On Efficient Data Exchange in Multicore Architectures

Author(s):
Tretter, Andreas

Publication Date:
2018

Permanent Link:

https://doi.org/10.3929/ethz-b-000309314 →


On Efficient
Andreas Tretter
Data Exchange
in Multicore Architectures
Diss. ETH No. 25425

DISS. ETH NO. 25425

On Efficient Data Exchange in Multicore Architectures

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZÜRICH
(Dr. sc. ETH Zürich)

presented by
ANDREAS TRETTER
Diplom-Ingenieur, RWTH Aachen
Ingénieur des Arts et Manufactures, Ecole Centrale Paris

born on 6. 5. 1985
citizen of Germany

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jeronimo Castrillon, co-examiner

2018

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 175

Andreas Tretter

On Efficient Data Exchange in Multicore Architectures

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences
DISS. ETH NO. 25425
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jeronimo Castrillon, co-examiner
Examination date: 17 September 2018

Abstract

In contemporary multicore architectures, three trends can be observed: (i) A growing number of cores, (ii) shared memory as the primary means of communication and data exchange and (iii) high diversity between platform architectures. Still, these platforms are typically programmed manually on a core-by-core basis; the most helpful tool that is widely accepted are library implementations of frequently used algorithms. This complicated task of multicore programming will grow further in complexity with the increasing numbers of cores. In addition, the constant change in architecture designs and thus in platform-specific programming demands will continue to make it laborious to migrate existing code to new platforms.

State-of-the-art methods of automatic multicore code generation only partially meet the requirements of modern multicore platforms. They typically have a high overhead for different threads when growing numbers of cores and thus shrinking thread granularities demand the opposite. Also, they typically use message passing models for implementing data exchange when memory sharing should be the natural mode of data exchange. As a result, they often fail to produce efficient code, especially when large data throughput is required.

This thesis proposes a data-oriented approach to multicore programming. It shows how dividing a program into discrete tasks with clearly specified inputs and outputs helps to formalise the problem of optimising high data throughput applications for a large range of multicore architectures, at the same time enabling an efficient, low-overhead implementation. In detail, its contributions are as follows.

- Inefficiencies in existing programming models are demonstrated for the cases of the CAL actor language and Kahn process networks. Methods are shown to reduce these inefficiencies.

- Ladybirds, a specification model and language for parallel programs is presented. A Ladybirds program consists of a tasks with clearly defined inputs and outputs and of dependencies between them. It is explained how Ladybirds aims at execution efficiency also in the domains of data placement and transport and what steps are necessary to get from a Ladybirds specification to executable program code. The examples of comfortable debugging and of minimising state retention overhead for transient systems underline the usability and versatility of Ladybirds.

- An optimisation method for Ladybirds programs on the Kalray MPPA platform is presented. It tries to place data on different memory banks such as to avoid access conflicts. Afterwards, the Ladybirds optimisation problem for

the general case of arbitrary target platforms is formalised. Different aspects of it are discussed in greater detail and requirements for particular target platforms are examined.

- Also, a better understanding of contemporary hardware is sought. For that purpose, different probabilistic descriptions and models for interleaved on-chip memory are proposed and evaluated.

Zusammenfassung

In aktuellen Mehrkernarchitekturen lassen sich drei Trends beobachten: (i) Eine wachsende Anzahl an Kernen, (ii) gemeinsam genutzter Speicher als das Hauptinstrument der Kommunikation und des Datenaustauschs und (iii) eine hohe Diversität zwischen den Plattformarchitekturen. Dennoch werden diese Plattformen typischerweise manuell und Kern für Kern programmiert; das nützlichste breit akzeptierte Werkzeug sind Bibliotheksimplementierungen häufig verwendeter Algorithmen. Diese schwierige Aufgabe der Mehrkernprogrammierung wird mit den wachsenden Zahlen der Kerne weiter an Komplexität gewinnen. Darüber hinaus wird der ständige Wandel in Architektur-Designs und damit in plattformspezifischen Programmieranforderungen weiterhin großen Aufwand beim Migrieren bestehenden Codes auf neue Plattformen verursachen.

Derzeit genutzte Verfahren zur automatischen Mehrkern-Codegeneration entsprechen nur teilweise den Anforderungen moderner Mehrkernplattformen. Typischerweise weisen sie einen hohen Overhead für Threads auf und implementieren Datenaustausch mit Nachrichtenübergabemodellen, wohingegen eine wachsende Anzahl von Kernen, somit kleinere Thread-Granularitäten, geringeren Overhead erfordern und die gemeinsame Nutzung von Speichern die natürliche Art des Datenaustauschs sein sollte. Infolgedessen schaffen es diese Verfahren oft nicht, effizienten Code zu erzeugen, insbesondere, wenn ein hoher Datendurchsatz erforderlich ist.

Diese Arbeit schlägt einen datenorientierten Ansatz für die Mehrkernprogrammierung vor. Sie zeigt, wie die Unterteilung eines Programms in getrennte Tasks mit klar spezifizierter Ein- und Ausgabe dabei hilft, das Problem der Optimierung von Anwendungen mit hohem Datendurchsatz für eine große Auswahl an Mehrkernarchitekturen zu formalisieren und gleichzeitig eine effiziente Implementierung mit geringem Overhead zu erreichen. Im Einzelnen sind liefert sie folgende Erkenntnisse:

- Unzulänglichkeiten der Effizienz bestehender Programmiermodelle werden in den konkreten Fällen der CAL Actor Language und der Kahn-Prozessnetzwerke dargelegt. Methoden zur Verringerung dieser Unzulänglichkeiten werden aufgezeigt.

- Ladybirds, Spezifikationsmodell und -sprache für parallele Programme, wird vorgestellt. Ein Ladybirds-Programm besteht aus Tasks mit klar definierten Ein- und Ausgaben und Abhängigkeiten zwischen diesen. Es wird erläutert, wie Ladybirds eine hohe Ausführungseffizienz auch bei Datenplatzierung und -transport anstrebt und welche Schritte notwendig sind, um von einer

Ladybirds-Spezifikation zu ausführbarem Programmcode zu gelangen. Als Beispiele unterstreichen komfortables Debugging und die Minimierung des Zustandssicherungsaufwands für transiente Systeme die Nutzbarkeit und die Vielseitigkeit von Ladybirds.

- Eine Optimierungsmethode für Ladybirds-Programme auf der Kalray-MPPA-Plattform wird vorgestellt, die versucht, Daten auf verschiedenen Speicherbänken zu platzieren, um Zugriffskonflikte zu vermeiden. Anschließend wird das Ladybirds-Optimierungsproblem für den allgemeinen Fall beliebiger Zielplattformen formalisiert. Verschiedene Gesichtspunkte davon werden detaillierter besprochen und Anforderungen für bestimmte Zielplattformen werden untersucht.
- Weiterhin wird ein besseres Verständnis der heutigen Hardware angestrebt. Hierfür werden verschiedene probabilistische Beschreibungen und Modelle für verschränkten On-Chip-Speicher vorgeschlagen und ausgewertet.

Acknowledgements

This thesis would not have become what it is without the help, support and advice of many people. More importantly, I would not have become who I am today (and I would certainly not have been able to write this thesis) if I hadn't learned so many things from so many people whom I had the good fortune to meet and to work with during the course of my doctoral studies. I feel deeply grateful to each and everyone of them, even if there most probably is someone I forgot to mention below.

First of all, I would like to thank Lothar Thiele for giving me the opportunity to work on this thesis and for sparking my interest in the topic of data placement and transport. His ever helpful advice substantially brought my research forward and his unceasing fight for good presentation of science in spoken and written greatly contributed to the understandability of this document. Knowing that good research needs time, he never asked for paper mass production but gave me the possibility to understand relations and mechanisms in detail before publishing on them, thereby enabling research results that would not have been possible otherwise. Likewise, I want to thank Jeronimo Castrillon for reviewing this thesis and for serving on my committee board. It was his research that initially got me interested in the challenges of programming multicore systems.

My thanks also go to all my colleagues, together with whom I had a great time at and after work, getting to know a multitude of views to the world, scientifically and in general. In particular, I would like to mention Lars Schor, Devendra Rai and Pratyush Kumar, who were a big help for me in the beginning of my thesis, and Matthias Meyer and Stefan Drašković, who were the greatest office mates I could have wished for.

Over the entire course of my studies, I had the privilege to carry out research together with many different, excellent colleagues, namely Lars Schor, Pratyush Kumar, Jani Boutellier, Georgia Giannopoulou, Andres Gomez and Pascal Alexander Hager. Not only did they contribute to the results presented in this thesis, I also got familiar with their approaches, methodologies and ways of asking questions, learning something new from each of them. A number of student theses contributed to the results in this work — I would like to thank Harshavardhan Pandit, James Guthrie, Matthias Baer and Praveenth Sanmugarajah. There were, however, more student theses for which I was an adviser, namely those of Tobias Scherer, Felix Wermelinger, Dominik Böhi, Andreas Kurth, Bartlomiej Grzeskowiak, Marc Urech, Michael Walter, Alexander Sage, Marc Beusch and Joel Büsser. Even if they did not directly contribute to this work, their results often helped me in gaining a broader overview on my research domain and I can gratefully say I learned something from each student for whom I served as an adviser.

A recurring problem in the code optimisation domain is that of finding good benchmarks for experiments. I would like to thank all those people who by publishing their code eased this task for me and allowed better comparability between different publications. In particular, I my thanks go to the developers of the DOL/DAL, ORCC, MiBench, StreamIt and CConvNet projects.

I am deeply grateful to Beat Futterknecht, who was always there to help me with any administrative, logistical and practical questions and problems I had. He made my life much easier and my studies much smoother than they could have been otherwise. My most personal thanks go to my family, who made every effort to help and support me, always and unconditionally.

The work presented in this thesis was supported in part by the UltrasoundToGo RTD project (no. 20NA21 145911), evaluated by the Swiss NSF and funded by NanoTera.ch with Swiss Confederation financing. This support is gratefully acknowledged.

Zürich, September 2018

Contents

1

Introduction

The ubiquity of multi- and manycore platforms amongst today's electronic systems is contrasted by a lack of generally accepted high-level methods of programming these devices. While code parallelisation, i.e., the distribution of computational workload amongst the individual cores, is often seen as the major obstacle, it is by itself only a subproblem of a bigger challenge.

A less studied, but likewise important part of this challenge is the problem of data storage and transport. A typical calculation needs input data and produces results, which, in turn, may be needed for further calculations. Distributing the calculation to multiple cores now raises the question of how input data need to be distributed between the cores, and how the individual results can be merged afterwards. This may, or may not, require duplication of data, e.g. to prevent mutual input alterations, data races, access conflicts and alike. With higher amounts of data to be processed, these effects can have a vital impact on the system performance.

At the same time, hardware architectures are numerous and highly diversified with respect to memory and data transport. Shared on-chip memory and networks on chip are two trends that have emerged in the last years; even combinations of both are not uncommon. Each of these platform types needs a different programming approach. Careful data placement, well-managed DMA operations and the right amount of data duplication are only some of the software characteristics that may be a prerequisite to good performance on some platforms — and irrelevant on others.

This thesis is devoted to the question of how data storage and transport can be optimised and how this optimisation can be unified and automated for a large range of multicore platforms, where the latter term shall henceforth also include manycore platforms. A novel specification model for parallel programs is proposed that aims at optimisations for a large class of multi-core architectures. Examples for such op-

timisation automations are given, and requirements as well as pitfalls for efficient data storage and transport in concrete program implementations are discussed.

In the following, the diversity in multi-core architectures and programming requirements is demonstrated using the example of several contemporary platforms. A set of challenges is derived that must be met for achieving good performance. After separating the problem into four domains that must be addressed, a short outline of aim and contributions of this thesis follows.

1.1 Contemporary Multi-Core Platforms

As popularity and utilisation of multicore systems spread over the last years, a number of such platforms and architectures came into existence. These architectures differ in target audience and use cases; as a result, they have different features, design choices and programming models. Four different architecture approaches shall be presented here together with the demands they impose on programmers.

Intel Xeon Phi

Xeon Phi is a high-performance architecture by Intel, which exists in different models and versions. Particularly relevant are the generations "knights corner" [Chr14] and "knights landing" [Sod+16].

All "knights corner" platforms contain around 60 processor cores clocked between 1053 and 1238 MHz. Special vector units in the cores allow for exploiting data parallelism, each core has a 32 KB L1 data cache and a 512 KB L2 cache. A bidirectional ring bus connects the cores to each other and to the four memory controllers on the chip. The memory has a capacity of between 6 and 16 GB and uses pseudo-random interleaving [Rau91]. Communication between the cores is memory mapped, however, a sophisticated cache synchronisation protocol between the cores allows for efficient data exchange without going through the memory.

The "knights landing" generation brought several improvements to the architecture, such as more cores with higher computation power, a mesh interconnect instead of the ring bus and higher memory capacity (16 GB plus up to 384 GB additional, slower bulk memory).

While the programmer can significantly influence the performance of his program by promoting the use of vector units and other instruction-related optimisations, memory-related performance issues are automatically handled by the hardware. This frees the programmer from the burden of organising efficient data storage and exchange, but also marginalises his influence in this respect. "Knights landing" allows the programmer to choose between different hardware modes of either using

the main memory as a cache for the bulk memory or a flat address model in which data placement is manual. In the latter case, all frequently accessed data must be placed in the main memory.

P2012 and PULP

P2012 [Ben+12] and PULP [Con+15] are two different platform architectures based on similar design principles. Both consist of *clusters* with multiple processor cores and a fast on-chip cluster memory also called *L1 memory* or tightly-coupled data memory. Data caches are not used, but each core has a private instruction cache. The L1 memory consists of multiple sequentially interleaved banks that form one memory module. The cores are linked to the memory banks via a "logarithmic interconnect" consisting of tree structures of multiplexers and arbiters. All clusters on the platform share a common *L2 memory*, which is slower to access. Program code is stored on the L2 memory. External memory modules of larger capacities can be connected and would be referred to as *L3 memory*. DMA controllers effect data transfer between the different memories in the background.

The P2012 platform, also known as "STHORM", consists of 3 clusters with 16 cores and 32 memory banks each. The local L1 memory can be accessed within one cycle and has a capacity of 256 KB per cluster. The L2 memory capacity is 1 MB. The PULP architecture is used in different platforms; one multicore example is HERO [Kur+17] with up to 8 clusters of 8 cores and 16 memory banks each. The L1 memory of each cluster as well as the L2 memory have a capacity of up to 256 KB.

These platforms are examples for hierarchical memory architectures — different levels of memory from small scratchpad memories near to the cores to large memories far away from the cores allow to combine short-latency access to frequently accessed data with a large overall memory capacity. This principle is also the base of cache hierarchies; the difference is that while the latter fetch and write back data automatically, the programmer needs to take care of this with scratchpad memories. On the other hand, memories consume less power than caches and a good data transfer and placement strategy allows for higher and more predictable performance with scratchpad memories. Using memory hierarchies in P2012 and PULP is part of a strategy of achieving high computational performance on embedded systems, i.e., at a low power consumption.

For these systems to be efficient, it is important to place frequently accessed data on the L1 memory; DMA operations must ensure that at any moment in time, the data required for the ongoing operations is available there. In order to avoid processing elements waiting for their input data to arrive or their output buffers to be freed, DMA operations and computations must take place concurrently; efficiently scheduling these operations is an important task for the programmer.

Adapteva Epiphany

The Epiphany architecture [Olo+11] was designed by Adapteva with a clear focus on energy efficiency. It only contains the most essential components and all components are reduced to the most essential functionality, the idea being that the die area saved by these measures can be used instead for placing a higher number of processing elements. Epiphany consists of multiple identical tiles connected through a mesh network on chip (NoC). Each tile contains a processor core, a DMA controller, 4 non-interleaved memory banks of 8 KB each (for code and data) and a NoC router. Each router can send up to 8 bytes per direction (north, south, east, west) per cycle; these are routed following a fixed mechanism and they advance one tile per cycle. Communication between the cores is purely memory based; each core can access all memory banks on the chip. Memory writes are executed asynchronously, i.e., a core needs one cycle to place a write command on the NoC (if there is no congestion) and then immediately resumes operation. Memory reads can be performed within one cycle on the local memory banks (those on the same tile), reads on other tiles, however, are blocking and need to travel two-way through the NoC. To ensure swift operation of the NoC, memory access requests coming from the NoC always have priority over local access requests.

Because of its concept of identical tiles connected through a one hop mesh, the Epiphany architecture scales well and versions of different sizes exist. Epiphany-III chips feature 16 cores and are commercially available; they are also used on a credit-card sized "miniature computer" called Parallella. Epiphany-IV chips with 64 cores have been produced as well, are however not freely available on the market. Epiphany-V exists as a 1024 core IP (intellectual property) core.

For the programmer, data placement is even more complex with Epiphany than with the P2012/PULP architectures, since each core has its own memory. While P2012/PULP allow easy data sharing between cores in the same cluster, Epiphany requires the programmer to choose on which tile to place each chunk of data, and, possibly, whether it is useful to create multiple copies of it. Within one tile, the programmer must also choose which bank to use for which data chunk such as to minimise the number of memory access conflicts. While the considerations on background DMA transfers on P2012/PULP also apply for Epiphany, the latter also promotes the use of asynchronous write operations for data exchange. A core can directly write results it produces to the local memory of another core, which then processes them further. This is an important technique on Epiphany.

Kalray MPPA

The "massively parallel processor array" [Din+14] produced by Kalray is a platform conceived for embedded systems, however with real-time requirements in mind. It was built such that it behaves deterministically and predictably — experimental

results obtained on it in the course of this thesis (see Section 5.1) varied only marginally, if at all, between different runs. This thesis will concentrate on the first version of it ("Andey"), but later versions ("Bostan", "Bostan2" and "Coolidge") do not differ from it in the fundamental principles.

The MPPA consists of 4 input/output clusters and 16 compute clusters, each of which has 16 cores plus one "resource manager" (a dedicated core for system and resource management operations). 2 MB of memory distributed over 16 banks is available on each cluster and stores code and data. The programmer can choose whether to use memory interleaving or not. Each core has an instruction and a data cache. While the instruction caches are always enabled, the programmer can turn the data caches on or off. Cores can only access the local cluster memory. The clusters are connected via a NoC, which can be described as a torus mesh with configurable routing. Communication between the clusters is implemented via message passing, but DMA controllers also exist on each cluster. The input/output clusters can be used to access, for instance, external memory.

The programmer has to tackle data placement questions and DMA transfer scheduling tasks similar to the P2012/PULP architectures, but the choice between interleaved and non-interleaved memory yields more degrees of freedom. In particular, the problem of manually placing data on different memory banks considering that the latter are accessed by multiple cores can be intricate. At the same time, the relatively large on-chip memory eases the task of fitting all necessary data in it.

1.2 Challenges in Optimising Data Storage and Transfer

The last section presented a wide variety of contemporary multicore platform architectures. It contained power-hungry high-performance manycore architectures like Xeon Phi as well as highly energy-efficient low-power 16-core chips like Epiphany-III. Even if manifold differences in design principles and ideas became apparent, the comparison also revealed interesting commonalities.

- The number of cores is growing, not only with new architectures but also with the different generations of the same architecture families. Scalability is an important design factor that is often met by connecting identical tiles or clusters through networks on chip.
- Memory sharing is the prevalent concept of data exchange. Forming clusters of processing elements that share memory is a frequent pattern as well as non-uniform memory access models. Only the Xeon Phi architecture is even more extreme by featuring just one large coherent memory space; in this case, however, sophisticated caches are there to organise efficient data exchange.

In any case, none of the hardware architectures has a memory module that is private to a core.

- DMA controllers are standard on all platforms. The only exception is Xeon Phi, which has only one large memory space and which uses automatic cache synchronisation to achieve a similar goal.
- Direct communication between cores, for instance by signals or message passing, as it was done on Intel SCC, does not play an important role. Except for the Kalray MPPA, which allows sending messages between clusters, and certain low-level hardware supports for interrupt signals between cores in a cluster, memory based data exchange is the only supported form of communication between cores on all platforms. Message passing can of course be implemented in software, but no hardware support is provided.

Still, the differences must be taken into account as well:

- Large versus small memory capacities.
- Caches versus no caches. The latter case requires more memory optimisation code within the program.
- Uniform versus non-uniform memory access designs. This is not a binary question, but there are multiple degrees — completely uniform, uniform within clusters, completely non-uniform for each core.
- Memory transparency, e.g. interleaved versus non-interleaved. Interleaved memory requires less optimisation effort for placing data.

From these findings, the following requirements for a program optimiser can be derived. Firstly, the produced code must efficiently parallelise also for large numbers of cores, i.e., the overhead for executing independent threads and for exchanging data between them must be low. In particular, the philosophy of sharing data by sharing memory must be embraced and implemented together with DMA transfers in the background as an important means of efficient data exchange. Secondly, an optimisation methodology is needed that can be adapted to different target platforms with their individual requirements. Sophisticated algorithms for data placement in case of non-uniform memory access or non-interleaved memory are required, but must be replaceable with simpler optimisations in case of hardware-based memory management. In general, the optimiser must "understand" its target platform, i.e., it needs an accurate model of it. Thirdly, memory being a scarce resource on many embedded systems, optimisers for such architectures must be able to be as economical as possible with it. Since access times to different memory modules can easily differ by orders of magnitude, this is not only a question of being able or not to execute

an application on a platform, but it is a vital performance question. An optimiser should know what data is needed where at what time, and it should then make sure that the data will be available on a low-latency memory module when needed.

These are the challenges that need to be met for successful program optimisation and efficient program implementation. For this purpose, analyses and optimisations are required in various domains from high-level optimisation models to low level bit tweaks. These domains are tightly connected to different models and model types: The next section will try to establish a taxonomy of them.

1.3 Models Involved in Program Creation

A software implementation process, if it comprises compilation or design automation, can be divided into three steps: Program specification by the programmer, optimisation combined with code creation by one or multiple compilation tools and finally execution on the target platform. During these steps, the program is described using different representations, which, in turn, can be characterised by different models. In this work, these models shall be categorised as follows (see also Figure 1.1 on the following page).

- A *specification model* describes the semantics of the program specification, defining the basic elements and patterns that the programmer can combine to specify an application. It also defines the rules the program must conform to, and thereby the assumptions that can be safely made during the optimisation process and even during program execution.

- *Optimisation models* are abstractions of the program that are used during the optimisation process. They contain all information that is necessary for the corresponding optimisations to be carried out, typically in a format that is favourable for these optimisations. As there can be multiple optimisation phases, multiple optimisation models can be used during the optimisation process.

- An *implementation model* describes the semantics of the code to be executed on the platform. For that purpose, as a complement to the optimised program, it also provides basic primitives for program execution and a strategy for orchestrating them to a correct and possibly efficient execution. This implies that the optimisation process and maybe even the models it relies on are adapted to the implementation model.

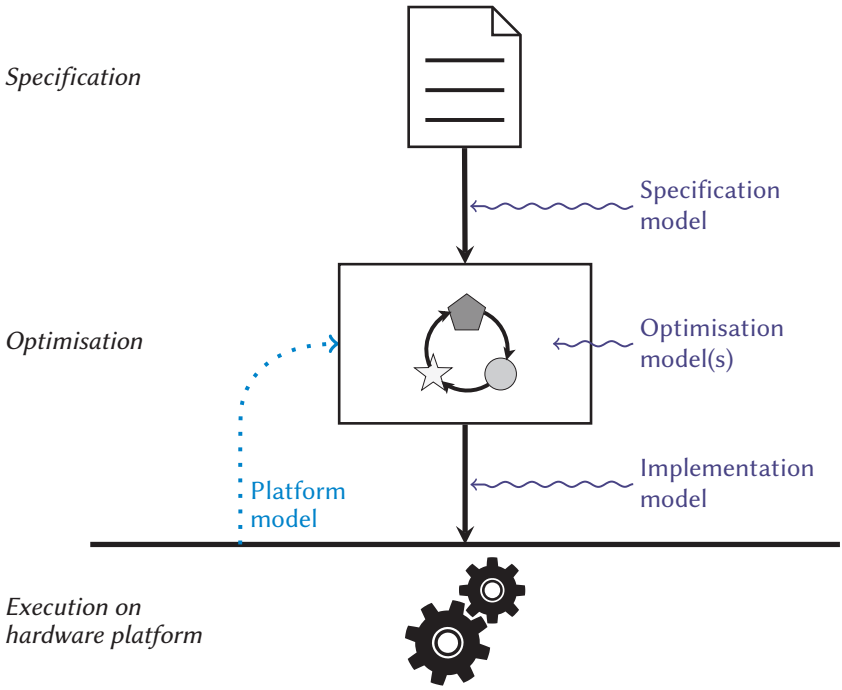


Figure 1.1: Models required in the compilation process of an application. Specification, optimisation and implementation model all describe representations of the given program, while a platform model predicts the behaviour of the hardware.

Since for optimisation, not only the program itself is interesting but also certain execution parameters, another class of models is required:

- A *platform model* is used to predict the behaviour of the target hardware platform, for instance in terms of execution time of certain program parts, memory availability, caching strategies etc. It is important to note that optimisation is in fact performed against this model rather than the actual hardware.

The combined set of all these models for a particular programming scenario shall be denoted here as a *model set*. The following examples give different such scenarios and illustrate the model sets for them.

Single threaded C program: The specification model is given by the source language C. The optimisation model is the intermediate representation of the compiler, typically together with control and data flow graphs, dominance tables and other information. The implementation model is given by the target processor, for instance as sequential execution of the generated assembly instructions. The platform model comprises the target instruction set, the execution times of the individual instructions, and maybe even information on the caches and their behaviour.

C program with POSIX threads: In addition to what was said about the single-threaded C program, the specification model here includes the POSIX threads API. The optimisation and description models are the same as for the single-threaded case (some compilers do additional checks, but no inter-thread optimisation). The implementation model is now augmented by the POSIX multithreading mechanisms, which are typically implemented by an operating system.

Process networks shall be considered here only at the network level rather than through the individual processes. Here, the specification is the same as the optimisation model – a directed graph of processes. The platform model would typically provide coarse-grained information about the platform (number and types of cores, connectivity between them, available memory) and about process affinities (e.g. certain runtimes on different processor types). There exist different implementation models (cf. Chapter 2); one possibility is to combine an existing multithreading mechanism like POSIX threads with an implementation of reading and writing functions for the channels.

From the description of the model types, it is clear that communication efficiency and high data throughput can only be attained if the all models in a model set are well-suited for this task, individually and in their interplay.

1.4 Aim and Contributions of This Thesis

The research conducted in this work leads to and justifies the following hypothesis:

> *A model set based on the concept of separate tasks with clearly defined inputs and outputs allows to reason about automatic program parallelisation with efficient use of memory and communication resources on any multicore platform.*

The following chapters will detail the findings and show the steps that lead there.

Chapter 2 gives an insight into **implementation models**, their crucial role for execution efficiency and how they connect and relate to specification models. First, a novel transformation technique from CAL actors to KPN processes is presented, making the case that switching the implementation model (to KPN) can yield not only performance gains. Secondly, the KPN implementation model in turn is shown to be inefficient on shared memory systems and a new extension to it is presented such as to overcome this issue in a simple way, which is, however, proven to maintain correct and determinate execution. For both cases, the implications on the corresponding specification models are discussed and conclusions are drawn on required and desired properties of possible specification models.

Chapter 3 presents Ladybirds, the **specification model** proposed in this work and derives the related optimisation and implementation models. It is shown how these models apply the lessons from the previous chapter, why they promise efficiency in data exchange and why they achieve the goals from Section 1.2 better than existing models. Examples underline the usefulness of the models, for muli-core systems as well as in completely unrelated areas.

Chapter 4 discusses the importance of **platform models**, with a special focus on interleaved memories. While all other components in current multicore platforms have been widely analysed or can easily be modelled, there exists no description model for interleaved memories that has been tested for suitability. This chapter introduces different possibilities and analyes their accuracy.

Chapter 5 finally comes to the subject of **optimisation models and techniques** for Ladybirds. First, an optimisation algorithm is introduced for assigning memory blocks to the different banks in a multi-bank shared-memory system with non-interleaved memory configuration. Such optimisations existed only for instruction level parallelism on VLIW processors. Afterwards, the generic case of optimising Ladybirds programs for arbitrary target platforms is discussed. The problem is formally described and important aspects of it are regarded in greater detail. Concrete target platforms are considered to establish a relation to the formalised optimisation problem.

2

Implementation models

Out of all models in a model set, the implementation model is certainly the one with the most direct influence on execution performance: Programs with an inefficient implementation model can hardly show good performance, and improvements to the implementation model typically are reflected one to one during execution. At the same time, the potential of an implementation model cannot be exploited unless the optimisation model(s) and sometimes even the specification model from the model set properly fit to it. This chapter will shed light on both of these aspects.

The CAL actor language will be used as an example of how switching from one implementation model to another can achieve a significant increase in performance. Such a switch, however, cannot be carried out without accompanying optimisation steps, which will be explained and discussed also with respect to their limitations.

Afterwards, the focus will be on inefficiencies of Kahn process networks and their implementation models when executed on shared memory platforms. An extension of these models will be shown which remedies these inefficiencies while fully maintaining the philosophy and the advantageous characteristics of Kahn process networks. As it will turn out, however, these improvements come at the cost of complicating the specification model.

A final discussion will wrap up the findings of this chapter, analyse the tradeoffs and derive desirable properties for implementation and corresponding specification models.

At first, however, an overview shall be given on the paradigm of process networks, in particular Kahn process networks. Being the subject of several studies, comparisons and considerations throughout this whole thesis, these networks are worth a closer look and a careful definition.

Algorithm 2.1: Specification of a Kahn process

```
process ECHO (in channels: X; out channels: Y)
    while TRUE do
        x ← READ (X)
        WRITE (Y, x)
    end
end
```

2.1 Process Networks

In this thesis, the term *process networks* shall be used to integrate multiple different programming concepts, which are similar in their topology but differ in many details, mostly terminology and semantic details of components. Some of these concepts have developed independently, for instance Kahn process networks (KAHN 1974 [Kah74]) and synchronous dataflow (KARP and MILLER 1966 [KM66], LEE and MESSERSCHMITT 1987 [LM87]). This explains notable differences not only in naming but also in the approaches, given especially the different backgrounds of the originators of these models.

The following definitions can be seen as an attempt to obtain a unified and consistent representation of the different concepts and terminologies throughout this thesis.

Definition 2.1. A **process network** is a directed multigraph, the nodes of which represent computational elements. Its edges are called *channels* and represent unbounded FIFO buffers. Computational elements can write *tokens* to outgoing channels by enqueueing them at the back of the FIFO and read tokens from incoming channels by dequeueing them from the front of the FIFO. No other communication is allowed between the computational elements.

In different models, these computational elements have different names. In Kahn process networks, the computational elements are called *processes*, whereas in the dataflow domain, they are called *actors*. This differentiation is maintained here because of a fundamental semantic difference between processes and actors (see below).

The different types of process networks shall be explained in the following. The approach taken here will be to first discuss the concept of Kahn process networks in detail and then to present the other models in where they differ from it.

2.1.1 Kahn Process Networks

The idea of *Kahn process networks* (KPNs) was introduced by Gilles Kahn [Kah74]. It has become popular in the parallel programming domain, but is not always defined in the same way. This section will give a KPN definition for this thesis, summarise the most important properties of these networks and discuss how they can be implemented.

2.1.1.1 The KPN specification model

As Kahn points out, a KPN can be thought of as a process network with multi-band Turing machines as computational elements. The first band of each machine is its private working band, and for each channel in the network, a one-way band is added on which one machine writes and another reads. This notion of KPNs shall be adopted here, albeit in a different, but equivalent definition, which is nearer to programming practice.

Definition 2.2. A **Kahn process network** is a process network with *Kahn processes* as computational elements.
A **Kahn process** is a stateful, sequential program which in any arbitrary sequence performs calculations with predictable results, write accesses to the outgoing channels, and blocking, destructive read accesses to the incoming channels.

This is a very generic definition of processes, imposing only the restriction that each channel read access has to be *blocking* and *destructive*. *Blocking* means that when a process tries to read from an empty channel, it will wait (possibly for infinite time) until a token arrives on the channel. *Destructive* means that upon reading a token, the token is also removed from the channel.

Example 2.1. Algorithm 2.1 gives the specification of a process Echo, which has one input and one output channel. Its functionality is to repeat all tokens from the input channel one to one on the output channel.

2.1.1.2 Formal description and analysis of KPNs

The formal description of KPNs requires a number of terms and concepts, firstly that of channel, input ant output histories.

Definition 2.3. Let γ be a channel. For any execution of the process network, the **history** of γ is the (possibly infinite) sequence of all tokens that are written to the channel.

Correspondingly, the **input history** of a process is the combination of the histories of all its incoming channels and its **output history** that of its outgoing channels.

Furthermore, the following notations shall be used. Let $\mathscr{A}$ be an alphabet of tokens. Then $\mathscr{A}^*$ denotes the set of all finite and infinite sequences of arbitrary length over $\mathscr{A}$, including the empty sequence. For two sequences $X, Y \in \mathscr{A}^*$, the notation $X \sqsubseteq Y$ means that X is a prefix of Y, i.e. $|X| \leqslant |Y|$ and $X_i = Y_i, 1 \leqslant i \leqslant |X|$. In particular, the empty sequence is a prefix of any sequence.

A function $f : \mathscr{A}_1^* \longrightarrow \mathscr{A}_2^*$, with $\mathscr{A}_1, \mathscr{A}_2$ two alphabets, is called *monotonic* if $X \sqsubseteq Y \implies f(X) \sqsubseteq f(Y)$.

The definition of monotonicity for functions with multiple arguments and/or returned sequences is analogous: The prefix property is extended to tuples of sequences such that a tuple is the prefix of another if each of its sequences is the prefix of the corresponding sequence in the other tuple. A monotonic function then maintains the prefix relation of two tuples when applied to them.

With these prerequisites, a Kahn process can be formally described as a monotonic[1] function assigning its input history to its output history.

Example 2.2. Consider the process Echo from Example 2.1 on the preceding page. Assuming an alphabet $\mathscr{A}$, it can be described as

$$\textsc{Echo} : \mathscr{A}^* \longrightarrow \mathscr{A}^*, \quad \textsc{Echo}(X) = X\,.$$

This is the formal description which Kahn uses for describing and analysing Kahn processes. It is, however, not fully equivalent to the previous definition of Kahn processes as sequential programs. In fact, it includes a larger set of processes, as the following example shows.

Example 2.3. Consider the Echo process from before. Two Echo processes are now merged into a single process named Echo2.

Using the formal description, Echo2 could simply be declared as

$$\textsc{Echo2} : \mathscr{A}^* \times \mathscr{A}^* \longrightarrow \mathscr{A}^* \times \mathscr{A}^*, \quad \textsc{Echo2}(X_1, X_2) = (X_1, X_2)\,.$$

A possible attempt to specify Echo2 as a Kahn process according to Definition 2.2 might look like Algorithm 2.2. It is, however, not fruitful, as the following considerations show.

Assume the input sequences $X_1 = [1]$ and $X_2 = [1, 1, 1, \ldots]$ are provided to the Echo2 process. The formal description function would return X_1 and X_2 as expected. Now consider the actual specification. In the first iteration of the loop, from each channel one token would be copied. In the second iteration, however, X_1 is empty and the read operation on it would block infinitely. The returned sequences would therefore be [1] and [1], which is different from the input. Since Kahn processes are

[1] For the sake of simplicity, this definition is weaker than that of Kahn (monotonic vs. "continuous"). It is, however, sufficient for the considerations in this work.

Algorithm 2.2: Failed specification attempt for the ECHO2 process

```
process ECHO2 (in channels: X_1, X_2; out channels: Y_1, Y_2)
    while TRUE do
        x ← READ (X_1)
        WRITE (Y_1, x)
        x ← READ (X_2)
        WRITE (Y_2, x)
    end
end
```

not allowed to know which channel is empty and which is not, there is no way of avoiding this problem. Thus, ECHO2 cannot be specified as a Kahn process.

As it was mentioned earlier, there are different definitions of KPNs. For instance, [LP95] considers all monotonic functions to be Kahn processes, i.e., ECHO2 would be a Kahn process according to that definition. While this approach is suitable for formal analysis, the functional description of Kahn processes does not give any hints on how to specify or implement such processes in practice, given in particular that monotonic functions can express any amount of concurrency. Those aspects, however, play an important role in this work. Therefore, while it is acknowledged that [Kah74] leaves room for both interpretations of KPNs, this work adopts the notion of Kahn processes being sequential programs, as it was given in Definition 2.2.

2.1.1.3 Properties of KPNs

With the notions from above, the following characteristics of KPNs can be derived.

Determinacy: Kahn processes are *determinate* [Kah74], i.e., for any given input history, they will always produce the same output history. The property of determinacy holds for each KPN as a whole. This is what makes KPNs particularly interesting for parallel programming, since a relatively small set of requirements in the implementation (correct message passing, blocking reads) guarantees correct execution independently of timing conditions.

Composability: Inside a KPN, multiple processes and the channels between them can be merged into one process. This process can then can be described by a monotonic function [Kah74] and therefore holds all the properties of a Kahn process. However, in general it cannot be specified as a sequential program, as Example 2.3 showed.

Parallelism: Because of the multiple independent processes, concurrency is explicit in the model and therefore easily accessible to automation and optimisation tools.

Analysability: Kahn processes are Turing-complete sequential programs to which the halting problem applies. Yet, even the networks by themselves can become Turing complete already with very simple processes [Buc93]. Therefore, the possibilities for automatic analysis of KPNs with compilation tools are limited. In particular, it is impossible in general to decide at compile time whether the memory consumption of a KPN will remain bounded [Par95].

2.1.1.4 Implementation models for KPNs

Until now, the KPN specification model has been presented and the characteristics of KPNs have been discussed. In this work, however, KPNs are important mostly with respect to the implementation of parallel programs. Therefore, implementation models, methods and characteristics of those networks shall be discussed in this section. The focus will be on typical models for efficient execution on multi-core platforms; other approaches, which for instance support unfolding of recursive KPN definitions at runtime [KM76], shall be omitted here.

Overall, the implementation of KPNs is mostly straightforward. Since the processes are specified as sequential programs, they can simply be compiled by a conventional compiler. Read and write directives can be implemented by an application programmers' interface (API) that provides functions for these purposes. These functions need to implement communication: either in form of sending and receiving data through a network, network on chip etc. (message passing) or by storing and loading in a local memory. In the latter case, correct synchronisation between the processes must be ensured.

Once the processes are implemented, their execution must be scheduled. Simple options include having one process per processor core or relying on multithreading functionality provided by an operating system. It is also possible, however, to execute multiple or even all processes on one single processor in a single thread, for instance by executing one process until it blocks and then switching to the next.

Finally, the channels need to be implemented as well. Since these are of infinite size in the specification, care must be taken not to infringe on the semantics of the original program when dimensioning the (finite) buffers that implement the channels. As already discussed, it is not possible to predict at compile time how large the channels need to be for correct program execution (i.e. for retaining the specified program semantics). The following solution approaches address this issue, but none in a fully satisfying way.

- Dynamically adjusting the buffer sizes as needed [Par95; BH01]. This method is hard to implement on embedded systems. Moreover, since it is limited by the available resources, buffer dimensioning can fail at runtime.
- Dimensioning the buffers according to a pre-compilation instrumentation run [LC10]. This does not guarantee correct execution in all cases.
- Adding buffer sizes to the specification [Thi+07; Sch+12]. This obviously ensures that the KPN executes as specified, but offloads the non-trivial task of buffer dimensioning on to the programmer.

In summary, most of the tasks related to implementing KPNs are easy to accomplish, but some challenges need to be addressed in practice in the individual cases.

2.1.2 Dataflow Graphs

Apart from KPNs, a second concept has become popular for describing parallel programs: *Dataflow graphs* are a family of process network types consisting of *Synchronous Dataflow* (SDF) graphs [LM87] and generalisations thereof (e.g. cyclo-static dataflow [Bil+96] or boolean dataflow [Lee91]). SDF, in turn, is derived from the "model for parallel computations" by Karp and Miller [KM66].

SDFs are process networks with so-called *actors* as computational elements. An SDF actor, unlike a Kahn process, is not an infinitely running program. Instead, it repeatedly performs one atomic operation called *firing*, which consists of destructively reading fixed amounts of tokens from the input channels, processing them and writing a fixed amount of tokens on the output channels. An SDF actor also does not have a state, i.e., no information is stored between the firings. However, a "self-edge" (an edge having the same actor as source and target) can be used to model such a state. Due to the simple communication patterns of the actors, *initial tokens* that can be placed on the channels before execution of the network play a larger role in SDF graphs than in KPNs.

The SDF model of computation is often referred to as a specialisation of the KPN model. This is correct when only the specification models are considered. Each SDF actor can be expressed as a Kahn process (in an infinite loop, read the inputs, process them, write the outputs) and therefore each SDF graph can also be specified as a KPN. When it comes to implementation, however, implementation models exist for SDF which exploit the potential of SDF graphs better than the KPN implementation models. One reason is the SDF actors being stateless and the resulting intrinsic parallelism. This parallelism allows the simultaneous execution of multiple firings of the same actor. KPN implementation models do not account for such parallelism, since a Kahn process is a sequential program. Other reasons are that actor firings

can be scheduled statically and context switching overhead can be avoided due to the stateless nature of the actors.

Generalisations of SDF can be described — as [LP95] points out — as actors which can fire different *actions* (i.e. input/output token counts as well as input-to-output assignment may differ). *Firing rules* determine which action is actually fired. For instance, *cyclo-static dataflow* actors consist of a tuple of actions, which are fired repeatedly in the same order. Note that this introduces a (predictable) state to the actors which needs to be considered by the implementation model. *Boolean dataflow* actors have two actions; boolean tokens on one input channel determine which of them is fired. Boolean dataflow graphs are already Turing complete [Buc93]. All of these extensions can be expressed by the KPN specification model. Further generalisations with more generic firing rules are proposed in [LP95].

Another model of computation that is often seen in the tradition of dataflow graphs is given by the CAL actor language, which shall be the subject of the next section.

2.2 Executing CAL actors as KPN processes

After the theoretical considerations on different specification and implementation models from the last section, this section will show the impact and implications of using different implementation models in the concrete case of the CAL actor language as compared to Kahn process networks. As will be shown, changing implementation models can not only have a strong influence on performance, but also help to uncover flaws in program specifications. The necessary prerequisites for such a change will be discussed as well as possible disadvantages. The results presented here were obtained in collaboration with Jani Boutellier, James Guthrie and Lars Schor.

2.2.1 Introduction

The antagonism between expressivity and analysability has led to a number of different types of process networks, which provide different degrees of freedom to the programmer.

As opposed to KPNs, certain extended dataflow models allow non-determinacy and other features, however at the price of static analysability and thus a potentially less efficient implementation. An example for this kind of dataflow networks is the CAL actor language [EJ03], a subset of which named RVC-CAL [MAR10] has been

standardized by ISO/IEC 23001-4:2009 MPEG to specify multimedia applications. Since it represents a very generic kind of dataflow, we will take it as a reference for dataflow actors in this section.

Other than the dataflow models described in the last section, CAL allows, for instance, to fire different actions depending on the availability of data. Clearly, such features enable higher expressiveness and flexibility. On the other hand, if an actor's behaviour depends on data availability, it may also depend on timing. Consequently, an actor classification into either static, dynamic or time-dependent has been proposed [WR12; Zeb+08].

KPNs, being always determinate, bring the benefit of a more efficient low-level implementation. As a Kahn process is at any moment either computing or waiting for input from a specific channel, it is easy to determine whether the process is ready for execution. For a dataflow actor, multiple rules and criteria have to be evaluated first to achieve the same goal. This is one of the reasons why the KPN model has been widely used in high-level synthesis frameworks for parallel systems as, for instance, MAPS [LC10] or DAL [Sch+12]. Since even in non-determinate dataflow specifications, many actors are in fact determinate, it appears favourable to analyse these actors for KPN compatibility in order to exploit the related optimisation potential.

In this section, we present a formal method for translating KPN compatible dataflow actors to Kahn processes. To this end, we first show that the aforementioned classification into static, dynamic and time-dependent actors is inadequate for evaluating KPN compatibility. Afterwards, we propose an algorithm to classify a dataflow actor as KPN compatible or potentially KPN incompatible (it tries to identify sufficient conditions for KPN compatibility in a high number of real cases, the problem being undecidable in general). Based on this algorithm, we then propose a method to translate a KPN compatible dataflow actor to a Kahn process. Finally, we implement the proposed method in the RVC-CAL framework [Yvi+13] and show that more than 75 % of all actors of the RVC-CAL application suite [ORCAR] that have left the active development stage can be proven to be KPN compatible by our algorithm and that their execution time can be decreased by up to 1.97 x when executing them as Kahn processes instead of dataflow actors. In a manual classification effort, we analyse the KPN compatibility of all these actors and discover a high accuracy of the proposed method. We list the different patterns and situations leading to KPN incompatibility or non-recognition of KPN compatibility.

The remainder of the section is organized as follows. In Section 2.2.2, an overview on CAL is given. In Section 2.2.3, we describe the proposed translation technique. Experimental results are presented in Section 2.2.4. Finally, we review related work in Section 2.2.5.

Algorithm 2.3: Illustration of the behaviour of a dataflow actor α.

while TRUE **do**
  $A_\alpha^* \leftarrow \{a \in A_\alpha \mid a \text{ can be fired}\}$
  if $A_\alpha^* \neq \{\}$ **then**
    $a^* \leftarrow \arg\max_{a \in A_\alpha^*} q_\alpha(a)$
    fire a^*
  end
end

2.2.2 Background

In this section, the CAL programming model is introduced in detail. Afterwards, compatibility between CAL actors and Kahn processes is defined and the problem to be solved is formulated.

2.2.2.1 Dataflow Actors

An overview of different dataflow models has been given in Section 2.1.2. This section, as already mentioned, will concentrate on CAL, a very generic dataflow extension in which an actor may have a state and a set of different actions.

In order to properly define input and output of an action, we first introduce the notion of a *token set*. A token set contains all tokens that are read or written by an action; they are represented by their position immediately before or after firing, e.g. the second token to be read from a specific input channel or the fifth token to be written to a specific output channel during the firing. As FIFO channels only allow in-order accesses, all the token positions on a specific channel must form a sequence without any gaps in it. Formally, we define a token set as follows.

Definition 2.4. Let Γ be a set of channels and $\nu : \Gamma \longrightarrow \mathbb{N}_0$ a function assigning each channel $\gamma \in \Gamma$ a number of tokens to be read from or written to it. Then the **token set** ψ over Γ defined by ν is a set of token positions $\psi = \{(\gamma \in \Gamma, n \in \mathbb{N}) \mid n \leq \nu(\gamma)\}$. $\Psi(\Gamma)$ is the set of all token sets over Γ.

To be able to formally describe functions using these tokens as input or output, we introduce the notion of a *value space*:

Definition 2.5. The **value space** $V(\psi)$ of a token set ψ is the set of all possible combinations of values which the tokens represented in ψ can have.

Now, we formally define an actor as a stateful computational element with a prioritised set of actions it can fire.

Definition 2.6. An **actor** is a tuple $\alpha = (S_\alpha, A_\alpha, q_\alpha, s_\alpha^0)$, with S_α the set of possible states of the actor, A_α a set of actions for the actor, $q_\alpha : A_\alpha \longrightarrow \mathbb{N}$ a function assigning each action a *priority* and $s_\alpha^0 \in S_\alpha$ the initial state.

We consider the states of an actor to be an arbitrary combination of variables of any kind. This means in particular that the state set of an actor does not have to be finite.

We define an action as follows:

Definition 2.7. Let α be an actor with I_α the set of its incoming and O_α the set of its outgoing channels. An **action** for this actor is a tuple $a = (r_a, w_a, f_a, g_a)$ with $r_a \in \Psi(I_\alpha)$ and $w_a \in \Psi(O_\alpha)$ the *input* and *output token sets*, $f_a : S_\alpha \times V(r_a) \longrightarrow S_\alpha \times V(w_a)$ the *fire function* and $g_a : S_\alpha \times V(r_a) \longrightarrow \{\text{TRUE}, \text{FALSE}\}$ the *guard function* of the action.

The action *a can be fired* if:

1. all input tokens according to r_a are available and
2. g_a evaluates to true for the current state and the available input tokens.

Upon firing, it will destructively read the input tokens (i.e. all tokens in r_a) from the channels, evaluate f_a for the current state and the tokens just read and use its return values for updating the state of α and for writing tokens to the output channels according to w_a.

Now we can define the behaviour of an actor α as an infinite repetition of the following operations: It will determine the set $A_\alpha^* \subseteq A_\alpha$ of actions that can be fired. If this set is non-empty, the action $a \in A_\alpha^*$ with the highest priority $q_\alpha(a)$ will be fired. If there are multiple actions with the same, highest priority that can be fired, it is not defined which of those actions is fired. An illustration of this behaviour is given in Algorithm 2.3.

In summary, for a dataflow actor, the action which is fired (and thus the amount of tokens read and written) can depend on the state of the actor, on the value of tokens in the incoming channels and on the existence of tokens in the incoming channels. A dataflow actor must therefore be able to non-destructively read the tokens on its incoming channels (*peeking*). Furthermore, since there can be situations when the action to be fired (and thus possibly the output to be produced) depends on which token arrives first, actors can be non-determinate.

Listing 2.1: An absolute value actor written in CAL.

```
actor Abs() int In ==> int Out:  //one channel in, one out
  pos:  //action: reads a token i from In and writes it to Out
    action In:[i] ==> Out:[i]
    guard i >= 0  //only fired if i is non-negative
  end
  neg:  //action: reads a token i from In and writes -i to Out
    action In:[i] ==> Out:[-i]
    guard i < 0  //only fired if i is negative
  end
end
```

Listing 2.2: A non-deterministic merge actor written in CAL.

```
actor NDMerge()  //two channels in, one out
    int InA, int InB ==> int Out:
  actionA:  //reads i from InA and writes it to Out
    action InA:[i] ==> Out:[i]
  end  //no guard
  actionB:  //reads i from InB and writes it to Out
    action InB:[i] ==> Out:[i]
  end  //no guard
end
```

This dataflow model is implemented by the CAL Actor Language [EJ03] and its standardised variant RVC-CAL [MAR10] [2]. Listings 2.1 and 2.2 show two code examples written in RVC-CAL. The `Abs` actor in Listing 2.1 has two actions `pos` and `neg`. Both read one token from the input channel `In` and write one token to the output channel `Out`. The guard expressions $i \geq 0$ and $i < 0$, respectively, ensure that, depending on the value of the token at the FIFO head of `In`, only one of both actions can fire. This actor is determinate. For comparison, the two actions of the `NDMerge` actor in Listing 2.2 have no guards specified, i.e. their guard function always evaluates to true. Therefore, `actionA` and `actionB` can fire whenever a token is available at `InA` and `InB`, respectively, forwarding this token to `Out`.

[2] The model and (RVC-)CAL differ in that (RVC-)CAL does not require the specification of priorities. Also, the latter are specified as partial orders, i.e. one only defines e.g. $q(a_1) > q(a_2)$. The slightly simpler notation we chose does, however, cover all the cases required for this chapter, since multiple actions can still have the same priority number.

Listing 2.3: Example for a determinate, but not KPN compatible actor.

```
actor DeterminateNotKPN()
    int InA, int InB, int InC ==> int Out:
  actionA: action InA:[a], InB:[b] ==> Out:[a]
           guard a>0  && b>0  end
  actionB: action InB:[b], InC:[c] ==> Out:[b]
           guard b<=0 && c>0  end
  actionC: action InA:[a], InC:[c] ==> Out:[c]
           guard a<=0 && c<=0 end
  actionX: action InA:[a], InB:[b], InC:[c] ==>
           guard (a<=0 && b>0 && c>0) ||
                 (a>0 && b<=0 && c<=0) end
end
```

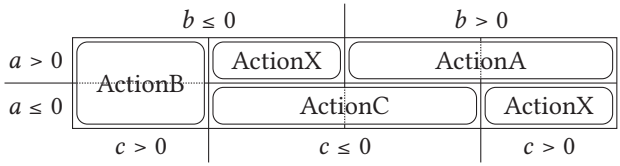


Figure 2.1: A Karnaugh map showing the actions to be fired for the actor from Listing 2.3, depending on the input token combination.

2.2.2.2 Problem Statement

In the following, we will define the problem to be solved in this section. To this end, we first define when an actor and a process can be regarded as equivalent.

Definition 2.8. Let α be a dataflow actor according to Definition 2.6 and π be a Kahn process as specified in Definition 2.2. α and π are **functionally equivalent** iff for any equal input history, α and π always produce equal output histories.

The problem regarded in this section can now be formulated as follows:

> *Given a dataflow actor α. Is there a functionally equivalent Kahn process π and, if so, how can it be constructed?*

The following examples shall illustrate the complexity of the problem. Of course, the actor in Listing 2.1 is KPN compatible while the actor in Listing 2.2 is non-

determinate and thus clearly not KPN compatible. However, things are more complicated for the actor shown in Listing 2.3 on the previous page. The Karnaugh map in Figure 2.1 on the preceding page shows the different actions to be fired for each combination (`a`, `b`, `c`) of the first tokens to be read from each of the input channels (provided that they exist). Since none of the guards overlap in the diagram, the action to be fired can be clearly determined from the *values* of the tokens and does *not* depend on *priorities* or the *availability* of tokens. The behaviour of the actor is thus determinate. Although some actions can still be fired if one of the channels is empty, this will only happen if the respective action would also be fired if the channel was filled.

This feature, however, cannot be achieved with KPN: A Kahn process would have to choose one channel to read from without knowing about the availability of tokens. If, for instance, this channel was `InA`, the process would block on an infinite sequence of negative integers on `InB` and of positive integers on `InC` if `InA` remained empty. This, however, is not the behaviour of the given actor. In other words, this actor is determinate but KPN incompatible[3]. A graphical interpretation of this example is that it is impossible to split the Karnaugh map at the borders for `a`, `b` or `c` without cutting through one of the action guards.

This example shows that the problem regarded in this work is not the same as the problem of classifying an actor as time-dependent or not, which was discussed, e.g., in [WR12].

2.2.3 Translating Dataflow Actors to Kahn Processes

In this section, we discuss the translation of dataflow actors to Kahn processes. While the two-step procedure we propose consists of a KPN compatibility evaluation and a subsequent Kahn process construction, our compatibility analysis method is constructive and thus works the other way round: We first construct a Kahn process *imitation* of the dataflow actor, i.e., a Kahn process that tries to produce the same output as the dataflow actor. Afterwards, if we can show that this process is functionally equivalent to the actor, we have proved the actor's KPN compatibility.

Obviously, this approach cannot detect all cases of KPN compatibility (that problem is undecidable [Zeb+08]). However, we will show in the next section, using a state-of-the-art dataflow benchmark suite with 381 actors, that the method works for a large subset of KPN compatible dataflow actors in real applications.

The section continues by showing how to build the mentioned Kahn process imitation of a given dataflow actor. For given input sequences, we establish criteria that

[3] Note the actor *could* be described by a monotonic assignment as discussed in Section 2.1.1.2. According to our Definition 2.2, however, this is not sufficient for KPN compatibility.

Algorithm 2.4: Template for a Kahn process translation π of a dataflow actor α. The template uses initial state $s_\pi^0 = s_\alpha^0$ and an action set $A_\pi = A_\alpha$.

```
s ← s_π^0
while TRUE do
    // Find next action to be fired
    a ← SelectNextAction(s, A_π)
    // Fire the action
    in = ReadInputTokens(r_a)
    (s, out) ← f_a(s, in)
    WriteOutputTokens(w_a, out)
end
```

guarantee functional equivalence in the concrete cases. Afterwards, we will show a formalisation of this approach in which we statically analyse if these requirements are met for all possible inputs, thereby establishing a sufficient condition for KPN compatibility. Finally, we will discuss different translation implementations and compare the efficiency of the produced code.

2.2.3.1 Constructing a Kahn Process from a Dataflow Actor

In the following, we propose a method to build a Kahn process imitating the functionality of a given actor. The difficulty is that in contrast to the dataflow actor, the constructed Kahn process can only access the input channels using blocking, destructive reads.

We propose to construct the process using the template shown in Algorithm 2.4. The behaviour of the process can be described as an endless loop, each iteration consisting of two operations: Finding the next action to fire and actually firing it. As firing a dataflow action can be done natively in a Kahn process, the only difficulty lies in finding the correct action to fire, i.e. in determining the function SelectNextAction. The following theorem shows that functional equivalence between such a Kahn process and a dataflow actor can be attained if for any input, the sequence of actions fired by the process is same as with the actor.

Theorem 2.1. *Let α be a dataflow actor and let π be a Kahn process constructed according to the template given in Algorithm 2.4. π is functionally equivalent to α if for any input history, π and α always fire the same actions in the same order.*

Proof. From Definition 2.8, we know that π and α are functionally equivalent if both generate the same output history for any given input history. According to the

process template, the operation of finding the next action neither alters the process state nor produces any output. Anything that can influence the output therefore has to happen when firing actions. Thus, π and α are functionally equivalent if they always fire the same actions on the same input sequence. ■

The challenge is now to be able at any moment to determine which action the actor will fire without knowing about the availability of tokens or their content. In general, this is not always possible for dataflow actors where, for instance, actions can be fired or not depending on the availability of tokens or their values. However, for a large group of actors, there are possibilities of exploiting certain properties of the action guards.

- In general, guard functions do not depend on the full input token set of the associated action. In particular, many guard functions only depend on the state of the actor and can thus be evaluated without reading any tokens.
- Actions may have common input tokens. The `Abs` actor in Listing 2.1 on page 22 shows a typical example of this situation: There are two different actions, both with guards that peek an input token. However, both of these guards peek the same input token, and one of these two actions must fire next. Consequently, the token will be read in any case and can thus be *prefetched*. After reading it, the process can decide which action to fire and pass the token on to it.
- Oftentimes, the return values of guards can be predicted without knowing all of the required input tokens. If, for instance, a guard function is a boolean AND combination of multiple terms, the result will always be FALSE if only one of these terms evaluates to FALSE. In this case, the input tokens for the other terms are not required to know that the guard is not met.

These ideas can now be combined to an algorithm for determining the action which is to be fired next, i.e. an implementation of SELECTNEXTACTION in Algorithm 2.4. For this, we assume that for each guard g, there is function predict$\langle g \rangle$(...), which, provided with the state of the actor and the values of the input tokens prefetched so far, evaluates to TRUE, FALSE or UNKNOWN. The following theorems shall provide a theoretical basis for the operation of the SELECTNEXTACTION algorithm.

First, we show that if the guard of a given action within a dataflow actor can be predicted to FALSE, this action will not be fired next, independently of any additional input tokens that may arrive.

Theorem 2.2. *Let α be an actor and $a \in A_\alpha$ be an action, with* predict$\langle g_a \rangle$ *evaluating to* FALSE. *Then a will not be the next action fired by α.*

Proof. If predict$\langle g_a \rangle$ (and thus g_a) evaluates to FALSE, a cannot be fired. The return value of g_a depends on the state of the actor and on certain input tokens. Both can only be changed when firing an action. So another action has to be fired first before g_a can evaluate to TRUE. ■

Now we will analyse which tokens the constructed Kahn process can prefetch at a given moment without losing functional equivalence to the original dataflow actor. The difficulty here is to avoid additional blocking which is not there in the original actor. Such a blocking could be induced by a (blocking) read operation in the Kahn process.

Theorem 2.3. *Let α be an actor to be translated to a Kahn process π. Let $A \subseteq A_\alpha$ be a set containing all actions the guards of which are predicted to* TRUE *or to* UNKNOWN. *Then π can prefetch all tokens from the* ***prefetch token set*** *of A, $\bigcap_{a \in A} r_a$, without losing the functional equivalence to α.*

Proof. According to Theorem 2.2, only elements of A are eligible to be fired next. Each of these actions $a \in A$ needs all tokens out of its input token set r_a before it can be fired. Therefore, α will not fire any action until those tokens which are contained in all of these input token sets have been fetched. A Kahn process prefetching any token from the prefetch token set will thus never be blocked for longer than α will. When in control again, it can continue imitating α. ■

With these prerequisites, we can now describe a possible implementation of SELECTNEXTACTION for the imitation of an actor α. The algorithm performs an iterative reduction of a set A of actions and can be summarised by the following steps:

0. Start with $A = A_\alpha$.
1. Prefetch all tokens from the prefetch token set of A.
2. If the guard of an action from A is predicted to FALSE considering all prefetched tokens, remove the action from A.
3. Iterate steps 1 and 2 until convergence. (Since A can only shrink, convergence is guaranteed.)

Once the iteration has converged, there are two possibilities: If all input tokens of the action in A with the highest priority have been prefetched, this action is to be fired next. Otherwise, the next action to be fired cannot be determined using this method.

Table 2.1: Example actions of an example dataflow actor with the state $s \in \mathbb{N}$ and the input channels X and Y. $X[0]$ is the first token the actor receives when reading from X, $X[1]$ the second token etc. The actions' priorities increase with increasing numbers.

Action	Input tokens	Guard	Priority
a_1	—	$s = 1$	1
a_2	$X[0], X[1], Y[0]$	$s = 2 \wedge X[1] > 0 \wedge Y[0] > 0$	2
a_3	$X[0]$	$s = 2 \wedge X[0] = 1$	3
a_4	$X[0], X[1], Y[0]$	$s = 2 \wedge X[0] = 2 \wedge X[1] \cdot Y[0] \leqslant 0$	4
a_5	—	$s = 3$	5
a_6	$Y[0]$	$s = 3 \wedge Y[0] < 0$	6

We will illustrate this algorithm for the example actor given in Table 2.1. The actor has a rather simple state $s \in \mathbb{N}$ and two input channels, X and Y. Each of the actions a_k, $k \in \{1, \ldots, 6\}$ has a different priority $q(a_k) = k$ (this is not necessarily the case in practice). Assuming that $s = 2$ and the input tokens on both X and Y are 1, 2, 3, 4, 5, ..., the algorithm would behave as follows:

- Initialisation: $A = \{a_1, \ldots, a_6\}$. No tokens to prefetch. Since $s = 2$, a_1, a_5 and a_6 can be eliminated (i.e. removed from A).
- First iteration: $A = \{a_2, a_3, a_4\}$. Prefetch $X[0]$. Since $X[0] = 1$, a_4 can be eliminated.
- Second iteration: $A = \{a_2, a_3\}$. No further tokens can be prefetched. No further eliminations are possible.

In this case, the algorithm stops with $A = \{a_2, a_3\}$. Since all input tokens of a_3 have been prefetched and its guard evaluates to TRUE, a_3 can be fired. a_2 is also still a candidate but cannot be fired yet because some of its input tokens are still missing. Since, however, a_3 has the higher priority and can be fired, the actor will fire a_3, independently of a_2. Thus, in this case the algorithm is able to determine a_3 as the next action to be fired. For other actor states or inputs, however, the situation may be different. Clearly, the constructed Kahn process is only functionally equivalent to the original actor if the next action to fire can be determined for *any* actor state and input. This is expressed in the following theorem.

Theorem 2.4. *Let α be an actor to be translated to a Kahn process π using the method described above. α and π are functionally equivalent if the proposed implementation of* SelectNextAction *is able to determine the next action to fire for any state of the actor and of the input channels.*

Proof. The correctness of the operations applied by the algorithm has been proven in Theorems 2.2 and 2.3. Therefore, the set A^*_α of actions that can be fired at a given moment is a subset of the set A of candidates obtained by the algorithm. If the action in A with the highest priority can be fired, it is (i) an element of A^*_α and (ii) the action with the highest priority in A^*_α, since $A^*_\alpha \subseteq A$. π will thus fire the same action as α.

If this method works for any state and input combination, π and α will always fire the same actions and are therefore functionally equivalent, which follows from Theorem 2.1. ■

2.2.3.2 Classification of Dataflow Actors

So far, we have seen a technique to construct a Kahn process from a dataflow actor. It tries to determine at runtime the next action to be fired. Only if this always succeeds, a correct translation was obtained and the actor can be shown to be KPN compatible. In the following, we present a static analysis method that determines if this holds true by systematically checking all possible outcomes of the SelectNextAction function introduced previously.

To this end, we construct a tree containing all possible operations that might be performed by SelectNextAction, i.e. reducing the set of firing candidates and prefetching tokens, depending on certain conditions that can be fulfilled or not. As this tree gives information about which tokens are fetched in which order, we call it the *peek sequence tree* (PST); a more formal definition is given later on.

For the example actor from Table 2.1, which was discussed above, the PST is given in Figure 2.2 on page 31. Every node in it represents a possible iteration of SelectNextAction, with the set A of firing candidates and the set P of tokens to be prefetched. The root node (marked as ***a***) represents the initialisation step. The outgoing edges of each node (i.e. those leading further away from the root node) represent the different possibilities of how SelectNextAction may proceed, leading to the next iteration step in the respective cases. They are annotated with a condition to be met such that the edge is followed. Since the prefetch token set of the root node is empty, the conditions leading away from it only contain the actor state s. The edge annotations further down will also have conditions concerning the prefetched tokens. Note that all these conditions are mutually exclusive for edges leaving the same node. However, they need not cover all possible cases, but only those which are covered by the actions of the original actor.

The leaves of the tree correspond to all possible outcomes of SelectNextAction:

- The iteration scenario discussed in Section 2.2.3.1 is represented by the path ***a*** – ***c*** – ***e***.
- Node ***b*** is a very straightforward case in which the action to fire is determined only by the state of the actor.
- Nodes ***h***, ***i*** and ***g*** represent cases in which, due to repeated token prefetching and firing candidate elimination, only one action to fire is left.
- Finally, in the case of node ***d***, the action to fire cannot be determined. This is because action a_6 has a higher priority than a_5, but also needs more input tokens. The actor would thus fire a_6 if these tokens are available and a_5 otherwise. The example actor regarded here is thus *not* KPN compatible.
- Another possible case, which does not occur in this example, is that of an ambiguous actor specification. An actor is specified ambiguously if it has two actions with the same priority, without mutually exclusive guards and if the input token set of the one action is a subset of that of the other action. In the PST, this would lead to a leaf with multiple actions of the same priority. One possible way of handling this issue would be to arbitrarily give priority to one of the actions. This is done in many backends as well as in [WR12]. In this work, however, since our final goal is translation to a KPN process, we choose a conservative approach and do not classify the actor as KPN compatible in order to prevent a translation when the actor semantics as intended by the programmer are not clear.

In the following, we will give the formal definition of a PST and we will show how to construct it. To this end, we first discuss how the predict$\langle\cdot\rangle$ function introduced earlier can be implemented. We do so by assuming that every guard is a boolean AND combination of multiple terms referred to as *constraints*. According to our following definition, a constraint requires a set of tokens (the *peek tokens*) in order to be evaluated and it can be met or not, according to a boolean function:

Definition 2.9. Let I_α be a set of input channels and S_α a set of possible states of an actor α. A **constraint** to an action for α is a tuple $c = (p_c, e_c)$ with $p_c \in \Psi(I_\alpha)$ a token set of *peek tokens* and $e_c : S_\alpha \times V(p_c) \longrightarrow \{\text{TRUE}, \text{FALSE}\}$ the *evaluation function*.

The negation of a constraint c is given as $\bar{c} := \big(p_c,\ \text{not}(e_c)\big)$. Likewise, the combination of two constraints c and d is given as $c \wedge d := (p_c \cup p_d,\ e_c \wedge e_d)$.

With this definition, a guard can be expressed as the boolean AND combination of all elements in a set of constraints. Therefore, we can assign each action such a set representing its guard. To be able to evaluate as many constraints as early as

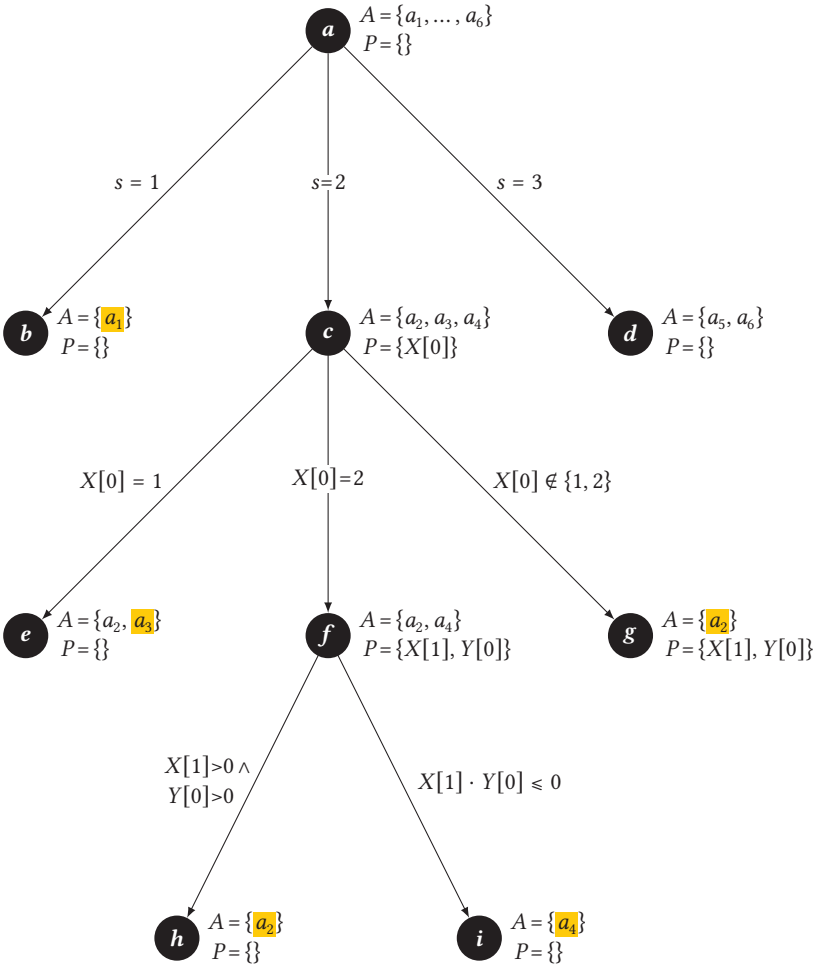


Figure 2.2: The peek sequence tree for the actor described in Table 2.1 on page 28. The action to be fired in each case is highlighted.

possible and thus maybe to eliminate certain actions as firing candidates early on, we would like to break down each guard into as many constraints with as small peek token sets as possible.

Definition 2.10. Let a be an action. The **constraint set** C_a of a is a set of constraints such that

$$\bigwedge_{c \in C_a} c \;=\; (p, g_a), \quad p \subseteq r_a$$

and that for each constraint $c \in C_a$, there are no two constraints c', $c'' \neq c$ such that $c' \wedge c'' = c$.

In the example actor described in Table 2.1 on page 28, the guard of action a_4 can be decomposed to the constraint set

$$C_{a_4} = \Big\{ \; \big(\{\},\ s = 2\big), \quad \big(\{X[0]\},\ X[0] = 2\big), \\ \big(\{X[0], X[1], Y[0]\},\ X[1] \cdot Y[0] \leqslant 0\big) \; \Big\}.$$

Our definition of a PST is now as follows:

Definition 2.11. A **peek sequence tree (PST)** for an actor α is a tree $T = (N, L)$ on which each node $n \in N$ is annotated with a set of actions $A(n) \subseteq A_\alpha$ and each edge $l \in L$ is annotated with a constraint $c(l)$.

We define the prefetch token set of a node according to Theorem 2.3 on page 27:

Definition 2.12. Let n be a node in a PST. Then its **prefetch token set** is

$$P(n) = \bigcap_{a \in A(n)} r_a \,.$$

A PST $T = (N, L)$ must fulfil the following conditions: For each edge $l \in L$ with $n \in N$ being its parent (source) node, it must hold that $p_{c(l)} \subseteq P(n)$. For any two edges $l, m \in L$ with a common parent node, it must hold that $c(l)$ and $c(m)$ cannot be satisfied at the same time, i.e. $e_{c(l) \wedge c(m)} \equiv \textsc{false}$.

The rest of this section describes the construction of a PST for an actor α. This procedure can be formalised as follows: A root node n_0 is created with $A(n_0) = A_\alpha$. For each action $a \in A(n_0)$, the set of evaluable constraints $C'_a = \{c \in C_a \mid p_c \subseteq P(n_0)\}$ is determined and combined to the *strictest evaluable constraint* $c'_a = \bigwedge_{c \in C'_a} c$, i.e. the largest top-level sub-expression of g_a that can already be evaluated with the tokens available through prefetching. (If C'_a is empty, we have $c'_a = (\{\}, \textsc{true})$.) For action a_4 from the example actor, the strictest evaluable constraint for the root node is $c'_{a_4} = (\{\}, s = 2)$.

From these $k := |A(n_0)|$ strictest evaluable constraints, each one can theoretically be met or not, which in total gives 2^k cases. These cases can be expressed as a set of constraint combinations

$$C_{\text{theo}}(n_0) := \left\{ \bigwedge_{a \in A(n_0)} x_a \;\middle|\; x_a \in \{c'_a, \overline{c'_a}\} \right\}.$$

For the example actor, the possible cases for the root node are

- $s = 1 \wedge s = 2 \wedge s = 2 \wedge s = 2 \wedge s = 2 \wedge s = 3 \wedge s = 3$,
- $\overline{s = 1} \wedge s = 2 \wedge s = 2 \wedge s = 2 \wedge s = 2 \wedge s = 3 \wedge s = 3$,
- $s = 1 \wedge \overline{s = 2} \wedge s = 2 \wedge s = 2 \wedge s = 2 \wedge s = 3 \wedge s = 3$,
- …

As constraints are often related and some contradict each other, not all of the combinations are satisfiable, as clearly seen in the example. Using a satisfiability modulo theories (SMT) solver, which is also provided with the set S_α of possible states of α, one can eliminate the unsatisfiable combinations. Also the case that none of the constraints is met can be eliminated, since this case is not covered either in the original actor. In the example, all of the $2^6 = 64$ possibilities are eliminated except for three:

- $s = 1 \wedge \overline{s = 2} \wedge \overline{s = 2} \wedge \overline{s = 2} \wedge \overline{s = 3} \wedge \overline{s = 3}$
- $\overline{s = 1} \wedge s = 2 \wedge s = 2 \wedge s = 2 \wedge \overline{s = 3} \wedge \overline{s = 3}$
- $\overline{s = 1} \wedge \overline{s = 2} \wedge \overline{s = 2} \wedge \overline{s = 2} \wedge s = 3 \wedge s = 3$

These combinations obviously simplify to $s = 1$, $s = 2$ and $s = 3$, which have been noted down in Figure 2.2 on page 31. In the real implementation, the number of satisfiability evaluations can be reduced by applying various optimisations that shall not be discussed here.

For each of the cases that have not been eliminated, a child node is inserted. The edge to it is annotated with the constraint combination corresponding to the case. The child node itself is annotated with the set of all actions for which the strictest evaluable constraint was assumed to be met in the constraint combination. See Figure 2.2 on page 31 for the example actor.

For all the child nodes, the same procedure is carried out recursively. However, only constraints that could not be evaluated before are regarded now. The old constraints are taken into account by combining all constraints of the edges that lead from the root node to the current node and by adding this combination as an additional constraint for the SMT solver. As soon as only one child node would be

Table 2.2: Satisfiability calculations for the PST in Figure 2.2 on page 31 at node ***c***. The constraint "TRUE" comes from action a_2, which also reads $X[0]$, but does not put a constraint on it.

Actions	Constraint combination	Eliminate?
{}	$s = 2 \wedge \overline{\text{TRUE}} \wedge \overline{X[0] = 1} \wedge \overline{X[0] = 2}$	Yes, empty set
$\{a_2\}$	$s = 2 \wedge \text{TRUE} \wedge \overline{X[0] = 1} \wedge \overline{X[0] = 2}$	**No, retain**
$\{a_3\}$	$s = 2 \wedge \overline{\text{TRUE}} \wedge X[0] = 1 \wedge \overline{X[0] = 2}$	Yes, unsatisfiable
$\{a_2, a_3\}$	$s = 2 \wedge \text{TRUE} \wedge X[0] = 1 \wedge \overline{X[0] = 2}$	**No, retain**
$\{a_4\}$	$s = 2 \wedge \overline{\text{TRUE}} \wedge \overline{X[0] = 1} \wedge X[0] = 2$	Yes, unsatisfiable
$\{a_2, a_4\}$	$s = 2 \wedge \text{TRUE} \wedge \overline{X[0] = 1} \wedge X[0] = 2$	**No, retain**
$\{a_3, a_4\}$	$s = 2 \wedge \overline{\text{TRUE}} \wedge X[0] = 1 \wedge X[0] = 2$	Yes, unsatisfiable
$\{a_2, a_3, a_4\}$	$s = 2 \wedge \text{TRUE} \wedge X[0] = 1 \wedge X[0] = 2$	Yes, unsatisfiable

inserted for a node, the recursion is stopped. Table 2.2 shows the procedure for node ***c*** in the PST for the example actor. The constraint "inherited" from above is $s = 2$; it is therefore only added at the beginning of each combination.

We can upper bound the complexity of the proposed PST construction algorithm:

Theorem 2.5. *For an actor with k actions, the maximum number of nodes in the PST is smaller than* $2^{1/2\,(k^2+k)}$.

Proof. If a child node has the same number of actions as its parents, no progress is made and the recursion is stopped. Therefore, a child has in the worst case one action less than its parent. In the worst case, the root node can have up to $2^k - 1$ child nodes. Each of these child nodes can then have up to $2^{k-1} - 1$ children, which again can have $2^{k-2} - 1$ children each and so forth. Multiplying these numbers, one obtains the maximum number of leaves in the tree. Also counting the non-leaf nodes, one has $1 + (2^k - 1) \cdot \left(1 + (2^{k-1} - 1) \cdot (1 + \ldots)\right) < (1 + 2^k - 1) \cdot (1 + 2^{k-1} - 1) \cdot \ldots = 2^k \cdot 2^{k-1} \cdots 2^0 = 2^{0+1+2+\ldots+k} = 2^{1/2\,(k^2+k)}$. ∎

Note that this is an upper bound for pathological cases. In our experimental evaluations with 381 real actors, the number of nodes stayed way below it in each case. Section 2.2.4 will show that even for large actors, the tree can be constructed in an acceptable time frame despite its theoretically exponential complexity. In extreme cases, one could stop PST construction prematurely without a classification result.

2.2.3.3 Constructing the translated Kahn process

With the results from the classification problem in mind, the solution to the translation problem is straightforward. If an actor has been classified as KPN compatible, one just needs to construct a process as described in Section 2.2.3.1.

One can simply implement the SelectNextAction function as shown there for determining the action to fire. This has the advantage that the complexity of this algorithm is polynomial with respect to the number of actions.

In practice, however, more lightweight code can be generated directly following the structure of the PST constructed during classification. For each node, prefetching code needs to be produced whereas each edge in the tree will be a branch in the code. Like this, dynamic predictions of guards can be replaced by simple, hard-coded if statements.

2.2.4 Experimental Results

In this section, we evaluate the performance of the proposed classification and translation algorithm using a state-of-the-art dataflow benchmark suite. The goal is to answer the following questions:

- What percentage of realistic RVC-CAL actors does the proposed algorithm classify as KPN compatible?
- What are the reasons for KPN compatible actors not being classified as such?
- Does the proposed translation of KPN compatible dataflow actors into Kahn processes indeed improve the performance of streaming applications?

2.2.4.1 Experimental Setup

The proposed classification and translation algorithm has been implemented as an extension to the Open RVC-Cal Compiler (ORCC) [Yvi+13] using the z3 SMT solver [DB08]. The corresponding ORCC benchmark suite [ORCAR] contains a total of 549 CAL actors. In order to provide meaningful data, current research projects (i.e. immature work under construction) as well as overly simplistic actors such as "hello world" examples have been left out from the evaluation. With the exception of those, the proposed classification algorithm has been tested on all available actors, 381 in total. In particular, the set of applications contains various video decoders (H.265 part2, H.264 PHiP, H.264 CBP and MPEG-4 SP, AVS), the JPEG and JPEG2000 image compression codecs, for telecommunications the ZigBee transmitter baseband description and a digital predistortion filter [Gha+14], a number of basic digital filters, a cryptographic library, a WAV audio player, a

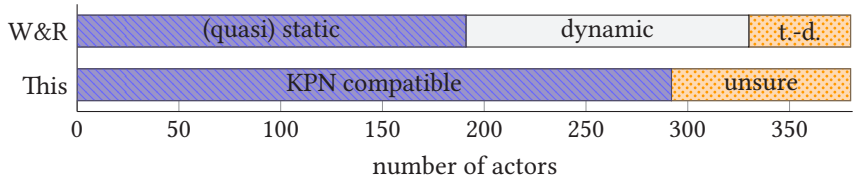


Figure 2.3: Overall comparison of the classification results of the algorithm of Wipliez and Raulet ("W&R") and that proposed in this section ("this"). The abbreviation "t.-d." stands for *time-dependent*; "unsure" designates potentially KPN incompatible actors.

GZIP decompressor and implementations of several CHSTONE benchmark suite applications.

The classification algorithm discussed above was run on an Intel Core i5-3210M processor, as a single threaded implementation. The classification of all 114 actors of the H.264 PHiP decoder, one of the most elaborate dataflow applications in the benchmark suite and with several highly complex actors, took about 65 seconds.

2.2.4.2 Comparison to Other Classifiers

In the following, we evaluate the performance of the proposed classification algorithm. To this end, we first regard the counts of the different classification results for the algorithm proposed in this work and for the algorithm by Wipliez and Raulet (W&R) [WR12]. These numbers are given in Figure 2.3.

For W&R, the group *(quasi) static* combines the three possible results SDF, CSDF and quasi-static [Bou+09], which are all KPN compatible by construction.

Actors are marked as *time-dependent* by the W&R classifier if it finds situations similar to that in node ***d*** in Figure 2.2 on page 31. As explained in Section 2.2.3.2, such actors are KPN incompatible. Note, however, that time-dependency is *not* the same as non-determinacy; it is only a necessary condition for the latter.

Finally, all other actors are classified as *dynamic*, i.e. determinate but not (quasi) static. These actors may or may not be KPN compatible.

The two classifiers regarded here cannot be compared directly for two reasons: Firstly, the different classification categories and secondly, their different treatment of ambiguous actor specifications as described in Section 2.2.3.2. The W&R classifier enforces (arbitrarily) a total priority ordering of all actions, which, in the extreme case, leads to an actor being classified either as time-dependent or as quasi static, depending on the order of the action specifications in the source code. The classifier

we propose always classifies actors with ambiguous specifications as potentially KPN incompatible.

Consequently, we have to look at the cases in closer detail:

- The results of the (quasi) static group of W&R can be confirmed by our algorithm in so far as it classifies all of the concerned actors as KPN compatible. The only exception is given by three ambiguously specified actors.

- The W&R classification of actors as time-dependent uses similar criteria to those in the algorithm we propose. Accordingly, none of the actors classified as time-dependent was classified as KPN compatible by the algorithm proposed in this work. However, manual classification showed that more than a quarter of these actors are KPN compatible, which was not recognised by the algorithms. The reasons of W&R time-dependent misclassifications and their frequency are similar to those for non-classifications in our algorithm, which will be discussed later on in detail.

- Out of the actors classified as dynamic by W&R, 75 % were classified as KPN compatible by the proposed algorithm. Another 7 % of these actors is KPN compatible as well but were not recognised as such. Note that these rates do not differ significantly from the overall KPN classification rate of 77 % with additional 6 % not recognised. In other words, for the regarded set of actors a W&R classification as dynamic does not provide information about the KPN compatibility of an actor.

In summary, the W&R classifier can – leaving aside the ambiguously specified actors – classify 185 out of 363 actors (or 51 %) with certainty as KPN compatible, whereas the algorithm proposed in this work can do the same with 292 actors (or 80 %). The number of recognised KPN compatible actors is thus 58 % higher.

2.2.4.3 Comparison to Manual Classification

In addition to the comparison with other classifiers, we also investigated on an absolute scale the classification quality of the algorithm proposed in this work. To this end, we undertook a manual classification effort of all the actors in the set.

Since the algorithm we propose guarantees KPN compatibility for all actors classified accordingly, we did not cross-check all of these actors manually, but we took samples at random and were able to validate the correctness of the algorithm and its implementation.

All actors which the algorithm did not classify as KPN compatible were checked manually with the help of its output. Figure 2.4 on the next page shows the results of

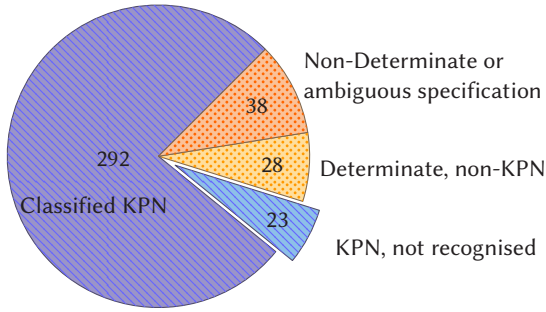


Figure 2.4:
Manual classification results

Listing 2.4: Simplified example for a KPN compatible actor not recognised as such.

```
bool ax:=false, bx:=false, cx:=false;
a: action => guard        !ax do ax := true; end
b: action => guard ax and !bx do bx := true; end
c: action => guard bx and !cx do cx := true; end
```

this manual classification effort. While 292 of the 381 actors were correctly classified as KPN compatible, there are 23 more KPN compatible actors, which, however, were not recognised as such. 66 actors are not KPN compatible for various reasons, which will be discussed in Section 2.2.4.4. In summary, our classifier in 94 % of the cases obtained the same result as an ideal classifier would. The optimisation possibilities for the other 23 cases will be discussed below. Note that we classified all actors *on the basis of the given implementation*, not on the basis of whether there could exist a KPN compatible implementation of their functionality.

During manual classification, we analysed why KPN compatible actors were not recognised as such. With the exception of one actor, which caused an error of the SMT solver, the reason was always the determination of the actors' state sets (S_α for an actor α). The implementation we used for the experiments is very simplistic: It assumes all combinations of all possible values of the actor's state variables to be the set of states the actor can have instead of analysing the actions to find out which values and which combinations thereof can actually be attained.

Listing 2.4 shows a simplified, but realistic example, in which there are three actions a, b and c, each of which during execution sets a state variable to true to indicate it has been fired. Their guards ensure no action is fired twice and each action is only fired after the one above it. Thus, actions a, b and c are fired in exactly that order. The classifier will, however, notice that the guards of actions a and c

are not mutually exclusive since it cannot establish a link between the three state variables. It will thus conclude that these two actions can be fired at the same time and it will assume an ambiguous specification. Intelligent state analysis, however, would yield that the combination `ax=false and bx=true` is not contained in the state set of the actor. Using this information, the guards of actions a and c could be recognised as mutually exclusive and the actor as KPN compatible.

2.2.4.4 Further results of manual classification

The manual classification also provided results concerning the nature of the KPN incompatible actors. Although these results do not affect the classification performance of the algorithm proposed in this work, they can give hints about how it might be used to aid programmers in writing actors or about which other classifications might be desirable to have.

The actors analysed here can be divided into two groups: determinate but KPN incompatible actors, which always produce the same output for the same input but cannot be expressed as Kahn processes, and non-determinate or ambiguously specified actors, which may produce different output for the same input. Both groups will be discussed in the following.

The group of **determinate, yet KPN incompatible actors** is quite diversified. In addition to actors similar to that shown in Listing 2.3 on page 23, it features two more kinds of actors.

One kind of actors performs multiple unrelated operations. These actors could, in fact, be replaced by multiple Kahn processes. A (sequential) Kahn process as defined in Definition 2.2, however, cannot produce this behaviour[4].

Another kind of actors contains two sets of actions: The first set describes the actor's main behaviour and is completely KPN compatible. In parallel to it, the second set has the task of pre-buffering input tokens in internal buffers of the actor. This is a low-level optimisation with the idea that if the actor cannot perform the main calculations, it can still use the time for pre-buffering data. While the order and the firing counts of the actions thus vary, the output is still always the same and these actors are determinate.

Non-determinate or ambiguously specified actors may produce different output for the same input. However, the two groups differ in one point: While ambiguous specifications should clearly be avoided, non-determinacy is sometimes necessary, for instance in the case of a video streaming application which has to react to video input not arriving within a certain deadline.

From the semantics of each of the 20 non-determinate actors amongst the actors regarded in this evaluation, however, it can be concluded that non-determinacy in

[4] For an example, cf. the Echo2 actor in Example 2.3.

Listing 2.5: Simplified example for an unintentionally KPN incompatible actor.

```
actor Sum() int DataIn, bool EndOfStream ==> int SumOut:
  int sum := 0;
  readData: action DataIn:[i] ==> do sum := sum + i; end
  done: action EndOfStream:[eos] ==> SumOut:[sum] end
  priority
    readData > done
  end
end
```

these cases is unintended. This meets the fact that all of the applications (video decoders, cryptographic applications etc.) in the set are supposed to be determinate.

The possible programming mistakes we identified amongst non-determinate and ambiguously specified actors are often the same. In most cases, it seems the author of the concerned actor did not realise that two guards actually overlap each other. In the case of ambiguity he may also have forgotten to specify a higher priority for one of the actions. The fact that most backends in such cases typically fire the action which comes first in the source code leads unfortunately often to this kind of error not being discovered.

In other cases assumptions about the input are made, usually founded on the concrete data sent in a particular graph. However, the behaviour of an actor is clearly defined only if it is unambiguous for *any* input.

In the case of non-determinacy, we found another pattern, which is illustrated in Listing 2.5. This actor reads data on one channel and is informed about the end of the data stream on a second channel (typically, both channels come from the same process). While this apparently worked well in the tests of the programmers, wrong output would be produced if the data channel delayed the tokens for longer than the end-of-stream channel, such that the end-of-stream token arrived before the last data tokens. This situation could be avoided if the data channel supported the transmission of special control tokens. In this case, the second channel would not be necessary and the actor would actually be KPN compatible.

All these results show that KPN-incompatibility is often unintended. Especially for larger actors (there are several with more than 2000 lines of code), a KPN compatibility analysis, as performed by the algorithm presented here, may thus prove to be a valuable tool for a programmer, even if he does not target a KPN implementation of his actors.

Table 2.3: Execution time of CAL actors when being scheduled either using the default scheduler of ORCC or as a Kahn process.

actor	platform	scheduler		speed-up
		ORCC	Kahn	
mvseq	DSP	156 691 cycles	85 249 cycles	1.84 x
invpred	DSP	129 641 cycles	83 478 cycles	1.55 x
mvseq	RISC	241 544 cycles	151 636 cycles	1.59 x
invpred	RISC	453 836 cycles	230 188 cycles	1.97 x

2.2.4.5 Performance of a Dataflow Actor and a KPN Process

Next, we evaluate if the proposed translation of a KPN compatible actor to a Kahn process can be used to improve the performance of dataflow graphs. To this end, C versions of the translated Kahn processes were generated as described in Section 2.2.3.3 and compared to C code generated conventionally by ORCC [Yvi+13].

The code was compiled and then run on two systems:

- A Texas Instruments TMS320C6416 Fixed Point DSP featuring L1 instruction and data caches of 16 KB each. The evaluation was done on the Texas Instruments cycle accurate device simulator, which takes account of cache behavior. The CCS IDE version was 5.5.0.
- An Altera Nios II/f RISC processor with 4 KB L1 instruction and 2 KB L1 data cache. The evaluation was done by synthesizing the processor core on an Altera Stratix III FPGA and by measuring the cycle time with the SignalTap II logic analyzer. The Quartus II software version was 13.1.

The measurements were performed with two different actors:

- "Mgnt_MVSequence_LeftAndTopAndTopRight" (mvseq) and
- "Algo_DCRInvPred_LUMA_16x16" (invpred),

both from the MPEG-4 Part 2 Simple Profile decoder. The former consists of 7, the latter of 10 actions. The achieved results are summarized in Table 2.3.

For these actors, a speed-up between 1.55 x and 1.97 x was achieved. The reason for these improvements is that, instead of linearly iterating over all actions like CAL implementations, the KPN translation follows the structure of the PST, i.e. it performs a sort of binary search for the next action to be fired. It also does not need to check the availability of tokens.

Of course, the influence of this overhead reduction decreases with a higher computational complexity of the actions to be fired. Still, the examples show its relevance in real production code.

2.2.5 Related Work

Classifying dataflow actors into more restrictive dataflow models has recently been considered as an efficient technique to improve the execution performance of dataflow graphs, e.g. by reducing the number of communication channel accesses. In particular, a methodology to classify dataflow actors into SDF and CSDF actors is presented in [Zeb+08]. In order to model dynamic and time-dependent behavior, each actor is described by a finite state machine that controls the communication behavior of the actor. In contrast to our work, their approach is limited to only classify static dataflow actors.

A method to classify dataflow actors into static, dynamic, and time-dependent actors is presented in [WR12] based on satisfiability and abstract interpretation. This method has been discussed in Section 2.2.4. While it can identify SDF, CSDF and quasi-static actors, which are KPN compatible by construction, it cannot identify more general patterns of KPN compatibility. The method for detecting time dependency could be used for showing KPN incompatibility, but is somewhat inaccurate as seen before. With its ability to identify (quasi) static actors, it can, like [Zeb+08], be regarded as a complement to our approach.

In [PSB09], a scheduling approach for semi-dynamic dataflow graphs is presented. To this end, a novel dataflow model is introduced that constructs actors with sets of modes representing fixed behaviors. Then, it is shown that a set of static dataflow graphs can be derived by decomposing the graph by its modes.

Other approaches trying to improve the performance of dataflow actors exist, e.g. by scheduling the actor more efficiently. For instance, the technique proposed in [BRS12] identifies most scheduling decisions of a dynamic dataflow actor at compile time by determining most of the schedule statically. In [Ers+12], this approach has been extended to also analyze the state space of certain network partitions.

Outside the field of the CAL language, the problems of availability of input variables and of obtaining them has been discussed as well. Among others, [BC82] and [Win87] analyse formal representations of algorithms with a particular focus on fetching variables. [LP95] sketches a theoretic method for analysing actors in dataflow process networks for actions with common input. These actors, however, are stateless and have less complex guard functions than CAL. [BCG00] investigates in the domain of synchronous programming whether a given module can iteratively infer the availability of all required input variables from its state (*endochrony*). All these approaches have in common that they work on program descriptions in which

every input variable has to be fetched explicitly. This form of specification has natural representations as trees or graphs similar to the PST shown in this work. In dataflow networks, however, fetching *all* input variables upon firing an actor is *one atomic operation* as well as checking if an actor can fire. Breaking up these atomic operations and constructing a PST is thus less obvious than with other programming models.

2.2.6 Conclusion

In this section, we have presented a novel algorithm to classify dataflow actors that are specified according to the CAL language into KPN compatible and potentially KPN incompatible actors. A dataflow actor is KPN compatible if it can be represented as an infinite program that only performs blocking, destructive read accesses, calculations and non-blocking write accesses. Based on the classification algorithm, we have described a formal method to translate a KPN compatible dataflow actor into a Kahn process. We have demonstrated the viability of our algorithms by implementing them in the RVC-CAL framework. In fact, the proposed classification algorithm has been capable to identify 93 % of all KPN compatible, mature actors from the ORCC benchmark suite. The performance of KPN compatible actors can be improved by up to 1.97 x when executing them as Kahn processes instead of CAL actors.

Manual code inspection found reasons for KPN incompatibility and in particular time-dependent actors. While time-dependency is an important language feature of CAL, in all the cases observed in the analysed code base, it was unintended. Section 2.4 will discuss in detail what these findings mean for the optimal choice of an implementation model.

2.3 Deterministic Memory Sharing in KPNs

The last section showed how switching to a KPN implementation model can improve the performance of a CAL program. However, as this section will establish, also the KPN implementation model can be inefficient in certain cases, especially when implemented on shared memory. Given that many modern multi-core platforms concentrate on shared memory as a means of communication and data exchange, this is a relevant drawback.

In the following, a concept for deterministic memory sharing in KPNs will be introduced which can be applied to any KPN. It allows to take advantage of shared memory data exchange mechanisms while still preserving determinacy. Throughout the section, an ultrasound image reconstruction algorithm will be used as an

example for a high-throughput real-world KPN application. It will be shown where the deficiencies of conventional KPN implementations lie for this class of applications and that applying deterministic memory sharing combines significantly better performance with a drastically smaller memory footprint. These insights arise from a collaboration with Harshavardhan Pandit and Pratyush Kumar.

2.3.1 Introduction

While KPNs can be thought of as advocating message passing to correctly manage concurrency, the current trend in hardware platforms, especially in the embedded domain, goes into another direction. Many of the modern multi-core platforms (e.g. Kalray MPPA, Intel Xeon Phi, PULP, adapteva Epiphany or ARM Multicore platforms) count on shared memory architectures for data exchange. These platforms minimise the latency of accessing shared memory using hardware features such as crossbar communication, multi-banked memory modules and hierarchical cache architectures. Applications with high communication demands need to make use of these features in order to attain maximum efficiency on such platforms. Traditional KPNs, however, explicitly do not allow this. It would thus be desirable to combine the determinacy of KPNs and the improved performance of memory sharing on modern multi-core architectures.

Hereinafter, we propose a set of transformations that enable the use of shared memory communication patterns without affecting the determinacy of KPNs. We call this Deterministic Memory Sharing (DMS). There are four primary features of DMS. First, we propose transformations to convert any standard channel of a KPN to allow a memory block to be shared between the producer and consumer processes of the channel. Second, we propose transformations to allow multiple processes to concurrently read from a shared memory block. We formalise the intricate conditions under which such concurrent reads can be allowed. Third, we propose using memory blocks for in-place modifications and direct re-transmissions. Fourth, motivated by streaming applications which expose data parallelism, we propose dividing a memory block into smaller sub-blocks, which can be concurrently read from and written to by different processes.

In addition, we propose the insertion of *recycling* channels which significantly reduce the cost of allocation and deallocation of the shared memory blocks by allowing reuse of memory blocks once they are not used by any process. All the transformations mentioned above have been conceived such that they can be employed selectively to transform a standard KPN into a DMS-enabled KPN, sequentially and only looking at one process at a time. We show that applying each set of transformations preserves the determinacy of the KPN, by construction.

We illustrate DMS with an ultrasound image reconstruction algorithm, which is representative of most streaming applications: There is a large communication overhead, and explicit data and task parallelism. We show that a subset of our proposed transformations can be employed to correctly transform the KPN model of the algorithm. We implement the original KPN, the transformed DMS-enabled KPN, and a windowed-FIFO-based KPN [HGT07], using the DAL framework [Sch+12], on the Intel Xeon Phi processor. With extensive experimental tests we conclude that sharing memory enables a higher throughput of the application, while using a smaller memory footprint.

The remainder of the section is structured as follows: In Section 2.3.2, related work is reviewed. In Section 2.3.3, the ultrasound image reconstruction algorithm is presented in detail. In Section 2.3.4, the basic ideas of memory sharing in KPNs and how this can be done deterministically will be explained. In Section 2.3.5, these ideas are formalised. In Section 2.3.6, transformations will be given for existing KPNs to apply these ideas and to optimise the resulting networks. In Section 2.3.7, the correctness of these transformations will be shown. In Section 2.3.8, the ultrasound imaging algorithm is revisited and it is demonstrated how the transformations from the section before can be applied to it. Section 2.3.9 will show how the concept can be implemented in C code. Finally, experimental results are presented in Section 2.3.10.

2.3.2 Related Work

Memory related publications in the domain of process networks can be divided into three groups. The first group regards channel capacities (i.e. the amount of tokens the actual implementations of the individual KPN channels can hold), usually trying to minimise the memory footprint via these capacities. The second group tries to further reduce the memory footprint by reusing the same memory for multiple channels. A third group finally tries to avoid unnecessary copying overhead rather than looking at the memory consumption.

In the first group, which regards channel capacities, one idea is to start with small channel capacities, dynamically increasing them as required [Par95; BH01]. Another approach is to entirely eliminate certain channels by automatically merging processes where appropriate [SLA12]. For networks with regular patterns, such as synchronous dataflow [LM87] or cyclo-static dataflow [Bil+96], the minimal channel capacities can be calculated at design time. There are a number of approaches which try to further reduce these minimal capacities for special cases of these dataflow graphs [OH04; VNS07] or performing special analyses [Cho+07].

All the methods mentioned so far are complementary to our work; we assume these optimisations, if required, to have been carried out prior to applying the transformations presented here.

Another work on channel capacities, however focusing on their relation to application performance, is [CHB07]. While this relation is regarded as a trade-off there (more memory for better performance), we can reduce the memory footprint of an application and achieve a better performance at the same time.

In the second group, which reuses buffers for multiple channels, [OH00] tries to minimise the memory footprint of synchronous dataflow graphs in the case of single-core systems with pre-determined schedules. While greatly reducing memory requirements, it is unclear if this approach could be applied to multi-core implementations.

[Goo01] uses a global memory manager. Processes can obtain buffers through the memory manager from so-called pools. The programming model used there is not compatible to KPNs, though; in fact, it is non-deterministic. Also, it is the task of the programmer to decide which pool to obtain buffers from or how many buffers these pools should be provided with. In this work, we show how a deterministic, DMS-enabled application can be obtained from any KPN by applying simple transformations. No complicated synchronisation or buffer allocation decisions have to be taken care of by the programmer.

In [PCH02], the SDF model is extended with a sort of global buffers for keeping track of common global states such as sampling frequency or gain in a multimedia stream. The motivation of that work are synchronisation purposes; memory footprint or performance are not taken into account.

The papers from the third group do, although often achieving it, not primarily target a low memory footprint. Their main idea is rather to boost the application performance by avoiding unnecessary copying overhead. In [HGT07], this is done by using so-called *windowed FIFOs*, which can replace the standard FIFOs in KPN channels. Instead of copying the tokens from a sender or to a receiver process, a windowed FIFO provides these processes with a shared memory region (a *window*) they can directly write to or read from. A technique of managing the access to this window ensures that processes do not overwrite unread data or do not read stale data. This saves a certain amount of copying overhead between two processes; still, when the data has to be passed on to a third process, copying cannot be avoided.

A more general approach is discussed in [Sat+03]. The idea there is that processes can allocate memory blocks, and then send tokens representing these blocks over the channels. The token gives a process the permission to access the corresponding memory block. Further, processes can send read-only copies of tokens to multiple receivers. However, it is not clearly mentioned whether the data in these memory blocks can be edited *in-place* and then sent on to another process. Also, the notion of block allocation and deallocation is only abstractly defined; using memory allocation functions provided by an operating system would be rather slow for regular allocation as part of a data streaming process network.

In this section, we generalise the concepts from the third group of works mentioned here and we discuss several implementation details. We show simple transformations allowing to turn a traditional KPN into one using shared memory, still staying deterministic. Also, we introduce new techniques, such as recycling channels and memory sub-blocks. Furthermore, formalise the concept of in-place editing in KPNs and show how it can be implemented.

2.3.3 Ultrasound Image Reconstruction

In the following, we will describe in detail the ultrasound image reconstruction algorithm we use for our experiments. It will be shown what it does and how it can be implemented as a KPN.

The different hardware variants, methods, reconstruction algorithms and parameters of ultrasound imaging are as manifold as the applications in medicine and in other domains. In this work, we will limit ourselves to one single configuration, which we implement in different ways. First, we will explain the general principle of ultrasound imaging. Then, we will give details about the individual steps to be performed during image reconstruction. Finally, we will show how the whole algorithm can be implemented efficiently.

2.3.3.1 Principles behind ultrasound imaging

In the medical domain, one typically uses sound waves with a frequency of 1 to 50 MHz, which are simplifyingly assumed to travel at constant speed through human tissue. At every boundary of materials with different physical properties, transmission, absorption and reflection occur. The latter effect is taken advantage of for ultrasound imaging.

The tool for obtaining the images is called a *probe* and, in our case, is an array of linearly arranged piezoelectric crystals called *transducers*. A transducer changes its shape when subjected to an external voltage and can therefore be used to generate sound waves. Conversely, when changing its shape due to mechanical pressure (like sound waves), it produces a voltage which can be measured.

The image capturing process now works as follows. First, a plain wave signal is sent out from the transducers; this signal is e.g. a short window of a sine wave. As the wave travels through the tissue, it gets reflected varyingly strongly at the different locations. After sending out the signal, no more voltage is applied to the transducers and instead, the voltage generated by the transducers is measured over time. For n transducers, this gives n individual traces of recorded sound waves. From these traces, a two-dimensional image is reconstructed. This can be done using the

algorithm which we implement and optimise in this work, and which is described in the next section.

2.3.3.2 Individual steps of the algorithm

The image reconstruction can be decomposed into multiple independent steps, which are explained in the following.

Attenuation compensation: The longer a wave travels through the tissue, the more it gets attenuated. This attenuation can be calculated and reversed on each trace by multiplying the samples with an exponentially growing function.

High pass filter: Each trace is convolved with a high-pass filter to eliminate DC biases.

Beamforming and apodisation: This is the most important reconstruction step. When the signal is reflected, the reflection arrives at all transducers, however at different points in time due to the different geometrical distances between the reflection origin and the transducers. For every geometrical position, one can calculate at what times a reflection from there arrives at the individual transducers. The different samples at these times are summed up for all positions considered, thus creating a first image. The image quality can be improved by weighting the different samples according to the angle in which the reflection hits a transducer. This is called apodisation. In practice, for each transducer, one image column is calculated such that all the samples from the transducer's trace can be used. For each column, this can be achieved by taking the prepared traces from all transducers, extracting samples at precalculated indices, multiplying the extracted samples with precalculated factors and finally summing the results up. All these operations are element-wise vector operations.

Demodulation: The beamformed image still contains the sine waves of the echoed signal. These are removed by applying an envelope detection and a low-pass filter. Both can essentially be implemented as a convolution.

Log compression: To stress differences at weaker reflections, the logarithm is taken of each point in the image.

Figure 2.5 shows how the ultrasound image reconstruction can be implemented as a KPN. An input process obtains the data from the transducers, which is then

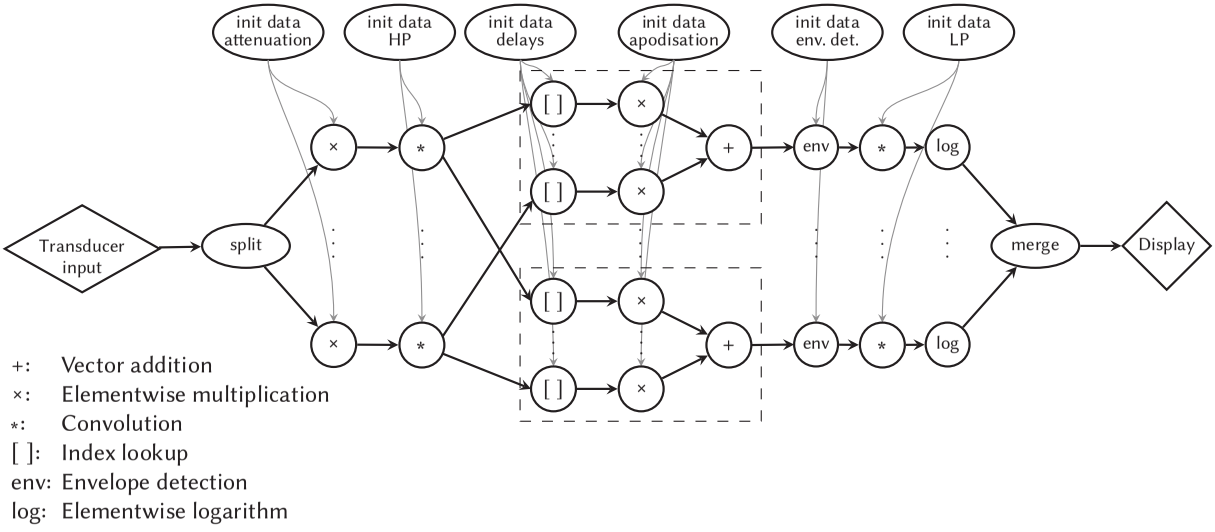


Figure 2.5: KPN implementation of the ultrasound image reconstruction algorithm. The dashed rectangles group all the processes involved in the beamforming of one image column each.
The processes in the upper-most row precalculate certain vectors and tables like the filter kernels, the delay indices for the beamforming process or the apodisation tables. This precalculated working data is then sent to all the processes which need it. The processes will read the data once and keep it stored; afterwards, they will no longer read from those channels and instead keep working on the incoming samples.

split into the individual transducer traces and sent through the channels. The computation processes accomplish rather simple work like convolutions, element-wise multiplications or index-lookups. At the end, the final image is merged together.

2.3.4 Deterministic memory sharing

It can be easily seen from the last section that the ultrasound image reconstruction algorithm works on large amounts of data, which need to be exchanged between different cores on multi-core systems. Also, multiple processes have to work on the same data. In a traditional KPN, all this data has to be sent over the channels, and there has to be a separate copy of it for each process. Clearly, this leads to a considerable overhead. We try to avoid this overhead using a different method of data exchange that is based on memory sharing.

This section explains how memory blocks can be shared by multiple Kahn processes while still preserving the advantages of the KPN model, in particular its determinacy. We will introduce the basic ideas of KPN memory sharing and that of an efficient memory management technique.

2.3.4.1 Basic idea of the model

As previously mentioned, our approach is to have multiple KPN processes share certain memory blocks. In general, however, when multiple processes share one memory block, they can communicate through it, thereby circumventing the actual KPN communication mechanism (which uses channels). This would not only destroy the determinacy of the process network, it would also reintroduce races, glitches and all the other multi-processor issues KPN originally set out to avoid. Therefore, it is essential to have a synchronisation mechanism which regulates the accesses to shared memory blocks.

This can be done by the concept of *access tokens*. An access token gives a process the right to access (i.e. read from and write to) a certain memory block. There is only one access token for each memory block, and a process is not allowed to create copies of it. This ensures that only one process can access a memory block at a time. Access tokens can be sent to other processes over the conventional KPN channels; the sending process has to destroy its local instance of the token once it has sent it (s.t. there are no two copies of the token).

In summary, instead of sending data directly over a channel, the data is stored in a memory block, the access token to which is then sent over the channel. This is illustrated in Figure 2.6. We will show in Section 2.3.7 that the determinacy property of KPN still holds when sharing memory through access tokens.

Figure 2.6: Two KPN approaches: a) Classic process network; b) Process network with shared memory blocks. The spade represents an access token linked to a memory block with the "classic" token from before.

2.3.4.2 Relaxations and additions to the model

The mechanism presented above already brings clear improvements, but there are more possibilities of eliminating overhead. In this section, we will discuss how it can be relaxed in order to allow further useful optimisations.

Multiple memory copies
One desirable feature would be to avoid multiple copies of the same data. This can be achieved by allowing multiple processes to simultaneously share one memory block. However, that leads to problems when these processes simultaneously write (or read and write) the same memory locations. As it was shown above, this would violate the determinacy of the KPN. On the other hand, it is not a problem if multiple processes concurrently *read* from the same memory locations. Thus, it is possible to relax the uniqueness constraint for the access tokens in a way that access tokens can be duplicated if it can be guaranteed that no write accesses are performed to the memory blocks they are linked to. Conversely, one can say that memory blocks may not be written to if multiple access tokens are linked to them. This relaxation does not compromise the determinacy of the KPN for the simple reason that no communication can be established by only reading.

A typical use case for duplicating access tokens could be as follows. Some data is produced and written to a memory block. Once the writing is finished, the producing process duplicates the access token multiple times and distributes the access tokens linked to the memory block to multiple receivers, which can then read it simultaneously.

Different levels of process granularity
A second relaxation that can be made to the access-token principle helps to deal with different levels of granularity of different processes. Depending, for instance, on the workload of different tasks, it may be advantageous if one process works on a large amount of data and afterwards multiple processes work on distinct subsets of this data. After that, it may be desirable again to have one more process working on the entire set of data. This can be allowed if it is ensured that these subsets are distinct, i.e. that in the memory block, the locations accessed by the different processes do

not overlap. Again, it must be ensured that these processes cannot communicate through shared memory blocks.

To this end, we introduce the notion of *memory sub-blocks*. A memory block can be split up into multiple sub-blocks, which denote distinct, non-overlapping memory regions in the original memory block. Every sub-block can now have its own access token. The different access tokens can be sent to different processes, which can then only access different memory regions. Thus, no communication between them through the memory block is possible.

It is also possible to introduce a reverse operation to the *split* mechanism, which we will call *merge*. The merge operation can join two (or multiple) adjacent memory sub-blocks to one single (sub-)block, reducing the set of access tokens provided to it to one single access token.

Memory recycling

Until now, memory blocks have been discussed as something that exists, but without mentioning where they actually come from. The conventional way of obtaining them is through dynamic allocation [Sat+03]. Similarly, they are deallocated when no more access token is linked to them. This, however, may be rather slow on many systems or even not supported by the underlying software stack. Thus it appears sensible to look for alternatives to dynamic memory allocation.

One such alternative would be to introduce a *recycling channel* which goes from a consuming to a producing process. Instead of deallocating memory blocks, the consuming process sends the access tokens linked to them back to the producing process for later use. Initial access tokens (with corresponding memory blocks) are placed on the recycling channel such that the producing process can always use this channel as a source for obtaining (access tokens to) memory blocks. We call this technique memory block *recycling*.

In general, this change to the process network also changes the semantics of the program, especially when there is no more access token on the recycling channel and the consuming process blocks trying to obtain one. However, we will show that under certain conditions, the same change in semantics is also induced by using channels with limited capacity (which is anyway necessary when actually implementing a KPN).

2.3.5 Formalisation of the model

Above, we have informally described the different ideas DMS is based on. Now, we are going to give a formal definition of all the mechanisms included. This will help in the next sections, when we discuss KPN transformation methods and show their correctness. For this purpose, we will first define the data structures involved

and the properties they have. Afterwards, the different operations on these data structures will be defined.

The model works on *memory blocks*, which are regions of memory that can be shared between processes. $\mathscr{B}$ is the set of memory blocks. For each block $b \in \mathscr{B}$, there is

- $\text{size}(b) \in \mathbb{N}$, the size of the memory block
- $b[n],\ n \in \{0..\,\text{size}(b)-1\}$, the access operator for the memory block. It returns a memory location which can be read or written.

The processes do not have direct access to the memory blocks. They can only access the blocks through *access tokens*. An access token is an abstract entity with the following properties:

1. It is linked to a memory block.
2. It allows read accesses to the block.
3. It only allows write accesses to the block when it is the only access token linked to the block.
4. It can be sent over KPN channels.
5. Send operations are destructive, i.e. the sending process does not retain a copy of the token sent.

$\mathscr{T}$ is the set of *access tokens* which currently exist in the application. For each $t \in \mathscr{T}$, there is

- $\text{link}(t) \in \mathscr{B}$, the memory block the token is linked to.
- $t[n] := \text{link}(t)[n]$, the access operator for the access token.

The subject of whether write accesses are allowed or not needs some more discussion. It relates to a global property, which we call *ownership*: A process *owns* a memory block if it has the only access token linked to that block. One could have mechanisms to check for ownership at runtime before each write operation. This, however, would have to be done carefully in order not to allow global communication through this checking mechanism. In this work, the approach is to formally ensure at design time that write accesses are only performed when they are allowed.

For defining split and merge operations later, we also need the concept of a *sub-block*. A sub-block is a part of a memory block, which can be created by splitting a

memory block. A sub-block behaves like a normal block, in particular it can have access tokens linked to it.

With these definitions, we can now describe the set of DMS operations that can be executed by the processes.

Allocation: Creates a new memory block of a given size and returns an access token to it.

Duplication: A copy of a given access token is created, thereby ending a possible ownership of (and thus inhibiting further write accesses to) the memory block linked to the token.

Splitting: The memory block (or sub-block) a given access token is linked to is split into two or more sub-blocks. The access token which was provided is destroyed. Instead, access tokens to the sub-blocks are returned. For simplicity reasons, we demand that the calling process must own the memory block for splitting. All sub-blocks created are then owned by the calling process, but can later also be owned each by different processes.

Merging: Two or more sub-blocks of the same memory block that are adjacent in memory are merged together to a bigger sub-block or back to the entire memory block. The calling process must own all the sub-blocks. The access tokens which were provided are destroyed. Instead, an access token to the merged (sub-)block is returned.

Release: Destroys an access token. If the calling process owns the memory block linked to the access token, this block is destroyed as well. In the case of a sub-block, destruction only happens to the entire memory block once the last access token linked to it or one of its sub-blocks is released.

These operations can now be used to implement a DMS-enabled KPN.

2.3.6 Translation to DMS

In the last sections, the idea of DMS was explained and formally described. However, it is not clear yet how it can be applied and in particular how an existing KPN can be transformed to use DMS. Note that the transformation process must be organised such that the generated code follows DMS rules and that the semantics of the original KPN are preserved (no deadlocks are introduced etc.).

It is important to note that the translation is not an all-or-nothing operation; in fact, it may sometimes be advantageous to convert a KPN only partially. Note that due to the high expressiveness and the intricate process interactions of KPNs,

it is not possible in general to always implement the optimisation ideas shown in Section 2.3.4.

This section will introduce a set of transformations that can be carried out in process networks and we will try to establish an intuitive understanding for them. We will show in the next section that they all are correct and do not change the semantics of the process network.

We have conceived the translation such that it works step by step, each time applying one transformation. The translation is performed in three stages:

1. Basic transformations: All channels are transformed to DMS for which this is desired. Recycling channels can be added.
2. Optimisation: The performance of the application is increased and its memory footprint is decreased by taking advantage of techniques like splitting and merging or token duplication.
3. Final clean-up: The attained process network is simplified.

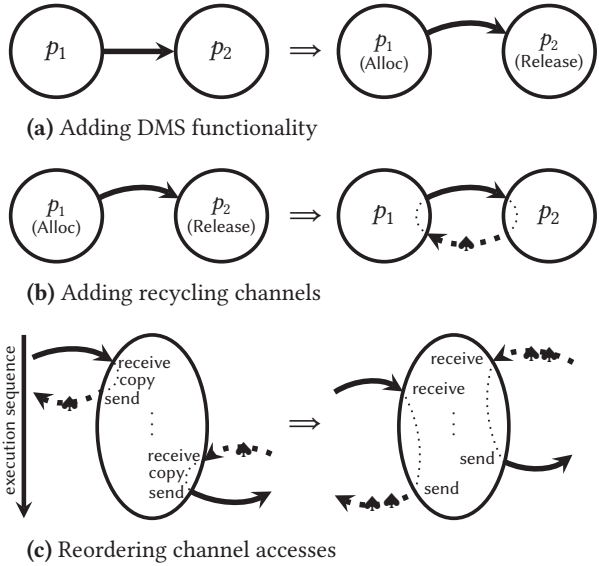
In the following, the transformations in each stage shall be discussed. Afterwards, it is shown how the optimisation transformations can be coordinated.

2.3.6.1 Basic transformations

The basic transformations are described below and illustrated in Figures 2.7a to 2.7c on the following page. We assume each channel to have been assigned a maximum capacity. (This is necessary for any implementation. Dynamically increasing channel capacities later as in [Par95; BH01] is also possible with our approach.)

Transformation 1: Enabling DMS on KPN channels
For every channel in the network that is intended to use DMS, its sending process and its receiving process are altered such that they send and receive access tokens instead of traditional "data" tokens. The access tokens are obtained by allocating memory blocks in the sending process and directly released after reading in the receiving process.

Transformation 2: Adding recycling channels
For each channel converted to DMS as shown above (referred to as *data channel*), a recycling channel can be added. The recycling channel goes into the opposite direction of the data channel and has the same capacity. Initial access tokens linked to separate memory blocks are added to the recycling channel such that the total number of initial tokens on both channels is equal to the capacity of the data channel. Releasing tokens and memory block allocation are replaced with sending and receiving tokens from the recycling channel, respectively.



(a) Adding DMS functionality

(b) Adding recycling channels

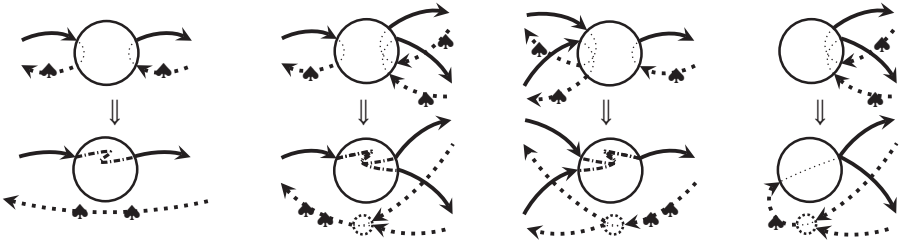
(c) Reordering channel accesses

Figure 2.7: Illustrations showing the basic transformations for introducing DMS to a KPN. Recycling channels are shown as dotted lines.

Transformation 3: Reordering channel accesses
After applying the transformations above, accesses to a data channel and its corresponding recycling channel happen in pairs, i.e., one channel is accessed directly after the other, with no other channel access in between. This means that only one access token is available to a process at a time. Simultaneous access to two or more memory blocks can be achieved by moving reads from or writes to recycling channels such that they happen earlier or later in the execution path, respectively. However, an additional initial access token may have to be added to the recycling channel in order to prevent a change in the semantics of the process network.

2.3.6.2 Optimisation transformations

For the optimisation transformations, we will give a non-exhaustive list of the most common optimisations that can be applied to a DMS-enabled KPN. We will assume recycling channels have always been added to the channels involved (the other case can be easily derived). All these transformations can be applied by looking at one process and a *subset* of its data channels. We look at processes which access all the channels in this subset only in the form of *elementary transactions*, which consist of reading/writing exactly one token from/to each channel. Note that this only restricts the access pattern of a process concerning the *subset of channels considered*. Its other behaviour – in particular, its accesses to other channels – does not play a role.



(a) In-place editing **(b)** Splitting **(c)** Merging **(d)** Duplicating access tokens

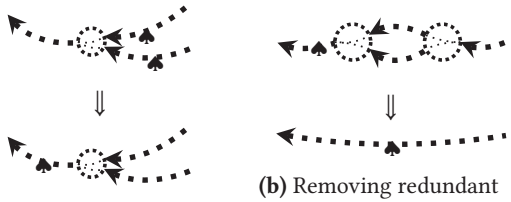
Figure 2.8: Illustrations of DMS optimisation transformations. Dummy processes generated during transformation are shown as dotted circles.

We look at three groups of transformations here, which are illustrated in Figures 2.8a to 2.8d.

Transformation 4: In-place editing
If a process always reads from one channel c_i and writes to another channel c_o, and if the operations to the two memory blocks concerned are such that they could happen *in-place* in only one memory block, then the process can be altered such that it performs the in-place operation on the memory block received from c_i and then sends the access token to c_o. The recycling channels corresponding to c_i and c_o are joined as shown in Figure 2.8a, with their capacities and number of initial tokens adding up for the joint recycling channel.

Transformations 5 and 6: Splitting and merging
These transformations work in a similar way as in-place editing. The difference is that in the case of split, the input memory block is split into smaller sub-blocks, which are then distributed to multiple output channels. In the case of merge, multiple input-sub-blocks are merged to give the output block. As sub-blocks can only be merged when they belong to a common memory block, a dummy split process has to be generated as a part of the recycling infrastructure when applying a merge transformation (cf. Figure 2.8c). In the case of a split transformation, a dummy merge process is generated to ensure correct memory block recycling (cf. Figure 2.8b).

Transformation 7: Duplicating access tokens
If a process always sends the same data to multiple channels, it can be transformed such that it only allocates one memory block which is filled with this data. The access token to this block is then duplicated and sent to each of the channels. As with split, an additional dummy process is generated in order to collect all these duplicate tokens again, such that the memory block can be safely recycled (cf. Figure 2.8d).



(a) Gathering initial tokens

(b) Removing redundant splits and merges

Figure 2.9:
Illustrations of DMS simplification transformations. Dummy processes generated during previous optimisation transformations are shown as dotted circles.

2.3.6.3 Simplifications

Simplification transformations become necessary due to the overhead caused by the previous optimisations. While this overhead is necessary to ensure correctness of the optimisations, it can be safely eliminated after they are finished. We limit ourselves to two important simplifications, which are illustrated in Figures 2.9a and 2.9b.

Transformation 8: Gathering initial tokens
Initial access tokens should always be linked to entire memory blocks, not to sub-blocks. Therefore, initial access tokens linked to sub-blocks are moved behind a merge or in front of a split. This implies a merge operation on the initial access tokens, as illustrated in Figure 2.9a. Should the number of initial tokens on different branches differ, this situation can be resolved by adding additional initial access tokens to certain branches.

Transformation 9: Removing split and merge processes
In certain situations, a dummy split and a dummy merge process just neutralise each other after a sequence of transformations. In that case, both can be removed and the channels connected to them are joined, as shown in Figure 2.9b.

2.3.6.4 Optimisation coordination

All of the optimisation transformations shown above can be applied if their requirements are met; however, not all of them can be applied together. For instance, an in-place edit transformation is not allowed after an access token has been duplicated. As transformations are applied individually to the processes, one after another, a mechanism is required which keeps track of the transformations applied and reveals which transformations are still allowed. In particular, one must be able to prove at design time whether or not a process, after a given sequence of operations, owns the memory blocks arriving from a certain channel. For this purpose, we use

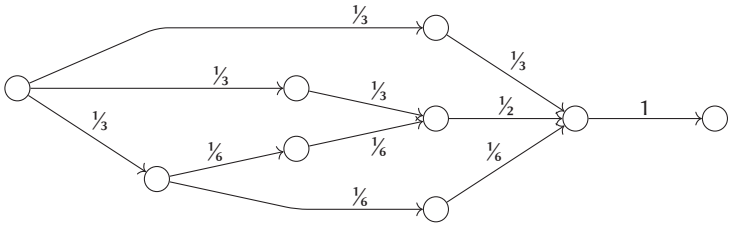


Figure 2.10: Example for the ownership annotation of channels. The graph shown only contains channels transporting duplicate access tokens linked to the same memory blocks. Note that the rightmost process owns the memory block.

an *ownership annotation* of the channels: $\text{own}(c) \in (0, 1] \cup \{*\}$ for every channel c using DMS.

Directly after applying the basic transformations to a channel, its target process always owns the memory blocks sent over it. On the other hand, it only reads from the blocks and then recycles them, i.e. it does not need to own them. Therefore, the ownership annotation is $*$ directly after the basic transformation to indicate that optimisations changing the ownership are still possible.

When a process needs ownership of the tokens coming through a channel (i.e. for in-place edit, merge and split transformations), the annotation of the channel is changed to 1 during the transformation. When a token is duplicated, the channels which the duplicates go to are marked with an ownership < 1. Once a channel has been annotated with an ownership other than $*$, this annotation must not be changed any more. Any transformation may only be applied if the corresponding annotations are still possible or if the channels already have the annotation the transformation would entail.

In case of token duplications or collecting duplicates, the exact value of the ownership annotation is determined such that the sum of the annotations of the outgoing channels is always equal to that of the incoming channels carrying access tokens linked to the same memory blocks. In case of token duplication, this means that the annotation value of the incoming channel (or recycling channel, i.e. 1) is divided by the number of duplicates. In case of collecting duplicates, the annotation values of the incoming channels are added up to obtain the value for the outgoing channel. This is illustrated in Figure 2.10. As the ownership value is always one before the first duplication, the sum of the ownership values after any combination of duplications and collected duplicates is always equal to one. If all the channels transporting a duplicate of an access token are collected again, the resulting ownership value must be one. Conversely, if there is one duplicate channel which has

not yet been collected, this ownership value cannot be one because the ownership value of the uncollected channel is greater than zero by definition. Remember that an ownership annotation of one for a channel means that its target process owns the memory blocks linked to the tokens it receives from that channel.

2.3.7 Correctness of the Translation

In this section, we show that the translation described in the previous section is correct. This comprises three points:

1. Memory integrity: We show that all memory blocks are correctly deallocated.
2. Determinacy: We show that the modified KPN is still determinate.
3. Semantics: We show that the semantics of the original KPN are preserved.

The first two points will be shown in general, whereas the third point will be shown to hold for each transformation individually.

2.3.7.1 Memory integrity

If an access token is sent to a channel – including recycling channels –, the memory block linked to it stays in use and must not be deallocated. If an access token is released, the memory block linked to it will be deallocated if it is no longer in use, as described in Section 2.3.5.

With the transformations shown in this work, one of both is always the case in each process for each access token. Therefore, no memory leaks can occur among the memory blocks allocated using the DMS mechanisms.

2.3.7.2 Determinacy

To show the determinacy of the modified KPNs, we have to show that they meet two properties:

1. Communication only happens through channels.
2. Read accesses on the channels are blocking and destructive.

Then, determinacy follows from [Kah74].

The second property is inherited from the underlying KPN channels, which still transport the access tokens.

The first property is met because a process can only write to a memory block if it owns it, i.e. if no other process can access the block. The only possibility for the

process of making the data available to other processes is to send the access token over a channel. At this point, it loses the access to the memory block and thus cannot use it for any further communication. Therefore, any data transfer needs a sending operation over a KPN channel.

2.3.7.3 Semantics

In the following, we show that each of the transformations given in Section 2.3.6 preserves the semantics of the process network, i.e., that the same data is sent over the channels and that no deadlocks are introduced by the transformations[5]. For this purpose, we give exact specifications of these transformations and draw conclusions about their correctness.

Transformation 1 (Enabling DMS on a KPN channel)

Prerequisites: A channel c that does not use DMS
Annotations: $\text{own}(c) := *$

This transformation is always valid, since any data that can be sent using normal KPN channels can also be sent using DMS.

Transformation 2 (Adding recycling channels)

Prerequisites: A DMS-enabled channel c with fixed capacity

This transformation's influence on the process network semantics is identical to that of introducing feed-back channels as described in [Par95]. There, feed-back channels are used to model the fixed capacities of KPN channels. As we assume a fixed capacity for c, the transformation is neutral to the semantics of the process network.

Transformation 3 (Reordering channel accesses)

Prerequisites: A process p accessing multiple channels, at least one of which uses DMS

As mentioned in Transformation 2, a recycling channel models the fixed capacity of its corresponding data channel. Postponing a send operation to a recycling channel thus delays the reinstatement of available channel capacity after a (destructive) read. Similarly, preponing a receive operation from a recycling channel prepones the beginning of a write operation in the sense that channel capacity is claimed. A possibly blocking operation between a pair of data and recycling channel accesses

[5] We do not consider limits of channel capacities as a part of the semantics, since KPN channels are theoretically unbounded. If one needs this feature, one can always add feed-back channels.

may therefore, in connection with other processes, lead to deadlocks. As these deadlocks, however, are only related to a limitation of virtual channel sizes, they can be prevented by increasing these virtual channel sizes by adding initial access tokens to recycling channels.

The next transformation to be considered is in-place editing. To give an accurate specification of it, however, we need to formalise the conditions which guarantee that the transformation can be applied. To this end, we will introduce the notion of *inplaceable behaviour*. One precondition for inplaceable behaviour is that every write operation corresponds to a read operation and vice versa. We formalise this pattern with the term *regular behaviour*.

Definition 2.13 (Regular behaviour)**.** A process p performs an **elementary transaction** on a subset C_i of its input channels and a subset C_o of its output channels iff

- it reads exactly one token from each channel $c \in C_i$ and releases/recycles it after usage and afterwards
- it allocates/obtains through recycling exactly one memory block for each $c \in C_o$ and sends it over that channel.

p **behaves regularly** on C_i and C_o iff all accesses to the channels in C_i and C_o can be described as a sequence of elementary transactions on C_i and C_o.

Note that regular behaviour is defined with respect to particular sets of input and output channels. This means that it is not affected by any arbitrary accesses to other channels, see also Algorithm 2.5. Inplaceable behaviour can now be defined as follows:

Definition 2.14. The behaviour of a process p on an input channel c_i and an output channel c_o (other channels may exist) is **inplaceable** iff

- p behaves regularly on $\{c_i\}$ and $\{c_o\}$ and
- p never writes to memory blocks received from c_i and
- in every elementary transaction, a being the memory block received from c_i and b being the block sent to c_o, no location in a is read after writing to the corresponding location in b — more formally, there is no l s.t. p reads $a[l]$ after writing $b[l]$.

If this is the case, one can set $a = b$ because (i) before any write operation to a location $b[l]$, $b[l]$ is still undefined and $a[l]$ contains the expected value and (ii)

Algorithm 2.5: Example on regular behaviour. The given Kahn process receives access tokens from two channels X_1 and X_2 and sends access tokens to a channel Y_1. The function SomeOp arithmetically combines two blocks, storing the result in a newly allocated third block.
The process shows regular behaviour on $\{X_2\}$ and $\{Y\}$, since it repeatedly receives one token from X_2 and sends one on Y. It does *not* behave regularly on $\{X_1\}$ and $\{Y\}$, since each read on X_1 is followed by two writes on Y.

```
process MyProc (in channels: X1, X2; out channels: Y)
    while true do
        t1 ← Read (X1)
        t2 ← Read (X2)
        t3 ← SomeOp (t1, t2)
        Free (t2)
        Write (Y, t3)  // Destroys local copy of t3
        t2 ← Read (X2)
        t3 ← SomeOp (t1, t2)
        Write (Y, t3)
        Free (t1, t2)
    end
end
```

after a write operation to a location $b[l]$, $b[l]$ contains the expected value and $a[l]$ is not accessed any more.

Note that these conditions are only sufficient. Without a doubt, there is a plethora of other access patterns which also allow for performing operations in-place. Listing all of these, however, would go too far at this point. We believe that our definition of inplaceable behaviour holds a good balance between simplicity and practical applicability on the one hand and a large amount of real-world code examples it covers on the other. This leads to:

Transformation 4 (In-place editing)

Prerequisites: A process p with inplaceable behaviour on input c_i and output c_o
Annotations: $\text{own}(c_i) := 1$

The transformation of the process behaviour with respect to the tokens on the data channels does not change the semantics of the process network as it has been shown above. The same holds for the recycling channels, because (i) the overall number of initial memory blocks does not change and therefore all the data channels affected

still can be filled completely and (ii) the elimination of a receive and a send operation in p can only decrease, not increase, the possibilities for an (unwanted) blocking.

Note that, as described in Section 2.3.6.4, the ownership annotation of c_i cannot be performed if $\text{own}(c_i) < 1$. Therefore, this transformation cannot be applied to input channels transporting duplicated tokens.

To extend the definition of inplaceable behaviour for split and join operations, we gather multiple channels to one virtual channel, which we call *compound channel*. A *compound channel* over a tuple of channels is interpreted such that it reads/writes one token each from/to each channel of the tuple, virtually concatenating them to a *compound block*. For a tuple of n different memory (sub-)blocks $b_1 \ldots b_n$, the compound block k over these blocks then has an access operator defined as an access to the concatenation of all (sub-)blocks. Formally,

$$k[s(j) + l] = b_j[l] \quad \forall\, l \in \{0..\,\text{size}(b_j) - 1\} \quad \forall\, j \in \{1..n\}, \quad s(j) = \sum_{i=1}^{j} \text{size}(b_i).$$

With this notion, the definition of inplaceable behaviour naturally extends to sets of input and output channels:

Definition 2.15 (Inplaceable behaviour on channel sets)**.**
The behaviour of a process p on a subset C_i of its input channels and a subset C_o of its output channels is inplaceable iff there exist ordered arrangements x_i and x_o of the channels in C_i and C_o, respectively, such that the behaviour of p is inplaceable on the compound channel over x_i and that over x_o.

Transformation 5 (Splitting)

Prerequisites: Process p with inplaceable behaviour on input $\{c_i\}$ and output set C_o

Annotations: $\text{own}(c_i) := 1$

In general, we do not assume to have separate split processes, but rather consider the split as an operation happening inside a process of any kind. Therefore, we require inplaceable behaviour, allowing the process to write to the memory block before splitting it.

For the transformation of the process behaviour concerning the data channels, the above considerations apply. Likewise, the process network semantics are not altered by the transformations to the recycling channels, because (i) the overall number of initial memory blocks for each cycle containing a split branch does not change and therefore all the data channels affected still can be filled completely and (ii) the outsourcing of multiple receive and a send operation from p to a dedicated process can only decrease, not increase, the possibilities for an (unwanted) blocking.

Transformation 6 (Merging)

Prerequisites: Process p with inplaceable behaviour on input set C_i and output $\{c_o\}$

Annotations: $\forall c_i \in C_i,\ \text{own}(c_i) := 1$

For this transformation, all the considerations from Transformation 5 apply analogously.

Transformation 7 (Duplicating access tokens)

Prerequisites:
- Process p with regular behaviour on input set $\{\}$ and output set $C \cup \{c^*\}$
- The channels in C just receive copies of the data going to c^*

Annotations: $\forall c \in C \cup \{c^*\},\ \text{own}(c) := \omega,$ where $\omega = \frac{1}{|C|+1}$ or, if the access tokens sent to c^* already come from an input channel c_{orig}, $\omega = \frac{\text{own}(c_{orig})}{|C|+1}$.

Transformation correctness for the data channels is trivial here, since no write operation is performed. By annotating the output channels with a "non-ownership", it is ensured that no split or in-place modification transformation is performed on one of the ouput channels. On the side of the recycling channels, all the considerations from Transformation 5 apply.

For the simplification transformations, all prerequisites and annotations can easily be deduced from their previous descriptions. Also their correctness is easy to see and therefore not further discussed here.

2.3.8 Applying DMS to the Ultrasound Algorithm

Having theoretically discussed the transformation of a classic KPN to a KPN using DMS in the previous section, we will now show how these transformations are applied to a given KPN. We choose to use the ultrasound image reconstruction algorithm explained in Section 2.3.3 as an example for a multimedia application which has a high demand of computation power and handles large amounts of data.

Starting point is a slight variation from the KPN shown in Figure 2.5 on page 49. To reduce the large number of processes in the application, the index extraction processes and the apodisation multiplication processes have already been merged together to beamforming processes. Due to the moderately complex structure of the network, all channels can be limited to a capacity of one token, where a token usually is a vector or a matrix. The different transformation steps which are performed are going to be explained below.

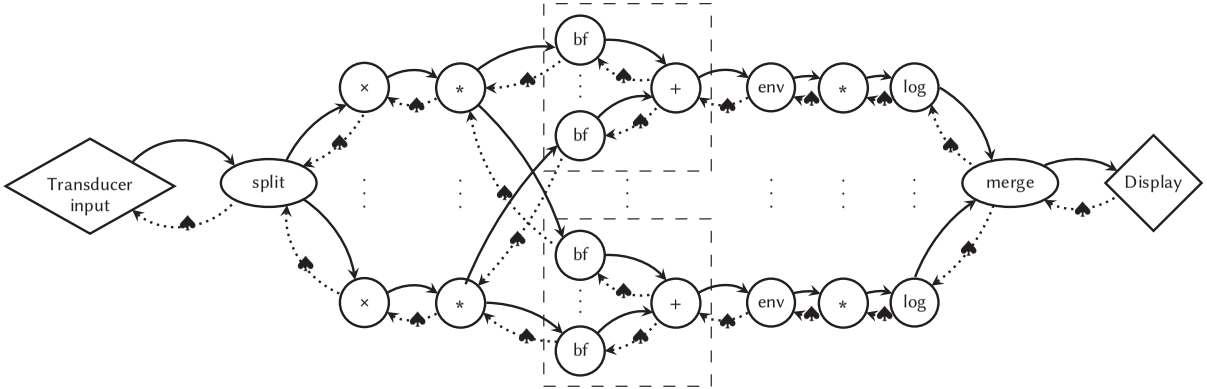


Figure 2.11: Implementation of the ultrasound image reconstruction algorithm after basic DMS transformations. The data precalculation processes have been left out for reasons of clarity.

In a first step, all channels are transformed to DMS channels according to Transformation 1. To all channels drawn in horizontal direction in Figure 2.5, Transformation 2 is applied, i.e. recycling channels are added. The other channels just transport the precalculated working data to the processes that use it. These processes will however keep the data forever, thus rendering recycling channels pointless. Wherever necessary or advantageous, the processes are transformed such that they permit simultaneous access to certain memory blocks (Transformation 3). Again, due to the moderate complexity of the network structure, the additional initial tokens suggested in this rule are not necessary here. Figure 2.11 shows the process network after these transformation steps.

In the following, it is described how the optimisation transformations can be applied. This is essentially done traversing the process network from left to right, although other transformation sequences are also possible.

For the `split` process, both splitting and token duplication are an option. We thus postpone the decision here. The element-wise multiplication processes next to it (attenuation compensation) lend themselves to applying an in-place editing transformation (Transformation 4). During this transformation, the channels coming from `split` are annotated with an ownership of one. With this annotation, Transformation 7 (token duplication) can no longer be applied to the `split` process, so this process is transformed according to Transformation 5 (splitting). This transformation requires a new merge process to be put in place which takes all the recycling channels coming from the high-pass filter processes (the first convolution processes from the left), merges them back together and then sends a token linked to the full memory block back to the transducer input process for later use.

The high-pass filter processes have a hybrid functionality: Each one convolves the incoming vector with a high-pass kernel. The convolution is done such that the resulting vector has the same size as the incoming one. Its implementation allows the convolution to be carried out in-place. On the other hand, one copy of the resulting vector is sent to a beamforming process in each beamforming block. We therefore apply Transformation 4 (in-place editing) first for one of the output channels. Then, we apply Transformation 7, access token duplication, for this output channel together with the other output channels. The procedure will be such that the high-pass process obtains an access token from the attenuation compensation process, convolves the data in-place and then duplicates the token, sending one duplicate to each out-going channel. This also necessitates a new process which collects all the duplicate tokens coming back through recycling channels from the different beamforming processes. It will then release these sub-block access tokens except for one, which is sent further to the previously generated merging process.

For the beamforming processes, in-place editing is not an option, since the access tokens they receive are duplicates. This is reflected by the fact that the prerequis-

ites of Transformation 4 are not met, the input channels being annotated with an ownership of a fraction of one.

The following summation processes also fulfil the requirements for in-place editing: From the vectors obtained through the input channels, it is possible to take one out and then add all the others to it. Thus, the operation between the input channel that provides this vector and the output channel is inplaceable, which makes it possible to apply the in-place editing transformation to the summation processes considering one of the input channels and the output channel. The other input channels remain untouched.

For the following processes (envelope detection, low-pass filter and logarithm), in-place editing can again be applied. The `merge` process can be optimised using Transformation 6 (merging). For this, a new splitting process is created, which takes the full blocks recycled from the `display` process and splits them again for reuse at one beamforming process in each beamforming block.

Finally, the processes generating the initial working data are transformed according to Transformation 7 (duplication). All the data generated by them is thus no longer copied but instead only the access tokens are duplicated.

The resulting KPN after applying the clean-up transformations is shown (without the initialisation processes) in Figure 2.12. Note that the number of initial access tokens has decreased; this, however, is just due to the merging of smaller vector to bigger matrix tokens. The amount of memory linked to these initial tokens is still the same.

2.3.9 Implementation in DAL

In the previous sections, DMS has been theoretically specified and it was shown abstractly how a given KPN can be transformed to a KPN using DMS. It has, however, not been explained yet how DMS can be implemented on a target architecture. In particular, the notion of an access token was only introduced as an abstract concept. This section will show how the ultrasound image reconstruction algorithm was implemented as a C-based program using the Distributed Application Layer (DAL) framework [Sch+12].

DAL is a programming framework which allows the user to specify a KPN and then translates this definition to parallel C code. The specification of a KPN application in DAL consists of two parts. In the first part, one specifies as C code the behaviour of a set of processes with input and output ports. Sending and receiving data works through these ports, using special `read` and `write` functions. The second part is an XML specification of how many copies of these processes exist and how they are connected through channels. There are different back-ends producing

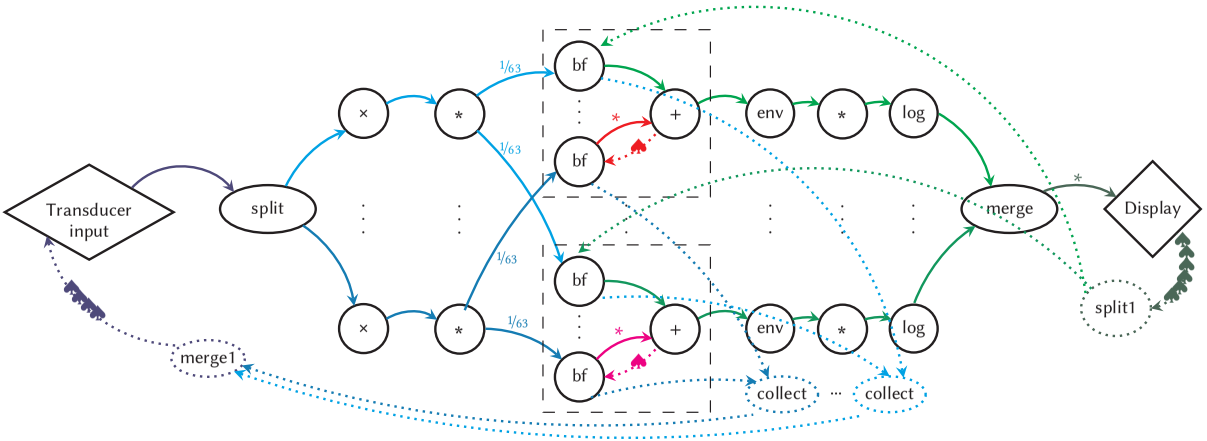


Figure 2.12: Implementation of the ultrasound image reconstruction algorithm after DMS transformations and optimisations. Colours have been used to mark the individual token cycles. Splits are illustrated by using lighter colours of the same hue. Next to each data channel, its ownership annotation is shown if it is not equal to one.

native C code for different target platform types; in our case we use a back-end creating POSIX threads since it provides a shared memory model.

The implementation of the DMS mechanisms can contain arbitrarily many safety checks. For instance, one could add a thread-safe reference counting mechanism to each memory block for keeping track of its usage and deallocating it when necessary. One could also store privilege information in the access token to inform a process whether it is allowed to write to the memory block linked to it. Another possibility would be to use the reference counter to dynamically check if a write access is allowed and, if not, block until this is the case.

Our approach concentrates rather on performance than on runtime assertions. As it has been shown in the previous sections, one can formally ensure that the code one creates is correct by construction. If the programmer strictly follows the rules and mechanisms of DMS, no global runtime checks are necessary.

We therefore implement access tokens as simple pointers. We use `malloc` and `free` for allocating and deallocating memory blocks and pointer arithmetic for splitting and merging. The sending and receiving of tokens is done by using the DAL `read` and `write` functions on the pointer itself, sending it over the channel as one would send a normal integer.

DAL allows to specify an initialisation and a clean-up function for each process. We use the former for creating the initial tokens on the channels and the latter for deallocating the memory. As the clean-up function is only called once the whole process network has stopped executing, every process can just store a reference to the memory blocks it allocated during initialisation an then deallocate those during clean-up.

2.3.10 Experimental results

The previous sections have shown many optimisation possibilities that are provided by DMS. Now we examine if these theoretical advantages translate to actual performance improvements with the ultrasound image reconstruction algorithm.

To this end, we execute the algorithm on an Intel Xeon Phi 5110P accelerator running a Linux kernel (version 2.6.38.8). This accelerator has 60 processor cores, each running at a clock frequency of 1053 MHz. Each core has four instruction decoding pipelines, which allows for a better utilisation of the ALU and for an overhead-free context switch between four threads per core. The cores are linked by a token ring communication infrastructure to each other and to the memory, which has a total capacity of 8 GB. They can not directly communicate; data exchange is done exclusively through memory. However, each processor has a 32 KB L1 data cache and a 512 KB L2 cache. A complex hardware-implemented cache synchronisation mechanism allows data exchange directly through the caches without accessing

the memory. The code is compiled using the Intel compiler ICC, version 14.0.1 with optimisation level 2.

Two implementations of the ultrasound algorithm are tested. One is the configuration discussed earlier. The second implementation is obtained by aggressively merging processes in the original KPN before translating it to DMS. In particular, all the beamforming and apodisation processes for one output image column (all the processes in a dashed rectangle in Figure 2.5 on page 49) are merged to one single process. The transducer samples are obtained from pre-recorded data loaded into memory during program initialisation. The configuration is for 63 transducers and 2048 12 bit samples (stored as 16 bit integers) per transducer. This gives a process count of roughly 4000 for the first implementation and 200 for the second implementation.

The 4000 thread implementation is tested using dynamic mapping (operating system decides on binding and scheduling of the processes at runtime) with two configurations, namely classic KPN channels and DMS channels.

The 200 thread implementation is also tested using static mapping (binding of processes is decided at design-time according to load-balancing considerations, scheduling is done by hardware, since there are more instruction pipelines than threads). Furthermore, windowed FIFO channels are tested as a third option for channel implementations.

The performance of the different configurations is measured in the form of the amortised average image reconstruction framerate. To this end, the execution time of the program is measured for 50 frames and for 250 frames, 30 times each. The values obtained are then fitted using the least squares method on a linear model (time vs. number of frames).

In general, the channel capacities have a considerable influence on the performance of an application. While too low capacities restrict the parallelism and the scheduling options for a KPN, too high capacities will result in higher memory footprints and thus a worse performance of the caches. We have therefore tested different values for the capacity of all the channels in the range of 1 to 250 tokens per channel. For the DMS implementation, we have also varied the number of initial access tokens in all the token cycles in the range of 1 to 250. In Table 2.4 on the next page, we give two configurations for each implementation: The configuration achieving the best framerate and, if it exists, the configuration coming next to this framerate with a smaller memory footprint. The memory footprint is also given for these configurations. Note that in the case of classic or windowed FIFOs, it depends mainly on the channel capacities whereas for DMS implementations, it is the number of initial tokens that counts.

The numbers show that: (i) the framerate with DMS is significantly higher than with windowed FIFOs and classic channels, (ii) the memory footprint with DMS

Table 2.4: Experimental results for the ultrasound image reconstruction algorithm. For each method, the configuration with the highest throughput is listed here as well as, in some cases, a second configuration coming next to this throughput with a smaller memory footprint. *Init* denotes the number of initial tokens (i.e. memory blocks) on recycling channels. *Cap* denotes the capacity (in tokens) of the channels (except for the initialisation channels, which only hold one token). *Mem* denotes the total amount of memory used for all channels and the initial tokens. *Rate* denotes the amortised average reconstruction framerate achieved.

Threads	Mapping	Method	Init	Cap	Mem (MB)	Rate (s^{-1})
4000	dynamic	classic		12	397	65.3
		classic		5	212	61.1
		DMS	4	1	47	121.5
200	dynamic	classic		12	254	147.6
		windowed		2	109	157.7
		DMS	30	7	32	187.3
		DMS	3	2	5	180.0
200	static	classic		50	805	154.3
		classic		3	124	151.0
		windowed		2	109	161.7
		DMS	6	14	8	192.1
		DMS	4	2	6	191.8

is drastically lower than with windowed FIFOs and classic channels and (iii) using DMS, it is possible to achieve good performance already with small amounts of memory.

Especially from the fact that no special optimisations for the target platform were applied, it can be concluded that the effort of transforming a KPN to use DMS pays off in terms of performance and memory footprint.

2.3.11 Summary

In this section, we have presented deterministic memory sharing, a concept for sharing memory blocks between different processes in a KPN. We have shown how the concept of access tokens ensures that the determinacy of KPNs still persists even when multiple processes access the same memory regions. Rules have been set up which allow to transform a traditional KPN application such as to make use of DMS. A first set of rules transforms the channels into DMS channels. A second set of rules

optimise individual processes. They can be applied locally, i.e. to one process and the channels connected to it without having to consider the rest of the process network.

An ultrasound image reconstruction algorithm was explained and a classic KPN implementation of it was presented. This KPN was then transformed according to the rules mentioned above. Experiments on the Intel Xeon Phi accelerator show that even without any special adaptations, a significant speed-up can be achieved while immensely reducing the memory footprint of the application.

2.4 Discussion

The last two sections demonstrated the influence of implementation models on application performance. In both cases, a given implementation model was replaced with a different one, yielding considerable improvements in execution time and throughput. As a result, one could argue that the performance goals formulated in the beginning of this thesis have already been reached.

Unfortunately, these performance gains come at a price. In fact, both examples have shown that one cannot simply exchange the implementation model in a model set without greatly affecting the entire rest of the application programming procedure.

When converting CAL actors to Kahn processes, the problem arises how to handle KPN incompatible actors. Since CAL allows the programmer to specify them, the two unsatisfying options are either not to support the full spectrum of the language or to mix different implementation models. The first approach would inevitably raise the question why to use CAL as a specification model in the first place. The second approach would introduce manifold implementational intricacies as well as possible losses in analysability. Together this might easily introduce new implementation overhead, which might even neutralize the performance gains from the KPN implementation model.

Introducing deterministic memory sharing in a KPN worked well in the example, but was done manually. In other words, all the additional programming primitives and rules implicitly became part of the specification model. This resulted in a high amount of programming work (remember the number of 4000 processes in one case), including many of the typical multi-core programming issues like error-proneness and laborious debugging.

At the same time, the given example was implemented on the Intel Xeon Phi platform, which only featured one big shared memory. This is still rather straightforward as compared to other possible platforms, which could consist of multiple clusters and different memory layers. On such platforms, each pair of processors may have a different preferred way of communication between them, e.g. different

shared memory banks or a network-on-chip. In the extreme case, each change in the process binding can entail a re-evaluation of the entire DMS configuration and a complete relaunch of the DMS optimisation process. This means no less than mixing up specification and design space exploration procedures, rendering the entire application optimisation process a fully manual, time-consuming endeavour.

Since Kahn processes as well as Kahn process networks are Turing-complete, it is not possible in general to automate DMS optimisations. For instance, it is undecidable in general whether a Kahn process shows regular behaviour with respect to certain channels or if channel access reordering requires additional initial tokens. Even if such optimisations can be automated in simple cases, the question remains how they could be combined such as to obtain a *globally optimised* DMS network and if a set of simple, automated optimisations is sufficient to attain this goal.

To summarise, in both cases the replacement of the implementation model was either incomplete or it required a change in the specification model. While there are also counter-examples to this (e.g., windowed FIFOs [HGT07] can improve performance even with the traditional KPN specification model), it is safe to say that especially in the case of fundamental changes to an implementation model, these changes are likely to also have an impact on the related specification and optimisation models. In short, all models in a model set must be conceived together.

The approaches presented in this chapter sections yielded excellent results in terms of performance, in addition to other valuable findings. However, for these approaches to be useful in the real-world programming domain, they need to be embedded in an overall approach including adequate specification and optimisation models. Such an overall approach will be the subject of the next chapter.

3

The Ladybirds specification model

Specification models can be regarded as the interface between programmer and compiler. As such, they need to "connect" well to both sides: To the programmer with his view of the target application and to the compiler with its optimisation model. A specification model which is unintuitive to the programmer can quickly lead to clumsy code or may not be used altogether. A specification model which does not translate well to an adequate optimisation model for the targeted goal is unlikely to yield an efficient implementation of a given specification.

This chapter will discuss the challenge of finding a good specification model for achieving efficient data exchange in multi-core architectures. At first, the prerequisites for such a model will be derived from the results of the previous chapter as well as from general considerations, including those mentioned above. A short survey on existing solutions for programming multi-cores will substantiate the need for a new specification model. Eventually, a new model set (consisting of specification, optimisation and implementation model) called *Ladybirds* will be introduced: Its specification model will be presented along with the corresponding optimisation and implementation models. It will be explained how the models translate into each other and to what extent they fulfil the prerequisites mentioned previously. Finally, two different implementation models targeted for use cases outside the multicore domain will demonstrate the wide range of application of the Ladybirds specification model.

3.1 Requirements for a model set

As the last chapter showed, the models in a model set cannot be considered independently. A specification model must be devised already with optimisation and implementation models in mind. Consequently, when assembling a list of requirements for a specification model, the requirements for optimisation and implementation model must be included as well.

In the field of **implementation models**, important requirements have already been discussed in this thesis. As it was shown in Section 2.3, an efficient implementation model needs to allow **efficient data exchange** over shared memory. Costly emulation of message passing through coping must be avoided. In particular, the functionality of multiple cores **sharing the same data** must be implemented natively on shared memory platforms, i.e. cores must be able to concurrently access the same memory regions. Along the same lines, **in-place data modifications** should be supported, especially for cases in which only a small part of a data block is to be modified. The case of multiple cores modifying data in the same memory region is of course more complicated, as tight synchronisation may be necessary to avoid uncoordinated reading and writing. However, there should be a possibility of **dividing memory blocks** into smaller regions which can then be modified in parallel by different cores. This is a simple feature, but fundamental for exploiting data parallelism.

The **amount of memory** that is allocated should be **kept low** as well as the runtime overhead for gaining access to a new buffer (unlike, e.g., with typical dynamic memory allocation algorithms). Section 2.3 showed the high impact an allocation scheme can have on memory footprint as well as execution time. It is important to note that in many cases the former has a direct influence on the latter: Whenever caches are involved, using less memory reduces the chances of cache misses. On a platform with on-chip memory, a lower memory footprint typically means that more data can be stored in fast memories near to the core. In such cases, speed-ups of orders of magnitudes can be expected for the concerned memory accesses.

Another source of inefficiency can be induced by the implementation model. As Section 2.2 showed, the **overhead for task management** should be kept low and **scheduling different tasks** should be kept efficient by having a low-overhead mechanism to determine whether a task can execute or not.

More abstract and yet essential requirements have to be imposed on **optimisation models**. First and foremost, they must pave the way for an **automated optimisation procedure**. In particular, **easy extraction of concurrency** in a program is vital for unobstructed parallelisation. Secondly, with non-uniform memory access (NUMA) architectures and memory hierarchies gaining popularity in multi-core platforms, a good optimisation model must support these technologies in the sense

that it needs to take account of **data placement on different memory modules**, thereby providing a starting point for optimisation of memory and data exchange efficiency. Finally, today's large and still growing variety in architectures and platform types, especially with respect to memory, requires **universal and flexible optimisation models** in order to attain a desirable degree of portability. This is not only relevant for application code that is to be used on different platforms, but just as well for optimisation frameworks themselves: While it is a standard procedure to provide with a new processor architecture a suitable compilation toolchain, automatic parallelisation and efficient parallel implementation of programs typically require intensive research for each new platform. The goal should be to have a set of standard optimisation techniques that can be adapted to a new platform in the same way as single-threaded compiler backends can be adapted to new processors.

For **specification models**, a number of features are desirable. Apart from a **high expressiveness**, which allows a large range of applications to be specified, this includes features aiming to ease the programmer's job. A good specification model should present itself to the programmer in an **intuitive** way, typically in form of an intuitive programming language. Along the same lines, it is helpful if this programming language is **close to existing languages**, not only to allow code reuse, but more importantly to take advantage of abilities and competences which programmers have already acquired.

When it comes to multi-core architectures, concurrency plays a big role in specification models. One particularly important form of concurrency is data parallelism, for a number of reasons. Data parallelism in an application is easy to recognise. It also scales well; for instance when an image is divided into multiple blocks, their number can be chosen freely (within certain bounds). Moreover, data parallelism allows to maximise processor usage, since multiple cores typically take the same time for executing the same code. This helps to avoid processor inactivity due to dependency-induced waiting times, even on different platforms. As a consequence, a program specification model for multi-core architectures should provide an **intuitive way of specifying data-parallel execution**.

Finally, **debugging** parallel programs is particularly intricate. With high amounts of concurrency, it is difficult to isolate the origin of a faulty program state and thus to locate a defect in program code. A specification model providing an easy method to find programming mistakes, possibly even through automatic analysis or in a simple test environment independent of the target platform or hardware simulators, could save the programmer tedious and time-consuming debugging operations.

This long list of requirements shows the difficulty in finding a well-suited specification model that allows memory and data exchange efficiency optimisations in multi-core architectures. The situation is complicated further by the fact that some of the criteria mentioned above actually conflict with each other, for instance

high expressiveness and easy extraction of concurrency. A good specification model thus requires not only smart solutions, but also tradeoffs and sometimes painful decisions. What this means in practice and how the problems are approached in different existing models will be the topic of the next section.

3.2 Existing Concepts for Parallel Programming

A large number of solutions exists for programming multi-core systems. This section will present the most important ones and discuss them with respect to the requirements listed in the last section, focusing in particular on their suitability for data exchange efficiency.

Manual thread synchronisation and data management is still the most widespread approach. Typical examples are POSIX threads (pthreads) or hardware-specific software stacks, programming language in-builts (e.g. the C++ thread support library), frameworks like OpenCL [Mun09] and Grand Central Dispatch [SF12] or libraries like Open MPI [Gab+04]. The problem with these approaches is that manual code parallelisation is an intricate and error-prone task: There are many pitfalls like deadlocks, unwanted sequentialisation, inefficient organisation of data transfer etc. Moreover, debugging multi-core platforms is tedious. All this leads to a long development process.

Another problem all these approaches have in common is their lack of portability: A code written for one platform using these methods either cannot be executed at all on a dissimilar platform or, like in the case of OpenCL or programming language directives, only with serious performance losses (not even considering the different memory capacities on target platforms). It is not uncommon that a change in the target platform entails a complete re-design of an algorithm implementation.

Compiler based parallelisation as introduced by OpenMP [DM98] is also a popular option. The idea of this mechanism is that the programmer gives parallelisation directives, e.g. as to which loops are to be parallelised and how. The drawback of this mechanism is that only a small number of parallelism types (like *for* loops) can be optimised automatically. For more complex concurrency structures, OpenMP directives require the user to manually orchestrate concurrency and synchronisation, leaving him with all the related problems as described above[1].

Certain **domain-specific libraries or frameworks** exist, in particular for image processing (e.g. OpenCV [Bra00] or OpenVX [GR14]). These solutions provide

[1] OpenMP supports the specification of tasks with explicit inputs and outputs, from which it can derive dependencies. Since, however, for each pair of input and output specifications in the program, their memory areas must be either fully identical or fully disjoint and since dependencies are not verified by the compiler, this feature is hardly more helpful than manual specification of execution dependencies between tasks.

parallel implementations of commonly used algorithms, which can then be combined to a program. This works well as long as the application only uses these pre-implemented algorithms *and* is to be run on a supported platform for which these implementations already exist. As soon as one of these conditions is not met, the programmer has to resort, partially or entirely, to manual synchronisation and data management (see above).

Even more purpose-built specification models exist for databases and big data applications, e.g. SQL [DD89] or MapReduce [DG08]. These models allow highly efficient processing of large amounts of data, however solely for a small set of well-defined operations. Low-power or embedded systems are not targeted by these models.

Functional programming languages have been proposed for parallel program specification; SISAL [FCO90] is an example of a language that supports parallel and communication-aware programming and that was used in high-performance clusters. Single-Assignment C [Sch94] is a newer, C-oriented functional language that is subject to steady research in the multicore parallelisation domain. Unfortunately, the paradigm of functional programming substantially differs from conventional programming approaches taken for multicore architectures. This means that not only all existing code would have to be entirely rewritten but also most programmers' habits and competences (in fact, even their formation) could only restrictedly be applied to the functional programming methods. Furthermore, functional programming does not correspond to the internal working principle of processors, which is purely instructional. Translating functional program specifications to instructional code can induce significant inefficiencies. In particular, automatic parallelisation monolithic code — functional or single-threaded instructional — is hard to achieve and not reliably effective with state-of-the-art compilation techniques.

In the domain of **high-performance computing**, multiple parallel programming languages have evolved. Cilk [Blu+96] provides an easy syntax for instantiating concurrent tasks and a memory model allowing task execution on distributed memory. Data synchronisation is implemented based on a tagging system and thus requires unconditional copying of data, which may be difficult on embedded systems with low memory capacities. Also, task coordination and guaranteeing data coherency between the tasks are left to the programmer. StarPU [Aug+11] goes one step further and allows the specification of tasks with explicit inputs and outputs. Like with OpenMP tasks, task arguments can be specified with read, write or read-write access. Data dependencies are automatically calculated and result in a correct and thread-safe task execution order. All these systems have in common that tasks are generated dynamically at runtime, such that a compile-time analysis is not possible. Task mapping and scheduling, buffer allocation and data transfers are

managed *ad hoc*, which may cause an unbearable runtime overhead on embedded and particularly on low-power systems. Moreover, runtime resource management makes it impossible in non-trivial cases to give guarantees for execution times and even for execution as such.

Finally, **process networks** are a popular approach in science, for instance representing programs as KPNs, SDF graphs or using CAL, as Chapter 2 showed. The big advantage of this type of representation is that parallelism is made explicit and can thus easily be exploited. The downside, however, is that these models implicitly assume message passing as the only method of data exchange, which is suboptimal particularly on shared memory systems. As discussed previously, this can severely impact application performance.

With all the individual drawbacks which were discussed for all of the solutions above, one characteristic is common to all of them. Some of these solutions result in efficient code which is however not portable to all platforms; others result in portable code that is not efficient. Finding a solution that leads to a parallel program implementation that is portable *and* efficient, in particular with respect to data placement and exchange, constitutes an important task that remains to be solved.

Closing this gap will require a specification model that accounts for the issues and requirements discussed so far. The remainder of this chapter will present a specification model that was designed to go hand in hand with a later optimisation and implementation model. It is based on the ambition of finding an easy way to specify an application such that it can readily be parallelised and optimised for data storage and exchange. The idea behind this is that such a specification model allows to concentrate on the related optimisation problems without having to think about or to rely on complex code analysis techniques.

3.3 Specification Model

The last sections showed manifold problems to solve and requirements to meet. In the course of this thesis, an experimental model set called *Ladybirds* evolved with the aim of addressing these issues. It is a first prototype based on the idea of limiting the complexity in code analysis such that code parallelisation and optimisation are facilitated. Ladybirds consists of a specification model, an optimisation model and an implementation model, the former shall be discussed in this section.

In a nutshell, a Ladybirds program consists of a set of atomic *tasks* with clearly defined inputs and outputs. A task carries out a well-defined operation, which is given by a *kernel*. *Metakernels* can be used to compose the functionality of kernels to an analysable and parallelisable program. The Ladybirds specification model is

Listing 3.1: Example for a Ladybirds kernel. The kernel describes a 1-d convolution of an input signal with a filter kernel; the input signal and the filter kernel are provided as arguments in the form of floating point arrays, the output signal is written to another floating point array which is returned.

```
kernel(Convolution)(in  float signal[128],
                    in  float filter[5],
                    out float response[124])
{
    for(int i = 0; i < 124; ++i)
    {
        float f = 0;
        for(int n = 0; n < 5; ++n)
            f += signal[i+n] * filter[n];
        response[i] = f;
    }
}
```

Listing 3.2: A Ladybirds kernel sorting an array in-place

```
kernel(Quicksort)(inout float array[128]) {…}
```

represented by a C-based programming language called *Ladybirds C*, the concepts of which shall be introduced in the following.

3.3.1 Kernels

Listing 3.1 shows an example for a kernel. Essentially a kernel is a conventional C function, however with fixed and explicitly specified inputs and outputs called *packets*. Packets consist of fixed size arrays with a specified type of access. `in` means that the packet is only read but not modified, `out` means that the contents of the packet will be provided as an output by the kernel. A kernel is allowed to read from packets declared as `out`, but the contents are uninitialised at the start of its execution.

In certain cases, the clear categorisation of packets into input and output is unsuitable. Therefore, Ladybirds C also allows the specification of combined input/output (`inout`) packets for algorithms that modify data in-place (see Listing 3.2). Packets declared as `inout` contain input data for the kernel before its execution; the kernel can then arbitrarily modify them, returning the modified version. Note that

`inout` is meant for algorithms that *need to* modify data in-place, such as many sorting algorithms or FFT implementations.

This raises the question of how to specify algorithms which could be implemented such as to modify data in-place or such as to return the output in a different packet than the input. Take as an example a subtraction of two vectors. It could be regarded as an operation with two input packets and one output packet. However, it could just as well implemented by writing the result back to one of the input packets, saving memory with this in-place modification. Which variant is best-suited for the purpose does not depend on the kernel itself but only on its later use in the program (examples will be given in the following sections); it would thus be unsatisfying to have to take this decision already with the kernel declaration. The Ladybirds approach to this are *buddies*. The idea is to independently specify an `in` and an `out` packet and to then declare them as buddies, which means that the semantics of the kernel would be the same if the two packets were just aliases of the same `inout` packet. A buddy declaration allows the optimiser to later transform the kernel accordingly and *merge* the buddy packets to one, enabling in-place modification. A packet may be involved in multiple buddy relationships; the latter are not transitive, however, and each packet may only be merged with one other packet at the same time. Ladybirds C code for the vector subtraction example is given in Listing 3.3.

For certain kernels, e.g. such with generic or often-used functionality, it is desirable to be able adapt them, for instance to different packet sizes. For these cases, Ladybirds C supports *kernel parameters*. Listing 3.4 shows an example of a parametrised kernel.

To summarise, a kernel is a function consisting of a C code body and a special declaration. It must have the following properties.

- It may be parametrised; for each parameter value that is chosen in the program, all of the following properties must be fulfilled.
- The body must describe a finite amount of calculations.
- Its inputs and outputs must be clearly specified; they consist of `in`, `out` and/or `inout` packets. `in` and `inout` packets, upon execution of the kernel, contain values constituting the inputs of the kernel. The elements of `out` and `inout` packets can be modified during execution of the kernel. All packets can be read from during kernel execution. At the end of kernel execution, all `out` packets must have been assigned values, which, together with the `inout` packet values, constitute the outputs of the kernel.
- Pairs of one `in` and one `out` packet of the same data type may be declared as buddy interfaces. For each such pair, it must hold that the kernel output is

Listing 3.3: A Ladybirds kernel subtracting two vectors. The output packet `res` is marked as a buddy of each of the input packets, i.e., `a` and `res` are buddies and `b` and `res` are buddies. This tells the optimiser that it may decide to store back the output `res` directly to `a` or to `b`.

```
kernel(VecSub)(in float a[128], in float b[128],
               out float res[128] buddy(a) buddy(b))
{
   for(int i = 0; i < 128; ++i) res[i] = a[i] - b[i];
}
```

Listing 3.4: A parametrised kernel. It returns a packet in which all array elements have a value of zero.

```
kernel(Zero)(param int size, out int res[size])
   { for(int i = 0; i < size; ++i) res[i] = 0; }
```

the same if the elements at each common index of both packets are stored in the same location. A packet can be part of multiple buddy relations.

- It must not perform pointer arithmetic on packets or packet elements, since the results are undefined.
- A kernel must not have any side effects. This directly relates to the property of clearly defined inputs and outputs. A consequence of this restriction is that kernels must be stateless in the sense that they must not preserve a state outside their execution. For the programmer, this means that no static and no global variables must be used inside kernels.

Formally, a packet can be defined as a multi-dimensional data vector of fixed size and with a given data type. A kernel can be interpreted as a (parametrisable) algorithmic description of how to transform a tuple P_i of input packets to a tuple P_o of output packets. With the sets P_i^* and P_o^* of all elements contained in P_i and P_o, respectively, there may exist a set $J^{io} \subseteq P_i^* \times P_o^*$ of pairs of packets that have to be placed in the same memory locations (specified as `inout`) and a set $J^{bud} \subseteq P_i^* \times P_o^*$ of pairs of packets that *can, but do not have to* be placed in the same memory location (specified as buddies). No packet may appear in both J^{io} and J^{bud}.

The Ladybirds kernel concept addresses many requirements from Section 3.1:

- Since the kernel description is based on C and can contain any arbitrary C code, a high **compatibility** to existing code but also to programmers' habits is achieved.

- The explicit declaration of kernel inputs and outputs guarantees a high **analysability** of the code and in particular of its memory usage patterns. The optimiser knows which task needs which data and can allocate memory accordingly. The specification of input and output being a classic concept from computer science, it is not alien to the programmers and easily integrated in programming practice. At the same time, compared to automatically analysing the C code inside the kernels, manual specification drastically improves accuracy and reliability of this information while increasing the programmer's freedom inside the kernel.
- As compared, for instance, to manual parallelisation or to KPN specifications, communication is not a part of the Ladybirds code. A kernel just declares inputs and outputs in its header, but does not contain any communication directives (like sending or receiving data) in its body. This saves the programmer from large amounts of **repetitive code** which is often tedious to write and a major source of errors. Additionally, it greatly enhances the **portability** of the code, since all the instructions for data storage and exchange, which change from implementation to implementation, are not a part of the kernel and can be auto-generated by an optimisation framework without having to touch the kernel code.
- The packet types (in, out etc.) directly relate to some of the previously mentioned techniques of achieving **memory efficiency**.
  - `inout` packets allow **in-place modification** of data.
  - `in` packets allow **memory sharing** between multiple tasks, since these packets are known to remain unmodified by the tasks.

Clearly, these advantages come at a price; just from the kernel specifications, one can already see that Ladybirds C is more formal and restrictive than C. This point will be discussed in greater detail after the next section, which shows how kernels can be composed to a program specification that allows optimisation for data exchange efficiency.

3.3.2 Metakernels

Ladybirds kernels provide basic functionality and can be implemented using arbitrary instructional specifications. This allows for high efficiency at instruction level, but severely limits global code analysability inside their bodies. While at function level such an approach is convenient, at inter-thread level its converse is required: Individual inefficiencies at instruction level may be tolerated as long as the coordination of the threads (synchronisation, data exchange) is efficient. For an automated

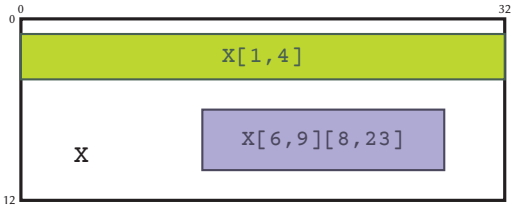


Figure 3.1: Graphic illustration of the sub-packet operator.
For a 12 × 32 packet `X`, the regions covered by sub-packets obtained using the sub-packet operator `[]` are shown.

optimisation of this efficiency, it thus makes sense to restrict the programmer's freedom for the benefit of high analysability.

This idea is taken to extremes by Ladybirds *metakernels*. Metakernels do not provide own functionality but merely compose the functionality provided by other kernels. To the outside, metakernels provide the same interface as kernels, and both can be used interchangeably in Ladybirds C. Metakernels can have `in`, `out` and `inout` packets like kernels; buddy declarations are allowed but have no effect, since these relations will be automatically detected in later optimisation procedures. The definition of a metakernel only consists of an ordered list of references to other (meta-)kernels.

Listing 3.5 on the next page shows a metakernel. Its body contains declarations of temporary packets and calls to other (meta-)kernels. Packet declarations are not to be understood as concrete instructions for immediate memory allocation, they merely announce the existence of a packet with particular data type and dimensions. Like with a variable in a traditional compiler, an optimiser later decides whether and where to store it. For instance, the declaration of a packet that is never used will simply be ignored.

In kernel calls, entire packets are provided to the kernels for reading or writing (in conventional programming languages, this would be referred to as "by reference"). These calls are meant to be executed in consecutive order, like in a sequential program. A metakernel is not allowed to call itself, directly or through other metakernels, since this would lead to infinite recursion.

Listing 3.6 on page 87 shows an optimised implementation of the metakernel from Listing 3.5. The hard-coded image dimensions have been replaced with constants for better readability and adaptability, but this does not alter the semantics of the program. More importantly, parallelism was doubled by splitting up each operation into two new operations working on sub-blocks of the image. For this purpose, Ladybirds C introduces the *sub-packet* operator `[`l`,`u`]`, which, applied to a packet, provides a sub-packet of it, which, in its highest dimension, is restricted to indices l up to and including u. Multiple sub-packet operators can be concatenated to obtain a sub-packet that is restricted in multiple dimensions (see Figure 3.1). Packet dimen-

Listing 3.5: Ladybirds C specification of a (part of a) Canny edge detection filter. For an input image, it calculates pseudo partial derivatives in horizontal and vertical direction by 2-d convolutions with Sobel filters, thus obtaining a gradient given in Cartesian form. This gradient is then converted into a polar representation (steepness and direction) and the still blurred edges given by this gradient are sharpened by a non-maximum suppression.

```
kernel(SobelX)(param int height, param int width,
               in uchar Img[height][width],
               out char GradientX[height-2][width-2]) {…}
kernel(SobelY)(param int height, param int width,
               in uchar Img[height][width],
               out char GradientY[height-2][width-2]) {…}

kernel(EdgeGrad)(param int height, param int width,
                 in char GradientX[height][width],
                 in char GradientY[height][width],
                 out uchar Gradient[height][width]
                    buddy(GradientX) buddy(GradientY),
                 out uchar Direction[height][width]
                    buddy(GradientX) buddy(GradientY)) {…}

kernel(NonMaxSupp)(param int height, param int width,
                   inout uchar Gradient[height][width],
                   in uchar Direction[height][width]) {…}

metakernel(Canny)(in uchar Img[1038][1038],
                  out uchar Edges[1036][1036])
{
    char GradX[1036][1036];
    char GradtY[1036][1036];
    uchar Direction[1036][1036];

    SobelX(1038, 1038, Img, GradX);
    SobelY(1038, 1038, Img, GradY);
    EdgeGrad(1036, 1036, GradX, GradY, Edges, Direction);
    NonMaxSupp(1036, 1036, Edges, Direction);
}
```

sionality can also be reduced by sub-packet operations. E.g., if a kernel requires a packet of size 3 × 3, a sub-packet of a 3 × 3 × 3 packet `X` may be provided, such as `X[1]` or `X[0..2][1]`.

Listing 3.7 shows how this concept can be generalised using *generator variables* and a *generator loop*. Generator variables (declared as `genvar`) can be assigned any value and used in any operation. However, they are interpreted at compile time and then treated as a constant. Similarly, generator loops are fully unfolded and

Listing 3.6: Specification with more parallelism of the `Canny` metakernel of Listing 3.5. Each operation has been split into two operations working on the upper and the lower half of the image, respectively. Note that the input sub-packets passed to the Sobel kernels overlap by two pixel rows.

```
enum { ImgWd = 1038, ImgHt = 1038,
       EdgeWd = (ImgWd-2), EdgeHt = (ImgHt-2) };

metakernel(Canny)(in uchar Img[ImgHt][ImgWd],
                  out uchar Edges[EdgeHt][EdgeWd])
{
   char GradX[EdgeHt][EdgeWd];
   char GradY[EdgeHt][EdgeWd];
   uchar Direction[EdgeHt][EdgeWd];

   enum { blkht = EdgeHt/2 }; // height of each block

   SobelX(blkht+2, ImgWd, Img[0,blkht+1],     GradX[0,blkht-1]);
   SobelX(blkht+2, ImgWd, Img[blkht,ImgHt-1], GradX[blkht,EdgeHt-1]);
   SobelY(blkht+2, ImgWd, Img[0,blkht+1],     GradY[0,blkht-1]);
   SobelY(blkht+2, ImgWd, Img[blkht,ImgHt-1], GradY[blkht,EdgeHt-1]);

   EdgeGrad(blkht, EdgeWd, GradX[0,blkht-1], GradY[0,blkht-1],
            Edges[0,blkht-1], Direction[0,blkht-1]);
   EdgeGrad(blkht, EdgeWd, GradX[blkht,EdgeHt-1], GradY[blkht,EdgeHt-1],
            Edges[blkht,EdgeHt-1], Direction[blkht,EdgeHt-1]);

   NonMaxSupp(blkht, EdgeWd, Edges[0,blkht-1], Direction[0,blkht-1]);
   NonMaxSupp(blkht, EdgeWd, Edges[blkht,EdgeHt-1],
              Direction[blkht,EdgeHt-1]);
}
```

Listing 3.7: Extension of the parallelism in Listing 3.6. Only the operations in the body are shown. This time, the image is split into multiple horizontal stripes, and the Sobel filter is applied independently to each stripe. Setting `nstripes=2` would yield the same metakernel as in Listing 3.6.

```
   genvar int nstripes = 4; //change here for more/less concurrency
   genvar int blkht = EdgeHt/nstripes;

   for (genvar int y1 = 0; y1 < EdgeHt; y1 += blkht)
   {
       genvar int y2 = y1+blkht-1;
       SobelX(blkht+2, ImgWd, Img[y1,y2+2], GradX[y1,y2]);
       SobelY(blkht+2, ImgWd, Img[y1,y2+2], GradY[y1,y2]);

       EdgeGrad(blkht, EdgeWd, GradX[y1,y2], GradY[y1,y2],
                Edges[y1,y2], Direction[y1,y2]);
       NonMaxSupp(blkht, EdgeWd, Edges[y1,y2], Direction[y1,y2]);
   }
```

Listing 3.8: Top-level metakernel for the `Canny` program

```
metakernel(mainkernel)()
{
    uchar Image[ImgHt][ImgWd];
    uchar Edges[EdgeHt][EdgeWd];
    CaptureImage(Image);  // obtains a picture, e.g. from a camera
    Canny(Image, Edges);
    OutputImage(Edges);  // Shows the calculated edges, e.g. on a screen
}
```

interpreted at compile time, such that the resulting code is still a sequential list of kernel calls. They are thus meant as a helper tool for the programmer, but do not add functionality to the specification model.

The concept of metakernels leads the way to specifying entire Ladybirds applications. As already mentioned, metakernels can contain calls to other metakernels; this way, a call hierarchy develops. At the root of such a call hierarchy is one metakernel which serves as the entry point to a Ladybirds program. Listing 3.8 shows a typical example for such top-level metakernel, in this case for the `Canny` code from above.

In summary, a metakernel is a special kernel which has the following properties:

- Its inputs and outputs are clearly defined and consist of `in`, `out` and/or `inout` packets. The same properties as with kernels apply.
- Buddy relationships may be declared, but have no effect.
- The bodies of metakernels consist of only two types of statements, viz. temporary packet declarations and calls to other (meta-)kernels. All numeric values, i.e. array sizes, indices and parameter values have to be known at compile time. Expressions with generator variables and generator loops are evaluated at compile time; this evaluation has to result in statements of the types discussed previously.
- In (meta-)kernel calls, a value must be provided for each parameter and for each kernel argument, a packet (either one of the metakernel arguments or a temporarily declared one) or sub-packet thereof of the correct size must be provided.
- For each output packet of the metakernel, the element at each index of it must be part of at least one (meta-)kernel call such that its value is defined at the end of the metakernel execution.

Formally, a metakernel can be interpreted as a description of how to transform a tuple P_i of input packets to a tuple P_o of output packets. As with kernels, there may exist a set J^{io} of pairs of packets that have to be placed in the same memory locations (`inout` packets). The packet transformation is described as an ordered list of operations, where each operation consists of the execution of one (meta-)kernel on a combination of packets in P_i, sub-packets thereof, packets obtained in previous operations and sub-packets thereof.

With their C-like syntax, metakernels are **intuitive to understand** and to write for any programmer used to instructional programming. Furthermore, the sequential specification also allows for **easy debugging**. The sub-packet functionality together with the generator variables and loops allows for a comfortable and at the same time flexible specification of data parallelism in Ladybirds code. This makes it easy to **exploit data parallelism** as a major source of concurrency (see above) while increasing **portability** and **maintainability** of the code. In addition, the sub-packet operator has counterparts in other programming languages, for instance in Matlab, which enjoys great popularity exactly for that reason. Passing sub-parts of multi-dimensional arrays as arguments to functions is in fact a useful functionality, which in C, however, is cumbersome and needs advanced language experience to implement.

At the same time, dependencies between the operations are easily analysable and thus, **parallelism and data exchange** requirements can be **readily extracted**. As already mentioned, however, this analysability was achieved by restricting the set of allowed operations inside a metakernel. The next section will discuss the effects of this restriction in detail.

3.3.3 Expressiveness of Ladybirds C

Previously, the Ladybirds specification model and its specification language, Ladybirds C, have been presented. They have useful features and were designed to meet most of the requirements given in Section 3.1. However, there is also standard functionality which is not part of Ladybirds.

- Iterations (i.e. loops) cannot be expressed on meta-kernel level. Each operation in a metakernel is executed exactly once. This makes it impossible, for instance, to run a sensing program (acquiring data from a sensor, processing it, transmitting the results) in an infinite loop and profit from pipelining parallelism.

- Conditional execution of operations is not supported in a metakernel. For instance, semantics like not processing uninteresting data (e.g. a signal below a given threshold) or executing an operation a varying number of times, depending on the amount of received data, cannot be implemented at metakernel level.

- Input and output packet sizes are fixed and cannot be adjusted dynamically (e.g. when compressing data).

This limits the expressiveness of the Ladybirds specification model significantly, which is a conflict with the goals set out in Section 3.1.

On the other hand, the expressiveness of Ladybirds is sufficient for many real-world programs. As numerous experiments will show in Chapter 5, there is a high number of real-world applications that can be efficiently implemented using Ladybirds. Also, workarounds exist for emulating certain features:

- Even if no iterations are possible inside a metakernel, one can still repeatedly execute the entire program.

- Conditional execution can be controlled inside kernel definitions.

Furthermore, even when the amount of input or output data of an operation varies, one can still establish an upper bound, which then determines the packet sizes. If less data is produced, parts of these packets will carry undefined values. This is anyway a well-known strategy and sound practice on embedded multi-core systems: Dynamic allocation may be expensive, especially in the case of shared memory, where runtime allocation must be synchronised between all cores making use of it. Also, the risk of memory insufficiency at runtime may be unacceptable when a system must operate reliably.

Adding functionality to the Ladybirds specification model is also possible; in particular, a directive for specifying loops in metakernels would be entirely feasible. Since this would complicate program optimisation, it is beyond the scope of this work; however, it constitutes an interesting addition to consider in the future.

In more general terms, it is the deliberate strategy of Ladybirds to resolve the conflict of expressiveness and analysability such that the former is clearly subordinated to the latter. The goal is to allow automatic optimisation of parallel applications on multicore platforms to as high an extent as possible. Once the necessary optimisation techniques and methodologies have been established, it is always possible to research on possible extensions to the specification model.

3.4 Optimisation Model

As discussed previously, kernels and metakernels describe functions transforming packets, with metakernels consisting of lists of references to other kernels. Consequently, an entire Ladybirds program can be interpreted as a chain of kernel calls. In order to implement a Ladybirds program, i.e., to translate it to an implementation model, the abstract kernel calls now need to be planned in a more concrete fashion, namely as *tasks* requiring computational resources. Memory must be allocated to hold the packets and in certain cases packets may need to be transferred from one memory module to another. Modelling, orchestrating and scheduling these operations is the challenge of the program optimisation phase, and this section will introduce an optimisation model designed for it. First it will be explained how a Ladybirds C specification can be translated to an optimisation model. Then, the different optimisation decisions mentioned above will be discussed in detail. An abstract description finally summarises the steps and problems of the optimisation phase.

3.4.1 Translation from the specification model

In Figure 3.2 on the following page, the `Canny` application presented in the last section has been translated to a *packet dependency graph*, which is part of the Ladybirds optimisation model. Each kernel call is represented by a *task*; a task is the *instance* of a kernel. As opposed to a kernel, which is a piece of code describing a transformation of packets, a task represents the concrete action of performing this transformation, which needs to be carried out by a processor core. The call hierarchy of the metakernels has been flattened, i.e., calls to metakernels do not appear in the optimisation model but are replaced recursively with the translated kernel calls from their body.

Through so-called *interfaces*, tasks are provided with memory slots called *buffers*, which are there to carry the input and output packets specified in the kernels. Interfaces are represented by rectangles in the graphical representation. Input interfaces are shown on the *input wing* (left-hand side) and output interfaces on the *output wing* (right-hand side) of the task body. Combined input-output interfaces (for carrying `inout` packets) cross both wings, and thick, dashed lines connect interfaces for buddy packets.

Packets declared in metakernels are not directly represented in the optimisation model. Instead, the results of an indexed static single-assignment analysis for multi-dimensional arrays are shown in form of *packet dependencies* between the inter-

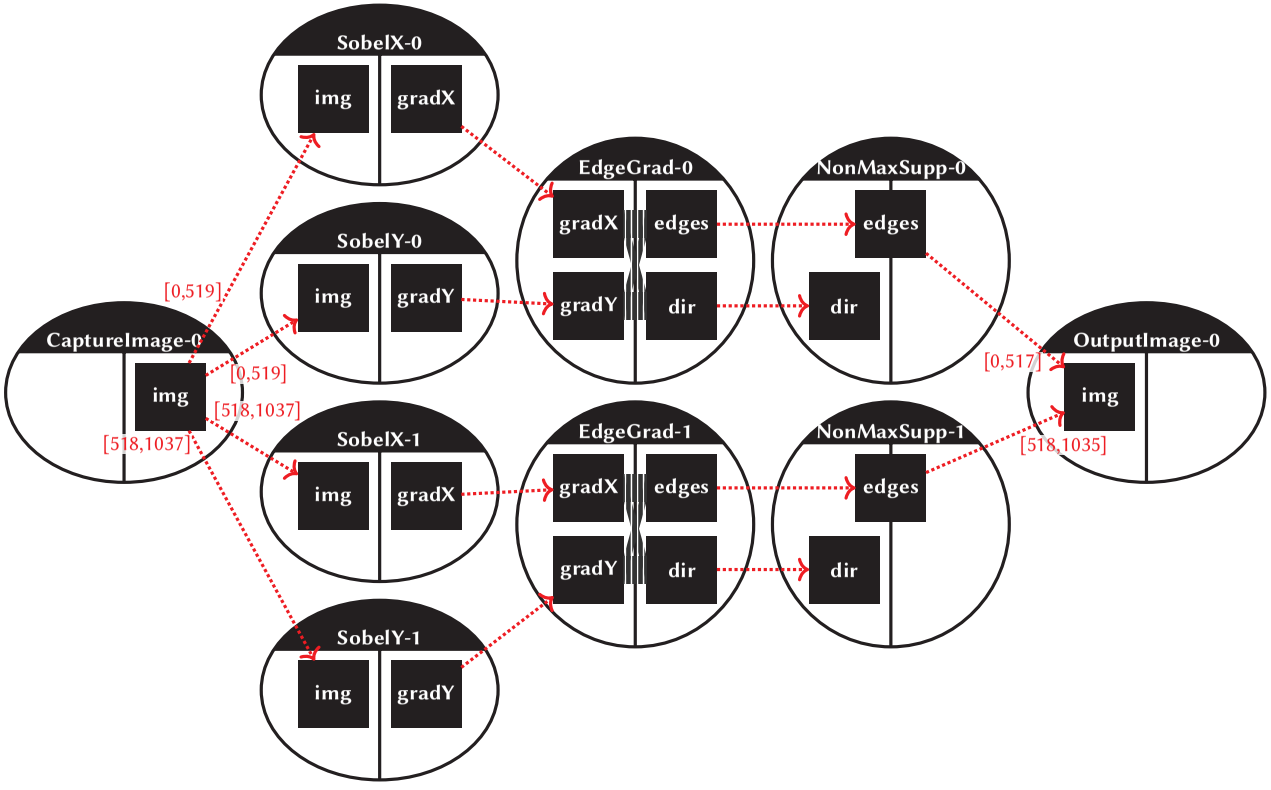


Figure 3.2: Optimisation model for the application from Listing 3.8. The variant from Listing 3.6 has been chosen for the `Canny` metakernel.

faces. Each such dependency is represented by an edge and denotes that the packet produced by its source task and stored to the buffer provided at its source interface will be required by the destination task at the destination interface. For `inout` packets, the conservative assumption that must be taken during the static single-assignment analysis is that they are always entirely modified by the respective kernel call. Some of the dependencies may refer only to parts of the interfaces they link. In Figure 3.2 the outgoing dependencies of `CaptureImage-0:img` denote that only sub-packets of `img` (with the specified indices) will be required by the Sobel tasks. The incoming edges of `OutputImage-0:img` denote that one part of `img` is provided by `NonMaxSupp-0`, the other by `NonMaxSupp-1`. Note that the edges only express data *dependency* relations, but do not include any notion of data transport or exchange.

Figure 3.3 on the next page illustrates the adaptation of packet dependencies when flattening a metakernel hierarchy. Each metakernel can be expressed as a packet dependency graph of task prototypes which contains additional dependencies between the prototypes and the metakernel input and output interfaces. The task prototypes are no proper tasks yet because they could be instantiated multiple times or not at all, depending on how often the metakernel is referenced. In this representation, an `inout` packet in a metakernel declaration leads to two interfaces, one at the input and one at the output side. When the metakernel is instantiated, its tasks have indirect connections to other tasks of the program via the metakernel interfaces. In the course of flattening the metakernel hierarchy, the metakernel interfaces are eliminated by replacing these indirect connections with direct edges between the task interfaces. In certain cases, this leads to `inout` interfaces effectively being replaced with separate input and output interfaces (an example would be if `k2` in Figure 3.3 had separate input and output interfaces instead of a combined one). Still, the possibility of declaring `inout` packets in metakernels is necessary: In the original setting given in the figure, `meta:m` must be declared as `inout` to allow the call to `k2`.

Once the packet dependency graph for an application has been constructed, a further transformation may be helpful for the subsequent analysis and optimisation steps: The feature of packet dependencies referring only to sub-packets of task inputs or outputs, while useful and desired, complicates the formal discussion of program analysis and optimisation, especially when multiple dependencies exist. This can be avoided by *splitting interfaces* into multiple parts such that each dependency always refers to an entire resulting interface. This typically implies that multiple dependencies replace what was a single dependency before. Figure 3.4 on page 95 shows the result of this transformation for the first tasks of the `Canny` example. Since this transformation is always possible, the feature of sub-packets (or sub-interfaces) can be neglected in the following considerations.

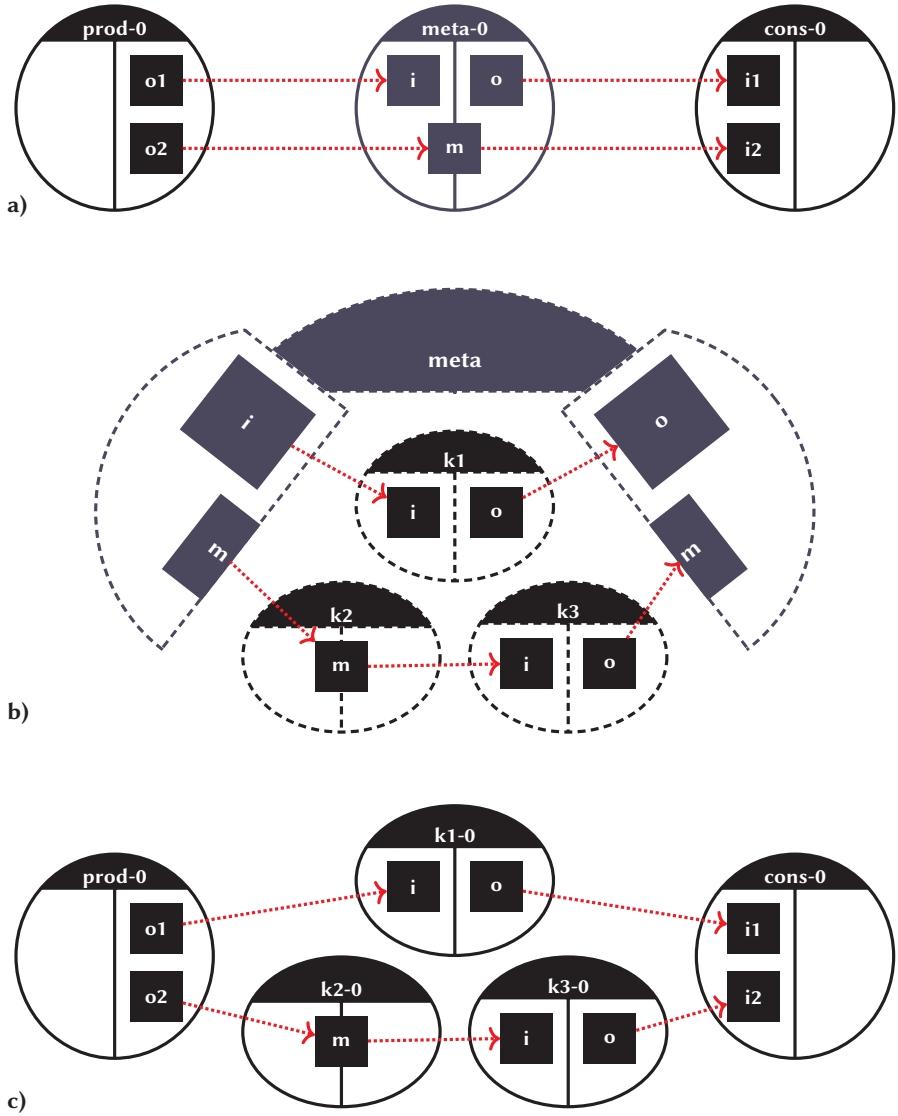


Figure 3.3: Resolving dependencies when flattening metakernel hierarchies. The application depicted in a) contains a reference to a metakernel `meta`, the definition of which is illustrated as a packet dependency graph in b). c) shows the flattened version of the application.

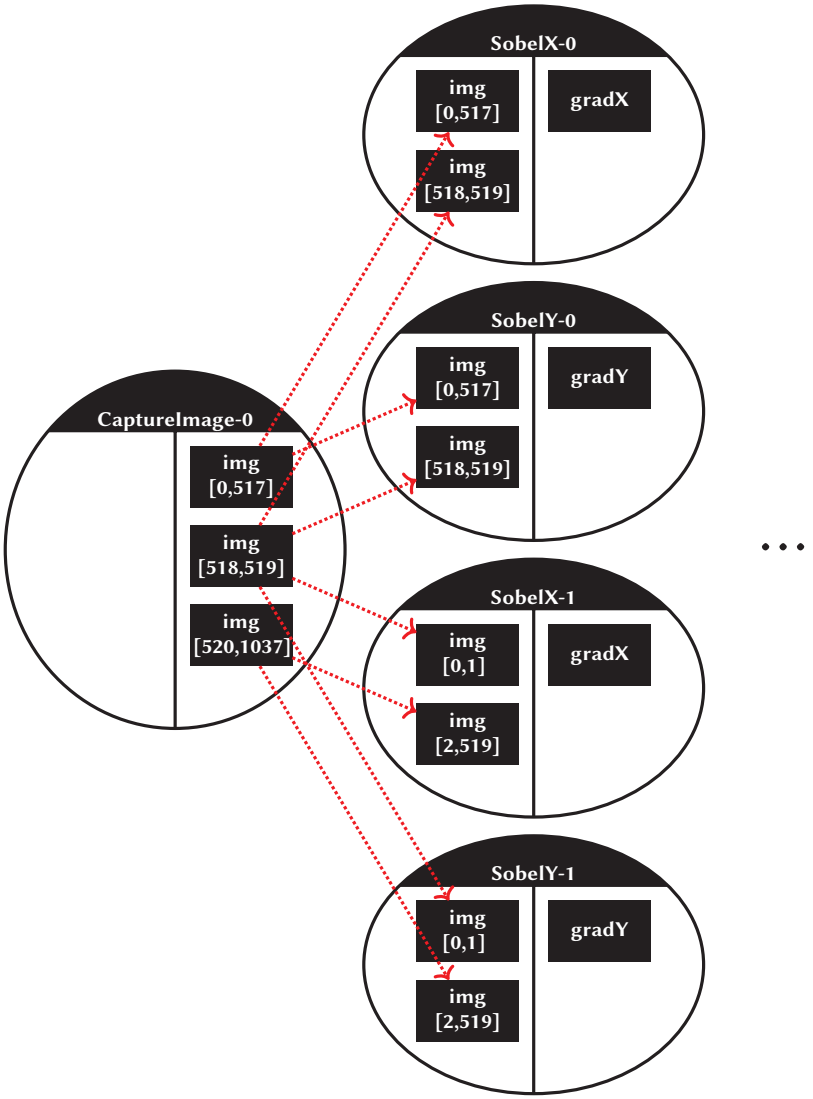


Figure 3.4: Split-interfaces version of Figure 3.2. Only the first tasks are shown.

Figure 3.5: A packet dependency graph (a) and a possible implementation of it, given by a buffer dependency graph (b). While edges in packet dependency graphs are drawn with dotted lines to stress their abstract character, solid lines are used for the concrete semantics of buffer sharing expressed in buffer dependency graphs.

3.4.2 Translation to Implementation Model

Packet dependency graphs are comparable to process networks in the sense that concurrency is explicit in them and parallelisation can be achieved by binding the tasks to processors and scheduling them. There are, however, also intricacies and decision problems that are unique to Ladybirds, in particular the question of how the abstract packet dependencies discussed so far can be met and which mechanisms need to be put in place to do so. These questions must be resolved before a Ladybirds program can be implemented; they shall be focused on in the following.

The simplest mechanism of exchanging data between two tasks is to make them work on the same buffer, e.g., one task writes its output to a buffer which is then provided to another task, which reads its input from it. Typically, this is also the most efficient implementation — for two cores exchanging data via shared memory, this is the natural way of communicating, and for one core sequentially executing both tasks, it can achieve a similar performance, depending on the semantics of the tasks, to merging the kernels of both tasks to one. In the following, *buffer dependency graphs* will be used to represent the data exchange mechanisms. In a buffer dependency graph, the tasks are the same as in a packet dependency graph; however, edges denote data exchange by sharing buffers, i.e., the two interfaces connected by an edge will be provided the same buffer. Figure 3.5 shows the two different graph types for a simple application. Unlike packet dependency graphs, buffer dependency graphs have clear execution semantics: A partial execution order of the tasks is given by the edges, and using the same buffers for all interfaces connected by edges implements communication between the tasks as specified.

Buffer sharing is, however, not always possible. An obvious example is that of two tasks being executed on two cores that do not share any common memory. *Data transfer* is necessary in these situations. It can be modelled in buffer dependency graphs by inserting additional data transfer tasks, as Figure 3.6 shows.

Even if shared memory is available and can be used, buffer sharing is not always possible, as the application in Figure 3.7 on page 98 shows. In this example, a

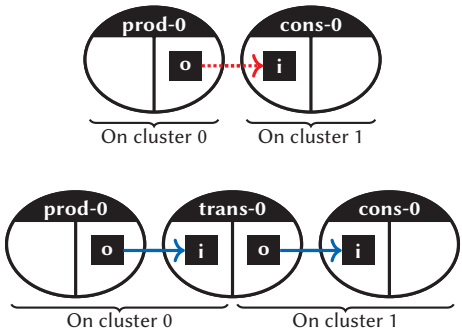


Figure 3.6: Insertion of data transfer tasks. Two tasks are to be executed on different clusters of a platform that do not share any memory. During the translation from the packet dependencies (above) to the buffer dependencies (below), a *transfer* task is inserted which represents the packet transmission from the first to the second cluster, e.g. over a network on chip. Typically, the implementation of this task will consist of two parts — one that sends on the one cluster and one that receives on the other. In such a case, the execution of the program can then be managed independently on each cluster, since the synchronisation of the clusters is accomplished by the transfer task(s), with the data transmission by its nature ensuring the correct scheduling order.

produce-read-modify situation occurs, namely two tasks (`read-0` and `mod-0`) operate on the same data and one of them (`mod-0`) wants to modify it. While such a configuration is perfectly valid in an abstract packet dependency graph, a one-to-one translation to a buffer dependency graph is not possible: Simply replacing the packet dependency edges with buffer dependency edges would allow `mod-0` and `read-0` to work simultaneously on the same buffer, which would likely lead to unsynchronised reading and writing, breaking the semantics of the application. For a valid buffer dependency graph, two possible efficient solutions exist, which are shown in Figure 3.8 on the following page. The first solution is to execute the processes sequentially, essentially adding a dependency from `read-0` to `mod-0`. The second solution is, instead of providing the original buffer, to create a copy of it, which then `mod-0` can safely modify. Both solutions make sense in different scenarios. If, for instance, all tasks are mapped to the same processor, the first solution is the most natural. Configurations requiring data transfers between `prod-0` and `mod-0` (as seen before) automatically lead to the second solution. In other cases, the decision is not so clear and other factors may play a role, like size of the packet, available memory, task execution times, numbers of memory accesses, other tasks that need to be scheduled etc.

```
int packet[…];
prod(/*out*/ packet);
read(/*in*/ packet);
mod(/*inout*/ packet);
read(/*in*/ packet);
```

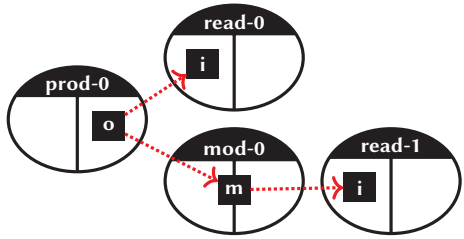


Figure 3.7: Example for a produce-read-modify situation. A sample code for a metakernel is given on the left and the resulting task graph on the right.

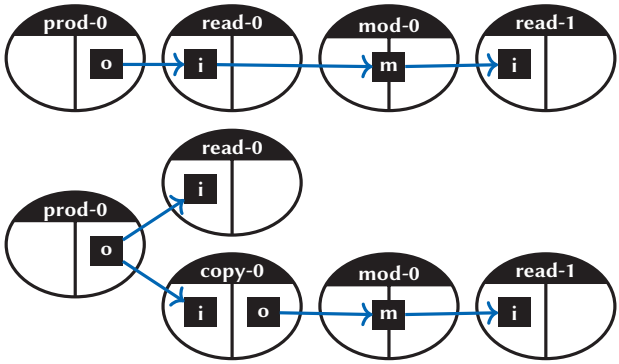


Figure 3.8: Two buffer dependency graphs with efficient implementation options for the produce-read-modify configuration as given in Figure 3.7

Such considerations, in addition to the well-known mapping problems from process networks, are the core of the Ladybirds optimisation phase. Given a specification of which packets (i.e. data) are produced and required in which tasks, the optimiser must allocate buffers to carry those packets and arrange all necessary copying and data transport. Two additional considerations may be relevant in certain situations.

- **Additional copy tasks** or data transfer tasks may, even when not necessary for program execution, still increase application performance. A typical example are memory hierarchies with a larger memory shared between multiple cores and small, fast scratch-pad memories private to each core. A task can execute significantly faster if its input packets have been copied into a scratch-pad memory or if its output buffers are allocated there, necessitating a later transfer to shared memory.

- **Buddy interfaces**, i.e. pairs of an input and an output interface declared as buddies, can be merged by simply providing the same buffer to both of them. The question of whether or not to do this depends again on the application context. If, for instance, the `mod-0` task in the produce-read-modify example (Figure 3.7), instead of an `inout` interface had two interfaces, `in` and `out`, declared as buddies, `mod-0` and `read-0` could execute in parallel without an additional copy task (one would just pass a different buffer the output interface of `mod-0`). On the other hand, also the solution of merging the interfaces and executing all tasks sequentially may be appropriate in certain cases (e.g. all tasks are mapped to the same processor, or not much memory is available).

3.4.3 Summary of the Optimisation Procedure

The previous descriptions have introduced the concepts and ideas of transformation and optimisation steps that have to be carried out during the optimisation phase of a Ladybirds application. These steps shall now be generalised and summarised. Since this section is about the Ladybirds optimisation model but not about the actual optimisation techniques, it will only give a list of steps, but no concrete algorithms.

As discussed previously, the first analysis performed on a Ladybirds application specification is a dependency analysis leading to the construction of a packet dependency graph. The goal of the optimisation procedure is now to transform this packet dependency graph into a buffer dependency graph, where

- Each task is assigned to a processing element (*task binding*),
- At least one buffer is allocated for each packet (*buffer allocation*), such that the processors executing the tasks can access the required buffers and that the limitations of memory availability are respected,
- Additional copy tasks or buffer dependencies are inserted wherever the choice of the buffers makes it necessary (*copy optimisation* — cf. Figure 3.8),
- For buddy interfaces, it is decided whether they are merged (*buddy optimisation*),
- Data transfer tasks are inserted wherever necessary (*transfer management*),
- Each copy or transfer task is assigned to a processing element or DMA controller (*transfer binding*),
- The order of all tasks (including copy and transfer tasks) is determined (*task scheduling*) such that all buffer dependencies are fulfilled.

These decisions and transformations are carried out such that a certain target metric, for instance the execution time of the entire program, is optimised.

The purpose of these optimisations is that all the requirements set out previously for efficient data exchange and memory usage are met, in particular also sharing of data and in-place modifications. The optimisation model is generic enough to allow such optimisations for an arbitrary platform; different transformations for different architectures and memory hierarchies have been discussed, but none of them was fixed or presumed for the overall optimisation process.

Note that the above description focuses on static bindings and schedules. It is possible to have a more flexible implementation, however at a higher complexity in optimisation procedure, implementation and runtime system. In the special case of dynamic scheduling, the additional flexibility can also be bought at the price of a higher memory consumption: More buffers must be available simultaneously such that different tasks can be dynamically scheduled. In short, the above description, which also sets up the largest possible design space, is just one of many other possible optimisations procedures. However, it prepares a program specified in Ladybirds C for being implemented, as will be shown in the next section.

3.5 Implementation Model

A buffer dependency graph of an application, once constructed, contains all necessary semantics of an efficient parallel execution of the application. The semantics of the tasks are given by the kernels, and each communication or data exchange is explicit — either through buffer sharing edges or through dedicated data transfer tasks. What remains to be shown is that there exist efficient techniques for implementing these functionalities on the platforms. The two relevant points in this context are buffer dependencies and reading and writing packets, in particular subpackets, from/to buffers in the course of task execution.

As to the buffer dependencies, an implementation model needs to provide an efficient technique to guarantee that they are respected. As it turns out, however, these dependencies are no longer necessary in this degree of detail. Since buffers have been allocated for all interfaces during optimisation, it is sufficient to ensure that the scheduling constraints they impose on the tasks are respected. Therefore, the constraints can be merged at task level. Figure 3.9 illustrates this using the Canny example from the previous sections. The result of this transformation can be regarded as an SDF graph in which all firing counts are one (also known as homogeneous SDF or marked graph). Note, however, that the tokens sent in this graph do not contain any data, they just represent scheduling constraints. Data exchange is performed purely through the buffers, which, however, are not shown in the figure.

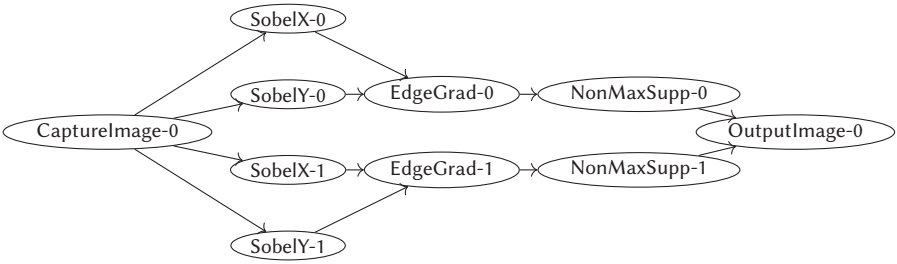


Figure 3.9: Implementation model for the `Canny` application (cf. Listing 3.8 on page 88 and Figure 3.2 on page 92).

Since such a graph is also acyclic by construction and each task is executed exactly once, the edges can be implemented by simple bit masks that indicate whether a task has already finished or not. A scheduler can then easily check whether all dependencies of a task have been fulfilled such that the task is ready for execution.

The point that remains to be discussed is how to implement the concept of (sub-)packets (cf. p. 85) in tasks. In the Ladybirds model, a task execution boils down to a C function call, where the function is an implementation of the kernel which the task instantiates. Possible kernel parameters (declared as `param` in the specification) need to be passed to the function as well as references to the buffers that were allocated for the task interfaces. Since these buffers could also be parts of a larger buffer (when a sub-packet was passed in a meta-kernel declaration), the question is how to implement such functionality in C. To answer this question, it helps to regard how an array access is interpreted in C. An element `arr[1][2][3]` of an array declared as `int arr[4][5][6]` is translated to the byte address `&arr+`$\big((1 \cdot 5 + 2) \cdot 6 + 3\big)\cdot$`sizeof(int)`. More formally, for an n-dimensional array with a size given by a vector $\boldsymbol{s} \in \mathbb{N}^{n+1}$, (where s_1 shall denote the size in bytes of the base type), an n-dimensional address given by a vector $\boldsymbol{a} \in \mathbb{N}^{n}$ is translated to the byte address

$$base + \Big(\big(\cdots\big(a_n \cdot s_n + a_{n-1}\big) \cdot s_{n-1} + \ldots\big) \cdot s_2 + a_1\Big) \cdot s_1 = base + \langle \boldsymbol{a}, \boldsymbol{s}^* \rangle,$$

with the base address *base* of the array, $\langle \cdot, \cdot \rangle$ the scalar product and $\boldsymbol{s}^* \in \mathbb{N}^{n}$ given as

$$s_i^* = \prod_{k=1}^{i} s_k.$$

For a sub-packet starting at the multi-dimensional index $(0, 0, \ldots, 0)$ of the original packet, the called function can therefore correctly calculate all byte addresses if it knows the size $\boldsymbol{s}$ of the original buffer that was allocated. If a sub-packet is located "in the middle" of the original packet, i.e., the sub-packet index $(0, 0, ..., 0)$ is mapped to an index $\boldsymbol{d} \in \mathbb{N}^n$ in the original packet, the byte address of a sub-packet index $\boldsymbol{a}$ can be calculated as

$$base + \langle \boldsymbol{a} + \boldsymbol{d}, \boldsymbol{s}^* \rangle = \left(base + \langle \boldsymbol{d}, \boldsymbol{s}^* \rangle \right) + \langle \boldsymbol{a}, \boldsymbol{s}^* \rangle.$$

This corresponds to an array of dimension $\boldsymbol{s}$ with a base address equivalent to the byte address of element $\boldsymbol{d}$ in the original packet. In short, a sub-buffer reference can be passed to a function by passing a pointer with a base address and an array with the dimensions of the original buffer that was allocated. Along the same lines, sub-packets with reduced dimensionality can be implemented by removing elements in $\boldsymbol{s}^*$, which corresponds to multiplying the sizes of the removed dimensions into the sizes of the dimensions below. For instance, for a packet declared as `int arr[4][5][6]`, a two-dimensional sub-packet `arr[0,2][3][1,3]` can be emulated as an array of dimensions 4×30 with the base pointer at the address `&arr[0][3][1]`. Reducing dimensions at the lowest level is equally possible (e.g. `arr[0,2][0,2][3]`); however, it requires modifications to the original code.

Listing 3.9 shows how a kernel specification code can be modified such as to accept sub-buffers. Note that the C code in the kernel body is not changed. The only difference as compared to a direct implementation with fixed array sizes is that the C compiler cannot perform strength reductions which were possible before, e.g. bit shifts instead of multiplications when array dimensions were powers of two. However, when iterating over arrays (which is a very typical case), element addresses can also be calculated by adding offsets in each iteration, such that no performance is lost. As a complementing optimisation, it would be possible to hard-code the kernel parameters and buffer sizes that actually appear in a program, possibly producing multiple implementations of the same kernel.

The reference-based sub-packet implementation is also the reason why Ladybirds C does not support pointer arithmetic on packets or their elements: A programmer specifying a kernel has no knowledge about how the packet elements will later be arranged in memory. Calculating their addresses according to conventional C programming practices is therefore not possible.

This section discussed two simple mechanisms, task dependency bitmasks and (sub-)buffer references, which together are sufficient to implement a Ladybirds application. Data exchange and memory efficiency have been addressed in the optimisation process, the results of which can be implemented one to one. The overhead for scheduling is reduced to bitmasks and the execution of a task is as simple as a

Listing 3.9: Code transformation from a Ladybirds C kernel specification (**a**) to a conventional C function (**b**). Each packet in the argument list of the kernel is replaced with two function arguments, viz. base pointer of the (sub-)buffer and size of the originally allocated buffer. Those two arguments are then used to construct an array pointer with the name of the original packet at the beginning of the C function implementation.

(a) Ladybirds C kernel specification

```
kernel(SobelX)(param int height, param int width,
               in uchar Image[height][width],
               out char GradX[height-2][width-2])
{
   for (int y = 0; y < height-2; ++y)
      for (int x = 0; x < width-2; ++x)
         GradX[y][x] =   Image[y  ][x]   - Image[y  ][x+2]
                       + 2*Image[y+1][x] - 2*Image[y+1][x+2]
                       +   Image[y+2][x] -   Image[y+2][x+2];
}
```

(b) Implementation as C function

```
void SobelX(const int height, const int width,
   const int _lb_size_Image[2], const void *_lb_base_Image,
   const int _lb_size_GradX[2], void *_lb_base_GradX)
{
   const uchar (*Image)[ _lb_size_Image[1] ] =
      (const uchar (*)[_lb_size_Image[1]]) _lb_base_Image;
   char (*GradX)[ _lb_size_GradX[1] ] =
      (char (*)[_lb_size_GradX[1]]) _lb_base_GradX;

   for (int y = 0; y < height-2; ++y)
      for (int x = 0; x < width-2; ++x)
         GradX[y][x] =   Image[y  ][x]   - Image[y  ][x+2]
                       + 2*Image[y+1][x] - 2*Image[y+1][x+2]
                       +   Image[y+2][x] -   Image[y+2][x+2];
}
```

function call. Since the tasks are stateless, no preemption techniques are necessary. In summary, it can be stated that the Ladybirds implementation model achieves a high degree of efficiency and addresses well the requirements for implementation models set out in Section 3.1.

3.6 Other implementation models

The previous sections have introduced the Ladybirds model set and shown how it can be used to obtain efficient parallel application implementations on different multicore platforms and architectures. This section will show that Ladybirds is also useful even for single-threaded program execution. Two such cases shall be presented, firstly that of debugging Ladybirds C code and secondly that of state retention in systems with volatile power supply.

3.6.1 Debugging

As already mentioned, debugging parallel code is known to be a notoriously tedious and complicated endeavour. The major difficulties lie in locating the source of an error among multiple independent threads, and in the fact that parallel programming introduces a large number of additional error sources such as synchronisation problems, glitches, out-of-bounds array accesses impacting other cores etc.

Ladybirds substantially simplifies this issue because the code specification in Ladybirds C is a single-threaded one. This makes it possible to circumvent the entire optimisation phase and instead obtain a single-threaded implementation directly from the specification. The last section showed how a kernel can be transformed to obtain a C function implementation; the same transformation can also be applied to metakernels. When doing so, however, also the metakernel bodies must be adapted such that the kernel calls are transformed into native C function calls with the correct arguments for the sub-buffers. While packet declarations in metakernels constitute valid C code, their meaning in C is to allocate arrays on the stack. For small packets, this is not problematic; to cover the case of larger packets, however, it is advisable to transform these directives to dynamic memory allocation instructions. Generator variables and loops can be left unchanged and will be executed as normal C instructions. The `genvar` keyword is easily neutralised with a preprocessor directive. Listing 3.10 shows the debug implementation generated for the `Canny` metakernel.

With these small and straightforward transformations, any Ladybirds C program can be compiled using a conventional C compiler and debugged using conventional debugging tools. Adding C preprocessor `#line` directives during code transformation even allows to display the original Ladybirds C code in the debugger. Moreover, since packet sizes are well-defined in Ladybirds C, it is possible to add instrumentation code that detects out-of-bounds accesses during the debug run. A programmer can thus comfortably execute a sequential implementation of his application on his computer, spotting and removing possible bugs. The correct(ed) program specification is then optimised to an efficient parallel implementation, which is correct by construction. Parallel debugging can therefore be entirely avoided in many cases.

Listing 3.10: Debug implementation of the `Canny` metakernel in Listing 3.7. The packet declarations in the body have not been transformed in this example.

```
void Canny(
   const int _lb_size_Image[2], const void *_lb_base_Image,
   const int _lb_size_Edges[2], void *_lb_base_Edges)
{
#line 95
   const uchar (*Image)[ _lb_size_Image[1] ] =
      (const uchar (*)[_lb_size_Image[1]]) _lb_base_Image;
#line 95
   uchar (*Edges)[ _lb_size_Edges[1] ] =
      (uchar (*)[_lb_size_Edges[1]]) _lb_base_Edges;

#line 96
   char GradX[EdgeHt][EdgeWd];
   char GradY[EdgeHt][EdgeWd];
   uchar Direction[EdgeHt][EdgeWd];

   genvar int nstripes = 4;
   genvar int blockht = EdgeHt/nstripes;
   for (genvar int y1 = 0; y1 < EdgeHt; y1 += blockht)
   {
      genvar int y2 = y1+blockht-1;
      SobelX(blockht+2, ImgWd, (int []){_lb_size_Image[0],
         _lb_size_Image[1]}, Image[y1],
         (int []){1, 1036}, GradX[y1]);
      SobelY(blockht+2, ImgWd, (int []){_lb_size_Image[0],
         _lb_size_Image[1]}, Image[y1],
         (int []){1, 1036}, GradY[y1]);
      EdgeGrad(blockht, EdgeWd, (int []){1, 1036},
         GradX[y1], (int []){1, 1036}, GradY[y1],
         (int []){_lb_size_Edges[0], _lb_size_Edges[1]},
         Edges[y1], (int []){1, 1036}, Direction[y1]);
      NonMaxSupp(blockht, EdgeWd,
         (int []){_lb_size_Edges[0], _lb_size_Edges[1]},
         Edges[y1], (int []){1, 1036}, Direction[y1]);
   }
}
```

3.6.2 State retention on transient systems

Another case in which Ladybirds is useful beyond parallel programming are *transient systems*. A transient system is a digital device that is powered only by a volatile energy source such as a solar panel. Batteries are not used because of their limited number of recharge cycles, their cost and their possible environmental impact, so only capacitors are used as small energy buffers to allow for a controlled shut-down in case of energy breakdown. Typical examples are small *sensor nodes*, i.e. devices which acquire data from a sensor, process it and then transmit the results over a wireless connection. Being highly autonomous and maintenance-free, such transient systems lend themselves for instance to autonomous monitoring installations in unreachable terrain.

Since, however, their power supply is gained through energy harvesting, they must be prepared for a power cut at any moment — consider, for instance, the case of a solar panel subjected to varying lighting conditions. Before power runs out, transient systems need to save to a non-volatile memory, e.g. FRAM, all data relevant for further execution (*state retention*). Due to technical reasons related to the manufacturing process, the non-volatile memory is located on an external chip in many systems, such that the storing and the subsequent loading procedure have significant costs in terms of time and energy. For these systems, it is vital to minimise the size of the state that needs to be retained, thereby increasing the energy available for productive tasks or even enabling a reduction of the capacitor size. This requires, however, accurate knowledge about which data really needs to be saved and how this can be determined. Such knowledge makes it possible not only to avoid unnecessary data retention overhead but also to leverage other application optimisations which can further reduce state retention overhead. Unfortunately, current approaches to this issue face considerable limitations.

- Manual specification is tedious and error-prone.
- Automatic analysis of generic code is infeasible in general.
- Some software solutions [RSF11; BM16] take the conservative approach of saving the contents of all occupied regions of the volatile memory. This leads to an overly high amount of precious energy being spent on state retention overhead.
- An improvement to this is to save only those areas that have been modified. To achieve that, it has been proposed [BM16] to first read the non-volatile memory (which, in the case of flash memory, is much less costly) or to build special hardware using techniques comparable to dirty bits in caches (e.g. in [Liu+16], however for on-chip ReRAM). This reduces, but does not solve

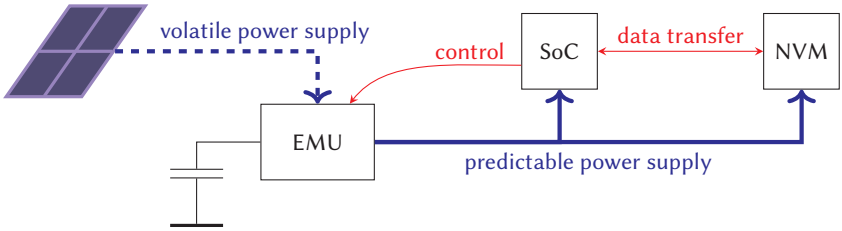


Figure 3.10: Illustration of a transient system architecture according to [Gom+16]. An energy management unit (EMU) gathers volatile energy supplies in a capacitor and uses them to reliably power a system-on-a-chip (SoC) and a non-volatile memory (NVM) for predefined time spans.

the problem of the conservative approach. Consider, for instance, a neural network based image classification algorithm that stores tens of kilobytes or even megabytes of intermediate data for its neurons, whereas only the classification result is important, which typically can be stored in one single byte.

This problem is remedied by Ladybirds: With its clearly defined task inputs and outputs, it allows the automatic analysis of data dependencies and therefore of what data must be conserved for later use. This consideration will be substantiated by the following methodology for minimising the state retention overhead of a given Ladybirds application, which results from a collaboration with Andrés Gomez, Pascal Alexander Hager and Praveenth Sanmugujarah.

We consider a type of platforms as proposed in [Gom+16]. Their architecture is illustrated in Figure 3.10. It contains a conventional low-power system-on-a-chip (single-core processor, volatile memory module) which is connected to a separate chip providing non-volatile memory. Power is supplied by a volatile source, which is managed by an *energy management unit*. This unit uses the incoming power to load a capacitor. When enough energy has been stored, the processor is powered on, initialises the system-on-a-chip, executes a number of tasks and turns itself off again. Prior to task execution, it fetches required data from the non-volatile memory and afterwards, it writes results back. The entirety of all the actions between power-up and turn-off is called a *burst*; the energy management unit guarantees that each burst can be completed without power cuts.

This architecture has a number of distinct advantages. Firstly, the energy management unit decouples the input voltage for the power supply and the output voltage for the load, thereby achieving a high degree of energy efficiency. Secondly, due to the principle of energy bursts, it also works for configurations in which the power

supply from the source is significantly less than the power demand required by the load. Thirdly, it guarantees application progress even with highly intermittent power sources instead of repeatedly starting up the processor without finishing any task. Finally, it allows to plan ahead for the task execution such that state retention can be performed when the amount of data that must be saved is small.

The question that arises now is how many and which tasks should be combined in a single burst. In general it is advantageous to end a burst where the amount of data that needs to be stored and restored is small; however, this may not always be possible, since burst sizes are limited by the energy amount that can be maximally stored in the capacitor. The problem is complicated by the fact that the decisions taken for one burst influence the options for the next, thereby turning the local optimisation problem into a global one. The following will show how this problem can be solved using Ladybirds.

We model the system as follows. The program to be executed is a Ladybirds application consisting of n_t tasks $t_1, \ldots, t_{n_t}$, which are executed in that order. This order follows the (sequential) Ladybirds C specification. Each task t_i reads and writes, respectively, a set of packets $P_i^r \subseteq P$ and $P_i^w \subseteq P$, with $P = \bigcup P_i^w$ the set of all packets in the application. Each packet $p \in P$ is contained in exactly one P_i^w, but can belong to one or multiple[2] P_i^r. This corresponds to the Ladybirds optimisation model with split interfaces; however, `inout` packets and buddies are not of interest in this context and are thus modelled as separate inputs and outputs.

Each task t_i consumes an amount $E^x(t_i)$ of energy for its execution, the loading and storing of a packet p from/to the non-volatile memory requires energy amounts of $E^r(p)$ and $E^w(p)$, respectively, and the start-up energy needed for initialising the system-on-a-chip is E^s. The problem to be solved is now as follows:

Given a maximum amount E^{cap} of energy that can be stored in the capacitor, partition the tasks into a set of bursts such that (i) no burst consumes a higher amount of energy than E^{cap} and (ii) the total amount of energy consumed during all bursts is minimised.

To approach that problem, we first calculate the energy required for one burst. We denote as $E\langle i, j\rangle$ the energy required to execute a burst that executes the tasks $t_i, \ldots, t_j$. If only one task is executed in a burst, i.e. $i = j$, the burst consists of starting up the processor, loading the input packets of the task, executing the task and storing its output packets to non-volatile memory. The burst energy can thus be calculated as

$$E\langle i, i\rangle = E^s + \sum_{p \in P_i^r} E^r(p) + E^x(t_i) + \sum_{p \in P_i^w} E^w(p).$$

[2] According to the Ladybirds model, there could also be packets that are not read by any task. Since, however, these packets never need to be saved or restored, we can leave them out of our considerations without loss of generality.

If now a second task is added to the burst, the burst energy can be computed analogously, except for a number of additional effects that need to be considered:

- An input packet of the second task could also be an input packet of the first task. In that case, it would not have to be loaded twice.
- An input packet of the second task could have been produced by the first task. In that case, it would already be in the volatile memory as well.
- An output packet of the first task could be used solely by the second task. In that case, it would not have to be saved to the non-volatile memory.

Similar considerations hold for tasks consisting of multiple bursts. To formalise these constraints, we define the *last use* $l_j(p)$ of a packet p prior to an index j as

$$l_j(p) = \max\left\{ i < j \;\middle|\; p \in P_i^r \;\vee\; p \in P_i^w \right\},$$

i.e. as the highest index less than j of a task that reads or writes p. This definition includes $l_\infty(p)$, the index of the last task in the application that reads or writes p. We can now express the sets of packets $P_k^r\langle i, j\rangle$ and $P_k^w\langle i, j\rangle$ that must be loaded and stored, respectively, for a task t_k contained in a burst executing tasks $t_i, \ldots, t_j$:

$$P_k^r\langle i, j\rangle = \left\{ p \in P_k^r \;\middle|\; l_k(p) < i \right\}, \qquad P_k^w\langle i, j\rangle = \left\{ p \in P_k^w \;\middle|\; l_\infty(p) > j \right\}.$$

The burst energy can then be computed as

$$E\langle i, j\rangle = E^s + \sum_{k=i}^{j} \left(\sum_{p \in P_k^r\langle i,j\rangle} E^r(p) \;+\; E^x(t_k) \;+ \sum_{p \in P_k^w\langle i,j\rangle} E^w(p) \right).$$

Note that due its explicit input/output specification, Ladybirds allows parallel data transfer and execution. For instance, after loading the packets in $P_i^r\langle i, j\rangle$, task t_i can start executing while a DMA controller already loads the packets in $P_{i+1}^r\langle i, j\rangle$ and maybe even data for later tasks. Similarly, it can also transfer output packets of t_i to the non-volatile memory while later tasks still execute. This technique saves time and thus energy. The burst energy calculation could be adapted accordingly; due to the intricacies involved in predicting task execution times and thus data transfer concurrency, this is however beyond the scope of this work.

Now that the burst energies are known, we can solve the task partitioning problem, i.e. the question of how the tasks should be partitioned to bursts such that the overall application execution energy is minimised while respecting the upper energy bound E^{cap} for all bursts.

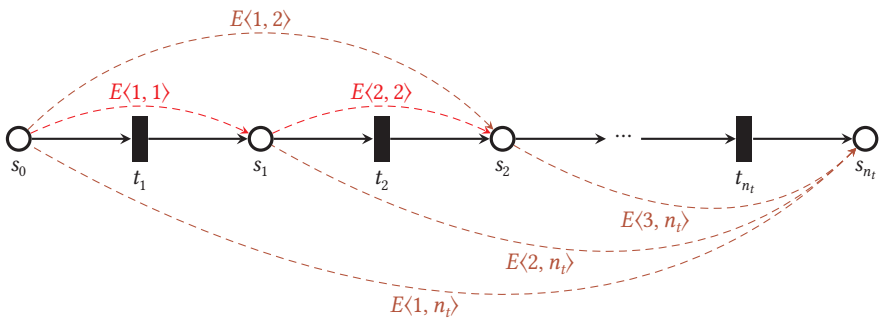


Figure 3.11: Graphic illustration of the states $s_0, \ldots, s_{n_t}$ between the bursts for the case of single-task bursts executing the tasks $t_1, \ldots, t_{n_t}$. Burst energies for transitions between the states have been added (lighter colour for single-task bursts). This leads to the construction of a burst graph.

To address this problem, we first consider the case that each task is executed in its own burst. In this case, we have n_t bursts and we can define $(n_t + 1)$ power-off states $s_0, \ldots, s_{n_t}$. In this definition, s_0 is the state before application execution has started, s_1 is the state after the burst with t_1 has executed etc. (see Figure 3.11). For getting from s_0 to s_1, the system consumes the energy $E\langle 1, 1\rangle$, for getting from s_1 to s_2 it consumes $E\langle 2, 2\rangle$ and so forth. If we now take bursts with multiple tasks into consideration, the system can also get from s_0 directly to s_2 by executing a burst with t_1 and t_2, consuming the energy $E\langle 1, 2\rangle$. In this way, we can construct a *burst graph* connecting each state s_i to any state s_j, $j > i$, and assign to each edge a cost equivalent to the corresponding burst energy.

This graph offers a new interpretation of the task partitioning problem. The minimisation of the total application execution energy can be interpreted as the problem of getting from s_0 to s_{n_t} with the smallest cumulative cost. The constraint of keeping the burst energy below E^{cap} translates such that no edges with a cost larger than this upper bound may be considered. The task partitioning problem therefore is equivalent to the problem of finding the shortest path from s_0 to s_{n_t} on a burst graph from which the edges with a cost higher than E^{cap} have been removed.

Shortest path problems can be efficiently solved, for instance, using Dijkstra's algorithm [Dij59], with a complexity of $\mathcal{O}({n_t}^2)$. The complexity of the proposed calculations is, in fact, dominated by the computation of the burst energies $E\langle i, j\rangle$: There are $\frac{1}{2}\, n_t\,(n_t + 1)$ possible burst energies to calculate. For each such burst, one needs to iterate over all tasks contained in it and check each packet accessed by

each task for whether it needs to be transferred or not. These checks can be implemented efficiently (amortised $\mathcal{O}(1)$) using lookup tables, which leads to an overall complexity of $\mathcal{O}({n_t}^3 \cdot |P|)$. While this is more complex than the mere shortest path calculation, we still have presented a method which finds the optimal solution to the task partitioning problem in polynomial time.

In addition, the proposed method can not only be used to minimise the overall application execution energy. It can also determine the smallest energy storage capacity $E_{\text{min}}^{\text{cap}}$ necessary to execute a given application. Note that this energy is *not* equivalent to the highest burst energy of all single-task bursts, since adding tasks to a burst could save more data transfer energy than it costs additional task execution energy. $E_{\text{min}}^{\text{cap}}$ can be calculated by modifying the shortest path algorithm such that a path length is not calculated by adding all costs along the path, but instead by choosing the maximum cost along the path. Applying the modified algorithm to the complete burst graph (with no connections removed) will yield $E_{\text{min}}^{\text{cap}}$ as the shortest path cost. Another run of the original partitioning algorithm then returns the optimal partitioning for this maximum burst cost.

In the following, the proposed approach for grouping tasks to bursts for execution on a transient system shall be discussed. The previous considerations demonstrated that for Ladybirds applications, there is an efficient method for optimally partitioning a sequence of tasks. There is, however, one optimisation lever which has not been regarded yet. In the proposed methodology, the order of the tasks was assumed as given (we used the sequential order from the application specification). The Ladybirds optimisation model, however, makes parallelism explicit and therefore also allows to reorder the tasks. A promising idea would be to do this in such a way that those tasks sharing large amounts of data are close to each other in the execution order, allowing them to be executed in the same burst and therefore reducing state retention costs. This optimisation problem, however, does not seem to have a polynomial solution. Possible approaches might include greedy, but also evolutionary heuristics with the results of the proposed partitioning algorithm as an evaluation metric during the selection step.

In that sense, the previous approach of following the sequential execution order from the specification can also be regarded as a heuristic. In fact, this heuristic is far from arbitrary and produces good results in many cases: Programmers typically write their code such that they can easily trace the data that is modified throughout the code. This coding scheme is likely to result in a specification in which tasks exchanging large amounts of data are placed close to each other. However, like with every heuristic, exceptions exist. One example is when multiple processing steps are parallelised independently of each other as shown in Listing 3.11 on the following page. Knowing the compilation mechanism, however, the programmer can easily influence this behaviour.

Listing 3.11: Code snippet of a metakernel that would lead to suboptimal results on transient systems when executing the tasks in specification order. In this extreme example, an image `img` is split into 8 blocks, all of which are subjected to the same filter before the next filter is applied. This may lead to each intermediate result having to be stored on non-volatile memory when following the specified order. It would be clearly advantageous to put all filter calls into one generator loop, such that all filters are applied subsequently to each block. In this case, multiple filter tasks might be combined in one burst, leaving intermediate results in volatile memory only.

```
uchar img[1024][1024];
AcquireImg(img);

for(genvar int i = 0; i < 8; ++i)
   Filter1(/*inout*/ img[i*128,i*128+127]);

for(genvar int i = 0; i < 8; ++i)
   Filter2(/*inout*/ img[i*128,i*128+127]);

for(genvar int i = 0; i < 8; ++i)
   Filter3(/*inout*/ img[i*128,i*128+127]);

//further processing...
```

In summary, we have presented a method to group a sequential list of tasks into bursts such that the total energy consumption is minimised for the transient system architecture described in [Gom+16]. The minimisation is performed under the constraint of a maximum energy that can be stored in a capacitor, and we can calculate the minimum energy a capacitor must be able to store to execute an application. To be able to perform these computations, we have shown how the state that must be retained can be accurately determined for any Ladybirds program. Different heuristics have been discussed as to finding the optimal task order of a program, although further research on this topic may be useful. The important take-away from this example is that this promising optimisation methodology was only made possible by the Ladybirds programming model and its concept of dividing programs into tasks with clearly specified inputs and outputs. Ladybirds C allows to comfortably and intuitively specify such programs; by restricting the programmer to a limited set of features in the top level program architecture, it enables analyses and optimisations that would not have been possible otherwise.

3.7 Concluding remarks

In this chapter, a set of requirements and desirable properties for a specification model — and thus an entire model set — has been derived from the results of the previous chapter as well as from generic considerations. One major requirement was an intuitive, comfortable and platform-independent way of specifying parallel programs with the goal of achieving efficient data exchange between the cores on any arbitrary platform architecture. A survey of parallel programming methodologies showed that no existing specification model fulfilled all of these requirements. As a result, the Ladybirds model set was developed with the requirements in mind. Apart from expressiveness, which was deliberately attributed lower priority, it was shown that all of the mentioned requirements and desirable properties were taken into consideration. The two examples of debugging and transient systems showed the usefulness of the Ladybirds model set also in other domains.

An important part of the Ladybirds concept is to simplify program analysis. A Ladybirds application consists of the one-time execution of a set of atomic tasks with clearly defined inputs, outputs and dependencies. There is no iteration, parallelism is explicit and all required data exchange is known exactly at compile time, making complex and possibly inaccurate analysis unnecessary. Yet, as the considerations about the possible optimisations for efficient data exchange or for energy-efficient execution in transient systems have shown, complex problems still remain to be solved. The simple optimisation model only seemingly contradicts the perception that the optimisation problems are at least as complicated as for many other optimisation models. In fact, the simplicity in the optimisation model provides access to new optimisation levers that were unavailable before. Chapter 2 clearly showed the optimisation potential lying in these new options; Chapter 5 will show in further concrete methods and examples how this potential can be exploited. To do so, however, it is vital to have an accurate model of the underlying hardware. Details on this shall be provided in the next chapter.

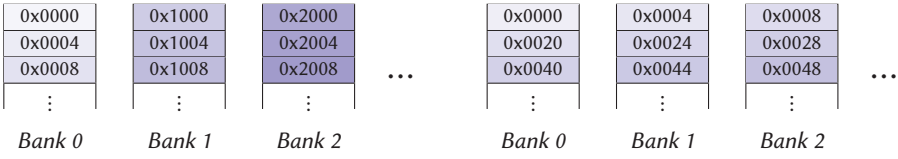
4

Modelling Interleaved Memory Platforms

Platform models, although not directly involved in processing and translation of an application code, play an important role in the compilation process. They do not have to be complicated, certainly, but if they are inaccurate, they can significantly impair program optimisation. An optimiser relies on the platform model not only for evaluating different options during decision-making, but also for assessing the quality of a produced solution. If these decisions and assessments are based on wrong assumptions, the progress of the optimisation metric in the optimisation procedure is decoupled from the real system. In short, a faulty platform model is just as bad as an inadequate optimisation algorithm.

Many different platform models have been conceived for different platform components and different optimisation algorithms. For the execution time of tasks, for instance, tools exist that perform a worst-case analysis based on detailed processor models; often, however, task execution time is simply assumed to be constant for each task and known at compile time. The cost for copying data in memory or for sending it over a NoC is typically estimated as a function of the data volume, using linear models or stair-case functions. Most components in a hardware platform can indeed accurately be described using such methods, and in many cases, these models are natural and easy to derive.

One type of components exists, however, which appears in increasingly many platforms and is far from obvious to model: *Interleaved memories*. The only models proposed for them are several decades old and were developed for entirely different hardware. Their suitability for today's hardware and optimisation processes never was evaluated. In this chapter, that gap shall be closed by adapting different models and comparing their accuracy. Practical consequences for hardware and software



(a) Contiguous mapping **(b)** Sequentially interleaved mapping

Figure 4.1: Examples for different memory address mapping types. For each of the first three banks, the addresses of the first three words stored therein are shown and optically illustrated by darker colours for higher addresses.

design will be presented with the goal of establishing an intuitive understanding of the characteristics of interleaved memory architectures. These findings were obtained in collaboration with Pratyush Kumar.

4.1 Introduction

While sharing memory between multiple processor cores brings many advantages, it also introduces the problem of *interference* amongst the cores. One solution to this is to increase the number of memory banks: The more banks exist, the lower is the expected probability of multiple cores accessing the same bank. However, apart from having scalability issues, this solution by itself cannot solve the problem. We also need to optimise the *address mapping* – the mapping of the logical addresses to physical locations.

The first such mapping that comes to mind is certainly to assign each bank its own, contiguous address space, thereby providing explicit access to the individual banks and leaving the storage concept to the programmer. This approach, which is for instance used by the Adapteva Epiphany chip, is called a *contiguous mapping* (cf. Figure 4.1a).

A different approach is *memory interleaving*. The idea of it is to have one common, big address space for all the banks, with the data being spread at fine granularity over all the banks. Among these interleaved mappings, the most popular in the embedded systems domain is *sequential interleaving*. This mapping, which is adopted for instance by the P2012 [Ben+12] or the PULP [Con+15] platforms, divides the address space into small chunks (typically of word granularity), and each subsequent chunk is assigned to a subsequent bank (cf. Figure 4.1b). Typically, this means that some of the low-order bits of the word address are assigned to the bank address.

Contiguous memory mapping configurations can be modelled with conventional memory models: Each bank is regarded as one memory module, and two simultaneous accesses to it will lead to an *access conflict*. If a conflict happens, only one access can be served, and the other one has to be stalled meanwhile.

Modelling interleaved memory, on the other hand, is far from straightforward. Model-wise, there is only one memory module, and two simultaneous accesses to it may lead to a conflict — or not. Apart from the (infeasible) approach of exactly predicting each memory access of each core at each point in time, there is no way of knowing which of both cases will occur. The conservative approach — assuming that an access conflict will always arise — leads to safe upper bounds, but would be overly pessimistic.

Probabilistic models certainly promise more accurate results for this problem. The obvious idea they rely on is that for b banks, the probability that two accesses are to the same bank is $1/b$. This simple idea, however, quickly grows in complexity when trying to model multiple simultaneous accesses. Moreover, when a core is stalled because of a pending access, it cannot request any further accesses during that time. This means that the problem actually has a state that needs to be considered.

A second question arising with this idea is that it is based on the assumption that the banks to be accessed are chosen randomly, while sequential interleaving provides a fixed, predictable mapping of the memory addresses to the banks. It is questionable whether a core's memory access patterns can really be described by a random variable under these circumstances. In fact, one consideration brought forward by advocates of sequential interleaving is that multiple cores with the same memory access patterns (e.g. iterating over the same array) will, after a small number of initial collisions, always be displaced by one access each and thus work in lock-step, causing no further conflicts. We will call this phenomenon the *synchronisation effect*. If it is significant, this will clearly undermine the validity of the random variable approach.

In this chapter, we shed light on all these questions. With the aim of quantitatively evaluating the properties and characteristics of sequentially interleaved memory systems, we study the metric of *average throughput of the memory subsystem*, i.e. the number of access requests that are served per cycle. We propose and compare two methods of predicting memory throughput: One based on the (stateless) *occupancy distribution* and one based on a Markov model. We demonstrate that with these methods, the average memory throughput of many real-world applications can be accurately predicted based only on two stochastic parameters: The probability of a core accessing the memory in any given cycle and the probability that the requested address is subsequent to the previous one.

To evaluate the accuracy of the proposed abstraction and analysis methods, we perform several experiments. Using the gem5 simulator [Bin+11], we model a multi-

banked memory module accessed by multiple ARM-based processors. We consider different applications that are common in several benchmarks. We profile these applications, compute their stochastic parameters, and then analyse them with the two methods. Across experiments, we find that the Markov-based method computes a throughput that is within 2.5 % deviation of the measured throughput. For settings containing twice as many banks as cores, i.e. where waiting accesses do not play a dominating role, the error of the occupancy distribution method is within 5 %.

The rest of the chapter is organized as follows. We define the system model and the problem statement in Section 4.3. In Section 4.4, the classic occupancy distribution and the model based on it will be presented. Section 4.5 does the same with the Markov based model. The models will then be evaluated experimentally in Section 4.6. Before, however, we review other works on the same or related issues in the next section.

4.2 Related work

Interestingly, the main line of research in the domain of interleaved memory took place as early as the late 1960s and the 1970s, at the time using magnetic core memories. With memory accesses taking considerably longer than a processor clock cycle, the common simplification was to partition time into "access cycles", assuming that any access demand arrived at the beginning of a cycle and that every core accessed the memory in each such a cycle.

Some of the analysis models (e.g. Burnett and Coffman [BC70; BC73]) were conceived for architectures which we do not regard here. Others (e.g. Skinner and Asher [SA69]) are computationally challenging and therefore only applicable to smaller clusters.

Two models turned out to be successful and were adopted in multiple publications. The first such model is that of Strecker [Str70], which assumes that every memory access will be to a random bank in each cycle, leaving out of consideration the pending accesses from previous cycles. This allows for a stateless analysis of the system and therefore for high analysability. Rabinowitz [Rab91] uses this model for further calculations.

The second popular model was first published by Bhandarkar and Fuller [BF73; Bha75]. Also Chewning, Baskett and Smith researched on this model [BS76]. The idea of it is to use a Markov description of the system and to extract results from a steady state analysis. Rau [Rau79] provides a simpler and more accurate estimation for the results.

Many recent works use these two models for their applications; however, to the best of our knowledge, no publication has altered their basic assumptions. In this

chapter, we revisit both the models and check if they can be applied to state-of-the-art multi-processor systems with fast on-chip memories. We take account of the fact that a processor will no longer access the memory in every cycle by extending the models by *memory access probabilities*. Also, we introduce a *sequential access probability* into the Markov model in order to reflect certain memory access patterns present in real applications. We believe that this is more relevant for systems with dedicated data memory than for the machines regarded in the aforementioned publications, which fetch instructions and data interchangeably from the same memory.

4.3 System Model and Problem Definition

This section explains which kind of platforms and applications we consider and how we model them. With these prerequisites, we then formalise the problem we are aiming to solve.

4.3.1 Architecture Model

As for the hardware we look into, we consider a Harvard architecture, i.e., data and program memory are separately stored and accessed. Our focus is only on mapping and throughput of the data memory. We consider a platform with c processor cores and b independently accessible SRAM banks. Each memory bank takes 1 processor cycle to serve an access. Such fast memories are becoming common in cache-less many- and multi-core platforms [Ben+12; Olo+11; Con+15]. This memory model does not consider open-row effects in other memory types such as in SDRAM. There is no contention in the communication between the processors and the banks, i.e. there exist dedicated communication links (usually fully parallel crossbars) between the processors and the banks.

4.3.2 Application Model

By an application, we refer to a thread executing on a core. Each such application is characterized by a timed trace of memory accesses. We do not consider applications which are synchronized across processors.

For the analysis, we consider an *approximate stochastic* model of the application. This model characterizes each application by only two *stochastic parameters* – the access probability p_{a} and the sequential access probability p_{seq}. The access probability p_{a} represents the probability that in any given processor cycle the application will perform a memory access (assuming that perfect and contention-free memory access is given). The sequential access probability p_{seq} represents the probability that

any two subsequent memory accesses will be to consecutive addresses in the local address space, i.e. to two subsequent memory banks. Both of these mean quantities are obtained by profiling traces of the application: p_a as the ratio of cycles with memory accesses to the total number of cycles; and p_{seq}, counting the frequency of the individual address offsets between each pair of subsequent memory accesses from the trace, as the relative frequency of one-word offsets.

4.3.3 Problem Definition

The metric of interest is the throughput of the entire memory subsystem. This metric expresses the aggregated performance of all applications in the system. We do not regard how this performance is distributed between the individual applications; the problem of arbitration is orthogonal to the problem discussed here.

For an accurate application model, the mentioned metric can be measured using memory simulators like gem5. For the approximate stochastic model mentioned above, we aim to design analysis methods which can estimate the metric. To this end, we define two random variables: A and I, where A denotes the random number of accesses requested in any given cycle, and I represents the number of banks serving accesses in any given cycle. As an example, $P(I = 3)$ denotes the probability that there are exactly 3 banks serving an access in any given cycle. Then, the average throughput is given as the expectation of I, written as $E[I]$. The distribution of I provides information about how much this throughput can vary and about the best and worst cases to expect.

This defines the problem which we will study in the next two sections: *Under the approximate stochastic model, given c, b, p_a and p_{seq}, compute the distribution of the number I of memory banks serving accesses.*

For a small number of memory accesses (p_a close to zero), it can be expected that every access request is served immediately ($I = A$ in every cycle). With p_a growing, however, the probability increases that two cores try to access the same memory bank. Since each bank can only serve one access at a time, this may lead to $I < A$. At the same time, a bank might also serve a pending access request from a previous cycle, such that $I > A$ is possible as well. While it must hold that $I \leq b$ and $I \leq c$ and steady state considerations yield that $E[I] = E[A]$, the exact distribution of I, given c, b and p_a only, is far from obvious.

These considerations are further complicated by the influence of sequentiality in memory accesses. It is clear that if all accesses are always sequential ($p_{seq} = 1$) one would – after a first phase of collisions and waiting – expect all accesses to happen in lock-step with no further collisions (synchronisation effect). Yet, for smaller values of p_{seq}, it is not clear a priori if this effect still plays an important role. In particular, only the comparison with $p_{seq} = 0$ can show if it is significant at all.

4.4 Classic occupancy based model

In this section, we will analyse memory bank access conflicts using the *classic occupancy distribution*. This necessitates the simplification from [Str70] that pending accesses from previous memory cycles are ignored.

We will first summarise the results from [Str70] (which assume $p_a = 1$) and then extend the model for different values of p_a. Finally, we will evaluate the limitations of this model.

4.4.1 The classic occupancy distribution

This section summarises the existing approach to the problem. If the number of actual memory accesses a in a cycle is known (the usual assumption is $a = c$), the throughput in the cycle follows the so-called *classic occupancy distribution*. This distribution is defined as follows. n balls are distributed at random into m urns. How many urns will contain at least one ball?

Setting $n = a$ and $m = b$, one can reuse the known results for this distribution [Fel68; JK77]:

$$\mathrm{P}^{occ}_{a,b}(I = i) = \left\{ {a \atop i} \right\} \cdot \frac{b^{\underline{i}}}{b^a} \qquad \text{with} \qquad \left\{ {a \atop i} \right\} = \frac{1}{i!} \sum_{k=0}^{i} (-1)^{i-k} \binom{i}{k} k^a, \tag{4.1}$$

where $b^{\underline{i}}$ is the falling factorial $b \cdot (b-1) \cdots (b-i+1)$ and $\left\{ {a \atop i} \right\}$ is the *Stirling number of the second kind*, sometimes also written as $\mathrm{S}(a, i)$. It represents the number of possibilities of partitioning a distinct elements into i non-empty sets.

The expected throughput is

$$\mathrm{E}^{occ}_{a,b}[I] \;=\; b \;-\; b \cdot \left(1 - \frac{1}{b}\right)^a . \tag{4.2}$$

4.4.2 Adding access probabilities

Previously, we have seen how to predict the throughput if the total number a of simultaneous accesses is known. This, however, is not the case for $p_a < 1$. To extend the model accordingly, we therefore also have to describe a.

We thus model the actual number of accesses as a random variable A, which, according to our definition follows the binomial distribution

$$\mathrm{P}(A = a) = \binom{c}{a} \cdot {p_a}^a \cdot (1 - p_a)^{c-a} .$$

Combining this with (4.1), we obtain

$$\mathrm{P}^{occ}_{c,b,p_\mathrm{a}}(I = i) = \sum_{a=0}^{c} \binom{c}{a} \cdot {p_\mathrm{a}}^{a} \cdot (1 - p_\mathrm{a})^{c-a} \cdot \left\{ {a \atop i} \right\} \cdot \frac{b^{\underline{i}}}{b^a}. \tag{4.3}$$

The expected throughput for this distribution is

$$\mathrm{E}^{occ}_{c,b,p_\mathrm{a}}[I] = b - b \cdot \left(1 - \frac{p_\mathrm{a}}{b}\right)^c. \tag{4.4}$$

This can be shown in a similar way as (4.2) was shown in [JK77]. For each bank, the probability of being accessed by one given core is $p_\mathrm{a} \cdot \frac{1}{b}$, and thus the probability *not* to be accessed by *any* core is

$$\left(1 - \frac{p_\mathrm{a}}{b}\right)^c.$$

This is also the expected number of banks *out of this one bank* which are not accessed by any core. As the expected values for the different banks can be just added up, multiplication with b and subtraction from b (to get the number of banks that are accessed) yields the result above.

Since usually $p_\mathrm{a} \ll b$, we can introduce the following simplification. If the ratio $r = \frac{c}{b}$ of cores and banks is constant, the approximation

$$\mathrm{E}^{occ}_{c,b,p_\mathrm{a}}[I] = b - b \cdot \left(1 - \frac{p_\mathrm{a} \cdot r}{c}\right)^c \approx b - b \cdot e^{-p_\mathrm{a} \cdot r} \tag{4.5}$$

holds, with $\left(1 - \frac{\lambda}{x}\right)^x \approx e^{-\lambda}$ for $\lambda \ll x$, e being the Euler number.

4.4.3 Limitations of the model

While the model allows interesting insights, it also has some shortcomings. Firstly, since it is stateless, sequential access patterns of the applications (i.e. $p_\mathrm{seq} \neq 0$) cannot be taken into account. Secondly, as discussed earlier, it ignores the fact that accesses that cannot be immediately served are served in subsequent cycles, then interfering with new accesses. As long as the number of such accesses having to wait is small, the occupancy distribution can therefore be regarded as an approximation for the real system. With a higher number of conflicts, however, the numbers can be expected to deviate substantially from the real values.

4.5 Markov model

In this section, we will analyse memory bank access conflicts using a Markov model. This is more calculation intensive, but also more accurate than the occupancy model.

We will again start with the known model described in [BF73], i.e. $p_{\mathrm{a}} = 1$ and $p_{\mathrm{seq}} = 0$. We will then extend the model to include sequential accesses and finally, we will introduce the memory access probability again.

4.5.1 Known model

In the following, it will be explained how Markov model steady states are used in general to calculate certain distributions and how this is done in [BF73] for modelling interleaved memory throughput.

In general, a Markov model consists of a set of k states $S = \{\boldsymbol{s}^1, \ldots, \boldsymbol{s}^k\}$ and a transition probability function $f : S \times S \longrightarrow [0, 1]$. This function $f(\boldsymbol{s}, \boldsymbol{t}) = \mathrm{P}(\boldsymbol{s} \longrightarrow \boldsymbol{t})$ describes the probability of a transition from state $\boldsymbol{s}$ to $\boldsymbol{t}$.

One can define a *transition matrix* $\boldsymbol{T} \in \mathbb{R}^{k \times k}$ with $T_{i,j} = P(\boldsymbol{s}^j \longrightarrow \boldsymbol{s}^i)$. If a vector $\boldsymbol{v} \in \mathbb{R}^k$ contains the probabilities v_j for the system to be in a state $\boldsymbol{s}^j$ at a certain point, $\boldsymbol{T} \cdot \boldsymbol{v}$ will contain the state probabilities after one state transition. After a large number of rounds, the system has reached the *steady state* described by the probability vector $\boldsymbol{\sigma} \in \mathbb{R}^k$. It holds that

$$\boldsymbol{T} \cdot \boldsymbol{\sigma} = \boldsymbol{\sigma}. \tag{4.6}$$

If the transition probabilities are known, one can thus calculate the steady state probabilities by solving the mentioned eigenvector problem.

Since one is usually not interested in the probabilities of certain states but rather in the probabilities of certain quantities associated with the states, one will define a quantity function $q : S \longrightarrow \mathbb{R}$ mapping the states to the quantities. Then one can extract the probability of a certain value q^* as the probability sum of all the concerned states

$$\mathrm{P}(Q = q^*) = \sum_{j \in J} \sigma_j \quad \text{with} \quad J = \left\{ j \;\middle|\; q(\boldsymbol{s}^j) = q^* \right\}. \tag{4.7}$$

To apply the tool of the Markov steady state to the interleaved memory throughput problem, i.e., to calculate $\mathrm{P}^{mkv}_{c,b,p_{\mathrm{a}}=1,p_{\mathrm{seq}}=0}(I = i)$ and $\mathrm{E}^{mkv}_{c,b,p_{\mathrm{a}}=1,p_{\mathrm{seq}}=0}[I]$, one needs to define the set of states, derive the transition probabilities and provide a mapping from the states to the throughput. In the following, it will be shown how this is done in [BF73]. Accordingly, it is assumed that $a = c$ and that $p_{\mathrm{seq}} = 0$.

State set. To provide better extensibility for later problems, we encode the states differently here than it is done in [BF73]. The states themselves, however, are still the same. The state vectors we use have the form

$$\boldsymbol{s} = (s_1, \ldots, s_c),$$

where s_n is the number of banks having exactly n accesses in their queue. For example, for $c = 4$ cores, a state $\boldsymbol{s}^{\text{example}} = (2, 1, 0, 0)^\top$ would mean that two banks have enqueued one access each and one bank has enqueued two accesses. In this example, there are no banks with three or four accesses enqueued. Summing up the queue lengths of all banks, one can see that the state set is

$$S = \left\{ \boldsymbol{s} \in \mathbb{N}_0^{\,c} \;\middle|\; \sum_{j=1}^{c} j \cdot s_j = a \right\}. \tag{4.8}$$

We define $s_0 = b - \sum_{j=1}^{c} s_j$, the number of idle memory banks for a state $\boldsymbol{s}$.

The number of states is equal to the number of partitions of a: As an example, for $a = 16$, there are $P(a) = 273$ different states.

Transition probabilities. The probability of attaining a state $\boldsymbol{s}$ out of the initial state $\boldsymbol{s}^0$ (with all banks idle) can be calculated directly according to this formula [Mis38]:

$$\mathrm{P}(\boldsymbol{s}^0 \longrightarrow \boldsymbol{s}) = \frac{a! \cdot b!}{b^a \cdot \prod_{j=0}^{c} [(j!)^{s_j} \cdot s_j!]}. \tag{4.9}$$

However, for attaining a target state $\boldsymbol{t}$ from an arbitrary state $\boldsymbol{s}$, there are multiple different access redistribution possibilities, which makes it hard to come up with a closed formula for calculating the associated probability. (We use the term "access redistributions" to model the fact that as soon as an access is served, the concerned processor will make a new access request to a new bank.) The problem can be solved by first determining the "stripped" state

$$\boldsymbol{s}' = (s_2, s_3, \ldots, s_c, 0), \tag{4.10}$$

in which one access request has been removed from each bank's queue, and by then enumerating the possibilities for distributing new access requests such that $\boldsymbol{t}$ is attained. One algorithm for this calculation is described in [BF73; Bha75].

State throughput mapping. For a state $\boldsymbol{s}$, the associated throughput is given by

$$b - s_0 \,.$$

4.5.2 Adding sequential access patterns

In order to analyse the access synchronisation effect for sequentially interleaved memories, which was mentioned in the introduction, we now extend the model by a *sequential access probability* p_{seq}. We assume a (circular) order of the memory banks and that upon each access, with a probability of p_{seq} the memory bank assigned to the accessed address will not be random but instead the bank next to the previously accessed one.

Clearly, an exact solution to the problem would need distinction of the individual banks, since their relative position to each other has now become relevant. Such a distinction would, however, blow up the number of states to $\binom{a+b-1}{b-1}$, e.g. $1.5 \cdot 10^{12}$ states for the configuration $a = 16$, $b = 32$ (P2012), which is computationally difficult. We thus stick to the states introduced in the last section and simplifyingly assume that all combinations of relative bank positionings are equiprobable.

The redistribution of the accesses in this model can be regarded as consisting of two distinct steps: First a sequential distribution (access requests to the "next" banks) of n_{seq} balls and then a random distribution of n_{rnd} accesses, with $n_{\text{seq}} + n_{\text{rnd}} = n_{\text{redist}}$ the total number of accesses redistributed. (4.6) can then be rewritten as

$$\boldsymbol{T}_{\text{rnd}} \cdot \left(\boldsymbol{T}_{\text{seq}} \cdot \boldsymbol{\sigma} \right) = \left(\boldsymbol{T}_{\text{rnd}} \cdot \boldsymbol{T}_{\text{seq}} \right) \cdot \boldsymbol{\sigma} = \boldsymbol{\sigma}, \tag{4.11}$$

defining a new transition matrix $\boldsymbol{T} = \boldsymbol{T}_{\text{rnd}} \cdot \boldsymbol{T}_{\text{seq}}$.

Note that there has to be a higher number of "intermediary" states between applying $\boldsymbol{T}_{\text{seq}}$ and $\boldsymbol{T}_{\text{rnd}}$, since a subset of the accesses has not yet been redistributed and thus (4.8) must be weakened to

$$\sum_{j=1}^{c} j \cdot s_j \leq a, \tag{4.12}$$

increasing the number of intermediary states to $\sum_{j=0}^{a} P(a)$, e.g. 915 states for $a = 16$, with $\boldsymbol{T}_{\text{rnd}} \in \mathbb{R}^{273 \times 915}$ and $\boldsymbol{T}_{\text{seq}} \in \mathbb{R}^{915 \times 273}$.

The algorithm from [Bha75] mentioned in the previous section can also be used to calculate $\boldsymbol{T}_{\text{rnd}}$. Note that $\boldsymbol{T}_{\text{rnd}}$ does not depend on p_{seq}, it can thus be reused when testing different values of p_{seq}.

$\boldsymbol{T}_{\text{seq}}$ can be calculated as follows. For a transition from a state $\boldsymbol{s}$ to an intermediary target state $\boldsymbol{t}$, one calculates the "stripped state" $\boldsymbol{s'}$ according to (4.10) and then the *upgrade vector* $\boldsymbol{u} \in \mathbb{Z}^{0\ldots c}$ as

$$u_j = \sum_{v=0}^{j} u_v^* \quad \text{with} \quad \boldsymbol{u}^* = \boldsymbol{s'} - \boldsymbol{t},\ u_0^* = s'_0 - t_0. \tag{4.13}$$

The idea is that if in $\boldsymbol{s}'$, s'_0 banks are idle and in $\boldsymbol{t}$, only t_0 banks should be idle, $u_0 = s'_0 - t_0$ banks have to be *upgraded* by assigning one access to them. Now there are $s'_1 + u_0$ banks with a queue length of exactly one; if only t_1 of these are required, $(s'_1 + u_0) - t_1 = u_1$ of them have to be upgraded again etc. If any element of $\boldsymbol{u}$ is negative, a transition is not possible, i.e. the transition probability is zero. Also, the transition is only possible if $\forall j,\ s'_j \geq u_j$, since no bank can receive more than one access through sequential redistribution.

For each upgrade class j, there are now u_j out of s'_j banks that must be upgraded; in total, there are $n_{\text{seq}} = \sum_{j=1}^{c}(t_j - s'_j)$ accesses to be sequentially distributed and thus n_{seq} out of b banks to be upgraded. Together with the (binomial) probability that indeed n_{seq} accesses are redistributed sequentially, the joint probability for the sequential transition from $\boldsymbol{s}$ to $\boldsymbol{t}$ is equal to

$$\mathrm{P}(\boldsymbol{s} \xrightarrow{\text{seq}} \boldsymbol{t}) = \binom{n_{\text{redist}}}{n_{\text{seq}}} \cdot {p_{\text{seq}}}^{n_{\text{seq}}} \cdot (1 - p_{\text{seq}})^{n_{\text{rnd}}} \cdot \frac{\prod_{j=0}^{c} \binom{s'_j}{u_j}}{\binom{b}{n_{\text{seq}}}}. \tag{4.14}$$

4.5.3 Adding access probabilities

In this section, we will remove the previous assumption $a = c$ and introduce the memory access probability p_{a} for each core.

The solution to this problem is a combination of different approaches already shown in this chapter. It is clear that dropping the assumption $a = c$ increases the number of possible states. In fact, since each state $\boldsymbol{s}$ has to fulfil the requirement

$$\sum_{i=1}^{c} i \cdot s_i \leq c, \tag{4.15}$$

which is similar to (4.12), the new states are identical to the intermediary states from the last section.

Like in Section 4.4.2, the different possible values of a and thus of n_{redist} for the target state have to be considered separately:

$$\left(\sum_{a=0}^{c} {\boldsymbol{T}_{\text{rnd}}}^{A=a} \cdot {\boldsymbol{T}_{\text{seq}}}^{A=a}\right) \cdot \boldsymbol{\sigma} = \boldsymbol{\sigma}.$$

For each possible transition, the probability that $A = a$ must be included as well. This can be done by multiplying it into ${\boldsymbol{T}_{\text{seq}}}^{A=a}$, i.e.

$${T_{\text{seq}}}^{A=a}_{i,j} = \binom{n_{\text{avail}}}{n_{\text{idle}}} \cdot {p_{\text{a}}}^{(n_{\text{avail}} - n_{\text{idle}})} \cdot (1 - p_{\text{a}})^{n_{\text{idle}}} \cdot \mathrm{P}(\boldsymbol{s}^j \xrightarrow{\text{seq}} \boldsymbol{s}^i),$$

with $n_{\text{avail}} = b - s_0$ the number of cores available for requesting new accesses and $n_{\text{idle}} = c - a$ the number of cores that are not going to request an access.

For calculating ${T_{\text{rnd}}}^{A=a}$, the algorithm from [Bha75] mentioned in the previous sections can be used again.

4.6 Experimental evaluation

In this section we compare the estimates based on the analysis of the approximate probabilistic model with the simulated results of different sequentially interleaved memory configurations with real benchmark applications. First we describe the setup and then the obtained results. We also draw conclusions from the models and give hints for system designers.

4.6.1 Experimental setup

All benchmarks were compiled for and run on the gem5 [Bin+11] ARM simulator. For every core (we simulated $c = 16$), a separate memory access trace was created, recording the accessed addresses as well as the pauses between the accesses. Wherever possible, program parameters or inputs were varied in order to create different access patterns.

A bank access and collision simulation was then carried out on a custom simulator, replaying each of the traces with short, but random initial delays. The memory was configured either with $b = 16$ or $b = 32$ banks (the latter is the configuration of P2012) with round robin arbitration.

A selection of benchmarks from the MiBench [Gut+01] embedded benchmark suite was used for the experiments. The GSM, FFT, blowfish, string search and JPEG examples were chosen to obtain a high diversity in behaviour.

From the traces, p_{a} and p_{seq} were extracted and used as parameters for the occupancy and for the Markov model.

4.6.2 Accuracy of the occupancy model

The occupancy model makes drastic simplifications, especially when a high number of waiting accesses is expected in a system. Figure 4.2 on the following page compares the simulated throughput of the Blowfish benchmark for $c = 16$ and $1 \leq b \leq 64$ with the predicted value from the occupancy model. For a small number of banks, it is likely that each bank serves exactly one access per cycle and thus the throughput is accurately estimated. For a sufficiently large number of banks, the number of waiting accesses is small and thus the accuracy of the model is good as well. As

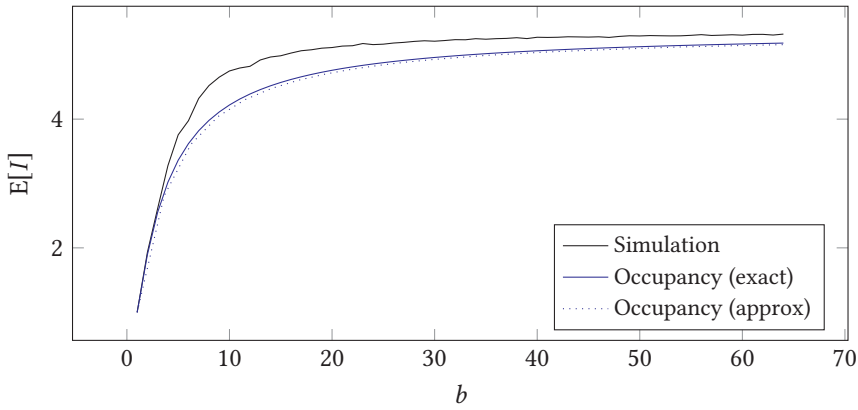


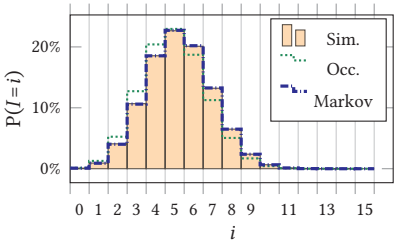
Figure 4.2: Accuracy of the occupancy model. The simulated average throughput for the Blowfish benchmark ($p_a = 0.34$) and the expected throughput according to the occupancy model are plotted against the number of memory banks for $c = 16$. The approximation (4.5) on page 122 is plotted as well. The maximum relative error is of 12.0% for $b = 8$.

expected, the deviation from the simulation is highest in between, i.e. around the value $b = p_a \cdot c$. The maximum error observed is about 12%. However, the trend of the throughput is still captured by the occupancy model.

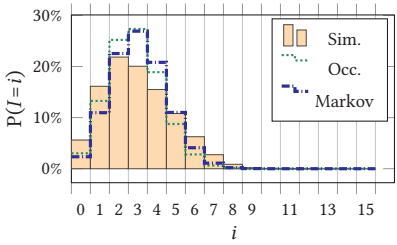
4.6.3 Benchmark Results

Figure 4.3 shows the results of the simulations for all the benchmark applications. In general, both the occupancy and the Markov models reflect well the trends that emerge in the simulation. As expected, the Markov model fits particularly well for small values of p_{seq} and for $p_{seq} \longrightarrow 1$. In these settings, the Markov results are nearly congruent with the simulation results. For the string search example, the deviations can be explained with the inexactness resulting from the simplified sequentiality model as described in Section 4.5.2.

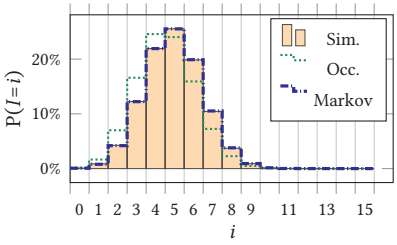
Only in the FFT example, the simulation shows a clearly large variance compared to the Markov model. We believe this is because of the specific access pattern of the FFT algorithm. The algorithm recursively splits arrays with a dimension of a power of two into two sub-arrays. It switches accesses between these arrays and thus produces multiple accesses in a row on the same memory bank. This is a well-documented effect (cf., e.g., [Rau91]) in which multiple processors with similar memory access patterns frequently access the same banks, thus repeatedly block-



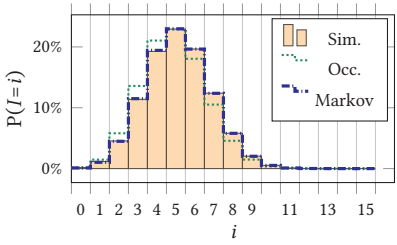
(a) Blowfish. b=32, p_a=0.34, p_{seq}=0.27.
$\bar{I}^{sim}$=5.23, $E^{mkv}[I]$=5.24, $E^{occ}[I]$=4.98.



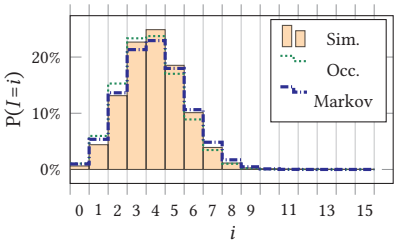
(b) FFT. b=16, p_a=0.20, p_{seq}=0.49.
$\bar{I}^{sim}$=3.02, $E^{mkv}[I]$=3.09, $E^{occ}[I]$=2.87.



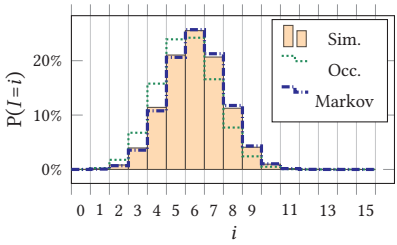
(c) GSM. b=16, p_a=0.33, p_{seq}=0.07.
$\bar{I}^{sim}$=4.93, $E^{mkv}[I]$=4.94, $E^{occ}[I]$=4.53.



(d) GSM. b=32, p_a=0.33, p_{seq}=0.07.
$\bar{I}^{sim}$=5.12, $E^{mkv}[I]$=5.13, $E^{occ}[I]$=4.89.



(e) String search. b=32, p_a=0.25, p_{seq}=0.86.
$\bar{I}^{sim}$=3.94, $E^{mkv}[I]$=3.95, $E^{occ}[I]$=3.75.



(f) JPEG. b=16, p_a=0.42, p_{seq}=0.14.
$\bar{I}^{sim}$=6.01, $E^{mkv}[I]$=6.07, $E^{occ}[I]$=5.57.

Figure 4.3: Simulation results for different benchmarks. The simulated systems have c=16 cores and either b=16 or b=32 memory banks. For each benchmark, the simulated throughput frequencies are plotted together with the predictions from the Markov and the occupancy model. Model parameters like b, p_a and p_{seq} are given as well as the simulated average throughput $\bar{I}^{sim}$ and the expected throughput according to the Markov and the occupancy models.

ing each other. A similar effect is observed when multiplying matrices with powers of two as their dimensions. Customized programming is required to avoid such instances on sequentially interleaved memories. [Rau91] proposes *pseudo-random* interleaving, which tries to avoid access collision patterns by quasi-randomising the assignment of memory addresses to physical banks. The FFT benchmark was the worst fit from all the benchmarks we tested.

In summary, the simulations show that for several benchmark applications the simple probabilistic model of an access probability and a sequential access probability accurately estimates the throughput of an interleaved memory system.

4.6.4 Conclusions from the occupancy model

The advantage of the occupancy model is the analytical form of its result. We now draw some conclusions from these results, which hold only under the conditions as described in Section 4.6.2. In (4.5), the expected throughput of a system was approximated as $\mathrm{E}[I] \approx b - b \cdot e^{-p_\mathrm{a} \cdot r}$ with $r = \frac{c}{b}$. This simplification yields the following conclusions.

- As long as the ratio of banks and cores is constant, a system can be arbitrarily scaled without changing the throughput expectation per bank or per core.
- The throughput converges exponentially with the product $p_\mathrm{a} \cdot r$ to the maximum value b. Note that $p_\mathrm{a} \cdot r = \frac{1}{b} \cdot \mathrm{E}[A]$ is the expected number of accesses per memory bank.
- For $p_\mathrm{a} \cdot r < 0.3$, the throughput can be regarded as growing approximately linearly with p_a.

4.6.5 Application example: System design

The occupancy model can also be used to answer simple design questions. Consider the comparison between two different systems, each with $c = 16$ cores and $b = 32$ banks. System 1 uses complete interleaving over all 32 banks. System 2 uses interleaving for 16 banks and provides each core with one "private" memory bank. Let now p_priv be the probability that an access goes to the private memory bank on System 2. The question is which setting is better, and for which values of p_priv.

Since every access to the private memory is immediately served, the expected throughput for System 2 is

$$\mathrm{E}^{occ}_{\mathrm{System2}}[I] = c \cdot p_\mathrm{a} \cdot p_\mathrm{priv} + \mathrm{E}^{occ}_{c, \frac{b}{2}, p_\mathrm{a} \cdot (1 - p_\mathrm{priv})}[I],$$

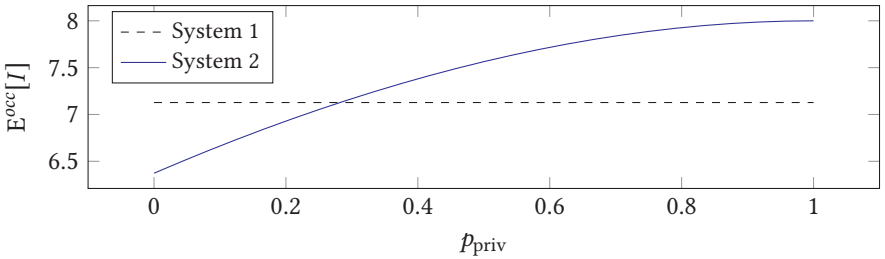


Figure 4.4: Comparison of two systems: System 1 with fully interleaved memory and System 2 with partially interleaved, partially private memory.

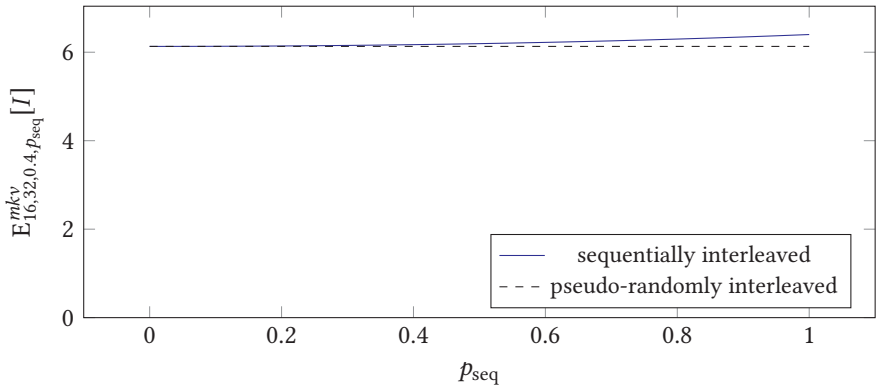


Figure 4.5: Evaluation of the synchronisation effect in sequentially interleaved and pseudo-randomly interleaved memory systems. The throughput according to the Markov model is plotted as a function of p_{seq} for $c = 16,\ b = 32,\ p_{\mathrm{a}} = 0.4$.

whereas (4.2) can be used for System 1. Both expected throughputs are shown for $p_{\mathrm{a}} = 0.5$ as a function of p_{priv} in Figure 4.4. For System 1, the throughput is constant as p_{priv} does not play a role there. For System 2, the worst case is $p_{\mathrm{priv}} = 0$ with effectively only 16 banks, the private banks not being used. The throughput increases with p_{priv} increasing, the maximum value being attained for $p_{\mathrm{priv}} = 1$, in which case no conflicts occur as each core only uses its private memory. Comparing both systems, System 2 performs better for $p_{\mathrm{priv}} > 0.2875$. Many other such comparisons can be made with the analytical results from the occupancy model.

4.6.6 Discussion of the synchronisation effect

The support for sequential access patterns in the Markov model allows to analyse the synchronisation effect described earlier. For this, Figure 4.5 on the previous page shows a plot of the throughput of a sequentially interleaved memory system with $c = 16$ and $b = 32$ as a function of p_{seq}. The memory access probability has been chosen rather high with $p_{\text{a}} = 0.4$. As a comparison, we show the throughput of a system with pseudo-random interleaving as hinted above and discussed in [Rau91]. In these systems, the assignment of word addresses to physical memory can be regarded as random and therefore, their throughput is equal to sequential interleaving with $p_{\text{seq}} = 0$. Two things can be seen from the plot: (i) For $p_{\text{seq}} < 0.4$, the synchronisation effect is insignificant. (ii) Even the speed-up from $p_{\text{seq}} = 0$ to $p_{\text{seq}} = 1$ is less than 5 % in this system.

This small benefit is opposed by the danger of collision patterns like with the FFT benchmark as described earlier. Hence, software designers who appreciate code compatibility or who do not know their target platform in detail, also on sequentially interleaved memory systems may want to randomise their applications' memory accesses. This could for example be achieved by running multiple different types of tasks on a cluster instead of having a homogeneous set of threads.

4.7 Concluding remarks

In this chapter, interferences in sequentially interleaved memory systems have been theoretically analysed using a simple probabilistic abstraction. Two models — a fast occupancy distribution based model and a more accurate Markov model — have been presented and discussed. General principles and character traits of interleaved memory systems have been deduced theoretically from the models. Finally, the models have been validated in an extensive set of simulations.

Apart from many concrete results such as the small impact of the synchronisation effect, the most important result of this study is that interleaved memory systems can be adequately described by a simple probabilistic abstraction which only knows the memory access probability of the processes and the probability of accessing consecutive memory banks. This gives room for extensive abstract analyses as well as for lightweight and calculation-efficient, yet accurate platform models for systems with interleaved memory. The next chapter will show how these findings can be taken advantage of in a code optimisation framework, but also how interleaved memory actually performs compared to contiguous mappings.

5

Optimisation models and algorithms

Optimisation models are often seen as program representations used internally by an optimiser. This is, while correct, only one part of a bigger picture. Optimisation models are abstractions which allow systematic analysis of a code: The structure of a program specification is not only determined by the application to be implemented but typically also influenced by the programmer's needs and personal taste. Such a structure, however, is typically unrelated to the requirements of generic program optimisation. An optimisation model now brings the code into a canonical form and defines a metric to be optimised. This provides a lever for handling the optimisation problem, and, as a result, a base for an optimisation algorithm.

These considerations show that optimisation models are a crucial step towards good optimisation algorithms. In some cases, the right model directly leads to an optimal solution (remember the task partitioning algorithm for transient systems in Section 3.6.2); in other cases, it is more difficult. A good model should seek the balance between sufficient abstraction and thus tractability on the one hand and an accurate representation of optimisation potential on the other. Also, generality and adaptability decisively add to a model's use.

This chapter will present two optimisation problems and discuss related optimisation models and strategies. First, multicore platforms with multi-bank memories are revisited. Unlike in the last chapter, which concentrated on memory interleaving, a comparison is drawn between the latter and contiguous memory mapping. In the case of the latter, the problem arises how to organise the data and how to distribute it between the different banks. An optimisation model and algorithm for this task is devised which allows a fair comparison of the two different memory mapping strategies.

In a second step, the generic optimisation problem is defined based on the Ladybirds model set. Starting from the Ladybirds optimisation model, a more detailed optimisation model is conceived for the special needs of different optimisation goals. These different optimisation subproblems are then described in more detail and requirements are discussed for how to progress towards the aim of optimising any Ladybirds program for any platform (with sufficient resources) and achieving a memory and transport efficient implementation.

5.1 Minimising Access Conflicts on Shared Multi-Bank Memory

The last chapter introduced the notions of interleaved and contiguous memory address mapping. In this section, both approaches shall be compared — with theoretical considerations and experimentally on the Kalray MPPA-256 platform. In the case of contiguous mapping, data must be distributed between the different banks. An optimisation model is proposed as well as a related algorithm, based on graph colouring techniques, to automatically perform this distribution with the goal of minimising access collisions and delays. These results were obtained in a collaboration with Georgia Giannopoulou and Matthias Baer.

5.1.1 Introduction

As it has been shown in the last chapter, memory interleaving has several advantages: All the cores see a uniform memory space, in which the programmer can accommodate any code or data without having to think of its particular placement. The bank access patterns of the cores and the resulting collision behaviour can be accurately described by random variables and probabilistic distributions derived from them. As a result, the memory space can be treated like a single bank and standard compilers and linkers can be used while still achieving good performance.

At the same time, contiguous mapping has its advantages as well. Firstly, the performance of many low-latency implementations of interleaved mapping is vulnerable to memory access patterns which repeatedly access the same bank (e.g. operations on matrices having dimensions of powers of two). This can be avoided by contiguous mapping with an appropriate data placement. Secondly, worst-case delays are harder to determine on interleaved memory systems, and calculated bounds are usually far from tight. This is why recent publications in that domain focus on contiguous mapping, see e.g. [Bec+16; Car+14; VY15; NYP16; Per+16].

When average case performance is more important than worst case execution, there is no such clear preference. It would therefore be desirable to have a direct comparison between both approaches, which up to now does not exist. However, to

perform such a comparison, one would also need to find a solution to the question of data placement in the case of contiguous mapping, i.e., on which bank which data should be placed. Currently, this has to be decided manually by the programmer, which constitutes a significant obstacle to the application of contiguous mapping. Finding a way of automatically partitioning data between the memory banks would thus not only allow a comparison between the different mapping approaches, but it would also demonstrate the practical feasibility of implementing applications on multi-bank platforms with contiguous address mapping.

Several approaches and algorithms have been proposed to tackle the data placement issue for single-processor multi-bank systems, in particular VLIWs and DSPs. Unfortunately, the corresponding optimisation problem significantly differs from the multi-core problem, since compiler backends for individual cores can easily detect simultaneous memory accesses on instruction level. On multi-core systems, in contrast, the individual cores are independent and only loosely synchronised. Access conflicts happen non-deterministically. As a result, existing single-core solutions cannot be directly applied to multi-processor systems.

In this section, we try to close the aforementioned gaps with a two-fold contribution: Firstly, we propose a heuristic algorithm to automate buffer placement such that every buffer is assigned a memory bank for contiguous mapping. The proposed approach was designed to find mappings that minimise access conflicts as well as other delays, and thus the application runtime. Secondly, with the help of and as an evaluation for the newly introduced algorithm, we compare the average case performance of contiguous and interleaved mapping, theoretically and experimentally, on the Kalray MPPA platform. This platform is very well-suited for this comparison, since it allows the programmer to choose between interleaved and contiguous mapping. To obtain meaningful results in a wider context, we have conducted dedicated, synthetic experiments on the MPPA and we have run real world benchmarks.

With a memory bank assignment optimised using the proposed algorithm, 96.7% of the tested benchmark configurations perform at least as well as with interleaved memory mapping. In 54.5% of the cases, the runtime is even significantly shorter.

In Section 5.1.3, we describe our optimisation and platform models. On this basis, we formally define the bank assignment problem for the contiguous memory mapping scenario. We then introduce an algorithm for solving this generic bank assignment problem in Section 5.1.4. Section 5.1.5 gives detailed information about the MPPA and its memory architecture. Based on the results of synthetic benchmarks revealing the platform's characteristics, we show how the generic bank assignment algorithm can be adapted to this particular platform. Section 5.1.6 finally shows the results of the experimental comparison of contiguous and interleaved mapping on the Kalray MPPA, using the proposed bank assignment algorithm for the case of contiguous memory address mapping.

5.1.2 Related Work

Various works address the problem of mapping data to memory banks for single-processor systems. In particular, there are many works on VLIW (in particular DSP) systems with dual-bank memories, e.g. by Saghir et al. [SCL96], Leupers and Kotte [LK01], Cho et al. [CPW02], Sipkova [Sip03], Ko and Bhattacharyya [KB03] and Murray and Franke [MF08]. All of these works try to enable two simultaneous memory accesses by mapping the corresponding variables (or arrays) to different memory banks. While [SCL96], [LK01] and [CPW02] propose implementations as compiler backends, [Sip03], [KB03] and [MF08] analyse the code on higher levels.

Zhang et al. [Zha+11] and Soto et al. [Sot+13] extend this optimisation to a variable number of memory banks. Conversely, Shyam and Govindarajan [SG07] try to minimise energy consumption in such a system by assigning the banks such that some of them can be brought to sleep mode as often as possible. [Zha+11] also supports this optimisation.

Most of the works discussed until now make use of so-called *conflict* or *interference graphs*, as we do in this section. The typical solution approaches are greedy algorithms, other heuristics, or integer linear programming. [Sot+13] and [MF08] treat the assignment task as a graph colouring problem, like it is done in this section. However, all these works consider single processor systems and rely on conflict analysis techniques that are not applicable to multi-processor systems, as discussed earlier.

Kim and Kim [KK07] propose a method to improve performance of multiple DRAM banks connected to a single processor. Their approach is to maximise spatial access locality within each memory bank, thus avoiding costly row opening operations required on this memory architecture. In this section, we consider platforms with on-chip SRAMs, on which access locality has no influence.

In the area of multi-core systems, [Kim+10] and [Mi+10] propose heuristics for mapping data of different application threads to DRAM banks to reduce the average thread execution times. Kim et al. [Kim+10] present a compiler approach targeting coarse grain reconfigurable architectures. Our approach is more general, since it is applicable to any homogeneous shared-memory architecture. Mi et al. [Mi+10] introduce a software/hardware scheme for static DRAM bank partitioning. Purely hardware-based solutions include the works of Reineke et al. [Rei+11] and Wu et al. [WKP13], which rely on DRAM controllers to implement bank privatisation schemes. Such approaches require special hardware, while we propose compiler techniques that are applicable to commercial off-the-shelf platforms.

Other works implement software approaches to completely eliminate bank-level conflicts. Liu et al. [Liu+12] implement a custom page-colouring algorithm inside the memory management of the Linux kernel for this purpose. Jeong et al. [Jeo+12]

propose a combination of bank partitioning and memory sub-ranking, implemented through an extension of the OS physical frame allocation algorithm. Yun et al. [Yun+14] implement a DRAM bank-aware memory allocator to allocate memory pages of different applications to private banks. Chandru et al. [CM16] implement a user space bank-aware and controller-aware allocator, which enables binding a core to a specific bank and controller in a cluster-based architecture. Pan et al. [PGM16] enable frame allocation on thread-specific cache, memory controller and memory bank combinations through an according modification to the Linux kernel. All these works try to cleanly (or at least largely) separate the banks accessed by the different applications or cores. In contrast, we assume that bank sharing is indispensable for communication between cores and that each core will thus need regular access to multiple banks.

Closer to our approach lies the approach of Giannopoulou et al. [Gia+14], which also tries to minimise bank conflicts through optimised data-to-bank mapping. However, their method aims at minimising the worst-case execution time of real-time applications and is bound to specific scheduling policies, whereas we address the average-case execution time of a wider class of applications. Rihani et al. [Rih+16] propose a solution for process networks (i.e. single-producer single-consumer) by assigning each consumer a bank for all its input FIFOs. This works well for the selected application model, which, however, necessitates abundant copying of data and is therefore inherently inefficient on shared memory platforms(cf. Section 2.3). Finally, Goens et al. [Goe+16] employ a buffer allocation approach similar to the one shown in this section, but based on integer linear programming. While they use a more general platform model and a more detailed application model, this comes at the price of longer optimisation time. Also, their model does not cover the particularities of the MPPA platform (cf. Section 5.1.5.2).

5.1.3 Considered Memory Bank Assignment Problem

As discussed earlier, in the case of contiguous memory mapping, each buffer or piece of data accessed by the application must be assigned to a memory bank on which it will reside. This assignment should be done in such a way that interferences between different threads are minimised. This section gives a detailed definition of the problem, showing how we model the application and the target platform. All models are kept as generic as possible, and an algorithm to solve the assignment problem for the generic case will be given in the next section. Later sections will then discuss how the generic models may or may need to be adapted to concrete target platforms such as the Kalray MPPA.

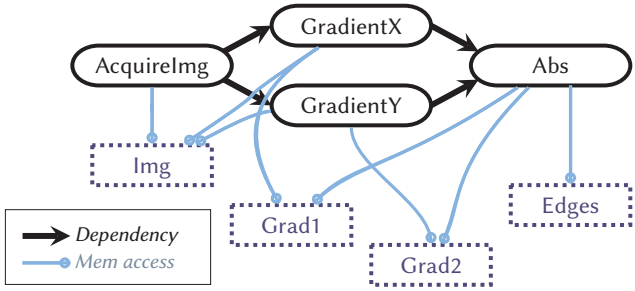


Figure 5.1: Representation of a simple image edge detection algorithm in the application model from Section 5.1.3.1. It reads an image into memory, numerically computes its gradients in x and y direction and then computes the edge intensity as the euclidean norm of both. The upper part shows the tasks, the lower part the buffers.

5.1.3.1 Application Model

Figure 5.1 illustrates how we model the applications that are executed on a cluster.

Definition 5.1. An **application** is given by a tuple (T, D, B, a) consisting of a set $T = \{t_1, \ldots, t_{n_T}\}$ of **tasks**, a set $D \subset T \times T$ of **dependencies**, a set $B = \{b_1, \ldots, b_{n_B}\}$ of **buffers** and an **access function** $a : T \longrightarrow \mathcal{P}(B)$.

Each task t executes exactly once and accesses only the buffers contained in $a(t) \subseteq B$. Each dependency $(t_i, t_j) \in D$ enforces that t_j can only start once t_i has finished. Each buffer $b \in B$ has a distinct **size** $|b|$, given as a number of bytes.

This model is a simplification of the Ladybirds optimisation model in which the packet types (`in`, `out` etc.) have been abstracted away as they are not relevant for this optimisation. It is assumed that we are working on a buffer dependency graph, i.e., optimisation steps like buffer allocation have already been performed (cf. Section 3.4). Also note that no assumptions are made as to how often or in what pattern a task t accesses the buffers given by $a(t)$.

5.1.3.2 Platform Model

Figure 5.2 gives an (abstract) example of the kind of target platform considered in this work. It can be formalised in the following, even more generic model.

Definition 5.2. A **platform** is a tuple (C, M), with $C = \{c_1, \ldots, c_{n_C}\}$ a set of cores or processing elements (PEs) that share access to a set of memory banks $M = \{m_1, \ldots, m_{n_M}\}$. Each memory bank $m \in M$ has a size $|m|$, given as a number of bytes.

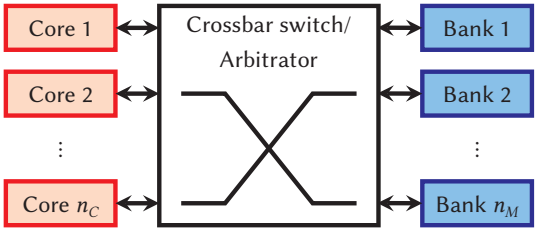


Figure 5.2: Exemplified illustration of the generic type of platform considered in this work

We make the following assumptions about a platform:

- The banks are configured for contiguous address mapping.
- Each PE takes the same time to access each bank.
- All banks can be accessed in parallel, but each bank can only be accessed by one PE at a time, e.g. by a crossbar communication structure between PEs and banks.
- The arbitration between bank accesses is fair, e.g. Round Robin.
- The PEs do not perform task preemption.

This model fits to many existing multi-core multi-bank platforms, for instance the NXP LPC family or the Kalray MPPA (details will be given later). It would fit to the P2012 [Ben+12] and the PULP [Con+15] architectures as well if they supported a contiguous memory address layout. Also, it comes very near to platforms like the Adapteva Epiphany [Olo+11] chips. Since the latter have a non-uniform memory access architecture (different access times to different banks), certain adaptions would be necessary, but feasible.

5.1.3.3 Problem Description

The problem to be solved now is assigning buffers to banks. With the previous definitions, it can be described as follows.

Let (T, D, B, a) be an application to run on a platform (C, M). Let there further be a given mapping $T \longrightarrow C$ and a schedule for execution of the tasks. Find a mapping $f : B \longrightarrow M$ that assigns each buffer to a bank such that:

- All buffers fit into the banks they are assigned to, i.e.,

$$\forall m \in M, \quad \sum_{b \in B_m} |b| \;\leq\; |m| \quad \text{with} \quad B_m = \left\{ b \in B \,\middle|\, f(b) = m \right\}.$$

- The time between the execution start of the first and the execution end of the last task in the schedule is minimised. In this context, "time" denotes average values, as runtimes can vary due to synchronisation and data dependencies, for example.

5.1.4 Generic Memory Bank Assignment Algorithm

This section describes the approach proposed for solving the problem defined in the previous section, i.e., for assigning buffers to memory banks in the generic case with the generic platform model. First, the challenges of conceiving such an algorithm are discussed, and a general overview of the proposed method is given. Then the algorithm itself is presented and simple optimisations are discussed.

5.1.4.1 Overall Approach

A simple idea for solving the problem might be a heuristic that follows the concept of static *load balancing*, i.e. of distributing the buffers evenly among the banks. This heuristic would assign banks to all buffers in the order of their size, starting from the largest buffer. For each buffer, it would select the bank which, at the current state of assignment, has the most free space. Such an algorithm attempts to find a valid bank assignment if one exists and to distribute the access bandwidth over the banks. However, it is fully agnostic to program semantics and cannot detect possible access collision hotspots. For instance, in the application example from Figure 5.1 on page 138, the most promising optimisation would be to assign `Grad1` and `Grad2` to different banks. With a load balancing algorithm, however, in case of resource scarcity, whether this happens or not is coincidence.

A good bank assignment algorithm therefore needs to model possible access collisions between two tasks. These depend on multiple factors, such as whether the execution times of the tasks overlap, how often they access the buffers, in what pattern they do so, etc. Note that there exist cyclic dependencies between the timing of the tasks (overlaps) and the bank assignments. Also, there may be conflicting optimisation criteria on particular platforms. For instance, in order to avoid access collisions between different cores on the MPPA platform, one would often like to place buffers on different banks. On the other hand, if these buffers are later accessed simultaneously by one core, it may be advantageous to place them on the same bank, as will be shown later.

For modelling access collisions and the aforementioned related problems, many different analysis methods have been proposed and could be successfully applied here. For reasons of simplicity, however, we choose a heuristic approach: We take the time that two tasks execute in parallel as an indication for the occurrence of

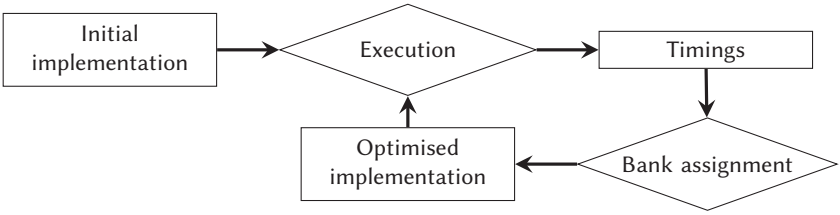


Figure 5.3: Flow chart diagram of the iterative approach proposed for assigning buffers to banks

access collisions on their associated buffers. This time is determined by measurements. By cumulating the execution time overlaps from all pairs of tasks, we obtain pairwise access conflict potentials between all buffers. Whether these conflicts materialise depends on the bank assignment of the buffers, which we try to optimise.

This leads to the iterative approach shown in Figure 5.3. First, the application is executed in an initial implementation, measuring the start and finish times of all tasks. This initial implementation could either be with interleaved memory address mapping if the platform allows it, or contiguous memory mapping with a bank assignment obtained by a simple heuristic like the load-balancing based method mentioned previously. Based on these times, a bank assignment is obtained by our algorithm. Application execution with this new assignment yields new timings, which are then used for a refined bank assignment as the execution leads to different assumptions on overlapping accesses to shared memory banks. This is continued until the assignment converges or for a fixed number of iterations. In our experiments, ten iterations proved to be sufficient to get a steady-state behaviour or a limit cycle.

The following section will present the above described algorithm that assigns banks to buffers.

5.1.4.2 Basic Bank Assignment Algorithm

As already mentioned, we need an algorithm that, given the execution start and end times for all tasks in an application, finds a bank assignment that minimises access conflicts between different tasks. To attain this goal, we express it as a graph colouring problem. In the graph to be coloured, each buffer is represented by a node and an edge exists between two nodes when the corresponding buffers are accessed in parallel by different tasks. The banks are represented by colours, so the number of colours is fixed. A fundamental difference to classic graph colouring is that the latter does not allow two neighbouring nodes to have the same colour. For bank as-

Algorithm 5.1: Graph colouring based bank assignment

input : Application (T, D, B, a), platform (C, M),
task execution times $start : T \longrightarrow \mathbb{N}$, $end : T \longrightarrow \mathbb{N}$
output: Bank assignment $colours : B \longrightarrow M$

$(V, E) \leftarrow$ CreateCollisionGraph$((T, D, B, a), start, end)$
for $i \leftarrow 1 \ldots |B|$ **do**
  $R_i \leftarrow$ SelectRemoveCandidate$((V, E), (T, D, B, a))$
  $(V, E) \leftarrow$ RemoveNode(V, E, R_i)
end
for $i \leftarrow |B| \ldots 1$ **do**
  $(V, E) \leftarrow$ ReinsertNode(V, E, R_i)
  $colours[R_i] \leftarrow$ ChooseColour$((V, E), (C, M), colours, R_i)$
end

signment, this case would mean possible access conflicts on the concerned bank, but the program would still execute correctly. Neighbours of same colour are therefore allowed but will yield a performance penalty.

The algorithm has been inspired by a well known graph colouring based solution to the register allocation problem [Cha82] and is shown in Algorithm 5.1. It consists of constructing a *collision graph*, gradually removing all nodes from it and re-inserting and colouring them in reverse order. The idea of the node removal phase is to fix an order in which the nodes are coloured: Since each colouring decision reduces the degrees of freedom for the remaining nodes, it is important on the one hand to colour those nodes first for which conflicts would have the greatest impact and on the other hand those that have a low degree of freedom already from the start. The details of the different phases shall be layed out in the following.

a) Construction of the collision graph: Given an application (T, D, B, a), the collision graph is an undirected, weighted graph with the buffers in B as the nodes. A description of its construction (function CreateCollisionGraph) is given in Algorithm 5.2. For each pair of tasks $t_1 \neq t_2 \in T$ that execute in parallel, an edge is inserted between each pair of buffers $b_1 \in a(t_1), b_2 \in a(t_2), b_1 \neq b_2$. These are the buffers that are accessed in parallel by both tasks. The weight of the edge is equal to the time that t_1 and t_2 execute in parallel and multiple edges between the same nodes are combined to one edge with the sum of the weights.

Figure 5.4 shows the conflict graph for the case of the example application from Figure 5.1 on page 138. It is assumed that `GradientX` and `GradientY` execute in

Algorithm 5.2: Function CreateCollisionGraph

input : (T, D, B, a), *start*, *end*
output: Collision graph (V, E)

$(V, E) \longleftarrow (B, \{\text{pairs}(B) \longrightarrow 0\})$
foreach pair $t_1 \neq t_2 \in T$ **do**
  // calculate execution time overlap o
  $o \longleftarrow \min\left(end(t_1), end(t_2)\right) - \max\left(start(t_1), start(t_2)\right)$
  if $o > 0$ **then**
    foreach $b_1 \in a(t_1), b_2 \in a(t_2), b_1 \neq b_2$ **do**
      $E[\{b_1, b_2\}] \longleftarrow E[\{b_1, b_2\}] + o$
    end
  end
end

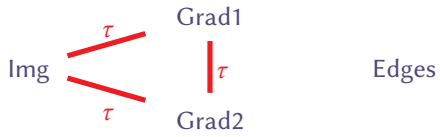


Figure 5.4: Conflict graph for the sample application from Figure 5.1 on page 138

parallel for τ cycles. As a result, the graph contains three edges between the buffers `Img`, `Grad1` and `Grad2`, each with the weight τ.

b) Node removal: In this step, repeatedly a node will be chosen and removed from the graph, until the graph is empty. The node to be removed is determined by SelectRemoveCandidate, which can be described by the function

$$r((V,E),(T,D,B,a)) = \arg\max_{v \in V} \Big(\lambda^3 \cdot \left|\{t \in T \mid v \in a(t)\}\right| - \lambda^2 \cdot \left|\{v' \in V \mid E[\{v,v'\}] > 0\}\right| - \lambda \cdot \sum_{v' \in V} E[\{v,v'\}] - |v| \Big). \quad (5.1)$$

In this notation, λ is considered to be a symbolic constant that is very large compared to all other numbers in the formula. The function will thus return that node $v \in V$ for which the λ^3 term is the largest. Only if there are multiple nodes with the same λ^3 term, the node with the least absolute value for the λ^2 term will be returned (least because of the negative sign). Only in case of another tie will lower power terms of λ be taken into consideration.

The function can be described as follows. It selects the nodes to be removed *first* and later coloured *last*, i.e., the nodes with *lower priority*. For this purpose, it performs a multi-criteria comparison, where the first criterion is the most important and later ones are only considered in case of a tie. The criteria for this comparison of the nodes are explained below, ordered from high to low priority. The node to be removed is that node with

The highest number of tasks accessing the buffer. The reasoning behind this somewhat counter-intuitive criterion is that buffers accessed by only few tasks yield a high optimisation potential as opposed to buffers accessed by many tasks, which are anyway susceptible to access collisions.

The lowest number of neighbours in the graph. This is known as an important criterion also in register allocation, essentially because a high number of neighbours means a lower degree of freedom when trying to avoid conflicts. Therefore, we try to colour nodes with many neighbours first to still have a higher number of colours left for them. Note that the removal of nodes influences the number of neighbours left for the other nodes, often uncovering further nodes with high degrees of freedom. This is the essential idea behind the removal procedure.

The lowest cumulative weight of all adjacent edges. This is again an indicator for the optimisation potential of a buffer.

The lowest size of the buffer. This is only a minor criterion meant to improve algorithm reliability (cf. Section 5.1.4.3).

For the example graph from Figure 5.4 on the previous page, this would result in `Edges` being coloured first, then `Grad1` and `Grad2`, then `Img`. While this seems unintuitive for this simple example application, it does make sense in more complex scenarios with more simultaneous tasks and more buffers involved. If a node (like `Edges` in this example) has no neighbours in the conflict graph, there is a chance that it can be accessed by the corresponding tasks without any conflicts. Colouring such a node first increases the probability that this chance is exploited.

c) Node re-insertion and colouring: The graph is reconstructed by re-inserting and colouring the nodes in the reverse order of their removal before. The colour for a node and thus the bank assignment of a buffer is given by ChooseColour, which can be described by the function

$$x((V,E),(C,M),colours,v) = \underset{m\in M^*(v)}{\arg\max}\left(-\lambda\cdot\sum_{v'\in V|\,colours[v']=m} E[\{v,v'\}] \;+\; \text{free}(m)\right), \quad (5.2)$$

where

$$\text{free}(m) = |m| - \sum_{v \in V \mid colours[v]=m} |v|$$

gives the free space on m and $M^*(v) = \left\{ m \in M \,\middle|\, \text{free}(m) \geq |v| \right\}$ is the set of banks with enough free space to accommodate buffer v. λ is again used like in (5.1).

This is again a multi-criteria comparison, in which those banks are chosen that have (in that order):

The least sum of weights on adjacent edges leading to nodes of the same colour. As discussed previously, we regard the weights of the edges as an indicator for the occurrence of simultaneous accesses and thus for the conflict potential between two buffers. If we decide to assign two adjacent nodes to the same bank, the conflict potential of the connecting edge will materialise.

The most space left on the bank. The idea behind this criterion is load balancing.

If there is no bank with enough space left to accommodate a buffer, the algorithm fails.

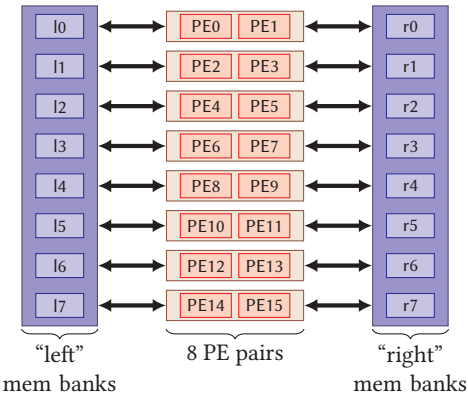
For the conflict graph from Figure 5.4 on page 143, if we assume that only two memory banks are available, the colouring would take place as follows.

1. `Edges` would be assigned to any of the two banks; in the following, we assume it is to the first one.
2. `Grad1` would be assigned to the second bank, since the latter has more free space left.
3. `Grad2` would be assigned to the first bank, since `Grad1` is assigned to the other one and there is an edge between the two nodes in the conflict graph.
4. `Img` has edges to `Grad1` and `Grad2`, each of which is mapped to one of the two banks. Therefore, the penalty is equally high for both banks, and `Img` is assigned to the second bank, again because of free space.

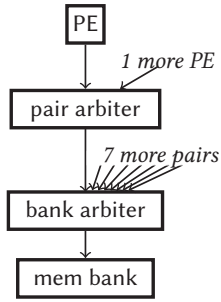
If there were three banks, `Edges` and `Img` would be mapped to one bank, `Grad1` and `Grad2` to the other two. This would completely avoid access conflicts between different buffers.

5.1.4.3 Improving algorithm reliability

The algorithm as described above fails if at one point no bank has enough space left to accommodate the buffer to be assigned. This can happen if smaller buffers are distributed first, not leaving sufficient space for the larger buffers. In this case, the algorithm is re-run, with a special *correction factor* γ. For this purpose, an additional



(a) Memory architecture overview



(b) Arbitration scheme for memory accesses

Figure 5.5: Illustrations of an MPPA cluster

term $\lambda^3 \cdot \gamma \cdot |v|$ is added in (5.1). It enforces a higher priority for the buffer size during the node removal phase. γ is small in the beginning, but if the algorithm fails again, it is increased exponentially until bank assignment succeeds. This drastically reduces the rate of algorithm-induced failures. In our experiments, they were no longer an issue.

5.1.5 Platform Specific Adaptions

Previously, we presented a memory bank assignment approach for a generic platform. This section now shows how this approach can be adapted to concrete platforms, which may slightly deviate from the assumptions made before.

The Kalray MPPA is going to be used as an example for such a platform. First, its cluster architecture is shown in detail and some of its characteristics are derived from experiments. In particular, the effects of interleaved vs. contiguous memory address mapping and those of the cache will be examined. Then, the generic bank assignment algorithm is adapted to deal with the platform's special properties.

5.1.5.1 MPPA-256 Memory Architecture

The Kalray MPPA-256 Andey processor [Din+14] integrates 256 PEs, which are grouped into 16 compute clusters. All PEs implement the same VLIW architecture.

Figure 5.5a illustrates the architecture of a cluster. It has a local on-chip memory with 2 MB capacity, which is organized in 16 independent SRAM banks. These are

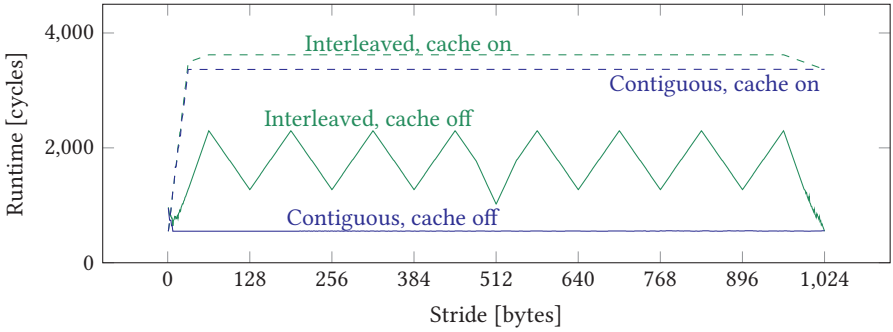


Figure 5.6: Runtime for reading 128 bytes with different stride

arranged in two sides (*left* and *right*). The PEs are organised in 8 *pairs*. Each pair has two memory buses (one for each side), which can be utilised in parallel by the two cores. Figure 5.5b shows the arbitration hierarchy for a PE that wants to access a certain memory bank. A first conflict arises when the other core of the pair wants to access a memory bank from the same side. A second possibility for a conflict is that another pair tries to access the same bank. All these conflicts are resolved by round robin arbitration.

The memory model is von Neumann; however, each core has two private, two-way set associative caches, one for instructions and one for data. While the instruction cache is always enabled, the data cache can be enabled or disabled as needed by the programmer. Cache coherency is not supported and is a responsibility of the programmer.

Within a compute cluster, the memory address mapping can be configured either as interleaved or as contiguous. In the contiguous mapping, each bank spans 128 kB consecutive addresses. In interleaved mode, the data is distributed over all banks with a granularity of 64 byte blocks. The blocks are placed on left and on right banks in alternation.

5.1.5.2 Synthetic Memory Benchmarks

The MPPA platform has a number of properties that are important to know for achieving optimal performance in a cluster. The following experiments will demonstrate these properties.

In a first experiment, a single core reads 128 bytes individually from memory. The addresses of the bytes increase linearly with a constant stride. Figure 5.6 shows the time needed for reading all the bytes, as a function of the stride. The experiment was conducted with data caches enabled and disabled, and with interleaved and

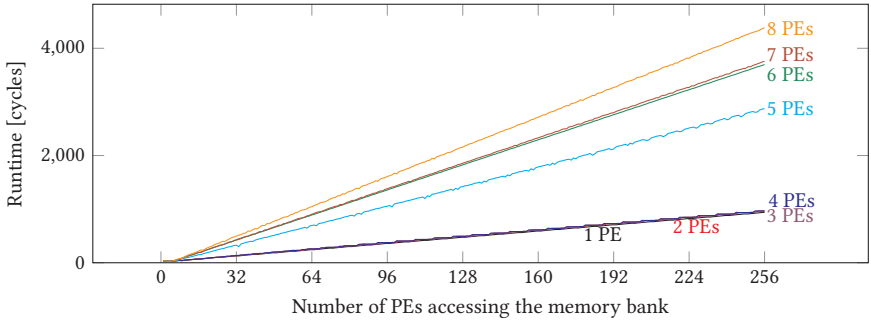


Figure 5.7: Runtime for consecutive memory accesses with contention

contiguous memory configuration. The contiguous configuration shows the expected behaviour, i.e., the access time is constant and independent of the target bank. In particular, since SRAM is used, the access time is constantly low also for bigger strides. The interleaved configuration, in contrast, needs some explanation. With a stride of one byte, 64 consecutive accesses are made to the same bank before the core needs to read from a different bank. With a stride of 64 bytes, each access is to a different bank. A delay for switching banks thus explains the different timings. Assuming an additional delay caused by switches between bank sides (left to right or vice versa), one can explain the lower runtime for strides of 128 bytes, which always access banks on the same side. Overall, the measurements from this experiment perfectly match a model that assumes 4 cycles for one access, 3 cycles for switching banks and 4 additional cycles for switching the side.

Another finding is that in this experiment, enabling the cache clearly worsens performance. The reason for this is that since there is no memory access locality in the code, the caches do not bring a benefit, but on the contrary, with increasing strides, lead to continuous loads of entire 32 byte cache lines for only one byte that is accessed. This effect is not limited to this particular experiment but could occur for any code with little locality.

Contention effects between multiple cores are evaluated in a second experiment. As before, one core reads data from a buffer and measures the time needed for it in contiguous mode. However, between 0 and 7 other cores access the same memory block simultaneously such as to create contention. Note that simultaneous accesses from more than 8 cores cannot occur in an MPPA cluster due to the PE pair architecture. Figure 5.7 shows the runtimes depending on the amount of data read for different numbers of contending cores. While for four or less cores in total, the effects of contention are insignificant, a large impact can be observed for more cores.

These results lead to the following general conclusions:

- For code with little memory access locality, enabling data caches can substantially decrease performance.
- With ideally partitioned data, the performance potential is higher for contiguous than for interleaved mode.
- In interleaved mode, due to the bank switching delays, even programs with data-independent control flow can have data-dependent execution times when they have memory accesses at data-dependent addresses.

With regards to contiguous memory organisation, these rules of thumb can be derived: (i) Simultaneous accesses from large numbers of cores to the same bank should be avoided. (ii) All memory accesses of one core should, if possible, be on a single memory bank to avoid bank switching delays. If multiple banks need to be used, all of them should be on the same side (left/right) to at least avoid side switching delays. (iii) The two PEs in a pair should access banks from opposing sides (this follows directly from the architecture).

5.1.5.3 Adaption of the bank assignment algorithm

From the point of view of the high-level architecture, an MPPA cluster, when configured for contiguous memory layout, clearly fits the generic platform model described earlier in Section 5.1.3.2. Looking at the results of the previous experiments on the platform, however, some deviations from the assumptions made there can be spotted. The main reasons for these deviations are the organisation in PE pairs and the delays for switching banks or sides in consecutive memory accesses. These architectural specifics lead to the fact that when the system is in a particular state, accesses to some banks will take longer than to others. This, however, contradicts the assumption made before that each PE takes the same time to access each bank.

While the generic bank assignment algorithm can still be used for the MPPA platform, its results will not be optimal, since it does not take account of the platform's particularities. To obtain better assignments, the algorithm therefore needs to be adapted. The following paragraphs will show how this can be achieved.

The adapted conflict graph, again for the sample application from Figure 5.1 on page 138, is given in Figure 5.8 on the following page. It is assumed that the task `GradientY` is mapped to the second PE of a PE pair, all other tasks to the first PE of the same pair.

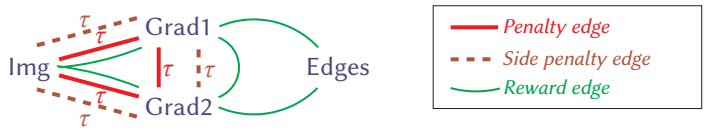


Figure 5.8: Conflict graph for the sample application from Figure 5.1 on page 138, with the MPPA-specific algorithm extensions.

a) Collisions within PE pairs: The two PEs in a pair should not access two banks from the same side simultaneously. To achieve this, we add a second type of weighted edges, the *side penalty edges*. Their weight is calculated like the weights of the other edges, except that only those task pairs mapped to the two PEs of a PE pair are taken into consideration. In Figure 5.8, since all penalty edges in the example come from the tasks `GradientX` and `GradientY`, there is a side penalty edges wherever there is also a "normal" penalty edge.

During the node re-insertion step, these weights are summed up for both bank sides, resulting in two side penalties w_{right} and w_{left}. The edge weight sums for each bank are then complemented with the corresponding side penalty, which corresponds to adding a term $-\lambda \cdot w_{\text{side}(b)}$ in (5.2) on page 144.

b) Banks accessed by one task: Switching between different banks takes time, so the buffers accessed by a task should be distributed over as few banks as possible. We approach this demand by inserting another type of (unweighted) edges in the collision graph, the *reward edges*. Such edges are inserted for each task; they are inserted between all the buffers it accesses. In Figure 5.8, the reward edges connected to `Img` come from the tasks `GradientX` and `GradientY`. All other edges come from the task `Abs`.

The reward edges are only considered in the node re-insertion step as the second criterion after the other edges: If two banks have the same sum of weights, the bank with the higher number of reward edges to adjacent buffers of the same colour is chosen. This corresponds to extending (5.2) on page 144 with an additional term $+\lambda^{0.5} \cdot n_{\text{reward}}(v, b)$, where $n_{\text{reward}}(v, b)$ is the number of reward edges between node v and other nodes with colour b.

c) Bank sides accessed by one task: Switching between banks takes even more time if they belong to different sides. Therefore, if a task needs to access multiple banks, these should be on the same side (left or right). We account for this in the node re-insertion step when a node is to be coloured: We count the number of adjacent reward edges going to nodes assigned to left banks ($n_{\text{reward,left}}$) and those going to nodes assigned to right banks ($n_{\text{reward,right}}$). The results are added (with a weighting

factor ω) to the reward edge count for each bank of the corresponding side. This corresponds to adding a term $+\lambda^{0.5} \cdot \omega \cdot n_{\text{reward,side}(b)}$ in (5.2) on page 144. Empirically, $\omega = 0.375$ has turned out to be a good choice.

d) Cache indices: Since the data cache on the MPPA is two-way set associative, no more than two buffers accessed by the same task should have similar cache indices. Otherwise, if the buffers are accessed in alternation, frequent cache misses would occur. For this reason, a mechanism that aligns the buffers within each bank was added as a second step after the bank assignment. It adds free spaces between the buffers such that the base addresses of all buffers accessed by a task have different cache indices. Again, an algorithm based on graph colouring is used for this purpose. The nodes are the buffers as before, edges are inserted between two buffers if they are used by the same task, and the colours are given by all possible cache indices. In the node removal phase, those nodes are removed first (and later coloured last) that have the most free space on the banks they have been assigned to. The nodes are re-inserted filling banks from their base address upwards with buffers. Cache indices of the buffers are adjusted by placing "gaps" between the buffers. These gaps must be small enough to still fit into the bank.

5.1.6 Performance Comparison of Interleaved vs. Contiguous Mapping

To compare application performance of interleaved and contiguous memory configurations, we executed different applications on one cluster of a Kalray MPPA developer board, model Andey, using toolchain 1.4.2, with a bare-metal configuration. The applications comprise six different, parametrised benchmarks:

- Matrices of different sizes were multiplied, one matrix per core. Powers of two were chosen as the matrix dimensions in some cases (denoted as "matrix2") and other, random numbers in other cases (denoted as "matrix").
- The Fast Fourier Transform (FFT) of different signals in different lengths was calculated using a benchmark from [SBS], one FFT per core.
- A Canny edge detection filter was applied to images of different sizes. The images were split into different numbers of blocks, with one PE working on one block.
- Using convolutional neural networks that were taken from the CConvNet library [CPB14; Con], up to four hand-written digits were simultaneously recognised out of images, with four PEs working together for one recognition.

- Floating point number arrays of different sizes were sorted using a merge sort algorithm. All involved cores worked in parallel for one array.
- Sparse matrices of different shapes and sizes were multiplied with dense vectors (experiment denoted as SpMV). The matrices were split into different blocks with the same size of non-zero entries, with one PE per block.

Variation of the mentioned parameters as well as of the number of active PEs (between 2 and 16) yielded a total of 345 different configurations. Note that some of the benchmarks have regular memory access patterns (e.g. Canny), while others show data-dependent control flow (e.g. merge sort or SpMV). Usually, the former is advantageous for interleaved address mapping (see Chapter 4).

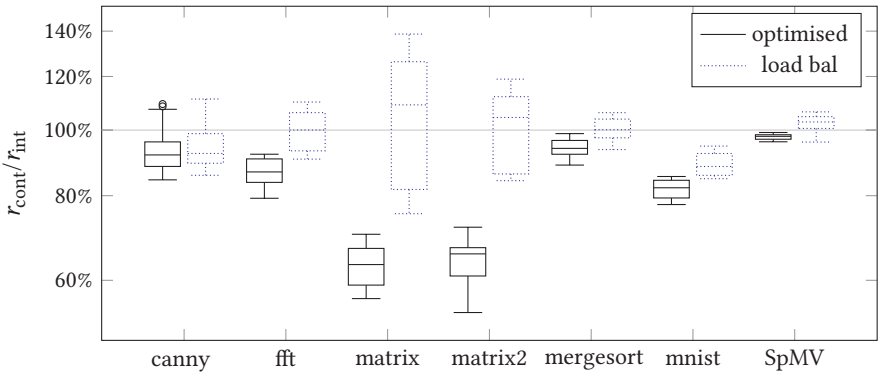
All benchmarks were implemented in Ladybirds C and parsed using the C++ frontend "Clang" of LLVM [LA04] into a newly created compilation framework. The tasks were assigned to the different PEs by hand following regular patterns. Dynamic scheduling was used. To exclude performance influences of the binding of the tasks to the PEs, the latter was kept constant in that the same task was always assigned to the same PE in all configurations of each benchmark. Bank assignment optimisation, code generation and benchmarking were conducted in an automated process. Each configuration was implemented with the local data caches enabled and disabled. All benchmarks were executed in three ways:

- With interleaved mode,
- With contiguous mode, optimised using the proposed algorithm,
- With contiguous mode and a bank assignment obtained using the load balancing based approach discussed in Section 5.1.4.1.

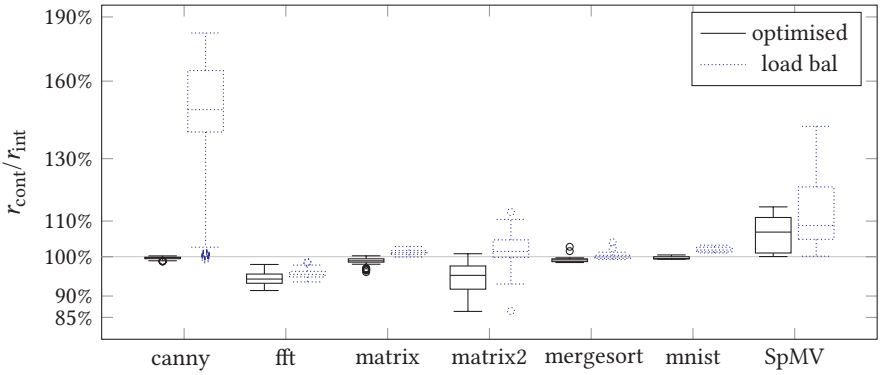
The results shall be presented in the following. Figure 5.9a compares the runtimes of the benchmarks in contiguous mode to those in interleaved mode, for the optimised bank assignment algorithm proposed in this work as well as for the load-balancing based solution. Data caches are disabled.

Using the assignments determined by the proposed algorithm, the applications run equally fast or faster with a contiguous memory configuration for 95.1% of all configurations. The speed-ups are significant (more than 5%) for 75.1% of all configurations. Only in the Canny experiment, 14% of configurations performed worse with contiguous mapping; in the worst case, the runtime was 9.0 % longer than with interleaved mapping.

Figure 5.9b shows the results of the same experiments with the data caches enabled. In this case, 87.8% of all configurations run at least equally fast with contiguous memory, while speed-ups are significant in still 20.3% of the configurations.



(a) Data cache disabled



(b) Data cache enabled

Figure 5.9: Ratio of runtime in contiguous mode (r_{cont}) and runtime in interleaved mode (r_{int}), with data cache enabled or disabled.
The results are shown as Tukey box plots: Each box describes their distribution for all different configurations of the corresponding benchmark. The bounds of the box mark the first and third quartiles, the band inside marks the median value. The whiskers extend to the most extreme results still within 1.5 times the inter-quartile range from the box. Results outside that range are marked separately as outliers.

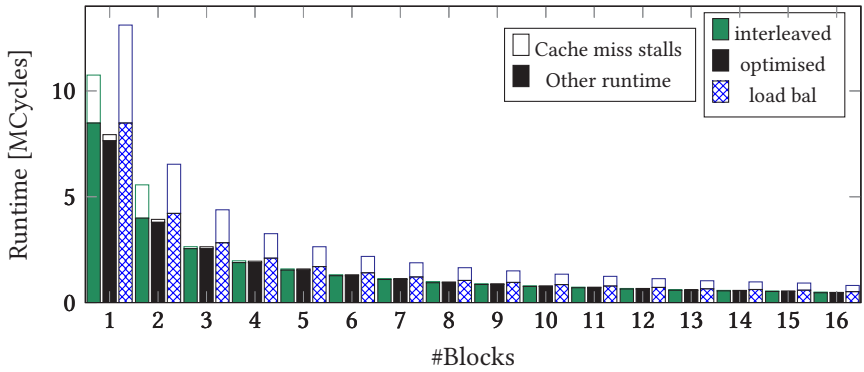


Figure 5.10: Measured runtime and cache miss stalls of the Canny benchmark for a 338 × 258 pixel image, depending on the number of blocks it was split into (i.e. the number of PEs involved). Cache was enabled.

A moderate performance degradation can be seen for the SpMV benchmark. It is due to the fact that all the PEs access with a highly irregular pattern the vector to be multiplied with the sparse matrix. This results in a 14.4% longer runtime in the worst case.

Deeper insight into one of the different configurations is provided in Figure 5.10 in an exemplary way for the Canny experiment. The figure shows details about the absolute runtimes for one particular configuration, but with a varying number of active PEs. With enabled data cache, in-built performance monitoring counters of the MPPA platform were used to measure the number of stall cycles due to cache misses as a reference of the memory access overhead. Since the work is split between the active PEs, an increasing number of the latter yields a smaller part of the memory to be accessed by each PE; as a result, the caches get more efficient.

Both with caches enabled and disabled, the matrix2 and fft benchmarks perform significantly worse in interleaved mode. This is because the Kalray MPPA uses **sequential interleaving**, i.e., the bank a memory address is assigned to is given by a number of lower-order bits of the address. Consecutive accesses with address offsets of powers of 2 (as they occur frequently in the applications mentioned before) therefore all go to the same bank, in the case of a collision causing multiple subsequent collisions as well. As discussed in the last chapter, this problem is well known and can be solved by **pseudo-random interleaving** [Rau91]. While the latter technique is frequent with off-chip memory systems, to our best knowledge no on-chip memory architecture is organised in this fashion. One can conclude from

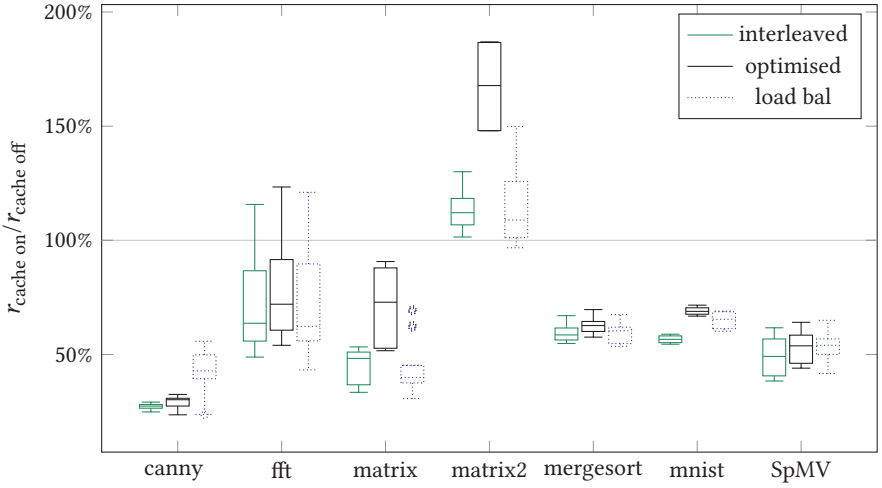


Figure 5.11: Ratio of runtimes with data cache enabled and disabled. The ratio is shown for interleaved mapping, for contiguous mapping optimised with the proposed algorithm and for contiguous mapping with the simple, load-balancing based method. The results are again shown as Tukey box plots like in Figure 5.9.
For most configurations, this ratio is below 100 %, i.e., the runtime is shorter with cache enabled. One can also see that the speed-up potential of the cache is lower for configurations that already profit from an optimised bank assignment when the cache is disabled (cf. Figure 5.9a).

this that sequential memory interleaving, in deviation from one of its frequently-named advantages, does not allow for memory-agnostic programming in every case.

Figure 5.11 compares the runtimes with enabled caches to those with disabled caches. In 86.9% of the configurations, enabling the cache yields a speed-up. Exceptions are the "matrix2" benchmarks and few configurations of the FFT benchmark; these applications only show little locality. For all configurations, it holds that enabling the cache yields a speed-up either in both contiguous and interleaved mode or in none of them. The figure also shows that the optimisation potential of enabling the cache in general is lower for contiguous memory with optimised bank assignment. This is because in interleaved mode, memory accesses have a higher cost due to performance-degrading patterns like switching between different banks. Since the cache reduces the number of memory accesses, its impact is determined by that cost.

In summary, it can be stated for many applications that while the cache is able to hide certain issues like access conflicts or bank switch latencies, still better performance can be attained by avoiding these problems altogether through the choice of an appropriate contiguous memory mapping. Particularly, in cases of low locality, only contiguous mapping boosts performance.

Independently of the aspect of performance improvement, contiguous mapping allows for better static analysability [Bec+16; Car+14; VY15; NYP16; Per+16]. As the results show, this analysability comes at a very moderate price, the possible average performance degradation being low in most of the cases.

The runtime of the mapping algorithm itself was in the order of tens of milliseconds on an Intel Core i7-3820 CPU based machined clocked at 3.60 GHz. This makes it very short as compared to the overhead for code generation, compiling the code for the target platform etc.

5.1.7 Concluding Remarks

In this first part of this chapter, we compared contiguous and interleaved memory configurations on the Kalray MPPA. Synthetic benchmarks showed important characteristics for the memory access delays on this platform.

For the contiguous memory configuration, we presented a relatively simple optimisation algorithm for assigning buffers to memory banks. This algorithm only needs little information about the application; in particular, fine-grained profiling or in-depth static analysis are not required. Still, in real-world benchmarks we were able to attain speed-ups of up to 86 % as compared to the interleaved configuration, while significant degradation in speed was only observed in a minority of the cases.

This shows that using contiguous memory mapping is a worthwhile alternative to interleaved mapping. With the presented algorithm, it is feasible in terms of programming effort and on the MPPA it is at least comparable in terms of average performance. Being clearly better suited for worst-case timing analysis is what makes it particularly attractive.

At the same time, the potential of this optimisation method has not been fully exploited yet. In particular, the algorithms could be clearly enhanced by adding static analysis, for instance for determining the actual numbers of accesses the tasks perform to buffers. Also, only the sub-problem of distributing data within one cluster is targeted; the bigger challenge of placing data on the numerous memory banks of the Epiphany chip still remains. The next section will detail this and other challenges and outline the necessary steps towards efficient data placement and exchange on any arbitrary platform.

5.2 Data Storage and Transport Optimisation on Multi-Memory Systems

Earlier in this thesis, the Ladybirds specification model was introduced with arbitrary multi- and manycore target architectures in mind. The focus was then directed towards such systems with one shared memory (which could, however, consist of multiple banks, with interleaved or contiguous address mapping). The mechanisms elaborated so far already cover platforms like the Intel Xeon Phi [Chr14], and they naturally extend to systems like the Kalray MPPA even when using all clusters — data transfer tasks can be inserted mechanically whenever data exchange between different clusters is required, whereupon the program can be optimised for each cluster individually. For other target platforms, unfortunately, these mechanisms are not yet sufficient. This comprises platforms with non-uniform memory architectures, such as Adapteva Epiphany [Olo+11], and platforms with memory hierarchies like P2012 [Ben+12] or PULP [Con+15]. These systems require methods of careful data placement on the different banks. In this context, also accurate and tight packet liveness analysis can be performance-critical, just as well as mechanisms of temporarily moving packets to other locations such as to make space on fast memory modules. Due to the high complexity of these individual optimisations combined with the manifold interdependencies between them, no complete optimisation algorithm will be presented in this thesis. To get an idea, however, about the feasibility of such an undertaking, goals and constraints of it shall be formalised in the following and the most important challenges shall be discussed together with possible solution approaches. Finally, it will be shown how optimisation methods can be adapted to concrete hardware architectures and their special requirements.

5.2.1 Hardware and Software Models

In Section 3.4, a Ladybirds optimisation procedure was defined as a transformation of a packet dependency graph to a buffer dependency graph. In addition to the former, more information is required for successful data storage and transport optimisation. This comprises further details about the application as well as a description of the target platform. What exact data is needed clearly depends on the optimisations that are performed; this section will give an overview on some parameters that can be readily obtained and aid the optimisation process. First, an accurate and detailed optimisation model shall be formalised, then a formal, generic hardware description model is presented.

To allow a better representation of the data dependencies in a packet dependency tree, the transformation of splitting interfaces according to the dependencies

was proposed in Section 3.4. On the other hand, since buffers must be allocated as a whole for each interface, the original packet dependency tree can be useful as well during certain optimisation stages. In the following considerations, a hybrid representation shall be used which depicts complete interfaces together with their breakdown into *regions*.

To describe the dimensions of interfaces and regions, the notion of *cubes* shall be utilised. An n-dimensional cube $\mathcal{q}$ shall be defined by two corner points $\boldsymbol{q}^l$ and $\boldsymbol{q}^u \in \mathbb{Z}^n$ such that $\mathcal{q} = \left\{ \boldsymbol{p} \in \mathbb{Z}^n \;\middle|\; \boldsymbol{q}^l \leqslant \boldsymbol{p} < \boldsymbol{q}^u \right\}$, where $<$ and $\leqslant$ are defined element-wise, e.g. $\boldsymbol{x} \leqslant \boldsymbol{y} \iff x_i \leqslant y_i \forall\, i$. $\mathbb{Q}^n$ is the set of all n-dimensional cubes.

An interface i for an n-dimensional packet can then be defined as a tuple $(\mathcal{q}_i, R_i)$, where $\mathcal{q}_i \in \mathbb{Q}^{n+1}$ is given by $\boldsymbol{q}_i^l = \mathbf{0}$ and $\boldsymbol{q}_i^u$, the dimensions of the packet. To account for the different sizes of the underlying data types (`char`, `int`, `double`, ...), the first dimension of $\boldsymbol{q}_i^u$ shall denote the size, in bytes, of the packet base type. For instance, for a packet declared as `double x[5][2][3]`, the corresponding interface i would have $\boldsymbol{q}_i^u = (8, 3, 2, 5)^\top$. With this technique, every array of any data type can be modelled as a multi-dimensional array of bytes.

R_i is a set of regions, with each region $r \in R_i$ covering a sub-cube $\mathcal{q}(r)$ of $\mathcal{q}_i$, i.e. $\mathcal{q}(r) \subseteq \mathcal{q}_i$. All these sub-cubes are pairwise disjoint and

$$\bigcup_{r \in R_i} \mathcal{q}(r) = \mathcal{q}_i \,.$$

For instance, if a sub-packet[1] `x[2,3][1][0,2]` of the packet `x` from the last example were used as a task input, the corresponding region r would cover a sub-cube $\mathcal{q} := \mathcal{q}(r)$ with $\boldsymbol{q}^l = (0, 0, 1, 2)^\top$ and $\boldsymbol{q}^u = (8, 3, 2, 4)^\top$.

A task t shall be a tuple $(I_t^\mathrm{i}, I_t^\mathrm{o}, k_t, J_t^\mathrm{io}, J_t^\mathrm{bud})$, where

- I_t^i is a set of input interfaces
- I_t^o is a set of output interfaces
- k_t is a kernel describing the operation to be performed
- $J_t^\mathrm{io} \subseteq I_t^\mathrm{i} \times I_t^\mathrm{o}$ describes interfaces for combined input/output (`inout`) packets. For each such packet, one input and one output interface is declared in I_t^i and I_t^o; the pair of these interfaces in J_t^io specifies that both must be provided with the same buffer.
- $J_t^\mathrm{bud} \subseteq I_t^\mathrm{i} \times I_t^\mathrm{o}$ describes interfaces for buddy packets, similar to J_t^io.

[1] For the Ladybirds sub-packet notation, see page 85, in particular also Figure 3.1. Note that the Ladybirds C language expects closed intervals, whereas this model works with half-open intervals.

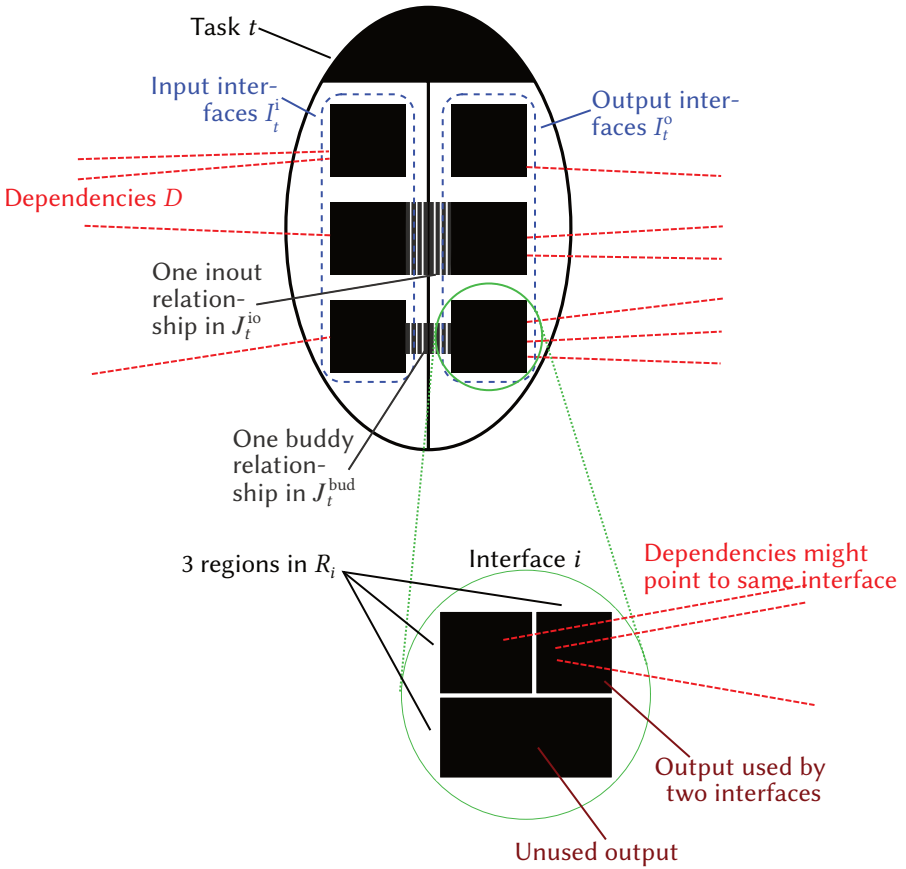


Figure 5.12: Illustration of the modelling of a packet dependency graph. Interfaces are composed into regions and combined input/output interfaces are depicted separately, but with a strong connection between them.

An application can then be represented as a packet dependency graph, which can be defined as a tuple $(R^{\mathrm{i}}, R^{\mathrm{o}}, T, D)$, where R^{i} and R^{o} are sets containing all regions in the application, R^{i} those used in input and R^{o} those used in output interfaces. T is the set of all tasks and $D \subseteq R^{\mathrm{o}} \times R^{\mathrm{i}}$ is the set of edges representing all packet dependencies in the application. Note that for each $r \in R^{\mathrm{i}}$, there is exactly one $d \in D$ which has r as the target. For regions contained in R^{o}, any number of dependencies from zero

to $|R^{\mathrm{i}}|$ can have them as source. Figure 5.12 on the preceding page illustrates the model.

A hardware platform can be modelled as a tuple $(C, C^{\mathrm{x}}, M, A, X)$, where

- C is the set of cores or processing elements in the platform that can execute tasks
- C^{x} is the set of cores, processing elements, DMA controllers and other devices that can perform data transfers between different memory modules or data copies within memory modules. C and C^{x} need not be disjoint.
- M is the set of memory modules on the platform. For each memory module $m \in M$, $|m|$ denotes its capacity in bytes.
- $A \subseteq C \times M$ is a set of access features. Each $a = (c_a, m_a) \in A$ denotes that the core c_a can access the memory module m_a.
  Cost functions $\tau^{\mathrm{r}} : A \longrightarrow \mathbb{N}$ and $\tau^{\mathrm{w}} : A \longrightarrow \mathbb{N}$ specify the time, in cycles, needed for one such read or one write access.
- $X \subseteq M \times C^{\mathrm{x}} \times M$ is a set of data transfer features. Each $x = (m_x^{\mathrm{from}}, c_x, m_x^{\mathrm{to}}) \in X$ denotes that data can be transferred from memory module m_x^{from} to memory module m_x^{to} using the transfer component c_x.
  A cost function $\tau^{\mathrm{x}} : X \times \mathbb{N} \longrightarrow \mathbb{N}$ gives the time, in cycles, needed for transferring a specified amount of bytes as indicated by the given transfer feature.

All elements of a platform description can be derived directly from the target platform architecture; in certain cases, however, cost functions are not known or not published by the platform producer. In these cases, measurements may be necessary. Access conflicts or congestion on buses or networks on chip, other than through the resource availabilities in C and C^{x}, are not included in this model, but could be added.

The packet dependency graph described earlier can be obtained directly from the application specification. However, the following additional parameters contain useful information about an application, where $I^{\mathrm{i}} := \bigcup_{t \in T} I_t^{\mathrm{i}}$ and $I^{\mathrm{o}} := \bigcup_{t \in T} I_t^{\mathrm{o}}$ are the sets of all input and output interfaces in the application.

- $\tau^{\mathrm{e}} : T \times C \longrightarrow \mathbb{N}$, the pure execution time (without read accesses) in cycles of a given task on a given core. These times are required to predict which tasks can and will run in parallel. They can be obtained through measuring (see last section), but also with static analysis tools like [FH04].
- $n^{\mathrm{r}} : I^{\mathrm{i}} \cup I^{\mathrm{o}} \longrightarrow \mathbb{N}$, the number of read accesses performed to the packet provided through a given interface. These numbers are important to predict

possible access conflicts and, more importantly, the impact of the choice of the memory module on which the buffers for the interfaces are allocated. n^{r} can again be obtained through measuring or through static analysis.

- $n^{\mathrm{w}} : I^{\mathrm{o}} \longrightarrow \mathbb{N}$, the number of write accesses performed to the buffer provided through a given interface. Similar considerations as with n^{r} hold for these numbers. The separation of these two functions is necessary because the costs of read and write operations can significantly differ on certain architectures, e.g. Adapteva Epiphany.

As already discussed, these parameters can be readily obtained; typically, they are also used in other optimisation methodologies. More information could be collected and included in the different models, but the following sections will show the large range of optimisations that is already possible with only the parameters mentioned above.

5.2.2 Problem Definition

The application model and the platform model proposed earlier allow a definition of the problem of optimising Ladybirds applications for given target platforms. The most important results to be produced are certainly a task binding and a scheme for allocating buffers on appropriate memory modules. There are, however, more decisions to be taken, and a high cross-dependency exists. If, for instance, one task produces data on one memory bank and another task expects this data on a different bank, a data transfer task must be inserted. On other occasions, to allow concurrent processing of the same data, the latter must be copied within a memory module. These operations must be scheduled such that all buffers fit into the memory, which, again, might require more data transfers etc.

In the following, all the results to be produced shall be formalised as well as the requirements that they must meet. These outputs are formulated as individual subproblems; they are presented in an order which makes it easy to describe the problem, whereas the order in which the subproblems have to be solved may be entirely different. Table 5.1 on page 164 can be used as a help to look up the meanings of symbols and variables.

Given a platform $(C, C^{\mathrm{x}}, M, A, X)$ and a packet dependency graph $(R^{\mathrm{i}}, R^{\mathrm{o}}, T, D)$ together with additional information consisting of $\tau^{\mathrm{e}}, n^{\mathrm{r}}$ and n^{r}, the optimisation procedure must establish the following results.

For one, a **buffer dependency graph** must be produced out of the packet dependency graph. This needs the following decisions:

- It must be decided which of the *buddy interfaces* in each task are *merged*, i.e., provided with the same buffer. These interface pairs then act like `inout` interface pairs, together with which they shall be stored in a common set.

  Formally, for each task $t \in T$, the optimiser must produce a set $J_t = J_t^{\mathrm{io}} \cup J_t^{\mathrm{bud}^*}$ with $J_t^{\mathrm{bud}^*} \subseteq J_t^{\mathrm{bud}}$. No interface must appear twice in J_t.

- For each dependency, it must be decided whether the corresponding (sub-) packet will be exchanged through buffer sharing or whether a *copy* or a *transfer task* is inserted. It may also be necessary to insert multiple such tasks to temporarily move data to a different memory module and later bring it back.

  Formally, the optimiser must generate a set T^{x} of transfer or copy tasks and a function $\gamma : D \longrightarrow \mathscr{P}(T^{\mathrm{x}})$ assigning each dependency zero, one or multiple such transfers or copies. Each task in T^{x} must be assigned to exactly one dependency. The same (sub-)buffer may be modified by one task at most, otherwise copy tasks must be inserted, i.e. for each $r \in R^{\mathrm{o}}$, there must be at most one dependency $(r, r') \in D$, with $\gamma\big((r, r')\big) = \{\}$, to a region r' that is part of an `inout` or merged buddy interface.

With these results, a buffer dependency graph $(R^{\mathrm{i}*}, R^{\mathrm{o}*}, T^*, D^*)$ can be constructed with $T^* = T \cup T^{\mathrm{x}}$. $R^{\mathrm{i}*}$ and $R^{\mathrm{o}*}$ extend R^{i} and R^{o} with the input and output regions of the tasks in T^{x}. D^* is constructed from D by replacing all dependencies $d \in D$, $\gamma(d) \neq \{\}$, with dependencies connecting the source and target regions of d via the regions of the newly inserted tasks $\gamma(d)$.

One further output that may also be necessary is tightly connected to the buffer dependency graph.

- A set $D^+ \subset (R^{\mathrm{i}*} \cup R^{\mathrm{o}*}) \times (R^{\mathrm{i}*} \cup R^{\mathrm{o}*})$ of *additional dependencies* must be established to ensure correct execution of the application. For instance, produce-read-modify situations that were not handled by copy or transfer tasks must be resolved by adding a dependency, i.e., for each combination of regions $r \in R^{\mathrm{o}*}$, $r', r'' \in R^{\mathrm{i}*}$ such that $(r, r'), (r, r'') \in D^*$, if r'' belongs to a merged or `inout` interface and r' does not, D^+ must contain (r', r''). Also, if a static scheduling order of the tasks is desired, it can be modelled by adding dependencies. However, no dependencies must be added that introduce cycles at task level.

While buffer dependency graph and additional dependencies define the execution semantics of the application, it must also be defined **which resources are used** for this execution. For this purpose, the following outputs must be generated, where $I^{\mathrm{i}*} = \bigcup_{t \in T^*} I_t^{\mathrm{i}}$ and $I^{\mathrm{o}*} = \bigcup_{t \in T^*} I_t^{\mathrm{o}}$ shall denote the sets of all input and all output interfaces in the application.

- A *task binding* $\beta : T \longrightarrow C$ and a transfer binding $\beta^X : T^{\mathrm{x}} \longrightarrow C^{\mathrm{x}}$ must be established to denote which execution resource executes which task.

- A *memory binding* $\beta^M : I^{\mathrm{i}*} \cup I^{\mathrm{o}*} \longrightarrow M$ must be found which specifies on which memory modules the packets are stored that will be provided to the tasks through the interfaces. The following requirements must be fulfilled. For each dependency $(r, r') \in D^*$, it must hold that the interfaces to which r and r' belong are bound to the same memory module (dependencies in the buffer dependency graph denote data exchange by buffer sharing). For each task $t \in T$, it must hold that combined input/output or merged buddy interface pairs are mapped to the same memory, i.e. $\forall(i^{\mathrm{i}}, i^{\mathrm{o}}) \in J_t,\ \beta^M(i^{\mathrm{i}}) = \beta^M(i^{\mathrm{o}})$ and that the core to which t is mapped can access the memory assigned to the interfaces, i.e. $\forall i \in I_t^{\mathrm{i}} \cup I_t^{\mathrm{o}},\ \left(\beta(t), \beta^M(i)\right) \in A$. For each $t \in T^{\mathrm{x}}$, the core or DMA controller executing t must be able to transport data from the source to the destination memory, i.e. $\left(\beta^M(i^{\mathrm{i}}), \beta^X(t), \beta^M(i^{\mathrm{o}})\right) \in X$ with $I_t^{\mathrm{i}} = \{i^{\mathrm{i}}\}$ and $I_t^{\mathrm{o}} = \{i^{\mathrm{o}}\}$.

- A *buffer allocation* $\alpha^B : I^{\mathrm{i}*} \cup I^{\mathrm{o}*} \longrightarrow \mathscr{B}$ must be made, where $\mathscr{B}$ is the set of all buffers. For each task $t \in T$, the interface pairs contained in J_t must be mapped to the same buffer, i.e. $\forall(i^{\mathrm{i}}, i^{\mathrm{o}}) \in J_t, \alpha^B(i^{\mathrm{i}}) = \alpha^B(i^{\mathrm{o}})$. All buffers must fit into the memory modules and buffer overlaps must be prevented during the buffer lifetimes. The buffers must be allocated such that for each dependency $(r^{\mathrm{o}}, r^{\mathrm{i}}) \in D^*$, the buffer addresses assigned to r_o and r_i are the same for the same elements.

The details for the buffers can be described as follows. A buffer shall be defined as an injective affine transformation of a cube to $\mathbb{N}_0$ (for each memory module $m \in M$, its address space shall be denoted as $\{0, \ldots, |m| - 1\}$). For instance, a buffer b for a cube $\mathscr{q} \in \mathbb{Q}^n$ is given as $b : \mathscr{q} \longrightarrow \mathbb{N}_0$ with $b(\boldsymbol{p}) = l + \langle \boldsymbol{p}, \boldsymbol{s} \rangle$, $l \in \mathbb{Z}$, $\boldsymbol{s} \in \mathbb{Z}^n$, $\langle \cdot, \cdot \rangle$ the scalar product, such that $\forall \boldsymbol{p}_1 \neq \boldsymbol{p}_2 \in \mathscr{q},\ b(\boldsymbol{p}_1) \neq b(\boldsymbol{p}_2)$. $b[\mathscr{q}]$ shall denote the image of b, i.e. the set of all its returned byte addresses.

For instance, to store the example packet `x` from before (which was declared as `double x[5][2][3]`) in a contiguous chunk of memory beginning at address `0x43210`, one would use the buffer $b(\boldsymbol{p}) = 0\text{x}43210 + \langle \boldsymbol{p}, (1, 8, 24, 48)^\top \rangle$.

Each buffer must fit into the corresponding memory module, i.e. for each interface i, $\max \alpha^B(i)[\mathscr{q}_i] < |\beta^M(i)|$. For each pair of regions $r_1 \neq r_2 \in R^{\mathrm{o}*}$ belonging to two interfaces i_1 and i_2, at least one of the following conditions must hold:

- They reside on different memory modules, i.e. $\beta^M(i_1) \neq \beta^M(i_2)$ or

- their buffers do not overlap, i.e. $\alpha^B(i_1)[\mathscr{q}(r_1)] \cap \alpha^B(i_2)[\mathscr{q}(r_2)] = \{\}$ or

Table 5.1: Symbols and variables used in the problem definition

Variable	Type	Explanation
A	$\subseteq C \times M$	Which core can access which memory module
$\mathscr{B}$	set	All possible buffers
C	set	All cores on a platform
C^{x}	set	All data transfer or copy resources on a platform (DMA controllers, cores, ...)
D	$\subseteq R^{\mathrm{i}} \times R^{\mathrm{o}}$	Dependencies in the packet dependency graph
D^*	$\subseteq R^{\mathrm{i}*} \times R^{\mathrm{o}*}$	Dependencies in the buffer dependency graph
D^+	$\subseteq (R^{\mathrm{i}*} \cup R^{\mathrm{o}*}) \times (R^{\mathrm{i}*} \cup R^{\mathrm{o}*})$	Additional dependencies for the buffer dependency graph
$I^{\mathrm{i}*}$	set	All input interfaces in the buffer dependency graph
$I^{\mathrm{o}*}$	set	All output interfaces in the buffer dependency graph
M	set	All memory modules on a platform
n^{r}	$I^{\mathrm{i}*} \cup I^{\mathrm{o}*} \longrightarrow \mathbb{N}$	Number of read accesses to an interface
n^{w}	$I^{\mathrm{o}*} \longrightarrow \mathbb{N}$	Number of write accesses to an interface
$\mathbb{Q}^n$	set	All n-dimensional cubes
R^{i}	set	All regions of input interfaces (packet dependency graph)
$R^{\mathrm{i}*}$	set	All regions of input interfaces (the buffer dependency graph)
R^{o}	set	All regions of output interfaces (packet dependency graph)
$R^{\mathrm{o}*}$	set	All regions of output interfaces (buffer dependency graph)
T	set	All computation tasks in the application
T^{x}	set	All transfer or copy tasks (automatically inserted)
T^*	$= T \cup T^{\mathrm{x}}$	All tasks (of any type) in the buffer dependency graph
X	$\subseteq M \times C^{\mathrm{x}} \times M$	Possibilities of transferring data from one module to another or itself using a transfer or copy resource
α^B	$I^{\mathrm{i}*} \cup I^{\mathrm{o}*} \longrightarrow \mathscr{B}$	Buffer allocated for each interface
β	$T \longrightarrow C$	Task binding (computation tasks)
β^X	$T^{\mathrm{x}} \longrightarrow C^{\mathrm{x}}$	Transport/copy task binding
β^M	$I^{\mathrm{i}*} \cup I^{\mathrm{o}*} \longrightarrow M$	Memory binding
γ	$D \longrightarrow \mathscr{P}(T^{\mathrm{x}})$	Copy/transfer tasks to be inserted for each packet dependency
τ^{e}	$T \times C \longrightarrow \mathbb{N}$	Pure computation time for executing a task on a core
τ^{r}	$A \longrightarrow \mathbb{N}$	Time for one read access
τ^{w}	$A \longrightarrow \mathbb{N}$	Time for one write access
τ^{x}	$X \times \mathbb{N} \longrightarrow \mathbb{N}$	Time for transferring or copying a given number of bytes
For each task $t \in T^*$:		
I_t^{i}	set	All input interfaces of the task
I_t^{o}	set	All output interfaces of the task
J_t	$\subseteq I_t^{\mathrm{i}} \times I_t^{\mathrm{o}}$	Merged interfaces (provided with the same buffer)
For each interface i:		
q_i	cube	Dimensions of the interface
R_i	set	Regions into which the interface is divided

- the lifespans of the (sub-)packets they carry do not overlap. These lifespans can be described by the set R_1 containing r_1 and its successors and the set R_2 of r_2 and its successors (e.g. $R_1 = \{r_1\} \cup \{\, r \mid (r_1, r) \in D^* \,\}$). R_1 is *before* R_2 if for each pair $r \in R_1, r' \in R_2$ there is either a direct or indirect dependency, given by all elements in $D^* \cup D^+$, from the task accessing r to that accessing r', or r and r' belong to the same task t and there is a $(i, i') \in J_t, r \in R_i, r' \in R_{i'}$. The lifespans do not overlap if either R_1 is before R_2 or R_2 before R_1.

Note that lifespans are defined only using dependencies and do not carry any notion of time. This is because the latter may be hard to predict and thus cannot serve as a guarantee of non-overlapping. Insertion of additional dependencies into D^+, however, can.

The buffer dependency graph and the bindings must be generated such as to optimise a specific metric. For instance, the execution time of the application would be such a metric and can be determined as follows. For each task $t \in T$, its execution time is given by the sum of the computation time on the selected core and the access times to all interfaces:

$$\tau(t) = \tau^{\mathrm{e}}(t, \beta(t)) + \sum_{i \in I_t^{\mathrm{i}} \cup I_t^{\mathrm{o}}} \tau^{\mathrm{r}}\left(\left(\beta(t), \beta^M(i)\right)\right) \cdot n^{\mathrm{r}}(i) + \sum_{i \in I_t^{\mathrm{o}}} \tau^{\mathrm{w}}\left(\left(\beta(t), \beta^M(i)\right)\right) \cdot n^{\mathrm{w}}(i). \quad (5.3)$$

The execution time of a copy or transport task $t^{\mathrm{x}} \in T^{\mathrm{x}}$ is given by

$$\tau(t^{\mathrm{x}}) = \tau^{\mathrm{x}}\left(\left(\beta^M(i^{\mathrm{i}}), \beta^X(t^{\mathrm{x}}), \beta^M(i^{\mathrm{o}})\right), |i^{\mathrm{i}}|\right) \quad \text{with} \quad I_{t^{\mathrm{x}}}^{\mathrm{i}} = \{i^{\mathrm{i}}\}, I_{t^{\mathrm{x}}}^{\mathrm{o}} = \{i^{\mathrm{o}}\}. \quad (5.4)$$

These formulae do not model possible access conflicts; such terms could, however, be added. As Chapter 4 showed, simple probabilistic models yield a satisfying accuracy.

The execution time of the entire application results from these times, considering that the dependencies in $D^* \cup D^+$ impose precedence relations on the tasks and that no two tasks may execute simultaneously on the same resource.

While most of the outputs described above are already hard to determine on their own, it is the interdependencies between them that significantly complicate the optimisation problem. For instance, all bindings β, β^X, β^M tightly depend on each other. On one hand, cores and DMA controllers must be able to access the memory modules chosen for the tasks they execute. On the other hand, the choice of the memory bindings heavily influences the execution times of the tasks, thereby possible schedules and thus the task bindings (think of memory modules with access times differing in orders of magnitude).

Another example are decisions of inserting additional copy tasks or merging buddy interfaces. On the one hand, they are largely determined by memory bindings and thus by task bindings — when data is produced on one memory module and

required later on another memory module, a transfer task must be inserted; since this creates a copy of the data, it may allow two buddy interfaces to be merged etc. On the other hand, copying can significantly increase and merging buddies can significantly decrease memory consumption, thereby influencing buffer allocation and thus memory mapping and thus task mapping. Such interdependencies, as well as individual optimisation subproblems, shall be touched upon in the next section.

5.2.3 Particular Subproblems

Some of the optimisation subproblems discussed before are well-known or at least similar to well-known problems. For instance, the question of task binding has been widely researched in different contexts. Other subproblems are specific to Ladybirds. In the following, some of these subproblems shall be regarded, discussing some of the main difficulties as well as the interdependencies with other subproblems. At first, the question of whether or not to merge buddy interfaces and where to introduce copy tasks shall be targeted. Afterwards, it will be discussed how schedules can help the rest of the optimisation and how they can be determined. Buffer-related subproblems will then form the second part of the discussion, which treats the problem of memory binding and that of buffer allocation.

5.2.3.1 Buddy and Copy Decisions

As mentioned before, for each pair of buddy interfaces, it must be decided in the optimisation process whether they are to be merged, i.e. provided with the same buffer. For each dependency in the packet dependency graph, it must be decided if and how many copy or transfer tasks are to be inserted on it. The insertion of transfer tasks is typically determined by other constraints: If an edge connects two regions that should be bound to different memory modules (for instance for performance reasons or because a core cannot access a memory module), one transfer task is required for the dependency. Memory insufficiencies on individual a memory modules may be tackled by temporarily "out-sourcing" a sub-packet to a different memory module — in such a case, two or more transfer tasks are required. These cases are part of different optimisation sub-problems and shall be regarded later on. In the following, the focus shall be put on the insertion of copy tasks (i.e. copying data within the same memory module) and on merging buddy interfaces.

Due to the sequential nature of Ladybirds C, a Ladybirds application can always be executed (resource sufficiency provided) in the sequential specification order without any additional copy tasks. It therefore makes sense to consider an implementation without any copy task and without any merged buddy interfaces as a

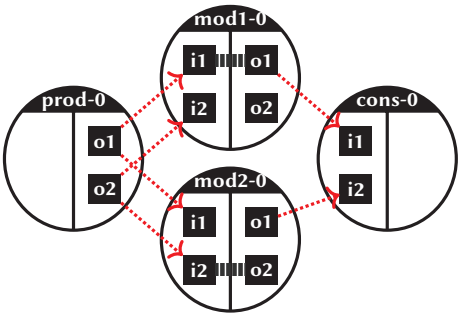


Figure 5.13:
Example for an application with mutually exclusive buddy relations

basic implementation, with copy tasks and buddy interface merging being optimisations that can be applied to it. From this angle, creating copies of packets is a trade-off: Additional memory is invested in return for higher concurrency and therefore better parallelisability, as can be easily seen in the produce-read-modify example in Figures 3.7 and 3.8 on page 98. Consequently, inserting copy tasks is mainly an option if enough memory is available and if more concurrency is required. One concrete criterion can be if the "modify" task is on a critical path of the application.

Merging buddy interfaces, on the other hand, does not directly influence execution time; it is rather an optimisation for saving memory. Indirectly, of course, this can also lead to better performance whenever space is created in fast memory modules such that more data can be placed there.

One caveat when merging buddy interfaces is that it can break certain applications. Consider the example in Figure 5.13. If the interfaces `mod1-0:i1` and `mod1-0:o1` are merged (and no copy/transfer task is inserted), `mod1-0` needs to be executed after `mod2-0` because of the write-after-read dependency between `mod2-0:i1` and `mod1-0:o1`. Similar considerations hold for `mod2-0` and its interfaces `i2` and `o2`. Merging the buddy interfaces in both tasks without inserting any additional copy/transfer tasks would thus lead to a cyclic dependency obstructing correct program execution.

Such issues are, however, easy to detect. Also, these corner cases should not disguise the common cases in which buddy interfaces can be merged without any problem, for instance when the input packets are not read by any other tasks. One could think of filtering out these simple cases beforehand and automatically merging these buddy interfaces; this may, however, be detrimental to performance on certain target platforms like Adapteva Epiphany (see below).

5.2.3.2 Finding Schedules

Many of the optimisation results mentioned earlier are hard to produce without a notion of time or precedence. For instance, establishing efficient task bindings requires a way of knowing which cores are occupied when. The same holds for transfer bindings; data transfer, however, also influences memory bindings because the efficiency of a certain memory binding may depend on packets being transferred swiftly between memory modules. Buddy and copy decisions also may depend on such factors, and clearly, buffer allocation and reallocation require timing information to know when memory can be reused. Finally, optimisation metrics typically also depend on timing, such as the application execution time. For all these reasons, an optimisation procedure will need a way of obtaining the timings of all tasks in the application, i.e., a schedule.

The following considerations will illustrate methods, difficulties and applications of scheduling using the example of list scheduling. List scheduling is a well-known and popular scheduling heuristic. Gradually advancing in scheduled time, it greedily assigns ready tasks to free resources according to a fixed priority scheme.

In the following, the assumption shall be made that a buffer dependency graph and all bindings β, β^X, β^M are given, but no buffer allocation has taken place yet (the latter cannot be established without a schedule if dependencies are sparse and memory resources are not abundant). In this case, the scheduler has to take two kinds of resources into consideration — the processing resources (cores and DMA controllers) and the memory resources. Each task (which, in the following, shall also include transfer tasks) can only be scheduled if both its processing resource is unoccupied and enough memory is available on the modules such that the buffers for all interfaces can be allocated. While the occupation of the processing resources follows a binary scheme, the memory occupation scheme is more complex. Before the start of a task $t \in T \cup T^{\mathrm{x}}$, buffers must be allocated for all its pure output interfaces, i.e. for all interfaces in I_t^{o} which are no `inout` or merged interfaces. When it finishes, it must release the buffers for each input region $r \in R_i, i \in I_t^{\mathrm{i}}$, if i is not an `inout` or merged interface and if all other tasks using the buffer have already finished. Also, upon finishing, it must release the buffers for each output region $r \in R_i, i \in I_t^{\mathrm{o}}$ if r is not used, i.e., if there is no r' s.t. $(r, r') \in D^*$.

An important issue to address is when to schedule data transfer tasks. If such a task is scheduled too early, it binds resources on the target memory that are still needed for other tasks. Scheduled too late, it binds resources on the source memory for too long. These considerations particularly apply in the case of multiple transfer tasks used for temporarily storing packets on other memory modules. One possible strategy might be to establish the schedule in two steps. In the first step, transfer tasks are scheduled but do not allocate buffers — this is instead done by the first

task that uses a buffer. In the second step, one can use the memory availability information obtained previously to schedule the transfer tasks.

Once a schedule has been established, it can be used for all the purposes mentioned before. In particular, one can now easily insert additional dependencies in D^+: With the scheduled start and end times of all tasks being known, it is safe to insert an edge if the source task ends before the destination task begins. While this may lead to performance losses in case of an inaccurate schedule, it can never deadlock the program because there exists at least one valid schedule supporting the newly inserted dependency.

5.2.3.3 Memory Binding

The problem of mapping packets to memory has been formally divided into two distinct sub-problems, memory binding β^M and buffer allocation α^B. This division is useful for a number of reasons. Firstly, it simplifies the formal problem and algorithm descriptions. Secondly, such a division can also be taken advantage of for simplifying the optimisation procedures. While there is a strong interdependency between the memory binding and other optimisation decisions, the interdependencies with buffer allocation are much weaker. As long as there is a valid solution to the latter problem, the question which exact solution is chosen does not influence application execution. Therefore, the two problems do not need to be handled simultaneously. Thirdly, a good prediction of the tractability of a buffer allocation problem can be given by simply calculating the amount of occupied space on each memory module. This prediction does not give false negatives, and false positives are only likely if the amount of occupied space comes close to the memory capacity.

Memory binding is crucial to performance on non-uniform memory access platforms, especially on all platforms with memory hierarchies. Therefore, good algorithms for obtaining such a binding are important. At the same time, the formulae for calculating task execution times, (5.3) and (5.4), constitute an easily applicable tool to reliably detect the implications of important mapping decisions. It therefore does not seem too unreasonable to be optimistic that already simple optimisation algorithms may yield acceptable results for typical applications.

In practice, cores typically have a set of small memory modules near to them such that the access is fast. On some systems (like the Kalray MPPA), the cores cannot even access other memories; for these systems memory binding is trivial (leaving aside possible access conflicts). On other systems, their are further memory modules a core can access, but typically at a considerably higher latency. On these systems, as a rule of thumb, all (sub-)packets accessed frequently should be stored on the near memory modules and packets accessed sporadically can reside on further away memory modules. As mentioned before, (5.3) quantifies this.

In all cases, the problem may arise that not enough memory may be available on the memory near to a core or on all memory modules a core can access. Since especially the fast scratch-pad memories are typically small, such a situation is not unlikely. Apart from changing the task binding, there are two methods to resolve such a situation. They shall be called *schedule relaxation* and *spilling*. To know which method must be applied in what cases, it is important to distinguish between *live* and *active* (sub-)packets. Similar to the concept of live variables, a (sub-)packet is live from the execution start of the task writing it to the execution end of the last task reading it. A (sub-)packet is active only during the execution of any task accessing it. That means an active (sub-)packet is always live, but a live (sub-)packet can be inactive between the executions of tasks accessing it.

If at a given point in time, the set of active (sub-)packets does not fit into the memory, it helps to relax the schedule by executing some tasks later in time, thereby reducing concurrency and thus simultaneous memory requirements. If at a given point in time, the set of active, but not that of live packets fits into the memory, it helps to spill inactive (sub-)packets, i.e. to temporarily move them to a different memory module. Together, the methods of schedule relaxation and spilling always make the execution of an application possible (provided enough spilling memory and individual tasks with sufficiently low memory requirements), in the extreme case by sequentialising task execution.

5.2.3.4 Buffer allocation

Previously, it has been shown how memory binding and buffer allocation can be separated. The following considerations will now focus on the subproblem of finding a valid buffer allocation for a given memory binding and a given schedule, provided that the capacity of each memory module is higher than the sum of the sizes of the live packets bound to it at any point in time. Since this task can be handled for each memory module individually, the focus will be on a single memory module in the following.

As discussed earlier, for two different (sub-)packets bound to the same memory module, it must hold that either their lifespans or their buffers or both do not overlap. Lifespans were defined with respect to the dependencies in $D^* \cup D^+$; however, these may not yield enough potential for reusing memory. Information from the schedule needs to be used in these cases, but such decisions must be backed up by inserting additional dependencies into D^+. To avoid overly high overhead and performance implications through these dependencies, their number should be kept low. In the temporal domain, this can be achieved by building chains and trees of dependencies, such that a region does not need dependencies from all possible re-

gions prior to it. In the spatial domain, dependencies are dispensable between two regions if their buffers do not overlap.

A method for checking buffer overlaps is also needed for those regions with overlapping lifespans. Thus, this problem shall be discussed here. Let $q_1 \in \mathbb{Q}^{n_1}$ and $q_2 \in \mathbb{Q}^{n_2}$ be two cubes of same or different dimensions. Let $b_1, b_2 \in \mathcal{B}$ be two buffers assigned to q_1 and q_2, respectively, with $b_k(\boldsymbol{p}) = l_k + \langle \boldsymbol{s}^k, \boldsymbol{p} \rangle$ for $k \in \{1, 2\}$. The question to answer is now if there exists an $\boldsymbol{x}^1 \in q_1$ and an $\boldsymbol{x}^2 \in q_2$ such that $b_1(\boldsymbol{x}^1) = b_2(\boldsymbol{x}^2)$. Inserting the definitions of b_1 and b_2 yields

$$l_1 + \langle \boldsymbol{s}^1, \boldsymbol{x}^1 \rangle = l_2 + \langle \boldsymbol{s}^2, \boldsymbol{x}^2 \rangle$$

or

$$\langle \boldsymbol{x}, \boldsymbol{s} \rangle = l_2 - l_1 \quad \text{with} \quad \boldsymbol{x} = \begin{pmatrix} \boldsymbol{x}^1 \\ \boldsymbol{x}^2 \end{pmatrix} \quad \text{and} \quad \boldsymbol{s} = \begin{pmatrix} \boldsymbol{s}^1 \\ -\boldsymbol{s}^2 \end{pmatrix}.$$

This is a linear diophantine equation. This type of equations also appears in the context of loop dependence analysis in compilers; several approaches to that problem have been established and can be used for the present problem (a list can be found for instance in [Muc97, Section 9.8]). Two simple methods specifically for the buffer overlap problem are as follows.

- A simple test that can quickly exclude overlaps, typically in a vast majority of cases, is to compare the lowest and highest buffer addresses $l_k^l := b_k(\boldsymbol{q}_k^l)$ and $l_k^u := b_k(\boldsymbol{q}_k^u)$ for $k \in \{1, 2\}$. If $[l_1^l, l_1^u) \cap [l_2^l, l_2^u) = \{\}$, an overlap is not possible.
- Based on the fact that the vectors $\boldsymbol{s}^k, k \in \{1, 2\}$, are structured such that s_j^k divides s_{j+1}^k (cf. Section 3.5), one can take advantage of the periodicity patterns in a buffer's image (see Table 5.2 on the following page). More concretely, if there is a pair $j_1, j_2 \in \mathbb{N}$ such that $s_{j_1}^1 = s_{j_2}^2$, one can compare the patterns with respect to this period. Let

  $$l_k^l := \left(l_k + (\boldsymbol{q}_k^l)_{(j_k-1)} \cdot s_{j_k-1}^k\right) \bmod s_{j_k}^k \text{ and } l_k^u := \left(l_k + (\boldsymbol{q}_k^u)_{(j_k-1)} \cdot s_{j_k-1}^k\right) \bmod s_{j_k}^k,$$

  $k \in \{1, 2\}$, denote the lower and bounds of the two periodic address occupations with period $s_{j_1}^1 = s_{j_2}^2$. If these occupations do not overlap, i.e., if $l_1^l < l_1^u \leqslant l_2^l < l_2^u$ or $l_1^u \leqslant l_2^l < l_2^u \leqslant l_1^l$ or $l_2^l < l_2^u \leqslant l_1^l < l_1^u$ or $l_2^u \leqslant l_1^l < l_1^u \leqslant l_2^l$, the images of the buffers do not overlap for the given cubes. This is particularly interesting for optimisation because it gives a simple method of adjusting $l_1, l_2, \boldsymbol{s}^1$ or $\boldsymbol{s}^2$ such as to avoid overlaps.

The reason why buffer overlap is analysed at the level of regions and not of interfaces is that packets do not always need to be fully present in memory. An extreme

Table 5.2: Memory occupation patterns for a sub-packet `y[1,2][0,2][1]` on the buffer $b(\boldsymbol{p}) = [base] + \langle \boldsymbol{p}, (1, 8, 16, 80)^\top \rangle$. The occupation of the linear address space can be visualised by considering each dimension of the sub-packet separately. Each dimension creates a regular occupation pattern, and the actual memory occupation can be obtained by combining all pattern with a boolean AND.

Dimension	indices	byte factor	period	occupation pattern (coloured addresses are occupied)
1	$[0, 8)$	1	8	
2	$[1, 2)$	8	16	
3	$[0, 3)$	16	80	
4	$[1, 3)$	80	—	
resulting occupation				

Listing 5.1: Ladybirds C specification of an application gradually replacing buffer contents. The type of the packets (in, out, …) requested by the kernels are as stated in the comments.

```
char img[16][128];

AcquireImg(/*out*/ img);
for(genvar int i = 0; i < 16; ++i)
{
   TransformRow(/*inout*/ img[i]);
}
DisplayImg(/*in*/ img);
```

example of this is given in Listing 5.1. It shall be assumed that no copy or transfer tasks have been inserted during optimisation. At the beginning of the code, the buffer allocated for `img` contains the output packet of `AcquireImg`. At the end of the code, it contains the input packet of `DisplayImg`. In between, however, some parts of it may contain sub-packets of the output of `AcquireImg` while others already contain sub-packets of the input of `DisplayImg`. In this example, this memory alignment is induced by the specification. Similar arrangements might, however, also be purely the result of an optimised buffer allocation. This is intended in Ladybirds as it allows for efficient use of memory.

A question that might arise is how, for a complex buffer dependency graph with interfaces consisting of many regions and multiple cross-dependencies between different tasks, the buffers have to be dimensioned and allocated and how the regions have to be placed there such that the address constraints of all dependencies are fulfilled. With the possibilities of merging multiple task outputs to one task input or of specifying sub-packets, particularly such with reduced dimensionalities, this may become an intricate multi-dimensional puzzle. However, since Ladybirds C only allows entire packets and sub-packets thereof, but no compositions of (sub-)packets, as kernel arguments, there always exists a solution to this problem. It can always be obtained by using the packet declarations and sub-packet indices in the specification as an orientation.

5.2.4 Application to Different Target Platforms

Previously, optimisation methods and goals have been discussed on a formal level. In this section, two concrete target platform architectures shall be considered: The P2012/PULP architecture family and Adapteva Epiphany. For both architectures, the most important programming hints and thus optimisation goals will be given and it will be discussed how to achieve these goals using Ladybirds optimisation models.

5.2.4.1 P2012 and PULP

The P2012 [Ben+12] as well as the PULP [Con+15] platform consist of clusters of multiple processing elements. Each cluster has a fast L1 scratchpad memory for storing data, and slower, but larger L2 memory is shared amongst the clusters. External L3 memory may be connected as well. DMA controllers manage the data transport between the different memory modules.

On such a platform architecture, as already mentioned, it is important to have frequently accessed data in the L1 memories. For efficient execution, it is recommended to establish a kind of pipeline of transferring input data to the L1 memories via DMA, processing it there and transferring back outputs. Similar considerations hold for transferring data directly between clusters. A more abstract formulation of this is that data transfer and processing must be performed in parallel such that the cores can continuously perform calculations and do not need to wait for data.

As discussed before, the difference between binding (sub-)packets to L1 and L2 memory can be quantitatively evaluated using (5.3). An optimisation algorithm thus has the information it needs to arrange for a good memory binding.

Concurrency between processing and data transfer is provided by the Ladybirds optimisation and implementation models. A data transfer can be scheduled as soon as the data is available, the target buffer is free and a DMA controller is unoccupied.

Each of these conditions starts to be fulfilled at the end of a task, when it can easily be detected by a runtime manager which then can immediately launch the data transfer. The data that must be transferred is explicit from the program specification.

It is, however, important to ensure that the target buffers for data transfers are free soon enough for the transfers to be completed in time. This may require certain (sub-)packets to be bound to L2 or even L3 instead of L1 memory. It might therefore be a good idea to clearly prioritise frequently accessed (sub-)packets in an optimisation algorithm, also taking account of the time required for their transfer. Finally, aggressive merging of buddy interfaces may be advantageous for saving L1 memory and therefore fitting more (sub-)packets in it.

5.2.4.2 Adapteva Epiphany

The Adapteva Epiphany architecture [Olo+11] consists of independent tiles that are connected over a network on chip. Each tile contains a processing element, a DMA controller and four memory banks, which store code as well as data. Each core can access each memory bank on the chip. Accesses to the local banks (those on the same tile) take place within one cycle; accesses to banks from other tiles are routed through the network on chip. Write accesses are performed asynchronously, i.e., a write request is posted within one cycle and the core immediately resumes its operation. Read accesses, on the other hand, involve sending a read request and waiting for the data: This operation takes multiple cycles, depending on the geometrical distance of the other tile.

An efficient program should place data that is frequently read on the same tile as the processing element that reads it; alternatively, at least on a tile that is geometrically near. Doing so does not only reduce the latency for a read operation but also the traffic and therefore the congestion potential on the network on chip. Since writing is asynchronous, it is not always necessary to only write to the local memory. In fact, one can often take advantage of this "free" data transport mechanism.

For a Ladybirds optimiser, this means that merging buddy interfaces is not always a good idea. In particular, when no read accesses happen to an output interface, much better performance can be achieved by mapping the latter to a memory bank on the tile of a core that later reads the produced packet. In general, output interfaces with low read counts are flexible: A promising, natural approach might therefore be to attribute higher priority to input interfaces when establishing a memory binding. When buddy interfaces are bound to the same memory module, the algorithm might then decide to merge them.

One particularity of the Epiphany architecture are certainly the complex interdependencies between task binding and data transfers. The task binding should be established such that those tasks exchanging large amounts of data are bound to

cores geometrically near to each other. The Ladybirds specification model, together with the timings of the tasks, provides the necessary information to predict the amounts of data exchanged during a given period of time. Using this information could be advantageous to speed up read operations and to reduce overall traffic and therefore congestions on the network-on-chip.

Finally, within each tile, the (sub-)packets need to be distributed between the different banks. An algorithm similar to the one introduced in Section 5.1 could be applied for this purpose.

5.2.5 Conclusion

The sections above covered problems and approaches related to the general case of optimising Ladybirds applications for multicore platforms, in particular for platforms with complex memory architectures. A formal model of Ladybirds application specifications with additional informations on runtimes and interface access counts was given as well as a model for describing hardware architectures. Based on these models, all required outputs of an application optimisation procedure were formalised. Details were discussed for the subproblems of buddy and copy decisions, scheduling algorithms, memory binding and buffer allocation. In several cases, possible solution approaches or ideas were outlined. Finally, P2012 and PULP on the on hand and Adapteva Epiphany on the other hand were shown as platform examples and it was discussed how the presented concepts could be applied to these architectures.

Although the platform model is reduced to the essentials and only few data about the application is needed in addition to the specification, the discussions have shown how many optimisations come to mind with the Ladybirds optimisation model. It was also shown that the generic and still sufficiently detailed descriptions of both application and hardware architecture gave room to tailor the applications to a large range of platforms and their particular requirements without making major modifications to the optimisation algorithms. What remains to be shown is that there exists an algorithm that can perform all the optimisations discussed in this chapter, producing a solution that is correct with respect to the definition from Section 5.2.2 and achieves good performance. Different optimisation methods like evolutionary algorithms, simulated annealing or even greedy heuristics come to mind here; since Ladybirds applications can be comfortably debugged on normal PCs and parallel implementations of correct Ladybirds applications are correct by construction, there is no pressing need for short optimisation time. Also, the Ladybirds implementation model unlocks optimisation potential that is not yet accessible to existing optimisation methods, for instance, it fully supports the shared memory data exchange

philosophy. Therefore, if an optimisation algorithm is found that can produce a correct solution, one could be optimistic that the resulting code is more efficient than existing auto-generated application implementations, in particular in the case of high data exchange rates.

6

Closing Remarks

This thesis tried to find answers to the question of how parallel programs could be optimised for modern multicore architectures, in particular with respect to storing and transporting data efficiently. It identified four important fields of work given as specification, optimisation, implementation and platform models.

The influence of **implementation models** was evaluated for the cases of the CAL actor language and for Kahn process networks on the Intel Xeon Phi platform. Inefficient implementation models were shown to lead to inefficient program execution. Methods were given for getting to more efficient implementation models, but this turned out to be problematic because of strong implications on the adopted specification model.

As a result, a new **specification model** was conceived with efficient data storage and transfer implementation primitives in mind. The Ladybirds model set consists of a specification model, a basic optimisation model and an implementation model that were devised together. Using a specification language based on C with some MatLab-like array functionality, the programmer can comfortably specify a parallelisable application as a set of tasks with clearly defined inputs and outputs together with data dependencies between them. Ladybirds applications can be debugged as single-threaded PC programs and later optimised for parallel execution. It was also demonstrated that the Ladybirds is not only useful in the area of multicore architectures, but can also be applied to minimise state retention overhead in transient systems.

Different sophisticated **optimisation models and methods** for Ladybirds were discussed. The problem of distributing data between multiple memory banks in non-interleaved memories was successfully addressed with the Ladybirds optimisation model. As a generalisation, the problem of optimising Ladybirds applications

for arbitrary platforms and their requirements was formally described and different aspects of it were discussed in greater detail.

In the area of **platform models**, the characteristics of interleaved memory were studied. Probabilistic models were proposed and evaluated, giving valuable hints of how to describe these components and predict their performance in optimisation algorithms.

Even if a complete algorithm for optimising data-intensive applications on memory hierarchies or non-uniform memory access platforms remains to be subject of further research, this work has contributed important results and shown a possible path towards this goal. Reaching the latter could constitute a decisive step towards more optimisation automation, less dedicated data management hardware and higher energy efficiency in embedded systems.

Besides of a generic Ladybirds optimisation algorithm, further extensions to Ladybirds could certainly be interesting.

- One clear deficiency of the Ladybirds specification model is that no loops can be specified in metakernels. Such a functionality would be important for many applications. If an Ladybirds optimisation algorithm as discussed previously existed, it might be extended to handle loops by making use of well-known compiler techniques like modulo scheduling. The question of how to handle nested loops might be more difficult in this context, but even without loop nesting, the possibility of specifying simple loops in metakernels would be a substantial improvement. A question deserving special attention in this context is that of the exact optimisation goal for the loop. It may be worthwhile to add programming directives that allow the programmer to specify throughput and/or latency constraints and to indicate whether a loop should be optimised for throughput or latency.

- Another possible extension that goes into the same direction might be support for if-else branches in metakernels. This would allow to better handle cases in which either one or another given set of tasks is executed, avoiding unnecessary memory allocation and producing more accurate schedules. Polyhedral analysis techniques might me of significant help, for instance in detecting regular patterns over multiple iterations.

- Assignment statements in metakernels might be useful as well, allowing, for instance, the programmer to assign the values of one (sub-)packet to another. This would make it possible to comfortably pass the same data in multiple kernel calls with combined input/output arguments. The optimiser could then decide where copies are necessary. The difficulty of this feature is that

it could significantly complicate buffer allocation (think of the multidimensional puzzle explained in Section 5.2.3.4).

- If loops can be specified in Ladybirds metakernels, a different approach to implementing them might be to produce Kahn process networks that make use of deterministic memory sharing. The slight overhead as compared to the Ladybirds implementation model might be outweighed in certain cases by the larger flexibility of dynamic execution. It would be interesting to find out if a Kahn process network shows more robustness towards large variations in task execution times or jitters in the input.
- An important feature to be considered on the optimisation side is task granularity. In the current version, all metakernels are flattened, i.e., replaced with their definition, such that a large graph of possibly small tasks is obtained. Depending on the target cores, it might be advantageous to execute entire metakernels as a whole, i.e. to stop the flattening process before the leaves in the call tree are reached.
- Finally, GPUs may be considered as target platforms. This is mostly an implementational issue, since GPUs are not programmed in plain C but in special languages like those provided by CUDA or OpenCL. Also, the degree of parallelism between the processing elements on GPU clusters is more fine-grained; one would therefore have to see if it makes more sense to work with tiny Ladybirds kernels or rather to transform loops within the kernels. From their architecture, however, GPUs would fit well to the Ladybirds model set.

Bibliography

[Aug+11] Cédric Augonnet, Samuel Thibault, Raymond Namyst and Pierre-André Wacrenier. 'StarPU: a unified platform for task scheduling on heterogeneous multicore architectures'. In: *Concurrency and Computation: Practice and Experience* 23.2 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631

[BC70] Gerald J. Burnett and Edward G. Coffman. 'A study of interleaved memory systems'. In proc.: *AFIPS spring joint computing conference*, May 1970, ACM, pp. 467–474. DOI: 10.1145/1476936.1477008

[BC73] Gerald J. Burnett and Edward G. Coffman. 'A Combinatorial Problem Related to Interleaved Memory Systems'. In: *Journal of the ACM* 20.1 (Jan. 1973), pp. 39–45. DOI: 10.1145/321738.321742

[BC82] Gérard Berry and Pierre-Louis Curien. 'Sequential algorithms on concrete data structures'. In: *Theoretical Computer Science* 20.3 (July 1982), pp. 265–321. DOI: 10.1016/s0304-3975(82)80002-9

[BCG00] Albert Benveniste, Benoît Caillaud and Paul Le Guernic. 'Compositionality in Dataflow Synchronous Languages'. In: *Information and Computation* 163.1 (Nov. 2000), pp. 125–171. DOI: 10.1006/inco.2000.9999

[Bec+16] Matthias Becker, Dakshina Dasari, Borislav Nicolic et al. 'Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform'. In proc.: *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, IEEE. DOI: 10.1109/ecrts.2016.14

[Ben+12] Luca Benini, Eric Flamand, Didier Fuin and Diego Melpignano. 'P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator'. In proc.: *Design, Automation & Test in Europe (DATE)*, Mar. 2012, IEEE, pp. 983–987. DOI: 10.1109/date.2012.6176639

[BF73] Dileep P. Bhandarkar and Samuel H. Fuller. 'Markov chain models for analyzing memory interference in multiprocessor computer systems'. In proc.: *Computer architecture (ISCA)*, 1973, ACM. DOI: 10.1145/800123.803965

[BH01] Twan Basten and Jan Hoogerbrugge. 'Efficient execution of process networks'. In: *Communicating Process Architectures 2001, WoTUG-24*. Amsterdam: IOS, 2001, pp. 1–14. *https://www.semanticscholar.org/paper/Efficient-Execution-of-Process-Networks-Chalmers-Mirmehdi/7df06f3d987124b3a6ef1a4c139844ded2ec1cd9*

[Bha75] Dileep P. Bhandarkar. 'Analysis of Memory Interference in Multiprocessors'. In: *IEEE Transactions on Computers* 100.9 (Sept. 1975), pp. 897–908. DOI: 10.1109/t-c.1975.224335

[Bil+96] Greet Bilsen, Marc Engels, Rudy Lauwereins and Jean Peperstraete. 'Cyclo-static dataflow'. In: *IEEE Transactions on Signal Processing* 44.2 (Feb. 1996), pp. 397–408. DOI: 10.1109/78.485935

[Bin+11] Nathan Binkert, Somayeh Sardashti, Rathijit Sen et al. 'The gem5 simulator'. In: *ACM SIGARCH Computer Architecture News* 39.2 (Aug. 2011), pp. 1–7. DOI: 10.1145/2024716.2024718

[Blu+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul et al. 'Cilk: An efficient multithreaded runtime system'. In: *Parallel and Distributed Computing* 37.1 (Aug. 1996), pp. 55–69. DOI: 10.1006/jpdc.1996.0107

[BM16] Naveed Bhatti and Luca Mottola. 'Efficient state retention for transiently-powered embedded sensing'. In proc.: *Embedded Wireless Systems and Networks (EWSN)*, 2016, Junction Publishing, pp. 137–148. HDL: 11311/1027579

[Bou+09] Jani Boutellier, Christophe Lucarz, Sébastien Lafond et al. 'Quasi-Static Scheduling of CAL Actor Networks for Reconfigurable Video Coding'. In: *Signal Processing Systems* 63.2 (July 2009), pp. 191–202. DOI: 10.1007/s11265-009-0389-5

[Bra00] Gary Bradski. 'The OpenCV Library'. In: *Dr. Dobb's Journal of Software Tools* (2000). *http://www.drdobbs.com/open-source/the-opencv-library/184404319*

[BRS12] Jani Boutellier, Mickaël Raulet and Olli Silvén. 'Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC-CAL Dataflow Programs'. In: *Signal Processing Systems* 71.1 (May 2012), pp. 35–40. DOI: 10.1007/s11265-012-0676-4

[BS76] Forest Baskett and Alan Jay Smith. 'Interference in multiprocessor computer systems with interleaved memory'. In: *Communications of the ACM* 19.6 (June 1976), pp. 327–334. DOI: 10.1145/360238.360243

[Buc93] Joseph T. Buck. 'Scheduling Dynamic Dataflow Graphs With Bounded Memory Using The Token Flow Model'. PhD thesis. University of California, Berkeley, 1993. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.7915*

[Car+14] Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru et al. 'Static Mapping of Real-Time Applications onto Massively Parallel Processor Arrays'. In proc.: *Application of Concurrency to System Design*, June 2014, IEEE. DOI: 10.1109/acsd.2014.19

[Cha82] Gregory J. Chaitin. 'Register allocation & spilling via graph coloring'. In proc.: *SIGPLAN symposium on Compiler construction*, 1982, ACM, pp. 98–105. DOI: 10.1145/800230.806984

[CHB07] Eric Cheung, Harry Hsieh and Felice Balarin. 'Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip'. In proc.: *High Level Design Validation and Test Workshop*, 2007, IEEE, pp. 37–44. DOI: 10.1109/hldvt.2007.4392782

[Cho+07] Youngchul Cho, Nacer-Eddine Zergainoh, Ahmed A. Jerraya and Kiyoung Choi. 'Buffer Size Reduction through Control-Flow Decomposition'. In proc.: *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug. 2007, IEEE, pp. 183–190. DOI: 10.1109/rtcsa.2007.25

[Chr14] George Chrysos. 'Intel Xeon Phi coprocessor — the architecture'. In: *Intel Whitepaper* 176 (2014).

[CM16] Vishwanathan Chandru and Frank Mueller. 'Reducing NoC and Memory Contention for Manycores'. In: *Architecture of Computing Systems. ARCS*. Springer, 2016, pp. 293–305. DOI: 10.1007/978-3-319-30695-7_22

[Con] Francesco Conti. *CConvNet open source project. https://micrel-web-services.dei.unibo.it/brain-inspired/cconvnet-release*

[Con+15] Francesco Conti, Davide Rossi, Antonio Pullini et al. 'PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision'. In: *Signal Processing Systems* 84.3 (Nov. 2015), pp. 339–354. DOI: 10.1007/s11265-015-1070-9

[CPB14] Francesco Conti, Antonio Pullini and Luca Benini. 'Brain-Inspired Classroom Occupancy Monitoring on a Low-Power Mobile Platform'. In proc.: *Computer Vision and Pattern Recognition Workshops*, June 2014, IEEE. DOI: 10.1109/cvprw.2014.95

[CPW02] Jeonghun Cho, Yunheung Paek and David Whalley. 'Efficient register and memory assignment for non-orthogonal architectures via graph coloring and MST algorithms'. In proc.: *Languages, compilers and tools for embedded systems / Software and compilers for embedded systems (LCTES-SCOPES)*, 2002, ACM, pp. 130–138. DOI: 10.1145/513829.513853

[DB08] Leonardo De Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24

[DD89] Chris J. Date and Hugh Darwen. *A guide to the SQL Standard: A user's guide to the standard relational language SQL*. Addison-Wesley, 1989.

[DG08] Jeffrey Dean and Sanjay Ghemawat. 'MapReduce: Simplified data processing on large clusters'. In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. DOI: 10.1145/1327452.1327492

[Dij59] Edsger W. Dijkstra. 'A note on two problems in connexion with graphs'. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. DOI: 10.1007/bf01386390

[Din+14] Benoit Dupont de Dinechin, Duco van Amstel, Marc Poulhies and Guillaume Lager. 'Time-critical computing on a single-chip massively parallel processor'. In proc.: *Design, Automation & Test in Europe (DATE)*, 2014, IEEE Conference Publications, pp. 1–6. DOI: 10.7873/date.2014.110

[DM98] Leonardo Dagum and Ramesh Menon. 'OpenMP: an industry standard API for shared-memory programming'. In: *Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313

[EJ03] Johan Eker and Jörn Janneck. *CAL Language Report*. Tech. rep. University of California at Berkeley, Dec. 2003. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.368.8244*

[Ers+12] Johan Ersfolk, Ghislain Roquier, Johan Lilius and Marco Mattavelli. 'Scheduling of dynamic dataflow programs based on state space analysis'. In proc.: *Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2012, IEEE, pp. 1661–1664. DOI: 10.1109/icassp.2012.6288215

[FCO90] John T. Feo, David C. Cann and Rodney R. Oldehoeft. 'A report on the Sisal language project'. In: *Parallel and Distributed Computing* 10.4 (Dec. 1990), pp. 349–366. DOI: 10.1016/0743-7315(90)90035-n

[Fel68] Willliam Feller. *An Introduction to Probability Theory and Its Applications*. Vol. 1. Wiley, 1968.

[FH04] Christian Ferdinand and Reinhold Heckmann. 'aiT: Worst-Case Execution Time Prediction by Static Program Analysis'. In: *Building the Information Society*. Springer, 2004, pp. 377–383. DOI: 10.1007/978-1-4020-8157-6_29

[Gab+04] Edgar Gabriel, Graham E. Fagg, George Bosilca et al. 'Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation'. In proc.: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Sept. 2004, Springer, pp. 97–104. DOI: 10.1007/978-3-540-30218-6_19

[Gha+14] Amanullah Ghazi, Jani Boutellier, Mahmoud Abdelaziz et al. 'Low power implementation of digital predistortion filter on a heterogeneous application specific multiprocessor'. In proc.: *Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, IEEE, pp. 8336–8340. DOI: 10.1109/icassp.2014.6855227

[Gia+14] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang and Lothar Thiele. 'Mapping mixed-criticality applications on multi-core architectures'. In proc.: *Design, Automation & Test in Europe (DATE)*, 2014, IEEE Conference Publications, 98:1–98:6. DOI: 10.7873/date2014.111

[Goe+16] Andrés Goens, Jeronimo Castrillon, Maximilian Odendahl and Rainer Leupers. 'An optimal allocation of memory buffers for complex multicore platforms'. In: *Systems Architecture* 66-67 (May 2016), pp. 69–83. DOI: 10.1016/j.sysarc.2016.05.002

[Gom+16] Andres Gomez, Lukas Sigrist, Michele Magno et al. 'Dynamic Energy Burst Scaling for Transiently Powered Systems'. In proc.: *Design, Automation & Test in Europe (DATE)*, 2016. DOI: 10.3850/9783981537079_0403

[Goo01] Kees Goossens. 'A protocol and memory manager for on-chip communication'. In proc.: *Circuits and Systems (ISCAS)*, May 2001, IEEE, pp. 225–228. DOI: 10.1109/iscas.2001.921048

[GR14] Susheel Gautham and Erik Rainey. 'The Khronos OpenVX 1.0 Specification'. In: (2014). *https://www.khronos.org/openvx/*

[Gut+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst et al. 'MiBench: A free, commercially representative embedded benchmark suite'. In proc.: *Workshop on Workload Characterization (WWC)*, 2001, IEEE, pp. 3–14. DOI: 10.1109/wwc.2001.990739

[HGT07] Kai Huang, David Grunert and Lothar Thiele. 'Windowed FIFOs for FPGA-based Multiprocessor Systems'. In proc.: *Application-specific Systems, Architectures and Processors (ASAP)*, July 2007, IEEE, pp. 36–41. DOI: 10.1109/asap.2007.4429955

[Jeo+12] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo et al. 'Balancing DRAM locality and parallelism in shared memory CMP systems'. In proc.: *High-Performance Comp Architecture*, Feb. 2012, IEEE, pp. 1–12. DOI: 10.1109/hpca.2012.6168944

[JK77] Norman L. Johnson and Samuel Kotz. *Urn models and their application: an approach to modern discrete probability theory*. Wiley, 1977, pp. 109f.+114.

[Kah74] Gilles Kahn. 'The Semantics of a Simple Language for Parallel Programming'. In proc.: *IFIP Congress in Information Processing*, 1974, North Holland, pp. 471–475. *https://www.semanticscholar.org/paper/The-Semantics-of-Simple-Language-for-Parallel-Kahn/d42a29e6977c28f7bf23d63b00c48f2e9100403e*

[KB03] Ming-Yung Ko and Shuvra S. Bhattacharyya. 'Partitioning for DSP Software Synthesis'. In: *Software and Compilers for Embedded Systems. SCOPES*. Springer, 2003, pp. 344–358. DOI: 10.1007/978-3-540-39920-9_24

[Kim+10] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava and Yunheung Paek. 'Operation and data mapping for CGRAs with multi-bank memory'. In: *ACM SIGPLAN Notices* 45.4 (Apr. 2010), pp. 17–26. DOI: 10.1145/1755951.1755892

[KK07] Taewhan Kim and Jungeun Kim. 'Integration of Code Scheduling, Memory Allocation, and Array Binding for Memory-Access Optimization'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.1 (Jan. 2007), pp. 142–151. DOI: 10.1109/tcad.2006.882639

[KM66] Richard M. Karp and Raymond E. Miller. 'Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing'. In: *SIAM Journal on Applied Mathematics* 14.6 (Nov. 1966), pp. 1390–1411. DOI: 10.1137/0114108

[KM76] Gilles Kahn and David B. MacQueen. *Coroutines and Networks of Parallel Processes*. Tech. rep. INRIA, 1976. *https://hal.archives-ouvertes.fr/inria-00306565v1*

[Kur+17] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi et al. 'HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA'. In: *CoRR* abs/1712.06497 (2017). arXiv: 1712.06497.

[LA04] Chris Lattner and Vikram Adve. 'LLVM: A compilation framework for lifelong program analysis & transformation'. In proc.: *Code Generation and Optimization (CGO)*, Mar. 2004, IEEE, pp. 75–86. DOI: 10.1109/cgo.2004.1281665

[LC10] Rainer Leupers and Jeronimo Castrillon. 'MPSoC programming using the MAPS compiler'. In proc.: *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2010, IEEE, pp. 897–902. DOI: 10.1109/aspdac.2010.5419677

[Lee91] Edward A. Lee. 'Consistency in dataflow graphs'. In: *IEEE Transactions on Parallel and Distributed Systems* 2.2 (Apr. 1991), pp. 223–235. DOI: 10.1109/71.89067

[Liu+12] Lei Liu, Zehan Cui, Mingjie Xing et al. 'A software memory partition approach for eliminating bank-level interference in multicore systems'. In proc.: *Parallel architectures and compilation techniques (PACT)*, 2012, ACM, pp. 367–376. DOI: 10.1145/2370816.2370869

[Liu+16] Yongpan Liu, Zhibo Wang, Albert Lee et al. '4.7 A 65nm ReRAM-enabled nonvolatile processor with 6× reduction in restore time and 4× higher clock frequency using adaptive data retention and self-write-termination nonvolatile logic'. In proc.: *Solid-State Circuits Conference (ISSCC)*, Jan. 2016, IEEE, pp. 84–86. DOI: 10.1109/isscc.2016.7417918

[LK01] Rainer Leupers and Daniel Kotte. 'Variable partitioning for dual memory bank DSPs'. In proc.: *Acoustics, Speech, and Signal Processing*, 2001, IEEE, 1121–1124 vol.2. DOI: 10.1109/icassp.2001.941118

[LM87] Edward A. Lee and David G. Messerschmitt. 'Synchronous data flow'. In: *Proceedings of the IEEE* 75.9 (Sept. 1987), pp. 1235–1245. DOI: 10.1109/proc.1987.13876

[LP95] Edward A. Lee and Thomas M. Parks. 'Dataflow process networks'. In: *Proceedings of the IEEE* 83.5 (May 1995), pp. 773–801. DOI: 10.1109/5.381846

[MAR10] Marco Mattavelli, Ihab Amer and Mickaël Raulet. 'The Reconfigurable Video Coding Standard'. In: *IEEE Signal Processing Magazine* 27.3 (May 2010), pp. 159–167. DOI: 10.1109/msp.2010.936032

[MF08] Alastair Murray and Björn Franke. 'Fast source-level data assignment to dual memory banks'. In proc.: *Software & compilers for embedded systems (SCOPES)*, 2008, ACM, pp. 43–52. DOI: 10.1145/1361096.1361105

[Mi+10] Wei Mi, Xiaobing Feng, Jingling Xue and Yaocang Jia. 'Software-Hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors'. In: *Network and Parallel Computing*. Vol. 6289. LNCS. Springer, 2010, pp. 329–343. DOI: 10.1007/978-3-642-15672-4_28

[Mis38] Richard von Mises. 'Über Aufteilungs- und Besetzungswahrscheinlichkeiten'. In: *Revue de la Faculté de Sciences de l'Université d'Istanbul* (1938–1939), pp. 145–163.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. *https://books.google.ch/books?id=Pq7pHwG1_OkC*

[Mun09] Aaftab Munshi. 'The OpenCL specification'. In proc.: *Hot Chips 21 Symposium (HCS)*, Aug. 2009, IEEE. DOI: 10.1109/hotchips.2009.7478342

[NYP16] Vincent Nélis, Patrick Meumeu Yomsi and Luis Miguel Pinho. 'The variability of application execution times on a multi-core platform'. In proc.: *Worst-Case Execution Time Analysis (WCET)*, 2016. *http://www.cister.isep.ipp.pt/docs/the_variability_of_application_execution_times_on_a_multi_core_platform/1224/attach.pdf*

[OH00] Hyunok Oh and Soonhoi Ha. 'Data memory minimization by sharing large size buffers'. In proc.: *Asia south pacific design automation conference (ASPDAC)*, 2000, ACM, pp. 491–496. DOI: 10.1145/368434.368768

[OH04] Hyunok Oh and Soonhoi Ha. 'Fractional Rate Dataflow Model for Efficient Code Synthesis'. In: *VLSI Signal Processing-Systems for Signal, Image, and Video Technology* 37.1 (May 2004), pp. 41–51. DOI: 10.1023/b:vlsi.0000017002.91721.0e

[Olo+11] Andreas Olofsson, Roman Trogan, Oleg Raikhman and Lexington Adapteva. 'A 1024-core 70 GFLOP/W floating point manycore microprocessor'. In proc.: *High Performance Embedded Computing (HPEC)*, 2011. *http://www.adapteva.com/wp-content/uploads/2011/10/adapteva_hpec11.pdf*

[ORCAR] *Open RVC-CAL Application Repository. https://github.com/orcc/orc-apps*

[Par95] Thomas M. Parks. 'Bounded scheduling of process networks'. PhD thesis. University of California, Berkeley, 1995. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.144.7822*

[PCH02] Chanik Park, Jaewoong Chung and Soonhoi Ha. 'Extended synchronous dataflow for efficient DSP system prototyping'. In proc.: *International Workshop on Rapid System Prototyping. Shortening the Path from Specification to Prototype*, 2002, IEEE Comput. Soc, pp. 295–322. DOI: 10.1109/iwrsp.1999.779053

[Per+16] Quentin Perret, Pascal Maurere, Eric Noulard et al. 'Temporal Isolation of Hard Real-Time Applications on Many-Core Processors'. In proc.: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2016, IEEE, pp. 1–11. DOI: 10.1109/rtas.2016.7461363

[PGM16] Xing Pan, Yasaswini Jyothi Gownivaripalli and Frank Mueller. 'TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring'. In proc.: *Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, IEEE, pp. 363–372. DOI: 10.1109/ipdps.2016.26

[PSB09] William Plishker, Nimish Sane and Shuvra Bhattacharyya. 'A generalized scheduling approach for dynamic dataflow applications'. In proc.: *Design, Automation & Test in Europe (DATE)*, Apr. 2009, IEEE, pp. 111–116. DOI: 10.1109/date.2009.5090642

[Rab91] Stanley Rabinowitz. 'A model for interference in multiprocessors'. In: *Missouri Journal of Mathematical Sciences* 3 (1991), pp. 12–19. *http://stanleyrabinowitz.com/bibliography/cacheModel.pdf*

[Rau79] B. Ramakrishna Rau. 'Interleaved Memory Bandwidth in a Model of a Multiprocessor Computer System'. In: *IEEE Transactions on Computers* 28.9 (Sept. 1979), pp. 678–681. DOI: 10.1109/tc.1979.1675436

[Rau91] B. Ramakrishna Rau. 'Pseudo-randomly interleaved memory'. In proc.: *Computer architecture (ISCA)*, 1991, ACM, pp. 74–83. DOI: 10.1145/115952.115961

[Rei+11] Jan Reineke, Isaac Liu, Hiren D. Patel et al. 'PRET DRAM controller: bank privatization for predictability and temporal isolation'. In proc.: *Hardware/software codesign and system synthesis (CODES+ISSS)*, 2011, ACM, pp. 99–108. DOI: 10.1145/2039370.2039388

[Rih+16] Hamza Rihani, Matthieu Moy, Claire Maiza et al. 'Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor'. In proc.: *Real-Time Networks and Systems (RTNS)*, 2016, ACM. DOI: 10.1145/2997465.2997472

[RSF11] Benjamin Ransford, Jacob Sorber and Kevin Fu. 'Mementos. system support for long-running computation on RFID-scale devices'. In: *ACM SIGARCH Computer Architecture News* 39.1 (Mar. 2011), p. 159. DOI: 10.1145/1961295.1950386

[SA69] Charles E. Skinner and Jonathan R. Asher. 'Effects of storage contention on system performance'. In: *IBM Systems Journal* 8.4 (1969), pp. 319–333. DOI: 10.1147/sj.84.0319

[Sat+03] Kiran M. N. V. Satya, Jayram M. Nageswaran, Pradeep Rao and Soumitra K. Nandy. 'A complexity effective communication model for behavioral modeling of signal processing applications'. In proc.: *Design automation conference (DAC)*, 2003, ACM, pp. 412–415. DOI: 10.1145/775832.775939

[SBS] *StreamIt Benchmark Suite. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml*

[Sch+12] Lars Schor, Iuliana Bacivarov, Devendra Rai et al. 'Scenario-based design flow for mapping streaming applications onto on-chip many-core systems'. In proc.: *Compilers, architectures and synthesis for embedded systems (CASES)*, 2012, ACM, pp. 71–80. DOI: 10.1145/2380403.2380422

[Sch94] Sven-Bodo Scholz. 'Single-assignment C — Functional Programming Using Imperative Style'. In proc.: *Implementation of Functional Languages (IFL)*, 1994, University of East Anglia, Norwich, England, UK, pp. 21-1 –21-13. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.378*

[SCL96] Mazen A. R. Saghir, Paul Chow and Corinna G. Lee. 'Exploiting dual data-memory banks in digital signal processors'. In: *ACM SIGOPS Operating Systems Review* 30.5 (Dec. 1996), pp. 234–243. DOI: 10.1145/248208.237193

[SF12] Kazuki Sakamoto and Tomohiko Furumoto. 'Grand central dispatch'. In: *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 2012, pp. 139–145. DOI: 10.1007/978-1-4302-4117-1_6

[SG07] K. Shyam and R. Govindarajan. 'An Array Allocation Scheme for Energy Reduction in Partitioned Memory Architectures'. In proc.: *Compiler Construction (CC)*, 2007, Springer, pp. 32–47. DOI: 10.1007/978-3-540-71229-9_3

[Sip03] Viera Sipkova. 'Efficient Variable Allocation to Dual Memory Banks of DSPs'. In proc.: *Software and Compilers for Embedded Systems (SCOPES)*, 2003, Springer, pp. 359–372. DOI: 10.1007/978-3-540-39920-9_25

[SLA12] Anastasia Stulova, Rainer Leupers and Gerd Ascheid. 'Throughput driven transformations of Synchronous Data Flows for mapping to heterogeneous MPSoCs'. In proc.: *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2012, IEEE, pp. 144–151. DOI: 10.1109/samos.2012.6404168

[Sod+16] Avinash Sodani, Roger Gramunt, Jesus Corbal et al. 'Knights Landing: Second-Generation Intel Xeon Phi Product'. In: *IEEE Micro* 36.2 (Mar. 2016), pp. 34–46. DOI: 10.1109/mm.2016.25

[Sot+13] Maria Soto, Marc Sevaux, André Rossi and Johann Laurent. *Memory Allocation Problems in Embedded Systems: Optimization Methods*. Wiley-ISTE, Jan. 2013, p. 256. *https://hal.archives-ouvertes.fr/hal-00767031*

[Str70] William D. Strecker. 'An analysis of the instruction execution rate in certain computer structures'. PhD thesis. DTIC Document, 1970. *http://www.dtic.mil/docs/citations/AD0711408*

[Thi+07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid and Kai Huang. 'Mapping Applications to Tiled Multiprocessor Embedded Systems'. In proc.: *Application of Concurrency to System Design (ACSD)*, July 2007, IEEE, pp. 29–40. DOI: 10.1109/acsd.2007.53

[VNS07] Sven Verdoolaege, Hristo Nikolov and Todor Stefanov. 'pn: A Tool for Improved Derivation of Process Networks'. In: *EURASIP Journal on Embedded Systems* (Jan. 2007), pp. 1–13. DOI: 10.1155/2007/75947

[VY15] Prathap Kumar Valsan and Heechul Yun. 'MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems'. In proc.: *Cyber-Physical Systems, Networks, and Applications*, Aug. 2015, IEEE, pp. 86–93. DOI: 10.1109/cpsna.2015.24

[Win87] Glynn Winskel. 'Event structures'. In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Vol. 255. Lecture Notes in Comp. Science. Springer, 1987, pp. 325–392. DOI: 10.1007/3-540-17906-2_31

[WKP13] Zheng Pei Wu, Yogen Krish and Rodolfo Pellizzoni. 'Worst Case Analysis of DRAM Latency in Multi-requestor Systems'. In proc.: *IEEE Real-Time Systems Symposium*, Dec. 2013, IEEE, pp. 372–383. DOI: 10.1109/rtss.2013.44

[WR12] Matthieu Wipliez and Mickaël Raulet. 'Classification of Dataflow Actors with Satisfiability and Abstract Interpretation'. In: *Embedded and Real-Time Communication Systems* 3.1 (Jan. 2012), pp. 49–69. DOI: 10.4018/jertcs.2012010103

[Yun+14] Heechul Yun, Renato Mancuso, Zheng-Pei Wu and Rodolfo Pellizzoni. 'PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms'. In proc.: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2014, IEEE, pp. 155–166. DOI: 10.1109/rtas.2014.6925999

[Yvi+13] Hervé Yviquel, Antoine Lorence, Khaled Jerbi et al. 'Orcc: Multimedia Development Made Easy'. In proc.: *Conference on Multimedia (MM)*, 2013, ACM, pp. 863–866. DOI: 10.1145/2502081.2502231

[Zeb+08] Christian Zebelein, Joachim Falk, Christian Haubelt and Jurgen Teich. 'Classification of General Data Flow Actors into Known Models of Computation'. In proc.: *Formal Methods and Models for Co-Design (MEMOCODE)*, June 2008, IEEE, pp. 119–128. DOI: 10.1109/memcod.2008.4547699

[Zha+11] Lei Zhang, Meikang Qiu, Edwin H.-M. Sha and Qingfeng Zhuge. 'Variable assignment and instruction scheduling for processor with multi-module memory'. In: *Microprocessors and Microsystems* 35.3 (May 2011), pp. 308–317. DOI: 10.1016/j.micpro.2010.12.002

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

A. Tretter, J. Boutellier, J. Guthrie, L. Schor and L. Thiele. 'Executing Dataflow Actors as Kahn Processes'. In proc.: *Embedded Software (EMSOFT)*, 2015. (Section 2.2)

A. Tretter, H. Pandit, P. Kumar and L. Thiele. 'Deterministic Memory Sharing in Kahn Process Networks: Ultrasound Imaging as a Case Study'. In proc.: *Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2014. (Section 2.3)

A. Tretter, P. Kumar and L. Thiele. 'Interleaved Multi-Bank Scratchpad Memories: A Probabilistic Description of Access Conflicts'. In proc.: *Design Automation Conference (DAC)*, 2015. (Chapter 4)

A. Tretter, G. Giannopoulou, M. Baer and L. Thiele. 'Minimising Access Conflicts on Shared Multi-Bank Memory'. In: *ACM Transactions on Embedded Computing Systems (TECS)* - Special Issue ESWEEK 2017. (Section 5.1)