to hide further the overhead of Drake. However, as the tasks Ping and Pong perform a very small amount of computation, they struggle to hide the overhead. It is expected that heavier tasks can yield even better results.

6.3.2 Frequency Scaling

In this section, we assess the capacity of Drake to apply frequency scaling between tasks of a pipeline. For this experiment, we design a simple streaming application that consists in 2 independent, computationally heavy tasks, both achieving the same amount of work. We define both tasks' work as a loop that executes a *sine* hardware instruction 10 times per iteration. We run several experiments with $5 \cdot 10^i$ iterations for $i \in \{0, \dots, 6\}$, and 10^j executions of a pipeline stage for $j \in \{0, \dots, 6\}$, so that $5 \cdot 10^i \times 10^j$ is constant. A high number of iterations but a low number of pipeline stages gives Drake a lot of time to scale frequency and makes it suffer from a lower time penalty related to frequency switching than with a low number of iterations and high number of stages. For all variants, we devise two schedules

(a) Roundtrip time function of number of simultaneous Ping Pong pairs.

(b) Non-data ready rate of task function of number of concurrent Ping Pong pairs.

Figure 6.7: Effect of hiding communication delays with more task execution.

Figure 6.8: Schedules used to measure the impact of frequency scaling to task execution time.

where both tasks run on a unique processor. Figure 6.8a and 6.8b show both schedules. We schedule one task to run at high frequency (2.4 GHz) and the second at low frequency (1.2 GHz). We define another schedule where both tasks run at the average frequency between high and low (1.8 GHz). Finally, when we switch frequency between tasks, we monitor the impact of waiting for the frequency transition before running more code, or not waiting.

In Fig. 6.9a and 6.9b, we show the performance results for all experiment variants: alternate low and high frequencies (*Alternate*), deactivate frequency scaling but set a constant average value (*No Frequency Scaling*) or activate frequency switching but always use the same average frequency (*Constant*). We make Drake wait for the OS to apply frequency scaling for the delay it suggests before executing the task (*Wait*) or we skip this delay (*No Waiting*). Finally, in Fig. 6.9a, we show the frequency ratio applied to the *alternate* variants (*Frequency ratio*).

Figure 6.9a shows the average ratio between the execution time of the task running at low frequency and the task running at high frequency. We can see that, when the average execution time of a task is high (with higher number of iterations), then the average time ratio tends to 2. This is because Drake has a lot of time to apply frequency scaling and the fast tasks indeed run twice as fast as the slow tasks. On the contrary, when tasks run for a lower average time, the average time ratio gets closer to 1. When tasks run for about $100\mu s$, we can see that the slow task runs faster than the fast task; this is because the tasks run for a too short time, and a task actually runs at the frequency attributed to the task that runs before.

When tasks run faster, they perform fewer iterations per task execution but they run more pipeline stages so that the overall work stays constant. Therefore, when tasks run at different frequencies, fewer iterations per task instance translates into more frequency switching and associated penalty in the overall execution time. Figure 6.9b shows that having a lot of frequency

(a) Average execution time ratio between slow and fast tasks function of average task execution time.

(b) Average overall execution time function average task execution time.

Figure 6.9: Frequency scaling is effective and cheap when tasks run for 1 millisecond or more.

switching operations penalizes significantly the overall performance. Both Fig. 6.9a and 6.9b show that the effectiveness and the cost of frequency scaling is most interesting when tasks run for about a millisecond or more. Finally, Fig. 6.9b shows that frequency switching with Drake yields a time performance penalty of about 10%, but deactivating the frequency switching feature does not improve performance over just not using it.

6.3.3 Computation Speed with Drake on SCC

In this section, we test the performance of Drake with the implementation of a pipelined mergesort. We choose mergesort as we have elaborated on mergesort for the SCC [7, 104, 105] (Chapter 3 and 4) and we can compare the performance of Drake to our previous work. Since mergesort is a simple, well structured algorithm whose performance is mainly limited by memory, it is very suitable for a performance study of on-chip streaming implementations.

Our mergesort for Drake is comparable to the phases 0 and 1 of our on-chip pipelined Mergesort [105] as described in Chapter 4. We implement a 6 levels merging tree that we map to 6 cores sharing a unique memory controller in the SCC (see Sec. 2.3.2). Leaf tasks sort their input buffer using a sequential quicksort in their starting phase and other tasks switch directly to the main running phase, waiting for input to merge.

We begin with a comparison with phase 1 of our previous implementation of on-chip pipeline mergesort for the SCC [105], that is its on-chip pipelined merging part. We run two merge trees on the same quadrant so we can use all 12 cores attached to the same memory controller and thus reproduce the previously published experiment. As the other 36 cores are attached to other memory controllers, there is no on-chip communication between quadrants and we do not compare to phase 2 of our previous implementation, this variant is sufficient to compare both implementations. We measure the time to sort 4 bytes integer input buffers of 2^{20} to 2^{25} elements, between the first time a task is scheduled (at starting phase) until the last task switches to termination phase, without taking the initial sorting of leaf tasks' input buffers into account.

We test the behavior of our Drake implementations with 3 simple schedules. The *single* schedule maps all tasks of both merging trees to the same core. This schedule serves as a comparison baseline for other schedules. The *double* schedule places all tasks of a tree to one core and all tasks of the second tree to another core, to take profit of the parallelism of 2 cores while no tasks communicate from core to core. Finally, the *mixed* schedule maps all tasks of every second level alternatively to one core and another. This setting maximizes core to core communications and makes many tasks share a small communication buffer (64 bytes, that is 16 integers in this example). This prevents tasks from processing a large amount of data before they need to forward it to release their communication buffer and let other

Figure 6.10: Execution time to merge input subsequences (each is pre-sorted) of $2^{20..25}$ elements in total with simple schedules.

tasks run. We test the influence of delays to execute data-ready tasks in the overall performance. For instance, we want to run a task immediately after its predecessor has produced the data it needs to run. We call *level-first* a running sequence where all tasks are run in the order of their level and *depth-first* a running sequence where we run a task immediately after one of its predecessors is run.

Figure 6.10 shows that running both merge trees in two separate cores yields a significant speedup. This can be explained by the parallelism induced, where no communication can hinder the speedup. The fact that the use of two cores doubles the L1 and L2 cache space available may also favor the two-cores variant. We can also see that within the single core schedule neither a level-first or depth-first sequences improves the overall execution time. Finally, the mixed schedule is even slower than the single core, which can be explained by the numerous, small buffers they are forced to work with, yielding more task switching and communication overhead.

We use the 6-level merging-tree schedules *level* and *block* for 12 cores [105] as described in Chapter 4 to run two merge trees and directly compare our on-chip pipelined Drake mergesort implementation to our previous work. The level mapping is a simple mapping that yields perfect load-balancing, but it induces many core-to-core communications. The block mapping is less intuitive but it still yields a perfect load-balancing and decreases communication taking place between cores. For our on-chip pipelined mergesort implementation for Drake using a block mapping, we use the depth-first running sequence. Also, we run a *simpler* on-chip pipelined mergesort implementation, that is similar to the one of Drake but with no support for frequency switching or scheduling beyond the one previously described [105]). All variants sort 2^{20} to 2^{25} elements split in 2 subsequences per leaf node and each subsequence is individually pre-sorted before we start running our mergesort implementations. Figure 6.11 shows a single core schedule divided

Figure 6.11: Execution time to merge input subsequences (each is pre-sorted) of $2^{20..25}$ elements in total with block and level schedules.

by 12 to serve as comparison basis. Both block and level mappings yield a small performance difference despite the higher communication yield by the level mapping. All runs exhibit a 0.7 efficiency for any input size. This efficiency is lower than the double schedule described above, although this may be due to the penalties related to communications that do not happen in the double schedule. The simpler implementation exhibits a slight performance penalty compared to our Drake-based mergesort implementation.

We run our on-chip pipelined mergesort implementation for Drake with a unique 6-level merging tree mapped on 6 cores using both level and block mappings. When using the block mapping, we use the depth-first running sequence within a processor. We also run our Drake-based implementation with a single core and show its execution time divided by 6. Finally, we run a variant with a simpler scheduling support and no frequency scaling, and a Drake-based implementation using the same schedule and running sequence than the simpler variant previously mentioned. In this experiment, we include the time to presort input subsequences into the overall sorting time. Figure 6.12 shows that the level mapping is by far worse than the block mapping. This is because all leaf tasks are mapped to the same core, and this core needs to sort all input subsequences sequentially while other cores just wait for input data. In contrast, the block mapping splits this work across 4 cores that can perform the initial sorting phase in parallel. All variants using the block mapping yield an efficiency of approximately 0.6, demonstrating the relevance of balancing the load of the initial tasks. However, none exceeds the performance of the simpler implementation.

We devise alternative block and level mappings with an additional level of 64 tasks dedicated to an initial sorting. These tasks are distributed among all 6 cores as follows: the core running the root task receives 8 additional presorting tasks, the core running both predecessor tasks of the root task also receives 8 additional presorting tasks and all remaining cores receive 12 of these tasks. Fig. 6.13 shows that the overall execution time slightly benefits from the distribution of initial sorting across cores, yet it never

Figure 6.12: Execution time to sort and merge input subsequences (*not* pre-sorted) of $2^{20..25}$ elements in total, with block and level schedules and leaf tasks starting with a sequential sort.

Figure 6.13: Execution time to sort and merge random input subsequences of $2^{20..25}$ elements in total with block and level schedules and additional presorting tasks distributed among cores.

outperforms the simpler implementation.

This section shows that streaming applications based on Drake compete well with specialized implementations previously described for the SCC. Figures 6.10, 6.11, 6.12 and 6.13 show that the execution time of our Drake-based implementations scale with the size of input and that schedules can noticeably affect the overall execution time.

6.3.4 Energy Consumption with Drake on SCC

In this section, we use Drake to evaluate the quality of schedules regarding energy consumption. We use the Fast Fourier Transform [28] (FFT) as it is a very common algorithm in signal processing applications. FFT is suitable for stream programming, as it exhibits a simple structure and therefore it represents a good benchmark for an on-chip pipelined implementation and benchmark. We adapt an implementation of FFT from the Streamit

Figure 6.14: Graphic representation of FFT2 from the Streamit benchmark suite.

Task	Work
Source	1264
Split	1264
FFTReorderSimple	640
CombineDFT	2464
Join	2048
Sink	1264

Table 6.1: Tasks’ estimated work for FFT2.

benchmark suite [138] to Drake, compute schedules using variations of a Crown Scheduler [108], run it on the SCC using Drake and compare the energy consumption measured for each schedule.

We choose the “FFT2” variant from the Streamit benchmark suite because it has a simple structure and because of its efficiency as claimed by its authors. It contains two important tasks, *FFTReorderSimple* and *CombineDFT* as shown in Fig. 6.14. We use the Streamit compiler to estimate the workload of each task (Table 6.1) and we consider that all tasks run sequentially. Since the schedulers that we use do not support the multiple execution of a task within the same pipeline stage, we modify all tasks so they produce the same amount of data each run.

We derive this FFT2 application into taskgraphs of 3 different makespans in order to stress the scheduler so it schedules tasks to run at different frequencies. We give a *loose* variant of the target makespan of 200, an *average* variant of 15 and a *tight* variant of 4. As our crown scheduler implementations do not support target platforms with non-power-of-2 numbers of cores, we run 11 schedulers to produce schedules for 32 cores only. We run each resulting variant for 60 seconds and we measure the power consumption over time. We compare our measurement to the projections of the energy model described by Melot et al. [108].

Figure 6.15a shows the energy consumption measured by Drake on the SCC for all loose, average and tight variants. They show that a tight target makespan yields a higher energy consumption than average and loose target makespans. This is expected, as a tighter deadline may lead the schedulers to run tasks at a higher frequency. Figure 6.16 shows the details of two schedules for the tight FFT makespan and task graph, that result in the highest and lowest energy consumption as measured by Drake and shown in Fig. 6.15a. The schedule shown in Fig. 6.16a yields a high energy consump-

(a) Energy consumption measured on the SCC by Drake for FFT2 (in multiple of joules).

(b) Energy consumption evaluated by our energy model for FFT2 (in multiple of joules).

Figure 6.15: Energy consumption measured by Drake and projected by our energy model.

(a) Schedule that produces a high energy consumption.

(b) Schedule that produces a low energy consumption.

Figure 6.16: Two different schedules of the same application that yield different energy consumption.

tion and the second shown in Fig. 6.16b saves more energy. A star on a core denotes that at least one task mapped to this core is scheduled to run at the highest frequency on that core, requiring all cores in the corresponding voltage island to have their voltage increased. A circle on a core indicates that no task runs at the highest frequency. Cores whose voltage is increased are colored in dark orange.

We can remark that the schedule that yields more energy has more cores running at a higher voltage. Only one task that requires a higher voltage requires all cores of the voltage island to run with a higher voltage. The schedule shown in Fig. 6.16b succeeds in containing these tasks in a more restricted amount of frequency islands, although it could be further improved. No scheduler used in this experiment is aware of the constraints between frequency and voltage or the frequency or voltage islands restrictions. Figure 6.15b uses the power model

$$P = f^3,$$

which evaluates these schedules as equivalent. This experiment demonstrates the need of schedulers to use better energy models and to be aware of architecture constraints to takes better profit of voltage and frequency scaling capabilities and minimize the energy consumption when they perform optimization.

6.3.5 Performance Test with Mergesort on Intel IA64

In this section, we compare Drake and Intel TBB [120] through several variants of mergesort. Mergesort was designed as an external sorting algorithm, optimized to take profit of small fast memories to perform merge operations of data taken from big and slow storage memories, through a merging stream tree. Furthermore, its structure is simple, it is cache-friendly as data is accessed consecutively, and is mostly insensitive to input data. We first give

Figure 6.17: Merging tree pipeline of depth 3 for Drake. Only light green nodes are part of the actual pipeline as dark orange nodes only pre-sort data chunks before the pipeline begins.

details about our implementation of mergesort for Drake, then we give a short description of Mergesort for TBB and OpenMP which we use as comparison. Finally, we show the results obtained when comparing performance between different implementations.

We implement mergesort for Drake with a binary merging tree, where each node is a sorting or a merging task. See Fig. 6.17 for a merging tree of depth 3. We divide input data into as many chunks as there are leaves in the tree, and we distribute them to leaves. Leaves sort the data they receive using the C++ standard sequential `std::sort` function and forward the result to their parent task in the pipeline through Drake’s channels. Every non-leaf task reads input from its both input channels, merges it and outputs the data merged through its output channel. We model each task as sequential and with a relative workload of 2^{d-h} where d is the tree’s depth and h is the distance in number of channels between a task and the root. For example, for a merging tree of depth $d = 3$ (Fig. 6.17), leaves have a workload of 1 and the root task has a workload of 8. We generate an executable using the Drake tool chain and use a LPT (Longest Processing Time first, see [49]) static scheduler. Figures 6.19a and 6.19b show the resulting schedule for 12 cores and a 4 levels merging tree.

Figure 6.19a shows an efficient schedule for Drake, where pre-sorting and merging tasks are spread out evenly across cores, whereas Fig. 6.19b gives an example of a bad Drake schedule, where merging tasks are well spread out, but pre-sorting tasks do not use every core available. Since pre-sorting represents a significant work, this can lead to a high performance difference.

We use as a reference the parallel sort implementations provided by Robison [120] and based on quicksort and mergesort as well as TBB [69] and OpenMP [113]. We call *TBB Quicksort* a parallel quicksort for TBB with a sequential data partitioning. *TBB Mergesort* is a parallel mergesort with TBB constructs to express parallelism. *High* and *low* variants denote the use of high or low-level TBB constructs to implement parallelism; the low level variant allows more advanced thread management by TBB, such as in-

Figure 6.18: LPT static schedule for a depth 3 merging tree and 12 cores. The frequency scale on the right indicates the tasks' frequency from 1.2 GHz (light yellow) to 2.4 GHz (dark red).

(a) Balanced LPT schedule for a merging tree of depth 4 for 12 cores. Pre-sorting and merging tasks are respectively colored in dark orange and light green.

(b) Unbalanced schedule for a merging tree of depth 4 for 12 cores. Pre-sorting and merging tasks are respectively colored in dark orange and light green.

Figure 6.19: Balanced and unbalanced merging trees for Mergesort with Drakex. Sorting and merging tasks are respectively colored in dark orange and light green.

stance dynamic load-balancing. Finally, *OMP Mergesort* denotes a parallel mergesort using OpenMP for parallelization. All these variants are based on the compiler's C++ standard library to run efficient sequential implementations of stable sort and merge, when parallelism can no longer be applied. Also, all mergesort variants implement a parallel merging phase. In order to provide a better comparison to our Drake mergesort implementation, we adapt them to use C++ standard, non-stable sort and we provide an alternative for the High-level TBB-based mergesort where all merge operations are performed sequentially.

We use GCC 5.4 on Ubuntu 16.04 on a 12 cores Intel Xeon E5-2630L machine to compile all our parallel sorting variants. For each run, we generate pseudo-random arrays of 32-bit integers, including 2^i integers for $i \in 24 \dots 28$, always using the same random seed. We run every combination 2 times in order to reduce the noise due to external influences such as scheduling from the OS and we measure execution time as well as energy consumption as measured by Intel PCM library [141] based on processor hardware counters.

On Fig. 6.20a, we show the sorting time of our mergesort implementations, for 3-level merging trees (where relevant), using 2 to 12 cores at 2.4 GHz. We can see that Drake mergesort outperforms all other variants with 2 and 4 cores. From 6 to 12 cores, TBB mergesort with parallel merge take better profit of parallelism but Drake mergesort is still at least as good as TBB quicksort except for 6 cores. The OpenMP variant does not compete with Drake mergesort nor TBB quicksort or TBB mergesort with parallel merge; it is even outperformed by TBB mergesort with sequential merge for a low number of cores. With a high number of cores, TBB mergesort with sequential merge is the slowest, about twice slower than the fastest variant. The difficulty Drake has to scale with the number of core, and its poor performance with 6 cores can be explained by its absolute lack of any dynamic mechanism to balance the load. Just like the partitioning phase of TBB quicksort, its implementation of the merge operation is sequential, which severely affect its ability to scale well. However, it performs much better than TBB mergesort with sequential mergesort.

Figure 6.20b shows how Drake mergesort benefits from deeper merging trees, but not from a tree deeper than 3 levels. This is because deeper trees have more presorting tasks that can be better distributed through cores, hence taking better profit of parallelization. We see on Fig. 6.21 that, for any frequency level, the sorting time of every variant stays the same when compared to each other, although Drake mergesort shows a slight relative improvement with higher frequencies, but never outperforms TBB quicksort.

Figures 6.22 through 6.23 show the energy consumption measured on our platform with Intel PCM counters and RAPL library. We can see that the Drake mergesort performs always better than TBB quicksort, and better than TBB mergesorts with parallel merges for 2 and 4 cores. OpenMP mergesort and TBB mergesort with sequential merge perform much worse

(a) Sorting time with 1 to 12 cores for merge trees of depth 4.

(b) Sorting time with 12 cores for merge trees of depth 1 to 4.

Figure 6.20: Sorting time in milliseconds of 2^{26} integers at 2.4 GHz

Figure 6.21: Sorting time in milliseconds of 2^{26} integers at 1.2 to 2.4 GHz using 12 cores for merge trees of depth 4.

in all settings. We can compare energy consumption to sorting time with Fig. 6.21 and see that, although Drake mergesort and TBB quicksort have the same time performance, Drake mergesort consumes significantly less energy. We see also that for 6 cores, the energy consumption difference between Drake mergesort and TBB quicksort follows the time difference. We see on Fig. 6.22b and 6.20b that, although a 3-level mergesort tree is faster than a 2-level tree, the 2-level tree variant consumes less energy. Also, although 3 and 4 level trees run for about the same time, 3-level tree mergesort consumes less energy than a 4-level tree mergesort. This can be explained by the increasing merging work to be done by the increasing amount of merging tasks with deeper merging trees, the larger pipeline to fill and drain or by the higher latency.

6.4 Related Work

The topic of stream programming has been covered in many theoretical studies [77, 90]. There are numerous implementations of streaming algorithms on various architectures such as the Cell Broadband Engine [55, 81] and on the SCC [105]. On Intel Xeon, Shih-Wei et al. [129] maps kernels to one or several threads (moldable tasks) but does not support Frequency scaling as Drake; Gummaraju and Rosenblum [52] applies transformations such as fusion on kernels consisting of sequential loops (loop fusion), and maps to cores but does not apply kernel parallelization or frequency scaling. Streamit [138] and CAL [38] are programming languages specialized in

(a) Energy consumption with 1 to 12 cores for merge trees of depth 4.

(b) Energy consumption with 12 cores for merge trees of depth 4 to 4.

Figure 6.22: Energy consumption in Joules for sorting 2^{26} integers at 2.4 GHz

Figure 6.23: Energy consumption in Joules for sorting 2^{26} integers at 1.2 to 2.4 GHz using 12 cores for merge trees of depth 4.

stream computing. Unlike Drake applications written in C with the support of the vast amount of libraries available such as pthreads, they do not allow malleable tasks, i.e. tasks that can run in parallel.

Buurlage et al. [17] proposes an approach based on BSP for Epiphany accelerators [1]. They suggest algorithms composed of as many tasks as target cores, which simplifies considerably the scheduling problem. However, all cores of the chip read data and process it in parallel, much like a GPU. This might put a considerable pressure on on-chip networks and memory controller. In contrast, Drake implements on-chip pipelining; this reduces the number of cores reading from or writing to main memories and allows a considerable speedup [81]. Pelcat et al. [115] propose PREESM, an integrated environment for Synchronous DataFlow programming for DSPs. Recently, they added support for moldable tasks scheduling [148] but not for frequency scaling. Many other streaming languages have been developed, such as Steamit [48]. Savas et al. [125] implement communication FIFO buffers for CAL (see Sec. 2.4.6) and suggest specialized, architecture-specific libraries that implements 5 operations: *read*, *write*, *connect*, *disconnect* and *end_of_transmit*; this relates to Drake’s design that implements platform specific features such as core-to-core communication into a separate, platform-specific library. With their technique, Savas et al. [125] observe speedup between 1.3 and 4.3 over hand-written code. In [126], Savas et al. compare CAL-optimized scientific applications deployed on Epiphany [1] to hand-written GNU scientific library running on a unique ARM processor and observe a x30 speedup. They further note that the development of ap-

plications using CAL takes slightly more time than C for most developers observed in their study.

Numerous articles tackle the problem of scheduling tasks for multiprocessors, with various constraints and objectives [40, 49, 108, 116, 123, 143]; see also Sec. 5.8. Most of these papers use their own approach to compare the effectiveness of their technique to other existing ones and very few publish their benchmarks that could be used to compare schedulers with a common baseline.

6.5 Conclusion

In this chapter, we describe the *Drake* framework for stream programming with parallel tasks and frequency scaling. Drake generates executables whose performance is optimized with the use of static schedulers such as Crown Schedulers [108]. Power monitoring can take profit of measurement tools such as MeterPU [94] while scheduling can take profit of execution platform modeling languages as XPDL [85] for retargetability. We demonstrate the ability of Drake and its platform backend for Intel architectures to take profit of frequency scaling for individual tasks that run for 1 millisecond or more in a pipeline stage. It is interesting to note that even if an algorithm such as our Ping Pong tasks is memory-bound, its performance can be limited by a lack of processing power to manage the overhead of Drake [74]. Our computation speed tests demonstrate that Drake streaming applications can compete with specialized ones and other streaming frameworks with respect to execution time. However, the clear separation between algorithm implementation and platform management (especially communications) in Drake make the fine tuning and platform-specific optimizations more difficult. One opportunity may lie in windowed FIFO buffers [55] to lower the amount of copy operations when sending or receiving data, by reading or writing directly into its memory buffer instead of using manipulation functions that multiplies data copies. Finally, we demonstrate the usefulness of Drake by comparing the quality of static schedules in highly throughput-constrained applications. We show that Drake can emphasize the flaws of static schedulers, such as the omission of voltage-related constraints when scheduling DVFS operations, pinpointing the need of schedulers to be more aware of platform features and constraints to produce good schedules. Finally, we implement several variants of mergesort in order to compare execution time and energy consumption of Drake and other parallel sorting variant from Intel [120] based on TBB and OpenMP. Drake outperforms all variants in time and energy for a low number of cores and equals TBB quicksort in time with a high number of cores. It even outperforms it in energy consumption for every settings. However, specialized implementations of mergesort outperform Drake in both time and energy with a high number of cores. This can be explained by the absence of any parallel merge operation for our Drake mergesort implementation; in fact, the specialized mergesort for TBB using

sequential merge performs much worse in time and energy than Drake.

Future work include the implementation of more features for Drake. Synchronous Data Flow can help allocating communication memory so that Drake allows tasks to produce and consume data following rates defined in a taskgraph. We can complement the static schedules we use to generate Drake programs with an online scheduler that can manage load imbalance issues, or by implementing dynamic Crown rescaling as shown in Sec 5.5.3. Finally, Drake can benefit from an online reconfiguration feature to manage dynamic graph topology or task migration.

Chapter 7

Software

Scheduling for mono or multi or many processors is a very active research area [19, 116, 123, 143]. However, most articles only include a very limited set of related schedulers to compare their novel method with state-of-the-art algorithms. Furthermore, the task collection they test is often limited and rarely available for further fair comparisons. The *STG* (Standard Task Graph) format [79] defines a syntax to model collections of tasks with precedence constraints, where nodes and edges are weighted by their computational or communication load. The same page provides the *Prototype format* that includes the number of cores available to schedule a task graph. However, the STG and Prototype formats leave no room for extensions such as tasks' parallel degree or parallel efficiency. Also, because the number of processors is integrated to the task graph description, architectures and applications are tightly coupled and the possibilities to study scheduling are limited. Hönig and Schiffmann [59] use the STG format to provide a test bench of 36000 randomly-generated task graphs and the optimal solutions they could compute from the beginning for their work to present day (31756 out of 36000 at the time of the article writing in 2004). Kwok and Ishfaq [88] provide 350 task graphs of the same format, of which 250 have no known optimal solution. Finally, none of these articles provide a coherent and extensible framework to manipulate task graphs and architecture description, nor they provide any convenient tool to batch large problem collections to study various forms of the scheduling problem, and generate publishable representations of the results.

We propose an extensible format to describe task graphs and decoupled target platforms, as well as Mimer, an extensible software environment based on our batching and measuring tool to batch scheduling problems with various schedulers, evaluate the schedules' quality and generate publishable visual representations of data. We use the *Pelib* to manipulate task graph and target architecture textual representations. As shown in Sec. 6.2, our task graph format includes computational load, parallel degree and efficiency

or links between tasks as they load, among others. We aim at its extensibility for any further characteristic through the progress of scheduling research. It is derived from the XML-based *GraphML* [137] format for which many tools such as *yEd* [146] already exist to conveniently manipulate graphs. We model target platforms using the AMPL [41] syntax features such as the number of cores or topology in islands. While such description format makes difficult the expression of non-numeric parameters, we can easily use other formats such as XPDL [85] thanks to the flexible Pelib framework. Finally, we represent schedules in XML to express the task sequence and starting time for each processor.

We propose *Mimer* to manage scheduling experiments as well as to collect and to plot information about optimization time, complexity, optimality and the capability of schedulers to schedule a task graph for a target platform. Schedulers' optimization time may include allocation, mapping, frequency scaling and overall time, depending on information the scheduler provides. Schedules are assessed after their projected energy efficiency computed using our energy consumption model (see Sec. 5.6.2, after measurement on the SCC or on Intel architectures (see Sec. 6.2 or after their unscaled makespan. The goal of *Mimer* consists in automatizing scheduling experiments, reducing errors due to the repetition of tedious small tasks, and produce data that can be shared between scheduling researchers, offering a framework to peer review the data shown in scientific articles. Together with *Drake*, all software described in this section represents more than 40 000 lines of C, C++, bash and R code.

In this chapter, we give details about the Pelib, Crown Schedulers, Freja and *Mimer*, as well as how to use them. This chapter gives no other interest than introducing the reader to the software we use for this thesis, not research techniques or results. The reader interested in research techniques or experimental results only can skip this chapter. Section 7.1 gives instructions on how to get started on using the Pelib. Section 7.2 gives instructions on how to use our implementations of Crown Schedulers. We give a brief introduction to Freja in Sec. 7.3. Finally, Sec. 7.4 gives details about *Mimer* and as well as a brief introduction on how to use it.

7.1 Pelib

The Pelib (**P**rogramming **E**nvironment **L**ibrary, a reference to Pelab, the **P**rogramming **E**nvironment **L**aboratory in Linköping University) manages all basic operations over data structures such as task graph and platform descriptions. It is at the core of all our schedulers and energy model evaluators, as well as running our scheduler implementations and other tools such as task graph generators. The Pelib also provide basic and generic data structures for C programs; they are at the heart of *Drake* such as its communication channels. Section 7.1.1 gives an introduction on the Pelib through a few examples on how to use its terminal interface to manipulate

data structure implementations. Finally, Sec. 7.1.2 gives details about the basic data structures we use in Drake.

7.1.1 Terminal Front-End

A good way to apprehend the framework consists in using the *pelib-convert* interface and use the features provided by the pelib. The command below uses the pelib and a task graph generator extension to output a GraphML [137] representation of a 3-levels mergesort application for Drake. The main pur-

```
pelib-convert --process --lib generate --args --lib
mergesort --args --depth 3 -- --output --lib
taskgraph-graphml --stdout
```

pose of **pelib-convert** consists in translating the representation of one or several data structure instances from a form to another. For instance, assuming the command above is stored in *mergesort.graphml*, the command below transforms the mergesort task graph from GraphML format from a file, to an algebraic form on standard output that AMPL can read as input data of, e.g., a linear integer program. The command above is com-

```
pelib-convert --input --lib taskgraph-graphml --file
mergesort.graphml --output --lib taskgraph-ampl_input
--stdout
```

posed of one input and one output directives. They begin with **--input** and **--output**, respectively. Here the input directive reads the file *mergesort.graphml* and use the library *libpelib-taskgraph-graphml.so* to parse it and produce an instance of data structure. In this example, it expects a GraphML syntax as input, produce an instance of *Taskgraph* and stores it in memory for later use or output. Conversely, the output directive use the library *libpelib-taskgraph-ampl_input.so* takes an instance of *Taskgraph* and write it to standard output. Note that there is another directive, **--process**, that reads data structures as input and produce more data structures to output or consume in another process directive.

The more complex command below generates a task graph description for a 3 levels mergesort, takes the description of a 4-cores target platform (from the file *intel_i5_4690_haswell_p4-i2.dat* as shown in Listing 6.6) then run a binary search crown scheduler to produce a static schedule, then outputs

Figure 7.1: Schedule of a 3-level mergesort task graph for a 4-core platform.

it in XML on standard output and generates a figure in SVG to *mergesort-binary.svg*. Listing 7.1 and Fig. 7.1 give a XML and visual representation of the schedule obtained, respectively.

```

pelib-convert --input --lib platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --process --lib generate
--args --lib mergesort --args --depth 3 -- --process
--lib schedule --args --lib crown-binary -- --output
--lib schedule-xml --stdout --output --lib
schedule-tetris --args --ratio 2 -- --file
mergesort-binary.svg

```

Consult Appendix A for the structure of arguments to `pelib-convert` and the library we provide to manipulate data structures.

7.1.2 C Data Structures

In Ch. 6 we give a high-level description of our streaming framework Drake. Drake relies on basic data structures such as arrays and FIFOs to achieve its work. This section gives an introduction on these data structures, including arrays, FIFOs, pairs, sets and maps. We use the C preprocessor to design them so the compiler performs controls on the type of data they manipulate, in a similar way than C++ templates. The following sections give details on how to use each of the data structures implemented in the Pelib.

Fundamentals

In this section, we give the fundamentals of the Pelib's basic data structures for C programs. Unlike usual C functions, Listing 7.2 gives function prototypes with two lists of parameters between parentheses. The first list gives specialization types, for example `int`, that specializes the corresponding

Listing 7.1: Schedule of a 3-level mergesort task graph for a 4-core platform.

```

<?xml version="1.0" encoding="UTF-8"?>
<schedule name="Cr-Bin-newFast-newLTLG-newHeight-bin ( a=3,k
    =52.64,e=0.01861,z=0.649)" appname="Cr-Bin-newFast-
    newLTLG-newHeight-bin ( a=3,k=52.64,e=0.01861,z=0.649)">
  <core coreid="1">
    <task name="root" start="0.000" frequency="3.500000" width
      ="1.000000" workload="4.000000"/>
  </core>
  <core coreid="2">
    <task name="merge_2" start="0.000" frequency="2.700000"
      width="1.000000" workload="2.000000"/>
    <task name="merge_6" start="0.741" frequency="2.500000"
      width="1.000000" workload="1.000000"/>
  </core>
  <core coreid="3">
    <task name="merge_3" start="0.000" frequency="2.700000"
      width="1.000000" workload="2.000000"/>
    <task name="merge_7" start="0.741" frequency="2.500000"
      width="1.000000" workload="1.000000"/>
  </core>
  <core coreid="4">
    <task name="merge_4" start="0.000" frequency="1.700000"
      width="1.000000" workload="1.000000"/>
    <task name="merge_5" start="0.588" frequency="1.900000"
      width="1.000000" workload="1.000000"/>
  </core>
</schedule>

```

function, in the manner of C++ templates. The second list of parameters is the usual function parameters of a classic C function. We define `STRUCT_T` with desired any specialization type (for example `int`) and we use Listing 7.2 to define the basic functions that manipulate a Pelib C data structure. See the documentation for a brief description of these functions. Listing 7.4 gives a simple example on how to use the Pelib generic C data structure in a program.

Listing 7.2: Basic functions of all data structures in

```

1  STRUCT_T* pelib_alloc_struct ( STRUCT_T ) ( );
2  int pelib_alloc_buffer ( STRUCT_T ) ( STRUCT_T* obj , size_t n);
3  int pelib_set_buffer ( STRUCT_T ) ( STRUCT_T* obj , void* buffer
   , size_t n);
4  STRUCT_T* pelib_alloc ( STRUCT_T ) ( );
5  STRUCT_T* pelib_alloc_collection ( STRUCT_T ) ( size_t n);
6  STRUCT_T* pelib_alloc_from ( STRUCT_T ) ( void* buffer , size_t
   n);
7  int pelib_init ( STRUCT_T ) ( STRUCT_T *obj);
8  int pelib_copy ( STRUCT_T ) ( STRUCT_T src , STRUCT_T* dst);
9  int pelib_free_struct ( STRUCT_T ) ( STRUCT_T *obj);
10 int pelib_free_buffer ( STRUCT_T ) ( STRUCT_T *);
11 int pelib_free ( STRUCT_T ) ( STRUCT_T *);
12 int pelib_destroy ( STRUCT_T ) ( STRUCT_T);
13 int pelib_compare ( STRUCT_T ) ( STRUCT_T a , STRUCT_T b);
14 FILE* pelib_printf ( STRUCT_T ) ( FILE* str , STRUCT_T obj);
15 FILE* pelib_printf_detail ( STRUCT_T ) ( FILE* str , STRUCT_T
   obj , int lvl);
16 size_t pelib_fwrite ( STRUCT_T ) ( STRUCT_T obj , FILE* str);
17 size_t pelib_fread ( STRUCT_T ) ( STRUCT_T* obj , FILE* str);
18 char* pelib_string ( STRUCT_T ) ( STRUCT_T);
19 char* pelib_string_detail ( STRUCT_T ) ( STRUCT_T , int);

```

The Pelib provides interface and implementations for generic static arrays, fifos, iterators, pairs, sets, as well as specializations for 32 bits integers, 32 bits floating point scalars, integer complex and strings. Drake relies on Pelib to implement static arrays and maps of more elaborate types such as tasks or *linklink*. Appendix

7.2 Crown Schedulers

Our Crown Schedulers such as our integrated ILP scheduler use the Pelib to run AMPL and solve our models to generate a solution. The Pelib also facilitates the parsing and generation of input and output data in various formats. Our separated and integrated heuristics use several class hierarchies specialized in solving the allocation, mapping and frequency scaling sub-problems, or in providing feedback loops such as our simulated annealing to

Listing 7.3: Code example using the Pelib C data structures.

```
1  int *i, j;  
2  j = 10;  
3  i = pelib_alloc ( int ) ( );  
4  pelib_init ( int ) (&i);  
5  for ( *i = 0; !pelib_compare ( int ) ( *i, j); ( *i)++)  
6  {  
7      pelib_printf ( int ) ( stdout, *i);  
8  }  
9  char *str = pelib_string ( int ) ( *i);  
10 printf ( "%s\n", str);  
11 pelib_destroy ( int ) ( i);  
12 pelib_free ( int ) ( i);
```

Listing 7.4: Code example using the Pelib C data structures.

```
1  int *i, j;  
2  j = 10;  
3  i = pelib_alloc ( int ) ( );  
4  pelib_init ( int ) (&i);  
5  for ( *i = 0; !pelib_compare ( int ) ( *i, j); ( *i)++)  
6  {  
7      pelib_printf ( int ) ( stdout, *i);  
8  }  
9  char *str = pelib_string ( int ) ( *i);  
10 printf ( "%s\n", str);  
11 pelib_destroy ( int ) ( i);  
12 pelib_free ( int ) ( i);
```

Listing 7.5: pelib-convert command to run our integrated Crown Scheduler.

```

pelib-convert --input --lib platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --input --lib
taskgraph-graphml --file taskgraph.graphml --process
--lib schedule --args --lib crown-ilp-integrated --
--output --lib schedule-xml --stdout

```

find good allocation solutions and compute initial solutions for them. In this section, we give a few examples on how to invoke our schedulers using pelib-convert and we give an overview of the global architecture. All schedulers using AMPL require a working installation of AMPL [41] and a Gurobi [54]. We present only the most important options; more options exists such as energy model parameters; consult the source files of scheduler plugins to learn more about them.

We begin with our integrated Crown Scheduler. Listing 7.5 gives the command to run it with pelib-convert. Argument directive `--config` takes the same arguments as shown in Table A.1. It defines the crown configuration described in Sec. 5.5.2); alternatives include *config-organ* and *config-taskgraph*.

Listing 7.6 gives a command line with all explicit options to run a phase-separated crown scheduler using the *fast allocation* allocation strategy (Sec. 5.3), the LTLG mapping heuristic (Sec. 5.3.3) and the Height frequency scaling heuristic (Sec. 5.3.5). The `--config` directive is the same as above. Directives `--allocation`, `mapping` and `--scaling` also take the same arguments as shown in Table A.1. They set the algorithms used in the corresponding phase of the phase separated scheduler. Alternatives include *mapping-ilp* and *mapping-ilp-balanced* for the mapping phase and *scaling-ilp* for the scaling phase.

Listing 7.6: pelib-convert command to run our phase-separated Crown Scheduler.

```

pelib-convert --input --lib platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --input --lib
taskgraph-graphml --file taskgraph.graphml --process
--lib schedule --args --lib crown-modular --args --config
--lib config-binary --allocation --lib
allocation-fastest --mapping --lib mapping-ltlg --scaling
--lib scaling-height -- --output --lib schedule-xml
--stdout

```

In Listing 7.7 we run our binary search technique (Sec. 5.3.6), using LTLG and Height heuristics to compute schedules from an allocation solu-

Listing 7.7: pelib-convert command to run our binary search Crown Scheduler.

```
pelib-convert --input --lib platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --input --lib
taskgraph-graphml --file taskgraph.graphml --process
--lib schedule --args --lib crown-binary --args --config
--lib config-binary --mapping --lib mapping-ltlg
--scaling --lib scaling-height -- --output --lib
schedule-xml --stdout
```

tion. Argument directives *config*, *mapping* and *scaling* have the same meaning and alternative as for the phase separated crown scheduler invocation shown in Listing 7.6.

Listing 7.8 shows how to run our simulated annealing feedback loop to compute good allocation solutions; see Sec. 5.3.7 for more information about the technique. The `--composite` options takes options shown in Tab. A.1 and defines the scheduler that computes the initial solution; here we use our phase-separated scheduler with a binary crown configuration.

Listing 7.8: Using simulated annealing feedback loop and phase separated scheduler.

```
pelib-convert --input --lib platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --input --lib
taskgraph-graphml --file taskgraph.graphml --process
--lib schedule --args --lib crown-composite-annealing
--args --composite --lib crown-modular --args --config
--lib config-binary -- -- --output --lib schedule-xml
--stdout
```

We can use crown consolidation in the same manner as simulated annealing. Listing 7.9 gives the pelib-convert command to run it, using a phase-separated scheduler to compute solutions.

Listing 7.9: Using core consolidation with our phase separated scheduler.

```
pelib-convert --input --lib platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --input --lib
taskgraph-graphml --file taskgraph.graphml --process
--lib schedule --args --lib crown-composite-consolidation
--args --composite --lib crown-modular --args --config
--lib config-binary -- -- --output --lib schedule-xml
--stdout
```

Finally, Listing 7.10 combines crown consolidation and simulated annealing feedback loops, with the phase-separated scheduler to compute solutions. This is an example of scheduler composition with Pelib.

Listing 7.10: Using core consolidation with our phase separated scheduler.

```

pelib-convert --input platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --input --lib
taskgraph-graphml --file taskgraph.graphml --process
--lib schedule --args --lib crown-composite-consolidation
--args --composite --lib crown-composite-annealing
--args --composite --lib crown-modular --args --config
--lib config-binary -- -- -- -- --output --lib
schedule-xml --stdout

```

We also provide the script *crown* to provide a simple interface to use crown schedulers. Its options are the same as described in Table A.5.

7.3 Freja

Measuring and representing the performance of an algorithm represents quite a tedious task. Such activity requires a constant monitoring and intervention to collect data from an experiment, start the next one and check if everything works as expected. Such constant attention is error-prone. Once the data has been fully collected, it is often cumbersome and also error-prone to reshape it to some format convenient to read, such as a plot, or to a format allowing the computation of such plots. There are many benefits in automatizing this process: making it autonomous, more reliable or faster. Freja¹ assists researchers with managing experiments and to produce stronger and meaningful data thanks to an automatized process. This data can be interpreted by analysis tool such as R [118] to produce publishable-quality visual representations of this data.

Running long and complex experiments without supervision is particularly useful to use in clusters, where the user has typically no interactive access to computer nodes that actually run programs. A batch job in such cluster with Freja simply consists in starting an experiment and let Freja run each of them and collect the data. This section gives a brief introduction on the use of Freja to define and run a very simple, but rather meaningless, experiment and generate data that can be loaded into R for processing. For a more extended presentation of Freja with a more realistic example, consult Appendix C

Freja takes a list of variables and their possible values to generate all possible variable combinations and collect data they generate. A Freja experiment takes shape in at least 3 files: *variables* defines variables and their

¹Freyja is (*Freja* in Swedish) the goddess of love, sexuality, beauty, fertility, gold, *seidhr* (sorcery), war, and death in the old Nordic mythology.

Listing 7.11: Example of the experiment description file *variable*.

```

1  #!/bin/bash
2
3  compile= ( a b)
4  a="a aa"
5  b="b bb bbb"
6
7  run= ( c d e)
8  c="cc"
9  d="d dd"
10 e="ee eee"
11
12 restrict='match ( a, "^a$" ) && match ( b, "^bb$" ) || match ( a,
    "^aa$" )',
13
14 output_columns="num sec"

```

possible values, *compile* is a script that prepares the environment according to possible variable combinations (typically, it compiles programs with corresponding options) and the third one *run* runs the experiment for a given set of values, typically running a program with the corresponding arguments.

Listing 7.11 gives a simple example of an experiment description for Freja. In this example, we have 5 variables *a*, *b*, *c* and *d*. Each variable has an independent set of possible values. Line 3 defines the set of variables used when setting up the environment, here variables *a* and *b*. Lines 4 and 5 give the possible values for variables *a* and *b*, respectively, separated with a whitespace. Lines 7 to 10 define 3 more variables and their possible values to be used when running experiments. Line 12 gives a convenient way to prevent Freja from compiling or running one or several combinations of values. It is based on a boolean formula where literals are either a bash command or a test of the value of a variable against a regular expression. Finally, line 14 gives a list of names for the data produced when running experiments.

Listing 7.12 gives a minimal example of the script Freja uses to set the experiment's environment, usually compiling all program variants under test. This script is called for every variable combinations defined in lines 3 to 5 of Listing 7.11, given in order defined in its line 3. Listing 7.13 gives the Freja command to setup the environment after what is defined in Listings 7.11 and 7.12 as well as its output; Freja reports the success of setting up each setting.

Finally, Listing 7.14 gives an example of the script Freja uses to run each experiment. It is very close to Listing 7.12; it additionally includes the variables defined in lines 7 to 10 of Listing 7.11. As for setting up the

Listing 7.12: Example of the environment setting script *compile*.

```

1  #!/bin/bash
2
3  a=$1
4  b=$2
5
6  ## run make with various options, for example:
7  #make A=$a B=$b

```

Listing 7.13: Using Freja to setup environment.

```

user@host:~$ freja compile
[INFO] Will compile 6 settings
[INFO] Setting 2/6: a=a b=bb... OK. 0.00 sec finished on
      09/27/16 13:35:07; approx. 0.15 sec left ( 0.00 minutes)
[INFO] Setting 4/6: a=aa b=b... OK. 0.00 sec finished on
      09/27/16 13:35:08; approx. 0.08 sec left ( 0.00 minutes)
[INFO] Setting 5/6: a=aa b=bb... OK. 0.00 sec finished on
      09/27/16 13:35:08; approx. 0.04 sec left ( 0.00 minutes)
[INFO] Setting 6/6: a=aa b=bbb... OK. 0.00 sec finished on
      09/27/16 13:35:08; approx. 0.00 sec left ( 0.00 minutes)
[INFO] Success. Compilation took 0.28 seconds ( 0.00 minutes
      ) or 0.05 seconds per compilation setting ( 0.00 minutes)
.

```

environment, Listing 7.15 gives the trace produced by Freja when running all experiments one by one, reporting their success or failure. After completion, Freja produces the file *table.csv* in Comma-Separated Values (CSV) format and that contains all data produced by successful experiments for analysis with a tool such as R [118].

7.4 Mimer

Mimer² relies on Freja 7.3 to assist at the comparison of several schedulers' performance in experiments defined by the user. An experiment involves one or several schedulers to compare, schedule evaluators as well as a processing and scripting set of scripts. Figure 7.2 gives a general overview of the workflow Mimer provides. For a given scheduling experiment, the Mimer relies manages the following phases:

²Mimir (*Mimer* in Swedish) is a Nordic mythology figure for wisdom and advisor for Odin, the king of gods. It is associated with *Mimers brunn*, a well that provides knowledge to whomever drinks from it.

Listing 7.14: Example of the experiment script *run*.

```

1  #!/bin/bash
2
3  a=$1
4  b=$2
5
6  c=$3
7  d=$4
8  e=$e
9
10 echo $ ( echo $a$b$c$d$e | wc -m) $ ( date +%N)

```

1. Running all schedulers for all members of a set of task graphs and target platforms within a defined set (*schedule* phase).
2. Assessing generated schedules' quality through custom evaluation processes (*assess* phase).
3. Gather statistics generated in steps 1 and 2 (*gather* phase).
4. Process the data generated and generate publishable visual representations (*plot* phase).

Phase 1 uses schedulers through a calling interface as defined in Sec. A.3.4 by *pelib-scheduler* and options shown in Table A.5. Each phase produces intermediate data the next phase consumes; Mimer stores it in *./mimer*. Phase 2 takes the schedules generated in phase 1 and produces information on schedules' quality (i.e. their energy consumption). Phase 3 collects all data produced in phases 1 and 2 such as the number of cores in the target platform, optimization time or energy consumed and generates a single data source phase 4 can use to produce figures.

Listing 7.15: Using Freja to run individual experiments.

```

[INFO] Will run 24 instances
[INFO] Instance 5/24: a=a b=bb c=cc d=d e=ee... OK. 0.03 sec finished on
09/27/16 13:35:22; approx. 1.01 sec left ( 0.02 minutes)
[INFO] Instance 6/24: a=a b=bb c=cc d=d e=eee... OK. 0.02 sec finished on
09/27/16 13:35:22; approx. 1.04 sec left ( 0.02 minutes)
[INFO] Instance 7/24: a=a b=bb c=cc d=dd e=ee... OK. 0.02 sec finished on
09/27/16 13:35:22; approx. 1.04 sec left ( 0.02 minutes)
[INFO] Instance 8/24: a=a b=bb c=cc d=dd e=eee... OK. 0.02 sec finished on
09/27/16 13:35:22; approx. 1.02 sec left ( 0.02 minutes)
[INFO] Instance 13/24: a=aa b=b c=cc d=d e=ee... OK. 0.02 sec finished on
09/27/16 13:35:22; approx. 0.67 sec left ( 0.01 minutes)
[INFO] Instance 14/24: a=aa b=b c=cc d=d e=eee... OK. 0.02 sec finished on
09/27/16 13:35:22; approx. 0.63 sec left ( 0.01 minutes)
[INFO] Instance 15/24: a=aa b=b c=cc d=dd e=ee... OK. 0.03 sec finished on
09/27/16 13:35:22; approx. 0.58 sec left ( 0.01 minutes)
[INFO] Instance 16/24: a=aa b=b c=cc d=dd e=eee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.52 sec left ( 0.01 minutes)
[INFO] Instance 17/24: a=aa b=bb c=cc d=d e=ee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.47 sec left ( 0.01 minutes)
[INFO] Instance 18/24: a=aa b=bb c=cc d=d e=eee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.41 sec left ( 0.01 minutes)
[INFO] Instance 19/24: a=aa b=bb c=cc d=dd e=ee... OK. 0.03 sec finished on
09/27/16 13:35:23; approx. 0.35 sec left ( 0.01 minutes)
[INFO] Instance 20/24: a=aa b=bb c=cc d=dd e=eee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.28 sec left ( 0.00 minutes)
[INFO] Instance 21/24: a=aa b=bbb c=cc d=d e=ee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.22 sec left ( 0.00 minutes)
[INFO] Instance 22/24: a=aa b=bbb c=cc d=d e=eee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.14 sec left ( 0.00 minutes)
[INFO] Instance 23/24: a=aa b=bbb c=cc d=dd e=ee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.07 sec left ( 0.00 minutes)
[INFO] Instance 24/24: a=aa b=bbb c=cc d=dd e=eee... OK. 0.02 sec finished on
09/27/16 13:35:23; approx. 0.00 sec left ( 0.00 minutes)
[INFO] Success. Batch took 1.77 seconds ( 0.03 minutes) to run, or 0.07
seconds per instance run ( 0.00 minutes).
[INFO] Merging all results to stuff/tis_27_sep_2016_13-35-21_CEST/table.csv
... done.
[INFO] Merging all results to stuff/table.csv ... done.

```

Figure 7.2: Overview of the scheduler benchmark software environment and running process.

Schedulers and their interface are already widely discussed in Ch. 5 and Sec. A.2.4 and A.3.4. Additionally, a scheduler for Mimer use must return 0 if its execution yields no error, even if no correct schedule can be found, and returns any other value otherwise. Returning a non-zero value makes Mimer to ignore the data produced by the scheduler. Finally, mimer provides the environment variable *output* that contains the path to where Mimer stores the output it catches from a scheduler. By convention, such data should be stored in *output* with a filename matching the scheduler's name and its settings and its should end with extension *.raw*.

Section D.1 gives details on schedule evaluators. Section D.2 discusses the data collection, analysis and plotting of data. Finally, Sec. D.3 gives details on how to define an experiment and how to run them using Mimer.

Chapter 8

Conclusion

The recent processor architectures are designed to deliver more and more performance while facing issues such as the power wall and the memory wall. The energy spent in data centers to analyse big amounts of data is such that optimizations include compromises between energy consumption and accuracy [98]. Conventional approaches to high-performance computing through the parallelization of single-threaded computation are limited by the ILP (Instruction-Level Parallelism) wall and are still subject to the memory wall. Numerous architecture improvements such as larger caches and global memory coherency are too costly in processor area and energy consumption to be used in platforms involving several hundreds of cores.

We give a brief overview of the characteristics of newest *many-core* architectures that are designed for high throughput rather than low latencies and aim at the reduction of power and energy consumption. These architectures often relax their shared memory consistency model and some of them do not even provide any automatic shared memory consistency to the programmer. Although this allow the design of more powerful processors, it also increases the complexity of programming which is a task already challenging with traditional architectures. This is because of the explicit management of messages between cores programmers must additionally provide when designing a program. Such effort would need to be repeated for every new type of target processor if (performance) portability is not ensured.

Stream programming exhibits not only explicit parallelism, its use in conjunction with many-core architectures can also contribute to the reduction of the memory wall problem. Communicating tasks can be implemented using the on-chip network available on many-core processors, high latencies to main memories can be hidden by the concurrent computation of other tasks and accesses to distant memories with high latency costs can be drastically reduced thanks to external memory algorithms such as mergesort [87]. Computing over streams on many-core architectures yields many challenges that have been largely covered in recent publications. One of the major

problems is the scheduling of streaming tasks over processors to optimize for latency, for throughput, for energy/power (and possibly many other) or any combination of them.

Our contributions include the detailed study of the Intel Single-Chip Cloud computer (SCC) and the design and analysis of an efficient on-chip pipelined algorithm for it. We use mergesort because of its simple structure that is easy to reason about, because it is memory bound and because of its inherent capability to work in a streaming fashion. We find that the SCC may not be the best architecture to optimize memory accesses as we demonstrate that, in the default setting, its cores altogether fail to saturate the read memory bandwidth. However, we show that the SCC can still benefit from an on-chip pipelined mergesort implementation. Our implementation competes well with other classic mergesort implementations and is competitive with the on-chip pipelined mergesort implemented for the Cell Broadband Engine, given the limits of such comparison.

We propose Crown Scheduling, a novel static scheduling technique to optimize energy consumption for streaming applications under throughput constraints which takes profit of DVFS features offered by the target architecture. Unlike most existing scheduling techniques, our algorithm is able to allocate and map moldable tasks and assign them frequencies to reduce the energy consumption. Our experimental evaluation indicates that ILP-based crown schedulers can find optimal solutions (given the crown constraints, we call this *crown-optimality*) for small to moderate size problems in a reasonable time. We demonstrate that our heuristics produce solutions close to crown-optimality and that they can find solutions for big problem instances (thousands of tasks and hundreds of cores) in a reasonable time.

We extend the technique to target platforms with arbitrary number of cores through *Crown configuration* and we show that different configuration strategies do not affect significantly the efficiency of our technique. We investigate further energy saving by scheduling tasks to a reduced set of cores and switch off the unused ones and observe up to 91% additional energy saving in cases where processors are mostly unused. We adapt Crown Scheduling to take architecture constraints such as voltage and frequency islands into account so that the solutions computed can be used in practice and we show that Crown Schedulers are able to produce almost equally good solutions for platforms with large voltage and frequency islands, and much worse solutions when all cores are grouped in a unique island. This suggests that more energy optimization can be expected for architectures with voltage and frequency scaling for cores grouped in at least two different voltage and frequency islands. We suggest *Crown rescaling* to adapt dynamically a schedule computed statically, in case of tasks running slower or faster than expected, in order to save even more energy. Finally, we provide a mathematical argument to demonstrate that, for realistic target platform and energy consumption models, our integrated Crown Scheduler produce solutions estimated to make a stream application to consume at

worst 3.7 times more energy than with an optimal schedule.

We introduce *Drake*, a C-based programming framework and compilation tools that generates optimized streaming applications with static schedules. Drake allows the measurement of performance in execution time or energy consumption to assess the quality of arbitrary static schedulers. The quality of a schedule may be expressed in the execution time of the algorithm scheduled and/or the energy consumption it yields. Drake eases the implementation of streaming algorithms for an actual many-core architecture, thus facilitating the fair comparison of schedulers. Unlike many streaming programming environments, programming for Drake is based on C, allowing the use of many already existing libraries. We show that, although Drake yields a noticeable overhead, this overhead is mostly hidden when scaling up the number of tasks and the amount of work they perform. Finally we demonstrate the usefulness of this tool by showing the difference in energy consumption of two schedules that are predicted as equivalent when using a simple abstract mathematical energy consumption model.

Future work includes the refinement of our Crown Scheduling technique to rescale dynamically the execution clock frequency of tasks in the case where another task is skipped due to the lack of data, or if a task took a longer time to run than expected. We plan to extend Crown Scheduling to optimize the use of other resources such as main memory bandwidth or for heterogeneous multiprocessors. Additional developments include optimization of communication between tasks mapped to different cores, including with heterogeneous networks, similarly as we did for our pipelined merge-sort implementation for the SCC [105]. We can take into account the target platform topology such as shared memory islands to produce valid schedules for more realistic target platforms. Finally, we can include stream programming characteristics such as production and consumption rates of Synchronous Data Flow (SDF) to compute optimized schedules.

We plan to improve Drake to support more complex stream computing constructs such as the one described by Thies et al. [138] for Streamit. Drake needs to provide a way to reconfigure tasks quickly as well as to take into account dynamic data firing and consuming rates over communication channels. We can extend Drake to support Synchronous Data Flow's heterogeneous production and consumption rates. We plan to port Drake to other architectures such as Tilera, Epiphany, Xeon Phi, or portable programming environments such as MPI so that we can investigate further problems such as the scheduling of tasks for energy optimization in the presence of heterogeneous on-chip networks.

Appendices

Appendix A

Pelib

In this section, we give detailed information about how to use the tools derived from the Pelib through several examples and lists of arguments the command lines can take. We also give detailed information on the most important parts of the Pelib’s software architecture, such as its organization in a set of plugins for all the tasks the Pelib can perform. We further give detailed explanations about the existing plugins as well as how to use them through command line examples and the lists of arguments plugins admit.

A.1 Options for pelib-convert

The order of input and output directives does not matter, but data structures are consumed and produced by process directives (if any) in the same order as they appear in the command line. It is easier to read a command starting with inputs, finishing with outputs and having processing directives in the middle. In general, an input or an output directive can take the options given in Table A.1; Table A.4 gives other pelib-convert options.

Table A.2 gives the libraries we provide and the input or output directives they implement. The process directives are shown in Table A.3

--library, -l	Full filename of the library to use to parse or output a data structure.
--lib, -m	Type of data structure and format of data to parse or output using a system library. If parsing or outputting a task graph in graphml format, this looks for <i>libpelib-taskgraph-graphml.so</i> in the system library directory.
--name, -n	Name to give to a data structure instance when parsing it or name to look for in memory when outputting or processing it. If no name is specified, pelib-convert uses the name if the data structure type returned by the library employed. This is only useful if several input directives produce several instances of the same data structure and libraries used in output or process directives need both input data structure. This scenario does not happen with any software we provide, in their current version.
--args str --	Forwards str as arguments to the library employed. The syntax of str depends exclusively on the library. The -- marker ends the string to be passed to the library. If necessary such arguments can be nested in each others.
--file, -f	Filename pelib-convert reads from or writes to the data to be read or produced.
--stdin	Valid only for an input directive. Reads data from standard input.
--stdout	Valid only for an output directive. Writes data to standard output.
--stderr	Valid only for an output directive. Writes data to standard error.

Table A.1: Options admissible in input, output or process directives of pelib-convert.

Name	Class	Direction	Description
ampl_input	Algebra	IO	Reads and writes algebraic data structures in AMPL format, suitable as input data for a AMPL program.
ampl_output	Algebra	IO	Reads and writes algebraic data structures as output by a AMPL frontend for a e.g. linear integer solver.
taskgraph-graphml	Taskgraph	IO	Reads and writes task graphs in GraphML format.
taskgraph-ampl_input	Taskgraph	IO	Read and writes task graphs in AMPL input format.
taskgraph-ampl_output	Taskgraph	IO	Read and writes task graphs in AMPL output format.
platform-ampl_input	Platform	IO	Reads and writes platform descriptions in AMPL input format (See Listing 6.6).
platform-ampl_output	Platform	IO	Reads and writes platform descriptions in AMPL output format.
schedule-ampl_input	Schedule	IO	Reads and writes schedules in AMPL input format.
schedule-ampl_output	Schedule	IO	Reads and writes schedules in AMPL output format.
schedule-xml	Schedule	IO	Reads and writes schedules in XML format.
schedule-drake	Schedule	O	Writes data on streaming application in C code ready to compile as part as a Drake application (see Fig.6.1).
schedule-tetris	Schedule	O	Produces a graphic representation of a schedule in SVG format.

Table A.2: Existing Input and Output directives for pelib-convert. In column *Direction*, *I* shows that the directive can parse an input stream and *O* can produce an output stream.

Name	Consumes	Produces	Description
generate	none	any	Generates a new data structure instance such as a task graph.
solve	Algebra	Algebra	Runs a algebraic solver with an algebraic input data and produce a solution in the form of another algebraic data structure.
schedule	Taskgraph, Platform	Schedule	Runs a compatible static scheduler that produces a schedule given the corresponding task graph and platform descriptions available in memory.

Table A.3: Existing Process directives for pelib-convert.

<code>--seed s</code>	Initializes random generator to s . If s is <code>--random</code> then <code>time(NULL)</code> as initialization value; otherwise the random generator is not initialized at all.
<code>--sources</code>	Pelib-convert ignores all other arguments and prints the filename of the source package it has been compiled with.

Table A.4: Additional options for pelib-convert.

A.2 Architecture

This section gives a brief overview on the Pelib framework and gives directives to use it to implement new features. The framework provides several classes that model data structures, data parsers and outputs in various formats as well as bindings to AMPL-frontend algebraic solvers and dynamic linking points that pelib-convert uses extensively.

We begin with a short presentation of the core data structures of pelib. They are simple data containers and they are not intended for use in a high-performance program. Fig. A.1a shows that all simple data structures have as common ancestor class *Data* and the derived class *AlgebraData* is the root of algebraic data structures such as scalars, vectors, matrices and sets. More complex data structures we use to model streaming applications inherit from *Record*, such as *Algebra*, *Taskgraph*, *Platform* or *Schedule* (Fig. A.1b). Note that we mention these classes in Tables A.2 and A.3. An instance of *Algebra* is a container for algebraic simple data structures as scalars, vector, etc., *Taskgraph* represents the graphs of communicating tasks and their details, *Platform* holds information about an execution platform and *Schedule* models a static schedule of a streaming application. More information about these structures is given in later sections.

As illustrated in Sec. 7.1, the first role of Pelib consists in translating data structures from a format to another, e.g., a task graph from GraphML to AMPL input data ready to feed a AMPL program. The hierarchy starting in classes *DataParser*, *DataOutput*, *RecordParser* and *RecordOutput* provides all functionalities to achieve this work. In Fig. A.2, all leaves in a hierarchy are specialized classes for parsing or producing data streams representing a complex data structure, or do both. For example, *ScheduleXML* can both parse and output a schedule in XML format, but *ScheduleTetris* can only produce an output for a in instance of *Schedule*. Also, we see on Fig. A.2 two roots in the hierarchy, both giving a common ancestor class for every complex data structure parser or output classes. All intermediate classes exist so that the hierarchy can be extended at any point. For example, *XMLSchedule* is both a parser and output class of *Schedule*, but another class could be a Schedule output class only. This is the case of *ScheduleTetris* that inherits from *ScheduleOutput* only; one can imagine another Schedule output class in XML format; such a class would inherit from *XMLScheduleOutput*.

(a) Class hierarchy of simple data structures.

(b) Class hierarchy of complex data structures.

Figure A.1: Class hierarchy of data structures in Pelib.

Figure A.1a shows a parser and output class hierarchy for simple data structures. Pelib use this hierarchy to implement composable parser and output classes for the complex data structure *Algebra* from individual parser and output classes for each classes of *Scalar*, *Vector*, etc. The design of this hierarchy follows the one of complex data structures shown in Fig. A.2.

Figure A.3: Class hierarchy of all parsers and output implemented in Pelib.

In order to implement a new data structure in Pelib, first it is necessary to determine if it is a simple or a complex one. As a rule of thumb, a *simple* data structure makes little sense alone (such as a scalar) and must be stored together with other simple data structures to model any real-world object; otherwise it should be a *complex* structure. Note that pelib-convert can only manipulate complex data structures, and that a simple data structure requires another complex data structure as container in order to be usable it with Pelib; see *Algebra* as an example.

Both simple and complex data structures require dedicated parser and output in order to be of any use. It is a good practice to decompose the hierarchy between parser or output base classes and the new parser or output class as shown in Fig. A.1b and A.1a. This allows easier later extensions for alternative parser or output classes for this new data structure. In the case of a simple data structures, chances are that more simple data structures are required in order to implement any useful functionality. If so, all need their respective, dedicated parsers and output classes. In order to use them with Pelib, it also requires a container class as a complex data structure (inheriting from *Record*), whose parser and output classes can use a collection of parser and output classes of the simple data structures it can contain. This is the pattern we use to parse and output instances of *Algebra*

A.2.1 Algebra

This class provides a container for algebraic data structures such as scalars, vectors, matrices, sets or sets of sets, although more can be added. All classes of elements contained in an instance of *Algebra* must inherit from *AlgebraData*. Any such element is defined by its identifier as a string and its associated. For example, an instance of *Scalar* may be the pair ("a", 5) in order to model a scalar variable *a* with value 5. See the documentation of *AlgebraData* for detailed information about these elements. All existing classes *Scalar*, *Vector*, *Matrix* and *Set* are defined as templates and can be specialized into integers, single or double-precision values, string or anything else.

An important aspect of descendant classes of *AlgebraData* is the precision. In the case of a floating point scalar describing a schedule deadline, its value might be truncated when generating text output. This results in a slightly smaller value that can make the difference between a solvable scheduling problem to an unsolvable one. The precision parameter of a constructor of *AlgebraData* can take the value *leave* or *higher*. If set to *higher*, the output class is responsible for incrementing most precise digit of each scalar value to output, if necessary; otherwise (precision set to *leave*), it must truncate it. *AlgebraData* provides the member function *fixPrecision* specialized for **float**, **double** and **long double**, that takes a scalar and an output stream precision, and returns the correct value.

An instance of *Algebra* can be parsed or output in AMPL input our out-

put data format using the relevant parser and output classes. For example, see the parsing and output class *AmplInput* that reads and produce data in AMPL format as fed to a AMPL front-end input data. Its constructor takes a collection of *AlgebraData* parser classes and another for output classes. Default parsers and output for floating point and integers values are available with the static member function `AmplInput::getIntFloatHandlers()`. When reading input, *AmplInput* tries each parser in the collection until one reports a successful parsing. Conversely when producing output, *AmplInput* dynamically casts every element into the class taken by the output class and if the cast operation succeeds, it uses that output classes and proceeds to the next element. This scheme makes easy the implementation of additional parser or output classes, whenever necessary.

A.2.2 Taskgraph

This class models a graph of communicating tasks that form a streaming application. It takes a collection of *Task* that model each individual task and a collection of *Link* to model communication links between tasks. See the documentation for more detailed information about this class.

A task as modeled in class *Task* takes several information such as a name, a workload, a parallel degree, an efficiency function, a module (the filename that contains its program), a parallel degree, a frequency and an efficiency function. This function is a string that can either be a space-separated list of efficiency values for each possible number of cores (the first value is for 1 core, the second value for 2 cores, etc.), or an expression. Such an expression must be prefixed by “*exprtk:*” and pelib uses *exprtk* [114] to interpret the remainder of the string. Variable “*p*” stands for the number of cores, “*W*” is the maximal width of the task and “*tau*” is its workload. For example:

```
exprtk: p == 1 ? 1 : p <= W ? 1 - 0.3 * ( ( p * p ) / ( W * W
    ) ) : 1e-06
```

A task graph is defined by its name, its description, its tasks, its links between tasks and its target makespan. As for a task’s efficiency, a target makespan is a string that can either be a single floating point value, or, if prefixed by “*exprtk:*”, an expression interpreted by *exprtk*. In that expression, “*n*” is the set of tasks in the graph and “*n*[]” is the number of tasks, “*p*” is the set of cores and “*p*[]” is the number of cores, “*F*” is the set of frequency, “*tau*[*j*]” is the workload of task *j* and “*W*[*j*]” is the maximal width of task *j*.

```
exprtk: var parallel := 0; var pseudo_seq := 0; for ( var j :=
    1; j <= n[]; j += 1 ) { pseudo_seq += tau[j - 1] / e ( j ,
    min ( W[j - 1], p[] ) ); parallel += tau[j - 1] / ( min ( W[j
    - 1], p[] ) * e ( j , min ( W[j - 1], p[] ) ) ); }; pseudo_seq
:= pseudo_seq / p[]; parallel := ( 3 * pseudo_seq +
    parallel ) / 4; var minF := parallel / min ( F ); var maxF
:= parallel / max ( F ); minF + maxF
```

A.2.3 Platform

The *Platform* class models an execution platform with its number of cores and its topology. Refer to documentation for detailed information about this class.

A.2.4 Schedule

Schedule models a static schedule as computed by our schedulers described in Ch. 5. Find the documentation for detailed information about this class. See also Sec. 7.2 and A.3.4. for several pelib-convert commands to generate a schedule from a taskgraph and a platform descriptions.

A.3 Plugins

All pelib-convert commands described in Sec. 7.1.1 use one or several *directives* taking options such as `--library`, `-l`, `--lib` or `-m`. These refer to dynamic libraries pelib-convert loads and run after the directives given. This section shows how to implement these libraries and give more details about the existing ones given in Tables A.2 and A.3.

In order to be found by options `--lib` or `-m` of pelib-convert, a plugin library must be stored in a system library path and prefixed by the string “lib-pelib-”. For example, the second listing in Sec. 7.1.1 uses the library “taskgraph-graphml” to parse input. With this directive, pelib-convert loads and run “libpelib-taskgraph-graphml.so”. By convention, the name of a library as given to pelib-convert with the `--lib` option is composed of the class name of the data structure produced or written, a dash “-” and the format name being read or written.

A.3.1 Parsing and Output Plugins

All pelib-convert parsing plugin need to implement the two functions shown in Listing A.1. `pelib_parse()` takes as argument cin the input stream of the data to be parsed. `argc` and `argv` are the arguments passed by pelib-convert to the plugin between the directive options `--args --`. The function returns a pointer to a new instance of the object parsed by the library. We can see on Listing A.1 that this class must derive from `Record`. `pelib_delete()` deallocates the instances created by `pelib_parse`.

Listing A.1: Functions required by a pelib-convert parsing plugin.

```

pelib::Record* pelib_parse ( std::istream& cin , size_t argc ,
    char **argv );
void pelib_delete ( Record* rec );

```

An output plugin is simpler as it requires only one function (see Listing A.2). Note that a unique plugin library can implement one or several functionalities such as parsing or output data. Note further that, if relevant, it is best to implement the actual feature in the Pelib then call it from the plugin library code. This allows the functionalities to be also accessible from any program using the Pelib without using pelib-convert.

Listing A.2: Functions required by a pelib-convert output plugin.

```
void pelib_dump ( std::ostream& cout, std::map<const char*,
    Record*> records, size_t argc, char **argv)
```

A.3.2 Process Plugins

Listing A.3 gives the functions required to implement a process plugin for pelib-convert. pelib_process() takes in parameter records, the list of complex data structure already in the memory of pelib-convert, as well as the usual argc and argv parameters for arguments passing. It also need pelib_delete, which has the same role as for a parsing our output plugin library (i.e., deallocating an object instance produced by pelib_process).

Listing A.3: Functions required by a pelib-convert output plugin.

```
std::map<const char*, pelib::Record*> pelib_process ( std::
    map<const char*, pelib::Record*> records, size_t argc,
    char** argv);
void pelib_delete ( Record* rec);
```

In Table A.3, we give process plugins generate, solve and schedule. All implement the function of Listing A.3. generate takes as argument the same option as a pelib-convert directive (see Table A.1 for an exhaustive list of them). The `--lib` argument defines the library responsible to actually generate anything. Such a library must implement the functions in Listing A.4. pelib_generate() instantiates a new object deriving from *Record* and pelib_delete deallocates it. pelib_description() returns a short, but unambiguous description of the data the plugin generates, possibly using information given as arguments through `--args --`.

A.3.3 Solve Plugins

The plugin solve takes the same arguments as a pelib-convert directive. It reads a list of simple algebraic data structures (an instance of *Algebra*) from

Listing A.4: Functions required by a pelib-convert generate plugin.

```

const record* pelib_generate ( size_t argc, char **argv);
string pelib_description ( size_t argc, char **argv);
void pelib_delete ( Record* rec);

```

parameter records, and uses another plugin library (given with argument between markers `--args --`) to run a solver for this input. Such a solver plugin must implement the functions in Listing A.5. `pelib_calculus()` take as input data the content of parameter data, returns a pointer to a newly allocated instance of *Algebra* and inserts in parameter statistics various information about the solving process.

Listing A.5: Functions required by a pelib-convert solve plugin.

```

const Algebra* pelib_calculus ( const map<const string, const
    Algebra> &data, size_t argc, char**argv, map<const
    string, double> &statistics)
void pelib_delete ( Record* rec);

```

We provide the plugin library `ampl` that uses Pelib's *AmplSolver* to run an AMPL frontend to a solver. It takes arguments `--model` and `--run` for the filename of the model to solve and the top-level run script, respectively. Refer to AMPL documentation [41] for more information about them. It reports through `statistics` if the solver could find a solution and if the solution is optimal, regarding the model and the data. Additional options to `ampl` include `--model-name`, `--show-stdout` and `show-stderr`. The two latter instruct the Pelib to show raw output from the solver's standard output and standard error, respectively, and the former informs the Pelib of the model directive existing in the top-level run file (whose filename is given as argument to the `ampl` plugin), so it can replace it with the model file given as parameter. See the AMPL documentation [41] for more information about it.

A.3.4 Scheduling Plugins

Similarly, the `schedule` process plugin takes the options of a `pelib-convert` directive (Table A.1) to load a scheduling library plugin that runs a scheduler. Such a plugin needs to implement functions of Listing A.6. `pelib_schedule()` takes as arguments an instance of *Taskgraph* and *Platform* as well as arguments through options `--args --`, returns a pointer to a newly allocated instance of *Schedule* and reports additional informations through `statistics`.

`pelib_description()` and `pelib_delete()` have the same role as for the generate plugin described above. Section 7.2 gives numerous details on our architecture of Crown Scheduler implementations.

Listing A.6: Functions required by a `pelib-convert` schedule plugin.

```
const pelib::Schedule* pelib_schedule ( const pelib::
    Taskgraph &tg, const pelib::Platform &pt, size_t argc,
    char **argv, map<const string, double>& statistics);
string pelib_description ( size_t argc, char **argv);
void pelib_delete ( Record* rec);
```

Beside Crown Schedulers, we provide implementations of other schedulers. Plugin `lpt` provides an implementation of the LPT scheduling algorithm [49], where we set each task to the maximal frequency the target platform offers. `pruhs-heuristic` is an implementation of scheduling heuristics from Pruhs et al. [116], where we implement task mapping with load balancing and frequency scaling only, but where we ignore task precedence constraints. `pruhs-nlp` uses AMPL to implement a non-linear scheduling model for sequential tasks with precedence constraints and frequency scaling. Finally, `xu-ilp` is an implementation of the ILP model given by Xu et al. [143] and `sanders-speck` generates a mock schedule whose energy consumption matches the one their technique provides [123]. Finally `ilp-optimal-scheduler` is our ILP implementation of the unrestricted scheduler we show in Sec. 5.6. See Listing A.7 for an example on how to run it; many more examples for our Crown Schedulers are available in Sec. 7.2.

We also provide the script *pelib-scheduler* that offers simplified arguments to run a crown scheduler implemented with the Pelib. It is also useful to allow Mimer (see Sec. 7.4) to use it. Table A.5 gives the arguments *pelib-scheduler* admits. Some of them are for Mimer's internal use only. This script produces on standard output all data written by the scheduler in output parameter *statistics* of Listing A.6. Listing A.8 gives an example of

Listing A.7: `pelib-convert` command to run our unrestricted ilp optimal scheduler for task graph description in file *taskgraph.graphml* and target platform in file *intel_i5_4690_haswell_p4-i2.dat*.

```
pelib-convert --input --lib platform-ampl_input --file intel
_i5_4690_haswell_p4-i2.dat --input --lib
taskgraph-graphml --file taskgraph.graphml --process
--lib schedule --args --lib ilp-optimal-scheduler --
--output --lib schedule-xml --stdout
```


such output.

Listing A.8: Example of statistic information produced on pelib-scheduler's standard output.

```
optimal = 0
feasible = 1
complexity = 1269.502159
time_allocation = 0.000052
time_mapping = 0.000183
time_frequency = 0.000144
time_global = 0.000379
quality_mapping = 9.876448
```

A.4 Define C Data Structures for Pelib

In this section, we give details on how to use the Pelib generic C data structures to for customized data types. As an example, we propose a scenario where we define the new data type “newstruct” and we want to use the Pelib to obtain type checked arrays and FIFOs for it. Listing A.9 gives the header required for such a definition and Listing A.10 shows the implementation work to perform.

We see on Listing A.9 the function prototypes described in Sec. 7.1.2. Lines 1 to 7 are similar to any usual C header: we define the header guards then the new data structure we need. Note that the new data type must not include any space; therefore for a **union** of a **struct**, it is necessary to define an alias with **typedef**. Lines 9 to 11 define the basic function prototypes shown in Listing 7.2 that other data structures may require to work with our new data type. We define **STRUCT_T** to our new data type name in Line 9, then we include the data structure header. Finally, we define **DONE_newstruct_t** to 1 so that to prevent the prototypes to be defined again. In the same manner, lines 13 to 15 and lines 17 to 19 define prototypes for arrays and fifos with our type. Line 21 to 23 define operations that involve both arrays and FIFOs, such as conversions from one to another. Finally, lines 25 to 37 define successively pairs of **int** and **newstruct_t**, iterators to such pairs and finally, a map data structure with integer keys and **newstruct_t** values.

Listing A.10 gives the implementation for our new data structure. In line 1, we include the header as for any regular C implementation file. In lines 3 to 10, we implement the basic functions defined in Listings 7.2 and A.9; we do not show the actual implementation for brevity. In line 12 to 13 we define **ARRAY_T** as our new data structure type and we include the array implementation file to generate a specialized array for **newstruct_t**. Similarly, lines 15 to 30 generate the specialized implementations for FIFOs, arrays to FIFOs operations and maps.

<code>--taskgraph</code>	Filename of input task graph description in GraphML format.
<code>--platform</code>	Filename of input platform description in GraphML format.
<code>--taskgraph-output</code>	Filename of output task graph description in GraphML format. A scheduler may choose to modify the task graph description and output the new equivalent graph to this filename.
<code>--schedule-output</code>	Filename of output schedule in XML format.
<code>--taskgraph-solution</code>	Mimer Mimer's use. Filename of a task graph description generated by a scheduler when computing a schedule. If the file exists and is correct, then the scheduler can avoid running at all.
<code>--schedule-solution</code>	Mimer Mimer's use. Filename of a schedule solution generated by a scheduler in an earlier run. If the file exists and is correct, then the scheduler can avoid running at all.
<code>--output-solution</code>	Filename of the output schedule generated by the scheduler.
<code>--scheduler</code>	Scheduler to use. <i>pelib-scheduler</i> use this directly this value with the option <code>--lib</code> of the corresponding <i>pelib-convert</i> command.
<code>--description</code>	Do not run the scheduler and output a short, but unambiguous descriptive string of the scheduler employed and its settings.
<code>--sources</code>	Do not run the scheduler and output the archive filename that contains the source files of the scheduler and its dependencies.
<code>--src</code>	For Mimer's internal use. Add more source package filenames to be shown with the <code>--sources</code> options.
<code>--options</code>	Options to transmit to the scheduler. <i>pelib-scheduler</i> use this directly this value with the option <code>--args</code> of the corresponding <i>pelib-convert</i> command.

Table A.5: Options admissible in input, output or process directives of *pelib-scheduler*.

Listing A.9: Header of custom data type “newstruct” using the Pelib generic data structures.

```

1  #ifndef NEWSIRUCT
2  #define NEWSIRUCT
3
4  typedef struct {
5      int a;
6      float b;
7  } newstruct_t;
8
9  #define STRUCT_T newstruct_t
10 #include <plib/structure.h>
11 #define DONE_newstruct_t 1
12
13 #define ARRAY_T newstruct_t
14 #include <plib/array.h>
15 #define DONE_array_newstruct_t 1
16
17 #define CFIFO_T int
18 #include <plib/fifo.h>
19 #define DONE_cfifo_int 1
20
21 #define CFIFO_ARRAY_T newstruct_t
22 #include <plib/fifo_array.h>
23 #define DONE_cfifo_array_newstruct_t 1
24
25 #define PAIR_KEY_T int
26 #define PAIR_VALUE_T newstruct_t
27 #include <plib/pair.h>
28 #define DONE_pair_int_newstruct_t 1
29
30 #define ITERATOR_T pair_t ( newstruct_t, newstruct_t )
31 #include <plib/iterator.h>
32 #define DONE_iterator_pair_newstruct_t_newstruct_t 1
33
34 #define MAP_KEY_T int
35 #define MAP_VALUE_T newstruct_t
36 #include <plib/map.h>
37 #define DONE_map_newstruct_t_newstruct_t 1
38
39 #endif

```

Listing A.10: Implementation of custom data type “newstruct” using the Pelib generic data structures.

```

1  #include <plib/newstruct.h>
2
3  newstruct_t plib_copy ( newstruct_t ) ( newstruct_t s1,
    newstruct_t * s2) { ... }
4  newstruct_t plib_init ( newstruct_t ) ( newstruct_t *val) {
    ... }
5  newstruct_t plib_destroy ( newstruct_t ) ( newstruct_t val) {
    ... }
6  newstruct_t plib_compare ( newstruct_t ) ( newstruct_t a,
    newstruct_t b) { ... }
7  char* plib_string ( newstruct_t ) ( newstruct_t a) { ... }
8  char* plib_string_detail ( newstruct_t ) ( newstruct_t a,
    newstruct_t level) { ... }
9  size_t plib_fread ( newstruct_t ) ( newstruct_t* buffer, FILE*
    stream) { ... }
10 FILE* plib_printf ( newstruct_t ) ( FILE* stream, newstruct_t
    a) { ... }
11
12 #define ARRAY_T newstruct_t
13 #include "plib/array.c"
14
15 #define CFIFO_T int
16 #include "plib/fifo.c"
17
18 #define CFIFO_ARRAY_T newstruct_t
19 #include "plib/fifo_array.c"
20
21 #define PAIR_KEY_T int
22 #define PAIR_VALUE_T newstruct_t
23 #include "plib/pair.c"
24
25 #define ITERATOR_T pair_t ( int, newstruct_t )
26 #include "plib/iterator.c"
27
28 #define MAP_KEY_T int
29 #define MAP_VALUE_T newstruct_t
30 #include "plib/map.c"

```

Listing A.11: Declare 2 integers and print them on screen.

```
int i1, i2;
i1 = 1;
i2 = 2;
pelib_printf (int) (stdout, i1);
pelib_printf (int) (stdout, i2);
```

Listing A.12: Place 2 integers in an integer array and print it on screen.

```
array_t (int) ai;
pelib_alloc_buffer (array_t (int)) (&ai, 2);
pelib_init (array_t (int)) (&ai);
pelib_array_append (int) (&ai, i1);
pelib_array_append (int) (&ai, i2);
pelib_printf (array_t (int)) (stdout, ai);
```

In Listings A.11 through A.16 through, we give examples of data structure composition with Pelib. Listing A.11 provides a simple example with 2 integers printed to screen. In this example alone, it would be easier to use the standard C print functions to display integer values. However, the example of Listing A.12 insert these two integers in an array and prints it on screen. Pelib uses the print method of the data type held by the array, to print a detailed array.

In Listing A.13, we perform the same task with complex values. We could still use C to show the values, but instead we use the already defined print method for complex values defined using the Pelib framework. As for int, we create an array and show it on screen.

For the next example, we need a custom-defined data structure. We choose to implement a single linked list, where all elements hold an array of complex values as shown on Listing A.13. First, we define the structure and we use Pelib to generate the prototype of all basic structure manipulation functions. Finally, we define how we initialize a list and how to show it on screen. Note that the initialization of a list element initializes the array it contains using the generic initialization method from Pelib, specialized for an array of complexes. Our printing method takes an integer parameter *detail* that shows the level of details intended when printing the list on screen. If the detail level is zero, then it only prints the number of elements; otherwise, it prints all elements it contains with a lower level of details. On Listing A.16, we create a second array of complexes and two elements of our linked list that we initialize using the initialization method defined in Listing A.15. We set each list element's array: the first element receives

Listing A.13: Declare 2 integer complexes and print them on screen.

```

complex_t c1, c2;
c1.r = 10;
c1.im = 11;
c2.r = 20;
c2.im = 21;
pelib_printf ( complex_t ) ( stdout , c1);
pelib_printf ( complex_t ) ( stdout , c2);
array_t ( complex_t ) ac;
pelib_alloc_buffer ( array_t ( complex_t ) ) ( &ac , 2);
pelib_init ( array_t ( complex_t ) ) ( &ac );
pelib_array_append ( complex_t ) ( &ac , c1);
pelib_array_append ( complex_t ) ( &ac , c2);
pelib_printf ( array_t ( complex_t ) ) ( stdout , ac);

```

Listing A.14: Output obtained from running the code of Listings A.11, A.12, A.13, A.15 and A.16.

```

1
2
[1:2]
( 10,11)
( 20,21)
[ ( 10,11): ( 20,21) ]
2
[2:3: ( 2)]
[[ ( 10,11): ( 20,21)]:[ ( 30,31): ( 40,41): ( 50,51)]: ( 2)]

```

the array defined in Listing A.13 and the second takes the new array. We set both elements's pointer to next elements so we obtain a chain starting from the first element and finishing with the second one. We finish by using the printing function defined in Listing A.15 3 times with an increasing level of details. In the first call, we only obtain the number of elements in our list, i.e., 2. In the second call, we also print arrays contained in both elements, at minimal level, i.e., the number of elements each array contains and the number of elements in the list. We finish with a level of details where we use the pelib structure to automatically call the print function for our list, its arrays and all elements contained in all arrays. Listing A.14 shows the output for running Listings A.11 to A.16.

Listing A.15: Define single linked list data structure and make it a Pelib data structure.

```

struct list {
    array_t ( complex_t ) array;
    struct list *next;
};
typedef struct list list_t;

#define STRUCT_T list_t
#include <pelib/structure.h>
#define DONE_ struct list_t

int pelib_init ( list_t ) ( list_t *list ) {
    pelib_init ( array_t ( complex_t ) ) ( &list->array );
    list->next = NULL;
    return 0;
}
FILE* pelib_printf_detail ( list_t ) ( FILE* out, list_t list ,
    int details )
{
    if ( details > 0 ) {
        list_t *cur = &list;
        size_t size = 0;
        printf ( "[ " );
        while ( cur != NULL ) {
            char* str = pelib_string_detail ( array_t ( complex_t ) ) (
                cur->array, details - 1 );
            printf ( "%s: ", str );
            free ( str );
            cur = cur->next;
            size++;
        }
        printf ( "[%zu]\n", size );
    } else {
        size_t size = 0;
        list_t *cur = &list;
        while ( cur != NULL ) {
            cur = cur->next;
            size++;
        }
        fprintf ( out, " (%zu)\n", size );
    }
    return out;
}

```

Listing A.16: Build another array of 3 complexes, build a newly defined Pelib single linked list, insert both arrays of complexes and print the resulting list with varying details.

```

complex_t c3, c4, c5;
c3.r = 30;
c3.im = 31;
c4.r = 40;
c4.im = 41;
c5.r = 50;
c5.im = 51;
array_t ( complex_t ) ac2;
pelib_alloc_buffer ( array_t ( complex_t ) ) ( &ac2, 3 );
pelib_init ( array_t ( complex_t ) ) ( &ac2 );
pelib_array_append ( complex_t ) ( &ac2, c3 );
pelib_array_append ( complex_t ) ( &ac2, c4 );
pelib_array_append ( complex_t ) ( &ac2, c5 );

// Two elements list, initialized
list_t e1, e2;
pelib_init ( list_t ) ( &e1 );
pelib_init ( list_t ) ( &e2 );

// Both element take each one array
e1.array = ac;
e2.array = ac2;
// Element 2 comes after element 1
e1.next = &e2;

// Print minimum details: number of elements in linked list
pelib_printf_detail ( list_t ) ( stdout, e1, 0 );
// Print minimum details about arrays in list elements, i.e
// ., their size
// Use array's minimum detail string function
pelib_printf_detail ( list_t ) ( stdout, e1, 1 );
// Print more details about the list and its element arrays
// Use array's string function, that in turn uses complex's
// string function
pelib_printf_detail ( list_t ) ( stdout, e1, 2 );

// Free arrays buffers
pelib_free_buffer ( array_t ( complex_t ) ) ( &ac );
pelib_free_buffer ( array_t ( complex_t ) ) ( &ac2 );

```


Appendix B

Crown Scheduler Software Architecture

We design our implementation of Crown Schedulers by decomposing and reusing most of its features from a scheduler to another. For instance, we decompose a scheduler in 3 subsystems: allocation, mapping and frequency scaling. In this section, we give details about each important part of our Crown Scheduler implementation architecture. Figure B.1 gives an overview of the scheduler hierarchy. All Crown Schedulers are specialized classes of *Scheduler* defined in the Pelib (sec. 7.1). The *Crown Scheduler* class provides very little additional features to the *Pelib Scheduler* class; we use it as a common ancestor to all Crown Scheduler in order to compose them.

Figure B.1: Overview on the Crown Scheduler hierarchy.

Figure B.2: Overview of the scheduler benchmark software environment and running process.

B.1 Crown Modular

This is the most simple Crown Scheduler. A modular Crown Scheduler is composed of 3 subsystems that perform *allocation*, *mapping* and *frequency scaling* and run them in sequence. For each subsystem, we provide a hierarchy of classes that can solve each scheduling step with a common interface. Figure B.2 shows the hierarchy available for the allocation step; we provide only an implementation of Allocation Fastest that we describe in Sec. 5.3.1. Hierarchies for mapping and scaling phases are shown in Fig. B.1 with other Crown Schedulers. We consider solvers for these two steps as schedulers because they produce output that is sufficient to generate a schedule, although all tasks of a schedule produced by a solver of the mapping step are scheduled to run at the same frequency.

We provide 3 classes to perform the mapping step: *CrownMappingILP* and *CrownMappingILPBalanced* uses pelib class *AmplSolver* to run the separated mapping ILP model we give in Sec. 5.3.2. *CrownMappingILPBalanced* additionally provides the improved load-balancing strategy illustrated in Fig. 5.3. We also provide our implementation of LTLG (Sec. 5.3.3) generalized to admit arbitrary crown configurations (Sec. 5.5.2). Similarly, we provide *CrownScalingILP* to perform the separated Crown Scaling step of Sec. 5.3.4 and *CrownScalingHeight* that runs our frequency scaling heuristic of Sec. 5.3.5.

B.2 Crown ILP Integrated

This is our ILP (Integer Linear Programming) integrated scheduler. It solves all allocation, mapping and frequency scaling into one phase. We includes core consolidation (Sec. 5.5.4) and island-aware scheduling (Sec. 5.5.5) features. This class relies on Pelib class *AmplSolver* to run AMPL with the ILP model we provide.

B.3 Crown Binary

This class implements the binary search strategy discussed in Sec. 5.3.6. It always use *CrownAllocationFastest* to find allocations, but it can be composed with instances of *CrownMapping* and *CrownScaling* to perform the mapping and allocation phases for scheduling.

Figure B.3: Overview of the scheduler benchmark software environment and running process.

B.4 Crown Composite

A Crown composite scheduler uses another scheduler class and perform additional work to provide improved solution. We use this class to implement the feedback loop optimization strategies of our simulated annealing technique (Sec. 5.3.7) and our Core Consolidation heuristic (Sec. 5.5.4). Because all Crown Composite schedulers are Schedulers, then can be composed with each other. We give an example of such a composition in Listing 7.10.

B.5 Crown Configuration

Figure B.3 illustrate the class hierarchy we use to implement our crown configuration strategies. As discussed in Sec. 5.5.2, we provide the balanced binary configuration in class *CrownConfigBinary*, the *organ* decomposition in *CrownConfigOrgan* and the task graph-based configuration in *CrownConfigTaskgraph*. Note that the balanced binary configuration also ensures to produce a correct setting in the presence of islands (Sec. 5.5.5) and that it is the default configuration for all our Crown Schedulers.

Appendix C

Freja

This chapter provides a more extended example than Sec. 7.3 on the use of Freja and its main features. In our example, we propose to study the difference in processing time between math functions `sqrt ( )` and `log ( )`, both sequentially. We want to observe both the execution time and the number of processor ticks to process them.

C.1 Experiment Scenario

We begin with a simple C program able to produce the desired behavior. Listing C.1 gives the C program we use for our experiment. Note the preprocessor directives *MAX* and *SQRT* that makes the compiler to generate varying binaries after compiler options. In order to have better time measurement, we run the operation we want to assess several times and we can measure the overall time; preprocessor symbol *MAX* defines the number of repetitions. We also use the symbol *SQRT* to define whether we want to monitor square root or logarithm. In the case of logarithm, we use the program's first argument at runtime for the logarithm base. Finally, we monitor both the execution time in seconds (with a precision of up to a nanosecond) and in clock cycles, using the hardware instruction *rtdsc* and we output in one line the starting time in second and in nanoseconds, then the stop time in seconds and nanoseconds and the start and stop time in clock ticks. The actual execution time in seconds and clock ticks can be later calculated from these numbers.

Listing C.1: C program we use for our experiment

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>
// From http://stackoverflow.com/questions/17401914/why-should-i-use-rdtsc-diferently-on-x86-and-x86-x64
size_t rdtsc ( ) {
    unsigned long lo, hi;
    asm ( "rdtsc" : "=a" ( lo ), "=d" ( hi ) );
    return ( hi << 32 |} lo );
}
#if ! SQR
float ln ( float i, float base ) {
    return logf ( i ) / logf ( base );
}
#endif
int main ( int argc, char **argv ) {
    size_t rdtsc_start, rdtsc_stop;
    struct timespec timespec_start, timespec_stop;
    clock_gettime ( CLOCK_MONOTONIC, &timespec_start );
    rdtsc_start = rdtsc ( );
    #if ! SQR
    if ( argc < 2 ) {
        fprintf ( stderr, "Please give log base as argument\n" );
        return EXIT_FAILURE;
    }
    float base = atof ( argv[1] );
    #endif
    size_t i;
    for ( i = 0; i < MAX; i++ ) {
    #if SQR
        sqrt ( i );
    #else
        ln ( i, base );
    #endif
    }
    rdtsc_stop = rdtsc ( );
    clock_gettime ( CLOCK_MONOTONIC, &timespec_stop );
    printf ( "%zd %zd %zd %zd %zu %zu\n", timespec_start.tv_sec,
        timespec_start.tv_nsec, timespec_stop.tv_sec, timespec_
        stop.tv_nsec, rdtsc_start, rdtsc_stop );
    return EXIT_SUCCESS;
}

```

C.2 Preparing a Set of Experiments

In this section, we give the details on how to prepare and experiment with Freja. We derive from preprocessor symbols and the program's runtime argument of Listing C.1, a list of variables for our experiments. Because of preprocessor symbols *MAX* and *SQRT*, we need to define the number of iterations and the operation under test while compiling every program variants. This is done in Freja's *variable* file shown in Listing C.2, between lines 1 and 3. We further define the logarithm base values we want to test. We place it in the list *run* (line 5) since this is a variable whose value is set at runtime and we set its possible values to 2 and 10 (line 6). Note the second runtime variable *try* taking values 1 to 5 (lines 5 and 7). We do not use this value anywhere in our experiment, but it allows Freja to run several times an experiment variant and measure how the results it produce it stable for run to run.

Line 9 gives a name for each space separated values produced when running an experiment. As described above, we have the starting time in seconds and nanoseconds, the stopping time in seconds and nanoseconds, and the starting and stopping time in clock ticks. With line 11, we explicitly state that any combination of environment setting variables is valid and Freja should prepare the environment according to each of them. We need to state it explicitly because Freja would otherwise use implicitly the expression of line 12, that expresses the experiments to run and to set up if not expression is given for it. Both lines consists in a boolean expression interpreted with Bash [42]. Each literal in the expression is either the value returned when executing a program or a test of the value of a variable against a sed-interpreted regular expression. We can express the latter with the function *match()* that takes as parameter the variable's name and the regular expression its value should be tested against, between double quotes. Here, we allow every variable combinations when setting up the environment and we restrict the experiments to run to either logarithm operations of any base, or square root with only one base value because we define in Listing C.1 that the square root operation variant does not use the value of variable *base*.

Listing C.2: Freja experiment variables for our example (in file *variables*).

```
compile= ( op max)
op="sqrt log"
max="for i in $( seq 30 32); do echo $ ( ( 2 ** $i)); done"

run= ( base try)
base="2 10"
try="seq 1 5"

output_columns="start_sec start_nsec stop_sec stop_nsec
start_rtdsc stop_rtdsc"

restrict_compile="/bin/true"
restrict='match ( op, "^sqrt$" ) && match ( base, "^$ ( echo
$base |} tr -s ' ' |} cut -f 1 -d ' ' )'$" ) ||} match ( op,
"^log" )'
```

C.2.1 Preparing Environment Settings

In this section, we give details on how to use Freja to set up the environment so that Freja can run all experiments. Following our scenario of Sec. C.1, this consists in compiling all variants of the C program shown in Listing C.1. We first give in Listing C.3 a simple Makefile [47] able to generate a different executable depending on the compiling options we provide. Note that the binary filename Make produces with this Makefile varies depending on options *MAX* and *SQRT*. Consult the documentation of GNU make for more information about this Makefile.

Listing C.3: Freja experiment variables for our example (in file *compile*).

```
BASE=1
OP=sqrt

test-$ ( MAX)-$ ( BASE):
gcc -O3 -DMAX=$ ( MAX) -DSQRT=$ ( if $ ( filter $ ( OP),sqrt)
,1,0) test.c -o test-$ ( MAX)-$ ( OP) -lm
```

Listing C.4 gives the script Freja uses to setup the environment. In lines 1 and 2, it collects the values of variables *max* and *op* that define the number of iterations and the operation to monitor, respectively. In line 4, it calls make to produce one program file the variable combination received as arguments.

Listing C.4: Freja experiment setting up script for our scenario(in file *run*).

```

1 op=$1
2 max=$2
3
4 make MAX=$max OP=$op

```

C.2.2 Preparing Experiment Runs

We provide details to prepare Freja to run our experiment. Listing C.5 gives the script Freja uses to run each experiment variant. In lines 1 and 2, we read the variable used to set the environment (Sec. C.2.1) and lines 4 and 5 read the remaining arguments. Finally, line 8 runs the corresponding program variant. In order to illustrate the error handling feature of Freja, we do not provide the base value as argument to the program; this will make some experiments to fail and Freja will not take them into account, until the problem is solved. Freja capture any data produced on standard output within this script, and interpret is as result. For this reason, it is important that the program we run outputs nothing on standard output, but the the result it is meant to produce. This output takes shape in a matrix of space-separated values.

Listing C.5: Freja experiment setting up script for our scenario.

```

1 op=$1
2 max=$2
3
4 base=$3
5 try=$4
6
7 ## This will generate errors
8 ./test-$max-$op

```

C.3 Running Experiments

One we set Freja up to prepare the experimental environment and to run every experiment variants, we can instruct Freja to actually set the environment up by running the corresponding script for each possible variable combination. In our example, this compile a program variant for each of them. Listing C.6 gives the command to make Freja to prepare everything. This assumes that the files *variables*, *compile* and *run* exist and contain Listings 7.11, 7.12 and 7.14, respectively.

Listing C.6: Using Freja to perform the the environment setting work using the scripts defined in Sec C.2.

```

1 user@host:~$ freja compile
2 [INFO] Will compile 6 settings
3 [INFO] Setting 1/6: op=sqrt max=1073741824... OK. 0.10 sec
   finished on 09/28/16 15:46:19; approx. 0.58 sec left (
   0.01 minutes)
4 [INFO] Setting 2/6: op=sqrt max=2147483648... OK. 0.09 sec
   finished on 09/28/16 15:46:19; approx. 0.45 sec left (
   0.01 minutes)
5 [INFO] Setting 3/6: op=sqrt max=4294967296... OK. 0.09 sec
   finished on 09/28/16 15:46:19; approx. 0.33 sec left (
   0.01 minutes)
6 [INFO] Setting 4/6: op=log max=1073741824... OK. 0.10 sec
   finished on 09/28/16 15:46:20; approx. 0.22 sec left (
   0.00 minutes)
7 [INFO] Setting 5/6: op=log max=2147483648... OK. 0.10 sec
   finished on 09/28/16 15:46:20; approx. 0.11 sec left (
   0.00 minutes)
8 [INFO] Setting 6/6: op=log max=4294967296... OK. 0.10 sec
   finished on 09/28/16 15:46:20; approx. 0.00 sec left (
   0.00 minutes)
9 [INFO] Success. Compilation took 0.71 seconds ( 0.01 minutes
   ) or 0.12 seconds per compilation setting ( 0.00 minutes)
   .

```

We can see that Freja runs 12 times the environment setting script with all valid combination of variables *op* and *max*; it call *instance* each of these combinations. Note that for each run, Freja reports if the operation is successful or not. This is determined by the value returned by the environment setting up script. if it returns 0, then Freja considers the script ran successfully; otherwise it reports it as a failure. If all instances are reported successful, then Freja report that the environment was correctly set up. Freja tries to give a rough estimate of the remaining time after each run of the script. It considers the time to run all remaining instances is proportional to the number of instances already run and the total time spent on them. In our example and after Freja terminates and reports a success, the compiled program variants are generated in files *test-max-op* where $max \in \bigcup_{i \in 30..32} 2^i$ and $op \in \{\text{sqrt}, \text{log}\}$.

At this point, we can use Freja to run all experiments and collect data. Listing C.7 gives the Freja command to run all experiments and the output obtained. We need to provides Freja a name for the experiment set running; here, we use *test1*. On Listing C.7, we remove output lines where the instance numbers increase linearly. We see that there is no output line for instance 6, 7, 8, 9 or 10. This is because these lines correspond to running square root with a base value other than 2, which is not permitted by Line 12 of Listing C.2. However, Freja runs all experiment instances 61 to 120 as

they run operation *log* for every base value defined in Listing C.2. We see also that all instances for operation *log* failed. This is because we do not provide the base argument when running the corresponding binary program in Listing C.5.

Listing C.7: Using Freja to perform the the environment setting work using the scripts defined in Sec C.2.

```

user@host:~$ freja run test1
[INFO] Will run 60 instances
[INFO] Instance 1/60: op=sqrt max=1073741824 base=2 try=1...
      OK. 1.13 sec finished on 09/28/16 15:48:12; approx.
      71.12 sec left ( 1.19 minutes)
...
[INFO] Instance 5/60: op=sqrt max=1073741824 base=2 try=5...
      OK. 1.12 sec finished on 09/28/16 15:48:17; approx.
      63.72 sec left ( 1.06 minutes)
[INFO] Instance 11/60: op=sqrt max=2147483648 base=2 try
      =1... OK. 2.17 sec finished on 09/28/16 15:48:19; approx.
      36.33 sec left ( 0.61 minutes)
...
[INFO] Instance 15/60: op=sqrt max=2147483648 base=2 try
      =5... OK. 2.11 sec finished on 09/28/16 15:48:28; approx.
      50.59 sec left ( 0.84 minutes)
[INFO] Instance 21/60: op=sqrt max=4294967296 base=2 try
      =1... OK. 4.18 sec finished on 09/28/16 15:48:32; approx.
      39.69 sec left ( 0.66 minutes)
...
[INFO] Instance 25/60: op=sqrt max=4294967296 base=2 try
      =5... OK. 4.46 sec finished on 09/28/16 15:48:50; approx.
      54.67 sec left ( 0.91 minutes)
[INFO] Instance 31/60: op=log max=1073741824 base=2 try=1...
      Failed. 0.02 sec finished on 09/28/16 15:48:50; approx.
      36.77 sec left ( 0.61 minutes)
...
[INFO] Instance 60/60: op=log max=4294967296 base=10 try
      =5... Failed. 0.02 sec finished on 09/28/16 15:48:52;
      approx. 0.00 sec left ( 0.00 minutes)
[INFO] Failed. Batch took 41.48 seconds ( 0.69 minutes) to
      run, or 0.69 seconds per instance run ( 0.01 minutes).
[INFO] Merging all results to test1/27-09-16:17-25-30_CEST/
      table.csv ... done.
[INFO] Merging all results to test1/table.csv ... done.

```

If we replace lines 7 and 8 of Listing C.5 by Listing C.8 and run Freja again for the same experiment name, Freja find automatically what experiments were already successfully run and tries again to run all the ones that failed in previous attempts; we obtain the trace in Listing C.9. We see that Freja *Skipped* all instances that ran correctly in the previous attempt, and

ran successfully all other instances that previously failed. We can see that Freja concludes that the whole experiment ran successfully.

Listing C.8: Freja experiment setting up script for our scenario.

```
1 ## This is correct
2 ./test-$max-$op $base
```

Listing C.9: Trace from Freja after having fixed Listing C.5 with Listing C.8.

```
user@host:~$ freja run test1
[INFO] Will run 60 instances
[INFO] Instance 1/60: op=sqrt max=1073741824 base=2 try=1...
      Skipped. 0.00 sec finished on 09/28/16 15:50:14; approx.
      4.18 sec left ( 0.07 minutes)
...
[INFO] Instance 5/60: op=sqrt max=1073741824 base=2 try=5...
      Skipped. 0.00 sec finished on 09/28/16 15:50:15; approx.
      16.75 sec left ( 0.28 minutes)
[INFO] Instance 11/60: op=sqrt max=2147483648 base=2 try
      =1... Skipped. 0.00 sec finished on 09/28/16 15:50:15;
      approx. 5.29 sec left ( 0.09 minutes)
...
[INFO] Instance 15/60: op=sqrt max=2147483648 base=2 try
      =5... Skipped. 0.01 sec finished on 09/28/16 15:50:15;
      approx. 6.96 sec left ( 0.12 minutes)
[INFO] Instance 21/60: op=sqrt max=4294967296 base=2 try
      =1... Skipped. 0.00 sec finished on 09/28/16 15:50:15;
      approx. 3.93 sec left ( 0.07 minutes)
...
[INFO] Instance 25/60: op=sqrt max=4294967296 base=2 try
      =5... Skipped. 0.00 sec finished on 09/28/16 15:50:16;
      approx. 4.33 sec left ( 0.07 minutes)
[INFO] Instance 31/60: op=log max=1073741824 base=2 try=1...
      OK. 1.05 sec finished on 09/28/16 15:50:17; approx. 4.55
      sec left ( 0.08 minutes)
...
[INFO] Instance 60/60: op=log max=4294967296 base=10 try
      =5... OK. 4.37 sec finished on 09/28/16 15:51:34; approx.
      0.00 sec left ( 0.00 minutes)
[INFO] Success. Batch took 79.49 seconds ( 1.32 minutes) to
      run, or 1.32 seconds per instance run ( 0.02 minutes).
[INFO] Merging all results to test1/27-09-16:17-25-30_CEST/
      table.csv ... done.
[INFO] Merging all results to test1/table.csv ... done.
```

After running an experiment, Freja generates the files *labels.r* and *table.csv*. The former contains useful information to format plots generated

by R [118] and ggplot2 [46] from the data, and is discussed in Sec. C.5. The latter contains all the data collected while running the experiment, merged from all individual experiment results in subdirectory *test1/<date>/*, named after the experiment name and the date. Running Freja again results in running all instances again and store them in another subdirectory of *test1/*. The we call *experiment instance* the name of these subdirectory.

C.4 Manual Error Handling

let us assume that after following instructions of Sec. C.3, we realize that although Freja reported all instances as successful, some of them are actually wrong. This can be due to the code not measuring the exact desired value for example. It is possible to run manually again all the faulty experiments. Assuming the experiment name is *test1*, we first need to select the experiment instance to be fixed by its name. All experiment instances names are available as the immediate subdirectories's name, of the directory named after the experiment (here, *test1*). Assume we want to fix experiment instance *27-09-16:17-25-30_CEST* of experiment *test1* for all instances with operation *log* and 5 iterations; Listing C.10 gives the command to achieve it.

Listing C.10: Re-run instances of experiment instance *27-09-16:17-25-30_CEST* of experiment *test1* (we cut output lines of instances within a continuous serie of skipped or re-run instances).

```

user@host:~$ freja run test1 --instance 27-09-16:17-25-30\_
  CEST --retry --pattern 'match ( op, "log") && match ( max
    , "1073741824")'
[INFO] Processing retry list. This can take a while ... [
  INFO] Will run 60 instances
[INFO] Instance 1/60: op=sqrt max=1073741824 base=2 try=1...
  Skipped. 0.02 sec finished on 09/28/16 15:53:25; approx.
  6.53 sec left ( 0.11 minutes)
...
[INFO] Instance 5/60: op=sqrt max=1073741824 base=2 try=5...
  Skipped. 0.02 sec finished on 09/28/16 15:53:25; approx.
  25.52 sec left ( 0.43 minutes)
[INFO] Instance 11/60: op=sqrt max=2147483648 base=2 try
  =1... Skipped. 0.02 sec finished on 09/28/16 15:53:26;
  approx. 7.78 sec left ( 0.13 minutes)
...
[INFO] Instance 15/60: op=sqrt max=2147483648 base=2 try
  =5... Skipped. 0.03 sec finished on 09/28/16 15:53:26;
  approx. 10.10 sec left ( 0.17 minutes)
[INFO] Instance 21/60: op=sqrt max=4294967296 base=2 try
  =1... Skipped. 0.02 sec finished on 09/28/16 15:53:26;
  approx. 5.57 sec left ( 0.09 minutes)
...
[INFO] Instance 25/60: op=sqrt max=4294967296 base=2 try
  =5... Skipped. 0.03 sec finished on 09/28/16 15:53:27;
  approx. 6.22 sec left ( 0.10 minutes)
[INFO] Instance 31/60: op=log max=1073741824 base=2 try=1...
  OK. 1.11 sec finished on 09/28/16 15:53:28; approx. 5.78
  sec left ( 0.10 minutes)
...
[INFO] Instance 40/60: op=log max=1073741824 base=10 try
  =5... OK. 1.11 sec finished on 09/28/16 15:53:39; approx.
  11.09 sec left ( 0.18 minutes)
[INFO] Instance 41/60: op=log max=2147483648 base=2 try=1...
  Skipped. 0.03 sec finished on 09/28/16 15:53:39; approx.
  10.62 sec left ( 0.18 minutes)
...
[INFO] Instance 60/60: op=log max=4294967296 base=10 try
  =5... Skipped. 0.04 sec finished on 09/28/16 15:53:41;
  approx. 0.00 sec left ( 0.00 minutes)
[INFO] Success. Batch took 16.10 seconds ( 0.27 minutes) to
  run, or 0.27 seconds per instance run ( 0.00 minutes).
[INFO] Merging all results to test1/27-09-16:17-25-30_CEST/
  table.csv ... done.
[INFO] Merging all results to test1/table.csv ... done.

```

The new, fixed results are automatically merged into *table.csv*, together with results from other experiment instances, if any.

C.5 Plotting

We review here the features Freja provides to help at generating figures from the data produced following the example procedure given in Sec. C.3. We base our example on R [118] and ggplot2 [46] to process the data and produce visual plots from it. This section is not about R or ggplot2. Please consult their documentation on internet for more information about them. Freja stores all the data collected in an experiment in *table.csv* in CSV (**C**omma **S**eparated **V**alues) format, that R can read directly. Listing C.11 gives an example of the use R and ggplot2 to generate plots from Freja-generated data.

Listing C.11: Using R and ggplot2 to generate plots from data produced by Freja.

```

1  #!/usr/bin/env Rscript
2  library ( ggplot2 )
3  library ( plyr )
4  options ( error = function ( ) {traceback ( 2 ); dump.frames ;
      stopifnot ( FALSE ) } )
5  data.frame = read.csv ( "table.csv" )
6  source ( "labels.r" )
7  nsec2sec = function ( nsec ) {
8    return ( nsec / 1000000000 )
9  }
10 data.frame = ddply (
11   data.frame, c ( "op", "max", "base", "try" ), summarize ,
12   time = ( stop_sec + nsec2sec ( stop_nsec ) - start_sec -
      nsec2sec ( start_nsec ) ) / max ,
13   tick = ( stop_rtdsc - start_rtdsc ) / max
14 )
15 plot = ggplot ( data = apply_labels ( data.frame ) ,
16   aes ( interaction ( op, base, sep = " " ), time, fill =
      interaction ( op, base, sep = " " ) ) ) +
17   geom_boxplot ( ) +
18   ylab ( label ( "time" ) ) +
19   xlab ( label ( "op.base" ) ) +
20   ggtitle ( "Running time for loops of square roots or
      logarithm" ) +
21   scale_fill_manual ( name = "interaction ( op, base, sep = \"
      \")", values = freja_colors ( data.frame, c ( "op", "base"
      ) ), inter = " " ), guide = FALSE ) +
22   scale_color_manual ( name = "interaction ( op, base, sep = \"
      \")", values = freja_colors ( data.frame, c ( "op", "base"
      ) ), inter = " " ), guide = FALSE )
23 ggsave ( file="timing.svg", plot=plot, width=8, height=6 )

```

Line 4 makes R to print a trace to screen when the script fails; it helps considerably at finding the source of any error. In lines 5 and 6, we read the data and we load Freja-generated sets of functions that helps generating figures easier to read; this is covered in the remainder of this section. In lines 7 to 15, we compute the time in second and clock tick from the raw values output by our program in read from the data file in line 4, and we compute their mean and standard deviation. Lines 16 to 25 generate a box plot from the data collected and processed. Read documentation about R packages *plyr* and *ggplot2* fore more details about processing and plotting data with R.

Lines 6, 16, 21, 22, 23 and 24 illustrate the use of Freja-provided R functions aiming at providing a flexible way to manipulate texts and colors and

keep them consistent across figures. Function `apply_labels()` and `label()` of lines 16 and 20-21, replace the raw text found in data by strings easier to read, and defined in Freja's experiment description file (Listing C.2 and the addendum required shown in Listing C.12). Using these functions, it is even possible to redefine the text in final figures without breaking the plotting script. `freja_colors()` in lines 23 and 24 generate a vector of colors for each line of our plot, from colors defined by the user, once again in the experiment description file Listing C.2 and the addendum required shown in Listing C.12. In order to let R and ggplot2 to use the colors provided by Freja, it is important that all `interaction()` statements of lines 17, 23 and 24 are identical, and that all columns mentioned in them are also in the list of columns given as second argument of `freja_colors()`. Finally, all `sep` and `inter` arguments must be set to the same value in all `interaction()` and `freja_color()` statements.

Listing C.12: Addendum to Listing C.2 to define strings and colors to be shown in the final plots generated by R script of Listing C.11

```

1 labels_op="log: Logarithm, base; sqrt: Square Root, ^1 /"
2 labels_columns="time: Time in seconds; op.base: Operation"
3
4 combination="op_base"
5 colors_op_base="log.2: #DD7777; log.10: #882222; sqrt.2:
  #7777DD"
```

After adding the content of Listing C.12 to Listing C.2, we can run Freja again to regenerate `label.r` and use the R plotting script (See Listing C.13) and obtain a figure as shown in Fig. C.1. We see that in our test, square root and logarithm operations take about the same time on average, regardless of the base of logarithm employed.

Listing C.13: Use Freja to regenerate Freja text replacement functions and run the R plotting script (assuming Listing C.11 is saved in file `drawplots.r`) and obtain a SVG figure file of Fig. C.1.

```

user@host:~$ freja label
user@host:~$ ./drawplots
```

Figure C.1: Figure obtained after running Freja experiment and plot the data it produces.

C.6 Additional Options

In Sec. C.1 to C.5, we give a scenario that involves most features of Freja. In this section, we cover all operation and options Freja can perform. A Freja command consists in an operation and options that affects its execution; its general structure is

```
freja <operation> <options>
```

Table C.1 gives a quick description on operations supported by Freja. Table C.2 gives the list of options Freja supports for each operation.

Operation	Description
<code>sources</code>	Gives the filename of a package containing all Freja's source files.
<code>compile</code>	Prepares the environment before running an experiment. This typically consists in compiling program variants to be tested.
<code>compile-test</code>	Sets the environment of a specific combination of variable values. Requires additional arguments in the form <code>variable=value</code> to give a value for every variable in the experiment description file.
<code>run-test</code> <code>run</code>	Same as <code>compile-test</code> for running a specific instance. Run all instances for which there is no data produced by a successful previous run, if any. Requires a name for the experiment begin run, as additional argument.
<code>merge</code>	Merge all data found for a given experiment into <i>table.csv</i> . Requires a name for the experiment begin run, as additional argument.
<code>label</code>	Reads the experiment description and generate <i>labels.r</i> that contains values aliases and colors for use in a R script that generates plots for data.
<code>convert</code>	Convert data from a CSV (C omma- S eparated V alues) file such as <i>table.csv</i> produced by Freja, to a Matlab function script that contains all data. Requires the name of the input file as first argument after operation and <code>--output</code> argument; see Table C.2 for more information.
<code>fuse</code>	Fuses a list of data files in CSV format such as <i>table.csv</i> into a unique one, with an additional column. The first argument for this operation is the new column name, then options as described in Table C.2, token <code>--</code> and the list of all filenames to be merged together. The new column takes as value the position of the corresponding file in the list, starting with 0. Example: <code>freja fuse variants -- table.csv test1/table.csv.</code>

Table C.1: Operations for Freja.

Operation	Operation	Description
--retry	compile, run	Run a subset of instances again, replacing the data already existing. Requires exactly one of the following additional options: <ul style="list-style-type: none"> --all Run all instances again --number With an additional list of space-separated integers, run again all corresponding instances. --pattern With an additional boolean expression, run again all instances that satisfy the expression. See Listing C.10 for an example.
--retry-only	compile, run	Requires --retry and --count-selected-runs-only options. Do not attempt to run other instances than the one matching the retry constraint (e.g. instances that do not match the constraint but for which there is no data produced by any valid previous execution).
--instance	run	Gives a name to the instance of experiment being run. By default, the current date and time with the precision of a second.
--compile <file>	compile	Instructs Freja to use <file> as environment setting script instead of compile.
--run <file>	run	Instructs Freja to use <file> as experiment running script instead of run.
--setting <file>	any	Load Freja's settings from <file> instead of /etc/freja/settings or \$HOME/.freja/settings.
--variables <file>	any	Load experiment description from <file> instead of variables.
--output <file>	convert	Saves all data files being merged into <file>.
--log <file>	compile, run	Stores Freja's trace as well as error output produced by the script running. If operation is compile, then also capture standard output.
--count-selected-runs-only	run, compile	Only show instances selected by the retry constraint (very slow).

Table C.2: Operations and options for Freja.

Appendix D

Mimer

This chapter gives extended information on important plugins required to run Mimer. Section D.1 gives details on the interface expected by Mimer to run schedule evaluators. Section D.2 describes the design of data analysis and plotting scripts for Mimer and Sec. D.3 covers how to describe a scheduling experiment with Mimer.

D.1 Energy Evaluation

An energy evaluator can use an energy model to evaluate the quality of a schedule, or run the corresponding application with Schedeval or Drake described in Sec. 6. Mimer expects its output as scalars and one or several matrices in AMPL output format, showing the power consumption as a function of time. The power unit is not relevant here, as Mimer's phases 3 and 4 are the one interpreting the data produced. Scalar values and one matrix can then be captured in Mimer's phase 3. Listing D.1 shows a possible example of the data produced by schedule evaluator.

--taskgraph	Input task graph description filename in GraphML format.
--platform	Input platform description file in AMPL input format.
--schedule	Schedule to assess in XML format.
--label	Write a short descriptive string of the energy evaluation model.

Table D.1: Options required by Mimer for a Schedule evaluator command line interface.

```

#core + mmc power consumption[W]
time__power  [*,*]
: 0 1 :=
0 0.07 44.8901
1 103.175 44.8393
2 213.169 44.8393
3 323.172 41.2101
;

#core power consumption[W]
time__chippower  [*,*]
: 0 1 :=
0 0.07 31.982
1 103.175 31.982
2 213.169 31.982
;

```

Mimer requires schedule evaluators to admit the options shown in Table D.1. We provide an energy evaluator based on the energy model described in Sec. 5.6.2 through the command *static_dynamic_busy_active*.

D.2 Data Analysis

Both phases 3 and 4 of Mimer’s workflow includes the gathering of data of each problem instance solved and assessed in the previous steps. Mimer expects an executable script that admits options shown in Table D.2.

Since we updated Mimer to separate analysis and plotting tools, we published our paper on core consolidation [107] and we provide the analysis tools we employed then.

D.3 Mimer Experiment

A scheduling *experiment* (Listing D.1) defines in lines 1 to 6 a list of task graphs to be scheduled, a list of target platforms, a list of schedulers to be

<code>--settings</code>	Defines variables <i>scheduler_scalars</i> , <i>evaluation_scalar</i> and <i>analysis_scalar</i> , as space separated lists of the data collected from phase 1, 2 and 2, respectively. Values read from phase 1 come from the values schedulers produce in A.6 or from <i>pelib-scheduler</i> 's standard output (or any other compatible scheduler's standard output). This options also defines <i>evaluation_matrix</i> , the name of the matrix Mimer will read from schedule evaluators' output (exactly one). Mimer reads these definitions by "sourcing" the executable script with a bash line such as <code>. analysis</code> where <i>analysis</i> is the name of the script given to Mimer to run its phase 3.
<code>--analyse</code>	Takes input problem data and produce output more information. Additional option <code>--taskgraph</code> gives the filename of the task graph description in GraphML format, option <code>--platform</code> gives the filename of platform description in AMPL Input format and <code>--schedule</code> gives the filename of schedule in XML format.
<code>--plot</code>	Use the data produced in phase 3 to generate figures. Additional option <code>--batch</code> specified the mimer experiment name for which Mimer attempts to produce figures. This matches the file name of the experiment description explained in Sec. D.3.
<code>--sources</code>	Prints the filename of the package containing the source files of the analysis and plotting script being called.

Table D.2: Mimer command line options.

used, a list of evaluators to assess the schedules generated and a plotting process to gather and represent the data generated. For platform and task graph, it is possible to give a filename pattern and Mimer will include all filenames it finds in environment variables `MIMER_TASKGRAPH_PATH` and `MIMER_PLATFORM_PATH`, that match the pattern. It is important that all scheduler and evaluator names are valid variable names for bash. In particular, do not employ any character such as “.” or “-”. In lines 7, we set the number of time Mimer should run each scheduler; this can be useful to obtain reliable numbers if some schedulers rely on a random process to find solution, like our simulated annealing scheduler. Similarly in line 8, we set the number of time each schedule should be evaluated, so that Mimer returns more reliable number when measuring energy consumption with a real platform. Lines 9 to 20 define the actual commands necessary to run schedulers and energy evaluators. Note that variable names match the ones of lines 4 and 5. Mimer append to these commands the relevant options shown in A.5 to add input data such as task graph and target platform descriptions, as well as all other options for all other actions required by Mimer. Finally, line 21 describe what experiment among all possible combinations of parameters presented above should be run. This is a boolean formula in bash syntax, where literals are outputs of function `match` that takes an experiment parameter name (one of `platform`, `taskgraph`, `evaluation`, `scheduler`, `schedule_try` or `evaluation_try`) and a regular expression to test the value of the corresponding parameter. In this example, we allow experiments with a platform description that does not end with the string `_p32-i` and one or several digits, unless we experiment with our integrated crown scheduler variants.

Mimer must be invoked for each phase *schedule*, *assess*, *analyze* and *plot*. Lines 1 to 4 of Listing D.2 shows how to run mimer for all 4 phases. Line 5 produces an archive that contains all problem input data, the solutions produced while experimenting as well as schedulers, evaluators’ and analyzer source files and experiment description.

Listing D.2: Mimer commands to start Mimer’s 4 steps. *batch* denotes the experiment description file as shown in Listing D.1.

```

1 mimer schedule ./batch
2 mimer assess  ./batch
3 mimer analyse ./batch
4 mimer plot    ./batch
5 mimer archive ./batch

```


Listing D.1: Description of a complete scheduling experiment and plotting

```

1  #!/bin/bash
2  platform="intel_i5_4690k_haswell_p*.dat"
3  taskgraph="{fft_n,mergesort_n,preduce_n}*.graphml"
4  evaluation="static_dynamic_busy_off"
5  scheduler="crown_ilp_integrated_binary_idle crown_ilp_
    integrated_binary crown_ilp_integrated_organ crown_ilp_
    integrated_taskgraph ilp_optimal_scheduler crown_binary
    crown_binary_annealing pruhs_nlp pruhs_heuristic xu_ilp"
6  analysis="mimer-islands-analysis"
7  schedule_try="'seq 1 3'"
8  evaluation_try="'seq 1 5'"
9  crown_ilp_integrated_binary_idle="crown --scheduler crown-
    ilp-integrated --options --show-stderr --show-stdout --
    config --lib config-binary --zeta 1 -- "
10 crown_ilp_integrated_binary="crown --scheduler crown-ilp-
    integrated --options --show-stderr --show-stdout --config
    --lib config-binary -- "
11 crown_ilp_integrated_organ="crown --scheduler crown-ilp-
    integrated --options --show-stderr --show-stdout --config
    --lib config-organ -- "
12 crown_ilp_integrated_taskgraph="crown --scheduler crown-ilp-
    integrated --options --show-stderr --show-stdout --config
    --lib config-taskgraph -- "
13 ilp_optimal_scheduler="ilp-optimal-schedule --scheduler ilp-
    optimal-scheduler --options --show-stderr --show-stdout
    -- "
14 crown_binary="crown --scheduler crown-binary "
15 crown_binary_annealing="crown --scheduler crown-composite-
    consolidation --options '--args --composite --lib crown-
    composite-annealing --args --composite --lib crown-binary
    ' -- "
16 pruhs_nlp="pruhs-nlp --options --show-stderr --show-stdout
    -- "
17 pruhs_heuristic="pruhs-heuristic "
18 xu_ilp="xu-ilp --options --show-stderr --show-stdout -- "
19 static_dynamic_busy_off="static_dynamic_busy_active"
20 static_dynamic_busy_idle="static_dynamic_busy_active --zeta
    1"
21 restrict='! match ( platform , "_p32-i[0-9]\+$" ) ||} match (
    scheduler , "^crown_ilp_integrated_" )'

```

Bibliography

- [1] Adapteva. Epiphany multicore intellectual property, 2014. URL http://www.adapteva.com/docs/epiphany_arch_ref.pdf. Last access: February 2015.
- [2] T. Ainsworth and T. Pinkston. On characterizing performance of the Cell Broadband Engine Element Interconnect Bus. In *First International Symposium on Networks-on-Chip (NOCS 2007)*, pages 18–29, May 2007. doi: 10.1109/NOCS.2007.34.
- [3] G. Akl. *Parallel sorting algorithms*. Academic press, 1985.
- [4] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fast-flow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2012.
- [5] A. Amoura, E. Bampis, C. Kenyon, and Y. Manoussakis. Scheduling independent multiprocessor tasks. In R. Burkard and G. Woeginger, editors, *Algorithms – ESA ’97*, volume 1284 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1997. doi: 10.1007/3-540-63397-9_1.
- [6] A. Andrei, M. T. Schmitz, P. Eles, Z. Peng, and B. M. A. Hashimi. Quasi-static voltage scaling for energy minimization with time constraints. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE ’05, pages 514–519, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2288-2. doi: 10.1109/DATE.2005.250. URL <http://dx.doi.org/10.1109/DATE.2005.250>.
- [7] K. Avdic, N. Melot, C. Kessler, and J. Keller. Parallel sorting on Intel Single-Chip Cloud Computer. In *Proc. A4MMC workshop on applications for multi- and many-core processors at ISCA-2011*, 2011. ISBN 978-3-86644-717-2. URL <http://liu.diva-portal.org/smash/record.jsf?pid=diva2%3A624512>.

- [8] K. Avdic, N. Melot, C. Kessler, and J. Keller. Pipelined parallel sorting on the intel SCC. In *Fourth Swedish Workshop on Multi-Core Computing MCC-2011*, volume S. 96-101, pages 96–101. Linköping University, 2011.
- [9] H. M. Aziz, M. Fiedler, H. Grahn, and L. Lundberg. Adapting the streaming video based on the estimated position of the region of interest. In *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems*, pages 1–7, Nov 2012. doi: 10.1109/SITIS.2012.11.
- [10] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. on Computing*, 9(4):846–855, November 1980.
- [11] K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 72–75, 1990.
- [12] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: An introduction to ESTEREL. In *Proceedings of the First Franco-Japanese Symposium on Programming of Future Generation Computers*, pages 35–56, Amsterdam, The Netherlands, The Netherlands, 1988. Elsevier Science Publishers B. V. ISBN 0-444-70410-8. URL <http://dl.acm.org/citation.cfm?id=60661.60664>.
- [13] J. Blasewicz, M. Machowiak, J. Weglarz, M. Kovalyov, and D. Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan: Models and algorithms for planning and scheduling problems. *Annals of Operations Research*, 129:65–80, 2004.
- [14] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Cost considerations in network on chip. *Integration, the VLSI Journal*, 38(1):19 – 42, 2004. ISSN 0167-9260. doi: <http://dx.doi.org/10.1016/j.vlsi.2004.03.006>. URL <http://www.sciencedirect.com/science/article/pii/S0167926004000343>.
- [15] S. Borkar. The Exascale challenge. In *International Symposium on VLSI Design Automation and Test (VLSI-DAT)*, 2010, pages 2–3, April 2010. doi: 10.1109/VDAT.2010.5496640.
- [16] M. Broy and C. Dendorfer. Modelling operating system structures by timed stream processing functions. *Journal of Functional Programming*, 2:1–21, 1 1992. ISSN 1469-7653. doi: 10.1017/S0956796800000241.
- [17] J. Buurlage, T. Bannink, and A. Wits. Bulk-synchronous pseudo-streaming algorithms for many-core accelerators. *CoRR*, abs/1608.07200, 2016. URL <http://arxiv.org/abs/1608.07200>.

- [18] J. Castrillon, R. Leupers, and G. Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsoes. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, Feb 2013. ISSN 1551-3203. doi: 10.1109/TII.2011.2173941.
- [19] H.-L. Chan, J. Edmonds, and K. Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. *Theory Comput. Syst.*, 49(4):817–833, 2011.
- [20] A. P. Chandrasakaran and R. W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proc. IEEE*, 83(4):498–523, Apr. 1995.
- [21] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation – A performance view. *IBM Journal of Research and Development*, 51(5):559–572, Sept 2007. ISSN 0018-8646. doi: 10.1147/rd.515.0559.
- [22] I. Choi, M. Zhao, X. Yang, and D. Yeung. Experience with improving distributed shared cache performance on tilera’s tile processor. *Computer Architecture Letters*, 10(2):45–48, July 2011. ISSN 1556-6056. doi: 10.1109/L-CA.2011.18.
- [23] J. C. Chu and P. W. Dowd. Adaptive cache coherence over a high bandwidth broadband mesh network. *Parallel Computing*, 26(2–3): 285 – 311, 2000. ISSN 0167-8191. doi: [http://dx.doi.org/10.1016/S0167-8191\(99\)00106-4](http://dx.doi.org/10.1016/S0167-8191(99)00106-4). URL <http://www.sciencedirect.com/science/article/pii/S0167819199001064>.
- [24] P. Cichowski, J. Keller, and C. Kessler. Modelling power consumption of the Intel SCC. In *Proceedings of the 6th MARC Symposium*, pages 46–51. ONERA, July 2012.
- [25] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *Proc. Intl Conf. on High Performance Computing and Simulation (HPCS)*, pages 525–532, 2011.
- [26] E. Coffman, Jr., M. Garey, D. Johnson, and R. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.
- [27] N. A. Conos, S. Meguerdichian, F. Dabiri, and M. Potkonjak. Provably minimal energy using coordinated DVS and power gating. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014. doi: 10.7873/DATE.2014.307.
- [28] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

- [29] F. Dabiri, A. Vahdatpour, M. Potkonjak, and M. Sarrafzadeh. Energy minimization for real-time systems with non-convex and discrete operation modes. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09*, pages 1416–1421, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. ISBN 978-3-9810801-5-5. URL <http://dl.acm.org/citation.cfm?id=1874620.1874961>.
- [30] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. Cpu db: Recording microprocessor history. *Queue*, 10(4):10:10–10:27, Apr. 2012. ISSN 1542-7730. doi: 10.1145/2181796.2181798.
- [31] T. De Matteis and G. Mencagli. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '16*, pages 13:1–13:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4092-2. doi: 10.1145/2851141.2851148. URL <http://doi.acm.org/10.1145/2851141.2851148>.
- [32] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511.
- [33] C. Disco and B. van der Meulen. *Getting New Technologies Together: Studies in Making Sociotechnical Order*. De Gruyter studies in organization : innovation, technology, and organization. Walter de Gruyter, 1998. ISBN 9783110156300. URL <http://books.google.co.zw/books?id=1khs1Z-jbGEC>.
- [34] J. Edmonds and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. *ACM Trans. Algorithms*, 8(3):28, 2012.
- [35] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. *SIGARCH Comput. Archit. News*, 17(3):2–15, Apr. 1989. ISSN 0163-5964. doi: 10.1145/74926.74927. URL <http://doi.acm.org/10.1145/74926.74927>.
- [36] P. Eitschberger and J. Keller. Energy-efficient task scheduling in manycore processors with frequency scaling overhead. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 541–548, March 2015. doi: 10.1109/PDP.2015.64.
- [37] P. Eitschberger, J. Keller, F. Thiele, and C. Kessler. Exploring the placement of memory controllers on manycore processors: A case study for Intel SCC. In *Sixth Swedish Workshop on Multi-Core Computing MCC-2013*, pages 71–74, 2013.

- [38] J. Eker and J. W. Janneck. CAL language report: Specification of the CAL actor language. Technical report, University of California at Berkeley, 2003.
- [39] J. Falk, T. Schwarzer, M. Glab, J. Teich, C. Zebelein, and C. Haubelt. Quasi-static scheduling of data flow graphs in the presence of limited channel capacities. In *Embedded Systems For Real-time Multimedia (ESTIMedia), 2015 13th IEEE Symposium on*, pages 1–10, Oct 2015. doi: 10.1109/ESTIMedia.2015.7351766.
- [40] L. Fan, F. Zhang, G. Wang, and Z. Liu. An Effective Approximation Algorithm for the Malleable Parallel Task Scheduling Problem. *J. Parallel Distrib. Comput.*, 72(5):693–704, May 2012.
- [41] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press series. Thomson/Brooks/Cole, 2003. ISBN 9780534388096.
- [42] M. G. Bash programming - introduction how-to. <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>, 2000. Last accessed: 2012-08-06.
- [43] M. Garey and R. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975. doi: 10.1137/0204015.
- [44] B. Gedik, R. Bordawekar, and P. Yu. Cellsort: high performance sorting on the Cell processor. In *VLDB '07 Proceedings of the 33rd international conference on Very large data bases*, 2007.
- [45] M. E. T. Gerards, J. L. Hurink, and P. K. F. Hölzenspies. A survey of offline algorithms for energy minimization under deadline constraints. *Journal of Scheduling*, 19(1):3–19, 2016. ISSN 1099-1425. doi: 10.1007/s10951-015-0463-8. URL <http://dx.doi.org/10.1007/s10951-015-0463-8>.
- [46] GNU. ggplot2. <http://ggplot2.org/>, . Last accessed: 2014-10-22.
- [47] GNU. Gnu ‘make’. <http://www.gnu.org/software/make/manual/-make.html>, . Last accessed: 2012-08-06.
- [48] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proc. 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 151–162. ACM, 2006.
- [49] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

- [50] H. Grahn and P. Stenström. Evaluation of a competitive-update cache coherence protocol with migratory data detection. *Journal of Parallel and Distributed Computing*, 39(2):168 – 180, 1996. ISSN 0743-7315. doi: <http://dx.doi.org/10.1006/jpdc.1996.0164>. URL <http://www.sciencedirect.com/science/article/pii/S0743731596901641>.
- [51] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 250–256, 2000. doi: 10.1109/DATE.2000.840047.
- [52] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *the 38th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO’05)*, 2005.
- [53] A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *Proc. SPAA’10*, pages 11–20. ACM, 2010.
- [54] I. Gurobi Optimization. Gurobi optimizer reference manual, 2015. URL <http://www.gurobi.com>.
- [55] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele. Efficient execution of Kahn Process Networks on multi-processor systems using protothreads and windowed fifos. In *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, pages 35–44, Oct 2009.
- [56] E. Hansson, E. Alnervik, C. Kessler, and M. Forsell. A quantitative comparison of PRAM based emulated shared memory architectures to current multicore CPUs and GPUs. In *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pages 1–7, Feb 2014.
- [57] D. Helms, E. Schmidt, and W. Nebel. Leakage in CMOS Circuits — An Introduction. In *Proc. PATMOS 2004*, LNCS 3254, pages 17–35. Springer, 2004.
- [58] H. Hollerith. *An Electric Tabulating System*. Columbia College, 1889. URL <http://books.google.se/books?id=jXGwPAAACAAJ>.
- [59] U. Höning and W. Schiffmann. A comprehensive test bench for the evaluation of scheduling heuristics. In *Proc. of PDCS ’04*, 2004.
- [60] A. H. Hormati, Y. Choi, M. Kuldur, R. Rabbah, T. Mudge, and S. Mahlke. Flexstream: Adaptive compilation of streaming applications for heterogeneous architectures, 2009.

- [61] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart. A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling. *IEEE J. of Solid-State Circuits*, 46(1):173–183, Jan. 2011.
- [62] W. Huang and Y. Wang. An optimal speed control scheme supported by media servers for low-power multimedia applications. *Multimedia Systems*, 15(2):113–124, 2009. ISSN 1432-1882. doi: 10.1007/s00530-009-0153-5. URL <http://dx.doi.org/10.1007/s00530-009-0153-5>.
- [63] R. Hultén, J. Keller, and C. Kessler. Optimized on-chip-pipelined mergesort on the Cell/B.E. In *Proceedings of Euro-Par 2010*, volume 6272, pages 187–198, 2010.
- [64] Intel. Pentium processor family developer’s manual, 1997. URL <http://download.intel.com/design/intarch/manuals/24142805.pdf>.
- [65] Intel. An introduction to the intel quickpath interconnect, January 2009. URL <http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html>.
- [66] Intel. The SCC platform overview, 2010. URL <http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-platform-overview-paper.pdf>. last access: March 10th 2015.
- [67] Intel. SCC external architecture specification (EAS), 2010. URL https://communities.intel.com/servlet/JiveServlet/previewBody/5852-102-1-9012/SCC_EAS.pdf. Last access: March 10th, 2015.
- [68] Intel. The SCC programmer’s guide, 2012. URL <https://communities.intel.com/servlet/JiveServlet/previewBody/5684-102-8-22523/SCCProgrammersGuide.pdf>. last access: March 10th, 2015.
- [69] intel, 2012. URL <https://www.threadingbuildingblocks.org/>.
- [70] Intel. Intel Xeon Phi coprocessor system software developers guide, June 2013. URL <https://software.intel.com/sites/default/files/managed/b5/83/intel-xeon-phi-systemssoftwaredevelopersguide.pdf>.
- [71] Intel. *Combined Volume Set of Intel 64 and IA-32 Architectures Software Developer’s Manuals*, January 2015. URL <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.

- [72] Intel. Silvermont-based intel products, Last access: 2016-11-15. URL <http://ark.intel.com/products/codename/55844/Bay-Trail#@All>.
- [73] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.
- [74] J. Janzén. Evaluation of energy-optimizing scheduling algorithms for streaming computations on massively parallel multicore architectures. Master's thesis, Linköping University, 2014. URL <http://liu.diva-portal.org/smash/record.jsf?pid=diva2%3A756758>.
- [75] R. Jordans, F. Siyoum, S. Stuijk, A. Kumar, and H. Corporaal. An Automated Flow to Map Throughput Constrained Applications to a MPSoC. In P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASICS)*, pages 47–58, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-28-6. doi: <http://dx.doi.org/10.4230/OASICS.PPES.2011.47>. URL <http://drops.dagstuhl.de/opus/volltexte/2011/3081>.
- [76] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005. ISSN 0018-8646. URL <http://dl.acm.org/citation.cfm?id=1148882.1148891>.
- [77] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
- [78] Kalray. MPPA manycore, 02 2014. URL http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf. last accessed: 15-12-2014.
- [79] H. Kasahara. Standard task graph set, 2004. URL <http://www.kasahara.elec.waseda.ac.jp/schedule/index.html>.
- [80] J. Keller, C. Kessler, and J. Träff. *Practical PRAM Programming*. Wiley-Interscience, 2001. ISBN 0-471-35351-5.
- [81] J. Keller, C. Kessler, and R. Hulten. Optimized on-chip-pipelining for memory-intensive computations on multi-core processors with explicit memory hierarchy. *Journal of Universal Computer Science*, 18(14): 1987–2023, 2012.
- [82] C. Kessler, P. Eitschberger, and J. Keller. Energy-Efficient Static Scheduling of Streaming Task Collections with Malleable Tasks. In *Proc. 25th PARS-Workshop*, number 30 in PARS-Mitteilungen, pages 37–46, 2013.

- [83] C. Kessler, N. Melot, P. Eitschberger, and J. Keller. Crown Scheduling: Energy-Efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Malleable Streaming Tasks. In *Proc. of 23rd Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2013)*, 2013.
- [84] C. Kessler, L. Li, U. Dastgeer, A. Gidenstam, A. Atalar, and A. Dobre. D1.2 initial specification of energy, platform and component modelling framework. Technical Report FP7-611183 D1.2, The Execution Models for Energy-Efficient Computing Systems (EXCESS) Consortium, 2014.
- [85] C. Kessler, L. Li, A. Atalar, and A. Dobre. XPDL: Extensible platform description language to support energy modeling and optimization. In *Proc. of Int. Work. on Embedded Multicore Systems (ICPP-EMS)*, 2015.
- [86] H. Khdr, S. Pagani, E. Sousa, L. Vahid, A. Pathania, F. Hannig, M. Shafique, J. Teich, and J. Henkel. Power density-aware resource management for heterogeneous tiled multicores. *IEEE Transactions on Computers*, PP(99):1–1, 2016. ISSN 0018-9340. doi: 10.1109/TC.2016.2595560.
- [87] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0.
- [88] Y.-K. Kwok and A. Ishfaq. Benchmarking the task graph scheduling algorithms. In *IPPS/SPDP*, pages 531–537, 1998.
- [89] E. G. Larsson and O. Gustafsson. The impact of dynamic voltage and frequency scaling on multicore dsp algorithm design [exploratory dsp]. *IEEE Signal Processing Magazine*, 28(3):127–144, May 2011. ISSN 1053-5888. doi: 10.1109/MSP.2011.940410.
- [90] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- [91] K. Li. Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1484–1497, 2008.
- [92] K. Li. Energy efficient scheduling of parallel tasks on multiprocessor computers. *J. of Supercomputing*, 60(2):223–247, 2012.
- [93] K. Li. Energy and time constrained task scheduling on multiprocessor computers with discrete speed levels. *Journal of Parallel and Distributed Computing*, 95:15 – 28, 2016. ISSN 0743-7315. doi:

- <http://dx.doi.org/10.1016/j.jpdc.2016.02.006>. Special Issue on Energy Efficient Multi-Core and Many-Core Systems, Part I.
- [94] L. Li and C. Kessler. MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. *Journal of Supercomputing*, pages 1–16, 2016. doi: 10.1007/s11227-016-1792-x.
 - [95] J. Liu and J. Guo. Voltage island aware energy efficient scheduling of real-time tasks on multi-core processors. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICSS), 2014 IEEE Intl Conf on*, 2014.
 - [96] J. Liu and J. Guo. Energy efficient scheduling of real-time tasks on multi-core processors with voltage islands. *Future Generation Computer Systems*, 56:202 – 210, 2016. ISSN 0167-739X. doi: <http://dx.doi.org/10.1016/j.future.2015.06.003>.
 - [97] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241 – 252, 2002. doi: [http://dx.doi.org/10.1016/S0377-2217\(02\)00123-6](http://dx.doi.org/10.1016/S0377-2217(02)00123-6).
 - [98] E. G. Martín, N. Lavesson, and H. Grahm. Energy efficiency in data stream mining. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015, ASONAM '15*, pages 1125–1132, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3854-7. doi: 10.1145/2808797.2808863. URL <http://doi.acm.org/10.1145/2808797.2808863>.
 - [99] M. M. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012.
 - [100] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: the programmer’s view. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010. doi: 10.1109/SC.2010.53.
 - [101] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby. Evaluation of CPU frequency transition latency. *Computer Science - Research and Development*, 29(3-4):187–195, August 2014. doi: 10.1007/s00450-013-0240-x.
 - [102] md-rezaur rahman. Intel xeon phi coprocessor vector microarchitecture, Last access: 2016-11-15. URL <https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture>.

- [103] N. Melot. Programming Environment Library (PELib). <http://www.ida.liu.se/labs/pelab/pelib/index.en.shtml>, 2016. URL <http://www.ida.liu.se/labs/pelab/pelib/index.en.shtml>.
- [104] N. Melot, K. Avdic, C. Kessler, and J. Keller. Investigation of main memory bandwidth on Intel Single-Chip Cloud Computer. Intel MARC3 Symposium 2011, Ettlingen, 2011. URL <http://liu.diva-portal.org/smash/record.jsf?pid=diva2%3A664296>.
- [105] N. Melot, C. Kessler, K. Avdic, P. Cichowski, and J. Keller. Engineering parallel sorting for the intel SCC. *Procedia Computer Science*, 9(0): 1890 – 1899, 2012. doi: <http://dx.doi.org/10.1016/j.procs.2012.04.207>. Proc. of the Int. Conf. Computational Science, ICCS 2012.
- [106] N. Melot, J. Janzen, and C. Kessler. Mimer and Schedeval: Tools for Comparing Static Schedulers for Streaming Applications on Manycore Architectures. In *Parallel Processing Workshops (ICPPW)*, IEEE, pages 146–155, Sept 2015. doi: 10.1109/ICPPW.2015.24.
- [107] N. Melot, C. Kessler, and J. Keller. Improving Energy-Efficiency of Static Schedules by Core Consolidation and Switching Off Unused Cores. In I. Press, editor, *Proc. of Int. Conf. on Parallel Computing (ParCO 2015)*, pages 285 – 294, Edinburgh, UK, September 2015. doi: 10.3233/978-1-61499-621-7-285.
- [108] N. Melot, C. Kessler, J. Keller, and P. Eitschberger. Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Manycore Systems. *ACM Trans. Archit. Code Optim.*, 11(4):62:1–62:24, Jan. 2015. ISSN 1544-3566. doi: 10.1145/2687653. URL <http://doi.acm.org/10.1145/2687653>.
- [109] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, EMSOFT '07, pages 57–66, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. doi: 10.1145/1289927.1289941. URL <http://doi.acm.org/10.1145/1289927.1289941>.
- [110] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52–58, Apr 2001. ISSN 0018-9162. doi: 10.1109/2.917539.
- [111] N. Nedjah, M. V. Carvalho da Silva, and L. de Macedo Mourelle. Customized computer-aided application mapping on NoC infrastructure using multiobjective optimization. *J. of Syst. Arch.*, 57(1):79–94, 2011.
- [112] A. Olofsson, T. Nordström, and Z. Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726, Nov 2014. doi: 10.1109/ACSSC.2014.7094761.

- [113] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. URL <http://www.openmp.org/mp-documents/openmp-4.5.pdf>.
- [114] A. Partow. C++ mathematical expression library. <http://www.par-tow.net/programming/exprtk/>, 2015. Last access: February 2015.
- [115] M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan. Scalable compile-time scheduler for multi-core architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09*, pages 1552–1555, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. ISBN 978-3-9810801-5-5. URL <http://dl.acm.org/citation.cfm?id=1874620.1874991>.
- [116] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed Scaling of Tasks with Precedence Constraints. *Theory of Computing Systems*, 43(1): 67–80, July 2008.
- [117] X. Qi and D.-K. Zhu. Energy efficient block-partitioned multi-core processors for parallel applications. *Journal of Computer Science and Technology*, 26(3):418–433, 2011. ISSN 1860-4749. doi: 10.1007/s11390-011-1144-5. URL <http://dx.doi.org/10.1007/s11390-011-1144-5>.
- [118] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL <http://www.R-project.org>.
- [119] V. Raghunathan, M. Srivastava, and R. Gupta. A survey of techniques for energy efficient on-chip communication. In *Design Automation Conference, 2003. Proceedings*, pages 900–905, June 2003. doi: 10.1109/DAC.2003.1219148.
- [120] A. D. Robison. A parallel stable sort using C++11 for TBB, Cilk Plus, and OpenMP, 2014. URL <https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp>.
- [121] R. Rosales, C. Herglotz, M. Glaß, A. Kaup, and J. Teich. *Analysis and Exploitation of CTU-Level Parallelism in the HEVC Mode Decision Process Using Actor-Based Modeling*, pages 263–276. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30695-7. doi: 10.1007/978-3-319-30695-7_20. URL http://dx.doi.org/10.1007/978-3-319-30695-7_20.
- [122] J. Rosen, P. Eles, Z. Peng, and A. Andrei. Predictable worst-case execution time analysis for multiprocessor systems-on-chip. In *2011 Sixth IEEE International Symposium on Electronic Design, Test and Application*, pages 99–104, Jan 2011. doi: 10.1109/DELTA.2011.27.

- [123] P. Sanders and J. Speck. Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks. In *Proc. of the 18th Int. Conference on Parallel Processing*, Euro-Par'12, pages 167–178, 2012.
- [124] M. Sandrieser, S. Benkner, and S. Pillana. Using explicit platform descriptions to support programming of heterogeneous many-core systems. *Parallel Computing*, 38(1–2):52 – 65, 2012. ISSN 0167-8191. doi: <http://dx.doi.org/10.1016/j.parco.2011.10.008>. Extensions for Next-Generation Parallel Programming Models.
- [125] S. Savas, E. Gebrewahid, Z. Ul-Abdin, T. Nordström, and M. Yang. An evaluation of code generation of dataflow languages on manycore architectures. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–9, Aug 2014. doi: 10.1109/RTCSA.2014.6910501.
- [126] S. Savas, S. Raase, E. Gebrewahid, Z. Ul-Abdin, and T. Nordström. Dataflow implementation of qr decomposition on a manycore. In *Proceedings of the Third ACM International Workshop on Many-core Embedded Systems*, MES '16, pages 26–30, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4262-9. doi: 10.1145/2934495.2934499. URL <http://doi.acm.org/10.1145/2934495.2934499>.
- [127] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 71–80, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1424-4. doi: 10.1145/2380403.2380422. URL <http://doi.acm.org/10.1145/2380403.2380422>.
- [128] T. Schwarzer, J. Falk, M. Glaß, J. Teich, C. Zebelein, and C. Haubelt. Throughput-optimizing compilation of dataflow applications for multi-cores using quasi-static scheduling. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 68–75, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3593-5. doi: 10.1145/2764967.2764972. URL <http://doi.acm.org/10.1145/2764967.2764972>.
- [129] L. Shih-Wei, D. Zhaohui, W. Gansha, and G.-Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, page 12, March 2006. doi: 10.1109/CGO.2006.13.
- [130] SiSoftware. Benchmarks : Intel mobile haswell (crystalwell): Memory sub-system, 2015. URL http://www.sisoftware.co.uk/?d=qa&f=mem_hsw. Last access: March 10th, 2015.

- [131] D. C. Snowdon, S. Ruocco, and G. Heiser. Power management and dynamic voltage scaling: Myths and facts. In *Proceedings of the 2005 Workshop on Power Aware Real-time Computing, New Jersey, USA*, 2005.
- [132] A. Sodani. Knights landing (knl): 2nd generation intel xeon phi processor, Aug .
- [133] E. Sousa, F. Hannig, J. Teich, Q. Chen, and U. Schlichtmann. Run-time adaptation of application execution under thermal and power constraints in massively parallel processor arrays. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES '15*, pages 121–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3593-5. doi: 10.1145/2764967.2771933. URL <http://doi.acm.org/10.1145/2764967.2771933>.
- [134] P. Stenström, M. Brorsson, F. Dahlgren, H. Grahm, and M. Dubois. Boosting the performance of shared memory multiprocessors. *Computer*, 30(7):63–70, July 1997. ISSN 0018-9162. doi: 10.1109/2.596630. URL <http://dx.doi.org/10.1109/2.596630>.
- [135] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7): 491–541, 1997. ISSN 0001-5903. doi: 10.1007/s002360050095.
- [136] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multi-processor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th Annual Design Automation Conference, DAC '07*, pages 777–782, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: 10.1145/1278480.1278674. URL <http://doi.acm.org/10.1145/1278480.1278674>.
- [137] R. Tamassia. *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*, chapter Graph Markup Language (GraphML), pages 517–543. Chapman & Hall/CRC, 2007.
- [138] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A Language for Streaming Applications. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer Berlin Heidelberg, 2002. doi: 10.1007/3-540-45937-5_14.
- [139] Tilera. Company homepage. www.tilera.com, accessed February 16, 2015.
- [140] I. Ukhov, M. Bao, P. Eles, and Z. Peng. Steady-state dynamic temperature analysis and reliability optimization for embedded multiprocessor systems. In *Proceedings of the 49th Annual Design Automation Conference, DAC '12*, pages 197–204, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228399. URL <http://doi.acm.org/10.1145/2228360.2228399>.

- [141] T. Willhalm, R. Dementiev, and P. Fay. Intel performance counter monitor - a better way to measure cpu utilization. <https://software.intel.com/en-us/articles/intel-performance-counter-monitor>, April 2016.
- [142] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995. ISSN 0163-5964. doi: 10.1145/225830.223990. URL <http://doi.acm.org/10.1145/225830.223990>.
- [143] H. Xu, F. Kong, and Q. Deng. Energy Minimizing for Parallel Real-Time Tasks Based on Level-Packing. In *18th Int. Conf. on Emb. and Real-Time Comput. Syst. and Appl. (RTCSA)*, pages 98–103, Aug 2012.
- [144] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mossé. Energy-Efficient Policies for Embedded Clusters. *SIGPLAN Not.*, 40(7):1–10, June 2005. ISSN 0362-1340. doi: 10.1145/1070891.1065912. URL <http://doi.acm.org/10.1145/1070891.1065912>.
- [145] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 374–382, Oct 1995. doi: 10.1109/SFCS.1995.492493.
- [146] yWorks. yEd. URL http://www.yworks.com/en/products_yed_about.html. Last access: February 16, 2015.
- [147] S. M. Zahedi and B. C. Lee. REF: Resource Elasticity Fairness with sharing incentives for multiprocessors. *SIGARCH Comput. Archit. News*, 42(1):145–160, Feb. 2014. ISSN 0163-5964. doi: 10.1145/2654822.2541962. URL <http://doi.acm.org/10.1145/2654822.2541962>.
- [148] Z. Zhou, W. Plishker, S. S. Bhattacharyya, K. Desnos, M. Pelcat, and J.-F. Nezan. Scheduling of parallelized synchronous dataflow actors for multicore signal processing. *Journal of Signal Processing Systems*, 83(3):309–328, 2016. ISSN 1939-8115. doi: 10.1007/s11265-014-0956-2. URL <http://dx.doi.org/10.1007/s11265-014-0956-2>.

Dissertations

Linköping Studies in Science and Technology

Linköping Studies in Arts and Science

Linköping Studies in Statistics

Linköping Studies in Information Science

Linköping Studies in Science and Technology

- No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.
- No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.
- No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.
- No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.
- No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.
- No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.
- No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.
- No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.
- No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.
- No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.
- No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.
- No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.
- No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.
- No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.
- No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.
- No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.
- No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.
- No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.
- No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.
- No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.
- No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.
- No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.
- No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.
- No 221 **Michael Reiffrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.
- No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.
- No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.
- No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.
- No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.
- No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.
- No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.
- No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.
- No 270 **Ralph Rönquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.
- No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.
- No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.
- No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.
- No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.
- No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.
- No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.
- No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.
- No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.
- No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.
- No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ ABB 1968-1993, 1995, ISBN 91-7871-494-X.

- No 375 **Ulf Söderman**: Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.
- No 383 **Andreas Kågedal**: Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.
- No 396 **George Fodor**: Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.
- No 413 **Mikael Pettersson**: Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.
- No 414 **Xinli Gu**: RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.
- No 416 **Hua Shu**: Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.
- No 429 **Jaime Villegas**: Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.
- No 431 **Peter Jonsson**: Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.
- No 437 **Johan Boye**: Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.
- No 439 **Cecilia Sjöberg**: Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.
- No 448 **Patrick Lambrix**: Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.
- No 452 **Kjell Orsborn**: On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.
- No 459 **Olof Johansson**: Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.
- No 461 **Lena Strömbäck**: User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.
- No 462 **Lars Degerstedt**: Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.
- No 475 **Fredrik Nilsson**: Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.
- No 480 **Mikael Lindvall**: An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.
- No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.
- No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.
- No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.
- No 498 **Thomas Drakengren**: Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.
- No 502 **Jakob Axelsson**: Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.
- No 503 **Johan Ringström**: Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.
- No 512 **Anna Moberg**: Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.
- No 520 **Mikael Ronström**: Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.
- No 522 **Niclas Ohlsson**: Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.
- No 526 **Joachim Karlsson**: A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.
- No 530 **Henrik Nilsson**: Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.
- No 555 **Jonas Hallberg**: Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.
- No 561 **Ling Lin**: Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.
- No 563 **Eva L Ragnemalm**: Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.
- No 567 **Jörgen Lindström**: Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.
- No 582 **Vanja Josifovski**: Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.
- No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.
- No 592 **Mikael Ericsson**: Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.
- No 593 **Lars Karlsson**: Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.
- No 594 **C. G. Mikael Johansson**: Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.
- No 595 **Jörgen Hansson**: Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.
- No 596 **Niklas Hallberg**: Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.
- No 597 **Vivian Vimarlund**: An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.
- No 598 **Johan Jenvald**: Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.
- No 607 **Magnus Merkel**: Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.
- No 611 **Silvia Coradeschi**: Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.
- No 613 **Man Lin**: Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.
- No 618 **Jimmy Tjäder**: Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.
- No 627 **Vadim Engelson**: Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

- No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.
- No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.
- No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.
- No 688 **Marcus Bjärelund:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.
- No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.
- No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.
- No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.
- No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.
- No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.
- No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.
- No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.
- No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.
- No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.
- No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.
- No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.
- No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.
- No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.
- No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.
- No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.
- No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.
- No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.
- No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.
- No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.
- No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems' Development, 2003, ISBN 91-7373-604-X.
- No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.
- No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.
- No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.
- No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.
- No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.
- No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.
- No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.
- No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.
- No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.
- No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.
- No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.
- No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.
- No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.
- No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5.
- No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.
- No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.
- No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.
- No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.
- No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.
- No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.
- No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.
- No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.
- No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

- No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.
- No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.
- No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.
- No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.
- No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.
- No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.
- No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.
- No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.
- No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.
- No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.
- No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.
- No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.
- No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.
- No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.
- No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.
- No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.
- No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.
- No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.
- No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.
- No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.
- No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.
- No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.
- No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.
- No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.
- No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.
- No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.
- No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.
- No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.
- No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.
- No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.
- No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.
- No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 91-85715-65-7.
- No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.
- No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.
- No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.
- No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.
- No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.
- No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.
- No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.
- No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.
- No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.
- No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.
- No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.
- No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.
- No 1143 **Mehdi Amirjoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.
- No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.
- No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

- No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.
- No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.
- No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.
- No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.
- No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.
- No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.
- No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.
- No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.
- No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.
- No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.
- No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.
- No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.
- No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.
- No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.
- No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.
- No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.
- No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.
- No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.
- No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.
- No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.
- No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.
- No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.
- No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.
- No 1337 **Alexander Siemers:** Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis, 2010, ISBN 978-91-7393-317-9.
- No 1354 **Mikael Asplund:** Disconnected Discoveries: Availability Studies in Partitioned Networks, 2010, ISBN 978-91-7393-278-3.
- No 1359 **Jana Rambusch:** Mind Games Extended: Understanding Gameplay as Situated Activity, 2010, ISBN 978-91-7393-252-3.
- No 1373 **Sonia Sangari:** Head Movement Correlates to Focus Assignment in Swedish, 2011, ISBN 978-91-7393-154-0.
- No 1374 **Jan-Erik Källhammer:** Using False Alarms when Developing Automotive Active Safety Systems, 2011, ISBN 978-91-7393-153-3.
- No 1375 **Mattias Eriksson:** Integrated Code Generation, 2011, ISBN 978-91-7393-147-2.
- No 1381 **Ola Leifler:** Affordances and Constraints of Intelligent Decision Support for Military Command and Control - Three Case Studies of Support Systems, 2011, ISBN 978-91-7393-133-5.
- No 1386 **Soheil Samii:** Quality-Driven Synthesis and Optimization of Embedded Control Systems, 2011, ISBN 978-91-7393-102-1.
- No 1419 **Erik Kuiper:** Geographic Routing in Intermittently-connected Mobile Ad Hoc Networks: Algorithms and Performance Models, 2012, ISBN 978-91-7519-981-8.
- No 1451 **Sara Stymne:** Text Harmonization Strategies for Phrase-Based Statistical Machine Translation, 2012, ISBN 978-91-7519-887-3.
- No 1455 **Alberto Montebelli:** Modeling the Role of Energy Management in Embodied Cognition, 2012, ISBN 978-91-7519-882-8.
- No 1465 **Mohammad Saifullah:** Biologically-Based Interactive Neural Network Models for Visual Attention and Object Recognition, 2012, ISBN 978-91-7519-838-5.
- No 1490 **Tomas Bengtsson:** Testing and Logic Optimization Techniques for Systems on Chip, 2012, ISBN 978-91-7519-742-5.
- No 1481 **David Byers:** Improving Software Security by Preventing Known Vulnerabilities, 2012, ISBN 978-91-7519-784-5.
- No 1496 **Tommy Färnqvist:** Exploiting Structure in CSP-related Problems, 2013, ISBN 978-91-7519-711-1.
- No 1503 **John Wilander:** Contributions to Specification, Implementation, and Execution of Secure Software, 2013, ISBN 978-91-7519-681-7.
- No 1506 **Magnus Ingmarsson:** Creating and Enabling the Useful Service Discovery Experience, 2013, ISBN 978-91-7519-662-6.
- No 1547 **Wladimir Schamai:** Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool, 2013, ISBN 978-91-7519-505-6.
- No 1551 **Henrik Svensson:** Simulations, 2013, ISBN 978-91-7519-491-2.
- No 1559 **Sergiu Rafiliiu:** Stability of Adaptive Distributed Real-Time Systems with Dynamic Resource Management, 2013, ISBN 978-91-7519-471-4.
- No 1581 **Usman Dastgeer:** Performance-aware Component Composition for GPU-based Systems, 2014, ISBN 978-91-7519-383-0.
- No 1602 **Cai Li:** Reinforcement Learning of Locomotion based on Central Pattern Generators, 2014, ISBN 978-91-7519-313-7.
- No 1652 **Roland Samlaus:** An Integrated Development Environment with Enhanced Domain-Specific

Interactive Model Validation, 2015, ISBN 978-91-7519-090-7.

- No 1663 **Hannes Uppman:** On Some Combinatorial Optimization Problems: Algorithms and Complexity, 2015, ISBN 978-91-7519-072-3.
- No 1664 **Martin Sjölund:** Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models, 2015, ISBN 978-91-7519-071-6.
- No 1666 **Kristian Stavåker:** Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures, 2015, ISBN 978-91-7519-068-6.
- No 1680 **Adrian Lifa:** Hardware/ Software Codesign of Embedded Systems with Reconfigurable and Heterogeneous Platforms, 2015, ISBN 978-91-7519-040-2.
- No 1685 **Bogdan Tanasa:** Timing Analysis of Distributed Embedded Systems with Stochastic Workload and Reliability Constraints, 2015, ISBN 978-91-7519-022-8.
- No 1691 **Håkan Warnquist:** Troubleshooting Trucks – Automated Planning and Diagnosis, 2015, ISBN 978-91-7685-993-3.
- No 1702 **Nima Aghaee:** Thermal Issues in Testing of Advanced Systems on Chip, 2015, ISBN 978-91-7685-949-0.
- No 1715 **Maria Vasilevskaya:** Security in Embedded Systems: A Model-Based Approach with Risk Metrics, 2015, ISBN 978-91-7685-917-9.
- No 1729 **Ke Jiang:** Security-Driven Design of Real-Time Embedded System, 2016, ISBN 978-91-7685-884-4.
- No 1733 **Victor Lagerkvist:** Strong Partial Clones and the Complexity of Constraint Satisfaction Problems: Limitations and Applications, 2016, ISBN 978-91-7685-856-1.
- No 1734 **Chandan Roy:** An Informed System Development Approach to Tropical Cyclone Track and Intensity Forecasting, 2016, ISBN 978-91-7685-854-7.
- No 1746 **Amir Aminifar:** Analysis, Design, and Optimization of Embedded Control Systems, 2016, ISBN 978-91-7685-826-4.
- No 1747 **Ekhioz Vergara:** Energy Modelling and Fairness for Efficient Mobile Communication, 2016, ISBN 978-91-7685-822-6.
- No 1748 **Dag Sonntag:** Chain Graphs – Interpretations, Expressiveness and Learning Algorithms, 2016, ISBN 978-91-7685-818-9.
- No 1768 **Anna Vapen:** Web Authentication using Third-Parties in Untrusted Environments, 2016, ISBN 978-91-7685-753-3.
- No 1778 **Magnus Jandinger:** On a Need to Know Basis: A Conceptual and Methodological Framework for Modelling and Analysis of Information Demand in an Enterprise Context, 2016, ISBN 978-91-7685-713-7.
- No 1798 **Rahul Hiran:** Collaborative Network Security: Targeting Wide-area Routing and Edge-network Attacks, 2016, ISBN 978-91-7685-662-8.
- No 1813 **Nicolas Melot:** Algorithms and Framework for Energy Efficient Parallel Stream Computing on Many-Core Architectures, 2016, ISBN 978-91-7685-623-9.

Linköping Studies in Arts and Science

- No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle

Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

- No 586 **Fabian Segelström:** Stakeholder Engagement for Service Design: How service designers identify and communicate insights, 2013, ISBN 978-91-7519-554-4.
- No 618 **Johan Blomkvist:** Representing Future Situations of Service: Prototyping in Service Design, 2014, ISBN 978-91-7519-343-4.
- No 620 **Marcus Mast:** Human-Robot Interaction for Semi-Autonomous Assistive Robots, 2014, ISBN 978-91-7519-319-9.
- No 677 **Peter Berggren:** Assessing Shared Strategic Understanding, 2016, ISBN 978-91-7685-786-1.
- No 695 **Mattias Forsblad:** Distributed cognition in home environments: The prospective memory and cognitive practices of older adults, 2016, ISBN 978-91-7685-686-4.

Linköping Studies in Statistics

- No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.
- No 10 **Karl Wahlén:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.
- No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.
- No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlations in Daily and Sub-daily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

Linköping Studies in Information Science

- No 1 **Karin Axelsson:** Metodisk systemstrukturerings- att skapa samstämmighet mellan informationssystem- arkitektur och verksamhet, 1998. ISBN-9172-19-296-8.
- No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.
- No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.
- No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.
- No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.
- No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.
- No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.
- No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN 91-7373-736-4.
- No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.
- No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

- No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.
- No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.
- No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.
- No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponent-baserade informationssystem, 2006, ISBN 91-85643-22-X.

INSTITUTE OF TECHNOLOGY

Linköping Studies in Science and Technology, Dissertation No. 1813, 2016
Department of Computer and Information Science (IDA)

Linköping University
SE-581 83 Linköping, Sweden

www.liu.se