Synthesis of Synchronous Programs to Parallel Software Architectures

vom Fachbereich Informatik der
Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von

Daniel Baudisch

Datum der wissenschaftlichen Aussprache:	8. November 2013
Vorsitzender der Promotionskommission:	Prof. Dr. Arnd Poetzsch-Heffter
Berichterstatter:	Prof. Dr. Klaus Schneider Prof. Dr. Peter Marwedel
Dekan:	Prof. Dr. Arnd Poetzsch-Heffter

D 386

Danksagung

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mir bei der Erstellung dieser Arbeit geholfen haben.

Dazu zählt in erster Linie mein Betreuer Prof. Dr. Klaus Schneider, der mir die Möglichkeit gab, diese Arbeit zu erstellen. Seine stetige Unterstützung, konstruktiven Kritiken und Anregungen, sowie die hilfreichen Diskussionen waren für mich unabdingbar und hilfreich während der Erstellung dieser Arbeit. Durch ihn konnte ich an nationalen und internationalen Konferenzen und Workshops teilnehmen, um dort mit anderen Forschern auf dem Gebiet des parallelen Rechnens über Ideen zu diskutieren und Gedanken auszutauschen. Des Weiteren möchte ich mich bei Prof. Dr. Peter Marwedel für die Zweitkorrektur bedanken. Ich möchte mich ebenfalls bei der DFG bedanken, die mir durch die finanzielle Unterstützung die Verwirklichung dieser Arbeit ermöglicht hat.

Ebenso möchte ich mich bei allen Kollegen bedanken, die mir während meiner Zeit an der Universität zu Seite standen: Tobias Schüle, der mir noch während meines Studiums die ersten Einsichten in das wissenschaftliche Arbeiten gegeben hat. Jens Brandt, der jederzeit mit Rat und Tat zur Seite stand und mir mit seinem Wissen einen leichten Einstieg in die Promotion ermöglicht hat. Bai Yu möchte ich für das Korrekturlesen und für seine Tipps zur Verbesserung der Verständlichkeit dieser Arbeit danken. Selbstverständlich bedanke ich mich auch bei allen anderen Kolleginnen und Kollegen für die tolle Zusammenarbeit und die sehr angenehme Arbeitsatmosphäre.

Zuletzt gilt ein besonderer Dank meinen Eltern, meiner Schwester und meiner Freundin Jacqueline, die mir immer zur Seite standen. Deren Unterstützung hat es mir erst möglich gemacht, die Zeit und Kraft aufzubringen, die für diese Arbeit notwendig waren.

Zusammenfassung

Diese Arbeit behandelt die vollautomatische Übersetzung von synchronen Programmen in parallele Software für verschiedene Architekturen, genauer gesagt für Systeme mit gemeinsam genutztem Speicher und Systeme mit verteiltem Speicher. Dabei nutzen wir Eigenschaften des synchronen Berechnungsmodells (englisch: model of computation (MoC)) zur Reduzierung von Kommunikation und Out-of-Order Ausführung und Datenspekulation zur Erhöhung von Parallelität und zum dynamischen Ausgleich der Rechenauslast.

Die manuelle Programmierung paralleler Systeme erfordert von Entwicklern die Partitionierung eines Systems in Teilprozesse und das Hinzufügen von Synchronisation und Kommunikation. Der modellbasierte Ansatz der Entwicklung abstrahiert von Details der Zielarchitektur und erlaubt späte Entscheidungen über die Zielarchitektur. Das synchrone MoC unterstützt diesen Ansatz indem es von der Zeit abstrahiert und implizite Parallelität und Synchronisation erlaubt. Existierende Kompilierungsverfahren übersetzen synchrone Programme in synchrone bedingte Aktionen (englisch: synchronous guarded actions (SGAs)) – ein Zwischenformat dass von semantischen Problemen der synchronen Sprachen abstrahiert. Compiler für SGAs analysieren Kausalitätsprobleme und stellen logische Korrektheit und die Abwesenheit von Schizophrenieproblemen sicher. SGAs bieten daher einen einfachen und allgemeinen Startpunkt und behalten das synchrone MoC bei. Das unmittelbare Feedback im synchronen MoC macht die Übersetzung dieser Systeme in parallele Software nicht-trivial. Andere MoCs wie die Datenfluss-Netzwerke (englisch: data-flow processing networks (DPNs)) passen dagegen direkt mit parallelen Architekturen zusammen. Wir übersetzen die SGAs in DPNs, welche ein geläufiges Modell zur Erstellung paralleler Software darstellen. Die rein datengesteuerte Ausführung von DPNs benötigt keine globale Koordinierung, so dass DPNs einfach in parallele Software für Architekturen mit verteiltem Speicher übersetzt werden können. Die Erzeugung von parallelem Code aus DPNs fordert Compiler mit zwei Problemen heraus: Um ein paralleles System perfekt auszunutzen muss die Kommunikation und Synchronisation gering und die Auslastung der Recheneinheiten gleichmäßig sein. Die Vielfalt an Hardwarearchitekturen und dynamische Ausführungstechniken in deren Recheneinheiten macht eine statisch ausbalancierte verteilte Ausführung unmöglich.

Das synchrone MoC spiegelt sich in denen von uns generierten DPNs wieder und erlaubt aufgrund bestimmter Eigenschaften Optimierungen bezüglich der zuvor genannten Probleme. Wir wenden eine allgemeine Kommunikationreduzierung und Out-of-Order Ausführung aus dem Hardware Design an, um die Rechenauslast dynamisch auszugleichen.

Abstract

This thesis provides a fully automatic translation from synchronous programs to parallel software for different architectures, in particular, shared memory processing (SMP) and distributed memory systems. Thereby, we exploit characteristics of the synchronous model of computation (MoC) to reduce communication and to improve available parallelism and load-balancing by out-of-order (OOO) execution and data speculation.

Manual programming of parallel software requires the developers to partition a system into tasks and to add synchronization and communication. The model-based approach of development abstracts from details of the target architecture and allows to make decisions about the target architecture as late as possible. The synchronous MoC supports this approach by abstracting from time and providing implicit parallelism and synchronization. Existing compilation techniques translate synchronous programs into synchronous guarded actions (SGAs) which are an intermediate format abstracting from semantic problems in synchronous languages. Compilers for SGAs analyze causality problems, ensure logical correctness and the absence of schizophrenia problems. Hence, SGAs are a simplified and general starting point and keep the synchronous MoC at the same time. The instantaneous feedback in the synchronous MoC makes the mapping of these systems to parallel software a non-trivial task. In contrast, other MoCs such as data-flow processing networks (DPNs) directly match with parallel architectures. We translate the SGAs into DPNs, which represent a commonly used model to create parallel software. DPNs have been proposed as a programming model for distributed parallel systems that have communication paths with unpredictable latencies. The purely data-driven execution of DPNs does not require a global coordination and therefore DPNs can be easily mapped to parallel software for architectures with distributed memory. The generation of efficient parallel code from DPNs challenges compiler design with two issues: To perfectly utilize a parallel system, the communication and synchronization has to be kept low, and the utilization of the computational units has to be balanced. The variety of hardware architectures and dynamic execution techniques in processing units of these systems make a statically balanced distributed execution impossible.

The synchronous MoC is still reflected in our generated DPNs, which exhibits characteristics that allow optimizations concerning the previously mentioned issues. In particular, we apply a general communication reduction and OOO execution to achieve a dynamically balanced execution which is inspired from hardware design.

Contents

Acronyms

ADG	action dependency graph.
ALU	arithmetic-logical unit.
API	application programming interface.
CAS	compare-and-swap.
CBS	central buffer station.
DFG	data-flow graph.
DPN	data-flow processing network.
DSP	digital signal processing.
EPIC	explicitly parallel instruction computing.
FIFO	first-in-first-out.
FPGA	field programmable gate array.
ILP	instruction level parallelism.
MIMD	multiple-instruction-multiple-data.
MoC	model of computation.
MPI	message passing interface.
OOO	out-of-order.
RAW	read after write.
SDF	static data-flow.
SDFG	static data-flow graph.
SGA	synchronous guarded action.
SMP	shared memory processing.
SPMD	single program multiple data.
TLP	task level parallelism.
VLIW	very large instruction word.
WAR	write after read.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computer systems are more and more ubiquitous in our everyday lives. A couple of years ago, computation power of single cores has been focused by many manufacturers. Increasing the performance by steadily rising clock frequencies has been the major sales argument for years.

In addition, smaller manufacturing technology leads to a larger logical chip area that allows to implement more complex circuits. This has been mostly used to increase the ability of a processor to compute in parallel. For instance, so-called dynamic scheduling processors are capable of analyzing instruction streams of existing programs on the fly and to execute independent instructions in parallel. However, the parallelism provided by sequential programs is limited and often not capable to utilize the potential computational power of modern processors. Hence, the developers of processors had to provide the computational power of a single chip in a different way, namely by multi-core processors.

The trend of developing multi-core processors was also motivated by the energy consumption and an increased usage of chips in mobile devices. In particular, energy consumption became an important research topic. Using multi-core processors is basically reasoned as follows: while doubling the clock frequency of a processor leads to a quadratically increased power consumption, doubling the number of cores only doubles it. As a result, the N-core processor provides the same theoretical computational power as a single-core processor running at a N-times higher clock frequency, but consumes less power at the same time.

Techniques like pipelining and dynamic scheduling have been successfully implemented in many processors to increase instruction level parallelism (ILP). This can be managed by pure hardware logic. For instance, ILP can be recognized by considering a window of instructions as done in processors with dynamic scheduling or statically by compiler techniques, e. g., loop unrolling.

In contrast, multi-core machines require additional effort in programming. Their processing elements communicate through shared memory or explicit message passing. In particular, synchronization is done using semaphores, locks and network communication,

i. e., methods written in software that are usually provided by the operating system. This makes synchronization between different threads quite expensive and requires them to be more computationally expensive to compensate the communication overhead, and finally, to get a speed-up. This can be approached in two different ways: (1) One can try to increase the computational effort of threads or (2) to decrease the communication effort. Since the applications are given and cannot be simply "scaled" by our synthesis tool, we aim at the second goal in this thesis.

Load-balancing is an issue that has to be tackled in parallelization of applications, too. This expression refers to an evenly distribution of computational effort to processing units. Partitioning of a general application to parts with balanced computational effort is not a trivial task. This requires precise knowledge of the targeted hardware architecture to compute the computational effort for each partition. In addition, conditional instructions in a partition impede the precise computation of computational effort and results in a varying amount of load. This issue is hardened by the varying multi-core architectures, which can be especially found in the area of mobile devices. Variations of hardware require different functions to calculate the computational effort for partitions. This can be caused by varying number of cores, differences in available ILP of each core and internal architecture details.

Beside the previously mentioned issues of parallel programming, another key feature in multi-threaded and distributed programming is the high degree of non-determinism [110]. To this end, creating code for different target architectures requires to tackle different issues.

The goal of model-based development of applications is to abstract of details about the final architecture and to make decisions about the targeted hardware as late as possible. This is especially of interest when created software should be available for different architectures or when the targeted hardware platform is likely to change during the development process. Nowadays, the model-based development becomes more important due to the manifold hardware architecture and short development cycles of hardware, e. g., in the mobile market.

The system description languages Esterel [28], Lustre [50,84] and Quartz [156] are based on the synchronous MoC. It has many advantages for the model-based design of reactive embedded systems. It has effective use in formal verification, deterministic simulation, and synthesis with correctness-by-construction guarantees for single-threaded software and synchronous digital hardware circuits [24, 37, 83, 85, 106]. In the context of parallel programming, the logical time, the implicit parallelism and implicit synchronization in the synchronous MoC allows programmers to focus on the functional correctness of their algorithms instead of dealing with architecture-specific timing/synchronization problems. In contrast, programming parallel software in imperative languages, e. g., C or Java, often forces programmers to take care of explicit communication and synchronization of parallel processes, especially in distributed systems.

The other side of the coin is that the synchronous MoC makes the compilation quite difficult. The instantaneous feedback between instructions due to the abstracted time can therefore lead to so-called causality problems [27, 158, 159, 162] and schizophrenia problems [157]. These problems are well studied in the context of synchronous systems, and

many analysis procedures have been developed to locate and eliminate these problems. For instance, compilers check the causality of a program at compile time with a fixpoint analysis that essentially corresponds to those used for checking the speed-independence of asynchronous circuits via ternary simulation [42].

SGAs are an intermediate format for various synchronous languages [39, 156, 157]. In particular, they are the result of an already analyzed synchronous program, which ensures that the synchronous system described by the resulting set of guarded actions is free of *write conflicts*, i. e., several actions assign different values to the same variable in the same macro step, and previously mentioned causality and schizophrenia problems. This makes SGAs convenient for further processing of a synchronous system, i. e., for the creation of compiler back-ends.

The mismatch between the synchronous MoC and most real-world implementation environments poses serious problems, especially for distributed and parallel architectures. The synthesis procedures usually have to map a synchronous program to a target architecture that does not provide perfect synchrony. While creation of sequential code from synchronous systems has been solved [67], the mapping of synchronous systems to parallel architectures still challenges researchers.

DPNs [43, 97, 109, 115, 138] are an untimed MoC. They consist of a finite number of processes which run in parallel without global coordination. Instead, the individual processes perform their computations independently of other processes and start as soon as the data values required for a computation step are available. In order to exchange data, the processes are connected by a set of fixed first-in-first-out (FIFO) buffers. Each FIFO buffer has a unique source and a unique sink process that either writes values to or reads values from the FIFO buffer.

Since the MoC of DPNs is very general, it can be implemented in many ways. Its core has served as a basis for computer architectures [9,61,63,108,153], as well as for the design of programming languages and libraries [30, 91, 95, 128, 135, 152]. It does not demand the use of special programming languages, and instead, allows one to use traditional sequential programming languages for the implementation of the process nodes. The main functionality that the libraries have to provide is the ability that several nodes can run in parallel with a communication over FIFO buffers.

However, a drawback of this generality is the impossibility to guarantee certain properties of the network. For example, *determinism*, i. e., whether the same inputs lead to the same outputs (independent of transmission delays) is a desired property. Similarly, *boundedness* of buffers, i. e., whether a DPN can be run with buffers of finite size, is also a very crucial property: while the size of FIFO buffers is unlimited in the general DPN model, it has to be finite for any practical implementation. One way to guarantee both properties is to impose certain restrictions so that the considered subset of DPNs are deterministic and have decidable boundedness or liveness problems. Static data-flow (SDF) networks [29, 31, 43, 109, 113–115] and cyclo-static data-flow networks [32, 70] are such restricted nets, which have become very successful, in particular, for the synthesis of signal processing systems [6, 31, 75, 76, 110, 111, 133, 146]. They are a special kind of DPNs where

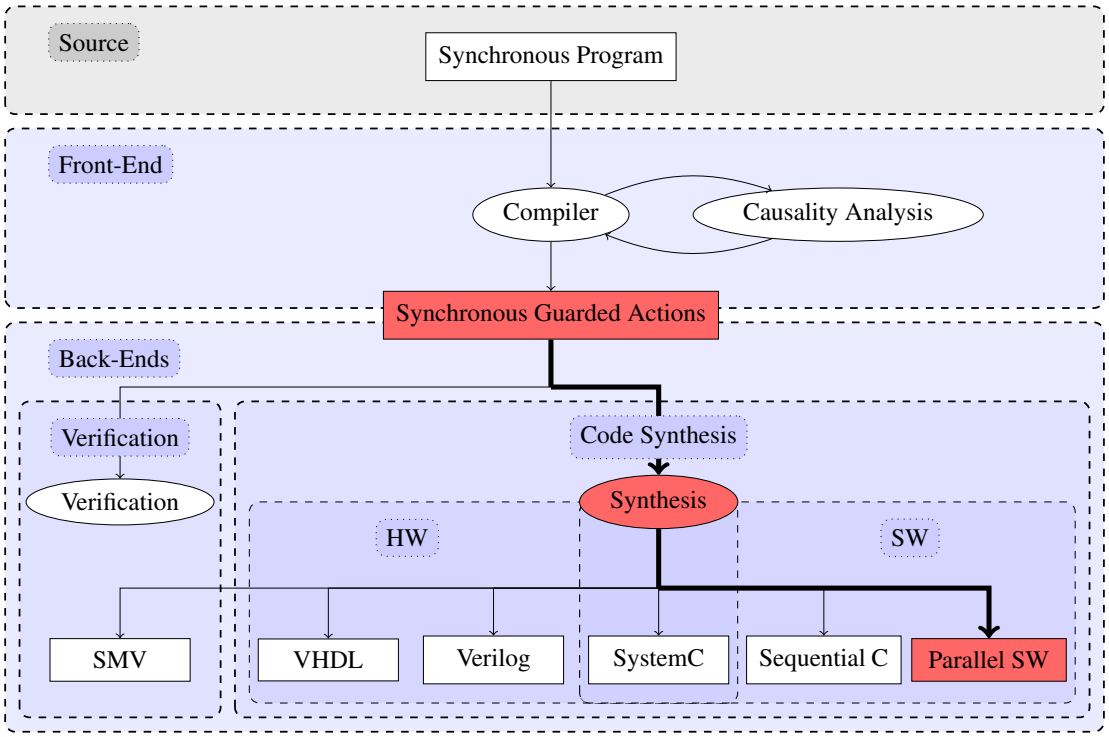


Figure 1.1: Contribution of this thesis are different approaches to create parallel software from synchronous programs: Integration of the synthesis tools presented in this thesis into the flow of the Averest system.

in each firing step, the process node always consumes the same number of data values from the input streams and produces the same number of data values for the output streams.

As DPNs are inherently parallel and completely abstract from timing and clocks, they can be straightforwardly mapped to parallel and distributed platforms. While the general data-flow driven MoC allows non-deterministic behavior, it can be restricted to so-called synchronous DPNs to only allow deterministic behavior [114]. Thus, it can be used to create deterministic parallel applications.

1.2 Contribution and Organization of the Thesis

This thesis presents a set of methods to create parallel software from SGAs, which is a generic format to represent synchronous systems. Synchronous systems already provide implicit and explicit parallelism and abstract from communication latencies, which makes them a convenient programming model to create parallel software. The compilation of synchronous programs to SGAs solves semantic issues of the synchronous MoC. Hence, we assume to start from a synchronous system that is logically correct and free of schizophrenia and causality problems. A translation from SGAs to DPNs as a further intermediate format provides a representation that already yields a desynchronized parallel system. The main

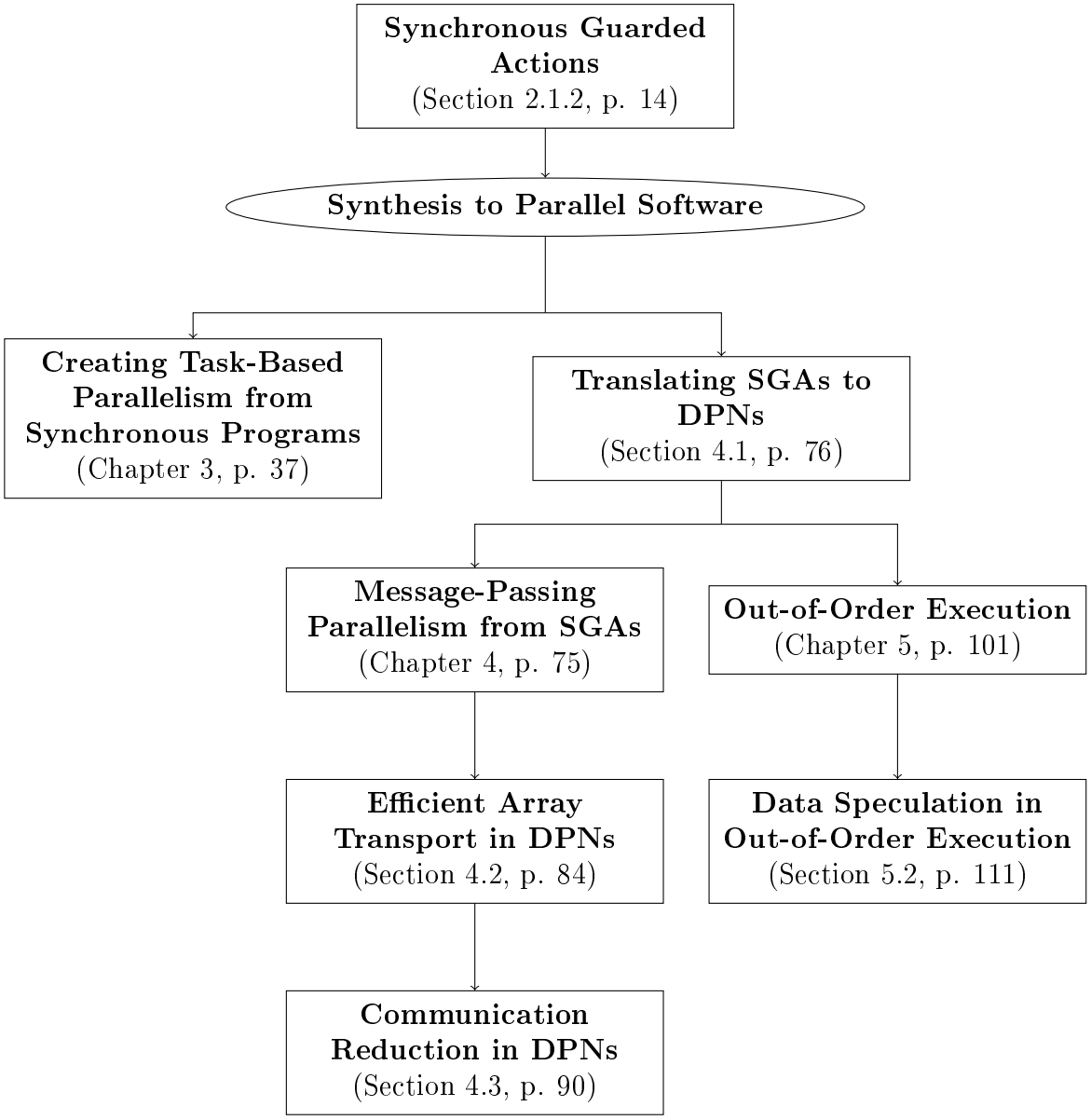


Figure 1.2: Contribution of this thesis are different approaches to create parallel software from synchronous programs: Overview and organization of the approaches presented in this thesis.

idea of providing different methods to create parallel software is to target the manifold parallel hardware architectures. We aim at creating parallel software for systems with distributed memory, thereby focusing on the reduction of communication costs, and SMP systems, thereby focusing on load-balancing using OOO execution, which is inspired by a technique from processor design.

All approaches have been implemented into the Averest[1] framework, whose work flow is depicted in Figure 1.1. Red nodes represent the flow which is considered in this thesis. Figure 1.2 gives an overview of the different synthesis approaches. In addition, it gives the reader an overview of the organization of this thesis and a guide to quickly find the chapter or section and the page for each approach.

Not handled or not in focus of this thesis are the following topics: Solving semantic issues in synchronous programs is not considered. As already mentioned, we start from SGAs, which are assumed to be free of causality and schizophrenia problems. Partitioning is a topic, which is often associated with mapping large sets of instructions to a smaller number of computational units. Partitioning is a complete research area on its own and entering on this subject exceeds the scope of this thesis.

The remainder of this thesis is organized as follows. Chapter 2 explains the preliminary work for this thesis. This includes the source format for the presented synthesis approaches, i. e., SGAs (see Section 2.1.2) and their underlying model of computation in general (see Section 2.1). In addition, DPNs (see Section 2.2) are used as an intermediate format in the synthesis process to provide a more convenient starting point for the message passing approach (see Chapter 4). The main issue in message passing is to reduce the communication, which is handled in the array communication reduction and the general communication reduction in Section 4.2 and 4.3, respectively. Chapter 5 presents OOO execution of DPNs, which was inspired by OOO execution used in processors with dynamic scheduling. As a result, an approach to improve the parallelism is to use data speculation, which is handled in Section 5.2. The evaluation of the different approaches using a couple of benchmarks is covered in Chapter 7. Finally, Chapter 8 concludes this thesis.

1.3 Related Work

Parallel Code Generation The creation of software for parallel architectures is eased by various application programming interfaces (APIs), e. g., OpenMP [136], StarSS [148, 169], Intel TBB [90], Cilk [74] and MPI [41]. These APIs have one in common: they abstract from operating system details, e. g., creation of threads and methods for thread synchronization. However, they do not parallelize fully automatic, i. e., programmers have to manually identify and explicitly program parallelism.

Several tools have been developed to automatically parallelize sequential code, e. g., par4all [7], S2P [5] and cetus [116]. Most of these approaches are source-to-source compilers that translate C or Fortran to parallel code using an existing API, e. g., OpenMP. Typical approaches to parallelize sequential code try to find data parallelism or to unroll loops, which requires an extensive data dependency analysis. Since we start our code creation from synchronous DPNs, we can apply specific optimizations, particularly the OOO execution. Moreover, dynamic analysis of dependencies as done in Section 4.3 has not been considered in the mentioned tools.

[1] http://www.averest.org

Another way to create multi-threaded code is approached by StreamIt [170]: This language aims at providing a better description language for digital signal processing (DSP) applications. However, we do not want to restrict our applications to streaming.

Our actual starting point are synchronous programs. Edwards presents in [67] several techniques that are used to compile synchronous languages into sequential code, e. g., by generating a sequential schedule from a dependency graph [183].

Automatic distribution of synchronous programs to asynchronous networks of processing elements has been considered in [49, 78, 79]. A clock-driven distribution of Lustre programs is presented which partitions and distributes the system according to the clock that triggers each part. While this approach has shown to produce quite efficient implementations, it may suffer from a significant drawback: Mutual data dependencies between components may require that some component must be further decomposed into smaller components, which may require in turn additional communication and synchronization effort. In our approach, this is avoided by construction. DSystemJ [123] allows one to create parallel code from manually desynchronized systems with explicit communication statements. This surely implies additional effort in programming, which is done automatically in our synthesis tools.

An approach of running synchronous code directly on a processor is given by Li et al. [117, 118]. They developed a processor that is capable of executing Esterel code. The core of the processor is called the KEP3a Reactive Multi-threaded Core and is scalable, i. e., depending on the used field programmable gate array (FPGA) board, multiple cores can be connected to increase the speed of the execution of Esterel code. This requires specific hardware, which is not the case for the approaches presented in this thesis.

The necessity for desynchronization of synchronous programs originally comes from hardware design. Handling of clock skews and clock distribution became a major problem in large circuits. Solutions have been presented as *latency insensitive design* in [46–48] and as *synchronous elastic design* [59, 60, 102]. These approaches lead to so-called *globally asynchronous locally synchronous* (GALS) systems [168], which are considered as a special variant of Kahn DPNs [97].

For DPNs, the scheduling of nodes can be organized in different ways, which may be roughly categorized into static (at compile-time) and dynamic (at run-time) variants [112]. While static variants [113] may be appropriate for applications that run on real-time DSP multi-core processors where the final architecture and all running processes are known, the dynamic variant is the best choice for most other applications for the following reasons: First, as the schedule depends on the architecture, the dynamic variant is necessary for transferring compiled code from one machine to another one. Second, even with a fixed architecture, it is very hard to estimate the run-time of tasks in the context of other running processes and cache effects.

Multi-threaded execution requires task level parallelism (TLP), i. e., coarse-grained parallelism. When starting from a system description with fine-grained parallelism, TLP can be achieved by application of a partitioning algorithm, e. g., Metis [99], Scotch [53], JOSTLE [178]. As already mentioned, partitioning is not in the focus of this work, because this is a research topic on its own. Although our DPNs may profit from a good partitioning

algorithm, such an algorithm does not solve all issues. This includes the reduction of array communication as described in Section 4.2 and the general communication reduction (see Section 4.3). Load-balancing is another concern of partitioning algorithms, which can be improved using our dynamic scheduling approach, namely the OOO execution.

Communication Reduction The reduction of communication costs in DPNs has been already addressed by many papers: For example, [82, 182] mainly consider the elimination of redundant messages. Similar to our approach, they are working on data-flow graphs (DFGs), but their source is a structured program consisting of loops and if-statements. Their optimization is based on the introduction of so-called 'Section Communication Descriptors' and 'Availability Section Descriptors' which are obtained from the analysis of loops in the source program. Restricted to the consideration of loops, their approach does not target the communication optimization of general DPNs. In addition, Tims et. al. [173] consider reduction of communication in clusters using the distribution interrelationship flowgraph (DIF), which is an extension of the control-flow graph of the source program. Their approach fully concentrates on the dynamic distribution of arrays in a structured single program multiple data (SPMD) program consisting of a sequence of several loops working on arrays that are parallelized to nodes in a cluster. In contrast, our approach is based on generic DPNs where the nodes contain different code and are not the result of loop parallelization.

A demand-driven optimization has been proposed in [140, 141] by Arvind. The basic idea is to generate and send data only, when it is requested from an output (similar to lazy evaluation in functional programming languages). Clearly, this approach does not try to reduce communication costs, but rather to avoid unnecessary computations at all, while keeping the communications as in the original DPN.

The problem to deal with compound data types in DPNs was heavily discussed in the construction of data-flow computers like the Monsoon computer and their related programming languages like ID or VAL [9,10,13,175]. To this end, Arvind presents in [9,10] special memories that were called *I*-structures to ensure the single-assignment rule to each variable. These memory cells were endowed with a special protocol to avoid reads before writes and to avoid overwriting a once generated value. *I*-structures were refined to *M*-structures in [13] to allow re-use of memory cells, which was not possible with *I*-structures.

The problem also exists in functional programming languages whose virtual machines also suffer from unnecessary copy operations. Variables cannot be modified directly, and therefore, the value of a variable can only be modified during a copy operation. For example, adding a single element to a list causes duplication of all elements in the source list. For this reason, some functional programming languages like F# introduced mutable data types which is – in the context of several threads – nothing else than shared memory. Hence, it has to be dealt with care due to potential data races.

Software creation from DPN descriptions, particularly CAL [69], has already been considered in [179]. The language CAL supports the usage of arrays only as local/state variables of nodes. Interaction between nodes by sending arrays is not supported. Hence,

the communication of arrays and its optimization must be manually implemented without any compiler support.

Communication reduction is also considered by endo-isochronous systems [143–145, 156]. In addition to general desynchronized synchronous systems, a variable is allowed to be absent in a macro step, i. e., it has no value in this case. Endochrony characterizes a node in a desynchronized program by its ability to derive the presence and absence of values from the communicated values. Two endochronous nodes are called isochronous if the presence of variables agree in these nodes, i. e., in each reaction a variable is either present or absent in both nodes. [38] proposes a method to desynchronize synchronous programs by modes, which requires a global mode manager. The global mode manager has to distribute the modes at least to a part of the nodes. Since this method has not been evaluated, it is not clear, if the global mode manager may become a bottleneck. Our approach manages communication reduction purely on existing communication without adding a centralized manager. Despite of the rich theory in this area, no implementation is yet known to us.

The closest approach to our communication optimization as done in Section 4.3 is considered by [49] as *"the when needed strategy"*. This is only roughly explained and assumes that a producer is always capable of being able to evaluate a *when needed* condition. We relax this condition and consider also a communication reduction which can be analogously described as *"the when changed strategy"*.

[172] introduces *early evaluation* where an evaluation is started as soon as sufficiently many data values are available. A practical example for early evaluation is token counterflow and derivatives [57, 58, 165]. One issue in early evaluation is to assign actual tokens to the correct calculation. Counterflow tokens are used to eliminate tokens that have been identified to be useless for calculations. However, this technique does not consider to reduce the communication overhead in a system.

OOO Execution The hardware architecture of a data-flow computer proposed in [62] is inspired from general data-flow computers, and it basically consists of four parts: the memory, an arbitration network, a distribution network, and the operation units. The memory only stores so-called instruction cells which describe the nodes of the SDFG and keep track of the presence of validity of tokens. With the help of this information, the hardware can mark instruction cells as soon as all their inputs are valid. Then, the arbitration network which connects memory and operation units handles the actual dispatching to resources. With the help of the distribution network, the results are transfered back to the corresponding instruction cells so that the next computations are triggered.

Task Superscalar [71] is also a hardware-based solution to execute DFGs out-of-order. The programming model corresponds to the one of SmpSS [148]. The graph and the dependencies must be explicitly given, i. e., partitioning and dependency analysis is left to the programmer. Beside the specific hardware extension for their OOO task pipeline, their implementation clearly differs to ours: The scheduling of new tasks is dedicated to a single thread, while our implementation has a distributed scheduling mechanism, i. e., each

processing unit is allowed to schedule new tasks. Finally, speculation as done in Section 5.2 has not been considered in their approach.

OoOJava [92, 93] provides a compiler back-end to create parallel code from Java programs. The proposed compiler mainly focuses on the analysis of data dependencies between explicitly given partitions of a source program. Programs created by OoOJava dynamically create trees of reorderable blocks, which can be executed in data-flow order. This limits its application to problems with limited size. In contrast, our OOO execution of DPNs allows the processing of (infinite) streams.

There is a lot of related work about speculative execution: [12, 185] present approaches to parallelize sequential programs using a master/slave design. Both approaches use speculation as a primary component to create parallelism. In our case, we already have a parallel program running and want to use speculation to exploit more parallelism. Therefore, our speculation has to be unobtrusively integrated into the existing OOO execution, i. e., speculations should have lower priority compared to the actual execution. Moreover, our approach is generally distributed and provides better scalability than master/slave approaches where a master usually becomes the bottleneck when the number of slaves is increased.

All methods presented in [54, 55, 71, 86, 94, 100, 120, 122, 125, 126, 137, 148, 150] require hardware support to speculate, e. g., by adding features to a processor to efficiently handle speculated data or to recognize wrong speculations. Rangan et. al parallelize loops where loop-carried dependencies are handled using control-flow speculation [176]. The approach differs to previously mentioned work in that it particularly considers loops, while others consider sequential programs in general. However, similar to previously mentioned related work, it requires special hardware, in particular versioned memory. Our approach runs completely in software and simply requires a SMP system, e. g., an Intel x86 or AMD64 multi-core architecture, which can be found in most desktop computers. Besides the requirement for special hardware, only [86, 120, 125, 126] include data speculation in their approaches, which is also done in our approach. Since our speculative approach is implemented in software, it can dynamically adapt to the needs such that it automatically selects tasks with high hit ratio. In case that the hit ratio falls below a user-defined threshold, the speculation turns itself off to save energy.

[132] considers techniques to efficiently speculate values for speculative execution. It addresses the issue of achieving a high hit ratio. Of course, our work would also benefit from good speculations, which is however not in the focus of the approach presented in Chapter 5.

Chapter 2

Preliminaries

2.1 Synchronous Model of Computation

The MoC describes the available set of actions and their costs with respect to complexity, i. e., the MoC is a formal description for interpreting a program description [154].

The synchronous MoC [24,83] splits the behavior of a system into logical steps, thereby communicating with an environment in each step. Each logical step is refered to as a reaction of the system or as a *macro step* [87]. In each reaction, the system reads all inputs from the environment to update its internal state and to compute all outputs. The behavior of the system, i. e., how it reacts on inputs, is described by a set of actions that are called *micro steps*. In each macro step, all micro steps are evaluated with the same variable environment. In particular, it is very important that variables do not change during the macro step. For this reason, all micro steps are viewed to be executed instantaneously, i. e., at the same point of time (as they are executed in the same variable environment). The instantaneous feedback due to immediate assignments to outputs can lead to so-called causality problems [27, 158–160, 162]. Compilers check the causality of a program at compile time with a fixpoint analysis that essentially corresponds to those used for checking the speed-independence of asynchronous circuits via ternary simulation [42]. Beside the causality analysis, compilers for synchronous languages often perform further checks to avoid runtime exceptions like out-of-bound overflows or division by zero. Moreover, most compilers for synchronous languages also allow the use of formal verification, usually by means of model checking.

To this end, the synchronous MoC abstracts from actual hardware architectures and their real-time behavior. This allows programmers to focus on their actual implementation of an algorithm without going into details about communication, synchronization and associated costs. Moreover, the way of evaluation provides implicit parallelism and synchronization, which is a promising feature for code generation for parallel architectures, which is in the focus of this thesis. The uniquely determined values of variables in a single macro step make synchronous programs deterministic, which is an important aspect in safety critical systems. Examples for languages that rely on the synchronous MoC are Quartz [156], Esterel [26] and Lustre [83].

All synthesis tools that have been implemented for the purposes of this thesis have been integrated into the Averest system[1]. The source language for the Averest system is Quartz. Nevertheless, the presented synthesis methods are *not* restricted to this language. They start from a very general representation of synchronous system, i.e., the SGAs [39, 156, 157]. Because all benchmarks have been implemented in Quartz, a brief introduction to this language is given in the next section. SGAs actions are explained afterwards in Section 2.1.2.

2.1.1 Quartz

The imperative synchronous language Quartz [156] is the source language for the Averest System and has been derived from Esterel. The most important addition to Esterel concerning this thesis are delayed assignments. They immediately evaluate the right-hand side of an expression in a macro step, while the actual assignment of the result is done in the succeeding macro step. Moreover, Quartz provides explicit non-determinism and foundations for a hybrid version of Quartz have been made in [22], thereby providing analog variable types and the capability to express hybrid systems. Non-determinism is an interesting feature for verification, whereas in synthesis of programs, one is only interested in deterministic programs.

A synchronous system described in Quartz is defined by an interface to its environment, consisting of a set of input and output variables, a set of local variables describing the internal state and the behavior of the system. Variables in Quartz have data types, which are split into atomic and composite data types, and also different storage types, which will be explained later on. Currently available data types are Boolean, signed and unsigned integers, real numbers and bit vectors. All types but Boolean *can* have an unbounded value range and in case of real numbers an arbitrary precision, i.e., the type definitions of Quartz correspond to the mathematical definitions. In turn, this implies that the types in Quartz are not restricted to hardware or software architectures, e.g., a 32-bit integer. In addition, Quartz supports composite types. In particular, arrays and tuples can be recursively nested to give a group of variables a meaningful representation, e.g., a three-dimensional vector consists of three components which can be represented as a tuple of three real numbers. The storage type basically declares how a variable behaves, i.e., which value it will have, if no assignment takes place in a macro step. The *event* typed variables are reset to a default value, i.e., a constant, while *memorized* typed variables keep their value from the preceding macro step.

The behavior of the system is described by different statements and modules. We will not deepen on the syntax of this language, because it might change in the future. Nevertheless, the semantics to describe synchronous systems is stable. Beside assignments to do calculations and to modify the system's state or outputs, the language provides a set of control statements. Figure 2.1 shows the synchronous program ABRO, which demonstrates the usage of typical control statements in synchronous languages. In principle, a module executes a statement S, which can be recursively defined. Thereby, S can consume

[1] http://www.averest.org

logical time steps, or it can be instantaneous, i. e., it is executed in zero time. Moreover, a statement can have conditional time consumption, i. e., only if a condition holds, time is consumed. This makes analysis more difficult and requires precise definition of a statement's behavior, e. g., using structural operational semantics transition rules as given in [156]. The translation of these statements to SGAs is considered to be given by a compiler front-end as described in [39]. To give the reader a first impression of synchronous programming, we will give an overview over some typical synchronous control statements instead of a formal description of these statements.

Some examples for statements that are used in the example in Figure 2.1 are

- $S := S_1; S_2; \ldots; S_n$: is a sequential composition of the statements S_1 to S_n. The control-flow passes through all statements, starting with S_1 and continuing with S_{i+1} when the control-flow leaves S_i. Although sounding quite simple, this is a non-trivial statement. If a statement S_i is instantaneous, i. e., it does not consume a logical time unit, its successor S_{i+1} is executed in the same macro step. Thus, the sequential composition behaves as a parallel composition for instantaneous statements.

- $S := S_1||S_2||\ldots||S_n$: is the synchronous parallel composition of the statements S_1 to S_n. All statements are driven by the same clock, i. e.,
  $\forall_{i,j\in\{1,\ldots,n\}}.(S_i$ consumes one logical step$) \leftrightarrow (S_j$ consumes one logical step).
  The control-flow of a synchronous parallel composition continues in S_i after other statements S_j terminate. The execution of a parallel statement S is completed after all sub-statements S_i terminated.

- $S :=$ `loop` $\{S';\}$: executes the statement S' an infinite number of times, whereas the statement must consume time, i. e., at least one macro step must be consumed. This is explained by the sequential composition: The `loop` $\{S';\}$ statement is the sequential composition of $S'; S'; S'; \ldots$. If S' is instantaneous, the statement must be unrolled and executed an infinite number of times in one macro step. However, the synchronous MoC requires a finite amount of micro steps per macro step.

- $S :=$ `abort` S' `when`(γ): statement S' is executed until it reaches the end of its control-flow or if γ holds. In particular, S' is executed macro step by macro step. As soon as γ holds, the control-flow directly leaves S', i. e., no micro steps of S' are executed in the macro step where γ holds. A variation of this statement allows one to execute all micro steps in the macro step where γ holds (weak abort). The statement above will always execute the very first macro step of S'. A variation checks also in the very first macro step condition γ (immediate abort) and behaves as described above.

- $S :=$ `await`(γ): this statement stops the control-flow until γ holds. The control-flow stops for at least one macro step. A variation of this statement allows one to exit the control-flow immediately when γ holds (immediate abort), i. e., `await`(γ) can be made instantaneous.

```
1: module ABRO(event bool ?a, ?b, ?r, !o) {
2:   loop {
3:     abort {
4:       { wa : await(a); || wb : await(b); }
5:       emit o;
6:       wr : await(r);
7:     } when(r);
8:   }
9: }
```

Figure 2.1: The synchronous program ABRO written in Quartz.

```
1: loop {
2:   i = ReadInputsFromEnvironment()
3:   (o, s') = F(i, s)
4:   WriteOutputsToEnvironment(o)
5:   s = s'
6:   macrostep : pause
7: }
```

Figure 2.2: Pseudo code for executing a synchronous system F.

2.1.2 Synchronous Guarded Actions

Synchronous guarded actions are a generic intermediate format to represent synchronous programs. They are in the spirit of classical guarded commands [64], which are widely used to describe the behavior of concurrent systems [52,65,96]. However, in our case, they follow the synchronous model of computation, i. e., their semantics is different compared to the classical counterpart.

Figure 2.2 shows the principle execution of a synchronous program F. According to the synchronous MoC, the behavior is split into macro steps. The loop body executes a single reaction of the system. For each reaction, the system reads inputs from its environment. This system also has an internal state, e. g., local variables in software or registers in hardware. F describes a function that computes output values and the updates of the internal state. Writing the outputs to the environment finalizes the reaction. During a single macro step, the values of all variables are uniquely determined. The configuration of all variables in a macro step is called a variable environment of a macro step.

In a synchronous system, the data type of variables is not restricted to a specific one, i. e., we can use commonly used types, e. g., bounded and unbounded, signed and unsigned integer numbers, real numbers, Boolean variables and bitvectors, but also commonly used structures as arrays and tuples (structures) of the mentioned types. In addition, variables can have different types of storage. In particular, we distinguish between *memorized* and

event variables. The actual behavior of these types is described later in this section. The distinction becomes of interest for the speculative OOO execution (see Section 5.2.1.2).

Assuming that a synchronous system is described by a set of SGAs, function F represents the execution of this set. Due to the synchronous MoC, the module executes all actions in zero time. A SGA is formally given as $\langle \gamma \Rightarrow \mathcal{A} \rangle$, where the guard γ is a Boolean condition that determines whether the action $\mathcal{A}$ is activated in the current macro step or not. Basically, $\mathcal{A}$ can be any type of atomic action, e. g., an assignment or a call of a function. Our approach is not limited to assignments, but for the sake of simplicity, subsequently, we will focus on this type of action.

In this thesis, we consider two types of assignments, i. e., *immediate assignments* of the form $x = \tau$ and *delayed assignments* of the form $\texttt{next}(x) = \tau$. When triggered by their guard, both types of assignment evaluate its right-hand side τ in the current macro step. Immediate assignments cause x to have the value of τ in the current macro step, while delayed assignments transfer the value in the following macro step, i. e., a delayed assignment does not change the value of x in the current macro step. These types of assignments are comparable to wires (immediate assignments) and registers (delayed assignments) in synchronous hardware circuits.

The value of a variable x depends on a set of SGAs having x on the left-hand side of the action. At this moment, if none of these actions is activated in the current step, i. e., if the guards of all of these actions are evaluated to false, x is not assigned a value. Synchronous languages like Esterel and Lustre allow a variable to have no value in a macro step. In this thesis, we focus on strictly synchronous programs, which, in contrast to Esterel and Lustre, require the value of a variable to be uniquely determined in each macro step. In case that no value is assigned to x by a SGA, the value is set by an implicit action, i. e., the so-called *default action*. The actual behavior of a default action depends on the variable's storage type, i. e., it either keeps its value from the preceding macro step (memorized variable) or it is reset to a default value (event variable). While memorized storage corresponds to the native behavior in software, i. e., a variable keeps its value until it is changed by an instruction, the event variable describes the behavior of a wire in hardware, that has a default state until it is set by a gate.

Figure 2.3 depicts the synchronous program ABRO (see Figure 2.1) after compiling it into SGAs. Beside the actual behavior, which is described by these actions, it contains also information about the interface, i. e., the inputs from and outputs to the environment, and local variables. Section **default actions** describes the actions, which assign a default value to the writable variables. A default action consists of two parts which is a default value for the very first macro step (initial assignment) and a default value for all other macro steps. This is necessary for two reasons: (1) It allows to set the boot location, which basically identifies, where the control-flow starts, and (2) default actions for memorized variables will assign the value from the preceding macro step, i. e., $\texttt{trans}\ x = \texttt{prev}\ x$, which is initially not available. Note, that default actions are given implicitly and have been explicitly added for this example. Moreover, default actions will become more important for the general communication optimization in Section 4.3.

```
 1: module ABRO
 2:   interface :
 3:       a : input event bool
 4:       b : input event bool
 5:       r : input event bool
 6:       o : output event bool
 7:   locals :
 8:       init : local memorized bool
 9:       wa : local event bool
10:       wb : local event bool
11:       wr : local event bool
12:   flow :
13:       init ⇒ next(wa) = 1
14:       init ⇒ next(wb) = 1
15:       ¬r ∧ wa ∧ ¬a ⇒ next(wa) = 1
16:       r ∧ (wr ∨ wa ∨ wb) ⇒ next(wa) = 1
17:       ¬r ∧ wb ∧ ¬b ⇒ next(wb) = 1
18:       r ∧ (wr ∨ wa ∨ wb) ⇒ next(wb) = 1
19:       ¬r ∧ wr ⇒ next(wr) = 1
20:       ¬r ∧ (a ∧ wa ∧ b ∧ wb) ⇒ next(wr) = 1
21:       ¬r ∧ (¬wa ∧ b ∧ wb) ⇒ next(wr) = 1
22:       ¬r ∧ (a ∧ wa ∧ ¬wb) ⇒ next(wr) = 1
23:       ¬r ∧ (a ∧ wa ∧ b ∧ wb ∨ ¬wa ∧ b ∧ wb ∨ ¬wb ∧ a ∧ wa) ⇒ o = 1
24:   default actions :
25:       init init = 1, trans init = 0
26:       init wa = 0, trans wa = 0
27:       init wb = 0, trans wb = 0
28:       init wr = 0, trans wr = 0
29:       init o = 0, trans o = 0
```

Figure 2.3: Synchronous guarded actions after converting control-flow to data-flow for ABRO program.

In this thesis, we assume that a given program is logically correct and constructive. Logical correctness of a program requires the absence of write conflicts and causality problems. The former refers to the ability that two or more SGAs write to the same variable in the same macro step (see (1) in Figure 2.4). Assigning two different values to a variable in a single macro step violates the synchronous MoC and is forbidden. Causality problems or causality cycles describe mutual dependencies between SGAs as shown in (2) and (3) in Figure 2.4. In (2) in Figure 2.4, both actions allow two valid variable configurations, i. e., $\gamma_A \wedge \gamma_B$ (both actions fire) and $\neg\gamma_A \wedge \neg\gamma_B$ (no action fires). In contrast, the actions in (3) in Figure 2.4 have no solution: assuming that γ_A is set, the action that emits γ_B must

(1) Write Conflict	(2) Non-Determinism	(3) Mutual Exclusion
$\gamma_A \Rightarrow x = 1$	$\gamma_A \Rightarrow \texttt{emit}\ \gamma_B$	$\neg\gamma_A \Rightarrow \texttt{emit}\ \gamma_B$
$\gamma_A \wedge \gamma_B \Rightarrow x = 2$	$\gamma_B \Rightarrow \texttt{emit}\ \gamma_A$	$\gamma_B \Rightarrow \texttt{emit}\ \gamma_A$

Figure 2.4: Semantic issues in compilation of synchronous programs. (1) Write conflict: if $\gamma_A \wedge \gamma_B$ holds, two different values are assigned to x; (2) Causality problem: two valid variable environments result in non-determinism; (3) Causality problem: incorrect program because mutual exclusion of actions results in no valid variable environment.

not be fired. As a result, the other action must not be fired, too, resulting in an unset γ_A which is a contradiction to the assumption. By analogy, assuming that γ_A is unset, the action that emits γ_B must be fired, and therefore, the second action has to be fired, too. This results in a set γ_A which is also a contradiction to the previously made assumption. Hence, this set of actions does not provide a valid variable configuration and is considered as a logically incorrect program.

Methods to detect and eliminate these problems [42, 68, 124, 134, 159] have been developed and integrated into compilers to guarantee the absence of these problems. *Hence, we can assume that a system for the synthesis process is logically correct and free of semantic issues of synchronous programs.*

In this thesis, we need to determine dependencies between SGAs and to modify SGAs. The dependencies between actions is handled in the following section, while modifications depend on the synthesis approaches. In the following, we distinguish between *variables* and *cells*. As mentioned, a variable can be a scalar but also a structure, e. g., an array or a tuple, whereas, a cell is always a scalar value, i. e., it describes a scalar or a part of a structure. For instance, let a be a one-dimensional array, then a is referred to be the variable and its elements $a[0],\ldots,a[N-1]$ are the (memory) cells. For most approaches presented in this thesis, it is sufficient to abstract from cells and to use variables. However, some approaches, e. g., the array optimization in Section 4.2, needs a precise distinction between variables and cells.

In the following, let $\mathsf{Cells}(\tau)$ denote the cells occurring in the expression τ. The variable that is addressed by an expression e is obtained by function $\mathsf{varOfCell}\,(e)$. In particular, the function $\mathsf{varOfCell}\,(e)$ returns the parameter e if it is a scalar variable, and if e designate a cell of a structure, e. g., of an array, then the name of the structure will be returned, i. e., $\mathsf{varOfCell}\,(a[0]) = \mathsf{varOfCell}\,(a[i+1]) = a$. Finally, $ID(e)$ denotes a set containing the index expression of an expression e unless this is a constant, e. g., $ID(a[i]) = \{i\}$, $ID(a[0]) = \{\}$, $ID(i) = \{\}$.

Definition 1 (Read Cells, Immediate Written Cells and Delayed Written Cells)**.** *The cells read and immediate/delayed written by a SGA are defined as follows:*

$$\begin{aligned}
&\mathsf{rdVars}\,(\gamma \Rightarrow x = \tau) := \mathsf{Cells}(\gamma) \cup (\textstyle\bigcup_{\delta \in ID(x)} \mathsf{Cells}(\delta)) \cup \mathsf{Cells}(\tau)\\
&\mathsf{wrNowVars}\,(\gamma \Rightarrow x = \tau) := \mathsf{varOfCell}\,(x)\\
&\mathsf{wrNowVars}\,(\gamma \Rightarrow \texttt{next}(x) = \tau) := \{\}\\
&\mathsf{wrNxtVars}\,(\gamma \Rightarrow x = \tau) := \{\}\\
&\mathsf{wrNxtVars}\,(\gamma \Rightarrow \texttt{next}(x) = \tau) := \mathsf{varOfCell}\,(x)
\end{aligned}$$

Note that the definition of $\mathsf{rdVars}\,(A)$ must consider the left-hand side of assignment A, which may contain variables that are read, i. e., indices in an array access. Based on $\mathsf{rdVars}\,()$, $\mathsf{wrNowVars}\,()$ and $\mathsf{wrNxtVars}\,()$, we can define the set of actions that read from, respectively write to a variable in Definition 2.

Definition 2 (Reading and Writing SGA)**.** *The set of SGAs reading and writing to a variable v is determined as follows:*

$$\begin{aligned}
&\mathsf{rdActs}\,(v) := \{\mathcal{A} \mid v \in \mathsf{rdVars}\,(\mathcal{A})\}\\
&\mathsf{wrActs}\,(v) := \{\mathcal{A} \mid v \in \mathsf{wrNowVars}\,(\mathcal{A}) \cup \mathsf{wrNxtVars}\,(\mathcal{A})\}
\end{aligned}$$

Definition 1 allows to analyze data dependencies between SGAs. This is necessary to build the dependency graph as explained in the following section.

2.1.3 Action Dependency Graph

The theory of the synchronous MoC requires SGAs to be executed in zero time, which is practically impossible. Even computations in combinational and digital circuits require time and do not allow a native execution of a synchronous program. Different methods have been presented to execute synchronous programs on sequential architectures. Edwards presents in [67] different ways to compile concurrent languages to sequential code. To this end, all methods end up with the same task, which is particularly to order instructions according to their data dependencies. All synthesis tools start from a set of SGAs and build the action dependency graph (ADG), which is a graph for analyzing the dependencies between the actions. Since SGAs allow us to write in different macro steps, we got different kinds of write accesses. In particular, we allow immediate and delayed writes, and therefore, we need to define write types:

Definition 3 (Immediate and Delayed Write Type)**.** $D = \{\mathit{immediate}, \mathit{delayed}\}$ *denotes the set of write types of SGAs. The SGA $\langle \gamma \Rightarrow x = \tau \rangle$ has write type immediate and $\langle \gamma \Rightarrow \texttt{next}(x) = \tau \rangle$ has write type delayed.*

The distinction between the write types becomes necessary due to their different semantics in the ADG, which will be explained in more detail soon. The write types of SGAs allow to define the ADG:

Definition 4 (ADG). *The ADG is a graph $G = (V, E)$ consisting of vertices V representing the SGAs, and directed edges $E \subseteq D \times V \times V$ representing the dependencies between the actions and the type D of a dependency (see Definition 3).*

Definition 5 (Immediate and Delayed Dependencies). *We call $e \in E$ an* immediate dependency *iff $e = (immediate, v_1, v_2)$, and* delayed dependency *iff $e = (delayed, v_1, v_2)$.*

Definition 6 (Direct and Indirect Dependencies). *Let $e = (d, v_1, v_2) \in E$, then e is called* direct dependency. *Let E^* be the transitive closure of E, then $e = (d, v_1, v_2) \in (E^* \setminus E)$ is an* indirect dependency.

The ADG is intended to determine the order of the SGAs for the execution of a single macro step. According to the synchronous MoC, the SGAs must be executed in the same (unique) variable environment. Hence, the basic idea for the scheduling is to execute all actions writing to a variable x before this variable is read by other actions to create the illusion that no variable changes during the execution. The different timing of writes, i. e., immediate and delayed writes, have different semantics and require different scheduling policies. An immediate write changes the value of a variable in the current macro step and must be executed before executing any action reading this variable. In the ADG this will result in an immediate dependency, which is a read after write (RAW) dependency. In contrast, delayed writes represent write after read (WAR) dependencies because the variable that is written to, must be set in the next macro step. In particular, the evaluation must take place in the current variable environment, while the assignment must take place at the beginning of the next macro step because it addresses the variable environment of the succeeding macro step.

Note that some literature has different semantics of the term immediate dependency. Marwedel uses in [127] the term immediate dependency to describe a dependency that is called a direct dependency in this context. However, we need the term immediate to describe the kind of write access in the source action in our graph.

Using Definition 1 we can determine the edges for an ADG G, i. e., the dependencies between the SGAs:

Definition 7 (Edges of ADG). *Given is the ADG $G = (V, E)$. Let $A_1, A_2 \in V$, then $(immediate, A_1, A_2) \in E \leftrightarrow \mathsf{wrNowVars}(A_1) \cap \mathsf{rdVars}(A_2)$, and $(delayed, A_2, A_1) \in E \leftrightarrow \mathsf{wrNxtVars}(A_1) \cap \mathsf{rdVars}(A_2)$.*

The direction of an edge denotes the flow direction, i. e., from writing to reading action. The interpretation of the edge for scheduling purposes depends on the final synthesis process.

When creating sequential software from an ADG, immediate writes must be executed before a value is read. Whereas, the scheduling of delayed writes allows different implementations. According to the synchronous MoC, the right-hand side must be evaluated in the current macro step, while the assignment itself must be done at the beginning of the next macro step. A general solution is to store the result of the evaluation of the right-hand side of a delayed action in a so-called carrier variable. At the beginning of the next macro

```
 1: init = 1
 2: wa = 0
 3: wb = 0
 4: wr = 0
 5: o = 0
 6: loop {
 7:    (a, b, r) = ReadInputsFromEnvironment()
 8:    wa' = 0
 9:    wb' = 0
10:    wr' = 0
11:    o = 0
12:    init' = 0
13:    if (init)   then wa' = 1
14:    if (init)   then wb' = 1
15:    if (¬r ∧ wa ∧ ¬a)   then wa' = 1
16:    if (r ∧ (wr ∨ wa ∨ wb))   then wa' = 1
17:    if (¬r ∧ wb ∧ ¬b)   then wb' = 1
18:    if (r ∧ (wr ∨ wa ∨ wb))   then wb' = 1
19:    if (¬r ∧ wr)   then wr' = 1
20:    if (¬r ∧ (a ∧ wa ∧ b ∧ wb))   then wr' = 1
21:    if (¬r ∧ (¬wa ∧ b ∧ wb))   then wr' = 1
22:    if (¬r ∧ (a ∧ wa ∧ ¬wb))   then wr' = 1
23:    if (¬r ∧ (a ∧ wa ∧ b ∧ wb ∨ ¬wa ∧ b ∧ wb ∨ ¬wb ∧ a ∧ wa))   then o = 1
24:    WriteOutputsToEnvironment(o)
25:    init = init'
26:    wa = wa'
27:    wb = wb'
28:    wr = wr'
29: }
```

Figure 2.5: Example for creating a sequential schedule for ABRO. The copying of carrier variables to the actual variables has been placed at the end of the loop.

step, the values of carrier variables are copied to their corresponding variables. This allows an arbitrary scheduling of delayed actions, and thereby resulting in more parallelism since there is no additional scheduling dependency. The downside of this method is the increase of memory usage and an additional overhead for copying the values of the carrier variables to the actual variables. Figure 2.5 depicts an example of mapping a set of immediate and delayed actions to sequential code. As can be seen, the values of the carrier variables are copied at the end of the loop. Since testing the (trivial) loop condition will have no side

effects on the values of the system's variables, this has the same semantics as executing the copying process at the beginning of the loop.

Another variant to schedule delayed actions is to add specific scheduling constraints which avoid the insertion of carrier variables. We previously explained that dependencies from delayed actions to actions reading the corresponding variable are WAR dependencies. The idea is to schedule delayed writes after all actions that read the corresponding variable using the WAR dependencies. This will not always result in a valid schedule due to potential cyclic scheduling dependencies. Thus, the insertion of carrier variables is in some cases inevitable to create a sequential schedule. The creation of sequential code is not in focus of this thesis. Hence, we assume to have a set of SGAs, where carrier variables have been added to simplify the scheduling of delayed actions.

From the graph, we can immediately derive the restrictions for the execution of the SGAs of a synchronous system. An action can only be executed if all read variables are known, i. e., all variables occurring in the guard, the right-hand side of the assignment and variables occurring in array indices of the left-hand side. A variable is thereby known if all actions writing to it have already been executed in previous micro steps of the same macro step. To this end, an action A can be executed if all actions that have a dependency to A have been executed before.

2.2 Data-Flow Process Networks

Data-flow process networks have a long history, and their roots date back to the 1970s. The most known variant are certainly Kahn networks introduced in [97], while many other variants exist [23, 135, 177]. All models have in common that they consist of a set of nodes which process data concurrently and independently from each other. These nodes communicate by sending tokens through unbounded point-to-point FIFO buffers. In particular, a DPN is a completely untimed model. The behavior of each node is described by firing rules which are triggered by the availability of data. Hence, the nodes in a DPN are not driven by a global clock as done in synchronous programs.

The behavior of a single node is not restricted to a specific MoC. In general, each firing rule consists of a guard that defines when the node has to be activated and a function that takes a given set of inputs to compute the outputs of a node. In this thesis, we will use the synchronous MoC to describe the behavior. Changing the parts of the synchronous system does not improve the translation steps in Chapter 4, but keeping them as long as possible in their original MoC avoids non-reversible conversions. Definition 8 gives a formal definition of DPN nodes.

Definition 8 (DPN Node)**.** *A node in a DPN is defined as follows:*

$$\begin{aligned} &\mathit{DPNNode} := \textbf{list of } \mathit{FiringRule} \\ &\mathit{FiringRule} := \textbf{list of } \langle \gamma_B \Rightarrow \mathit{Guarded\ Action} \rangle \end{aligned}$$

In this thesis, we use firing rules to describe the behavior of a single node. In general, a single node may have several firing rules, where a single firing rule consists of (1) a

condition stating when it has to be activated, (2) a list of input buffers that have to be read, (3) a function that is applied to the input tokens, and finally 4) the output buffers where the produced tokens are put.

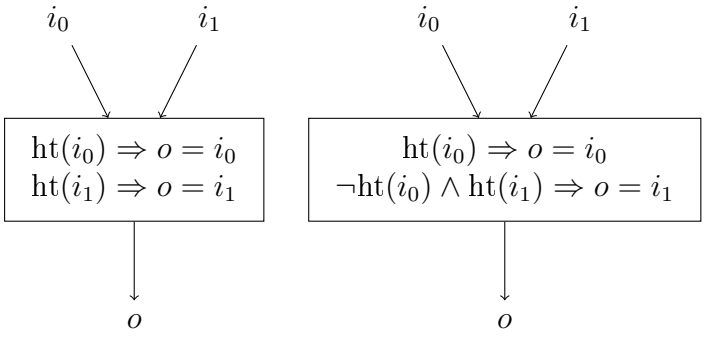


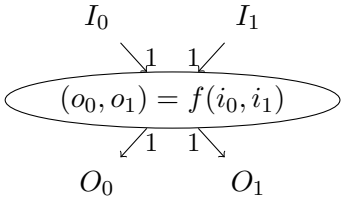
Figure 2.6: Non-determistic DPN

Although DPNs look pretty simple, their correct and efficient implementation has to consider several difficult problems. The first problem is to ensure boundedness of memory: In the MoC, the size of the buffers is typically unbounded, so for synthesis one has to avoid that there is a FIFO buffer where more data values are produced than consumed in the long run. To this end, additional means like static scheduling (if possible) [32, 33, 70, 113, 114] or the introduction of acknowledgements to introduce back-pressure are required to assure bounded memory requirements.

Another problem is to ensure the determinacy of a DPN: It is possible that firing rules of a node overlap, i.e., more than one is enabled at some point of time. In this case, any enabled firing rule can be chosen, which obviously gives rise to a nondeterministic behavior of the whole network. For instance, consider the node given on the left-hand side of Figure 2.6: if both input buffers are filled, both firing rules are activated. Furthermore, according to the Kahn principle [97], nondeterministic behavior of the network is even possible if rules do not overlap. The example on the right-hand side of Figure 2.6 illustrates this effect: the ordering of the values sent through o depends on the arrival time of inputs on i_0 and i_1. For this reason, one has to ensure that each node implements a continuous function, which can be done by simple additional requirements as in Kahn's non-blocking reads [97].

To this end, checking for liveness, deadlock-freedom [184], and boundedness of buffers is undecidable, but necessary – especially in safety-critical systems. The behavior of nodes must be taken into account for a precise analysis because it controls the number of tokens that are read and written per node.

The analysis is much easier for static data-flow graphs (SDFGs) [77, 114]. SDFGs represent a subset of DPNs, characterized by the number of tokens that are read from and written to buffers that is always constant. The left part of Figure 2.7 shows an example of a SDFG node which has two inputs and two outputs. Each time the node fires, exactly one token is consumed from each input, and exactly one token is produced for each output. The




```
thread T_N {
  loop {
    i_0 = I_0.pop()
    i_1 = I_1.pop()
    (o_0, o_1) = f_N(i_0, i_1)
    O_0.push(o_0)
    O_1.push(o_1)
  }
}
```

Figure 2.7: *Left:* SDFG node with 2 inputs and 2 outputs. *Right:* Pseudo code for left SDFG node

corresponding pseudo code to implement this behavior in a typical imperative programming language is shown on the right-hand side of Figure 2.7.

In the following, we will use the term DPNs to describe SDFGs. A translation from SGAs to DPNs and the mapping of DPNs to parallel software is given in Chapter 4. We also present two approaches to reduce communication costs. Moreover, Chapter 5 introduces OOO execution of DPNs to achieve a better load-balancing.

2.3 Parallel Programming APIs

Parallel architectures are manifold in their level of parallelism and their particularly programming. It seems natural to make use of existing APIs for parallel programming. These architectures have in common that parallel programming still needs to be coded explicitly by the programmer. In particular, it is necessary to manually partition a program into threads and to manage synchronization explicitly. In the following some of these APIs and their features are presented.

We will give a code example for each presented API based on the synchronous program given in Figure 2.8. The ADG for this program is shown in Figure 2.9. Assume that the nodes of the graph contain the following SGAs: node 1 reads value a and b from the environment. a is given to node 2 and b is given to node 3. Nodes 2 and 3 compute values for x and y based on these input values and forward their result to node 4, which computes the final output value. The actual output of the system is written by node 5.

2.3.1 PThreads

A thread is a single instruction sequence describing a behavior in a process. A process usually represents an application in an operating system. It may have several threads, which share the same resources, i. e., threaded programming is used in shared memory systems. Multiple threads in a single process usually result from a fork. Scheduling policies,

```
 1: module Example(nat ?a, ?b, nat !o) {
 2:   loop {
 3:     (a, b) = ReadInputsFromEnvironment()
 4:     x = F(a)
 5:     y = G(b)
 6:     o = H(x, y)
 7:     WriteOutputsToEnvironment(o)
 8:     pause
 9:   }
10: }
```

Figure 2.8: Synchronous program that serves as source for the code examples presented for the different APIs.

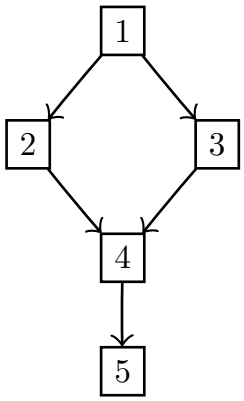


Figure 2.9: ADG of example in Figure 2.8.

i. e., mapping of threads to processors is up to the operating systems and may differ. If the number of threads to be executed on a system exceeds the number of available cores, many OS use a time-slice based scheduling to emulate a concurrent execution of the threads. This is achieved by quickly switching the context between threads.

One way to create parallel software is to create several threads running concurrently and executing independent instruction streams. Parallelism is usually limited, such that synchronization between threads is often necessary and can be done using synchronization primitives: semaphores, locks and conditions. An actual implementation of these primitives depends on the operating system and the underlying hardware architecture. A clear downside is the additional effort to program these systems, i. e., a programmer must explicitly partition its program / algorithm and finally add necessary synchronization on his/her own. Moreover, if the programmer does not provide a flexible usage of cores, i. e., if the number of threads is fixed, some cores of a system may stay un-utilized or additional overhead for scheduling can arise due to time-sliced scheduling. Usage of a memory hierarchy

with different levels is state-of-the-art in nowadays computers. This speeds up memory accesses for often accessed addresses, which is based on the idea of locality of data in many applications. Time-sliced scheduling can result in cache thrashing, i. e., context switches lead to a mutual dislodging of data of different threads.

One way to program with threads is to use the POSIX threads API, which is provided for UNIX based operating systems. The functions and structures from this API used in the context of this thesis are typical thread functions, particularly forks, semaphores, locks and conditions. Hence, they should be also available in other thread APIs.

The translation of the example given in Figure 2.9 is depicted in Figure 2.10. Data dependencies are translated by using FIFO buffers, which makes each node working on its own copy of variables. The FIFO buffers allow one to run all nodes decoupled as described in Section 2.2. Hence, this code exploits parallelism over multiple macro steps. In contrast, considering the given ADG as a DFG would result in a translation of data dependencies as control-flow constructs, e. g., locks and conditions. Variables would be then shared across the nodes and avoid decoupled execution. For instance, Node1 must wait until Node2 and Node3 have processed the values of a and b before they can be overwritten. This is basically done for the OpenMP version presented in the next section.

To this end, PThreads requires the programmer to describe parallelism in software from the hardware view.

2.3.2 Task-Based Execution

Approaches to get a better workload are based on task-based execution [34,51,90,136]. The concept of these approaches is to split computations into atomic parts – so-called tasks – instead of threads. Different libraries are available for task-based execution, e. g., OpenMP, KAAPI [107], StarSS [148,169]. In general, such a task is an atomic piece of work, i. e., communication with other tasks is done at the beginning and the end of the task. These tasks are light-weight substitutes of threads which can be scheduled to processing elements. The execution of a task is not interrupted, i. e., it can be completely executed, because it does not communicate with other tasks during its execution. In contrast, threads may be interrupted due to communication primitives or the prevalently used time-sharing scheduling policy.

Programmers must explicitly identify tasks in their applications. Tasks that are ready to be executed are scheduled into a task queue. A set of worker threads are responsible for executing these tasks. These worker threads are created by the API, e. g., OpenMP. The number of worker threads depends on the actual system and will be usually set to the number of available computational units, i. e., processor cores. A worker thread iteratively gets a task from the task queue and executes the corresponding task. Each task can thereby create new tasks. Fitting the number of worker threads to the number of available computational units reduces the overhead of context switches because tasks are handled as atomic work portions. Moreover, tasks can be more light-weight compared to threads to gain more parallelism. The overhead of getting a task from the task queue might then get too high for some specific architectures. In that case, grouping tasks can reduce the

```
 1: FIFOQueue A, B, X, Y
 2: thread Node1 {
 3:   loop {
 4:     (a, b) = ReadInputsFromEnvironment()
 5:     A.push(a)
 6:     B.push(b)
 7:   }
 8: }
 9: thread Node2 {
10:   loop {
11:     a = A.pop()
12:     x = F(a)
13:     X.push(x)
14:   }
15: }
16: thread Node3 {
17:   loop {
18:     b = B.pop()
19:     y = G(b)
20:     Y.push(y)
21:   }
22: }
23: thread Node4 {
24:   loop {
25:     x = X.pop()
26:     y = Y.pop()
27:     o = H(x, y)
28:     O.push(o)
29:   }
30: }
31: thread Node5 {
32:   loop {
33:     o = O.pop()
34:     WriteOutputsToEnvironment(o)
35:   }
36: }
```

Figure 2.10: Translation of the DFG from Figure 2.9 to parallel code using threaded execution.

overhead. To this end, task-based execution provides better flexibility to architectures, particularly the number of computational units.

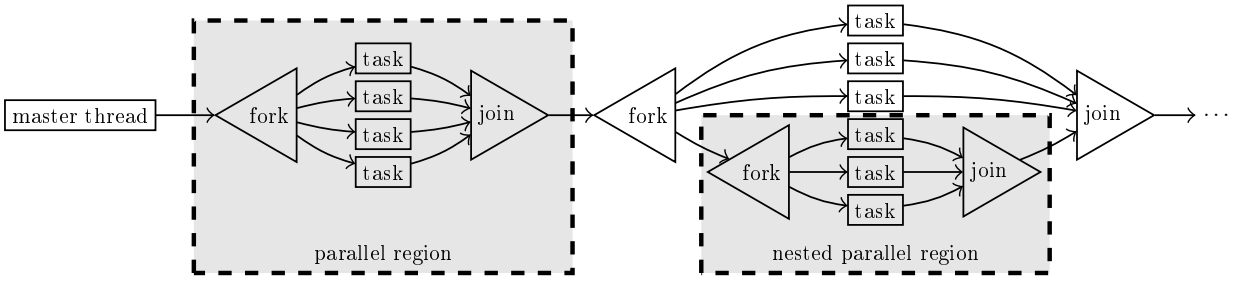


Figure 2.11: Fork-join model of OpenMP: structured control-flow allows concurrent execution.

```
1: #define N   1000000
2: int sum;
3: int a[N];
4: sum = 0;
5: #pragma omp for reduction(+ : sum)
6: for(i = 0; i < N; i++) {
7:     sum = sum + a[i];
8: }
```

Figure 2.12: Example for an OpenMP program: the loop computes the sum of the elements in a given array a.

2.3.2.1 OpenMP

Open Multiprocessing (OpenMP) [136] is an API for easy programming of parallel applications for SMP systems. It uses the task-based approach to exploit parallelism. The main advantage of OpenMP is its quite simple usage, especially to parallelize `for`-loops. The programmer does not have to care about creation or management of threads which execute the tasks. Nevertheless, parallelism must be explicitly expressed by compiler directives. There are two ways to declare parallel code: (1) loops can be declared as concurrently executable, and (2) explicit definition of work. The former method allows one to automatically split a `for`-loop, such that iterations are executed in parallel. This aims at the concurrent execution of the same code on different data. In contrast, the explicit task definition allows one to execute arbitrary blocks of code in parallel.

The execution model of OpenMP is based on a fork-join model. A program always starts with the execution of a sequence of instructions. A fork may split the control-flow into multiple paths which are merged at the join again. This can be recursively applied, i. e., a single path can be split again into multiple paths (see Figure 2.11). Each single path represents a task. Hence, OpenMP executes tasks in control-flow dependencies. Data to compute results of tasks is implicitly sent through the shared memory, i. e., OpenMP is intended to be used in SMP. All kinds of parallelization declaration in OpenMP require the

```
 1: loop {
 2:   (a, b) = ReadInputsFromEnvironment()
 3:   #pragma omp sections
 4:   {
 5:     #pragma omp section
 6:     {
 7:       x = F(a)
 8:     }
 9:     #pragma omp section
10:     {
11:       y = G(b)
12:     }
13:   }
14:   o = H(x, y)
15:   WriteOutputsToEnvironment(o)
16: }
```

Figure 2.13: Translation of the ADG from Figure 2.9 to parallel code using OpenMP.

programmer to take care of data dependencies. For instance, the iterations of a loop may have dependencies as shown in Figure 2.12. The loop computes the sum of the elements in array a. Additional directives allow one to express these dependencies. In the given example, the reduction directive results in the creation of a local copy of *sum* for each task. After all tasks have been executed, the sum of all local copies are added and stored in the actual (global) copy of *sum*. To this end, OpenMP is a quite convenient API for parallel programming, but it still requires the programmer to partition non-loop code and to do the dependency analysis.

Approaches to apply OpenMP to distributed memory systems have been proposed in [66]. Modifications to the code are necessary to ensure correct recognition of data dependencies. This is necessary due to the distributed memory system, where data has to be explicitly sent between the computation nodes. An installation of this tool failed. For that reason, it was not considered in the benchmarks for synthesis to cluster architectures.

The translation of the example given in Figure 2.9 is depicted in Figure 2.13. In contrast to PThreads, the solution for OpenMP considers the given ADG as a control-flow graph. The execution of nodes is still triggered in data-flow order, but only one macro step is executed at a time. Hence, this version exploits the parallelism in a single macro step and neglects explicit data transportation at the same time.

2.3.2.2 SmpSS

The StarSS compiler series from the Barcelona Supercomputing Center [148,169] target the execution of DFGs on parallel architectures. A particular derivative of these compilers is

SmpSS [139]. It allows one to execute DFGs, i. e., it allows one to execute tasks according to their data-dependencies. This is contrary to OpenMP, which requires a structured control-flow for parallel task execution. In SmpSS, the nodes of a given DFG are described as functions. Dependencies for each function are defined by a preceding compiler directive.

Similar to OpenMP, the ADG of the example given in Figure 2.9 is considered as a DFG. The translation of the example to parallel code using SmpSS is depicted in Figure 2.14. A source-to-source compiler for SmpSS translates the given compiler directives to compute the dependencies between the functions. The actual execution is triggered by calling the functions describing the tasks. To enable data-flow execution, the calls have to be enclosed into further compiler directives.

2.3.2.3 Intel TBB and Intel Cilk

The Intel Cilk [74] is a task-based library that focuses on the concurrent execution of recursive algorithms. For each recursive call, a new task can be generated using the Cilk library. This is especially helpful to parallelize divide-and-conquer algorithms. A recursive function often splits a problem into several smaller problems. Afterwards, a recursive call is made for each sub-problem, i. e., each call is usually independent of other calls to that function in the same recursion level. Hence, all these calls can be executed in parallel. Recursive problem solving is not a goal in the presented synthesis methods, therefore Cilk is not of interest.

The Intel Threading Building Blocks (TBB) [90] provides a wide collection of functions to concurrently execute tasks. The range starts from loop-parallelization as done in OpenMP to functions that allow spawning of recursive function calls. The syntax of OpenMP is more convenient for both, programming and code creation. Similar to Cilk, recursive problem solving is not a goal in this thesis. Hence, we did not use the Intel TBB library for generation of parallel code.

Apart from functions to create parallel applications, the Intel TBB library provides thread-safe classes and structures. In the context of this thesis, two important structures are used: (1) FIFO queues and (2) atomic data types. The former is used to implement communication in DPN applications (see Chapter 4). Atomic data types are mainly used to provide atomic read-and-modify operations which are required in the OOO execution in Chapter 5. To this end, we only make use of classes and structures of the Intel TBB library. The creation of threads is left to other APIs. For that reason we neglect code examples for Intel TBB and Cilk.

2.3.3 MPI

The message passing interface (MPI) [41] defines an interface standard to provide communication functions in architectures with distributed memory, e. g., clusters. These parallel architectures have multiple processing units where each has its own memory. MPI functions abstract from hardware and operating system dependent details, e. g., networking interface and network layers, by providing a unique standard for communication functions. More-

```
 1: #pragma css task output(a, b)
 2: function Node1(a, b) {
 3:    (a, b) = ReadInputsFromEnvironment()
 4: }
 5: #pragma css task input(a) output(x)
 6: function Node2(a, x) {
 7:    x = F(a)
 8: }
 9: #pragma css task input(b) output(y)
10: function Node3(b, y) {
11:    y = G(b)
12: }
13: #pragma css task input(x, y) output(o)
14: function Node4(x, y, o) {
15:    o = H(x, y)
16: }
17: #pragma css task input(o)
18: function Node5(o) {
19:    WriteOutputsToEnvironment(o)
20: }
21: function main() {
22:    loop {
23:       #pragma start
24:       Node1(a, b)
25:       Node2(a, x)
26:       Node3(b, y)
27:       Node4(x, y, o)
28:       Node5(o)
29:       #pragma finish
30:    }
31: }
```

Figure 2.14: Translation of the ADG from Figure 2.9 to parallel code using SmpSS.

over, MPI applications can be executed in clusters, network-on-a-chip (NoC) architectures and even in shared-memory systems, i. e., it aims at a broad range of architectures.

Beside the abstraction of communication, MPI provides a framework to start an application. This is done by executing the same program on a number of specified computational units. An MPI application follows the SPMD execution scheme: the application is executed on all available computational units, i. e., each computational unit executes the same instruction stream. A unique identifier for each node is used to determine the actual work

```
1: function main() {
2:   MPI_Init()
3:   MPI_Comm_rank(MPI_COMM_WORLD, &rank)
4:   if (rank = 0) {
5:     loop {
6:       (a, b) = ReadInputsFromEnvironment()
7:       MPI_Send(&a, target = 2)
8:       MPI_Send(&b, target = 3)
9:     }
10:  } else if (rank = 1) {
11:    loop {
12:      MPI_Recv(&a, source = 1)
13:      x = F(a)
14:      MPI_Send(&x, target = 4)
15:    }
16:  } else if (rank = 2) {
17:    loop {
18:      MPI_Recv(&b, source = 1)
19:      y = G(b)
20:      MPI_Send(&y, target = 4)
21:    }
22:  } else if (rank = 3) {
23:    loop {
24:      MPI_Recv(&x, source = 2)
25:      MPI_Recv(&y, source = 3)
26:      o = H(x, y)
27:      MPI_Send(&o, target = 5)
28:    }
29:  } else if (rank = 4) {
30:    loop {
31:      MPI_Recv(&o, source = 4)
32:      WriteOutputsToEnvironment(o)
33:    }
34:  }
35: }
```

Figure 2.15: Translation of the DFG from Figure 2.9 to parallel code using MPI.

that has to be done. Figure 2.15 depicts the translation of the example in Figure 2.9 to C code using MPI. The code is simplified for the sake of space but still shows the working principle: multiple-instruction-multiple-data (MIMD) execution requires that the main function is executed by all computational units. After the initialization of MPI, each com-

putational unit reads its identifier, i. e., a consecutively numbered integer. Depending on its identifier, a computational unit executes the code for a specific node of the ADG as shown in Figure 2.15.

The given example uses blocking communication, i. e., a call to MPI_Send or MPI_Recv blocks until the communication has been acknowledged by the corresponding receive and send command, respectively. Moreover, MPI provides non-blocking communication, which is initiated by functions with similar names: MPI_ISend and MPI_IRecv. They differ in that they immediately return without waiting for finishing a communication process. This allows one to continue computations while the actual communication is handled in the background. A dedicated function, namely MPI_Wait, ensures that a communication has been completed. In addition to blocking communication, values received without a matching call to a MPI_IRecv are buffered by MPI. Hence, the non-blocking send and receive functions in MPI are comparable to push() and pop() operations on FIFO queues. They can be used to implement DPNs in systems with distributed memory.

Particular implementations of MPI make use of shared-memory in case that they are executed in corresponding environments, e. g., pure SMP architectures or SMP clusters. This is achieved by creating a shared area in the memory, which is then used to communicate between the different processes on a single node. This clearly aims at the reduction of network utilization, which can improve the communication between processes on different computational units. The actual MPI implementation that was used in this thesis is OpenMPI.

To this end, MPI intends to define a common standard for computing in architectures with distributed memory. Partitioning of problems and assigning work to computational nodes has to be explicitly done by the programmer.

2.4 Parallelization in Hardware Design

Most processors execute sequential instruction streams. Historically, sequential execution is induced by early computer architecture, i. e., processors and their memory. Although offering random access, even modern memory requires several clock cycles to get a value from a memory address. Memories are optimized for reading consecutive cells, which can be read clock-wise after an initialization phase. The idea behind this concept is exploitation of the locality of data in applications, i. e., in general, there is a high probability to read a value whose address is close to the address of the value read before. Improvements usually focus on increasing bandwidth instead of decreasing latency, which is the key to get fast random access. Other approaches circumvent the latency issue by reading data ahead, which however does not solve the problem.

Although, the idea of parallel execution to increase performance is quite old [98, 171], all parallel computers consist of a connected couple of sequential machines. Hardware designers developed different techniques to improve performance and to keep compatibility to sequential execution at the same time. Some of these techniques have been considered in the context of this thesis, and they inspired some of the presented approaches. In

particular, this includes pipelining, OOO execution in dynamic processors and speculative execution. This section briefly introduces the principles of these techniques.

2.4.1 Pipelining

Pipelining is a simple principle for parallel computation that dates back to the late 1950s [101, 149]. It can be applied to all kinds of processes that repeatedly apply a sequence of actions $\alpha_1; \ldots; \alpha_p$ to an incoming stream of objects. Instead of processing single actions for each object one after the other, a pipelined system processes p objects $o_{t+p}, \ldots, o_{t+1}$ at every point of time t in parallel and applies thereby the actions $\alpha_1; \ldots; \alpha_p$ to these objects in parallel (i.e., action α_{i+1} is applied to object o_{t+p-i} at time t). Pipelining is generally used to increase the throughput of the system: Using p pipeline stages, a theoretical speed-up by a factor p can be obtained [103] without increasing the number of actors that perform the actions α_i. Essentially all microprocessors are nowadays implemented with pipelines to speed-up the processing of their instruction streams. However, pipelining is a much more general parallel processing technique and can therefore also be used to create pipelines of software threads. A very good survey on the architecture and analysis of pipelines can be found in [149].

2.4.2 Out-of-Order Execution

Out-of-order execution is well-known in the domain of computer architecture for a long time, originating in early work by Tomasulo [174]. Its basic idea is to execute a sequential instruction stream of a usual von-Neumann architecture in data-flow order, thereby establishing more parallelism and better load balancing of available functional units.

The arithmetic-logical unit (ALU) of a processor is one of its most important parts. It computes actual results of instructions and usually consists of different functions that are implemented in hardware, e. g., adder, multiplier, bitwise operations and so on. Especially in MIPS architectures, it is common to define an instructions set in a way, such that a single instruction uses only one of these functions provided by the ALU. The execution model as explained in Section 2.4.1 allows only one instruction to enter the ALU at a time, meaning that only one function of the ALU is used at a time, while all other functions remain idle. An intuitive approach for better utilization of these functional units in the ALU would be to schedule several instructions to the ALU at the same time, given that these instructions have no dependencies and use different functions of the ALU. This approach comes with several issues that have been solved by Tomasulo in a sophisticated manner.

The first task is to identify instructions that are independent. A dynamic analysis of the instruction stream allows the dynamic parallel scheduling of instructions. The instructions are loaded into the so-called *reservation station*. The instruction decoder concurrently loads several instructions into the reservation station. Each entry consists of the op-code, the target register, and the required input operands for the execution. The pending instruction remain in the reservation station until they are scheduled to the functional units. In addition, the reservation station keeps track of missing input operands,

i. e., whether an instruction needs a value which is computed by a preceding instruction that has not been finished, yet. This is achieved by adding a *tag field* to the register set which determines whether the contained value is up-to-date and otherwise the latest instruction that will overwrite that register. As soon as the result of an instruction is computed, the reservation station checks whether this value is required by any instruction, and in case, it assigns the value as input operand to the corresponding instruction. If all input operands of an instruction are available, the instruction is ready to be scheduled. When a functional unit is idle, the so-called *dispatching unit* searches for a matching instruction that is ready to be scheduled, e. g., if the ADD functional unit is idle, the dispatcher searches for an ADD instruction. On the contrary, the register set is not immediately updated when an instruction finishes. The ability of processors to be interrupted by external signals and the necessity of exception handling, e. g., a division by zero, requires special handling to preserve an external in-order-execution behavior. This is achieved by storing results of computations temporarily in the *reorder buffer*. Write-backs from the reorder buffer to the register set can then easily be done in the original ordering of instructions, which preserves an external visible in-order behavior. In-order write-backs of registers is also necessary for speculative execution, which will be explained in the next section.

In Tomasulo algorithm, tracking of dependencies and immediate broadcast of computations is a sophisticated mechanism to forward results. Moreover, it decouples the forwarding mechanism from the design of functional units. Functional units implement different functions with different complexities. Pipelining is often applied to these units, not only to increase clock frequency, but also to balance maximum clock frequencies between stages of different units. This ends up with functional units with different pipeline lengths, which does not require any special handling in Tomasulo's algorithm.

To this end, the scheduling of the instructions considers a window of instructions and dispatches them according to their data dependencies and available functional units, i. e., scheduling of instructions is driven by the data-flow instead of the original sequential ordering.

In contrast, a static analysis in a compiler with knowledge about the targeted architecture could group independent instructions to bundles as done in explicitly parallel instruction computing (EPIC), e. g., very large instruction word (VLIW) processor. In VLIW architectures, a bundle of instructions is loaded from a stream and scheduled in a single step to the processor's ALU. However, this requires a thorough preparation by the compiler, i. e., the bundled instructions must fit to the ALU architecture. Although static analysis relieves the processor from doing this analysis, its main drawback is that existing code requires re-compilation. Tomasulo targeted a better utilization of processors that are capable of executing existing programs, i. e., sequential instruction streams.

2.4.3 Speculative Execution

Historically, speculation in processors has its roots in pipelined processors [119]. Pipelined execution is state-of-the-art in the design of today's processors. Similar to a production line, the processing of an instruction is split into several steps. A pipelined CPU pro-

cesses several instructions in parallel; thereby each instruction is in a different stage. The efficiency of a pipelined processor strongly depends on how much its pipeline is utilized. Optimal utilization is usually given when each stage in a pipeline is busy. Conditional branches prevent the processor of continuously executing an instruction stream because the result of a branch instruction is usually determined in later stages of the pipeline. The processor is forced to stop reading the next instructions until the result of the branch instruction is known. To compensate this weakness, processors continue executing instructions from one branch by doing *speculation*, i. e., they make a guess of the result of the branch instruction. If the guess is correct, i. e., the speculatively executed instructions are then known to be correct, no time is lost due to stalls. The other side of the coin is that a false guess requires to remove all speculatively loaded instructions from the pipeline and to roll-back all changes that have been done by these instructions. To conclude, the speculation mechanism in hardware may speed up the execution and requires additional effort for its implementation. Related work in the context of this thesis is presented in Section 1.3.

Chapter 3

Creating Task-Based Parallelism from Synchronous Programs

This chapter presents the first approach of this thesis to concurrently execute SGAs. As explained in Section 2.1.2, a synchronous program is executed by computing iteratively its reactions to input stimuli from its environment, as depicted by the pseudo code in Figure 2.2. A schedule for a given set of guarded actions can be found by using the data dependencies between the actions. Building a sequence basically means to add further scheduling dependencies, i. e., over-specifying the ordering of a set of guarded actions. Hence, executing such a set of actions in data-flow order allows one to use available parallelism.

3.1 OpenMP

In [16], we present an approach to create multi-threaded C code from SGAs using the OpenMP API. This requires to transform an ADG into a structured task graph. The structured task graph is a representation of the control-flow that fits to the execution model of OpenMP (see Section 2.3.2.1). OpenMP allows recursive forking of an instruction stream to several sub-paths that can be executed in parallel, the so-called parallel region. To that end, two points are important: (1) Each fork must be closed by a join that merges exactly those paths that are created by that fork. (2) Each parallel region must be closed by a join, before its parent parallel region is closed. We will explain these points more detailed and formalize them later in this section.

The basic idea of the presented approach is to concurrently execute independent actions whenever it is possible. Therefore, a fork is inserted if actions are independent, and a join is inserted if an action depends on several actions. However, this is not a trivial task: OpenMP does not allow one to merge single paths from different parallel regions by a single join. Hence, the main challenge in this approach is to avoid interleaved dependencies in nested parallelism as shown in Figure 3.1. Figure 3.1 (a) shows an exemplary ADG with interleaved nested parallelism. The previously described basic idea suggests to place forks after nodes with multiple outgoing dependencies and joins before nodes with multiple

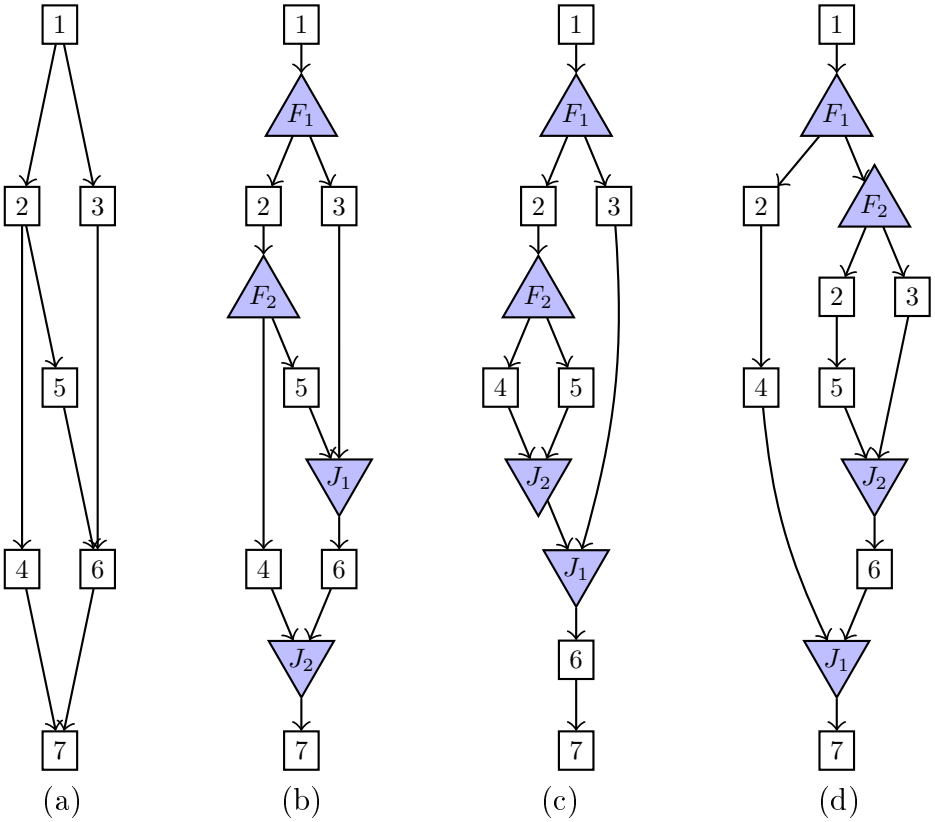


Figure 3.1: Mapping data-flow parallelism to structured parallelism : **(a)** shows a schematic ADG with interleaved dependencies, **(b)** forks (nodes with multiple outgoing edges) and joins (nodes with multiple incoming edges) are added. OpenMP does not support interleaved task groups. Interleaving can be forced by either **(c)** closing the second fork earlier, or **(d)** duplicating nodes, i. e., in this example node 2.

incoming dependencies. Hence, according to that description, forks and joins would be inserted as depicted in Figure 3.1 (b). In particular, the second fork (F_2) splits the control-flow into two paths. The dependency of node 6 to node 3 and 5 requires to close the parallel region that was created in the first place. OpenMP requires to close the parallel region opened by fork F_2 first. In particular, this means that all paths starting from F_2 have to be merged, which is not fulfilled in this graph. Figure 3.1 (c) shows a graph, where a join for the recursively created parallel region is inserted right before its parent parallel region is closed. The solution shown in Figure 3.1 (d) allows one to create longer paths at the expense of duplication of actions. Although node 4 has no dependency to node 6, it has to be joined with node 5 by join J_2. To separate node 4 from the inner parallel region node 2 can be duplicated, which results in an independent path starting at fork F_1.

Our approach uses the strategy to early close parallel regions. Referring to the example in Figure 3.1, we target to map an ADG from (a) to (c). Duplication of nodes as shown in (d) has been left because it introduces additional computational effort. Its benefit depends on the actual graph and the actual targeted hardware: A static analysis and rating of the benefits of a node duplication requires precise knowledge about the computational effort of a node which depends on the targeted hardware. This conflicts with our principle to make decisions about the actual target as late as possible.

3.1.1 From Data-Flow Graphs to Structured Graphs

In the following, we present an algorithm to transform an ADG to a structured graph. We assume that an ADG is given as described in Section 2.1.3. Our transformation to a structured graph from an ADG only adds nodes and edges, i. e., it makes no modifications to the existing nodes or edges of the ADG. Hence, the structured graph can be considered as an overlay which adds explicit scheduling information. The remaining part of this section is structured as follows: We start with a structural description and formal definition of the structured graph. A list of restrictions will define a *valid structured graph*, which gives a more detailed insight of the requirements for the transformation. Finally, the pseudo code of our transformation gives a formal description of the algorithm and its understanding is deepened by means of a set of examples.

3.1.1.1 Structured Graph

A structured graph represents a static schedule of a given ADG. It makes use of existing parallelism in the ADG by including synchronization points to explicitly split and merge the control-flow. In addition to the existing definition of ADGs (see Definition 4), a structured graph can have fork and join nodes and an additional type of dependencies to indicate scheduling dependencies. Scheduling dependencies define the control-flow in an ADG, which allows one to do a straight-forward mapping from the structured graph to OpenMP code. We begin with a definition of additional node types: Fork and join nodes.

Definition 9 (Fork and Join Nodes)**.** *"Fork" and "Join" denote the type of nodes that are used to fork and join, respectively, the control-flow in a structured graph. We furthermore define the operator $n = t$, with n is a node and $t \in \{Fork, Join\}$:*

- $n = Fork \leftrightarrow n$ *is a fork*
- $n = Join \leftrightarrow n$ *is a join*
- $\neg(n = Fork) \wedge \neg(n = Join) \leftrightarrow n$ *is an SGA*

The new dependency type for building structured graphs is defined as follows:

Definition 10 (Scheduling Dependency Type)**.** *"sched" denotes a scheduling dependency which explicitly forces the control-flow to follow a specific path in a graph. The set of dependency types for structured graphs is given as $D' = \{immediate, delayed, sched\}$.*

The additional node and edge types allow one to define the structured graph:

Definition 11 (Structured Graph of an ADG)**.** *Let $G = (V_G, E_G)$ be an ADG according to Definition 4. The structured graph is a graph $SG = (V, E)$ consisting of vertices $V = \{v \mid v \in V_G \vee v = Fork \vee v = Join\}$, representing the guarded actions, forks and joins, and directed edges $E \subseteq D' \times V \times V$ representing the dependencies between the actions, where $D' = \{immediate, delayed, sched\}$ defines the set of all dependencies that are valid in a structured graph.*

Note that V' denotes the set of nodes which contains all SGAs from the original ADG and optionally additional fork and join nodes, i. e., no SGAs are added, removed or modified. Definition 11 is only a structural definition of a structured graph. In the following, we will define a *valid structured graph* which is a structured graph that allows one to do the targeted straightforward synthesis. This requires to impose restrictions to the structural definition of the structured graph. Figure 3.2 prints functions to formally impose these restrictions in Definition 12, which defines a valid structured graph. $numPre(n)$ and $numSuc(n)$ (see lines 1f in Figure 3.2) denote the functions to determine the number of predecessors and successors, respectively, of node n. Function hasSchedulingPath determines whether a node n_2 is directly or indirectly scheduling dependent on another node n_1. Node n_2 is directly scheduling dependent on node n_1 if they are connected by a scheduling dependency (see line 4 in Figure 3.2). An indirect scheduling dependency from n_1 to n_2 is given if there exists a sequence of nodes (line 5) such that node n_1 has a scheduling dependency to the first node of the sequence (line 6), each adjacent pair of nodes in the sequence has a direct scheduling dependency (line 7) and the last node of the sequence has a scheduling dependency to n_2 (line 8). Finally, matchFJ denotes the function that determines whether the given fork node f and join node j are a matching pair. This is the case if the number of scheduling paths leaving fork node f and leading to join node j are equal (line 13) and if all outgoing scheduling paths of fork node f are closed by the join node j (line 14) and if all incoming scheduling paths to join node j are opened by fork node f (line 15). To this end, we can define a valid structured graph:

1: **function** $\mathsf{numPre}(n) = |\{v \mid v \in V \wedge \exists(\text{sched}, v, n) \in E\}|$
2: **function** $\mathsf{numSuc}(n) = |\{v \mid v \in V \wedge \exists(\text{sched}, n, v) \in E\}|$
3: **function** $\mathsf{hasSchedulingPath}(n_1, n_2)$ {
4: **return** $(\text{sched}, n_1, n_2) \in E \vee$
5: $(\exists v_1, \ldots, v_i \in V, i \geq 1.$
6: $(\text{sched}, n_1, v_1) \in E \wedge$
7: $(\forall k = 1, \ldots, i-1.(\text{sched}, v_k, v_{k+1}) \in E) \wedge$
8: $(\text{sched}, v_i, n_2) \in E)$
9: }
10: **function** $\mathsf{matchFJ}(f, j)$ {
11: $suc_f = \{v \mid v \in V \wedge \exists(\text{sched}, f, v) \in E\}$
12: $pre_j = \{v \mid v \in V \wedge \exists(\text{sched}, v, j) \in E\}$
13: **return** $|suc_f| = |pre_j| \wedge$
14: $\forall s \in suc_f.|\{p \mid p \in pre_j \wedge \mathsf{hasSchedulingPath}(s, p)\}| = 1 \wedge$
15: $\forall p \in pre_j.|\{p \mid s \in suc_f \wedge \mathsf{hasSchedulingPath}(s, p)\}| = 1$
16: }

Figure 3.2: Pseudo code for several functions to impose restrictions required by Definition 12. Assume $SG = (V, E)$ is given, then numPre and numSuc return the number of predecessors and successors, respectively. $\mathsf{hasSchedulingPath}(n_1, n_2) \leftrightarrow$ if n_1 has a direct or indirect scheduling dependency to n_2. $\mathsf{matchFJ}(f, j)$ returns true iff fork f and join j are a matching pair.

Definition 12 (Valid Structured Graph of ADG)**.** *Let $SG = (V, E)$ be the structured graph of an ADG. SG is a valid structured graph if it fulfills all of the following requirements:*

1. *A SGA in a valid structured graph must have at most one predecessor and at most one successor. Formally:*
   $(n \in V \wedge \neg(n = \mathit{Fork}) \wedge \neg(n = \mathit{Join})) \leftrightarrow (\mathsf{numPre}(n) \leq 1 \wedge \mathsf{numSuc}(n) \leq 1)$.
   The number of predecessors or successors can be 0 in case that node n is the very first node or very last node, respectively, that has to be scheduled.

2. *A fork must have at most one predecessor and at least two successors. Formally:*
   $(n \in V \wedge n = \mathit{Fork}) \leftrightarrow (\mathsf{numPre}(n) \leq 1 \wedge \mathsf{numSuc}(n) > 1)$.

3. *A join must have at least two predecessors and at most one successor. Formally:*
   $(n \in V \wedge n = \mathit{Join}) \leftrightarrow (\mathsf{numPre}(n) > 1 \wedge \mathsf{numSuc}(n) \leq 1)$.

4. *A graph must have a uniquely determined root, i. e., the control-flow starts at exactly one node. Formally:*
   $|\{n \mid n \in V \wedge \mathsf{numPre}(n) = 0\}| = 1$.
   Due to the requirement of a join node to have two or more predecessors, the root node must be an SGA or a fork node.

5. *A graph must have a uniquely determined last node, i. e., the control-flow stops at exactly one node. Formally:*
   $|\{n \mid n \in V \wedge \mathsf{numSuc}(n) = 0\}| = 1$.
   Due to the requirement of a fork node to have two or more successors, the last node must be an SGA or a join node.

6. *If SG has an immediate dependency between two SGAs n_1 and n_2, then there must be a scheduling path from n_1 to n_2 . Formally:*
   $(immediate, n_1, n_2) \in E \rightarrow \mathsf{hasSchedulingPath}(n_1, n_2)$.

7. *Each fork node must have exactly one matching join node, i. e., all paths opened by a fork must be merged by a single join. Conversely, all paths that are merged by a join must be opened by a single fork node. Formally:*
   $\forall f \in \{n \mid n \in V \wedge n = Fork\}.|\{j \mid j \in V \wedge j = Join \wedge \mathsf{matchFJ}(f, j)\}| = 1 \wedge$
   $\forall j \in \{n \mid n \in V \wedge n = Join\}.|\{f \mid f \in V \wedge f = Fork \wedge \mathsf{matchFJ}(f, j)\}| = 1$

8. *Parallelism can be nested but must not be interleaved. Let f_1 be a fork and j_1 be the matching join of f_1. If a fork f_2 is scheduled after f_1 and before j_1, then the matching join must be also scheduled before j_1. Formally:*
   $(\mathsf{matchFJ}(f_1, j_1) \wedge \mathsf{matchFJ}(f_2, j_2) \wedge$
   $\mathsf{hasSchedulingPath}(f_1, f_2) \wedge \mathsf{hasSchedulingPath}(f_2, j_1)) \rightarrow$
   $\mathsf{hasSchedulingPath}(j_2, j_1)$.
   Note that this restriction is a consequence of restriction 1, 2, 3 and 7.

Function matchFJ in addition to the restrictions of Definition 12 allows one to give a formal definition of parallel regions, which was only verbally explained until now:

Definition 13 (Parallel Region, Child Parallel Region and Parent Parallel Region)**.** *A parallel region P describes a set of paths created by a fork node f and merged by its matching join node j. Formally:*

$$\mathsf{matchFJ}(f, j) \rightarrow P(f, j) = \{f, j\} \cup \{v \mid v \in V \wedge \mathsf{hasSchedulingPath}(f, v) \wedge \mathsf{hasSchedulingPath}(v, j)\}$$

If $P(f_2, j_2) \subset P(f_1, j_1)$ holds for two parallel regions $P(f_1, j_1)$ and $P(f_2, j_2)$, then parallel region $P(f_2, j_2)$ is a child parallel region of $P(f_1, j_1)$, and $P(f_1, j_1)$ is called parent parallel region of $P(f_2, j_2)$

Restrictions 2 and 3 in Definition 12 require a parallel region to consist of two or more paths. Restrictions 7 and 8 require a child parallel region to be part of only one of these paths. In particular, paths created by a fork of a parallel region can only be merged by its matching join but not by a join of a child parallel region: Actions and forks allow only one incoming scheduling dependency (see restrictions 1 and 2) and the join node of a child parallel region merges only paths of its matching fork (see restrictions 7 and 8). For that reason, a child parallel region must be a proper subset of its parent parallel region.

In the following, the expression *structured graph* will always refer to a *valid structured graph*, i. e., we always assume validity. Moreover, an *open parallel region* or *open fork*,

respectively, refers to the state of a fork during the structuring algorithm, where a matching join has not yet been inserted. In particular, a fork represents the beginning and the matching join marks the end of a parallel region. If a matching join has not been scheduled for a fork then the structuring algorithm can still schedule nodes after that fork. In contrast, a closed parallel region does not allow one to schedule any further nodes.

3.1.1.2 Pseudo Code

Definition 12 is a restriction to structured graphs derived from ADGs but still permits different valid structured graphs for a single ADG (see introductory example in Figure 3.1). The pseudo code of our structuring algorithm is split into several parts. Our method is to traverse the given ADG in data-flow order and to iteratively add scheduling information. During these iterations, we add fork and join nodes at corresponding points, i. e., a fork node is inserted if the control-flow can be split to enable parallel execution and a join node is inserted as a barrier to synchronize SGAs.

Figure 3.3 contains the main loop (see function structureGraph), which iteratively selects nodes and initiates their scheduling. Figure 3.4 contains the functions to schedule actions and to insert fork nodes which are called by the function structureGraph. The function to insert and schedule join nodes is printed in Figure 3.5. Additional functions to schedule joins are required. They are printed in Figure 3.5 and 3.6. Some of the functions make use of functions shown in Figure 3.2.

We assume that the graph $SG = (V, E)$ is globally defined and already initialized with $SG = G$, i. e., the actions and edges from the original ADG have been copied to the structured graph. The traversal of the ADG requires to keep track of already scheduled nodes. Therefore, we define the set V_{sched} (see line 1 in Figure 3.3). Conversely, the set of unscheduled nodes can be defined as given in line 2. Function lastNodes(n) (lines 3-9) determines the set of nodes that has been scheduled last after the node n, i. e., all nodes that have a scheduling path (see function hasSchedulingPath) from node n and that do not have a scheduled succeeding node.

Function disjRel($relations$) (see line 10-16 in Figure 3.3) merges mappings with non-disjunctive values. In particular, the function gets a list of tuples ($relations$). Each tuple represents a pair of sets, and all tuples have to be of the same type. The basic idea of disjRel is to join two tuples in $relations$ when the secondary sets of these tuples are non-disjunctive. Obviously, this may apply to more than two tuples. For instance, let $(a_1, b_1), (a_2, b_2), (a_3, b_3) \in relations$ with $b_1 \cap b_2 \neq \{\}$ and $b_2 \cap b_3 \neq \{\}$, i. e., the secondary sets of the first and second tuple and the secondary sets of the second and third tuple are non-disjunctive. For that reason, all tuples have to be merged. A set of tuples is merged by merging all primary sets and secondary sets, respectively. For the given example the result is $(a_1 \cup a_2 \cup a_3, b_1 \cup b_2 \cup b_3)$.

Function structureGraph (see line 17ff in Figure 3.3) is the actual function to add scheduling information to get a structured graph. It iteratively adds scheduling dependencies by traversing the ADG in data-flow order (see line 18ff). First step in a single iteration is to determine the set of nodes that is ready to be scheduled (line 19). Note that the set of

```
1: V_sched = {}
2: function V_unsched = V \ V_sched
3: function lastNodes(n) {
4:   suc = {v | v ∈ V_sched ∧ (sched, n, v) ∈ E}
5:   if (suc = {}) then
6:     return {n}
7:   else
8:     return ∪_{s∈suc} lastNodes(s)
9: }
10: function disjRel(relations) {
11:   dr := (∀(a, b) ∈ relations. (∃(A, B) ∈ dr.(a ⊆ A ∧ b ⊆ B)))∧
12:         (∀(a_1, b_1), (a_2, b_2) ∈ relations.((b_1 ∩ b_2 ≠ {}) ∨ (b_1 ∪ b_2 = {}) →
13:           ∃(A, B) ∈ dr. ((a_1 ∪ a_2) ⊆ A ∧ (b_1 ∪ b_2) ⊆ B)))∧
14:         (∀(a_1, b_1), (a_2, b_2) ∈ dr.(a_1, b_1) ≠ (a_2, b_2) → b_1 ∩ b_2 = {})
15:   return dr
16: }
17: function structureGraph() {
18:   while (V_unsched ≠ {}) {
19:     V_schedulable = {v | v ∈ V_unsched∧ ∄v' ∈ V_unsched.(immediate, v', v) ∈ E}
20:     dep = {({v}, S_3) | v ∈ V_schedulable∧
21:                         S_1 = {s | s ∈ V_sched ∧ (immediate, s, v) ∈ E}∧
22:                         S_2 = {s | s ∈ S_1∧ ∄s' ∈ S_1.hasSchedulingPath(s', s)}∧
23:                         S_3 = ⋃_{s∈S_2} lastNodes(s)}
24:     dr = disjRel(dep)
25:     if (¬scheduleForks(dr)) then
26:       if (¬scheduleActions(dr)) then
27:         scheduleJoins(dr)
28:   }
29:   scheduleFinalJoin()
30: }
```

Figure 3.3: Pseudo code for structuring a DFG for execution using OpenMP (Part 1/4): **function** StructureGraph is the main function, further functions are explained in Figures 3.4 and 3.5 and 3.6

nodes that will be actually scheduled may only be a subset of the schedulable nodes. At that point of time, it is not clear which nodes of this set are going to be actually scheduled. The function must now determine from which nodes a scheduling dependency will be added in case that a schedulable node is actually scheduled.

```
31: function scheduleForks(dr) {
32:   dr' = {(V_g, D_g) | (V_g, D_g) ∈ dr ∧ |V_g| > 1 ∧ |D_g| ≤ 1}
33:   forall (V_g, D_g) ∈ dr' {
34:     f = CreateForkNode()
35:     V = V ∪ {f}
36:     E = E ∪ (⋃_{d∈D_g}(sched, d, f)) ∪ (⋃_{v∈V_g}(sched, f, v))
37:     scheduled = scheduled ∪ {f} ∪ V_g
38:   }
39:   return (dr' ≠ {})
40: }
41: function scheduleActions(dr) {
42:   dr' = {(v, D_g) | (V_g, D_g) ∈ dr ∧ V_g = {v} ∧ |D_g| ≤ 1}
43:   forall (v, D_g) ∈ dr' {
44:     E = E ∪ (⋃_{d∈D_g}(sched, d, v))
45:     scheduled = scheduled ∪ {v}
46:   }
47:   return (dr' ≠ {})
48: }
```

Figure 3.4: Pseudo code for structuring a DFG for execution using OpenMP (Part 2/4): Functions to schedule actions and to insert forks.

A dependency mapping *dep* is created which contains a tuple for each node that is ready for scheduling (lines 20ff). Each tuple consists of the schedulable node v and the set of nodes from which a scheduling dependency would be added with respect to the actual state of the structuring graph (line 21-23). The latter set is determined in three steps: (1) Set S_1 contains all preceding nodes of v with an immediate dependency (line 21). (2) Set S_2 neglects all nodes in S_1 that already have a scheduling path to another node in S_1 (line 22). This refers to implicit scheduling (see requirement 6 of Definition 12): A node n_2 that has to be scheduled after another node n_1, will implicitly be scheduled after all other nodes that are scheduled before n_1. This allows one in some cases to reduce synchronization overhead, e. g., in Example 4. Finally, (3) Other nodes might have been scheduled after the nodes in set S_2. To ensure the requirements 1, 2 and 3 in Definition 12, we compute S_3 which contains the last scheduled nodes after the nodes in S_2 (line 23). This determines the actual nodes from which the dependencies have to be created.

Application of disjRel to *dep* merges schedulable nodes with non-disjunctive sets of potential preceding nodes (line 24). This is essential to recognize parallelism and necessity for synchronization, i. e., whether a fork node or a join node, respectively, has to be inserted. As a result, *dr* will contain three different types of tuples. Thereby, the size of the sets in a tuple are the determining factor: (1) A 1-1-relation denotes a sequential scheduling dependency, (2) a 1-n-relation allows one to fork the control-flow into several paths, and

```
49: function getOpenParentForks(node) =
50:    {f | f ∈ V ∧ f = Fork ∧ node ∈ lastNodes(f)∧
51:            ∄j ∈ V.(j = Join ∧ matchFJ(f, j))}
52: function commonForks(V_g) =
53:    ⋂_{v∈V_g} getOpenParentForks(v)
54: function latestOpenCommonFork(V_g) = f_c with
55:    {f_c} := commonForks(V_g) \ ⋃_{f∈commonForks(V_g)} getOpenParentForks(f)
56: function openNonCommonForks(V_g) =
57:    (⋃_{v∈V_g} getOpenParentForks(v)) \ commonForks(V_g)
58: function scheduleJoins(dr) {
59:    dr' = {D_g | (V_g, D_g) ∈ dr ∧ |D_g| > 1}
60:    fc_÷ = {(D_g, openNonCommonForks(D_g)∪
61:              {latestOpenCommonFork(D_g)}) | D_g ∈ dr'}
62:    fc_⊙ = disjRel(fc_÷)
63:    forall (D_g, fc) ∈ fc_⊙
64:       closeForkPartially(D_g)
65:    return (dr' ≠ {})
66: }
```

Figure 3.5: Pseudo code for structuring a DFG for execution using OpenMP (Part 3/4): Functions to insert joins, i. e., close parallel regions opened by forks.

(3) a n-l-relation requires to synchronize several paths. Each type of tuple is assigned to a function that is dedicated to schedule a specific type of node.

We use the functions scheduleForks, scheduleActions and scheduleJoins to schedule fork, action and join nodes, respectively (line 25-27). Each function gets the complete list of relations and is responsible for itself to select relevant tuples. The return value of the scheduling function tags whether one or more nodes have been scheduled by the scheduling function. In case that one or more nodes were scheduled, the set dr may potentially change and has to be determined again. To this end, only one function is permitted to schedule nodes per iteration, which requires the nested if-then-statement. Note that the ordering of the calls in line 25-27 in Figure 3.3 to schedule nodes is not fixed. We choose the given ordering because our goal is to create parallelism as early as possible and to keep paths open as long as possible, i. e., we prefer to schedule fork nodes before actions and actions before join nodes. After all actions in a graph have been scheduled, some parallel regions might still be open. They are closed by using the function scheduleFinalJoin (line 29).

The code for the functions to schedule fork nodes and actions is printed in Figure 3.4. dr is the set of tuples describing the relation of schedulable nodes to nodes from which a scheduling dependency has to be added. A fork is added for each tuple where two or more nodes depend either on no node or on the same (single) node (line 32), which is necessary to fulfill requirement 2 of Definition 12. The former case is a special case

```
67: function closeForkPartially(D_g) {
68:   f = latestOpenCommonFork(D_g)
69:   suc_all = {n | (sched, f, n) ∈ E}
70:   suc_close = {n | n ∈ suc_all ∧ ∃d ∈ D_g.hasSchedulingPath(n, d)}
71:   if (suc_all = suc_close) then
72:     closeFork(f)
73:   else
74:     f_split = CreateForkNode()
75:     V = V ∪ {f_split}
76:     E = E \ {(sched, f, n) | n ∈ suc_close}
77:     E = E ∪ {(sched, f, f_split)} ∪ {(sched, f_split, n) | n ∈ suc_close}
78:     scheduled = scheduled ∪ {f_split}
79:     closeFork(f_split)
80: }
81: function getNextChildForks(n) {
82:   f_allChilds = {f | f ∈ V_sched ∧ f = Fork∧
83:                 ∄j ∈ V.j = Join ∧ matchFJ(f, j)∧
84:                 hasSchedulingPath(n, f)}
85:   return {f | f ∈ f_allChilds∧ ∄f' ∈ f_allChilds.hasSchedulingPath(f', f)}
86: }
87: function closeFork(f) {
88:   cf = getNextChildForks(f)
89:   forall f' ∈ cf
90:     closeFork(f')
91:   L = lastNodes(f)
92:   j = CreateJoinNode(f)
93:   V = V ∪ {j}
94:   E = E ∪ {(sched, l, j) | l ∈ L}
95:   scheduled = scheduled ∪ {f}
96: }
97: function scheduleFinalJoin() {
98:   L = {n | ∀v ∈ V.(sched, n, v) ∉ E}
99:   scheduleJoins({{}, L})
100: }
```

Figure 3.6: Pseudo code for structuring a DFG for execution using OpenMP (Part 4/4): Functions to insert joins, i. e., close parallel regions opened by forks.

that occurs in the very first scheduling iteration: If parallelism is available in the very first iteration, the corresponding nodes will not depend on any preceding nodes. Hence, the control-flow has to start with a fork. For each tuple (line 33) a new fork is created (lines 34f). Scheduling dependencies are added from the possibly given single node in D_g

to the newly created fork, which splits the control-flow to the schedulable nodes in V_g (line 36). The newly created fork and the nodes in V_G belong now to the scheduled nodes (line 37). Analogously, scheduleActions selects all tuples in dr with a 1-1-relation (line 42), i. e., when one schedulable node depends on no node or exactly on one (single) node (see requirement 1 of Definition 12. Similar to the scheduling of forks, the latter case is a special case that occurs in the very first iteration. If only one action can be scheduled in the very first iteration, D_g will have no elements. To this end, the scheduling of SGAs requires no additional nodes. Only a scheduling dependency from the possibly given single node d is added (line 44) and the schedulable node v is added to the set of scheduled nodes (line 45).

The function to schedule join nodes is given in Figure 3.5, lines 58ff. The idea of scheduling a join is similar to closing a fork, which requires several helper functions. The information for determining a fork to close is given by a set of nodes which describe the paths that have to be closed. To properly close a fork, we must determine all open parent forks of each path that has to be closed. An open parent fork is a fork that has no matching join and that has a scheduling path to node *node* (line 49-51). The intersection of all open parent forks of a set V_g determines the set of common open forks (lines 52f). The latest common open fork determines the fork that has a path to all nodes in V_g but no path to any other common open fork (lines 54f). This means that latestOpenCommonFork defines the fork that has to be closed to merge the control-flow of a set of nodes given as V_g. As a consequence, non-common open forks (lines 56f) of a set of nodes V_g must be closed, too.

Similar to the other scheduling functions, scheduleJoins selects all tuples from dr that are of interest, i. e., those tuples with a n-l-relation (see requirement 3 of Definition 12). Function scheduleJoins only adds join nodes to the structured graph, i. e., no SGAs are scheduled. Hence, when the relevant information from dr is filtered, it is sufficient to store the sets of paths that have to be closed. As a consequence, dr' contains only the last nodes of the paths that have to be merged (line 59). The next step is to create a mapping from these nodes to the set of forks that have to be closed (lines 60f). The basic idea to determine the set of forks is to consider requirement 4 of Definition 12, i. e., all paths that have to be closed must have a uniquely determined root node. All paths will follow a common control-flow and split at the fork which is defined by the function latestOpenCommonFork. This fork defines the fork that has to be closed. Moreover, requirement 8 of Definition 12 involves that all child parallel regions of that fork have to be closed, too. These parallel regions are opened by the set of non-common forks. Thus, $fc_{\div}$ results in the set of open non-common forks in addition to the latest common fork (lines 60f). Although the sets in dr' are disjunctive, the forks that have to be closed for different sets may be non-disjunctive. Application of disjRel merges tuples that have non-disjunctive lists of forks (line 62). Each of the resulting tuples is again a mapping of a list of nodes D_g to a set of forks. The latter is a side product of disjRel, which is not required anymore. The essential information is the new set enumerating the nodes that mark the paths that have to be merged. This set is given to the function closeForkPartially, which is explained in the following.

Figure 3.6 depicts the functions that actually add join nodes to close fork nodes. In the OpenMP execution model, fork and join nodes represent synchronization points. While fork nodes create parallelism, join nodes technically represent barriers to synchronize several

paths. Hence, to keep the synchronization overhead low, only nodes that *have to* be closed should be closed. Paths that do not have to be merged but that are created by a fork that has to be closed, should be kept open if possible. Therefore, we have two different implementations of functions to close a fork partially and completely, respectively.

Function closeForkPartially closes only specific paths of a fork. Requirement 7 of Definition 12 prohibits to partially join paths of a fork. A way to achieve partial merging is to add a sub-fork behind the actual fork and before the succeeding nodes that belong to the paths that have to be closed. First, the function determines the latest open common fork (line 68). This is the only fork for a given set D_g which can be partially closed. All non-common forks have to be closed completely. Next step is to get all directly succeeding nodes of the fork (line 69) and to determine the set of successors that start a path that has to be closed (line 70). If both sets are equal then all paths have to be closed (line 72), which is done by function closeFork. In case that $suc_{all} \supset suc_{close}$, only a part of the paths have to be closed. To leave other paths of f open, we insert a new fork (lines 74f) and redirect scheduling dependencies to the paths that have to be closed from f to f_{split} (lines 76-78). The newly created fork is added to the set of scheduled nodes (line 79) and is a sub-fork of f. All paths that have to be closed start at the newly created fork f_{split}, while all paths that should be kept open still start at the old fork f. Hence, by closing the newly created fork, we close only those paths that *have to* be closed (line 80).

As already mentioned, our structuring algorithm uses the early-join strategy, i. e., when a parallel region is closed, all of its child parallel regions are closed, too. Function getChildForks is a helper function (lines 81ff) to determine the open parallel regions in a path that start after node n. In particular, the result is computed by (1) determining the complete set containing all child forks of n, i. e., all forks that are scheduled after the given node (line 82-84), and (2) removing all forks that are scheduled after a node in $f_{allChilds}$, i. e., after another fork that belongs to the complete set of child forks (line 85). Thus, the function returns only the next level of nested regions, i. e., not the nested regions in already nested regions of the path starting at node n. Function closeFork (lines 87ff) closes the given fork f completely. In contrast to the function closeForkPartially, it does not consider sub-forking. First, application of closeFork to each nested region (lines 89f) recursively closes all nested regions to fulfill requirement 8 of Definition 12. Afterwards, the last nodes that have been scheduled after fork f are determined (line 91) to connect them to the newly created join (lines 92ff).

A call to scheduleFinalJoin is necessary to close potential open forks after all actions have been scheduled. This is accomplished by determining the set of nodes that have no outgoing scheduling dependency. This set is given to scheduleJoins, which will add a join for the given set of nodes and joins for potential child parallel regions. In case that the set consists only of a single node, it will be neglected by scheduleJoins, which adds joins only for tuples (V_g, D_g) with $|D_g| > 1$. To this end, a call to scheduleFinalJoin ensures that requirements 5 and 7 of Definition 12 are fulfilled.

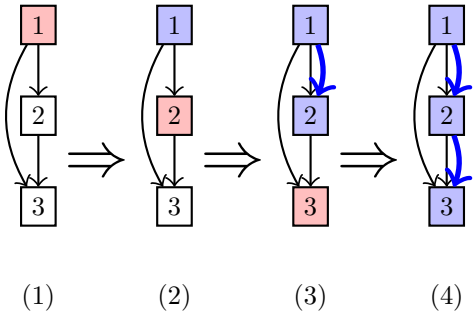


Figure 3.7: *Example 1:* ADG, structuring steps and the final structured graph.

3.1.1.3 Examples

This section deepens the understanding of Section 3.1.1.2, which presented the pseudo code of our structuring algorithm. We will consider a set of examples that demonstrate different cases that can appear in the structuring procedure. Each example starts from the ADG presentation and will apply the structuring algorithm to the given ADG. Line numbers will thereby refer to the pseudo code given in Figure 3.3, 3.4, 3.5 and 3.6. The examples show several steps, particularly one iteration of structureGraph per step until the structuring is completed. All exemplary ADGs have only immediate dependencies. Delayed dependencies are not considered in the structuring algorithm and have been neglected for better visualization. Generally, the direction of edges show the flow order, i. e., $A \rightarrow B$ means that B depends on a value of A. Transparent nodes represent unscheduled actions, red colored nodes represent schedulable nodes and a blue fill color denotes that the corresponding node has been scheduled. Finally, blue edges show scheduling dependencies and their direction defines the control-flow direction.

Examples 1-4 are recommended to get familiar with scheduling of actions and forks. Scheduling of join nodes is obviously mandatory but quite trivial for these examples. More subtle cases of scheduling joins, e. g., of nested parallelism, are given in the remaining examples.

Example 1 Figure 3.7 shows the structured graph that was initialized with an ADG as follows:

$SG_1 = (\{1, 2, 3\}, \{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 2, 3)\})$

The first example shows an ADG that provides no parallelism, i. e., all nodes have to be sequentially scheduled. A step-by-step execution of structureGraph(SG_1) results in the following behavior:

- *Iteration 1:* Node 1 can be scheduled, i. e., $V_{schedulable} = \{1\}$ (line 19). The node has no predecessors, i. e., $dep = \{(\{1\}, \{\})\}$ (line 20). Function disjRel results in $dr = dep$ because dep consists only of one tuple (line 24). The tuple in dr has a 1-0-relation, i. e., one node depends on no other node. A call to function scheduleForks returns false because it will only schedule n-1-relations and n-0-relations, respectively (line 32).

Hence, scheduleActions is called next (line 26), which schedules the first node. No scheduling dependencies are added because there are no predecessors to the scheduled actions (line 44). The action itself is added to the set of scheduled nodes (line 45) and will appear in blue color in the following iterations.

- *Iteration 2:* Only node 2 is schedulable in the second iteration: It has a dependency to node 1, which already has been scheduled, i. e., $V_{schedulable} = \{2\}$ (line 19). The computation of *dep* is still quite trivial: Node 2 has an immediate dependency from node 1, i. e., $S_1 = \{1\}$ (line 21). The list contains only one element, therefore $S_2 = S_1$. Because node 1 is already the last scheduled node in the corresponding path, S_3 turns out to be $S_2 = S_1 = \{1\}$. Similar to the first iteration, no fork is added because the list contains no potential parallelism. As a consequence, scheduleActions is called next (line 26), which schedules the node 2. Node 1 is the set of predecessors and a scheduling dependency is added from node 1 to node 2 (line 44). Finally, node 2 is added to the set of scheduled nodes (line 45).

- *Iteration 3:* Node 3 remains after the second iteration and is now listed in the set of schedulable node (line 19). The dependency mapping *dep* results in $\{(\{3\}, \{2\})\}$ (lines 20ff), which is computed as follows: S_1 lists the source node of the incoming immediate dependencies, i. e., $S_1 = \{1, 2\}$ (line 21). The nodes in S_1 that have a scheduling dependency to another node in S_1 are neglected in S_2, i. e., $S_2 = \{2\}$ (line 22). This is explained as follows: Node 2 is scheduled after node 1, and therefore, every node that is scheduled after node 2 is implicitly scheduled after node 1. Hence, node 1 does not have to be listed anymore. The advantage of filtering this node is not given in this example, but later in Example 4. S_3 turns out to be $\{2\}$ because node 2 is already the last scheduled node in the corresponding path (line 23). As a result, $dr = \{(\{3\}, \{2\})\}$ denotes the targeted scheduling dependency from node 2 to node 3 (line 24). The actual scheduling of node 3 proceeds similar to the scheduling of node 2: Function scheduleForks returns false, since *dr* contains only 1-1-relations. Function scheduleActions adds a scheduling dependency from node 2 to node 3. Furthermore, it adds node 3 to the set of scheduled nodes.

- *Final Join:* All nodes are scheduled now, i. e., $V_{unsched} = \{\}$, and the iterative scheduling stops now. Function scheduleFinalJoin will have no effect, because there is a uniquely determined node without outgoing scheduling dependencies as postulated by requirement 5 of Definition 12.

To this end, the resulting structured graph is given as

$$SG_1 = (\{1, 2, 3\}, \\ \{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 2, 3), (\text{sched}, 1, 2), (\text{sched}, 2, 3)\})$$

Example 2 Figure 3.8 shows the structured graph that was initialized with an ADG as follows:

$$SG_2 = (\{1, 2\}, \{\})$$

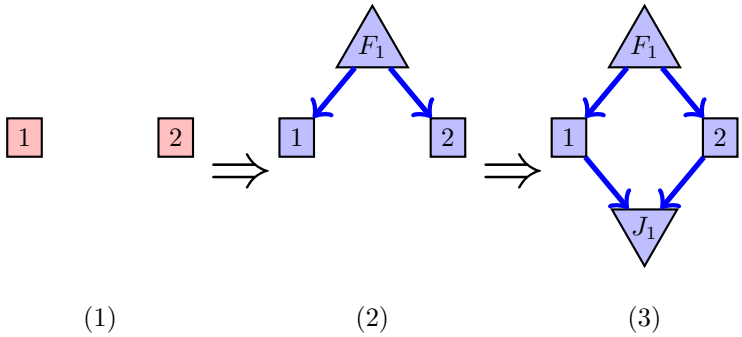


Figure 3.8: *Example 2:* ADG, structuring steps and the final structured graph.

This ADG shows two nodes that are completely independent of each other. It is useful to demonstrate the special case of scheduling a fork in the very first iteration, scheduling of concurrent nodes and scheduling of a final join after all nodes have been scheduled. A step-by-step execution of structureGraph(SG_2) results in the following behavior:

- *Iteration 1:* Node 1 and node 2 have no incoming dependencies and can be selected for scheduling, i. e., $V_{schedulable} = \{1, 2\}$ (line 19). The computation of the dependency mapping (line 20) yields $dep = \{(\{1\}, \{\}), (\{2\}, \{\})\}$. Function disjRel computes the non-disjunctive mapping with respect to the dependencies. Thereby, empty lists are considered to be non-disjunctive (line 12). As a result, the function returns $dr = \{(\{1, 2\}, \{\})\}$ (line 24). The single tuple in dr has a n-0-relation and is handled by scheduleForks (lines 32ff). A new fork F_1 is created (line 34) and connected with scheduling dependencies to the schedulable nodes (line 36). There is no need to add an incoming scheduling dependency to the fork, because no predecessor is available. The newly created fork and the scheduled nodes are added to the set of scheduled nodes (line 37).

- *Final Join:* All nodes are scheduled now, i. e., $V_{unsched} = \{\}$, and the iterative scheduling stops now. A call to function scheduleFinalJoin computes the set of nodes without scheduling dependencies, which results in $L = \{1, 2\}$ (line 98). Note that it is generally not sufficient to add a single node that merges the paths of the given set of nodes. Multiple parallel regions might still be open, which requires to add a join for each region. For that reason, the regular function to schedule joins is used to close all open parallel regions.

  The next step is to call function scheduleJoins (line 99), which starts with the computation of dr'. This is the filtered list of dr which contains only those elements with two or more predecessors, i. e., in this case $dr' = \{\{1, 2\}\}$. This list is mapped to a list of tuples $fc_{\div}$, where each tuple consists of the origin list of nodes that have to be merged ($D_g = \{1, 2\}$) and a list of forks that have to be closed to merge the paths ending at the nodes in D_g. The latter is computed as the union of openNonCommonForks($\{1, 2\}$) and lastOpenCommonFork($\{1, 2\}$) (lines 60f). Both functions need to compute the result of getOpenParentForks(1) = $\{F_1\}$, getOpenParentForks(2) = $\{F_1\}$ (line 49-51)

and $commonForks(\{1,2\}) = \{F_1\} \cap \{F_1\} = F_1$ (lines 52f). Further steps result in a call to openNonCommonForks$(\{1,2\}) = (\{F_1\} \cup \{F_1\}) \setminus \{F_1\}$ (lines 56f) and a subsequent call to latestOpenCommonFork$(\{1,2\}) = \{F_1\} \setminus \{\} = F_1$ (lines 54f) with getOpenParentForks$(F_1) = \{\}$ (line 55, lines 49-51). To this end, $fc_{\div}$ is determined to $fc_{\div} = \{(\{1,2\}, \{F_1\})\}$. Because $fc_{\div}$ consists only of a single element, a call to function disjRel will have no effect, i. e., the set of forks that have to be closed are determined to $fc_{\odot} = fc_{\div} = \{F_1\}$.

The call to closeForkPartially (line 64) detects that F_1 has to be completely closed, and therefore, it calls closeFork. F_1 is the only fork and no nested parallel regions have to be closed (lines 88-90). Final steps in the execution of closeFork are the computation of $L = \{1,2\}$ (line 91) and the creation of a new join node which merges the path by connecting all nodes in L to the newly created join node (line 94). Note that the naming of a new join is fit to the name of its matching fork. Finally, the new join node is added to the set of scheduled node (line 95). This will have no effect, because the structuring algorithm is finished at that point.

To this end, the resulting structured graph is given as

$SG_2 = (\{1, 2, F_1, J_1\}, \{(\text{sched}, F_1, 1), (\text{sched}, F_1, 2), (\text{sched}, 1, J_1), (\text{sched}, 2, J_1)\})$

Example 3 Figure 3.9 shows the structured graph that was initialized with an ADG as follows:

$SG_3 = (\{1, 2, 3, 4\},$
$\{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 2, 4), (\text{immediate}, 3, 4)\})$

In this example, most steps of the algorithm that are necessary to structure SG_3 have been handled in the previous examples. A new step will be to add action 4 that depends on two actions, which requires to merge the corresponding paths. The case of merging nodes to schedule further actions makes this example more practical compared to Example 2. A step-by-step execution of structureGraph(SG_3) results in the following behavior:

- *Iteration 1:* Node 1 is the only node that has no incoming dependencies. It is selected for scheduling ($V_{schedulable} = \{1\}$, line 19), which results in a dependency mapping $dep = \{(\{1\}, \{\})\}$. Analogous to Example 1, the 1-0-relation triggers the scheduling of node 1 in function scheduleActions (line 45).

- *Iteration 2:* Node 2 and 3 have incoming dependencies only to scheduled nodes, i. e., node 1. They are selected for scheduling ($V_{schedulable} = \{2, 3\}$, line 19). The computation of dep results in $\{(\{2\}, \{1\}), (\{3\}, \{1\})\}$ (line 20), and the application of function disjRel to this set yields $dr = \{(\{2, 3\}, \{1\})\}$ (line 24). This is a single n-1-relation which will only have an effect in function scheduleForks (line 25). dr' is the selection of all n-1-relations in dr, i. e., $dr' = dr$ (line 32). This set consists of one tuple, i. e., one fork is created and the fork and the corresponding actions are scheduled (lines 34, 35 and 37). In particular, a scheduling dependency is created

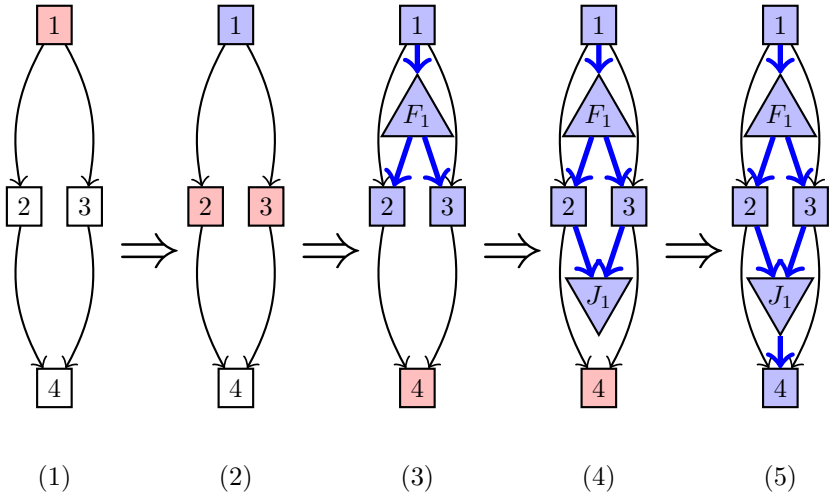


Figure 3.9: *Example 3:* ADG, structuring steps and the final structured graph.

from the source node 1 to the fork F_1 and from the fork to each schedulable node (line 36).

- *Iteration 3:* Finally, node 4 is selected for scheduling (line 19). The computation of *dep* yields a single element which is computated as follows: $S_1 = \{2,3\}$ is the set of nodes with a dependency to node 4 (line 21). Nodes 2 and 3 have no scheduling dependency to each other, i. e., $S_2 = S_1 = \{2,3\}$ (line 22). Determining the last node in S_2 results in $S_3 = S_2 = S_1 = \{2,3\}$ (line 23). Finally, the dependency mapping is determined to $dep = \{(\{4\},\{2,3\})\}$ (lines 20ff). Because *dep* contains only one element, the application of disjRel yields $dr = dep$ (line 24). The element in dr is a 1-n-relation and triggers the schedule of a join node (line 27). The computation of $fc_{\div}$ calls first openNonCommonForks$(\{2,3\})$ (line 60), which needs to compute getOpenParentForks(2) = getOpenParentForks$(3) = \{F_1\}$ (line 57, lines 49ff). The intersection of these sets results in the same set, i. e., $commonForks(\{2,3\} = \{\}$ (line 53). Hence, the call to openNonCommonForks$(\{2,3\})$ returns $\{\}$. The latestOpenCommonFork$(\{2,3\})$ returns the latest common fork *in* flow direction, i. e., the first common fork of nodes 2 and 3 against flow direction:
commonForks$(\{2,3\}) \setminus \bigcup_{f \in \text{commonForks}(\{2,3\})}$ getOpenParentForks$(f) =$
$\{F_1\} \setminus$ getOpenParentForks$(F_1) = \{F_1\} \setminus \{\} = F_1$
To this end, $fc_{\div} = \{(\{2,3\},\{F_1\})\}$ contains only one element and the application of disjRel to $fc_{\div}$ results the same set for $fc_{\odot}$ (line 62). The content of the first set in the tuple, i. e., $\{2,3\}$, is given to closeForkPartially (line 64). The execution of this function results in the following steps: $f = \{F_1\}$ (line 68), $suc_{all} = \{2,3\}$ (line 69), $suc_{close} = \{2,3\} = suc_{all}$ (line 70), which will trigger closeFork(F_1) (line 72). The parallel region contains no child parallel regions, i. e., the code in line 88-90 has no effect. The remaining part of the function determines the last nodes of F_1 and connects them to a newly created join node (line 91ff).

- *Iteration 4:* Except for a newly created join node, no action has been scheduled. Hence, the new set of schedulable nodes is still $\{4\}$. The computation of *dep* results in a different set because a join node has been scheduled after the nodes that node 4 depends on. The first two sets yield the same value as in the previous step, i. e., $S_1 = \{3, 4\} = S_2$ (lines 21ff). S_3 will contain the last scheduled nodes after node 2 and node 3, i. e., the previously created join node J_1. Hence, the dependency mapping is $dep = \{(\{4\}, \{J_1\})\} = dr$, which is a 1-1-relation. The tuple will be handled by scheduleActions (line 26), which schedules node 4 by adding a dependency from join $J - 1$ to node 4 (lines 44f).

- *Final Join:* The computation of the set of last nodes will result in a single node, i. e., node 4. Therefore, no joins have to be scheduled.

To this end, the resulting structured graph is given as

$$SG_3 = (\{1, 2, 3, 4, F_1, J_1\},\\ \{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 2, 4), (\text{immediate}, 3, 4),\\ (\text{sched}, 1, F_1), (\text{sched}, F_1, 2), (\text{sched}, F_1, 3),\\ (\text{sched}, 2, J_1), (\text{sched}, 3, J_1), (\text{sched}, J_1, 4)\})$$

Example 4 Figure 3.10 shows the structured graph that was initialized with an ADG as follows:

$$SG_4 = (\{1, 2, 3, 4, 5\}, \{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 1, 5),\\ (\text{immediate}, 2, 4), (\text{immediate}, 3, 5)\})$$

This example demonstrates a very special part in the structuring algorithm which considers implicit scheduling conditions to avoid early synchronization if possible (line 22). Especially the beginning of the example corresponds to Example 3. Hence, we can quickly start with the structuring part of interest. We recommend to read Example 3 before starting with this example. A step-by-step execution of structureGraph(SG_4) results in the following behavior:

- *Iteration 1-2:* The scheduling of node 1, 2 and 3 proceeds as in Example 3. Hence, we refer the reader to iteration 1 and 2 of Example 3.

- *Iteration 3:* Nodes 4 and 5 are selected for scheduling, i. e., $V_{schedulable} = \{4, 5\}$. The dependency mapping will yield a set of two tuples. The first tuple for node 4 is computed as $S_1 = S_2 = S_3 = \{2\}$ (line 21-23). The second tuple for node 5 is computed as follows: $S_1 = \{1, 3\}$ (line 21). The next step removes nodes in S_1 that have a scheduling path to any other node to this set. Node 1 is filtered out because it has a scheduling path to node 3, i. e., $S_2 = \{3\}$. To this end, the computation of *dep* yields $\{(\{4\}, \{2\}), (\{5\}, \{3\})\}$. The potential source nodes to create the scheduling dependencies, i. e., the secondary sets of tuples, are disjunctive. disjRel has no effect and assigns *dep* to *dr*. *dr* contains two tuples with a 1-1-relation which are both handled by scheduleActions (lines 26, 42f). Each action is scheduled

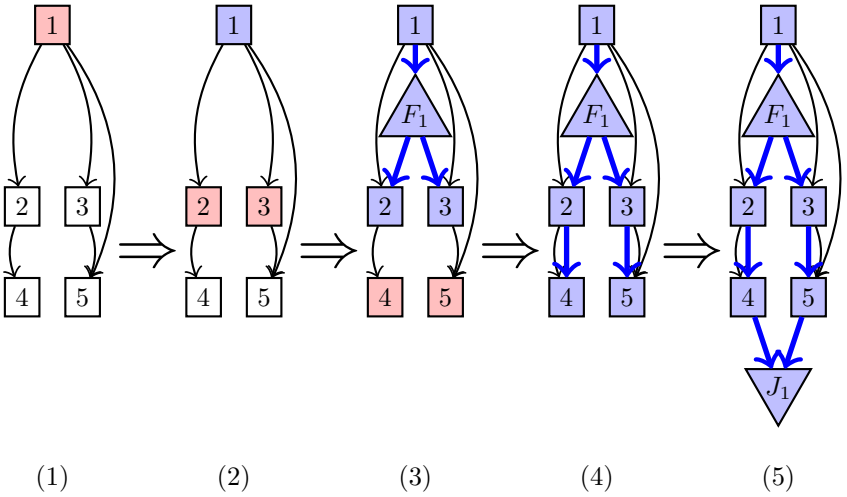


Figure 3.10: *Example 4:* ADG, structuring steps and the final structured graph.

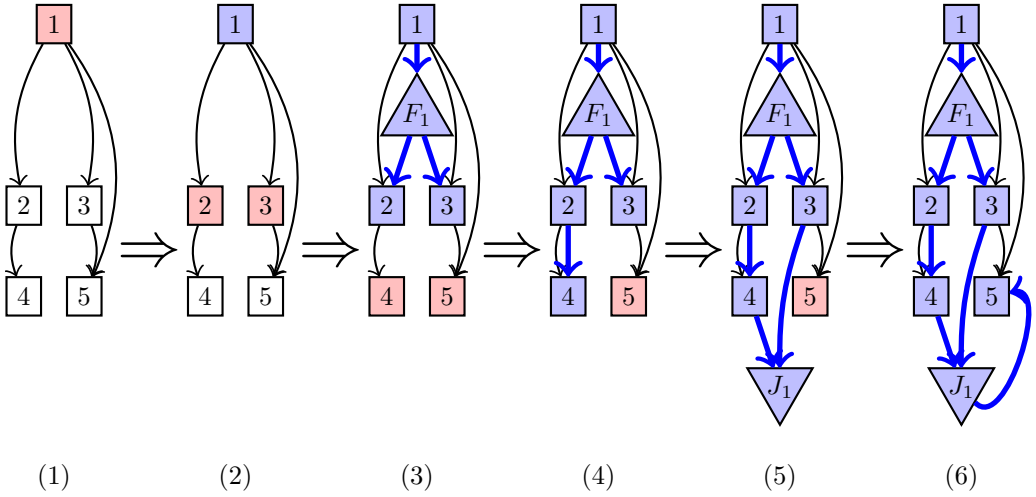


Figure 3.11: *Example 4b - without considering implicit scheduling:* ADG, structuring steps and the final structured graph.

by adding a scheduling dependency from the node in the corresponding secondary set (lines 43-45).

- *Final Join:* The scheduling of a finalizing join to close open forks is done as in Example 2. scheduleFinalJoin computes the set of nodes without outgoing scheduling dependencies, which results in $L = \{4, 5\}$ (line 98). Hence, scheduleJoins($\{4, 5\}$) is called next (line 58ff). The computation of the set of forks that have to be closed yields $fc_{\div} = \{(\{4, 5\}, \{F_1\})\}$ (lines 60f). Application of disjRel has no effect, because the set has only one element, i. e., $fc_{\odot} = fc_{\div}$ (line 62). Function closeForkPartially is applied to $\{(\{4, 5\}, \{F_1\})\}$ and determines that all paths have to be closed (lines 64).

As a consequence, closeForks is called to close F_1. A new fork is created and scheduling dependencies are added from the nodes 4 and 5 to that join (lines 91ff).

To this end, the resulting structured graph is given as

$SG_4 = (\{1,2,3,4,5\}, \{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 1, 5),$
$(\text{immediate}, 2, 4), (\text{immediate}, 3, 5),$
$(\text{sched}, 1, F_1), (\text{sched}, F_1, 2), (\text{sched}, F_1, 3), (\text{sched}, 2, 4),$
$(\text{sched}, 3, 5), (\text{sched}, 4, J_1), (\text{sched}, 5, J_1)\})$

Example 4b Figure 3.11 shows the same structured graph as given in Figure 3.10. The scheduling algorithm that was applied here is completely the same except for consideration of implicit scheduling dependencies. In particular, the computation of the set S_2 in line 22 of Figure 3.3 was ignored in the execution of this example. Due to that reason, the scheduling of node 5 keeps the dependency to node 1, which requires to join all paths starting after node 1. To this end, node 5 has to be sequentially scheduled and parallelism is lost. Example 4 has shown that node 5 can be scheduled in parallel to node 4.

Example 5 Figure 3.12 shows the structured graph that was initialized with an ADG as follows:

$SG_5 = (\{1,2,3,4,5\},$
$\{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 2, 4),$
$(\text{immediate}, 2, 5), (\text{immediate}, 3, 5)\})$

This example will yield a structure with more than one parallel region. We assume that the reader studied the previous examples and is already familiar with scheduling of forks and actions. In future examples, we will focus on scheduling of join nodes, which is more subtle in the structuring algorithm. A step-by-step execution of structureGraph(SG_5) results in the following behavior:

- *Iteration 1:* Node 1 can be scheduled. This means a single action without predecessors can be scheduled. Thus, the 1-0-relation will trigger a scheduling process in scheduleActions. Node 1 is scheduled without adding any additional dependencies.

- *Iteration 2:* Node 2 and node 3 can be scheduled. Both depend on node 1, which is a 2-1-relation. Similar to previous examples, the actual scheduling is done by scheduleForks, which creates fork F_1 and corresponding scheduling dependencies.

- *Iteration 3:* In this step, node 4 and node 5 have no immediate dependencies to an unscheduled node. They are selected for scheduling (line 19). The dependency mapping contains two sets, i. e., $dep = \{(\{4\}, \{2\}), (\{5\}, \{2, 3\})\}$ (lines 20ff). The application of disjRel to *dep* merges the tuples (line 24). The dependency lists are non-disjunctive and require the tuples to be merged (lines 11ff). Hence, *dr* turns out to be $\{(\{4, 5\}, \{2, 3\})\}$. This is a 2-2-relation which requires to schedule a join node (line 27).

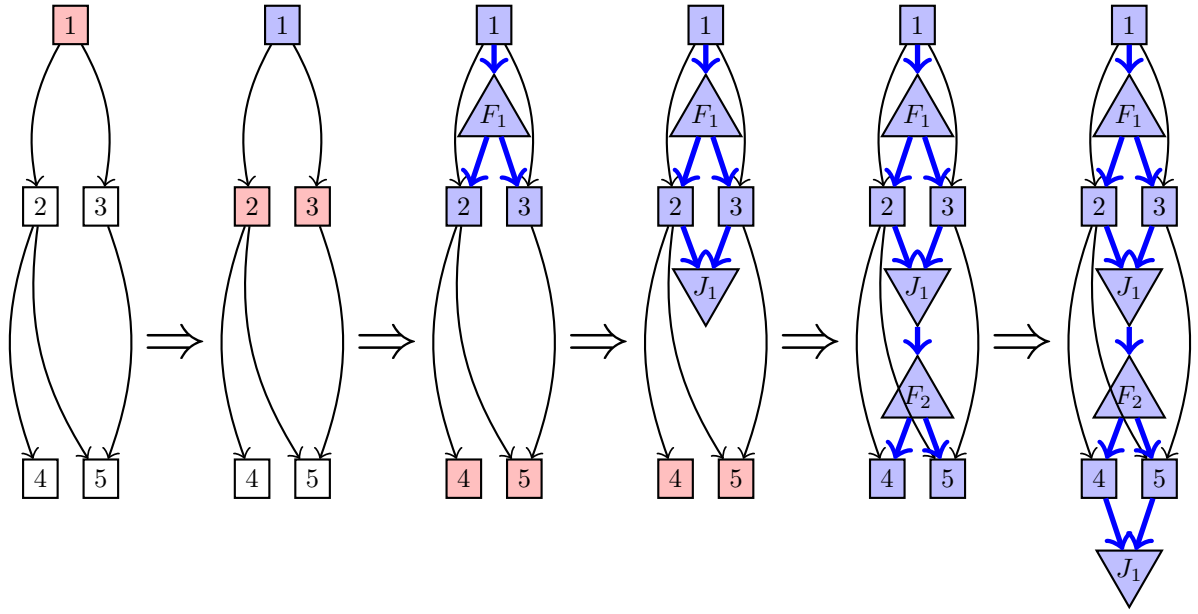


Figure 3.12: *Example 5:* ADG, structuring steps and the final structured graph.

The computation of the set of forks that have to be closed is determined in function scheduleJoins (lines 58ff). It starts with determining getOpenParentForks(2) and getOpenParentForks(3), which result both in $\{F_1\}$. Moreover, the following sets are computed: commonForks$(\{2,3\}) = \{F_1\}$, latestOpenCommonFork$(\{2,3\}) = F_1$ and openNonCommonForks$(\{2,3\}) = \{\}$.

Finally, $fc_{\div}$ can be determined as $\{(\{2,3\},\{F_1\})\}$. Application of disjRel is without effect on sets with only one element, i.e., $fc_{\odot} = \{(\{2,3\},\{F_1\})\}$. In the next step, closeForkPartially is applied to the set of nodes which defines the paths that have to be closed, i.e., $D_g = \{(2,3)\}$ (line 64). The function first checks whether the complete fork has to be closed. The set of direct successors of F_1 is exactly the set of nodes that have to be closed. Thus, the complete fork has to be closed, which results in a call to closeFork (line 72). All nodes that have been scheduled after F_1 are non-fork nodes. Hence, no nested parallel region has to be closed (line 88-90). To this end, fork F_1 is closed by inserting and scheduling a new join node and creating corresponding scheduling dependencies (lines 92ff).

- *Iteration 4:* The set of schedulable nodes remains unchanged because the preceding step scheduled only a join node. The mapping of dependencies yields a different set for both nodes: for node 4, it yields the tuple $(\{4\},\{J_1\})$ with $S_1 = \{2\}$, $S_2 = \{2\}$ and $S_3 = \{J_2\}$. Node 5 yields the tuple $(\{5\},\{J_1\})$ with $S_1 = \{2,3\}$, $S_2 = \{2,3\}$ and $S_3 = \{J_2\}$ (lines 20ff). The mapping $dep = \{(\{4\},\{J_1\}),(\{5\},\{J_1\})\}$ contains two non-disjunctive sets. Hence, disjRel yields $dr = \{(\{4,5\},\{J_1\})\}$. dr contains a

2-1-relation and enforces the scheduling of a fork (line 25). A new fork is created and scheduled by scheduleForks (line 34) and corresponding dependencies are added, thereby node 4 and node 5 are also scheduled.

- *Final Join:* This step is similar to Example 2 and Example 4: scheduleFinalJoin computes the set of nodes without outgoing scheduling dependencies (line 98). The result is $L = \{4, 5\}$ and requires to add a join that merges these nodes. The remaining part of the structuring is identical to the scheduling of the final join in Example 4.

To this end, the resulting structured graph is given as

$$SG_5 = (\{1, 2, 3, 4, 5, F_1, J_1, F_2, J_2\},\\ \{(\text{immediate}, 1, 2), (\text{immediate}, 1, 3), (\text{immediate}, 2, 4),\\ (\text{immediate}, 2, 5), (\text{immediate}, 3, 5),\\ (\text{sched}, 1, F_1), (\text{sched}, F_1, 2), (\text{sched}, F_1, 3),\\ (\text{sched}, 2, J_1), (\text{sched}, 3, J_1), (\text{sched}, J_1, F_2),\\ (\text{sched}, F_2, 4), (\text{sched}, F_2, 5), (\text{sched}, 4, J_2),\\ (\text{sched}, 5, J_2)\})$$

Example 6 Figures 3.13 and 3.14 show the structured graph that was initialized with an ADG as follows:

$$SG_6 = (\{1, 2, 3, 4, 5, 6\},\\ \{(\text{immediate}, 1, 3), (\text{immediate}, 1, 4), (\text{immediate}, 2, 5),\\ (\text{immediate}, 3, 6), (\text{immediate}, 4, 6), (\text{immediate}, 5, 6)\})$$

This example will result in a structured graph with nested parallel regions. It demonstrates scheduling with different priorities and the scheduling of multiple joins in a single iteration. The corresponding iteration has been split into two sub-steps for better readability. A step-by-step execution of structureGraph(SG_5) results in the following behavior:

- *Iteration 1:* Two nodes are able to be scheduled in the first iteration, i. e., node 1 and node 2. This ends up with the scheduling tuple $dep = \{(\{1, 2\}, \{\})\}$, i. e., a single 2-0-relation. This is handled by function scheduleForks: a fork is inserted and scheduled. Corresponding scheduling dependencies are added to node 1 and 2.

- *Iteration 2:* Nodes 3, 4 and 5 are now ready for scheduling. The computation of *dep* results in 3 tuples: $(\{3\}, \{1\})$, $(\{4\}, \{1\})$, $(\{5\}, \{2\})$. Application of disjRel merges tuples with non-disjunctive dependency sets, which yields in the given example $dr = \{(\{3, 4\}, \{1\}), (\{5\}, \{2\})\}$. The first tuple is a 2-1-relation and the second one is a 1-1-relation. Due to the priority of the scheduling (lines 25-27) the scheduling of forks gets priority. Thus, function scheduleForks(dr) is called and filters the tuples of interest, i. e., $dr' = \{(\{3, 4\}, \{1\})\}$ (line 32). It remains to create a fork node and to schedule it as usual, i. e., creating scheduling dependencies and adding node 3 and node 4 to the set of scheduled nodes.

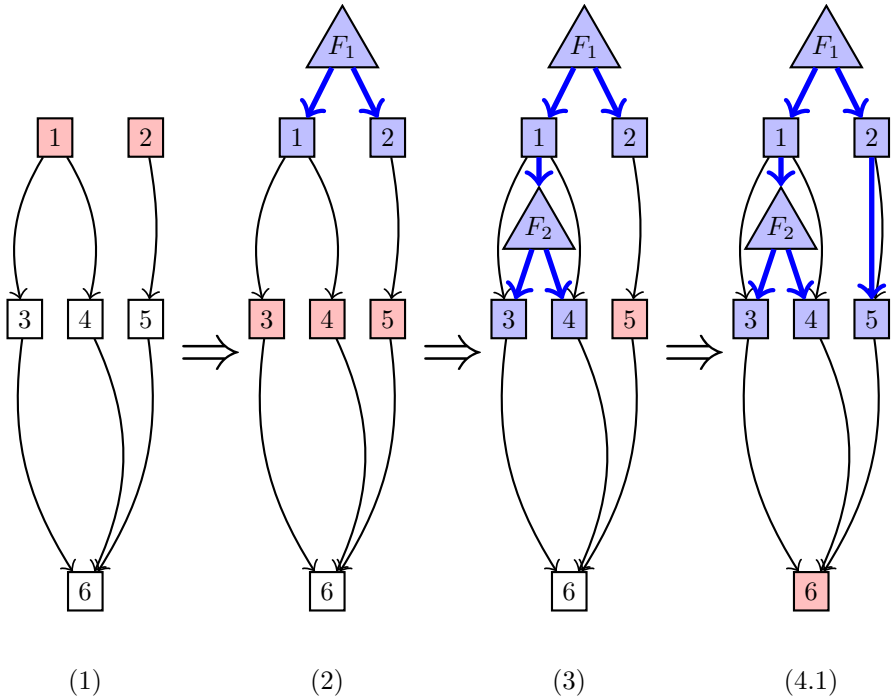


Figure 3.13: *Example 6 (Part 1):* ADG, structuring steps and the final structured graph.

- *Iteration 3:* Node 5 has not been scheduled in the previous iteration and is still ready for scheduling. All other unscheduled nodes, i. e., node 6, still depends on other unscheduled nodes. The computation of *dep* yields for node 5 in the same tuple as in the previous iteration: $(\{5\}, \{2\})$. This is a 1-1-relation and will trigger the scheduling of an action. The action is added to the set of scheduled nodes and a scheduling dependency is added.

- *Iteration 4.1:* Finally, node 6 is now schedulable (line 19). It depends on the nodes 3, 4 and 5. The dependency mapping is given as $dep = \{(\{6\}, \{3,4,5\})\} = dr$ (lines 20ff). This is a 1-3-relation and triggers the scheduling of a join node (line 27). The function starts with the computation of $fc_{\div}$, which requires to determine the set of open parent forks first: $\mathsf{getOpenParentForks}(3) = \{F_1, F_2\}$, $\mathsf{getOpenParentForks}(4) = \{F_1, F_2\}$ and $\mathsf{getOpenParentForks}(5) = \{F_1\}$. The intersection of these sets represent the set of common forks: $\mathsf{commonForks}(\{3,4,5\}) = \{F_1\}$. The set of non-common forks and the latest open common fork are then determined by $\{F_2\}$ and F_1, respectively. Thus, $fc_{\div}$ results in the single tuple $= \{(\{3,4,5\}, \{F_1, F_2\})\}$ which is not modified by $\mathsf{disjRel}$, i. e., $fc_{\odot} = fc_{\div}$.

  The next step is to apply $\mathsf{closeForkPartially}$ to $\{3,4,5\}$ which determines the last common parent fork (line 68). Next, it checks whether all paths have to be closed (lines 69-71) which is true in this case. Hence, $\mathsf{closeFork}(F_1)$ is called (line 72) which starts with closing child parallel regions (lines 88-90). The return value of $\mathsf{getNextChildForks}(F_1)$ is computed with the following steps: all open forks sched-

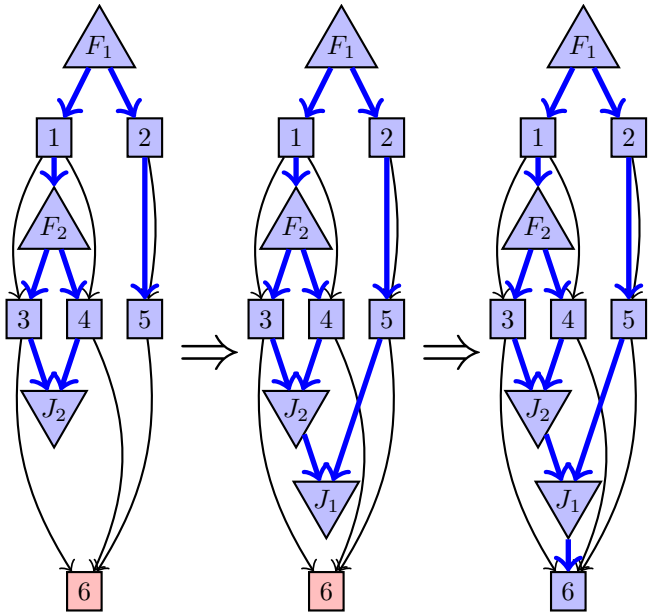


Figure 3.14: *Example 6 (Part 2):* ADG, structuring steps and the final structured graph.

uled after F_1 are determined: $f_{allChilds} = \{F_2\}$. Removing those forks that have a scheduling dependency to another fork in $f_{allChilds}$ obviously results in the same set because it contains only one fork. As a result, getNextChildForks(F_1) turns out to be $\{F_2\}$, which means that one parallel region has to be closed first (line 90). The parallel region that is opened by fork F_2 has no further child parallel regions (lines 88-90). The last nodes that have been scheduled after fork F_2 are merged by adding a join node and corresponding dependencies (lines 91-95). This finalizes the call of closeFork(F_2) and returns to the call of closeFork(F_1) which is handled in the second part of iteration 4.

- *Iteration 4.2:* closeFork(F_1) continues in line 91. To merge the paths of fork F_1 the function determines the last scheduled nodes after fork F_1, which yields $\{J_2, 5\}$. As usual, a new join node is created and the last scheduled nodes of F_1 are connected by a scheduling dependency to that join.

- *Iteration 5:* Due to the scheduling of join nodes, no action has been scheduled. Hence, node 6 is still schedulable. The dependency mapping for this iteration changes because all paths with a dependency to node 6 have been merged. The computation of *dep* yields $\{(\{6\}, \{J_1\})\}$, i.e., we get a 1-1-relation. This triggers the execution of function scheduleActions. The function creates a scheduling dependency from join node J_2 to node 6, and it adds node 6 to the set of scheduled nodes.

- *Final Join:* The computation of the set of last nodes will result in a single node, i. e., node 6. Therefore, no joins have to be scheduled.

To this end, the resulting structured graph is given as

$$\begin{aligned} SG_6 = (&\{1,2,3,4,5,6,F_1,J_1,F_2,J_2\},\\ &\{(\text{immediate},1,3),(\text{immediate},1,4),(\text{immediate},2,5),\\ &(\text{immediate},3,6),(\text{immediate},4,6),(\text{immediate},5,6),\\ &(\text{sched},F_1,1),(\text{sched},F_1,2),(\text{sched},1,F_2),\\ &(\text{sched},F_2,3),(\text{sched},F_2,4),(\text{sched},2,5),\\ &(\text{sched},3,J_2),(\text{sched},4,J_2),(\text{sched},J_2,J_1),\\ &(\text{sched},5,J_1),(\text{sched},J_1,6)\}) \end{aligned}$$

Example 7 Figures 3.15 and 3.16 show the structured graph that was initialized with an ADG as follows:

$$\begin{aligned} SG_7 = (&\{1,2,3,4,5,6,7\},\\ &\{(\text{immediate},1,2),(\text{immediate},1,3),(\text{immediate},2,4),(\text{immediate},2,5),\\ &(\text{immediate},3,6),(\text{immediate},5,6),(\text{immediate},4,7),(\text{immediate},6,7)\}) \end{aligned}$$

This example shows the ADG of Figure 3.1 in the beginning of this section. Our early-join strategy is explained by means of this example. Moreover, it yields a structured graph with more than one parallel region and with nested parallelism. A step-by-step execution of structureGraph(SG_7) results in the following behavior:

- *Iteration 1:* At the beginning only one node is schedulable, i. e., node 1. It results in a 1-0-relation which is handled by scheduleActions.

- *Iteration 2:* Node 2 and node 3 are now schedulable, which both depend on a single node. Hence, we get a single 2-1-relation which enforces the creation of a fork. F_1 is scheduled after node 1, which is the input dependency of node 2 and node 3. Consequently, both of these nodes are scheduled right after the fork. Thereby, each of these nodes opens an independent path.

- *Iteration 3:* Analogously, node 4 and node 5 are scheduled. Both nodes are schedulable and depend on a single node, namely node 2. The scheduling of the fork proceeds similar as in the preceding iteration. A new fork is created, added to the graph, and scheduling dependencies are created and added from node 2 to fork F_2 and from F_2 to node 4 and node 5.

- *Iteration 4.1:* Only one node is schedulable now. In particular, node 6 is the only node that depends not on any unscheduled nodes (line 19). It depends on node 5 and node 3. The dependency mapping will consist of one tuple, whose computation starts with $S_1 = \{3,5\}$. $S_2 = S_1$ must hold because no scheduling path exists between the nodes in S_1. Finally, S_3 determines the last scheduled nodes for the paths starting at node 3 and node 5, which also results in $S_3 = S_1 = \{3,5\}$. For this iteration, we have a 1-2-relation which is handled in scheduleJoins (line 27).

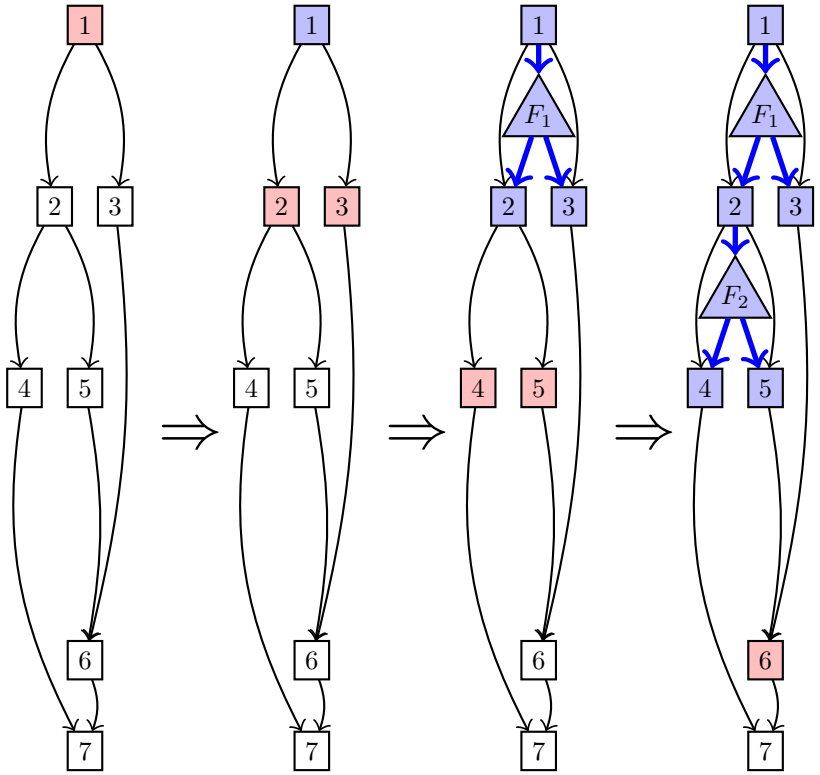


Figure 3.15: *Example 7 (Part 1):* ADG, structuring steps and the final structured graph.

As already done in previous examples, function scheduleJoins (line 27) starts with the computation of the set of forks which have to be closed to merge the control-flow at the given node set D_g (line 60). Therefore, the open parent forks for each node has to be determined: getOpenParentForks$(3) = \{F_1\}$ and getOpenParentForks$(5) = \{F_1, F_2\}$. The intersection of these sets yield $commonForks(\{3,5\}) = \{F_1\}$ (line 53). Further steps lead to calls to function openNonCommonForks$(\{3,5\}) = \{F_1, F_2\} \setminus \{F_1\} = \{F_2\}$ (line 57) and latestOpenCommonFork$(\{3,5\}) = F_1 \setminus$ getOpenParentForks$(F_1) = \{F_1\} \setminus \{\} = \{F_1\}$ (line 55). Consequently, $fc_{\div} = \{(\{3,5\}, \{F_1\})\}$ will result in $fc_{\odot} = \{(\{3,5\})\} = fc_{\div}$, i. e., function closeForkPartially is called next with parameter $\{3,5\}$.

The steps in the execution of closeForkPartially$(\{3,5\})$ yield the following intermediate results: $f = F_1$, the set of direct successors of this fork is $suc_{all} = \{2,3\}$. All elements in this set have a scheduling path to a node in $\{3,5\}$, i. e., all paths have to be closed (line 71) because $suc_{close} = suc_{all} = \{2,3\}$ (line 70). The first stage of closeFork(F_1) is to close all child parallel regions of F_1. The computation of the forks of child parallel regions is recursively done by calls to getNextChildForks. The call to getNextChildForks(F_1) yields a set of all successors, i. e., $suc = \{2,3\}$. This set

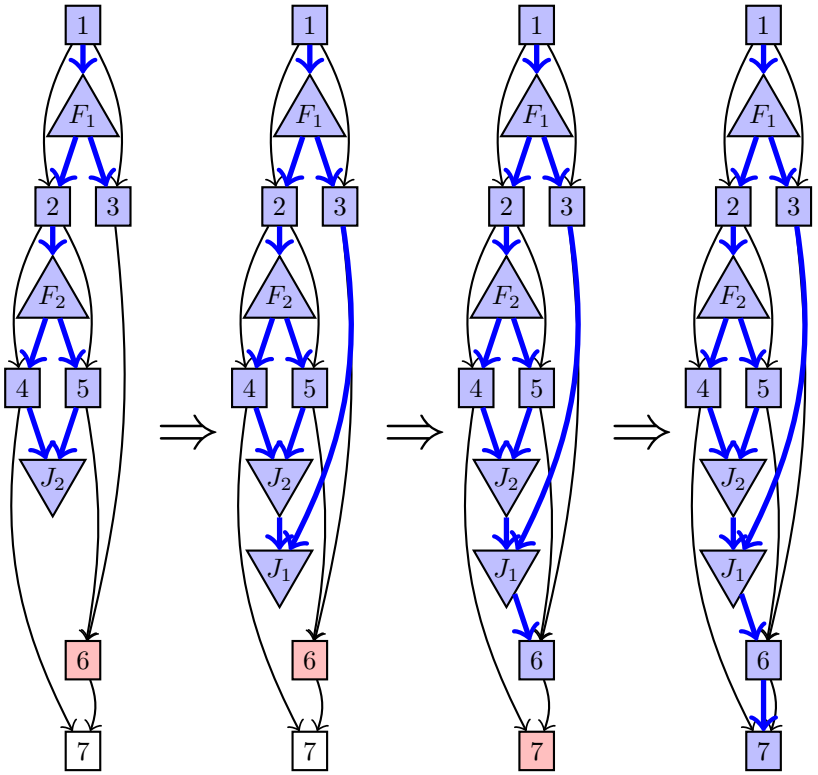


Figure 3.16: *Example 7 (Part 2):* ADG, structuring steps and the final structured graph.

is split into a list of forks $suc_f = \{\}$ and non-forks $\{2, 3\}$. For each non-fork node, the function descends recursively, i. e., in this case it computes getNextChildForks(2) and getNextChildForks(3). The call of getNextChildForks(2) yields $suc = \{F_2\}$, $suc_f = \{F_2\}$, $suc_a = \{\}$ and returns $suc_f \cup suc_a = \{F_2\}$. The call of getNextChildForks(3) yields $suc = \{\}$, $suc_f = \{\}$, $suc_a = \{\}$ and returns the empty set. The recursion stops at that point and returns to the first call of getNextChildForks, which computes $suc_a =$ getNextChildForks(2) $\cup$ getNextChildForks(3) $= \{F_2\} \cup \{\} = \{F_2\}$. The forks in this set have to be closed before F_1 is closed (lines 89f).

In the recursive call to closeFork(F_2), no further parallel region is closed, i. e., $cf = \{\}$ (lines 88-90). The last scheduled node of F_2 are $\{4, 5\}$ (line 91), which are connected to a newly created join J_2 (line 92). The function returns to continue the execution of closeFork(F_1), which is continued in the second part of iteration 4.

- *Iteration 4.2:* The continuation of closeFork(F_1) starts in line 91 to determine the set of last nodes of fork F_1, which yields $\{3, J_1\}$. The remaining execution proceeds similar to other schedules of joins: A new join node is created which is named after its parent fork. Scheduling dependencies are added from the previously determined set

of last nodes to the newly created join node. To this end, all paths with a dependency to node 6 have been merged, i. e., node 6 can now be scheduled in the next iteration.

- *Iteration 5:* The preceding iteration only scheduled synchronization points, namely join nodes. Hence, node 6 is still the only node that can be scheduled. The last nodes of all dependencies end up in join J_1, i. e., we get a 1-1-relation for this node. The scheduling is handled by scheduleActions which creates a scheduling dependency from join J_1 to node 6 and adds the node to the set of scheduled actions.

- *Iteration 6:* Finally, the last node is ready for scheduling. Node 7 is schedulable and yields a 1-1-relation in the dependency mapping. The dependency starts at node 6 and is handled again by scheduleActions which proceeds as usual. The node is scheduled and a scheduling dependency is added to the graph.

- *Final Join:* Node 7 is the only node without outgoing dependencies, indicating that all parallel regions have been closed. Thus, no creation of additional join nodes is required.

To this end, the resulting structured graph is given as

$$\begin{aligned} SG_7 = (&\{1,2,3,4,5,6,7,F_1,J_1,F_2,J_2\},\\ &\{(\text{immediate},1,2),(\text{immediate},1,3),(\text{immediate},2,4),(\text{immediate},2,5)\\ &(\text{immediate},3,6),(\text{immediate},5,6),(\text{immediate},4,7),(\text{immediate},6,7),\\ &(\text{sched},1,F_1),(\text{sched},F_1,2),(\text{sched},F_1,3),\\ &(\text{sched},2,F_2),(\text{sched},F_2,4),(\text{sched},F_2,5),\\ &(\text{sched},4,J_2),(\text{sched},5,J_2),(\text{sched},J_2,J_1),\\ &(\text{sched},3,J_1),(\text{sched},J_1,6)\}) \end{aligned}$$

Example 8 Figures 3.17 and 3.18 show the structured graph that was initialized with an ADG as follows:

$$\begin{aligned} SG_8 = (&\{1,2,3,4,5,6\},\\ &\{(\text{immediate},1,4),(\text{immediate},1,5),(\text{immediate},2,6),(\text{immediate},5,6)\}) \end{aligned}$$

This example demonstrates sub-forking of our algorithm. If a fork has to be closed and only a part of its paths has to be closed, an additional fork is used to split these paths from the select fork. The newly created fork is then the fork that is actually closed, which leaves the other paths open. The inserted join requires less paths to synchronize, and therefore, this may reduce synchronization overhead. A step-by-step execution of structureGraph(SG_9) results in the following behavior:

- *Iteration 1:* The nodes 1, 2 and 3 are schedulable. Because it is the very first iteration, a fork is inserted and a scheduling dependency is created for each schedulable node.

- *Iteration 2:* Only the two nodes 4 and 5 are schedulable. Both depend on a single node, i. e., node 1. This results in a 2-1-relation which triggers the creation of another fork. Fork F_2 is scheduled as usual by adding a scheduling dependency from node 1 to F_2 and one scheduling dependency from F_2 to each node 4 and node 5.

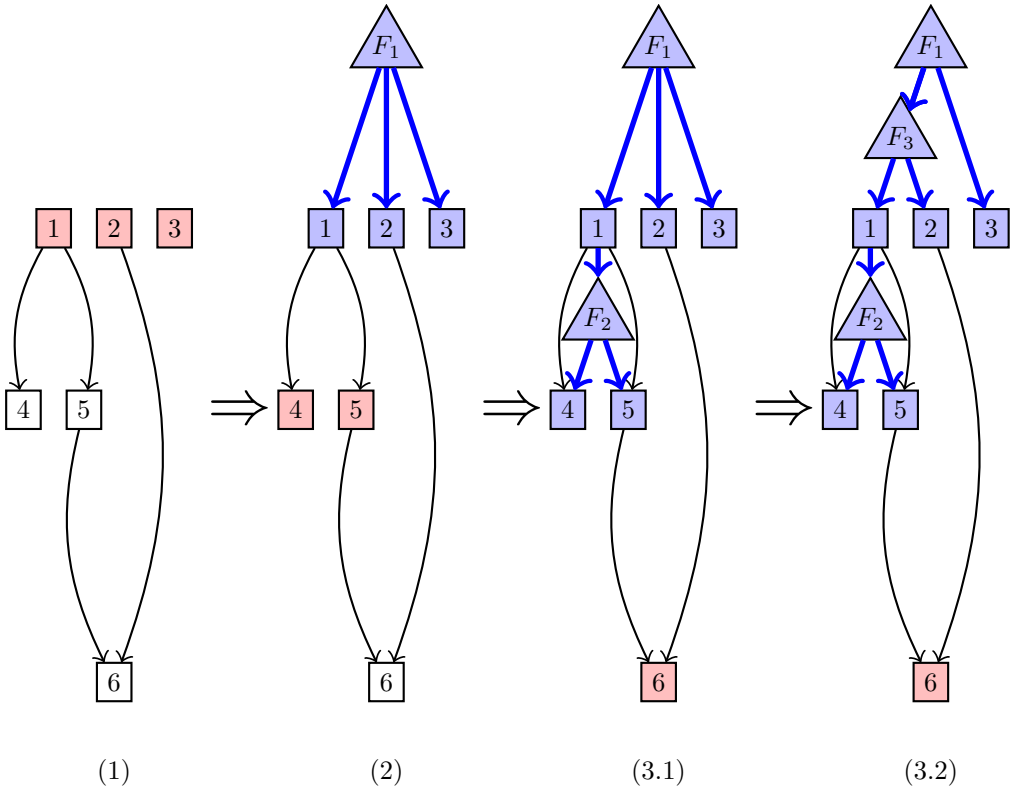


Figure 3.17: *Example 8 (Part 1):* ADG, structuring steps and the final structured graph.

- *Iteration 3.1:* Node 6 is now schedulable and requires to merge two paths of three of fork F_1. The dependency mapping yields $dep = \{2, 5\} = dr$ (lines 20-24), i. e., a 1-2-relation which requires to join the corresponding paths (line 27). Function scheduleJoins starts with the computation of $fc_{\div}$, which requires to compute the sets of open parent forks for node 2 and node 5. getOpenParentForks(2) and getOpenParentForks(5) yield $\{F_1\}$ and $\{F_1, F_2\}$, respectively. Thus, the set of common forks is $\{F_1\}$. To this end, $fc_{\div} = \{(\{2, 5\}, \{F_1, F_2\})\} = fc_{\odot}$ denotes the set of forks that have to be closed. As usual, the closing of parallel regions starts with closeForkPartially.

  A call to closeForkPartially($\{2, 5\}$) starts with determining the latest common fork, which yields $f = F_1$ (line 68). Further steps yield $suc_{all} = \{1, 2, 3\}$ and $suc_{close} = \{1, 2\}$ (lines 69f). Because $suc_{all} \supset suc_{close}$ holds, the insertion of a fork is triggered to split the paths that have to be merged. Here, the new fork is named $f_{split} = F_3$ and added to the graph (lines 74f). The insertion of a node means also to replace corresponding scheduling dependencies to ensure the requirements in Definition 12. Therefore, the scheduling dependencies from fork F_1 to node 1 and node 2 are removed (line 76). Afterwards, new scheduling dependencies are created, i. e., a dependency from F_1 to the newly created fork F_3 and from F_3 to node 2 and node 3 (line 77). Having the new fork properly inserted and scheduled (line 78), all paths that have to

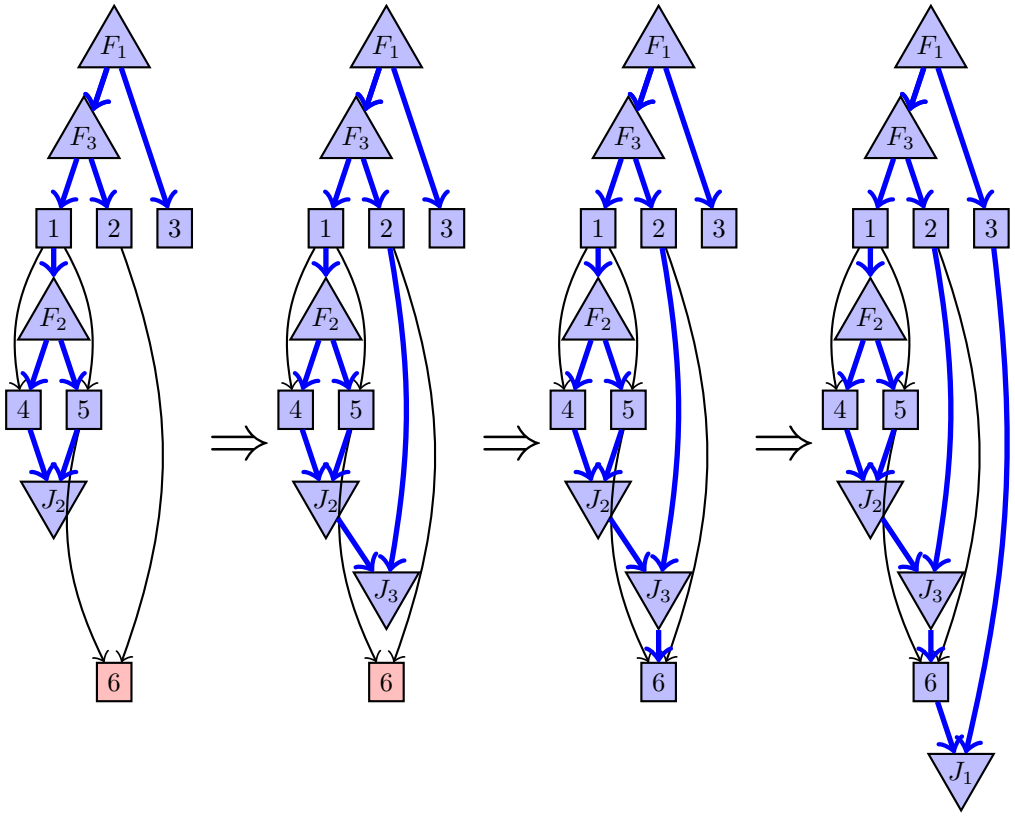


Figure 3.18: *Example 8 (Part 2):* ADG, structuring steps and the final structured graph.

be merged start now at the new fork F_3. Hence, the desired paths can be closed, by closing the new fork F_3 instead of the previously determined last common fork F_1. This is done by calling closeFork(F_3) in the second part of iteration 3.

- *Iteration 3.2:* A call to closeFork(F_3) starts with determining all open parallel regions. In this case, cf turns out to be F_2 (line 88). This fork must be closed first (lines 89f). Because F_2 has no further open parallel regions (lines 88-90), its paths are closed as usual. This is done by creation of a matching join and connecting the last nodes of F_2 to this join (lines 91-95). After closing fork F_2, the function returns to the call of closeFork(F_3) which is continued in the third part of iteration 3.

- *Iteration 3.3:* closeFork(F_3) continues its execution after closing F_2 in line 91. The closing of F_3 is analogous to previous schedules of joins. The last nodes of all paths starting at F_3 are connected to a new join node (lines 91-95).

- *Iteration 4:* Node 6 is schedulable again: its dependency mapping yields a 1-1-relation which triggers the scheduling of this node. A scheduling dependency is added from join J_3 to node 6.

- *Final Join:* After all nodes have been scheduled, two nodes in the structuring graph have no outgoing dependency. Hence, at least one parallel region is still open. A call to close these paths results in closing fork F_1. A new join is created, and scheduling dependencies from the last nodes of F_1 to the join node are added.

To this end, the resulting structured graph is given as

$SG_8 = (\{1, 2, 3, 4, 5, 6, F_1, J_1, F_2, J_2, F_3, J_3\},$
$\{(\text{immediate}, 1, 4), (\text{immediate}, 1, 5), (\text{immediate}, 2, 6), (\text{immediate}, 5, 6),$
$(\text{sched}, F_1, F_3), (\text{sched}, F_1, 3), (\text{sched}, F_3, 1),$
$(\text{sched}, F_3, 2), (\text{sched}, 1, F_2), (\text{sched}, F_2, 4),$
$(\text{sched}, F_2, 5), (\text{sched}, 4, J_2), (\text{sched}, 5, J_2),$
$(\text{sched}, J_2, J_3), (\text{sched}, 2, J_3), (\text{sched}, J_3, 6),$
$(\text{sched}, 6, J_1), (\text{sched}, 3, J_1)\})$

Example 9 Figures 3.19 and 3.20 show the structured graph that was initialized with an ADG as follows:

$SG_9 = (\{1, 2, 3, 4, 5, 6, 7, 8, 9\},$
$\{(\text{immediate}, 1, 3), (\text{immediate}, 1, 4), (\text{immediate}, 1, 5),$
$(\text{immediate}, 2, 6), (\text{immediate}, 2, 7), (\text{immediate}, 3, 8),$
$(\text{immediate}, 4, 8), (\text{immediate}, 5, 9), (\text{immediate}, 6, 9)\})$

This example shows an ADG with a larger number of nodes. It provides interleaving of nodes and nested parallelism. The actual intention is again the merging of paths. Here, we will have to add joins for two nodes with different predecessor sets. During the merging, it turns out to be the closing of a single fork. A step-by-step execution of structureGraph(SG_9) results in the following behavior:

- *Iteration 1:* Node 1 and node 2 are schedulable (line 19). Their dependency mapping yields $dep = \{(\{1\}, \{\}), (\{2\}, \{\})\}$ (lines 20-23). Both tuples have no incoming dependencies and are therefore merged by disjRel to $dr = \{(\{1, 2\}, \{\})\}$. The 2-0-relation enforces the creation and scheduling of a new fork. Both nodes are scheduled in this iteration.

- *Iteration 2:* In this step, the schedulable set of nodes consists of node 3 to node 7, i. e., $V_{schedulable} = \{3, 4, 5, 6, 7\}$ (line 19). The dependency mapping yields $dep = \{(\{3\}, \{1\}), (\{4\}, \{1\}), (\{5\}, \{1\}), (\{6\}, \{2\}), (\{7\}, \{2\})\}$, and the application of the function disjRel results in $dr = \{(\{3, 4, 5\}, \{1\}), (\{6, 7\}, \{2\})\}$ (line 24). Both tuples are a n-1-relation which requires to insert a fork for each tuple. The scheduling of a fork is done for each tuple by creating a new fork which is then connected to the corresponding nodes.

- *Iteration 3.1:* Next, node 8 and node 9 are schedulable. Computation of dep yields $\{(\{8\}, \{3\}), (\{8\}, \{4\}), (\{9\}, \{5\}), (\{9\}, \{6\})\}$, and the application of disjRel results in

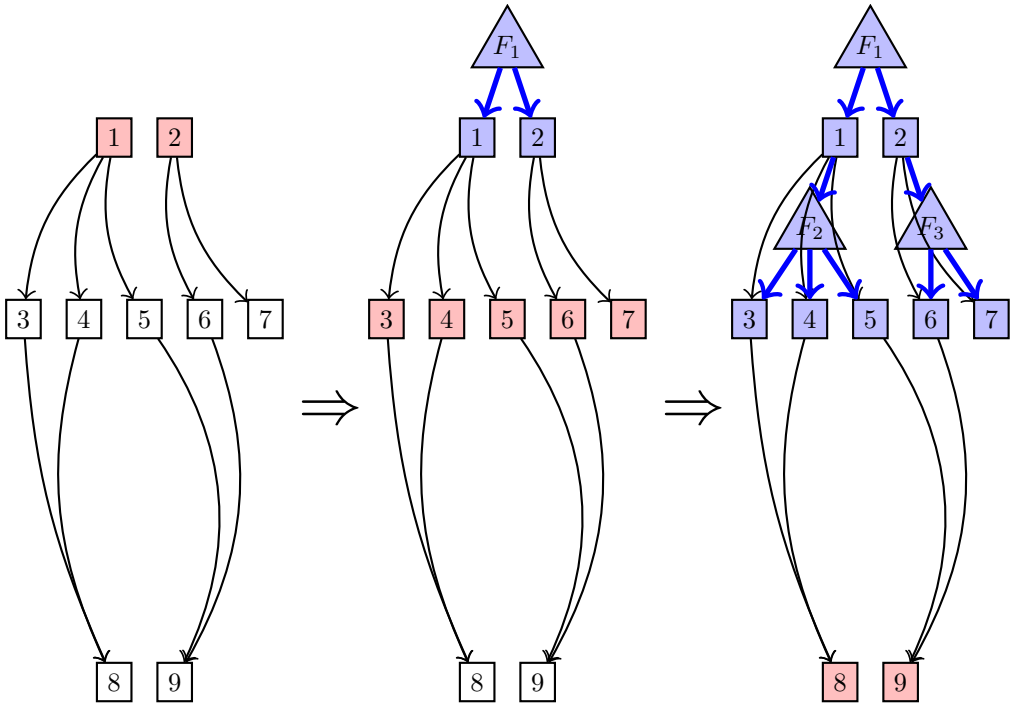


Figure 3.19: *Example 9 (Part 1):* ADG, structuring steps and the final structured graph.

$dr = \{(\{8\}, \{3,4\}), (\{9\}, \{5,6\})\}$ (line 24). Hence, a call to scheduleJoins is necessary to handle the tuples in dr with a 1-n-relation (line 27).

A call to scheduleJoins($\{(\{8\}, \{3,4\}), (\{9\}, \{5,6\})\}$) requires to compute two tuples for $fc_{\div}$ (lines 60f), i.e., (1) for $D_g = \{3,4\}$, which results in $(\{3,4\}, \{F_2\})$, and (2) for $D_g = \{5,6\}$, which results in $(\{5,6\}, \{F_1, F_2\})$. Application of disjRel results in a single tuple consisting of the merged sets of the tuples in $fc_{\div}$, i.e., $fc_{\odot} = \{(\{3,4,5,6\}, \{F_1, F_2\})\}$ (line 62). The next step is to call closeForkPartially which will determine that all paths of the latest open common parent have to be closed (lines 68-70), i.e., a call of closeFork(F_1) is triggered in line 72).

The procedure to close a fork requires to close child parallel regions first. Hence, closeFork(F_1) must determine the forks that are open and scheduled after F_1. In this case, cf turns out to be $\{F_2, F_3\}$. Recursive calls to closeFork(F_2) and closeFork(F_3) close the corresponding parallel regions, i.e., join J_2 is scheduled after nodes 3, 4 and 5, and join J_3 is scheduled after nodes 6 and 7. This finalizes the recursive calls.

- *Iteration 3.2:* The structuring algorithm continues the execution of closeFork(F_1) in line 91. To this end, the matching join for fork F_1 is scheduled after the previously created joins J_2 and J_3.

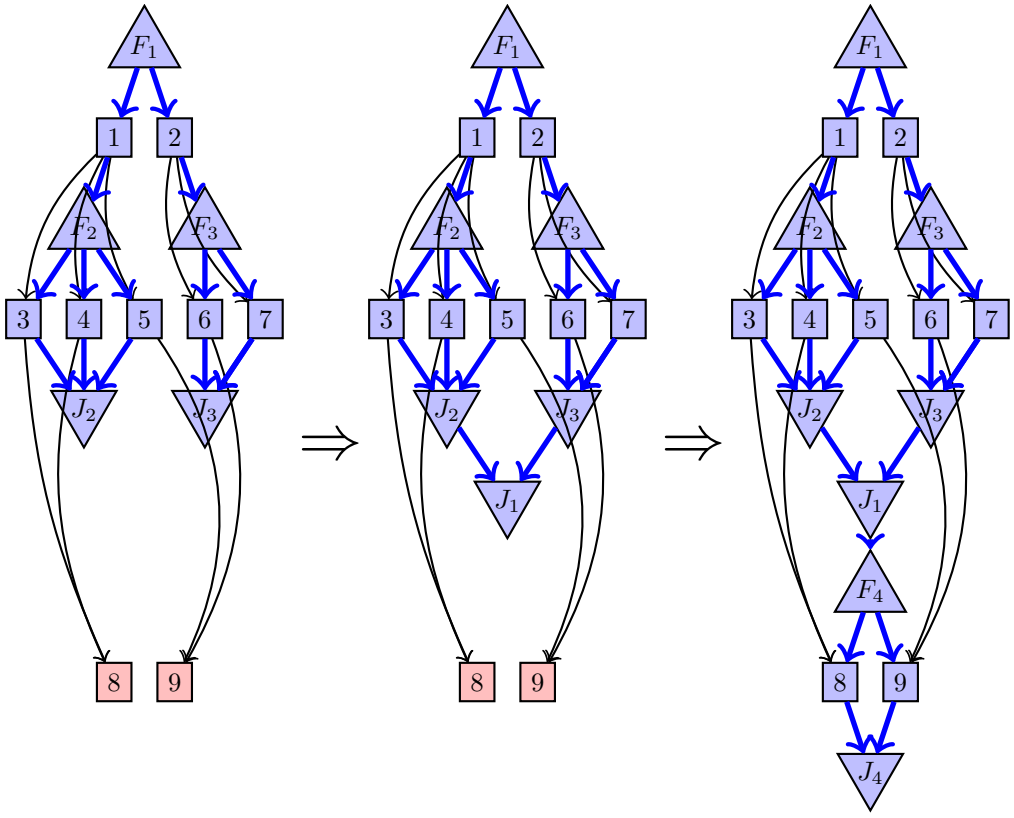


Figure 3.20: *Example 9 (Part 2):* ADG, structuring steps and the final structured graph.

- *Iteration 4:* The previous iteration only scheduled join nodes, i. e., the set of nodes that are ready to be scheduled consists still of node 8 and node 9. The dependency mapping turns now out to be $dep = \{(\{8\}, \{J_1\}), (\{9\}, \{J_1\})\}$, and the merged set is now $dr = \{(\{8, 9\}, \{J_1\})\}$. dr contains one tuple with a 2-1-relation which triggers the insertion of a new fork. Moreover, both nodes are scheduled in this iteration by adding scheduling dependencies from the newly created fork the node 8 and to node 9.

- *Final Join:* All nodes have been scheduled. The insertion of a final join starts to determine the set of nodes without outgoing scheduling dependencies, which is here $L = \{8, 9\}$ (line 98). The set contains more than one node, thus, a join must be inserted. Both nodes belong to the parallel region which is opened by fork F_4. Hence, the closing of this fork is initiated. A new join is created and node 8 and node 9 are connected to that join.

To this end, the resulting structured graph is given as

$$SG_9 = (\{1, 2, 3, 4, 5, 6, 7, 8, 9, F_1, J_1, F_2, J_2, F_3, J_3, F_4, J_4\},$$
$$\{(\text{immediate}, 1, 3), (\text{immediate}, 1, 4), (\text{immediate}, 1, 5),$$

$(\text{immediate}, 2, 6), (\text{immediate}, 2, 7), (\text{immediate}, 3, 8),$
$(\text{immediate}, 4, 8), (\text{immediate}, 5, 9), (\text{immediate}, 6, 9),$
$(\text{sched}, F_1, 1), (\text{sched}, F_1, 2), (\text{sched}, 1, F_2),$
$(\text{sched}, 2, F_3), (\text{sched}, F_2, 3), (\text{sched}, F_2, 4),$
$(\text{sched}, F_2, 5), (\text{sched}, F_3, 6), (\text{sched}, F_3, 7),$
$(\text{sched}, 3, J_2), (\text{sched}, 4, J_2), (\text{sched}, 5, J_2),$
$(\text{sched}, 6, J_3), (\text{sched}, 7, J_3), (\text{sched}, J_2, J_1),$
$(\text{sched}, J_3, J_1), (\text{sched}, J_1, F_4), (\text{sched}, F_4, 8),$
$(\text{sched}, F_4, 9), (\text{sched}, 8, J_4), (\text{sched}, 9, J_4)\})$

3.1.2 Synthesis of Structured Graphs to OpenMP

The synthesis of a structured graph SG to a parallel API for SMPs is straightforward. A recursive function walks through the structure and creates code for sequences of actions. Whenever a fork is reached, concurrent code can be created for the paths of the fork. The actual syntax to create parallelism using a specific API depends on the API itself. Concurrent paths in a structured graph, e. g., paths of a single fork, do not communicate. Hence, no communication is necessary between different paths of a concurrently running parallel region. A parallel region is closed by a join, which represents a synchronization point. To continue the execution, this point must be reached by all paths of the corresponding parallel region, e. g., by implementing a join using a barrier. Depending on the actual system, the barrier might call a `flush` instruction to ensure that all cached changes are visible in a system (see Section 6.4). The control-flow continues after the join.

The pseudo code to particularly synthesize a structured graph to OpenMP is given in Figure 3.21. We assume that a function to print an SGA is given as PrintSGA (see line 23). The creation of code is initiated by calling function SGtoOpenMP (line 1), which uses the function CreateCode to do the actual printing of code (line 3, 5). Function CreateCode recursively traverses the structured graph. Concurrent code is created from parallel regions as follows (line 7): OpenMP describes code that can run concurrently as parallel section. Thus, a parallel region is translated as parallel sections (line 9). Each path of the parallel region is then translated as one section (lines 11-15), thereby recursively descending into CreateCode (line 14). For that reason, the recursion stops at join nodes, which mark the end of a parallel region (line 20). The control-flow in a structured graph may continue after a join, which must be handled explicitly in the synthesis process (lines 17-19). The successor of the join is determined, which may be no or one successor (line 17). If a successor is available, then the code creation descends again, i. e., it recursively calls CreateCode (line 19).

The creation of code for an SGA node is done by printing the code for the given SGA (line 23) and creating code for its potential successor, which can be exactly zero or one (see requirement 1 of Definition 12).

```
1: function SGtoOpenMP(V, E) {
2:   {root} := {n | n ∈ V ∧ numPre(n) = 0}
3:   CreateCode(V, E, root)
4: }
5: function CreateCode(V, E, n) {
6:   suc = {v | v ∈ V ∧ (sched, n, v) ∈ E}
7:   if (n = Fork) then
8:     {j} := {j | j ∈ V ∧ matchFJ(n, j)}
9:     Print("#pragma omp parallel sections")
10:    Print("{")
11:    forall s ∈ suc
12:      Print("#pragma omp section")
13:      Print("{")
14:      CreateCode(V, E, s)
15:      Print("}")
16:    Print("}")
17:    suc_J := {v | v ∈ V ∧ (sched, j, v) ∈ E}
18:    forall s ∈ suc_J
19:      CreateCode(V, E, s)
20:  elseif (n = Join) then
21:    {} // stop recursion
22:  else
23:    PrintSGA(n)
24:    forall s ∈ suc
25:      CreateCode(V, E, s)
26: }
```

Figure 3.21: Pseudo code for printing a structured graph to OpenMP code. Function PrintSGA prints code for the given SGA and is assumed to be given.

3.2 SmpSS

Section 2.3.2.2 gave a brief introduction to the source-to-source compiler SmpSS from StarSS [148]. It allows one to execute tasks according to their data dependencies. That is, SmpSS provides a task-based programming environment to execute DFGs. In contrast to OpenMP, it does not require an explicit control-flow structure to declare parallelism and sequentiality, respectively. The downside of SmpSS is that SGAs writing to the same variable cannot be executed in parallel, which is due to the data-flow-driven MoC.

An ADG allows several nodes to write to a variable, a DFG requires the writer of a variable to be uniquely determined. Moreover, the synchronous MoC allows only one SGA per variable to be activated per macro step. Hence, SGAs that target the same variable can be executed concurrently because they cannot have a write conflict by construction.

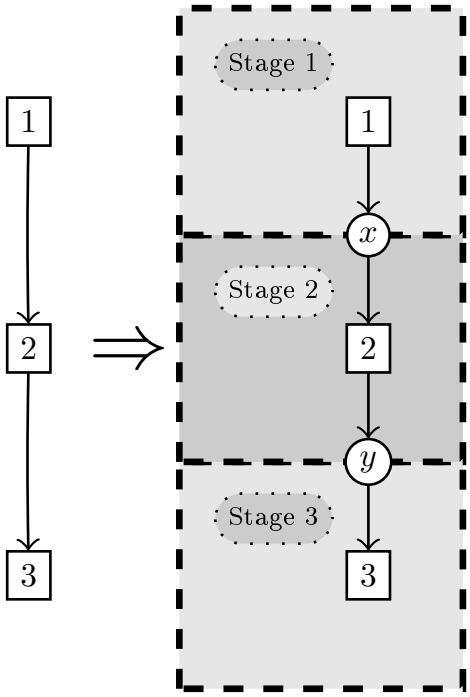


Figure 3.22: Principle of alternative approach to structuring algorithm: insertion of buffers (circle nodes) allows pipelined execution.

In a DFG, this cannot be expressed in such a way. Hence, to create code from an ADG for SmpSS, we need to transform a given ADG to a DFG. This is accomplished by grouping all SGAs that write to the same variable to a single node. We neglect on a formalization of this transformation because it is also necessary to create DPNs, which is done in the following chapter (see Chapter 4).

In SmpSS, a single task is defined as a C-function, i. e., each node of the DFG is translated as a single function. Moreover, the dependencies are defined as a compiler statement immediately before the task description. Finally, the main part of an SmpSS program is similarly build to the loop given in Figure 2.2. The actual execution of the SGAs is then done by another compiler statement that enables the execution in data-flow order and calling all tasks. To this end, the execution in data-flow order provides a more natural way to execute an ADG.

3.3 Closing Remarks

This chapter presented two approaches to create code for task-based execution of synchronous guarded actions. They start from an ADG description and use available parallelism by running independent nodes concurrently. The downside of this approach is the restriction to the available parallelism of a single macro step. An approach to counteract sequentializing dependencies has been presented in [17], which applies the idea of pipelin-

ing. The big idea of this approach is to split a graph vertically as shown in Figure 3.22. Similar to the insertion of pipeline registers to store intermediate results, variables are placed where dependencies cut the border of two stages. A more general variant is to use buffers instead of intermediate variables, where single-sized buffers are sufficient. It is also reasonable to still use available parallelism to combine parallelism and pipelining. This leads to our next approach: transforming ADGs to DPNs.

Chapter 4

Message-Passing Parallelism from Synchronous Guarded Actions

This chapter presents a translation from synchronous systems to DPNs [15, 18], thereby bridging the gap between two fundamentally different MoCs. This makes it easier to map synchronous systems onto distributed target architectures.

The main advantage of DPNs compared to task-based execution as shown in Chapter 3 is the usage of buffers between nodes. This allows all nodes to run decoupled and enables the processing of multiple macro steps at the same time. In contrast, the task-based approach considers only parallelism in a single macro step.

We propose to use DPNs as an abstract intermediate representation before final synthesis to concrete data-flow languages or multi-threaded software. On the one hand, this separates concerns, since the conceptual part, the switch of the MoC, is clearly separated from the technical part which deals with the concrete APIs or differences on the target platforms. On the other hand, this also gives us the possibility to reuse this translation as a common step for various architectures. In particular, we aim at creating multi-threaded software using PThreads for SMP systems and MPI for distributed memory architectures.

The usage of buffers in DPNs requires explicit transportation of data, and therefore, reduction of data transportation becomes an important key to get performance. Our targeted APIs provide easy creation of parallel code but they cannot hide communication effort [25]. We present two optimizations to reduce the communication. In particular, the first approach refers to a specific reduction for array structures [19] and the latter is a general reduction for scalar variables [14].

The rest of this chapter is structured as follows: Section 4.1 explains the basic transformation of SGAs to DPNs. The two optimizations to reduce the communication are given in Section 4.2 and Section 4.3. Finally, Section 4.4 gives a short discussion about the creation of code.

4.1 Translating Synchronous Guarded Actions to Data-Flow Process Networks

The starting point of our translation is a synchronous system represented by SGAs as described in Section 2.1.2. In the following, we assume that we have a set of guarded actions $\mathcal{G}$ which are defined over the set of variables $\mathcal{V}$. The goal of our translation is the creation of a DPN which will work as described in Section 2.2. In particular, each node of the DPN will be activated if all input buffers contain at least one token. If an activated DPN fires, it reads from all input buffers one token, processes them and places exactly one token in every output buffer.

Our translation is divided into several steps that are described in the following. Section 4.1.1 describes some initial preprocessing steps on the guarded actions, before Section 4.1.2 shows how DPN nodes are created from the guarded actions. Section 4.1.3 focuses on the dependencies between these nodes, and Section 4.1.4 discusses the creation of firing rules. The final step is the partitioning of the resulting DPN which is described in Section 4.1.5.

$$\left[\begin{array}{rcl@{\qquad}rcl} s & \Rightarrow & \texttt{next}(x) = a & \mathsf{true} & \Rightarrow & c = z \cdot z \\ \mathsf{true} & \Rightarrow & \texttt{next}(r) = s & \neg r & \Rightarrow & x = p \\ \mathsf{true} & \Rightarrow & \texttt{next}(o) = a \cdot b & s & \Rightarrow & y = q \\ \mathsf{true} & \Rightarrow & a = x + y & \neg s & \Rightarrow & y = o \\ \mathsf{true} & \Rightarrow & b = x - y & \mathsf{true} & \Rightarrow & m = b + c \end{array}\right]$$

Figure 4.1: *Example:* Synchronous Guarded Actions

Figure 4.1 lists an example for SGAs which does not represent a meaningful algorithm, but which will serve as a running example in the rest of this section. The set is defined over the inputs s, p, q and z (which can only be read), the output o (which is exposed at the system interface), and some other local variables. It contains three delayed actions writing x, r and o, and seven immediate actions. The storage type of the variables, i. e., whether they are event or memorized variables, only affects the default behavior. Since all variables are written in each macro step, the storage type does not matter in this example.

4.1.1 Preprocessing the Synchronous System

As already stated above, we start with a synchronous system represented by a set of guarded actions $\mathcal{G}$ defined over a set of variables $\mathcal{V}$. In general, several guarded actions write a variable $x \in \mathcal{V}$, and this set of actions may contain immediate *and* delayed actions. In particular, assignments of these actions target different macro steps for the same variable. Since this would complicate the translation presented in the following, we first normalize the set of guarded actions so that each variable is *either* written by immediate *or* delayed actions.

This can be accomplished by applying the function SeparateDAnIA printed in Figure 4.2 to a given set of SGAs. We introduce an auxiliary variable x' (sometimes called *carrier variable*) for each x (line 15) which is written by immediate and delayed actions (line 14). While the immediate actions still write the original variable x (line 21), all delayed actions are redirected to the new variable x' (lines 16ff). As we need to know whether a delayed action has written some value to x', we additionally need to track the guards in x'_{guard} of all delayed actions writing x' so that we know when to transfer the value of x' to x (lines 17 and 20).

As explained at the beginning of this section, a DPN node will always read a token from each incoming buffer whether or not the value is required for the calculations. Hence, we have to ensure that for every variable in every step a value has to be set by a guarded action. This must be also done for the auxiliary variables, which is accomplished by Lines 19 and 20 in function SeparateDAnIA.

```
 1: function RenameLHS(𝒢, lhs, lhs′)
 2:   𝒢′ := {}
 3:   forall G ∈ 𝒢
 4:     case G :
 5:     γ ⇒ next(lhs) = τ :
 6:       𝒢′ := 𝒢′ ∪ ⟨γ ⇒ lhs′ = τ⟩
 7:     default :
 8:       𝒢′ := 𝒢′ ∪ G
 9:   return (𝒢′)
10: function SeparateDAnIA(𝒱, 𝒢)
11:   𝒢′ := {}
12:   𝒱′ := 𝒱
13:   forall x ∈ 𝒱
14:     if (wrNowActs(x) ∩ wrNxtActs(x) ≠ {})  then
15:       𝒱′ := 𝒱′ ∪ {x′, x′_guard}
16:       g′ := ⋃{γ | 𝒜 ∈ wrNxtActs(x) ∧ 𝒜 = γ ⇒ A}
17:       G_guard,T := ⟨g′ ⇒ next(x′_guard) = true⟩
18:       G_absence := ⟨¬g′ ⇒ next(x′) = x_Default⟩
19:       G_guard,F := ⟨¬g′ ⇒ next(x′_guard) = false⟩
20:       G_copy := (x′_guard ⇒ x = x′)
21:       𝒢′ := 𝒢′ ∪ wrNowActs(x)
22:         ∪RenameLHS(wrNxtActs(x), x, x′)
23:         ∪{G_guard,T, G_guard,F, G_copy}
24:     else
25:       𝒢′ := 𝒢′ ∪ wrNowActs(x) ∪ wrNxtActs(x)
26:   return (𝒱′, 𝒢′)
```

Figure 4.2: Preprocessing of Synchronous Systems

If we apply the function SeparateDAnIA to our running example given in Figure 4.1, we obtain the guarded actions shown in Figure 4.3. Obviously, only variable x is written by immediate *and* delayed actions. Hence, only for that variable, auxiliary variables x' and x'_{guard} are inserted. The delayed action which writes x is redirected to x', and we set x'_{guard} to true if and only if the delayed write takes place. Finally, an action is added to copy the result of a delayed write if it has taken place. Note that the system behavior has not changed, and the modified system cannot be distinguished from the original one by the environment.

$$\left[\begin{array}{rcl@{\qquad}rcl}
s & \Rightarrow & \texttt{next}(x') = a & \mathsf{true} & \Rightarrow & a = x + y \\
\neg s & \Rightarrow & \texttt{next}(x') = 0 & \mathsf{true} & \Rightarrow & b = x - y \\
s & \Rightarrow & \texttt{next}(x'_{\text{guard}}) = \mathsf{true} & \mathsf{true} & \Rightarrow & c = z \cdot z \\
\neg s & \Rightarrow & \texttt{next}(x'_{\text{guard}}) = \mathsf{false} & \neg r & \Rightarrow & x = p \\
x'_{\text{guard}} & \Rightarrow & x = x' & s & \Rightarrow & y = q \\
\mathsf{true} & \Rightarrow & \texttt{next}(r) = s & \neg s & \Rightarrow & y = o \\
\mathsf{true} & \Rightarrow & \texttt{next}(o) = a \cdot b & \mathsf{true} & \Rightarrow & m = b + c
\end{array}\right]$$

Figure 4.3: *Example:* Synchronous Guarded Actions after Preprocessing

4.1.2 Creating DPN Nodes

The next step of our translation will transform the synchronous system to a DPN. Thereby we basically map the guarded actions $\mathcal{G}$ to nodes of the DPN, and the variables $\mathcal{V}$ to the FIFO buffers in between. In the following, we first consider the nodes, before the next subsection describes the creation of the buffers in between.

As the buffers of a DPN are point-to-point connections, they all have a single writer and a single reader for each variable. However, even after the preprocessing of the previous section, each variable still may be written by several actions.

To handle several readers of a variable, the buffer representing a variable $x \in \mathcal{V}$ needs to be split up into a buffer receiving new tokens for x, several other buffers providing the new values and a duplicator node in between. In the following, for the sake of simplicity, we assume that the DPN has implicit duplicators, i. e., we allow several nodes to read the same variable.

In contrast, several writers of a variable would have to write to a merging node that merges tokens from several incoming buffers in order as described in [15]. This introduces additional computational and data overhead: the merging node and the need for time stamps make this solution unattractive. Instead, we have to group the guarded actions before the translation to a DPN node such that each buffer is written by a single node.

The crucial step is to group all actions that write to the same variable (see Figure 4.4). To this end, we first create a DPN node with a single empty firing rule (thus containing no actions) for each writable variable and subsequently add all actions that write to a variable to the firing rule of the corresponding DPN node. The guard γ_B for this firing rule is

preliminarily set to true but it will be improved in the rest of the translation. Recall that the previous separation of immediate and delayed actions to the same variable will lead to a creation of two separate nodes for a single program variable x.

```
1: function CreateDPNNodes(V, G)
2:   N := {}
3:   forall x ∈ V
4:     if (wrNowActs(x) ≠ {})  then
5:       n := ⟨true ⇒ wrNowActs(x)⟩
6:       N := N ∪ new DPNNode(n)
7:     elseif (wrNxtActs(x) ≠ {})  then
8:       n := ⟨true ⇒ wrNxtActs(x)⟩
9:       N := N ∪ new DPNNode(n)
10:  return (N)
```

Figure 4.4: Creating DPN Nodes

4.1.3 Creating DPN Buffers

After the nodes of the DPN have been created, we have to set up the communication infrastructure by adding FIFO buffers to the network. These buffers basically need to be placed at two positions: (1) between nodes exchanging data and (2) at the system interface to pass the inputs and outputs of the system.

In our representation of DPNs we describe the sets of buffers by their characteristic functions. To this end, we define two function $\mathsf{hasImEdge}(n_1, n_2)$ and $\mathsf{hasDeEdge}(n_1, n_2)$ which return true if the DPN contains a buffer that obtains data from node n_1 and makes it available to node n_2. Similarly, we define a function $\mathsf{inbounds}(n)$ to describe the buffers read by a node and two functions $\mathsf{imOutbounds}(n)$ and $\mathsf{deOutbounds}(n)$ to describe the buffers written by a node. These definitions will later help us to define the buffers at the system interface.

For the internal and the output buffers, we distinguish buffers that are due to immediate and delayed actions. While both variants indicate that a buffer will be placed between the nodes, we have to distinguish them since they need to be initialized differently: the delayed variant must initially contain a token representing the default value for that variable so that its tokens are assigned to the right step.

In order to find the places where internal buffers must be placed, we have to analyze the data dependencies between the guarded actions. Basically, whenever a variable x is written in node n_1 and read by another node n_2, an edge has to be created from n_1 to n_2. We therefore refer to Definition 1 from Section 2.1.2 and lift it to DPN nodes, i. e., we define the variables that are read or written by the firing rules of a node n. Similar to the guarded actions, we distinguish immediate and delayed write accesses so that we define two functions $\mathsf{wrNowVarsDPN}(n)$ and $\mathsf{wrNxtVarsDPN}(n)$.

Definition 14 (Node Dependencies). *Let* $\mathsf{behavior}(n) \subseteq \mathcal{G}$ *be all guarded actions of a single DPN node* $n \in \mathcal{N}$*. Then, the dependencies from the DPN node* n *to variables are defined as follows:*

$$\begin{aligned}
&\mathsf{rdVarsDPN}(n) := \textstyle\bigcup_{G \in \mathsf{behavior}(n)} \mathsf{rdVars}(G) \\
&\mathsf{wrNowVarsDPN}(n) := \textstyle\bigcup_{G \in \mathsf{behavior}(n)} \mathsf{wrNowVars}(G) \\
&\mathsf{wrNxtVarsDPN}(n) := \textstyle\bigcup_{G \in \mathsf{behavior}(n)} \mathsf{wrNxtVars}(G) \\
&\mathsf{wrVarsDPN}(n) := \mathsf{wrNowVarsDPN}(n) \cup \mathsf{wrNxtVarsDPN}(n)
\end{aligned}$$

Based on $\mathsf{rdVarsDPN}()$ and $\mathsf{wrVarsDPN}()$, we can define the set of DPN nodes that read from, respectively write to a variable in Definition 15. Note that, according to the definition of DPNs, a variable can only be written by at most one node, hence $|\mathsf{writer}(v)| = 1$ holds.

Definition 15 (Reading and Writing DPN nodes). *The set of DPN nodes reading and writing to a variable* v *is determined as follows:*

$$\begin{aligned}
&\mathsf{reader}(v) := \{n \mid \mathsf{behavior}(n) \cap \mathsf{rdActs}(v) \neq \{\}\} \\
&\mathsf{writer}(v) := \{n \mid \mathsf{behavior}(n) \cap \mathsf{wrActs}(v) \neq \{\}\}
\end{aligned}$$

Thus, $\mathsf{reader}(v)$ is the set of nodes reading variable v, while $\mathsf{writer}(v)$ is the singleton set of nodes that writes values to variable v.

$$\begin{aligned}
&\mathsf{hasImEdge}(n_1, n_2) := n_1 \neq n_2 \wedge (\mathsf{wrNowVarsDPN}(n_1) \cap \mathsf{rdVarsDPN}(n_2)) \neq \{\} \\
&\mathsf{hasDeEdge}(n_1, n_2) := (\mathsf{wrNxtVarsDPN}(n_1) \cap \mathsf{rdVarsDPN}(n_2)) \neq \{\} \\
&\mathsf{inbounds}(n) := \{n_S \in \mathcal{N} \mid \mathsf{hasImEdge}(n_S, n) \vee \mathsf{hasDeEdge}(n_S, n)\} \\
&\mathsf{imOutbounds}(n) := \{n_T \in \mathcal{N} \mid \mathsf{hasImEdge}(n, n_T)\} \\
&\mathsf{deOutbounds}(n) := \{n_T \in \mathcal{N} \mid \mathsf{hasDeEdge}(n, n_T)\}
\end{aligned}$$

Figure 4.5: Creating DPN Buffers

Now we are in the position to define the functions mentioned above (see Figure 4.5), which are almost straightforward with the previous definitions. However, the clause $n_1 \neq n_2$ in the first line may not be obvious: To understand this restriction, consider a DPN node which reads x and also contains a delayed action writing to the same variable x. It requires an intermediate variable to store the result of a delayed write, such that the value of the current macro step is not overwritten and other actions are able to read it. This is easy to implement by adding a buffer where the current value is read at the beginning of a calculation and is written during the execution of a firing rule's guarded actions. This buffer is exactly the same that is implemented when a delayed edge is added to the graph from the concerned node to itself.

In contrast, the result of an immediate write access must be available in the same macro step, i. e., in the same iteration. Using a buffer would result in a deadlock. To execute a single iteration, the node would have to read a value that is going to be written in the same

iteration by the same node. To this end, in a DPN, there is no distinction between buffers that contain immediately or delayed written variables. But it is important to consider this definition for the final synthesis process as demonstrated in Section 4.4.

With the definitions of the nodes in the previous section and the buffers in this section, we can now present the DPN of our running example. Figure 4.6 shows the resulting network.

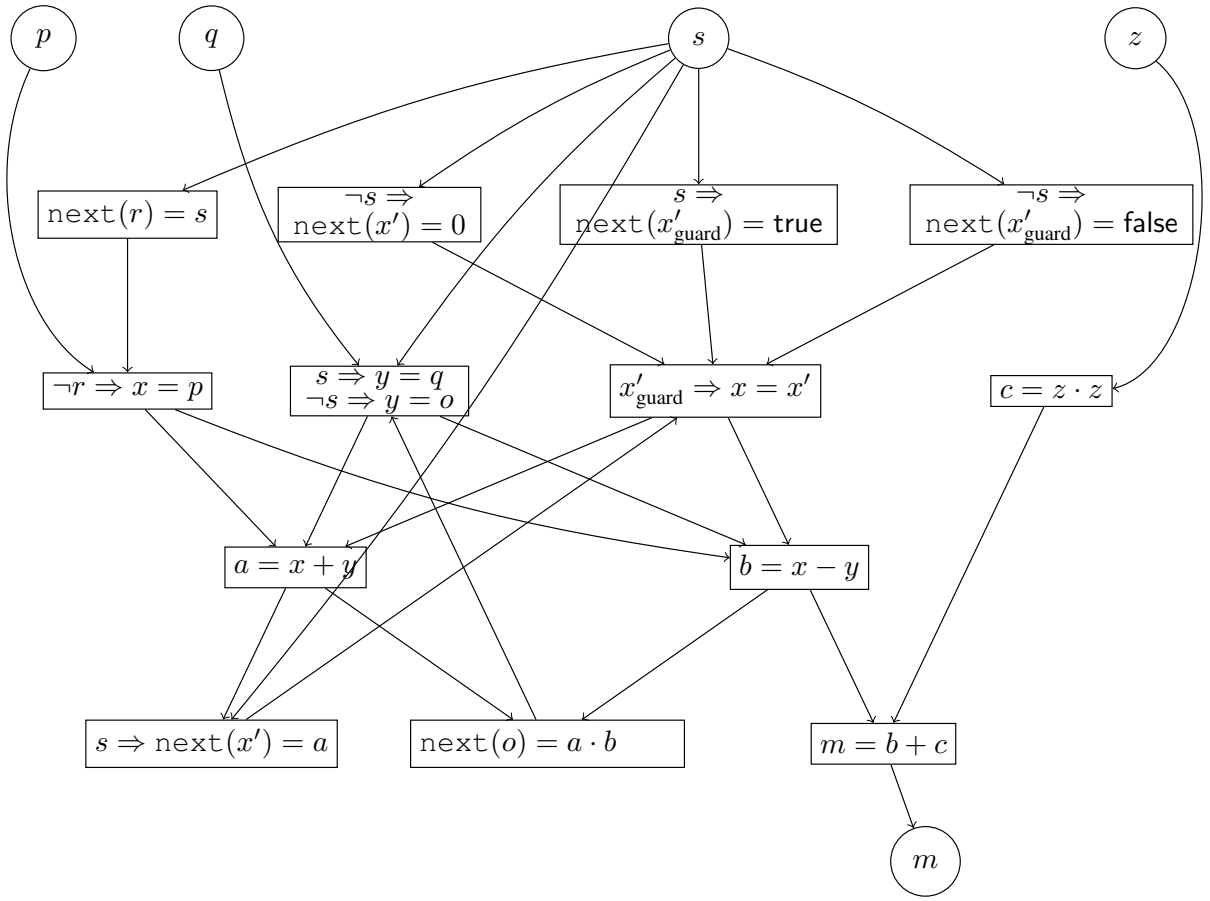


Figure 4.6: *Example:* DPN created by the translation

4.1.4 Determining the Guards of Firing Rules

After the creation of the DPN and determining the dependencies between the nodes, we can now determine the guards of the firing rules. As mentioned at the beginning of this section, each DPN node has exactly one firing rule that has to be activated as soon as all input buffers contain at least one token. In our implementation, we use the flag `present` (v) to check whether v contains a token or not. Figure 4.7 shows the code for assigning the guards to the firing rules. Note that a guard of a firing rule is not restricted to a conjunction of these flags, but can be any Boolean expression.

```
1: function DetermineGuardOfFR(𝒩)
2:   forall n ∈ 𝒩
3:     γ_B := ⋀_{v∈rdVarsDPN(n)} present (v)
4:     n := ⟨γ_B ⇒ behavior (n)⟩
5:   return (𝒩)
```

Figure 4.7: Function to determine the firing rule for each DPN node.

4.1.5 Post-Processing the DPN

At that point, we already have a DPN for a given synchronous system. Hence, we can use the created model to feed it into other tools, e. g., we could generate C code using PThreads or MPI (see Section 4.4). However, we could optimize the DPN as generated by the previous sections in a final step of our translation. Obviously, this post-processing does not change its behavior.

A post-processing of the DPN makes sense if subsequent synthesis steps require coarse-grained parallelism, i. e., nodes with a larger number of actions. As an example, we recall that creating multi-threaded code requires each thread to contain a minimum amount of computational effort such that the synchronization between threads is amortized. A higher computational effort can be reached by putting more actions into a thread. Another motivation comes from the fact that sometimes the number of available resources is known in advance so that a static clustering of nodes reduces the run-time overhead.

Due to this reason, it is interesting to partition a given DPN and to merge nodes of each partition to a single DPN node. In the following, we will not discuss strategies how to gain a *good* partitioning since the choice of the best algorithm heavily depends on the final architecture and its requirements. Partitioning requires a static analysis, which is neglected for the same reason as in Section 3.1: "a static analysis and rating of the benefits of a node duplication requires precise knowledge about the computational effort of a node which depends on the targeted hardware. This conflicts with our principle to make decisions about the actual target as late as possible." There is a large number of papers which exactly addresses these problems, e. g., to map a DPN to a MPI cluster, one could use the Ford-Fulkerson's algorithm [56] to get the min-cut/max-flow of the DPN to minimize communication between the resulting nodes.

In the following, we define what a valid partitioning is, which gives rise to the correctness of a network post-processing.

1. Each partition consists of a set of DPN nodes that can be merged to a single DPN node. This condition guarantees that the result of the partitioning step is a new DPN. In particular, merging of DPN nodes $n_1, n_2, \ldots, n_k$ in this context is done in four steps:

   (a) create a new DPN node n and add it to the DPN

(b) $\gamma_B(n) := \bigwedge_{i=1\ldots k} \gamma_B(n_i)$

(c) $\mathsf{behavior}\,(n) := \bigcup_{i=1\ldots k} \mathsf{behavior}\,(n_i)$

(d) remove $n_1, \ldots, n_k$ from the DPN

Note that due to the implicit definition of edges by the DPN nodes, there is no need for redirecting or creating edges.

2. A partition creates a set of groups and a bijective mapping of the created groups to DPN nodes. In particular, the sets of DPN nodes represented by the groups must be disjunctive. Furthermore, DPN nodes must not be duplicated or removed.

3. After merging all DPN nodes of each group to a single DPN node, the resulting graph must not contain loops that contain immediate edges. Loops containing at least one delayed edge are allowed because delayed write accesses refer to the next macro step, and therefore, these loops do not lead to a deadlock of the DPN.

These conditions are formally defined as the term *legal partition* in Definition 16.

Definition 16 (Legal Partition of a DPN)**.** *A* partition Π *of a DPN is a mapping from DPN nodes to groups* $\pi \in \Pi$*. Let* $\mathsf{partition}(A)$ *denote the group of a DPN node* $n \in \mathcal{N}$*, and let* $\mathsf{nodes}(\pi)$ *denote all the DPN nodes occurring in group* π*. A partition is* legal *iff all of the following points are fulfilled:*

1. **forall** $\pi_0, \pi_1 \in \Pi . \pi_0 \neq \pi_1 \rightarrow \mathsf{nodes}(\pi_0) \cap \mathsf{nodes}(\pi_1) = \{\}$

2. *Let* $\sqsubseteq$ *be the reflexive and transitive closure of the following relation* $R \subseteq \mathcal{N} \times \mathcal{N}$*:* $(n_1, n_2) \in R \Leftrightarrow n_1 \neq n_2 \wedge \mathsf{wrNowVars}\,(n_1) \cap \mathsf{rdVars}\,(n_2) \neq \{\}$*, then* $\sqsubseteq$ *is a partial order.*

4.1.6 Communication Optimizations

The DPN model uses communication through FIFO buffers to run its node decoupled. The one side of the coin is more parallelism compared to the previously introduced task-based execution (see Chapter 3) and the ability to run theses applications in clusters, i. e., systems with distributed memory. The other side of the coin is the more expensive communication: communication through FIFO buffers introduces additional explicit memory copy operations. Therefore, transportation of data between DPN nodes plays an important role to gain applications that perform well. For that reason, we introduce two optimization methods in the following sections. Section 4.2 tackles the expensive communication of arrays between nodes. Afterwards, Section 4.3 introduces a more general optimization for scalar values.

4.2 Efficient Array Transport

While the previous synthesis method works well for scalar data, the DPN programming model becomes expensive when compound data types have to be dealt with. For example, compound data types are trees, lists, dictionaries, or what is more important for embedded applications: arrays. Recall that process nodes implement mathematical functions; thus, they map input values to output values. Consequently, if a DPN deals with arrays, then the nodes writing values to and reading values from the array have to send the entire array from one node to the other, which clearly leads to high communication costs. Note that it is no solution to augment the DPN with a shared memory, since this will lead in most cases to a non-deterministic behavior due to the process nodes' independent executions (reading from and writing to shared data values will then probably suffer from data races).

In this section, we therefore consider the problem to reduce the communication costs in DPNs that deal with arrays. The main idea is thereby that we define for each array of the given program one process node (called the writer of the array) in the DPN that is responsible for all write updates of the array. Each node reading the array will maintain a local copy of the array and will receive from the writer of the array the corresponding updates. While the array may be large, the updates typically contain only a few assignments to array elements instead of the entire array. Depending on the application, the presented approach *automatically* reduces the amount of data that has to be sent through a DPN's communication channels. It is not difficult to see that our proposed algorithm is correct, so that we are able to automatically translate synchronous programs to deterministic DPNs having this optimized use of arrays.

Typical examples of applications that benefit from this approach are all kinds of embedded systems that deal with arrays. In particular, this covers multimedia applications or, more general, applications that deal with digital signal processing.

```
 1: module AudioDelay(nat ?i, ?delay, bool ?c, nat o) {
 2:   [N]nat a; // store the last N values of i
 3:   nat {N}j0, j; // array indices
 4:   loop {
 5:     p : pause;
 6:     if (c) a[j] = i;
 7:     next(j) = (j + 1)%N;
 8:     j0 = (j − delay)%N;
 9:     o = a[j0];
10:   }
11: }
```

Figure 4.8: Module AudioDelay as Quartz Program.

During this section, we will use a modified variant of the audio delay from the benchmark chapter (see Chapter 7.1.4). It represents a practical example of a FIFO implemen-

```
 1: system AudioDelay :
 2:   interface :
 3:     i : input memorized nat
 4:     delay : input memorized nat
 5:     c : input memorized bool
 6:     o : inout memorized nat
 7:   locals :
 8:     p : label bool
 9:     a : local memorized [N]nat
10:     j : local memorized nat N
11:     j0 : local memorized nat N
12:   init :
13:     control flow :
14:       true ⇒ next(p) = true
15:     data flow :
16:   main :
17:     control flow :
18:       p ⇒ next(p) = true
19:     data flow :
20:       c&p ⇒ a[j] = i
21:       p ⇒ next(j) = (j + 1)%N
22:       p ⇒ j0 = (j − delay)%N
23:       p ⇒ o = a[j0]
```

Figure 4.9: Guarded Actions of Module AudioDelay.

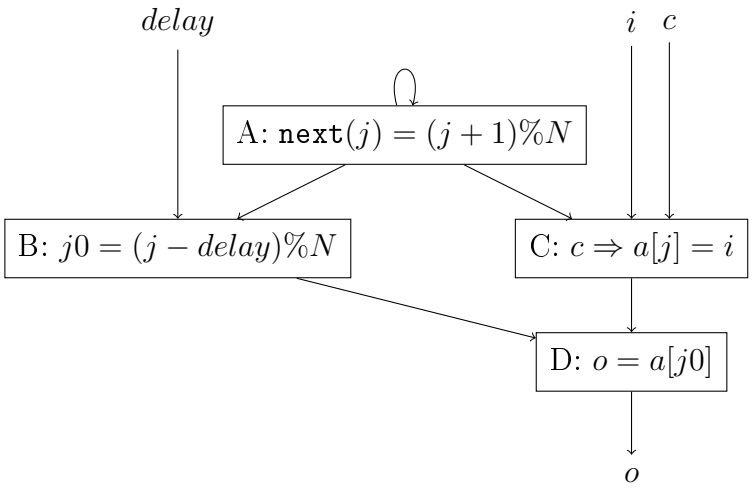


Figure 4.10: DPN of Audio Delay

tation. The code is depicted in Figure 4.8 and differs in an additional input from the environment: Variable c which controls whether an input sample is buffered. Its main purpose is to demonstrate the effect of guards in the presented array communication optimization. The set of SGAs for the given example is printed in Figure 4.9.

Starting from that description, we obtain the DPN shown in Figure 4.10 (we have omitted trivial guards that are true, which also pertains for label p in the example at hand). Note that the lines connecting the process nodes represent FIFO buffers. The orientation of the edges depicts the direction of the data-flow in these buffers, i. e., they are pointing from the node that produces tokens to the node that consumes tokens.

Recall also that in DPNs there is always exactly one producer and one consumer per FIFO buffer. The reader may have noticed that node A reads and writes variable j and that also nodes B and C read j. A copy of the initial value of j is in each buffer from A. Each time a value from the buffer from A to itself is consumed, it is incremented and three copies are produced in the buffers connecting A with A,B, and C.

4.2.1 Inefficiency due to Compound Data Types like Arrays

Efficient communication between DPN nodes is one of the main issues in mapping DPNs to concrete target architectures. If the data values that are sent between the process nodes are only of small scalar types like integers, etc., it is no problem to send and receive them via FIFO buffers. However, if the data types are large as in case of arrays, this becomes a major problem for the efficient implementation of DPNs. Recall that it is no solution to augment the DPN with a global shared memory because the independent execution of DPN nodes will suffer from data races that lead to nondeterminism. Another solution would be to automatically synthesize locks to shared memories which is also currently a research topic for weak memory models (see e.g. [44, 45, 81]), but very different to the approach considered here.

Consider again our example in Figure 4.10: Each time, node C is fired, one element in a is written, so that array a is modified. For this reason, the complete array a must be sent to node D by writing array a in the FIFO buffer that is represented by the edge between nodes C and D, such that D can read the correct element $a[j0]$ that must be sent to the environment.

Even this very simple example makes clear that this implementation is obviously unsatisfactory. In this example, we could simply merge nodes C and D to a single node to avoid the communication of the array. This is clearly a reasonable solution for this example, but in general this would impose hard restrictions to the partitioning of guarded actions to DPN nodes. In the following section, we discuss an algorithm that transforms the given DPN into another one that avoids the communication of the entire array, but does not modify the topology of the DPN.

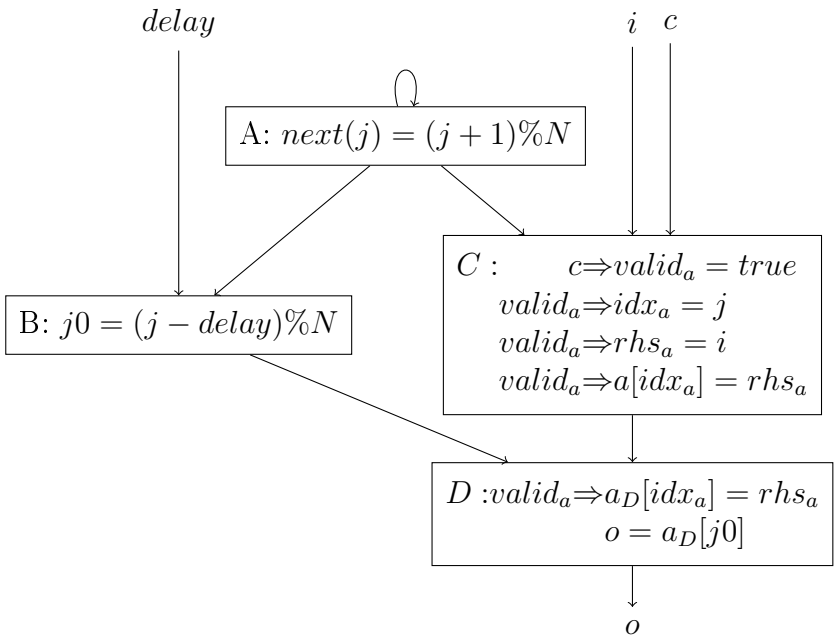


Figure 4.11: Optimized DPN of Audio Delay

4.2.2 Removing Arrays from Communication

Having explained the problem, we consider in this section an algorithm that can be used to reduce communication costs of arrays in DPNs that are generated from synchronous guarded actions. We first describe the main idea with the help of our running example, and then present the algorithm that automatically generates the optimized DPN.

4.2.2.1 Main Idea

The overall idea presented in this section is that arrays are no longer sent through the FIFO buffers. Instead, we distinguish between the (unique) writer of an array, and the readers of the array. Both the writer and the readers maintain local copies of the array, but only the writer holds the code for updating the array as defined in the guarded actions. Clearly, it will update its local copy by executing the code as before, but instead of sending the entire updated array to all readers, it will only send the required information to update the local copies of the readers.

For example, an accordingly optimized version of the DPN of Figure 4.10 is shown in Figure 4.11. Nodes C and D have been modified as follows: whenever array a must be updated, a signal $valid_a$ is used. Node C then generates values $valid_a$, idx_a for the index of an element of a that must be overwritten by value rhs_a. As before, node C performs this update $a[idx_a] = rhs_a$ on its local copy of a. Note that the computation of idx_a and rhs_a are typically more complex than in the considered example.

Instead of sending the updated value of a to node D, node C will now only send the values idx_a and rhs_a, so that node D will first update its local copy a_D of a by executing

$a_D[idx_a] = rhs_a$. The previous code to produce output o is changed accordingly, i.e., the value is now taken from the local array a_D instead of the now unknown array a.

4.2.2.2 Formal Foundation of the Transformation

After the informal description of the idea of our transformation, we will next define an algorithm to optimize the array communication of DPNs that have been obtained from SGAs: We consider only local variables for optimization since only local variables are communicated between nodes.

Recall that $\mathcal{N}$ is the set of the DPN nodes, and the guarded actions of a DPN's node n are denoted as $\mathsf{behavior}(n)$. Moreover, let $\mathcal{L}$ be the set of local variables of the DPN, i. e., all buffers except for buffers connecting the DPN with its environment. Conversely to definition of $\mathsf{behavior}(n)$, $\mathsf{owner}(A)$ is the name of a node that contains an action A (the node is uniquely determined since we *partition* the original set of guarded actions as DPN nodes).

Definition 17 defines the function to determine the set of SGAs in node n that write to variable v.

Definition 17 (Writing SGA in DPN node)**.** *The set of SGAs in a node n writing to variable v is determined as follows:*

$$\mathcal{W}_{\mathcal{A}}(v, n) := \mathsf{wrActs}(v) \cap \mathsf{behavior}(n)$$

Moreover, for any guarded action $\mathcal{A} = \langle \gamma \Rightarrow x = \tau \rangle$, the expression $\mathsf{grd}(\mathcal{A})$ denotes its guard γ, $\mathsf{lhs}(\mathcal{A})$ denotes its left-hand side expression x, and $\mathsf{rhs}(\mathcal{A})$ denotes its right-hand side expression τ. Finally, let $\mathsf{ReplaceVar}(\mathcal{A}, v, v')$ be the function that renames variable v to v' in the given actions $\mathcal{A}$, i. e., all occurences of variable v are replaced by the expression v'.

4.2.2.3 Array Communication Reduction Algorithm

The reduction of communication of arrays between nodes is done by function RMArrayCom shown in Figure 4.12. Function RMArrayCom thereby determines first the set of array variables $\mathcal{L}_A$ that have to be communicated from their writer to a reader (different to the writer). An improved algorithm may consider also whether a single variable is worth the transformation, which depends on the size of the array and the computations made for its updates. We then call function $\mathsf{RMArrayComOfVar}(v)$ to remove the array communication of array variable v. This call generates further variables that are collected and finally returned by RMArrayCom.

Each call $\mathsf{RMArrayComOfVar}(v)$ starts with the creation of a local copy v^n of the array v in each reader node n (lines 9–13). Then, each occurrence of v is replaced by the corresponding access to the local copy v_n of the node n. Since neither v nor one of its copies is communicated between nodes anymore, the algorithm has to change the set of actions that describe the behavior of the concerned nodes. In general, it is possible that a node reads its written variables. Hence, we must ensure that a copy of v is only added

```
1: function RMArrayCom(𝓛, 𝒩)
2:   𝓛_A = {v ∈ 𝓛 | v is array ∧ |writer(v) ∪ reader(v)| > 1}
3:   𝓛' = 𝓛
4:   forall v ∈ 𝓛_A
5:     𝓛' = 𝓛' ∪ RMArrayComOfVar(v)
6:   return 𝓛'

7: function RMArrayComOfVar(v)
8:   n := writer(v)
9:   // add copies of variable for reader nodes
10:  𝓛 = 𝓛' ∪ {v^r | r ∈ (reader(v) \ n)}
11:  // rename occurences of v in reader nodes
12:  forall r ∈ (reader(v) \ n)
13:    ReplaceVar(behavior(r), v, v^r))
14:  // update code in writer node
15:  forall A ∈ 𝒲_𝒜(v, n)
16:    𝓛' = 𝓛' ∪ {valid_{v,A}, ID_{v,A}, τ_{v,A}}
17:    behavior(n) = (behavior(n) \ A)∪
18:      {true ⇒ valid_{v,A} = grd(A)}∪
19:      {valid_{v,A} ⇒ τ_{v,A} = rhs(A)}∪
20:      {valid_{v,A} ⇒ ID_{v,A} = ID(lhs(A))}∪
21:      {valid_{v,A} ⇒ v[ID_{v,A}] = τ_{v,A}}
22:    // update code in reader nodes
23:    forall r ∈ (reader(v) \ n)
24:      insertCC(A(r),
25:        {valid_{v,A} ⇒ v^r[ID_{v,A}] = τ_{v,A}})
26:  return 𝓛'
```

Figure 4.12: Algorithm for removing/reducing array communication.

for and renamed in reading nodes that do not write to v, i. e., n must be excluded from $\mathsf{reader}(v)$ (lines 10, 12).

The remaining part of RMArrayComOfVar adds the transportation of changes by adding update tuples and corresponding actions. To ensure correctness, each change will be sent using a separate tuple containing the change. Because we use the synchronous MoC, all changes to the copy v^n of v may be evaluated at the same time. The parallel evaluation involves a potentially parallel access to the data structure that is used to transport changes. Hence, to avoid races, each change is stored in a separate update tuple.

Each update tuple consists of a valid flag, an index and the value. The valid flag *valid* indicates whether an update has to be made. If the valid flag is set, the index *ID* identifies

the element that has to be changed, and rhs() contains the value that must be assigned to the copy of v.

In addition, our implementation omits the communication of array writes if the written cell already contains the value to be assigned. In particular, for each assignment to the copied variable, a tuple for the change is created (line 16). The first action (line 18) stores the state, whether the assignment to v^n has to be executed - and therefore, to be communicated to reading nodes. The following two actions (line 19, 20) copy the right-hand side expression and the index of the element to the corresponding members of the tuple. In order to avoid duplicate evaluation of the right-hand side expression and the element identifier, we remove the original action (line 17) and replace it by an action that uses the members of the tuple (line 21).

Analogously, each reader obtains a similar action to commit the change to its local copy of v (line 24, 25). The action has to be inserted using the insertCC(behavior (n), A) function which is defined as behavior $(n) =$ behavior $(n) \cup A$.

4.3 General Communication Reduction

The following approach adds a communication optimization that is applied to all types of variables [14], while the previous optimization considers only array variables (see Section 4.2).

4.3.1 Relaxing Communication

Data values that are communicated through FIFO buffers will be at least *sometimes* but not necessarily always used by the consumer nodes. The *'sometimes'* is the key for communication reduction. The principle idea of the general communication reduction is to send a value from one node to another only if it is needed for the calculations of the target node. The actual necessity of a value for the program's outputs (which is actually of interest) is impossible or at least hard to ascertain because these values may also influence the behavior in the future, which can neither be detected at compile time nor at runtime without extensive effort. Nevertheless, they can be estimated [40]. Moreover, we are able to ascertain whether a variable influences the calculations that directly depend on that variable.

In the following, we use the term transport, formally $\zeta(x)$, as the description of the communication of a single value of variable x between two nodes, the writing node (*sender*), formally $\zeta_{\mathrm{wr}}(x)$, and a reader node (*receiver*), formally $\zeta_{\mathrm{rd}}(x)$. The respective antagonist of a transport will be called *partner node*, i. e., the partner node of the sender is the receiver and vice versa. To this end, let $\mathcal{T}$ be the set of all transports in a given DPN.

Transports can be saved by considering changes of variable values. In particular, the default action which defines the default value of a variable in a macro step, is a quite trivial action that either assigns a constant or just keeps the value from the previous macro step. Hence, executing the default action on the local copy of the consumer node, instead of

invoking a transport, will have the desired effect. In conclusion, we require each node to have a local copy for each variable that is written by another node. This local copy will either contain the up-to-date value or the symbol $\perp$ to signal that the local copy is out-of-date. In the following, *Bubble* will be the term for a value of a transport that is a non-required value or the result of the default action.

In the first part of our optimization, we assume that our optimization is based on a *per transport* analysis, i. e., for each transport $\zeta(x) \in \mathcal{T}$.

4.3.1.1 True Writes and True Reads

We assume that the nodes of the DPN contain SGAs, i. e., conditional assignments. Hence, we can determine when a variable is modified and when it is actually read by a node. If a value of a variable is changed by its writer or read by its reading node, it is referred to as *true write* or *true read*, respectively. Formally:

Definition 18. *The condition for a* true write *corresponds to the overall write guard for variable x and is defined as*
$\gamma_{wr}(x) := \bigvee\{\gamma \mid \mathcal{A} \in \mathsf{behavior}\,(N_{wr}(x)) \wedge x \in \mathsf{wrVars}\,(\mathcal{A}) \wedge \mathcal{A} = \langle \gamma \Rightarrow A \rangle\}$.

The condition for a true read *corresponds to the overall read guard for variable x in node $n \in N_{rd}(x)$ and is defined as*

$$\gamma_{rd}(x,n) := \begin{cases} \mathsf{true}, \textit{ if } \exists_{\langle \gamma \Rightarrow A \rangle \in \mathsf{behavior}(n)}.x \in \mathsf{Vars}(\gamma) \\ \bigvee\{\gamma \mid \mathcal{A} \in \mathsf{behavior}\,(n) \wedge x \in \mathsf{rdVars}\,(\mathcal{A}) \wedge \mathcal{A} = \langle \gamma \Rightarrow A \rangle\}, \textit{else} \end{cases}$$

Note that, if a variable x occurs in node n in a guard, then it must eventually be read by node n, i. e., $\gamma_{\mathrm{rd}}(x,n) = \mathsf{true}$. Moreover, $\gamma_{\mathrm{wr}}(x)$ states whether a variable is set by a SGA. As a conclusion, $\neg\gamma_{\mathrm{wr}}(x)$ is the condition when the default action must be activated to ensure determinism as described in Section 2.1.2.

4.3.1.2 Push and Pop Conditions

In general, a node of a DPN has only a partial view on the complete system. This includes the variables of the system, i. e., a node reads and writes only a part of the system's variables. As a result, a reader might not be able to evaluate $\gamma_{\mathrm{wr}}(x)$, while a writer might not be able to evaluate $\gamma_{\mathrm{rd}}(x,n)$. Nevertheless, we will first ignore this issue and consider this problem later in Section 4.3.1.3.

The condition whether a value for variable x has to be transported depends on the type of x and the kind of assignment (immediate or delayed). We start with an explanation when a value for an immediately written event variable x has to be transported:

1. $\gamma_{\mathrm{rd}}(x,n) \wedge \gamma_{\mathrm{wr}}(x)$: the value is transported from the sender to the receiver. The receiver (node n) has the up-to-date value for x and all calculations must yield the correct result.

2. $\neg\gamma_{\mathrm{rd}}(x,n) \wedge \gamma_{\mathrm{wr}}(x)$: the value is not sent. $\neg\gamma_{\mathrm{rd}}(x,n)$ states that x is not involved into any action. Hence, all variables written by the receiver must be correct. The local copy of the receiver n contains then $\perp$.

3. $\gamma_{\mathrm{rd}}(x,n) \wedge \neg\gamma_{\mathrm{wr}}(x)$: n executes the default action for its local copy. This yields exactly the same value as calculated by the sender, which is a constant for event variables. Hence, any actions that use x for their calculations use the correct value.

4. $\neg\gamma_{\mathrm{rd}}(x,n) \wedge \neg\gamma_{\mathrm{wr}}(x)$: x is not involved into any calculations. Hence, n may invalidate its copy of x or execute the default action. We do the latter.

In case that x is an immediately written memorized variable, the handling of communication gets more sophisticated. First of all, the writer of x must keep track whether the local copy of reader n is up-to-date:

Definition 19. *The flag $x^n_{\equiv}$ denotes the status, whether the local copy of the memorized variable x in node n has been up-to-date in the* previous *macro step, formally: $\mathsf{X}x^n_{\equiv} \leftrightarrow x \neq \perp$. The flag is stored at and handled by the sender of x, i. e., $N_{wr}(x)$,* and *the reading node n.*
The temporal relationship is given by the following formula:
$\mathsf{X}x^n_{\equiv} \leftrightarrow (\gamma_{rd}(x,n) \vee \neg\gamma_{wr}(x) \wedge x^n_{\equiv})$

The temporal relationship of $x^n_{\equiv}$ as defined in Definition 19 can be derived as follows. If x is read by n then the local copy of x either must already be up-to-date or it must be updated by invoking a transport of x. Hence, in the next macro step $x^n_{\equiv}$ must be set. If x is not changed by the writer, i. e., $\neg\gamma_{\mathrm{wr}}(x)$ holds, the writer will execute the default action for x. If the local copy of x in node n has been up-to-date in the previous macro step then n will also execute the default action on its local copy, thereby imitating the behavior of the sender. As a result, this will keep the local copy up-to-date, i. e., $\mathsf{X}x^n_{\equiv}$ must hold.

Definition 19 allows us to precisely define the behavior for immediately written memorized variables:

1. $\gamma_{\mathrm{rd}}(x,n) \wedge \gamma_{\mathrm{wr}}(x)$: the value is sent according to the idea above. Receiver n has the up-to-date value for x and all calculations must yield the correct result. In addition, $x^n_{\equiv}$ must be set, since the receiver has an up-to-date value.

2. $\neg\gamma_{\mathrm{rd}}(x,n) \wedge \gamma_{\mathrm{wr}}(x)$: the value is not sent. The local copy of receiver n contains then $\perp$, i. e., the local copy is invalidated. Hence, $x^n_{\equiv}$ must be *un*set to keep track of the invalid copy in the receiver. Nevertheless, all variables written by the receiver must be correct because the unset condition $\gamma_{\mathrm{rd}}(x,n)$ states that x is not involved into any action or calculation.

3. $\gamma_{\mathrm{rd}}(x,n) \wedge \neg\gamma_{\mathrm{wr}}(x)$: the behavior of the nodes depends on the flag $x^n_{\equiv}$, i. e., whether the local copy is up-to-date. If $x^n_{\equiv}$ holds, the reader must execute the default action, otherwise the sender must send the current value of x to the receiver. In both cases, the value of x will be updated. Hence, the flag $x^n_{\equiv}$ must be set. Finally, the condition whether to send / receive a value for this case is $\gamma_{\mathrm{rd}}(x,n) \wedge \neg\gamma_{\mathrm{wr}}(x) \wedge \neg x^n_{\equiv}$.

4. $\neg\gamma_{\mathrm{rd}}(x,n) \wedge \neg\gamma_{\mathrm{wr}}(x)$: if $x^n_{\equiv}$ holds, the reader must execute the default action to keep its local copy up-to-date. Otherwise ($x^n_{\equiv}$ is unset) the local copy stays invalid. This behavior guarantees that transports are avoided as long as the variable is not changed by the sender. To this end, the local copy of x and the flag $x^n_{\equiv}$ will simply keep their values.

The next task is to handle delayed actions. The read of a variable x takes place in the succeeding macro step of its write. Therefore, we have to consider the values of $\gamma_{\mathrm{rd}}(x,n)$ and $\gamma_{\mathrm{wr}}(x)$ from different macro steps, i. e., the sender has to evaluate $\mathsf{X}\gamma_{\mathrm{rd}}(x,n)$. Hence, the values that are necessary to evaluate the condition would have to be read ahead by the sender. This somehow corresponds to "reading into the future". Reading values that eventually will not be available, yet, will technically end up with a deadlock of the DPN. Thus, the sender must assume the worst case (in the sense of communication overhead): the receiver will read the value of x. To this end, if we set $\gamma_{\mathrm{rd}}(x,n)$ to true for all delayed written variables, we can use the distinctions above to check whether a value has to be sent or not. The different timing has to be considered for the overall write guard in the receiver, too. In particular, to determine whether to read x, a receiver must evaluate $\gamma_{\mathrm{wr}}(x)$ in the preceding iteration. Since all variables to evaluate this expression may already have been changed, we simply add a carrier variable $\gamma_{\mathrm{wr,pre}}(x)$ and the guarded action $\langle \mathsf{true} \Rightarrow \mathtt{next}(\gamma_{\mathrm{wr,pre}}(x)) = \gamma_{\mathrm{wr}}(x)\rangle$ to the behavior of the receiver.

Because the overall guard for reading a delayed written variable at the receiver must be assumed to be true, the transport of these variables provide the least potential for communication reduction. The condition for transporting a value for a macro step depends only on the overall write guard. As a result, all value changes of delayed written variables are committed to the receiver, i. e., the receiver's copy of the variable will be always up-to-date. If the variable of the sender does not change, the reaction to absence is applied to the variable in the sender and to the copy of each receiver. Hence, there is no need to keep track of whether the local copy of a receiver is up-to-date as done for immediately written memorized variables (see $x^n_{\equiv}$ in case 4 of the behavior definition for immediately written memorized variables).

To conclude, we can now precisely determine the conditions when the sender of an immediately written variable x has to send (*push*) the value to a receiver n through the corresponding buffer, and when receiver n of that variable has to read (*pop*) the value from the corresponding buffer:

- If x is an immediately written event variable:
  $\gamma_{\mathrm{push}}(x,n) = \gamma_{\mathrm{pop}}(x,n) := \gamma_{\mathrm{rd}}(x,n) \wedge \gamma_{\mathrm{wr}}(x)$

- If x is an immediately written memorized variable:
  $\gamma_{\mathrm{push}}(x,n) := \gamma_{\mathrm{rd}}(x,n) \wedge (\gamma_{\mathrm{wr}}(x) \vee \neg\gamma_{\mathrm{wr}}(x) \wedge \neg x^n_{\equiv})$
  $\gamma_{\mathrm{pop}}(x,n) := \gamma_{\mathrm{rd}}(x,n) \wedge (\gamma_{\mathrm{wr}}(x) \vee \neg\gamma_{\mathrm{wr}}(x) \wedge \neg x^n_{\equiv})$

- If x is a delayed written event variable:
  $\gamma_{\mathrm{push}}(x,n) := \gamma_{\mathrm{wr}}(x) \qquad \gamma_{\mathrm{pop}}(x,n) := \overleftarrow{\mathsf{X}}\gamma_{\mathrm{wr}}(x)$

- If x is a delayed written memorized variable:
  $\gamma_{\text{push}}(x,n) := \gamma_{\text{wr}}(x) \qquad \gamma_{\text{pop}}(x,n) := \overleftarrow{\mathsf{X}}\gamma_{\text{wr}}(x) \vee \mathit{init}$
  init denotes a build-in variable that is true for the very first macro step and false otherwise.

To handle immediately written event variables, it is sufficient to modify the conditions for executing a push and pop. Whereas, memorized and/or delayed written variables need to change the behavior of the sender and receiver, i. e., to add additional guarded actions. The modifications in concrete terms depend on the specific case as shown in the pseudo code in Figure 4.13.

4.3.1.3 Quantification

In the following, we assume that a variable is *known* in a node, if the node reads or writes that variable. Conversely, if a variable is neither written nor read by a node, it is stated as *unknown* in that node. This is based on the fact that nodes in a DPN will only receive data that is necessary to compute their own behavior. In addition, each node is supposed to be capable of reading variables that are written in the same macro step by itself.

Until now, we assumed that all variables that occur in the expressions $\gamma_{\text{wr}}(x)$ and $\gamma_{\text{rd}}(x,n)$ are known in both, the sender and receiver. In general, a node will not have knowledge of all variables of the overall system, i. e., some variables might be *unknown* in a node. As mentioned, a node only knows variables that are either necessary for the computation of its behavior or that are produced by the node itself. Hence, the sender and receiver may not be able to evaluate $\gamma_{\text{rd}}(x,n)$ and $\gamma_{\text{wr}}(x)$, respectively, without adding communication. In conclusion, the presented approach at that point is only applicable in case that all variables in both expressions are known by both nodes. To extend this approach, this section introduces simplified *valid evaluable* guards.

Let $\mathcal{V}$ be the set of all variables in the DPN including inputs from and outputs to the environment. Furthermore, let $\mathsf{VAR}(\tau) \subseteq \mathcal{V}$ denote all variables of the formula τ. In addition, $\mathsf{VAR}(n)$ denotes all variables that are known by node n.

Obviously, we must preserve sending of all non-bubble values. This leads to the following constraints of a simplified guard to ensure that values can be evaluated: (1) A simplified guard $\gamma^{\exists}$ that simplifies γ is valid iff $\gamma \rightarrow \gamma^{\exists}$. (2) A simplified guard $\gamma^{\exists}$ that simplifies γ is evaluable by node n iff $\mathsf{VAR}(\gamma^{\exists}) \subseteq \mathsf{VAR}(n)$. The simplification is transport specific, i. e., the simplified overall write guard gets individually adapted to each reading node n. Hence, n becomes a parameter in $\gamma^{\exists}_{\text{wr}}(x,n)$.

The requirement for validity leads to a pessimistic overestimation of guards, i. e., if γ cannot be evaluated due to missing variables, each node must assume a read/write by its partner node. For a transport of x between its sender and its receiver n, this is formally expressed by the following conditions: $\gamma_{\text{rd}}(x,n) \rightarrow \gamma^{\exists}_{\text{rd}}(x,n)$ and $\gamma_{\text{wr}}(x) \rightarrow \gamma^{\exists}_{\text{wr}}(x,n)$. The ability of a simplified guard $\gamma^{\exists}$ to be evaluated means that a node knows all variables that occur in $\gamma^{\exists}$. An existential quantification over the unknown variables in $\gamma_{\text{rd}}(x,n)$ and $\gamma_{\text{wr}}(x)$ will result in the desired simplified guards.

1: **function** OptTransport$((\mathcal{N}, \mathcal{B}), \mathcal{V}, \mathcal{T})$
2: $\quad \mathcal{V}_{next} := \{x \mid x \in \mathcal{V} \wedge$
3: $\qquad \langle \gamma \Rightarrow \texttt{next}(x) = \tau \rangle \in \mathsf{behavior}\,(N_{\mathrm{wr}}(x))\}$
4: $\quad \mathsf{VAR}(n) := \mathsf{wrVars}\,(n) \cup \mathsf{rdVars}\,(n)$
5: $\quad$ **forall** $t \in \mathcal{T}$
6: $\quad\quad (n_1, n_2) := (\zeta_{\mathrm{wr}}(x), \zeta_{\mathrm{rd}}(x)) = t$
7: $\quad\quad \mathcal{V}_1 = \mathsf{VAR}(n_1) \setminus \mathsf{VAR}(n_2)$
8: $\quad\quad \mathcal{V}_2 = \mathsf{VAR}(n_2) \setminus \mathsf{VAR}(n_1)$
9: $\quad\quad \gamma_{\mathrm{wr}} = \bigvee\{\gamma \mid \mathcal{A} \in \mathsf{behavior}\,(n_1) \wedge$
10: $\qquad\quad x \in \mathsf{wrVars}\,(\mathcal{A}) \wedge$
11: $\qquad\quad \mathcal{A} = \langle \gamma \Rightarrow A \rangle\}$
12: $\quad\quad \gamma_{\mathrm{rd}} = \begin{cases} 1, \text{if } \exists_{\langle \gamma \Rightarrow A \rangle \in \mathsf{behavior}(n_2)}.x \in \mathsf{Vars}(\gamma) \\ \bigvee\{\gamma \mid \mathcal{A} \in \mathsf{behavior}\,(n_2) \wedge x \in \mathsf{rdVars}\,(\mathcal{A}) \wedge \\ \qquad\qquad \mathcal{A} = \langle \gamma \Rightarrow A \rangle\}, else \end{cases}$
13: $\quad\quad \gamma^{\exists}_{\mathrm{wr}} = \exists_{\nu \in \mathcal{V}_2} \nu.\gamma_{\mathrm{wr}}$
14: $\quad\quad \gamma^{\exists}_{\mathrm{rd}} = \exists_{\nu \in \mathcal{V}_1} \nu.\gamma_{\mathrm{rd}}$
15: $\quad\quad$ **case** x is event $\wedge\, x \notin \mathcal{V}_{next}$:
16: $\quad\quad\quad \gamma_{\mathrm{push}} = \gamma^{\exists}_{\mathrm{rd}} \wedge \gamma^{\exists}_{\mathrm{wr}}$
17: $\quad\quad\quad \gamma_{\mathrm{pop}} = \gamma_{\mathrm{push}}$
18: $\quad\quad$ **case** x is memorized $\wedge\, x \notin \mathcal{V}_{next}$:
19: $\quad\quad\quad \gamma_{\mathrm{push}} = \gamma^{\exists}_{\mathrm{rd}} \wedge \left(\gamma^{\exists}_{\mathrm{wr}} \vee \neg\gamma^{\exists}_{\mathrm{wr}} \wedge \neg x^{n}_{\equiv}\right)$
20: $\quad\quad\quad \gamma_{\mathrm{pop}} = \gamma_{\mathrm{push}}$
21: $\quad\quad\quad A_{\mathrm{uptodate}} =$
22: $\quad\quad\quad\quad \langle \mathsf{true} \Rightarrow \texttt{next}(x_{\equiv}) = \left(\gamma^{\exists}_{\mathrm{rd}} \vee \neg\gamma^{\exists}_{\mathrm{wr}} \wedge x^{n}_{\equiv}\right) \rangle$
23: $\quad\quad\quad n_1 = n_1 \cup \{A_{\mathrm{uptodate}}\}$
24: $\quad\quad\quad n_2 = n_2 \cup \{A_{\mathrm{uptodate}}\}$
25: $\quad\quad$ **case** x is event $\wedge\, x \in \mathcal{V}_{next}$:
26: $\quad\quad\quad \gamma_{\mathrm{push}} = \gamma^{\exists}_{\mathrm{wr}}$
27: $\quad\quad\quad \gamma_{\mathrm{pop}} = \gamma^{\exists}_{\mathrm{wr,pre}}$
28: $\quad\quad\quad n_2 = n_2 \cup \{\langle \mathsf{true} \Rightarrow \texttt{next}(\gamma^{\exists}_{\mathrm{wr,pre}}) = \gamma^{\exists}_{\mathrm{wr}} \rangle\}$
29: $\quad\quad$ **case** x is memorized $\wedge\, x \in \mathcal{V}_{next}$:
30: $\quad\quad\quad \gamma_{\mathrm{push}} = \gamma^{\exists}_{\mathrm{wr}}$
31: $\quad\quad\quad \gamma_{\mathrm{pop}} = \gamma^{\exists}_{\mathrm{wr,pre}} \vee init$
32: $\quad\quad\quad n_2 = n_2 \cup \{\langle \mathsf{true} \Rightarrow \texttt{next}(\gamma^{\exists}_{\mathrm{wr,pre}}) = \gamma^{\exists}_{\mathrm{wr}} \rangle\}$
33: $\quad\quad n_1 = n_1 \cup \{\langle \gamma_{\mathrm{push}} \Rightarrow n_1\_n_2\_x.\mathsf{push(x)} \rangle\}$
34: $\quad\quad n_2 = n_2 \cup \{\langle \gamma_{\mathrm{pop}} \Rightarrow x = n_1\_n_2\_x.\mathsf{pop()} \rangle\}$

Figure 4.13: Pseudo code for general communication optimization. Variable *init* in line 29 denotes a build-in variable that is true in the very first macro step and false otherwise. $n_1\_n_2\_x$ denotes the buffer to communicate x from n_1 to n_2.

Let $\zeta(x)$ be the transport of variable x from sender $\zeta_{\mathrm{wr}}(x)$ to a receiver $\zeta_{\mathrm{rd}}(x)$. In addition, let $\gamma_{\mathrm{wr}}(x)$ be the guard for a write of x in $\zeta_{\mathrm{wr}}(x)$, and let $\gamma_{\mathrm{rd}}(x, \zeta_{\mathrm{rd}}(x))$ be the guard for a read of x in $\zeta_{\mathrm{rd}}(x)$. Then, $\mathcal{V}_1 = \mathsf{VAR}(\zeta_{\mathrm{rd}}(x)) \setminus \mathsf{VAR}(\zeta_{\mathrm{wr}}(x))$ denotes the set of variables that are known in the receiver but not in the sender, and $\mathcal{V}_2 = \mathsf{VAR}(\zeta_{\mathrm{wr}}(x)) \setminus \mathsf{VAR}(\zeta_{\mathrm{rd}}(x))$ is the set of variables that are known in the sender but not in the receiver. We can then define the simplified guards as follows:

$$\gamma^{\exists}_{\mathrm{rd}}(x, \zeta_{\mathrm{rd}}(x)) := \exists_{\nu \in \mathcal{V}_1} \nu.\gamma_{\mathrm{rd}}(x, \zeta_{\mathrm{rd}}(x)) \tag{4.1}$$

$$\gamma^{\exists}_{\mathrm{wr}}(x, \zeta_{\mathrm{rd}}(x)) := \exists_{\nu \in \mathcal{V}_2} \nu.\gamma_{\mathrm{wr}}(x) \tag{4.2}$$

Then, one can see why $\gamma^{\exists}_{\mathrm{rd}}(x, \zeta_{\mathrm{rd}}(x)) = \mathsf{true}$ must always hold for delayed written variables: to obtain a correct temporal access to the variables in the sender $\zeta_{\mathrm{wr}}(x)$, all variables v in $\gamma^{\exists}_{\mathrm{rd}}(x, \zeta_{\mathrm{rd}}(x))$ must be replaced by $\mathsf{X}v$. All variables for a macro step in the future are unknown. Applying the existential quantifier to $\gamma_{\mathrm{rd}}(x, \zeta_{\mathrm{rd}}(x))$ for all unknown variables will result then in true because all variables occurring in that expression are unknown.

As a result of the definition of $\gamma^{\exists}_{\mathrm{rd}}(x, \zeta_{\mathrm{rd}}(x))$ and $\gamma^{\exists}_{\mathrm{wr}}(x, \zeta_{\mathrm{rd}}(x))$, the condition whether to invoke a transport is determined by
$\gamma^{\exists}_{\mathrm{push}}(x, \zeta_{\mathrm{rd}}(x)) := \exists_{\nu \in \mathcal{V}_1 \cup \mathcal{V}_2} \nu.\gamma_{\mathrm{push}}(x, \zeta_{\mathrm{rd}}(x))$ and
$\gamma^{\exists}_{\mathrm{pop}}(x, \zeta_{\mathrm{rd}}(x)) := \exists_{\nu \in \mathcal{V}_1 \cup \mathcal{V}_2} \nu.\gamma_{\mathrm{pop}}(x, \zeta_{\mathrm{rd}}(x))$, respectively.

4.3.2 Refinement

The necessity of quantification due to unknown variables as described in the previous section may result in a highly overestimated push/pop condition. The output of our synthesis tool has shown that in some nodes the same variable had to be quantified quite often. Therefore, the question arised whether it is possible to furthermore reduce the communication by sending additional variables to avoid its quantification of guards. In turn, this should give a less overestimated guard and reduce the communication caused by transport of other variables. This refinement was also implemented and is part of the benchmarks discussed in Chapter 7.

We recognized that in some applications only a few transports have been modified. The reason can be found in the overall write and read guards, which are usually a disjunction of Boolean expressions. An existential quantifier applied to a disjunction may lead to true as soon as a sub-expression of the disjunction is evaluated to true. This case may occur, e. g., if a sub-expression is an unknown variable. Depending on the probability of a guard to be true and the number of additional variables that are required to evaluate the non-quantified guard, the overall communication effort may be decreased. To this end, this refinement can be understood as a communication optimization that considers a global view, i. e., several transports at the same time.

Figure 4.14 illustrates such an optimization using the sketched part of a DPN. After each node has fired exactly one time, 4 transports have to be done in the complete system on the left hand side. One value is transported through the input channel, two values (x and y) through the middle channel and a fourth value through the output channel. The

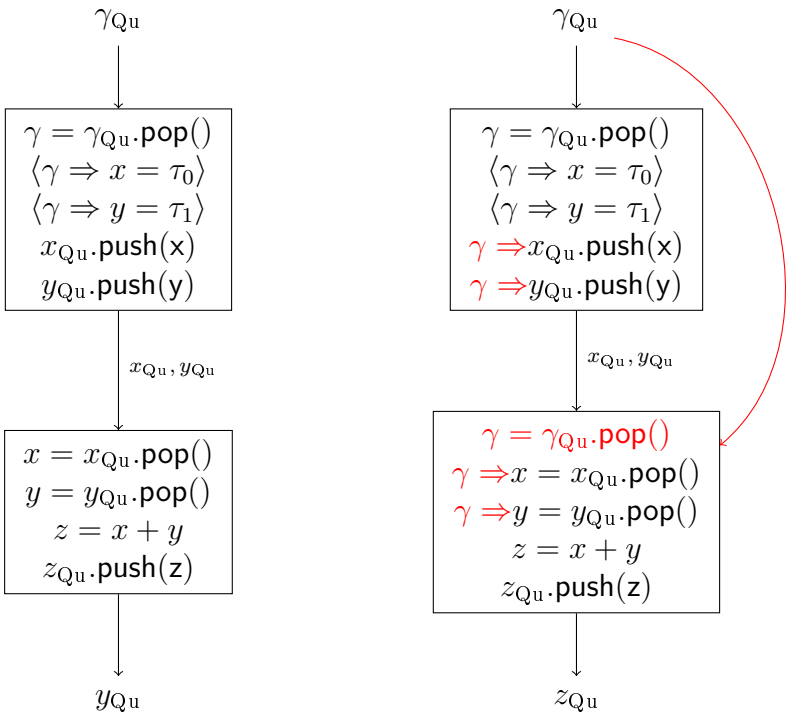


Figure 4.14: Simple sketch to demonstrate that additional communication may reduce overall communication. Left: without additional communication, right: with additional communication.

number of transports in the right-hand side depends on the probability of γ being true ($P(\gamma = \mathsf{true})$). In particular, the smaller $P(\gamma = \mathsf{true})$ is, the more communication can be saved. The average number of transports is then determined as follows: one value is transported through the input and output channel, i. e., three values in overall because the second node now become also the input value γ of the system. Between the given two nodes $2 \times P(\gamma = \mathsf{true})$ values are transported in average. In total, we have $3 + 2 \times P(\gamma = \mathsf{true})$ tokens to send which is less than 4 if γ holds in less than 50% of the cases. To this end, in this example, sending γ to the lower node as done on the right-hand side of Figure 4.14 reduces the amount of communication in the system if γ is false in most of the time.

To determine whether it is worth to send additional data depends on the amount of additional communication by these variables on the one side and communication reduction of sending other variables on the other side. In general, the net reduction of communication is determined as follows: Consider all transports between a node n_1 and n_2. Let $\mathcal{V}_{\text{unknown}} = \mathcal{V}_1 \cup \mathcal{V}_2$ be the set of variables that are unknown in n_1 or n_2. Consider $\mathcal{V}_{\text{add}} \subseteq \mathcal{V}_{\text{unknown}}$ as the set of variables that have to be additionally send to the reader or writer. Note that each variable in that set must be unknown by exactly one node because it comes from an expression from its partner node, i. e., it must be known there. In conclusion, $|\mathcal{V}_{\text{add}}|$ is the number of transports that have to be added to the involved nodes n_1 and n_2 to make $\mathcal{V}_{\text{add}}$ known in both nodes.

The additionally sent variables affect all transports between node n_1 and n_2. Let $\mathcal{T}_{n_1,n_2}$ be the set of transports between the sender n_1 and the receiver n_2:
$\mathcal{T}_{n_1,n_2} = \{\zeta \mid \zeta \in \mathcal{T} \wedge \zeta_{\text{wr}} = n_1 \wedge \zeta_{\text{rd}} = n_2\}$
Let $P(\zeta)$ be the probability whether the given transport ζ is invoked in an iteration in case that no additional variables are known. Furthermore, let $P_{\text{add}}(\zeta)$ be the probability whether the given transport ζ is invoked in an iteration, if the variable set $\mathcal{V}_{\text{add}}$ is known in both nodes. The overall number of transports that have to be invoked is then given as: $\Theta(n_1, n_2) = \sum_{\zeta \in \mathcal{T}_{n_1,n_2}} P(\zeta)$, if no additional variables are known, and $\Theta_{\text{add}}(n_1, n_2) = |\mathcal{V}_{\text{add}}| + \sum_{\zeta \in \mathcal{T}_{n_1,n_2}} P_{\text{add}}(\zeta)$, if the variable set $\mathcal{V}_{\text{add}}$ is known in both nodes. The difference $\Delta = \Theta(n_1, n_2) - \Theta_{\text{add}}(n_1, n_2)$ defines the number of transports that are not invoked per iteration, which already includes the transports for the additional variables. If $\Delta > 0$, the insertion of additional transports for the variable set $\mathcal{V}_{\text{add}}$ can be considered as reasonable under the assumption that costs for transported values are the same for all variables. However, the costs for transports can be different in reality, e. g., spatial distribution of nodes in a network results in different communication ways with different costs. Moreover, the calculation of probabilities requires precise knowledge of the probabilistic value distribution. This is a separate research topic and the interested reader may refer to Markov processes [181] for stochastic evaluation [11].

Additional inserted variables $\mathcal{V}_{\text{add}}$ are always read but may allow transport reduction depending on their overall write guard. Hence, recursive transport reduction is possible and would require an iterative application of the algorithm. To this end, a precise analysis whether the transport of additional variables is affordable would result in a computing extensive synthesis process. To keep our tool chain simple and fast, recursive optimization has been neglected. The decision whether to add additional communication to a node n is

based on the number of occurrences of the unknown variable in guards where this variable has to be quantified. We made this threshold θ a global parameter which was given to the synthesis tool, i. e., it was fixed at compile time.

Finally, we like to mention that another optimization could be obtained by the use of partial evaluation. At this point, all variables occurring in a guard have to be sent eventually from the corresponding writing nodes to the reading node. These guards may have a form like $\alpha \wedge \beta$ or $\alpha \vee \beta$. One could read first α and then decide whether it is necessary to read β for further evaluation. However, partial evaluation was not applied in our synthesis process due to its potential exponential blow-up of code.

4.4 Synthesis of Data-Flow Process Networks

The synthesis of DPNs is straightforward: Each node in a given DPN is translated to a single thread, e. g., as done in Section 2.3.1. Each node contains a set of SGAs, i. e., a set of actions that are executed in the synchronous MoC. Code creation for a single node can be done using a function to create sequential code as presented by Edwards in [67].

We decided to use explicit communication constructs, i. e., we introduced actions to explicitly send data to consumer nodes and receive data from producer nodes, respectively. This allows a straightforward implementation of the conditional push and pop instructions as required by the general communication reduction approach from Section 4.3. In contrast, the usage of firing rules (see Definition 8) to implement conditional communication may result in an exponential blow-up of code. The communication of a single node usually consists of multiple reads and writes. Different conditions for c communications with other nodes may potentially result in 2^c states. For each state, we have to create a new firing rule to preserve deterministic behavior, which finally results in a blowup of the generated code. For that reason, the translation of our DPNs to OpenDF [30] as done in [18] has been neglected.

Communication can be done using any thread-safe implementation of a FIFO queue, e. g., the concurrent_bounded_queue. In particular, for DPNs as created in Section 4.1, one can read from each incoming buffers of a node a single token by calling the corresponding pop() operation. After the code has been executed, the created values can be written to the outgoing buffers of a node by calling the buffer's push() operation.

Chapter 5

Out-of-Order Execution

This chapter introduces OOO execution for DPNs [20], which is a well known technique from processor architecture. In an unbalanced DPN, the values of some input buffers of a node may be produced faster than they were consumed by the node. Thus, it is reasonable to execute this node several times in parallel, leading to a better utilization of resources. Moreover, varying computational effort of single executions in this node will produce outputs that arrive OOO, enabling other nodes to fire OOO. This OOO execution avoids idle time and can speed up the program execution. As data is now reordered, the approach involves additional effort to reorder the correct flow of output data.

Our approach generates multi-threaded code exploiting OOO execution for synchronous DPNs. Thereby, we target standard commercial processors, e. g., ones of the i86 and i86-64 architecture and do not rely on any particular hardware extension.

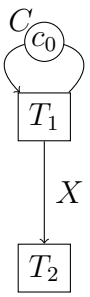
The rest of this chapter is structured as follows: The next section introduces a method to improve the multi-threaded execution of DPNs in several ways, i. e., load-balancing and additional flexibility in the provided parallelism. Section 5.2 will continue this idea by adding data speculation to further increase parallelism.

5.1 Out-of-Order Execution of Data-Flow Process Networks

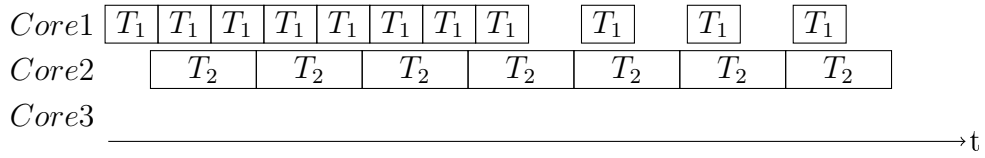
Our approach to execute synchronous guarded actions as DPNs as described in Chapter 4 already provides parallelism by executing nodes concurrently. Applications from the DSP domain often work in so-called bursts: they supply many inputs at once, and the system may use them to compute many outputs. Hence, a parallel implementation may benefit from tracking the dependencies between tasks from *different* executions of T. Thereby, we can exploit parallelism between different iterations and not only within a single iteration. As a consequence, we obtain an OOO execution (with respect to iterations).

Assume a system is given that has to be executed as shown in Figure 2.2. To execute the system on a parallel architecture, one could use the approach from Chapter 4 to get a DPN. Thereby, each node of the DPN is translated to a single thread. Communication

(a) DPN consisting of two nodes with $O(T_1) \ll O(T_2)$:



(b) Timeline for mapping each node to a single thread with a FIFO buffer of size=4:



(c) Timeline for a task-based execution considering (in-)dependencies across iterations:

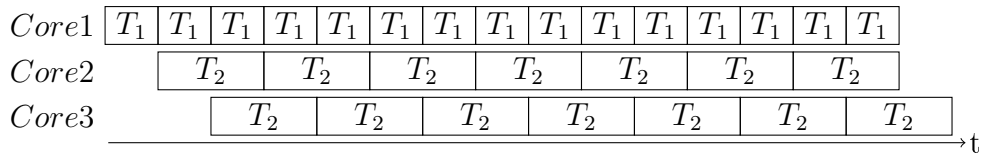


Figure 5.1: Motivation for the task-based OOO execution of DPNs. The timelines in (b) and (c) show the execution of the DPN for different approaches. We assume that the computational effort of T_1 is half of the computational effort of T_2. For both approaches, we used bounded FIFO buffers with size 4. Mapping each node to a thread will lead to a fixed amount of parallelism. For that reason, the third core in timeline (b) is idle. The unbalanced computational effort leads to backpressure, i. e., node T_1 will start stuttering after filling the buffer to T_2. T_2 has no self-dependency, and therefore, several iterations of T_2 can be executed in parallel, leading to a better utilization of the given system.

can be done, e. g., using the Intel TBB concurrent bounded queue. However, the amount of parallelism is (1) fixed to the number of nodes and (2) relies strongly on a balanced partitioning. While the former makes an application inflexible to the amount of parallelism provided by the targeted hardware architecture, the latter may result in an unbalanced workload.

Figure 5.1 (a) shows an example DPN, (b) the timeline of the execution using the mentioned approach and (c) a timeline of the execution using an improved approach. The DPN in Figure 5.1 consists of two nodes T_1 and T_2. To demonstrate the effect of unbalanced computational efforts, we assume that the computational effort of T_1 is half as that of T_2,

i. e., the execution of a single iteration of T_1 takes only half the time required for a single iteration of T_2.

The issue of using the approach as described in Chapter 4 can be seen in the timeline depicted in Figure 5.1 (b): Mapping each node to a single thread provides only two threads. A system with more cores will not be completely utilized, because one or more cores will be idle. Nodes with unbalanced computational effort will even lead to a worse utilization. A practical implementation of an application will use bounded buffers due to limited memory size. If two nodes communicate through a FIFO buffer and one node can be executed faster than the other node, then the faster node will have to wait sooner or later for the other node. In the given example, the producing node is faster and will start to stutter as soon as buffer X is filled. In particular, if buffer X is full, T_1 has to delay further computations until T_2 removes tokens from buffer X. The ability of full buffers to stall preceding nodes is called backpressure.

As can be seen, node T_2 has no self-dependency. This means that the computations of an iteration of this node do not depend on preceding iterations. It would be reasonable to execute several iterations of node T_2 concurrently as shown in the timeline given in Figure 5.1 (c). This would eventually lead to a better utilization of all cores.

A solution to execute several instances of a node in parallel has already been considered by Stulova et. al in [167]. They analyze a given SDFG with respect to the computational effort of nodes and their dependencies. Nodes with higher computational effort compared to other nodes of the SDFG are then duplicated to gain a more balanced workload. While this approach allows one to obtain more parallelism by node duplication and an improved workload, the final amount of parallelism is still statically determined. Moreover, that approach requires precise knowledge of the target architecture to determine the computational effort. However, real-world applications often contain control-flow, e. g., if-then-else statements, and therefore generally lead to a varying amount of computational effort. This also applies to our starting point which is particularly conditional actions, namely SGAs. This makes static analysis and scheduling very difficult. For this reason, we target a dynamic approach for distributing work to multi-core systems.

Section 2.3.2 introduced task-based execution and its advantages in multi-core environments. Instead of assigning parts of the work to threads, these parts are defined as tasks. A set of so-called worker threads execute the given tasks. This mainly provides flexibility with respect to the number of actual available computational nodes, i. e., available parallelism in an application is decoupled from the available hardware parallelism. To this end, we decided to use the advantages of task-based execution for the approach presented in this chapter. Thereby, a task is defined as the execution of a single step of a node.

Assuming that we are able to concurrently execute tasks of the same node, the following situation may occur: Varying amount of computational effort of a node and concurrent execution of corresponding tasks can lead to OOO arrival of computed outputs. The approach of Stulova et. al [167] does not only statically duplicate nodes, but also outgoing buffers of these nodes. A static merging mechanism has to be implemented into each dependent node, i. e., a node that reads from duplicated buffers. However, this solution will not work for dynamic duplication, i. e., concurrent task execution which is targeted

here. Inspired by OOO execution in dynamic processors, we handled this issue by using a centralized buffer. The centralized buffer provides different functions: random access to that buffer allows reordering of values without a merging element. Moreover, computed output values are input values of pending tasks. Hence, out-of-order arrival of computed values may even trigger other tasks out of-order, thereby allowing more parallelism. In the next section, we will give a detailed description of the required data structures and a pseudo-algorithm of our approach.

5.1.1 Data Structures and Algorithm

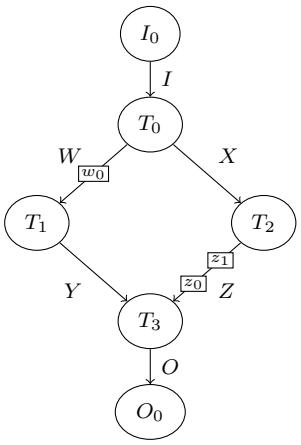


Figure 5.2: Running example for OOO execution, part 1 of 4: Exemplary SDFG that is used throughout of the running example.

SDFG and Task Functions. Throughout this section, we use a running example, an SDFG consisting of four nodes T_0, T_1, T_2 and T_3 (see Figure 5.2). The code of the individual nodes is given in Figure 5.3: when a node is triggered, the given task is executed in the final implementation, i. e., the given function is applied to the values belonging to the corresponding iteration.

Before we present our OOO execution, we first modify the given SDFG so that the communication with the environment is accomplished by additional nodes I_0 and O_0 (as shown in the *SDFG* section). These nodes are guaranteed to be executed in order (technically realized by a delayed self-dependency, i. e., the node has a feedback buffer to itself with one initial token) so that the behavior at the interface remains unchanged by internal OOO executions.

SDFG Description. As already explained above, our approach dynamically maintains a list of dependencies between task executions (which is defined by a tuple containing the SDFG node and the iteration). We use the data structures given in Figure 5.4 to store these dependencies and the actual structure of the SDFG.

```
1 :  macro CBS'[i] = CBS[i mod CBS_size]
2 :  macro CBS'_V[i] = CBS'[i].values
3 :  function Fun_I(i) {
4 :     CBS'_V[i].I = ReadInputs()
5 :  }
6 :  function Fun_0(i) {
7 :     I = CBS'_V[i].I
8 :     assert (I ≠ 0)  // avoid division by zero
9 :     CBS'_V[i+1].W = 1.0/I
10 :    CBS'_V[i].X = g_T0(I)
11 :  }
      ⋮
12 :  function Fun_O(i) {
13 :     WriteOutputs(CBS'_V[i].O)
14 :  }
```

Figure 5.3: Running example for OOO execution, part 2 of 4: The functions of the nodes of the exemplary SDFG. The code also shows how to access input and output buffers, i. e., the central buffer station, which is depicted in Figure 5.5.

- SystemState contains a list of all buffers of the actual system.
- TaskDesc identifies a specific task by the node and the iteration.
- DependencyDesc describes a dependency from a task to another task that is identified by its corresponding node *NodeName* and the distance of iterations which is given by the number of initial tokens. The distance is explained as follows: if a node T is connected to another node T' by a buffer, then these nodes are dependent. If that buffer contains N initial tokens, the ith iteration of T will produce a value that is consumed by the $i+N$th iteration of T'.
- NodeDesc contains a tuple describing all characteristics of a node that are important for the OOO execution. This includes the identifier of the node, a function to execute a task for the node and a list of outgoing dependencies.
- CBSEntry describes a single entry of the central buffer station as described in the next paragraph. It contains the state of a single iteration of the DPN in the OOO execution, i. e., the state of tasks (pending or executed), the remaining dependencies to keep track of whether a task can be executed (member *numInDeps*), the remaining number of dependencies before an entry can be removed (member *RT*) and the system state (member *values*), which is addressed by the macro definitions given in lines 1f on the right-hand side in Figure 5.3.
- *SDFG_Desc* depends on the application and describes the nodes of the actually given SDFG. *NodeNames* is a list of all nodes and is required by the functions for the

```
 1: structdef SystemState = {I, X, Y, Z, W, O}
 2: structdef TaskDesc = {NodeName : NodeID, iteration : int}
 3: structdef DependencyDesc = {NodeName : NodeID, numInitialTokens : int}
 4: structdef NodeDesc = {NodeName : NodeID, NodeFun : Function,
 5:                       Successors : list of DependencyDesc}
 6: structdef CBSEntry = {iteration : int, RT : int,
 7:                       executed : bool [], numInDeps : int [],
 8:                       values : SystemState}
 9: SDFG_Desc = {
10:    ( I_0, Fun_I, {(I_0, 1), (T_0, 0)}),
11:    ( T_0, Fun_0, {(T_1, 1), (T_2, 0)}),
12:    ( T_1, Fun_1,          {(T_3, 0)}),
13:    ( T_2, Fun_2,          {(T_3, 2)}),
14:    ( T_3, Fun_3,          {(O_0, 0)}),
15:    (O_0, Fun_O,           {(O_0, 1)})
16: } : list of NodeDesc
17: NodeNames = {n | T ∈ SDFG_Desc ∧ n = T.NodeName}
18: function GetDescOfTask(n) = {T | T ∈ SDFG_Desc ∧ T.NodeName = n}
19: function GetSucsOfTask(n) = {GetDescOfTask(n).Successors}
20: function GetNumSucsOfTask(n) = {List.length GetSucsOfTask(n)}
21: function GetPresOfTask(n) = {(k, t) | k ∈ NodeNames ∧
22:                                (n, t) ∈ GetSucsOfTask(k)}
23: function GetNumPresOfTask(n) = {List.length GetPresOfTask(n)}
24: function GetMaxInitialToken(n) =
25:    {max_{n ∈ NodeNames}(GetSucsOfTask(n).numInitialTokens)}
```

Figure 5.4: Running example for OOO execution, part 3 of 4: Structures and data for the example SDFG: the description of the SDFG and functions to access members of the description.

OOO scheduling. The functions given in line 18 to line 24 are used to access members of the previously defined structures and for convenient access to parameters that are implicitly given, e. g., the number of outgoing dependencies of a node (see function GetNumSucsOfTask in line 20).

Central Buffer Station. Due to the OOO execution of nodes of the SDFG, the nodes might not read and write the data in-order. Thus, we have to replace FIFO buffers by another data structure that gives nodes random access. As SDFGs allow the compiler to determine a static schedule, we can safely use buffers of fixed size and determine a lower bound for their size.

Inspired by the reservation station used by OOO processors, we store all buffers in a table – the central buffer station (CBS). This table includes the buffers for all input, output and state variables of the system. Similar to the reservation station of a microprocessor,

$head = 0, \quad tail = n, \quad TaskQueue = \{(I_0, 0), (T_1, 0)\}$

Entry	*iter−ation*	*RT*	*executed* $I_0 \dots O_0$	*numInDeps* I_0	T_0	T_1	T_2	T_3	O_0	*values* I	X	Y	Z	W	O	*speculationBuffer* $I_0 \dots$	O_0
0	0	6	$f \dots f$	0	1	0	1	1	1				z_0		w_0	{} …	{}
1	1	7	$f \quad f$	1	1	1	1	1	2				z_1			{}	{}
2	2	7	$f \quad f$	1	1	1	1	2	2				z_1			{}	{}
⋮	⋮	⋮	⋮ ⋮	⋮	⋮	⋮	⋮	⋮	⋮							⋮	⋮
$\text{CBS}_{\text{size}} - 3$	$\text{CBS}_{\text{size}} - 3$	7	$f \quad f$	1	1	1	1	2	2							{}	{}
$\text{CBS}_{\text{size}} - 2$	$\text{CBS}_{\text{size}} - 2$	7	$f \quad f$	1	1	1	2	2	2							{}	{}
$\text{CBS}_{\text{size}} - 1$	$\text{CBS}_{\text{size}} - 1$	7	$f \dots f$	2	2	1	2	2	3							{} …	{}

Figure 5.5: Running example for OOO execution, part 4 of 4: CBS for exemplary SDFG: the CBS after the initialization process and the task queue containing tasks that can be started. The *speculationBuffer* is only required for the speculative approach.

it is organized as a ring buffer that provides access to at least *WS* iterations where *WS* is the number of newest iterations that can be concurrently handled by the scheduler (called the window size of the scheduler). In contrast to the reservation station, the data in the CBS is not removed with a schedule, because each entry will require one schedule for each task. In addition, the CBS also serves as a reorder buffer to write outputs in-order to the environment. Hence, entries of the CBS are added and removed in-order. Since a buffer may have some initial tokens and a concurrent execution of *WS* iterations may write *WS* additional tokens into each buffer, the CBS must have a size of $CBSsize = N_{max} + WS$ entries where N_{max} is the maximum number of initial tokens. To allow OOO scheduling for all nodes, *WS* must be chosen to be greater than 1. The head entry (i. e., CBS entry indexed by *head*) and the tail entry (i. e., CBS entry indexed by *tail*) of the ring buffer address the oldest and the newest iteration, respectively, that are processed. The CBS for our running example after its initialization can be found in Figure 5.5. The *speculationBuffer* in the table is a member which is required for speculative execution. It will be added in Section 5.2 to the structure, and the reader can ignore it at this moment.

The read and write positions of a buffer in an iteration i are determined as follows: All nodes will always read the ith element, and therefore, the read position is $\text{ReadPos}(B) = i \bmod \text{CBS}_{\text{size}}$. The write position depends on the number of initial tokens for a buffer B: let B_{init} be the number of initial tokens for buffer B, then the write position for that buffer is defined as $\text{WritePos}(B) = (i + B_{\text{init}}) \bmod \text{CBS}_{\text{size}}$ (see pseudo code of nodes in Figure 5.3).

In addition to the system's values and its state, the scheduler requires information about the tasks, e. g., whether a task is pending or has been executed, or how many dependencies are left to enable the node. In particular, for each task T, we add a counter *numInDeps*.T and initialize it with the number of incoming dependencies which can be obtained from the SDFG description (see Figure 5.4). To keep track of whether the head of the CBS can be removed, we add for each entry E of the CBS a counter $RT[E]$ for the remaining non-executed tasks. Some of the counters need an additional dependency which is used to

```
 1 : thread Worker () {
 2 :   loop {
 3 :     T = TaskQueue.pop()  // blocking!
 4 :     LoadFence()  // required in weak-memory model
 5 :     GetDescOfTask(T.NodeName).NodeFun(T.iteration)
 6 :     StoreFence()  // required in weak-memory model
 7 :     CBS'[T.iteration].executed.(T.node) = t
 8 :     v = fetch_and_decr(CBS'[T.iteration].RT)
 9 :     if (v == 0)  then ReplaceHead()
10 :     DecrDepsOfSucs(T)
11 :   }
12 : }
13 : function DecrDepsOfSucs(TaskDesc T) {
14 :   forall S ∈ GetSucsOfTask(T.name)
15 :     i = T.iteration + S.numInitialTokens
16 :     v = fetch_and_decr(CBS'[i].numInDeps.S.NodeName)
17 :     if (v == 0)  then TaskQueue.push(S.NodeName, i)
18 : }
19 : function ReplaceHead() {
20 :   LoadFence()  // required in weak-memory model
21 :   CBS'[head].iteration = CBS'[tail].iteration + 1
22 :   StoreFence()  // required in weak-memory model
23 :   forall n∈NodeNames
24 :     CBS'[head].executed.n = f
25 :     CBS'[head].numInDeps.n = GetNumPresOfTask(n) + 1
26 :   CBS'[head].RT = 1 + List.length SDFG_Desc
27 :   head = (head + 1) mod CBS_size
28 :   tail = (tail + 1) mod CBS_size
29 :   StoreFence()  // required in weak-memory model
30 :   forall n∈NodeNames
31 :     i = tail − GetMaxInitialToken(n)
32 :     v = fetch_and_decr(CBS'[i].numInDeps.n)
33 :     if (v == 0)  then TaskQueue.push(n, i)
34 :   v = fetch_and_decr(CBS'[head].RT)
35 :   if (v == 0)  then ReplaceHead()
36 : }
```

Figure 5.6: Pseudo-code of the out-of-order scheduler.

ensure correct execution. A detailed explanation is given in the following by the description of function ReplaceHead.

Scheduler. Figure 5.6 shows the pseudo code for our worker thread, which is responsible for executing pending tasks. It selects a task from the *TaskQueue* (see line 3) and executes it using the task function of the corresponding node (see line 5). After a task has been executed, the corresponding *executed* flag is set. As explained in the previous paragraph, the RT counter is used to keep track of whether a head can be removed. Hence, after a task has been executed, the counter $RT[i \bmod \mathrm{CBS}_{\mathrm{size}}]$ is decremented (see line 8). When $RT[head]$ reaches 0, all tasks of the head have been executed, and the head can be removed (see line 9 and function ReplaceHead in Figure 5.6). Moreover, the executed task provides input values for its depending tasks. Hence, the scheduler must decrement each counter of its successors (see line 10 and function DecrDepsOfSucs). As soon as a counter reaches the value 0, the corresponding task must be scheduled (lines 17f).

Function ReplaceHead is responsible to replace the head of the CBS, which is done after all tasks of the head have been executed. In practice, this means to reset the counters to their initial values such that a new iteration can be processed. In particular, the counter $numInDeps.T$ is initialized for each task T with the number of incoming dependencies of task T plus 1 (line 25). The additional dependency is removed in a later step of function ReplaceHead. It is used to guarantee that the execution of a task is postponed until the CBS entries for writing computed values become available. Writes to a buffer may address an entry that does not correspond to the entry that is processed. The number of initial tokens in a buffer defines the relative offset where data has to be placed when a write access has to be handled. As a consequence, tasks can only be scheduled when all addressed entries in the CBS are available. For instance, buffer W in the running example has one initial token (see Figure 5.2). The ith iteration of node T_0 will write the $i+1$th token into buffer W, i. e., the value will be written to the $i+1$th entry of the CBS. Hence, the execution of task T_0 for iteration i has to be postponed until CBS entry $i+1$ becomes available. A single additional dependency is sufficient because entries in the CBS are added in order. All relative offsets of the addressed entries must be 0 or positive and at most the number of initial tokens of that buffer with the most initial tokens. Hence, whenever the entry addressed by the largest offset becomes available, then every preceding entry must be also available. In the following, we call the difference between the iteration of the targeted CBS entry and the iteration counter i of the executed task the *write distance*. The actual write distance depends on the node n and its outgoing dependencies. It is formally defined by function GetMaxInitialTokens(n) given in lines 24f of Figure 5.4, which is required at a later point of time in function ReplaceHead.

The counter $RT[head]$ is initialized with the number of tasks (line 26) plus 1. Analogous to the additional dependencies to avoid anticipated schedules, we have to avoid premature removal of CBS entries. In other words, we enforce the in-order removal of CBS entries (head first) by using a similar procedure. $RT[E]$ is incremented for all entries except for the head, i. e., after all tasks of an entry in the CBS have been executed, the corresponding

entry E will remain in the CBS due to its additional dependency until that dependency is removed by the replacement of the preceding CBS entry (see line 34).

After updating the *head* and *tail* counter (lines 27f), the function removes additional dependencies from tasks that ensure a safe write distance as explained before. The reset of the CBS head must be completed before checking whether tasks must be scheduled to prevent race conditions: Otherwise, a task might be scheduled before the reset phase is finished. Since there is no condition that might prevent the execution of the task, it is also possible that this task is executed during the reset phase. At the end of the task's execution, the counters of the task's successors must be decremented, and the corresponding thread might access non-initialized counters. As a consequence, the reset phase would overwrite the counters with wrong values. A separation of these phases such that the reset of the head must be completed before any schedule is made, solves this problem.

Removal of additional dependencies to ensure correct execution is first done for the nodes, i. e., for $numInDeps.T$. A task T in iteration i will eventually write to CBS entry $i + \mathsf{GetMaxInitialTokens(T)}$. As explained above, this means that only those tasks T can be scheduled for which $i + \mathsf{GetMaxInitialTokens(T)} \leq tail$ holds. This is equivalent to $i \leq tail - \mathsf{GetMaxInitialTokens(T)}$. If a CBS entry is replaced, then the new tail is determined by $tail_{new} = tail + 1$. As a consequence, the new condition for ensuring a safe write distance is then given by $i \leq tail_{new} - \mathsf{GetMaxInitialTokens(T)}$. We assume that the additional dependency already has been removed for all tasks where $i \leq tail - \mathsf{GetMaxInitialTokens(T)}$ holds. It follows that the condition must be changed for those tasks where $tail - \mathsf{GetMaxInitialTokens(T)} < \mathsf{i} \leq tail_{\mathsf{new}} - \mathsf{GetMaxInitialTokens(T)}$ holds, i. e., $i = tail_{new} - \mathsf{GetMaxInitialTokens(T)}$. Hence, the dependency counter of each node n is decremented at position $tail_{new} - \mathsf{GetMaxInitialTokens(T)}$ (see lines 31f).

The final step of function ReplaceHead is to remove the additional dependency of RT (see line 34). It is important to decrement RT at the end of ReplaceHead to avoid races to the *tail* counter in line 31. The OOO execution allows one that all tasks of the new head might already have been executed. Hence, RT of the new head can reach 0, and therefore, it might trigger the ReplaceHead function for the new head (see line 35). This allows a recursive removal of CBS entries.

In general, it is possible that several threads try to decrement and read the same counter. Hence, the decrement and read operations must be made atomic. Some libraries, e. g., Intel TBB, provide functions and/or data structures to execute such operations atomically. An alternative implementation using ordinary locks and conditions that are provided by most operating systems are not recommended due to their costs. On architectures like the x86 or AMD64, it is better to use spin-locks, e. g., using compare and swap (CAS) operations, because these operations are light-weight and the probability that races occur is for most systems quite low.

5.2 Data Speculation in Out-of-Order Execution

This section introduces data speculation in addition to the previously presented OOO execution. The main goal of this approach is to improve the overall performance of SDFGs execution by a better utilization of the available processing elements. Thereby, we aim at a seamless integration. In particular, the OOO execution should have priority. Speculation should only utilize idle processing units, i. e., speculation should fill in the gaps that would otherwise be obtained by idle processing elements. In addition, a software implementation gives us flexibility in the selection of tasks: we dynamically adapt the task selection such that the speculation automatically selects tasks with high hit ratio. In case that the hit ratio falls below a user-defined threshold, the speculation turns itself off to save energy.

We will first motivate our approach of speculative execution, which also explains how values are speculated. Afterwards, we will explain the concept and the required modifications of the OOO execution.

5.2.1 Motivation

5.2.1.1 False Dependencies

In general DFGs, the number of tokens consumed and produced by a node can vary in every reaction step of the node. These numbers may depend on internal states, but also on the values read from the one or the other input buffer in the same reaction step. Since SDFGs allow one to statically derive a schedule, we may wish to convert a general DPN to a SDFG. This can be done by simply adding dummy values for reading and writing to every reaction so that each reaction step will always consume and produce the same number of tokens. The dummy values introduced this way lead then to *false dependencies* in the sense that a node need not really have to wait for data tokens from these input channels.

A dynamic resolution of false dependencies can be achieved by partial evaluation. This means that we start the computation of the task that defines the reaction of a node to determine which values are currently really of interest. If a dependency turns out to be a true one, the current task must be either canceled, which will add further communication, or the task must be blocked. The latter violates the task-based execution model, which requires that a task is completely executed without blocking. Hence, adding this technique to OOO execution is definitely not an option.

An alternative way for early evaluation is to start some task whenever computational power becomes available even if there are no pending tasks. This means that we assume that all missing values are only false dependencies and optimistically start tasks. Obviously, this may fail because some of the dependencies may turn out to be true dependencies. To make a speculative execution of a task more effective, we therefore add data speculation.

5.2.1.2 Speculation of Values

Speculating values is a subtle and non-trivial task. It can be characterized as a process of making a guess on future values based on present and past values. Obviously, the behavior of applications varies, and therefore, the values they compute are also varying. As a result, the actual hit rate of a speculation depends on the application. Lipasti and Shen [120] and Richardson [151] figured out that many real-world applications rarely modify values of variables. Hence, many speculation algorithms just use previous values as speculated values. This fits well with the common requirement to keep the computational effort for speculations low, i. e., speculations should be as fast as possible and require as few resources as possible, e. g., memory accesses.

Synchronous languages originate from hardware design and provide variables with different storage types. To discuss our idea about speculation, we mainly have to distinguish between memorized and event variables: While *memorized* variables behave as registers, i. e., they keep their values unless they are changed, *event* variables are reset to a default value, i. e., a constant value, in case that no assignment is made. Based on the assumption that applications rarely change values of variables, we use a speculation that considers the storage type of variables: For memorized variables, the speculation will return the value from the previous iteration, and for event variables, the pre-defined default value is chosen. In case of memorized variables, the value from the previous iteration might not have been computed, yet, due to OOO execution. Special handling of this case is relinquished to keep the speculation process simple. The speculation generally might produce invalid input sets for a node, which is handled by automatically generated code inside of the *specExec* functions (see example, line 36 in Figure 5.9).

5.2.2 Concept

The idea of our speculation is to select a task for speculative execution as soon as a core is idle, i. e., when no pending tasks are available. According to our motivation, missing input values for executing a task are assumed to be either not required by the selected task or to be some default values, i. e., values of the previous iteration or constants.

In particular, we memorize speculated input values that are used for a speculative task execution. As soon as all inputs for a task are known, we can compare them with the speculated ones. This allows us to use the result of a speculatively executed task in case that some missing values turn out to be true dependencies and their speculated values were correct. Modified and additional data structures are depicted in Figure 5.7. In addition to the existing task functions from the OOO execution (see Figure 5.3), we need the functions shown in Figure 5.9. The modified code of the worker thread and additional functions are printed in Figure 5.8. Modifications of the worker thread have been marked with underlined line numbers.

In general, algorithms that use speculation must provide a roll-back mechanism to undo changes which have been made due to wrong speculations. Similar to other speculation approaches, we hold back changes until the result of a speculation is known to be correct. In particular, input and result values of speculations are not directly written back to the

```
 1: structdef SpecDesc = {NodeName : NodeID,  iteration : int,
 2:                       inputs : SystemState, outputs : SystemState}
 3: structdef NodeDesc = {NodeName : NodeID,  NodeFun : Function,
 4:                       specCopy : Function,  specGuess : Function,
 5:                       specExec : Function,  specCheck : Function,
 6:                       Successors : list of DependencyDesc,
 7:                       speculationAllowed : bool }
 8: structdef CBSEntry = {iteration : int,  RT : int,
 9:                       executed : bool [],  numInDeps : int [],
10:                       values : SystemState,
11:                       speculationBuffer : list of SpecDesc}
12: SDFG_Desc = {
13:    ( I_0, Fun_I, specCopy_I, specGuess_I, specExec_I, specCheck_I, {(I_0,1),(T_0,0)}, false),
14:    (T_0, Fun_0, specCopy_0, specGuess_0, specExec_0, specCheck_0, {(T_1,1),(T_2,0)}, true ),
15:    (T_1, Fun_1, specCopy_1, specGuess_1, specExec_1, specCheck_1,       {(T_3,0)}, true ),
16:    (T_2, Fun_2, specCopy_2, specGuess_2, specExec_2, specCheck_2,       {(T_3,2)}, true ),
17:    (T_3, Fun_3, specCopy_3, specGuess_3, specExec_3, specCheck_3,       {(O_0,0)}, true ),
18:    (O_0, Fun_O, specCopy_O, specGuess_O, specExec_O, specCheck_O,       {(O_0,1)}, false )
19: } : list of NodeDesc
```

Figure 5.7: Modified and additional data structures for speculative execution of our exemplary SDFG: the description of the SDFG requires additional functions for speculating values, doing a speculative execution, and to compare a speculated result.

CBS but into a buffer for speculations (see structure SpecDesc in line 1f of Figure 5.4). A completed speculative execution puts its structure containing the speculated input values and the result into a dedicated buffer in the CBS (see member *speculationBuffer* of CBS in Figure 5.5). For fast access to results of speculative executions, each task has a separated buffer.

In the following, we give a step-by-step explanation of code modifications and additions depicted in Figure 5.8. The very first modification of the worker is to replace the blocking read from the task queue by a non-blocking read (see line 3). In case that no pending tasks are available, the call to tryPop will return without a token. If a pending task is available (line 4), the worker will first call CheckSpeculation to check whether speculation results are available (see line 5). If no speculation results are available or if all speculations failed, the actual task execution is started (see line 7ff). The procedure for executing a non-speculative task is identical to the one of the non-speculative execution. If a speculation has been correct, CheckSpeculation is responsible for copying the corresponding results to the CBS. Hence, no further work to copy speculative results is necessary in the worker thread. When no task is ready for execution (see line 14), i. e., the worker is idle, a speculation is triggered by calling DoSpeculation (see line 15).

The purpose of function DoSpeculation (see line 18ff of Figure 5.8) is to do a complete speculative execution of a task including associated work. First, a speculation structure is

```
 1 : thread Worker () {
 2 :   loop {
 3 :     T = TaskQueue.tryPop()  // non-blocking!
 4 :     if (T ≠ nil)  then
 5 :       specOkay = CheckSpeculation(T)
 6 :       if (¬specOkay)  then
 7 :         LoadFence()  // required in weak-memory model
 8 :         GetDescOfTask(T.NodeName).NodeFun(T.iteration)
 9 :         StoreFence()  // required in weak-memory model
10 :         CBS′[T.iteration].executed.(T.node) = t
11 :         v = fetch_and_decr(CBS′[T.iteration].RT)
12 :         if (v == 0)  then ReplaceHead()
13 :         DecrDepsOfSucs(T)
14 :     else
15 :       DoSpeculation()
16 :   }
17 : }
18 : function DoSpeculation() {
19 :   S = new SpeculationResult
20 :   (S.iteration, S.NodeName) = pick a task for speculation
21 :   LoadFence()  // recommended in weak-memory model
22 :   validSpec = true
23 :   forall (n_P, t) ∈ GetPresOfTask(S.node)
24 :     if (CBS′[S.iteration].executed.n_P)  then
25 :       GetDescOfTask(n_P).specCopy(S, S.iteration, t)
26 :     else
27 :       validSpec = validSpec ∧ GetDescOfTask(n_P).specGuess(S, t)
28 :   if (validSpec)  then
29 :     validSpec = GetDescOfTask(n_P).specExec(S)
30 :   if (validSpec)  then
31 :     StoreFence()  // required in weak-memory model
32 :     CBS′[S.iteration].speculationBuffer.(S.node).push(S)
33 : }
34 : function CheckSpeculation(T : TaskDesc) {
35 :   specValid = false
36 :   while (¬specValid ∧ ¬CBS′[T.iteration].speculationBuffer.(T.node).isEmpty())
37 :     S = CBS′[T.iteration].speculationBuffer.(T.node).pop()
38 :     LoadFence()  // required in weak-memory model
39 :     specValid = GetDescOfTask(P.node).specCheck(S, T.iteration)
40 :   return specValid
41 : }
```

Figure 5.8: Pseudo-code of the out-of-order scheduler with speculation. Modifications to the worker thread are highlighted in dark red / marked by underlined italic line numbers. Functions DecrDepsOfSucs and ReplaceHead are left unmodified (see Figure 5.6). Functions DoSpeculation and CheckSpeculation are required in our speculative approach.

```
 1 : function specGuess_I(S, t) {
 2 :     S.inputs.I = I_default
 3 :     return true
 4 : }
 5 : function specGuess_0(S, t) {
 6 :     if (t = 1)  then S.inputs.W = W_default
 7 :     if (t = 0)  then S.inputs.X = X_default
 8 :     return true
 9 : }
       ⋮
10 : function specCopy_I(S, i, t) {
11 :     S.inputs.I = CBS'_V[i].I
12 : }
13 : function specCopy_0(S, i, t) {
14 :     if (t = 1)  then S.inputs.W = CBS'_V[i].W
15 :     if (t = 0)  then S.inputs.X = CBS'_V[i].X
16 : }
       ⋮
17 : function specExec_I(S) {
18 :     assert (false)    // environmental communication must not be speculatively executed
19 : }
20 : function specExec_0(S) {
21 :     I = S.inputs.I
22 :     if (¬(I ≠ 0))  then return false // avoid division by zero
23 :     S.outputs.W = 1.0/I
24 :     S.outputs.X = g_{T_0}(I)
25 :     return true
26 : }
       ⋮
27 : function specCheck_I(i) {
28 :     assert (false)    // environmental communication must not be speculatively executed
29 : }
30 : function specCheck_0(S) {
31 :     if (CBS'_V[i].I ≠ S.inputs.I)  then return false
32 :     CBS'_V[i + 1].W = S.outputs.W
33 :     CBS'_V[i].X = S.outputs.X
34 :     return true
35 : }
       ⋮
```

Figure 5.9: Additional task functions for the speculation approach. **Functions** $\mathsf{specCopy}_*$ copy valid output values from the CBS to the local speculation structure. **Functions** $\mathsf{specGuess}_*$ do guesses for output values. **Functions** $\mathsf{specExec}_*$ apply the node's behavior to a speculation structure. **Functions** $\mathsf{specCheck}_*$ check a speculation result and copy the output values to the CBS, when a speculation result is correct.

created (see line 19) which will contain all inputs and will have space for the outputs for the task that is going to be speculated. After selecting a task for speculative execution (see line 20), it gets a copy of the actual (available) input values (see line 24f). Known inputs are initialized by copying the values from the CBS to the newly created structure using the functions specCopy$_*$. Missing data is initialized by dedicated functions, i. e., specGuess$_*$ (see line 27). Speculated input values have to be buffered to ensure a safe roll-back mechanism in case that a speculation was wrong. They cannot be stored in the CBS, because we must *not* assume that the tasks will write the input values *after* they have been speculated. A task might be currently in progress or might start during the speculation. Hence, the write access to such an input would eventually end up in a conflict because either the actual value is irreversibly overwritten or the speculated value cannot be read for comparison in the speculation check.

In addition to the obvious goal to make a "good" guess on missing input values, it is more important to have valid values that are sensible for the program's semantics. This becomes more difficult due to race conditions that might occur during the initialization of input values for a speculation. In particular, the latter issue can be handled only by additional synchronization effort, which should be avoided for performance reasons. Hence, it does not matter if some input values are invalid due to race conditions or based on a speculation, both may cause an assertion / error / fault / exception in specific cases. For instance, if a value for one input is set to 0 and the task is going to divide by this value, an assertion / error / fault / exception might be triggered depending on the final operating system and hardware architecture that is used to run the program. An optimal solution would be to avoid taking values that might result in such a faulty execution, but this is in general a hard task that entails additional computational effort for value speculation. Instead, we solved this issue by automatically adding code to the speculation of input values and the speculative task execution which checks conditions that might trigger a runtime error. Typical candidates are e. g., division by zero, subtraction of unsigned numbers, out-of-bound array accesses, etc. Moreover, invalid input values can trigger application-specific assertions that have been created by the programmer. In particular, our synthesis method translates assertions in specGuess$_*$ and specExec$_*$ to if-then-statements. Figure 5.9 gives an example for adding safety code: All inputs of a node can potentially be speculated, and therefore, they can have arbitrary values. The divisor of the division in line 23 is based on an input. Hence, it is necessary to ensure that the divisor has a non-zero value (see line 22). In case that the divisor is zero, the speculative execution will return false to signalize invalid input values. If the speculation of input values or the speculative execution of the task signalize an invalid input value then DoSpeculation will set *validSpec* to false (see line 27 and line 29). As a result, no remaining code in the function will be executed, i. e., all computations of this function will be neglected.

The actual execution of a speculative task is done using the corresponding functions specExec$_*$ (see line 29 in Figure 5.8) and differs in the target buffer which is now the previously created structure. After the execution, we put the structure into the speculation buffer of the corresponding task (see line 32). The result remains there until a check for the correctness of the speculation is initialized (see line 37ff in Figure 5.8).

Buffers may have initial tokens. In addition, a node may write to several buffers with different numbers of initial tokens, i. e., the node produces values that are consumed in different iterations of the CBS. In turn, this means that the input values for a task may be produced by tasks in different iterations. For instance, a task of node T_3 in the example SDFG in Figure 5.2 has a dependence to the task of node T_1 in the current iteration and a dependence to the task of node T_2 in the iteration before the previous iteration. Even worse, consider the following example: let A and B be nodes in a DPN. A writes values to buffers x and y where buffer x is initialized with one token and buffer b is initially empty. Furthermore, B reads from both buffers. Now assume that we want to speculatively execute node B. The input dependencies would show that B has two dependencies to A. In particular, this is a dependency to A in the previous iteration, which is due to the required value of buffer x, and a dependency to A in the current iteration, which is due to the required value of buffer y. In OOO execution, it could be the case that A in the previous iteration has not yet been executed, while A in the current iteration has already been executed. Because each dependency is listed separately, we can decide separately whether to copy or to speculate about the corresponding value, i. e., whether to call $\mathsf{specCopy}_*$ or $\mathsf{specGuess}_*$. To this end, it is mandatory for a correct behavior of the speculation that the calls to these functions make sure that only the value of interest is written. Therefore, we need parameter t in our functions for copying and speculation of input values. It allows us to distinguish between the different iterations that can be written to.

Some nodes do not only represent a function that maps inputs to outputs. For instance, nodes that communicate with the environment cannot be speculatively executed, because their changes cannot be held back and will lead to irreversible modifications. An additional flag in the node structure (*speculationAllowed* in structure NodeDesc in Figure 5.7) determines, whether it is allowed to speculatively execute a task for a node.

Scheduling of non-speculative tasks is left unchanged, i. e., we do not check speculative results when tasks are scheduled. Hence, the functions DecrDepsOfSucs and ReplaceHead remain unchanged. Performing the check for correct speculations into the task execution, results in the additional benefit that the check is executed in parallel. When a task is dequeued, the corresponding worker starts with a check on whether speculative results are available. The worker dequeues all speculative results and checks for each speculation whether the inputs for this task were actually read or equal to the actual inputs, i. e., the inputs of the non-speculative one. Obviously, it should be sufficient to apply this check only to inputs that have been speculated. This raises however some issues due to potential races that can occur in the concurrent execution. In the following, we will discuss these issues and possible solutions.

5.2.3 Time-Insensitive Check

The easiest way to handle races is to ignore them at the point of the speculation. When a speculation result is being checked, we simply compare all inputs that actually have been used by the task. This is done by an additional function which is created by our code generator. The behavior of each node is deterministic. Hence, applying a task's function to

the same inputs will always yield the same outputs. In turn, if the inputs of a speculation and the actual inputs of a pending task are equal, then the speculation result is exactly the same as the result of the non-speculative execution.

As simple as this mechanism is, it provides two advantages: First of all, it never fails, because its correctness purely depends on the comparison of input values. Since the complete input set is taken for the comparison, there is no need to detect races that might occur during the initialization of the inputs for the speculative task. Second, this way to check speculation results provides more flexibility. In contrast to the method presented in Section 5.3.1 (least-effort check), it allows one to reuse speculations. It is easily conceivable that input sets for a node may reappear, such that a node may repeatedly produce the same output values. For instance, in audio synthesis and filtering programs, a node that generates continuously a wave, e. g., a sinus curve, the outputs will reappear. Hence, the check of reusable speculation results would work like a look-up table with a bounded size and a fall-back mechanism in case that a value is not available in the look-up table, i. e., the buffer with speculation results.

We propose the following sequence to do a speculation:

1. *Create a copy of valid input values and speculate about missing input values*: A task is the execution of one iteration of a node. The required inputs to execute the non-speculative task are found by the dependencies described in the SDFG. For a speculative task, we check for each preceding task (i. e., for each incoming dependency) whether it has already been executed. If this is the case, the corresponding inputs must be available, and therefore, we can copy them into the local copy of the CBS entry. If it is not the case, we call a function for the corresponding node that writes a set of default values to the local copy.

2. *Execute task*: The task is executed with the previously initialized inputs. Thereby, it checks safety conditions as described above, e. g., if the divisor of a division is non-zero. In case that such a safety-condition is not fulfilled, the execution of the task and the remaining speculation are canceled because the actual inputs will result in non-sensitive output values.

3. *Store speculation result*: If the task execution successfully terminates, i. e., it is not aborted by a safety condition as described above, the result is put into the corresponding speculation buffer. The content of this buffer is evaluated at the beginning of the non-speculative execution of the task.

5.2.4 Smart Task Selection for Speculation

This section describes a dynamic approach to select a task for efficient speculation. We will explain how the selection of a task for speculation influences the hit ratio of the overall speculation. Using the following approach, we are able to dynamically adapt the task selection until a user-defined hit ratio in the speculative execution is achieved.

The selection of a task for speculation has to consider the time (row in the CBS) and the node (column in the CBS). The time refers to the distance to the CBS head, i. e., the number of iterations that an entry is ahead of the oldest CBS entry. The node simply determines for which node a speculation is started. Both parameters are independent of each other and may have an effect on the speculation hit ratio.

Each node has different inputs that may be more or less suited for speculation. For instance, input variables that often have the same value are more suited than variables that are likely to change. Depending on the characteristics of a node's input variables, the hit ratio of speculations for this node may vary. For efficient speculation, it is reasonable to prefer nodes with a high speculation hit ratio.

The time and its effect in speculation is explained by the following thoughts: most SDFGs represent systems that have a state. This basically means that the outputs depend on the actual inputs and (directly or indirectly) on inputs of previous iterations. Hence, despite of the parallel and OOO execution, the execution of the tasks in the CBS usually underly a partial order. As a result, the CBS head entry will usually contain more executed / less pending tasks than the CBS tail. In turn, there will be a gradient of known values in the CBS where the CBS head entry will contain more known values than the CBS tail entry. The hit ratio of a speculation will be inversely proportional to the number of speculated values. Hence, doing a speculation for a task close to the CBS head will generally end up with a higher hit ratio due to more known input values. The more input values are known for a task, the higher is the probability that its actual execution is started in the near future and will overlap with the speculative execution. The actual execution will only check completed speculations (see timeline of S_2, ..., S_8 in Figure 5.10). Results of concurrently running speculations will arrive too late, and therefore, they will be neglected. To this end, choosing a speculation close to the CBS head will generally result in a higher hit ratio but also in a higher probability of arriving to late. Choosing a speculation close to the CBS tail will end up with a low hit ratio and a high probability to be completed before the actual execution is started. The optimal range for the time parameter of a speculative execution depends on the program and the final hardware architecture.

Our approach evaluates statistic information and provides a solution to dynamically adapt the use of speculations by means of a user-defined parameter for the desired speculation hit ratio. For each node, we define a speculation window, consisting of a minimal and a maximum distance to the CBS head. This window is initialized with optimistic values, i. e., the range covers the complete CBS. A set of counters is used to keep track of the number of speculations, speculations that came too late, and bad speculations. During the execution, we regularly check after a defined number of speculations whether the observed numbers exceed their thresholds. In these cases, the speculation window of the corresponding node is reduced and the counters are reset to measure the hit-rate in the next interval. The speculation window will level out after a while, such that the hit ratio for each task will be above the user-defined parameter. Note that a window may shrink to zero size in case that too many speculations fail. A zero-sized window does not allow us to select a task. Hence, speculations for nodes with a zero-sized speculation window are stopped.

5.3 Discussion

This section discusses several variations of some methods in our approach.

5.3.1 Least Effort Check

This section discusses a different way to check the results of speculations. Although it has not been implemented, we consider it as an interesting alternative to the "time-insensitive check" in Section 5.2.3.

The critical point in the check for correct speculations are potential races in the concurrent execution of a speculated task and its non-speculative antagonist. A speculation is always initiated before its non-speculative antagonist. However, this is not guaranteed for its completion, i. e., the non-speculative execution of a task may overtake the speculation. Hence, the result of this speculation may be put into the speculation queue after the non-speculative execution of this task already has been evaluated and cleared its speculation queue.

Adding a mechanism to prevent *old* speculation results from being added to the speculation queue requires some steps in the algorithm to be atomic. As a consequence, this will end up in adding synchronization, which would typically slow down the execution. In general, the execution of a speculative task should introduce as few changes as possible to the existing OOO execution. The worst case, i. e., if all speculations fail, should be that the performance of the speculative OOO execution is left unchanged compared to its non-speculative version. Hence, the execution of a speculative task should be as non-communicative as possible to avoid expensive synchronization overhead. As a result, there is no global knowledge about speculations that are currently in progress, i. e., no worker but the one that executes the speculative task knows about a running speculation. An alternative is to implement a mechanism into the speculation to safely recognize invalid speculations.

Adding a time-stamp to allow the association of speculations to an iteration seems to solve this issue. Basically, it would allow a fast way to check whether the speculation belongs to the currently checked iteration. Nevertheless, this is a very subtle solution since it leads to potential races.

Figure 5.10 shows an execution that can appear when a speculative task is executed. The potential speculative task executions are denoted with S_1 to S_8. Races occur when the initialization of the speculation, particularly copying the available inputs, overlaps with the non-speculative execution (denoted by T) or if the CBS entry that the speculation belongs to is removed (denoted by C). The best case that can happen is S_1 which simply means that the speculation is completed before the non-speculative execution is started. Case S_2 shows a start of the speculation initialization before the non-speculative task is scheduled, but the speculation is not completed in time, such that the result is useless. A check in the speculative task execution to avoid adding out-dated speculation results to the speculation queue, or clearing the speculation queue in the head-removal step is not sufficient, because both methods cannot be implemented atomically without additional

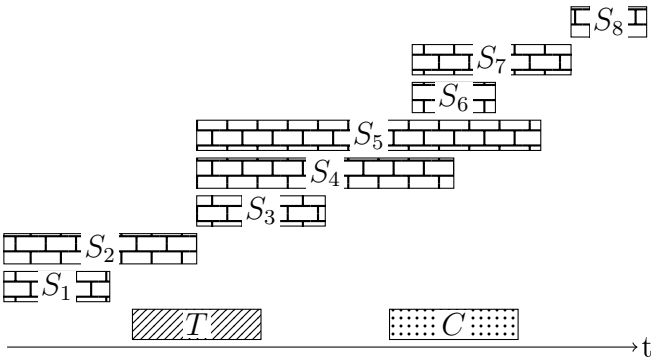


Figure 5.10: Different timelines of a concurrent execution of a speculative task initialization (boxes denoted with S_1 to S_8) compared with the execution of the non-speculative execution of the same task (the box denoted with T) and the removal of an entry in the CBS (the box denoted with C). The left and right borders of each box represent the start and end times of that execution, respectively. This figure is used to elaborate the different timings and their related race conditions.

synchronization (which still should be avoided). As a result, when speculation results are checked, the corresponding worker might access out-dated results, and therefore, it has to have the ability to recognize them. As already stated, using iteration indices as timestamps provides this ability.

The key is to make a consistent copy of the available inputs and to identify the iteration to which the copied input values belong to. A worker that is going to execute a speculative task might be interrupted during the copying process of the input values (e. g., case S_4 and S_5 in Figure 5.10). While the worker is postponed, the actual execution might take place. To make it worse, the CBS entry might be cleared before the speculative task is continued, i. e., the inputs are generally invalidated. When the worker which executes the speculative task is activated to proceed its calculations, it accesses invalid input values. As a result, its copy of the input values may contain values from different iterations, thereby making that copy inconsistent and invalid.

We extend the previous sequence to do a speculation in two steps:

1. *Create timestamp*: The behavior of a SDFG can be split into logical reaction steps. The index of the reaction step is stored in the CBS and copied to the speculation structure. This serves as the unique timestamp of the CBS entry that is handled. Whenever the entry of the CBS is cleared, this index is set to a new value and allows us to identify whether the speculation was overtaken by the non-speculative execution and the cleaning of the CBS entry.

2. *Create a copy of valid input values and speculate on missing input values*: remains as described before.

3. *Check timestamp*: At this step, we read again the unique timestamp from the CBS entry and compare it with the local copy. In case that they are not equal, the

speculation has been overtaken by the cleaning process of the CBS entry. The speculative process can be stopped because its result will eventually be neglected. If the timestamps are equal, the copies of the input values are either valid, making the speculation result valid, or - if the cleaning of the CBS entry has been started, but not finished - the inputs might be invalid (see S_4 and S_6 in Figure 5.10). The latter case will still result in a correct execution because the non-speculative execution of this task with exactly this timestamp must have already been done. Future task executions of the same node will have different timestamps and consider the result as outdated. Thus, they will neglect the result.

4. *Execute task*: remains as described before.

5. *Store speculation result*: remains as described before.

The existing OOO execution requires the following changes: when the CBS head is removed, the actual entry is reused by resetting its contents. To ensure a correct execution in the speculation approach, the update of the timestamp has to be finally done, i. e., after all other members of the CBS entry have been reset.

In general, timestamps require an infinite range of values, which is not available on computers. A state-of-the-art implementation for timestamps is to simply use a counter. Overflows of such a counter ("wrapping") can cause an incorrect execution because it makes timestamps reappearing and therefore non-distinguishable. Increasing the value range of a counter, i. e., its bit-width, will decrease the probability of such faults, but it does not avoid the issue. Complete correctness is only provided by methods that can be reduced to the following idea: the value range is split into subranges. When the global timestamp shifts from one subrange to another one, a synchronization mechanism ensures that all old timestamps are removed, e. g., by completing all tasks that use old timestamps. The use of timestamps was motivated to avoid synchronization for performance reasons. Hence, solving problems related with timestamps by synchronization is pointless. Since correctness without additional synchronization is however not possible, this approach has not been implemented.

5.3.2 Energy Consumption

Another important aspect concerning speculation is the increase of energy consumption. While correct speculations can speedup the execution of a program, wrong speculations will be simply neglected without any performance benefits. As a consequence, the energy that was used to execute such a speculation was wasted. Hence, the presented approach is *not* intended to be used in low-power systems, but rather in high-performance systems. Despite of the fact that the logic for speculations requires more energy, speculation techniques are still commonly used in desktop and server PCs, containing e. g., Intel x86(-64) and AMD processors. Considering energy consumption, Intel's VLIW architectures, i. e., the Intel Itanium, would have been definitely the more reasonable solution. Nevertheless, dynamic processors with speculation still dominate the market. The reason is found in the capability

to save time, which is an important economical aspect. Shorter development processes result in shorter time-to-market intervals and a potential better profit, i. e., an improved economy. To this end, the decision to use the speculative approach depends on whether performance or energy consumption has the priority. Independently, our software solution provides an invincible advantage compared to the hardware solution: our OOO execution is scalable (window size can be set at the beginning of the program) and its speculation can be switched on and off. Beyond that, the task selection in our speculation dynamically adapts to a desired hit ratio. To conclude, our software solution allows much more customization compared to hardware solutions.

5.3.3 Execution Time Analysis

The advantage of this approach is its highly dynamic adaption to available hardware, which is however a disadvantage for worst and average case execution time estimations. Since there are no assumptions about the hardware and especially about its performance, a precise execution time cannot be determined. The speculation complicates to predetermine the execution time because its benefits with respect to the performance generally depend on the actual input values. A general worst case execution time can be easily determined by considering a SDFG that only allows sequential execution of its tasks. In addition, if all speculations fail, the final execution time will correspond to the sequential execution time of all tasks including the overhead for scheduling and speculation checks (which depends on the final hardware).

Our approach still has a broad range of applications where the average execution time is of more interest than the worst case execution time. For instance, multimedia applications have deadlines, but no hard deadlines like in safety-critical applications. Moreover, our Averest framework allows us to develop hardware, and one would expect simulations in software to be as fast as possible.

To this end, this approach has to be considered as a generic throughput optimization for systems that do not have to comply with hard deadlines.

5.3.4 Multiple Speculations per Task

The necessity to store speculated values in a structure which is separated from the CBS allows us to use more than one speculation with different values for the same task. A similar technique is predicated execution and can be found in some VLIW architectures like the Intel Itanium. The predicated execution is basically the execution of an instruction that depends on a condition – the predicate. The result of the instruction is calculated, even if the predicate has not been evaluated, yet. The result is held back until the value of the predicate becomes known. An effective implementation requires a speculation that provides varying values which is not yet available.

5.3.5 Caching Speculation Results

In our implementation, we use the time insensitive check as explained in Section 5.2.3. After a check for the correctness of a speculation has been completed, the speculation result is removed from the speculation buffer. However, the time insensitive check allows us to keep the result in the buffer and to reuse it for later computations, i. e., we could use it as a speculation result of other iterations of the same node. Moreover, we could also keep results of actual node executions as speculation results in the speculation buffer. In particular, this idea is based on Richardson's approach in [151] which uses a result cache to speed-up time-consuming functions with frequently used input sets. Note that Richardson's approach did not consider speculation at all. For timing issues, we had to postpone this idea.

5.4 Closing Remarks

This chapter considered an approach to speed-up the execution of DPNs by techniques that are already used in processor architectures. In particular, we considered OOO execution that allows nodes to compute next reactions before the current one as long as the results are finally given to the environment in the correct order. Moreover, we considered in detail how speculation which is a widely used technique in processor architectures, e.g. for branch prediction, can be used to predict input values so that nodes can be executed before the real input values are available. While most related work is based on specialized hardware, our approach is completely implemented in software, thereby addressing a broad field of multicore processors. Moreover, our techniques are completely automated, i. e., no manual work has to be done by the programmer. As a main feature of our approach, we emphasize the flexibility in terms of the number of processing units (e. g., cpus, cores), i. e., there is no need to repartition a given DPN when the target system is extended or changed.

Chapter 6

Implementation

This chapter discusses some experiences that have been made during the implementation of the synthesis tools. Note that, this chapter is *not* a guide or a tutorial to (re)program the compiler. The concepts to create parallel code from synchronous programs have been presented in the preceding chapters. This also includes some pseudo code which allows a straightforward implementation.

Beside the implementation of the concepts, some interesting details, issues and potential optimizations came up which we are going to discuss here:

- Implementation of a lock-free FIFO queue: Section 6.1 explains a specialized FIFO queue implementation for shared memory systems that is made to DPN specifications.
- Non-blocking communication in MPI can cause exception due to full buffers. This issue is addressed in Section 6.2.
- Quitting a multi-threaded program: Section 6.3 discusses the issue to safely quit a multi-threaded application.
- Weak Memory Models: In shared memory systems, the weak memory models allow processors to apply changes to the memory out-of-order with some restrictions. This should result in a better performance, but requires special handling in software programming. Section 6.4 gives some essential information about weak memory models and how to take care of the presented approaches.
- Partitioning: TLP requires coarse-grained parallelism. A system that provides fine-grained parallelism has to be partitioned to allow a distribution to architectures that are based on threaded execution. This issue has not been tackled in this thesis, but we give a short discussion on that topic in Section 6.5.

6.1 Lock-Free Single-Writer-Single-Reader FIFO Queue

The message passing approach in Chapter 4 has been implemented for SMP systems using PThreads and for distributed memory systems using MPI. This section explains a special-

```
1: class P2PQueue < Type T > {
2:   T[] buffer;
3:   unsigned int readPos;
4:   unsigned int writePos;
5:   Lock l;
6:   Condition bufEmpty(l);
7:   Condition bufFull(l);
8:   unsigned int size;
9:
10:  constructor (unsigned int s) {
11:    size = s;
12:    buffer = new T[size];
13:    readPos = 0;
14:    writePos = 0;
15:  }
16:
17:  void push(T a) {
18:    l.acquire();
19:    while (writePos == ((readPos + size) mod (2 * size))) {
20:      bufFull.wait();
21:    }
22:    buffer[writePos mod size] = a;
23:    writePos = (writePos + 1) mod (2 * size);
24:    bufEmpty.notifyOne();
25:    l.release();
26:  }
27:
28:  void pop(T &r) {
29:    l.acquire();
30:    while (readPos == writePos) {
31:      bufEmpty.wait();
32:    }
33:    r = buffer[readPos mod size];
34:    readPos = (readPos + 1) mod (2 * size);
35:    bufFull.notifyOne();
36:    l.release();
37:  }
38: }
```

Figure 6.1: Implementation of a bounded FIFO queue for multiple writing and reading threads.

```
 1: class P2PQueue < Type T > {
 2:   volatile T[] buffer;
 3:   volatile unsigned int readPos;
 4:   volatile unsigned int writePos;
 5:   unsigned int size;
 6:
 7:   constructor (unsigned int s) {
 8:     size = s;
 9:     buffer = new T[size];
10:     readPos = 0;
11:     writePos = 0;
12:   }
13:
14:   void push(T a) {
15:     while (writePos == ((readPos + size) mod (2 * size))) {
16:       pthread_yield();
17:       LoadFence();
18:     }
19:     buffer[writePos mod size] = a;
20:     StoreFence();
21:     writePos = (writePos + 1) mod (2 * size);
22:   }
23:
24:   void pop(T &r) {
25:     while (readPos == writePos) {
26:       pthread_yield();
27:       LoadFence();
28:     }
29:     LoadFence();
30:     r = buffer[readPos mod size];
31:     readPos = (readPos + 1) mod (2 * size);
32:   }
33: }
```

Figure 6.2: Implementation of a bounded FIFO queue for uniquely determined writing thread and uniquely determined reading thread.

ized queue implementation that is custom-tailored to the usage in SMP systems. A FIFO queue in a DPN is characterized by its uniquely determined writing node and a uniquely determined reading node. Hence, in our implementation, where each node is mapped to an own thread, each buffer is written by a uniquely determined thread and read by a uniquely determined thread.

A commonly used implementation of a bounded FIFO queue is depicted in Figure 6.1. Such an implementation usually declares a buffer with a fixed size, e. g., an array, for storing the elements. The most important methods for such a queue q are $q$push() to store an element in and $q$pop() to load and remove an element from the queue. Multiple threads may access a queue, which leads to potential races. Hence, access to the queue must be synchronized using common synchronization elements, e. g., locks and conditions. These synchronization elements block a reading thread in case that a value is missing and a writing thread in case that the buffer is full.

Our particular synthesis to PThreads allows us to consider two crucial characterizations of the created code: (1) The writer and the reader for each buffer are uniquely determined, (2) The created threads are light-weight. The latter implies that communication in each buffer is done in short time intervals. Hence, if a writer attempts to write a value and fails due to a full buffer, we can assume that at least one value will be consumed "soon". Analogously, if a reader attempts to read a value and fails due to an empty buffer, we can assume that a value will be produced "soon". *Soon* in the previous context means that a spinning block is better compared to switching the thread state with respect to the performance. Moreover, PThread supports a function to release (yield) a processor, which means that a context switch is enforced without suspending the calling thread. This allows other runnable threads to continue their execution, i. e., threads that can be executed but have no assigned processor are then scheduled to a processor. At the same time, it reduces wasting of computational power due to spinning blocks.

Based on Lampert's peer-to-peer queue [104, 105], we implemented a queue that is custom-tailored to DPN execution. The implementation of this queue is printed in Figure 6.2. The principle of this implementation is to use a ring buffer with a fixed size and two pointers which are exclusively written by the writing and respectively reading thread. Whenever the write position is *size* elements ahead, then the queue is full (line 15). If both pointers have the same value, then the buffer is empty (line 25). In case that a value has to be written and the buffer is full (lines 14ff), the writing thread is stalled by yielding the processor (line 16). If a buffer is not full, the value is written to the buffer and afterwards the pointer is updated. This signalizes a reading thread that the value is now available. Hence, the ordering of these writes is important.

The compiler might try to optimize code generated from this source, e. g., by exchanging read and write orders of variables that are written and respectively read by other threads. The keyword "volatile" is used in C to avoid these optimization.

Other approaches to implement a lock-free queue are based on compare-and-swap (CAS) instructions [130], which are usually more expensive in their implementation than basic memory load and store instructions.

6.2 Nonblocking Communication in MPI

In blocking communication, the phase of communication and computation are clearly separated. Hence, a node gets idle in calls to communication functions until a communication

process has been completed. In contrast, our synthesis to create MPI code uses nonblocking communication. This enables overlapped communication and computation, which is a technique to hide communication latencies. Communication is initiated in advance, such that it can be handled in background. Meanwhile, the node can proceed its execution of computations. The advantage is clearly a better utilization of computational power [121].

One issue that has to be tackled in our MPI applications is the gap between actually limited buffer size and the communication that handles buffers as if their size is unlimited. In our particular used MPI implementation, i. e., OpenMPI, and MPI environment, i. e., x86 multi-core cluster, the message buffer provided by MPI to buffer incoming messages is limited. Moreover, nonblocking sends are eventually nonblocking, which means that there is no stall if the buffer of a receiver is "full". In other words: when a value is sent to the reader, it is assumed to have enough space to store that value. As long as that value is not read, it remains in the reader's buffer. Due to decoupled execution of the nodes and potential unbalanced computational effort, a node may send tokens much faster than its respective antagonist consumes them. Hence, the buffer is filled up until it is full and even worse, until it overflows, which will yield an exception. This issue can be handled in different ways.

One can increase the buffer size, which will clearly not solve the problem but only delay its symptoms. Another solution is to use a different communication mode of MPI, namely synchronous communication. This introduces an acknowledge of the reader, which ensures that the reading node received the sent token. Obviously, this requires additional bandwidth due to the required hand shake. Moreover, this limits the decoupling of nodes, which is not an option for our synthesis. The second solution considers also a hand shake protocol which is global and only *barely* used. *Barely* means, that the hand shake is not required for each single communication, but for a global synchronization. This is implemented by calling a barrier (MPI_Barrier in each node in each nth macro step). This provides a compromise between performance and bandwidth loss while preserving functionality. The latter is obviously important to guarantee a safe execution of an application. The maximal value of n depends on the application and the buffer size provides by MPI.

6.3 Quitting Multi-Threaded Programs

This section is about quitting a multi-threaded application safe and clean. At a first glance, this seems to be an easy task, but it turned out to be subtle to allow a clean exiting of a decoupled program. Although a reactive system can basically run for an infinite time, most applications will compute a result and then subsequently quit. Especially benchmarks need the control of stopping themselves. After calculating a result or reaching a specific state, they have to be able to initiate a safe stop for two reasons. First, a safe stop ensures that the application can do cleanup work, e. g., quitting threads, releasing allocated memory, flushing buffered data to disk, etc. A safe stop also ensures that we are able to measure the execution time.

The very first step is to specify how an exit should be triggered in an application that was created by our tools. We have the different options that are discussed in the following. We could trigger the exit

- *from the environment (input reader or output writer):* We use dedicated functions to read inputs from or write outputs to the environment. These functions have to be defined by the environment, i. e., by the programmer of the environment. They can for instance read sensor data or send control values to actors using library functions.

  We leave the decision whether to stop an application to the environment. It may trigger a stop by setting a stop flag which is handed to the previously mentioned functions. If the stop is triggered by the function that reads inputs from the environment, then the application has to continue execution until all input sets have been processed. This includes the input set, where the stop flag has been set. The second way to stop an application is to set the stop flag by the function that writes output values to the environment. In that case, the execution can be stopped immediately.

  The idea to stop an application from the input or output function originates from the manifoldness of applications. For instance, pure streaming applications will feed a system with a number of input sets. This number is not necessarily known in advance. Hence, it is necessary to dynamically stop the application as soon as all input sets have been given to the application. Examples for this kind of applications are Signal Delay (see Section 7.1.4) and Discrete Fourier Transform (DFT) (see Section 7.1.5), which can be found in the benchmark chapter.

  The computation of outputs for a given set of input values does not always have to be instantaneous. In particular, some applications trigger calculations which need an unknown number of macro steps to compute the output for a given input set. In that case, the output function will get a signal when the computation of the result has been finished. It is reasonable to quit this kind of application from the output function which receives the signal of a finished computation. Examples for this kind of applications are Height Field Renderer (see Section 7.1.11) and Ray Tracer (see Section 7.1.13).

  The idea of quitting an application as described above is the way how we implemented the triggering of a stop. Other options are discussed in the following.

- *by executing a specific guarded action:* This option is currently not supported by our language. More important is that it would require to add concrete semantics on how to execute such an action in a synchronous program. One could consider to add a *strong exit* and a *weak exit*. The former would require to not execute any other actions, and therefore to not produce any outputs. The latter would allow to finish the execution of the current macro step, i. e., to produce an output set for that macro step. Due to the need for additional semantics in the source language, we neglect this option.

- *on reaching a specific label in the program:* This is very similar to the previous option. Exiting on reaching a specific label would end up with the same issues from the previous point: the semantics has to be specified to either execute all remaining actions in the current macro step or not.

The last two solutions would increase the amount of communication: a correct execution will require to execute an action only if an exit action or label has not been reached. Concerning parallel execution, this would not only increase communication overhead but also limit the parallelism: An action can only be executed, if it is known whether to continue the execution of the program or not. To this end, our solution relaxes this condition, i. e., we neglect additional communication at the costs of potentially doing computations for macro steps that must not be executed.

6.3.1 Task-Based Execution

In task-based execution, particularly OpenMP and SmpSS, quitting of applications is straightforward. Parallelism is only used within macro steps, i. e., the program is synchronized between macro steps. As explained in Section 3, the synthesized program executes a loop which starts with reading variables from the environment, which is a sequential task. Afterwards, the guarded actions are executed using potentially available parallelism. At the end, potential parallel threads converge to a single thread, i. e., the program returns to sequential execution. The computed outputs can then be written to the environment. Before a new iteration is started, the synthesized program is able to check, whether a new macro step has to be executed.

6.3.2 Message Passing

Currently, there is no similar work about the topic on quitting multi-threaded applications but the state-of-the-art documentation of some APIs and programming guides. They only give a rough guide for our concern. A multi-threaded program consists of several concurrently running threads. Quitting a multi-threaded application requires to join all threads, running cleanup procedures and finally to exit the application.

Running an exit command inside a thread which is concurrently running with other threads, is considered to be an unclean termination. This will end up with a concurrent execution of the cleanup procedure and other threads that still do computations. The cleanup will usually release memory that might be currently in use by other threads. Hence, this will likely result in an exception. Moreover, an exception during the cleanup phase may lead to loss of data, e. g., buffered data may not be flushed properly to a file.

Communication through FIFO queues allows our threads to run decoupled, which particularly means that they can process data from different macro steps. Communication to and from the environment is done by one of the threads. Hence, a stop signal must be somehow communicated to all other threads. In an SMP architecture, this can be done using a global variable. Whenever a new iteration of a node is to be executed, the global stop flag is checked. The idea is basically to stop the execution of a thread in case that

the flag is set. However, this is not sufficient to avoid deadlocks. Consider the following scenario: A node starts an iteration while the stop flag is not set. Next, it executes a pop() to read a required input and is blocked due to non-available tokens. Meanwhile, another thread sets the stop flag, which was actually done by a function that communicated with the environment. Now consider a third thread, particularly the thread that produces the token that the first thread is waiting for. Assume that it currently has finished executing an iteration and is going to check the stop flag which is now set. Hence, the third thread will quit, while the first one is blocked and waiting for input values.

This issue can be solved by letting all threads execute the same number of iterations. In our created DPNs, this is a sufficient condition to ensure that all nodes can complete an iteration without being blocked. To achieve a state where all nodes executed the same number of iterations, the mechanism to stop an application is printed in Figure 6.3. In addition to the global stop flag *stop* (line 1), we add a global macro step counter *numItToQuit* (line 2). When *stop* is set, *numItToQuit* determines the number of iterations per node that have to be executed to enable a safe stop. Each node that recognizes the set stop flag, will register to the quit mechanism. Registered threads are tracked using additional variables which also require a mutex and a condition for synchronization (lines 3-5,9). The execution of a thread is left unchanged (lines 11f). As described above, a thread that receives a stop signal from the environment sets the global stop flag, which is done by existing code (line 12). During the execution, a node must track the number of iterations that have been executed (lines 8,13). After the execution of a single iteration, each node checks whether the global stop flag is set (line 14). If the flag is set, the actual stopping mechanism begins.

The big idea is to register each thread, when it recognizes the set stop flag for the first time (lines 16ff). During the registration, the variable *numItToQuit* is iteratively adapted (lines 17-20) and finally set to the largest number of iterations done by a node in the system. Each thread individually decides whether to continue execution or to wait for a collective stop (line 27-32). An additional count for *numItToQuit* (lines 17f) enforces a thread to execute an additional iteration. This is required to release potential blocked threads, which may happen as described by the exemplary scenario above. When all threads are registered, the execution is collectively stopped (lines 30f).

6.3.3 Out-of-Order Execution

To ensure a safe cleanup in the OOO execution, we add an additional member to the CBS which contains the value of the stop flag for the corresponding macro step. The head of the CBS is always removed in order. Hence, after removing the head for a macro step, where the stop flag is set, the execution can be stopped. This is particularly done by removing all pending tasks from the task queue and adding dedicated stop-tasks. These stop-tasks represent special token. When a worker receives a stop-task, it stops the execution. The main thread of the OOO execution is responsible for initialization and the cleanup. After the execution of the worker threads has been started, it will wait until all worker threads stop their execution. Afterwards, cleanup work can be done.

```
 1: bool stopFlag
 2: unsigned int numItToQuit
 3: pthread_mutex_t quitMutex
 4: pthread_cond_t quitAllRegistered
 5: unsigned int threadsRegistered
 6: thread Node () {
 7:   bool run = true
 8:   unsigned int it = 0
 9:   bool registeredToQuit = false
10:   while (run) {
11:     // execute behavior of node
12:     ⋮
13:     it = it + 1
14:     if(stopFlag) {
15:       pthread_mutex_lock(quitMutex)
16:       if(¬registeredToQuit) {
17:         if(numItToQuit < it + 1) {
18:           numItToQuit = it + 1
19:           pthread_cond_broadcast(quitAllRegistered)
20:         }
21:         threadsRegistered = threadsRegistered + 1
22:         if(threadsRegistered == NUMNODES) {
23:           pthread_cond_broadcast(quitAllRegistered)
24:         }
25:         registeredToQuit = true
26:       }
27:       while (it ≥ numItToQuit ∧ threadsRegistered < NUMNODES) {
28:         pthread_cond_wait(quitAllRegistered, quitMutex)
29:       }
30:       if(it ≥ numItToQuit ∧ threadsRegistered == NUMNODES) {
31:         run = false
32:       }
33:       pthread_mutex_unlock(quitMutex)
34:     }
35:   }
36: }
```

Figure 6.3: Stop mechanism in threads of DPN applications.

6.4 Weak Memory Models

Modern processors use weak memory models [129] to improve their performance, which give different cores different views on the memory accesses of other cores. Handling weak memory correctly is very subtle, and an extensive presentation of this topic is beyond the scope of this thesis. In the following, we focus on the particular problems in our context and their solutions (the interested reader is referred to [129, 166] for an overview on the topic).

First, consider a simple example. Assume that a particular core first updates a variable A and then it updates a variable B. In a sequential memory model, each core will see these updates in exactly the given order. In a weak memory model, in contrast, a core might see the change of B before A is updated in its cache.

Communication between threads requires sometimes to ensure that some memory accesses are performed in a specific order. This can be achieved using memory barriers (also called fences). In general, there are three types of barriers: the store fence ensures that all changes are committed, before further changes are written. The load fence ensures that the cache is updated before any further memory reads are done. Finally, the memory fence combines the store and load fences.

In the following, we discuss the specific topic of weak memory models with respect to the presented approaches. In general, we assume that each thread sees its own changes immediately, such that memory fences are only necessary to ensure correct communication between threads.

6.4.1 Task-Based Execution

The task-based execution considers two APIs: OpenMP and SmpSS. The memory model for OpenMP is defined in its reference manual [136] since version 2.5 [88]. Application of the OpenMP memory model with respect to data-flow analysis can be found in [89]. According to these papers, OpenMP provides implicit synchronization in barriers. In this thesis, we exclusively use (nested) parallel regions, which have an implicit barrier at the end of each region. Thus, each parallel region contains a memory fence (flush) operation that ensures that all written values are visible to all threads. In between a parallel region, there is no communication between single sections. Hence, there is no need to add further explicit memory fences.

6.4.2 Message Passing

In our message passing approach, communication is exclusively done using FIFO buffers. Variables in each thread are locally defined and invisible to other threads. To this end, the actual implementation of the FIFO queue is responsible to take care of weak memory models.

Our customized implementation of the FIFO queue in Figure 6.2 already contains the required fence operations to ensure correctness. A push function starts to check whether

the buffer is full (line 15). Access to *readPos* without a preceding load fence *may* result in an out-dated value which is considered as a non-optimal but still correct behavior. Non-optimal means that a thread assumes a full buffer which is actually not full. In contrast, adding a load fence at that position will enforce an update of the processors cache, which *definitely* results in lower performance. Neglecting the load fence leads in the worst case to stalling of the thread, which will trigger a load fence after the first iteration (line 17). After the value has been written to the buffer (line 19), we have to ensure that this write is visible to other threads (line 20) before the write position is updated (line 21). This is necessary because the update of the write position signalizes that a new value is available. After the write position has been updated, a store fence could be added to accelerate updates. Similar to the beginning of the function, neglecting the fence at that position may lead to non-optimal behavior but saves definitely a cache synchronization of the actual thread.

The pop works analogous to its antagonist: The function starts to check whether the buffer is empty (line 25). Similar to push, the write position of the antagonist is not updated and may lead to a non-optimal but still correct behavior. In the worst case, the function assumes an empty buffer although it may contain values. This will trigger a yield with a succeeding load fence to update the write position. Although the thread might have an up-to-date value of the write position, the buffer may still contain outdated values. Hence, a load fence is necessary to update the values in the buffer (line 29). Afterwards, a value can be read from the buffer. This has to be done before the read position is updated, which signalizes that the token already has been taken from the queue (line 31). A store fence could be added after the read position has been updated, to guarantee that the new value is visible to other threads. This would enforce a cache synchronization and would *definitely lead* to lower performance. Neglecting the store fence *might result* in non-optimal but still correct behavior.

6.4.3 Out-of-Order Execution

In our implementation, most variables in the CBS require atomic read-and-modify access, e. g., fetching and decrementing a counter. Since most processors have at least an internal RISC behavior, their instruction set does not provide atomic read-and-modify operations. We rely on functions or structures from other libraries that provide the required functionality. The template class `tbb::atomic` from the Intel TBB library allows the instantiation of atomic variables. Amongst others, this class provides also the fetch-and-decrement action used in our implementation of worker threads (see line 8 of Figure 5.6 and line 14 of Figure 5.8). The corresponding functions are responsible for considering weak memory models, such that we do not have to add further operations for accessing these variables.

Access to variables of the system is done using native operations, i. e., without any special functions to ensure memory consistency. Within the execution of a single task, this is fine, since it is executed by one thread which is assumed to have sequential memory consistency. Hence, consistency must be only considered after a task has been finished and other tasks may be scheduled, which might be executed by other worker threads. In particular, a worker has to ensure that all changes of a task are committed before any

succeeding task may read previously written variables. Therefore, a store fence is executed after the task execution. Conversely, a worker that removes a task description from the task queue must ensure that the system variables are updated before the task is executed. Therefore, a load fence is inserted before the task execution. The members "*iteration*" of the CBS and the variables *head* and *tail* require also explicit synchronization. Analogous changes to these variables have to be committed before any task is scheduled, and a store fence is put at corresponding places (see line 30 of Figure 5.6). An update of these variables can be done after an element is removed from the task queue. Since there is already a load fence, no changes are necessary.

One further detail is left: a worker might remove a head while another worker executes a task. Since the latter worker might trigger a removal of the head with outdated values in *head* and *tail*, it has to execute a load fence at the beginning of the head removal.

In our speculation, the communication especially relies on the pure memory communication without synchronization mechanisms. Hence, it is necessary to ensure memory consistency at certain points. For the speculation using the "Time Insensitive Check" (see Section 5.2.3), we extend the sequence for speculation as follows:

1. ***Load Fence (line 24 in Figure 5.8, optional)***: We recommend to use a load fence before input values are copied. This may reduce bad speculations due to outdated input values.

2. *Create a copy of valid input values and speculate about missing input values*: While the input values are copied to a local buffer, new updates may be done by other threads. Periodic updates during the copying process may result in more up-to-date inputs, but also in lower performance due to time-consuming cache updates of the processor, which depends on the final target architecture. We left memory fences at this point for performance reasons.

3. *Execute task*: During the speculation, the algorithm works on a local copy of the system which is not visible to other threads. Hence, the values will not change during the speculative execution. As a result, no memory consistency instructions are necessary during that time.

4. ***Store Fence (line 34 in Figure 5.8)***: Before the result of the speculation is put into the speculation buffer, a store fence might be necessary depending on the actual implementation. In our case, we store pointers into the speculation buffer which makes a store fence mandatory to ensure an update of the actual data.

5. *Store speculation result*

The existing OOO execution requires the following modifications:

- *Cleaning a CBS entry*: We must ensure that no data is reset after the timestamp has been updated. Hence, a store fence must be placed right before the timestamp is updated and after all other data of the CBS entry was updated (see line 23 of Figure 5.6).

- *Check of speculation results*: When a speculation result is available, a token is taken from the corresponding buffer and checked. As already mentioned, our specific implementation does not put the complete structure containing the speculation result into the queue, but only a pointer. Hence, to ensure that the data at that address is up-to-date, we must call a load fence after the token has been taken (see line 41 of Figure 5.8).

6.5 Partitioning

Multi-core systems use TLP, which is often considered as coarse-grained parallelism, while ILP is considered as fine-grained parallelism. Starting from a set of actions with fine-grained parallelism requires to apply a partitioning to get the coarser grain of parallelism. Graph partitioning is an own area of research which has not been tackled in this thesis.

Different tools have been developed for partitioning graphs, e. g., Metis [99], Scotch [53], JOSTLE [178]. To apply these partitioning tools successfully, one has to determine weights for computation and communication effort. This requires precise knowledge about the target hardware to estimate computation and communication costs. This is in conflict with our big goal, i. e., the model-based approach of development. In particular, we want to make decisions about the actual target hardware as late as possible. Moreover, some architectures are highly dynamic in the execution of instruction streams, e. g., the x86-processors that are used for the benchmarks in Chapter 7. To this end, partitioning to improve performance without details about the targeted hardware is hard to achieve. We therefore neglected the connection of a partitioning tool until now.

We use a fast and simple mechanism to create a coarse-grained DPN from an ADG description. Starting from a structured graph (see Chapter 3), we merge nodes (SGAs) that build sequences. This preserves execution order and avoids conflicts, e. g., deadlocks due to cycles built of immediate dependencies (compare legal partitioning in Definition 16).

Chapter 7

Evaluation

In this chapter, we evaluate the proposed approaches. All approaches have been implemented with the Averest[1] tool chain as shown in Section 1.2. This chapter is structured as follows: Section 7.1 introduces the benchmarked applications. It gives a detailed overview of the benchmarks which is important to draw conclusions from the runtimes. Section 7.2 gives an overview of the hardware systems that were used for the experiments. Finally, the results are presented in Section 7.3.

7.1 Benchmarks

In this section, we introduce a set of applications. To achieve representative results, we selected applications from different areas, particularly DSP, mathematical and graphical applications. Some of these are known to provide trivial parallelism, e. g., ray tracer (see Section 7.1.13), where all picture elements (pixel) can be computed in parallel. This can be used to observe the scalability of our synthesized code. Others provide characteristics to especially test specific optimizations, but still are usable in real-world applications. For instance, the signal delay (see Section 7.1.4) is basically a FIFO queue, which is used to validate the array optimization transport from Section 4.2.

7.1.1 Matrix Multiplication

This benchmark is a well known mathematical computation, which is used in many areas, mostly graphical and physical computations. Let A and B be two matrices given as follows:

$$A_{l,m} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, B_{m,n} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{l1} & b_{l2} & \cdots & b_{lm} \end{pmatrix}$$

The product $C_{l,n} = A_{l,m} \cdot B_{m,n}$ is then defined as $a_{i,j} = \sum_{k=1\ldots m} b_{i,k} \cdot c_{k,j}$ with $i \in \{1, \ldots, l\}$ and $j \in \{1, \ldots, n\}$. Each element of the target matrix can be independently computed from

[1] http://www.averest.org

```
 1: module MatrixMult([N][N]real ?matA,
 2:                   [N][N]real ?matB,
 3:                   [N][N]real !matC) {
 4:   loop {
 5:     for (k = 0..N − 1)
 6:       for (l = 0..N − 1)
 7:         matC[k][l] = sum (m = 0..N − 1)(matA[k][m] ∗ matB[m][l]);
 8:     pause;
 9:   }
10: }
```

Figure 7.1: Quartz implementation of the parallel matrix multiplication benchmark.

other elements. Hence, it serves as an example to analyze the scalability of a parallelization approaches. The Quartz code for the matrix multiplication is given in Figure 7.1. Note that the compiler will statically roll out the two `for`-loops. As a result, the result program will have $N \times N$ guarded actions to compute the target matrix. In our benchmarks, we set the dimensions of the matrices to the same value, i. e., $l = n = m = N$.

Beside software creation, synchronous languages are also suited for implementing hardware. To test and validate these implementations, many programmers may choose software as a testing platform. Parallel software can speed up the testing process, and therefore, it may reduce development time and costs. For that reason, we consider in the following a hardware implementation of a matrix multiplication.

A large circuit area would be required to implement a parallel matrix multiplication in hardware. Hence, a hardware implementation would be differently implemented to parallel matrix multiplication shown in Figure 7.1. Because the computations in a matrix multiplication are limited to adders and multipliers, one would limit the number of these units. An extreme case would be to use only one adder and one multiplier to compute the elements of the target matrix. Figure 7.2 shows such an implementation of a sequential matrix multiplication, where only one adder and one multiplier are used to compute the elements of the target matrix. As can be seen, the interface has to be modified because the semantics of the program changed. While the parallel implementation computed one matrix per cycle, the sequential implementation requires several cycles. The number of cycles required to compute one matrix multiplication depends on the dimensions of the matrix and is N^3 in our case. To signal the completion of a multiplication, the program sends the signal *ready* to the environment. Moreover, the start of the computation must be triggered explicitly from the environment by setting signal *start*. During the computations, the input matrices must not change. Beside the guarded actions to compute the control-flow in the sequential matrix multiplication, the actual computations on data are limited to a single guarded action. As a result, we expect that the sequential implementation will not benefit from any parallelization. Nevertheless, it represents a typical program that is going to be synthesized for hardware, e. g., field programmable gate arrays (FPGAs). Its

```
 1: module MatrixMultSequential(event ?start,
 2:                             [N][N]real ?matA,
 3:                             [N][N]real ?matB,
 4:                             event !ready,
 5:                             [N][N]real !matC) {
 6:   nat{N + 1} i;
 7:   nat{N + 1} j;
 8:   nat{N + 1} k;
 9:   loop {
10:     immediate await(start);
11:     i = 0;
12:     while(i < N) {
13:       j = 0;
14:       while(j < N) {
15:         next(c[i][j]) = 0;
16:         pause;
17:         k = 0;
18:         while(k < N) {
19:           next(c[i][j]) = c[i][j] + a[i][k] * b[k][j];
20:           next(k) = k + 1;
21:           pause;
22:         }
23:         next(j) = j + 1;
24:         pause;
25:       }
26:       next(i) = i + 1;
27:       pause;
28:     }
29:     emit ready;
30:     pause;
31:   }
32: }
```

Figure 7.2: Quartz implementation of the sequential matrix multiplication benchmark. In contrast to the implementation in Figure 7.1

```
 1: module LUDecomp(event ?start, [N][N]real ?a, b, l, u, event !ready) {
 2:   loop {
 3:     immediate await(start);
 4:     // store input matrix a in matrix b
 5:     for (i = 0..N − 1)
 6:       for (j = 0..N − 1)
 7:         b[i][j] = a[i][j];
 8:     // compute LU decomposition of b
 9:     for (k = 0..N − 1) {
10:       let piv = b[k][k];
11:       u[k][k] = piv;
12:       for (i = k + 1..N − 1) {
13:         l[i][k] = b[i][k]/piv;
14:         u[k][i] = b[k][i];
15:       }
16:       for (i = k + 1..N − 1)
17:         for (j = k + 1..N − 1)
18:           next(b[i][j]) = b[i][j] − l[i][k] ∗ u[k][j];
19:       pause;
20:     }
21:     for (i = 0..N − 1)
22:       l[i][i] = 1.0;
23:     emit ready;
24:     pause;
25:   }
26: }
```

Figure 7.3: Quartz implementation of the LU decomposition.

role in the benchmark suite is to analyze and measure the performance and overhead in these kinds of applications.

7.1.2 LU Decomposition

The *LU decomposition* or *LU factorization* of a square matrix A is its factorization into two triangular matrices, such that $A = L \cdot U$, where L is a lower triangular matrix and U is an upper triangular matrix:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nm} \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{23} & \cdots & u_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{mn} \end{pmatrix}$$

The LU decomposition is used in solving square systems of linear equations, to determine the determinant of matrices and to invert matrices [147, Chapter 2.3]. The algorithm has been taken from [56, Chapter 28.3] and its Quartz implementation is depicted in Figure 7.3. The program starts a computation as soon as the *start* flag is set by the environment. The computation itself takes N cycles to compute the result for a matrix with size $N \times N$. The number of guarded actions that will be fired per macro step varies on the iteration for the outer loop k. Hence, we will have a varying amount of computational effort for the partitioned program, which is an interesting property for benchmarking. The flag *rdy* signalizes that a computation has been finished. To this end, the main characteristics of the LU decomposition with respect to our synthesis approaches are: (1) It is working on large (scalable) data set, and (2) it has a varying amount of active guarded actions per macro step.

7.1.3 Square Root

The square root benchmark represents an exemplary hardware implementation to compute the square root of a given positive integer number. It is based on a digit-by-digit calculation which computes one bit of the result per iteration. The algorithm copes without multiplications and divisions, which are quite expensive to implement in hardware. Divisions by numbers that are a *constant* power of two can be implemented by bit shifting operations with complexity $O(1)$. Similar to the implementation of the sequential matrix multiplication, this benchmark represents a hardware implementation that has to be tested by synthesizing it to software. The actual implementation uses bit vectors, while the algorithm in Figure 7.4 uses integer numbers. Bits and bit vectors can be considered as hardware representation of data, while integer numbers abstract from hardware and provide a mathematical representation of values. Usage of integers provides more readability compared to the actual implementation.

7.1.4 Signal Delay

In audio processing, a delay is often used to create hall, echo and chorus effects. It consists of a ring buffer which is permanently filled with audio samples. The read position varies and can be chosen between 0 (no effect) and the number of samples in the ring buffer (maximum delay of the signal). The computations done during the signal delay are independent of the buffer size.

This benchmark is especially of interest for the array optimization approach in Section 4.2. The data size is scalable, while the computational effort is constant, i. e., the number of guarded actions is independent of the buffer size. Hence, we can use the delay

```
 1: module SquareRoot(event ?start, event nat ?a, nat !r, event !ready) {
 2:   nat n;          // intermediate result of the root
 3:   nat ns;         // = n^2
 4:   nat n2i;        // = 2^{2i}: used for iterative approximation of n.
 5:   nat ni;         // = 2^i: used for iterative approximation of n.
 6:   nat remainingA; // aCopy - n^2
 7:   nat rTemp;
 8:   while(true) {
 9:     immediate await(start);
10:     rTemp = 0;
11:     ns = 0;
12:     remainingA = a;
13:     if(a == 0) {ni = 0;} else {ni = exp(2, N/2);}
14:     n2i = exp(2, N - 2);
15:     while((ni ≠ 0) ∧ (remainingA ≠ 0)) {
16:       if(n2i + ns ≤ remainingA) {
17:         next(remainingA) = remainingA - n2i - ns;
18:         next(rTemp) = rTemp + ni;
19:         next(ns) = ns/2 + n2i;
20:       } else {
21:         next(ns) = ns/2;
22:       }
23:       next(n2i) = n2i/4;
24:       next(ni) = ni/2;
25:       pause;
26:     }
27:     emit (ready);
28:     r = rTemp;
29:     pause;
30:   }
31: }
```

Figure 7.4: Quartz implementation of the digit-by-digit square root algorithm which computes $r = \sqrt{a}$. For readability, the printed implementation uses positive integer number, i. e., `nat`. The actual implementation which was benchmarked, uses bitvectors, i. e., `bv`$\{N\}$. N is the bitwidth of the input variable a.

```
 1: module delay(nat ?delay, real ?sample, real !out) {
 2:   [BufferSize]real samples;
 3:                nat writePos;
 4:                nat readPos;
 5:   writePos = 0;
 6:   loop {
 7:     next(samples[writePos]) = sample;
 8:     next(writePos) = (writePos + 1) modulo BufferSize;
 9:     readPos = (writePos − delay) modulo BufferSize;
10:     out = samples[readPos];
11:     pause;
12:   }
13: }
```

Figure 7.5: Quartz implementation of the signal delay. *BufferSize* is the size of the ring buffer and determines the maximum number of samples that a signal can be delayed.

benchmark to spot those approaches, where performance loss occurs due to communication overhead.

7.1.5 Discrete Fourier Transform (DFT)

The theory of signal processing requires to apply filters to a signal for various reasons. Filters can be used to simulate the behavior of physical transport medias, e. g., wires, or the behavior of a circuit. A filter function is described by the signal that is output if the filter is stimulated with a impulse signal, i. e., the filter function is the impulse response of the filter. A mathematical computation of a signal that is output by the filter by any input signal can be done using the convolution integral of the input signal and the filter function. Application of further filters requires to calculate the convolution integral for each filter function and the corresponding intermediately output signal. Obviously, this can be an expensive task depending on the input signal and the filter functions.

A transformation from time domain to frequency domain allows an easier handling with respect to the mathematical operations: instead of integration, it is sufficient to multiply the filter functions. This is derived from the superposition theorem, i. e., if a system is stimulated by signals A and B and the system response is A' and B', respectively, then the system response of the input signal $A + B$ is $A' + B'$. Hence, the system response for an arbitrary signal $s(n)$ can be computed as follows: First, s is rewritten to a sum of functions $S(k)$. Next, the response for each part of the sum (i. e., $S(k)$) is computed, and finally, the sum of these results builds the response of the system for stimulating it with signal $s(n)$.

The Fourier transformation [147, Chapter 12] generalizes the previous method to the representation of a function as an integral of sine and cosine waves over different frequencies. Digital processing considers only discrete functions, which leads to the discrete Fourier

transformation [164, Chapter 8]. Based on the same principle, it represents an arbitrary discrete function as a sum of sine and cosine waves:

$$S(k) = \sum_{n=0}^{N-1} s(n) \cdot e^{-i2\pi \frac{k}{N} n}$$

This does not only rewrite a function s to a sum of sines and cosines, but it represents also the frequency spectrum of the function. For that reason, the DFT is described as a transformation from time to frequency domain. The back transformation from frequency domain to time domain is called inverse discrete Fourier transformation (IDFT). It is accomplished by using the following formula:

$$s(n) = \frac{1}{N} \sum_{k=0}^{N-1} S(k) \cdot e^{i2\pi \frac{k}{N} n}$$

The actual benchmark first applies the DFT to a signal and afterwards the IDFT to reconstruct the signal.

7.1.6 Pitch Shift

The pitch shifter is a DSP application that is mainly known in the area of audio processing. Its function is to transform an audio signal in a way that the pitch of the sound is changed without modifying the playback speed. Although there is no exact mathematical solution for such a filter, approaches presented in [73, 142] follow the same idea: They transfer the function of the input signal from time to frequency domain and re-synthesize a pitch-shifted signal from the frequency function.

The length of an audio signal is usually unknown in advance or quite large, which results in high computational effort to accomplish the pitch shift using a DFT-IDFT. Therefore, most approaches [73, 142] use the overlap-add method [164, Chapter 18] to apply a filter to an input signal. This method splits the input signal into pieces of equal size, the so-called windows. The actual transformation is then applied to each window. Assume that each window contains N samples and the samples of a window are given as $s(n)$. To shift the pitch of the signal by factor β, we start with determination of the coefficients of the DFT as follows:

$$S(k) = \sum_{n=0}^{N-1} s(n) \cdot e^{-i2\pi \frac{k}{N} n}$$

Instead of applying a filter function to these coefficients, we synthesize the new signal by changing the frequency of the carrier waves by factor β:

$$s'(n) = \frac{1}{N} \sum_{k=0}^{N-1} S(k) \cdot e^{\beta \cdot i2\pi \frac{k}{N} n} \tag{7.1}$$

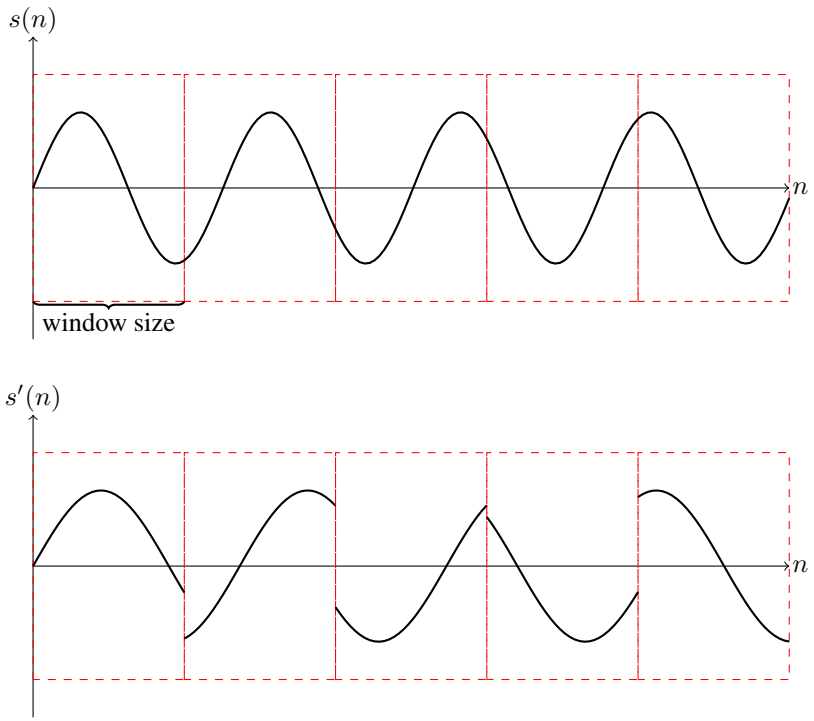


Figure 7.6: Operation principle of our *Pitchshift* benchmark. *Top:* An audio input $s(n)$ is chopped into equal-sized windows (red colored rectangles). *Bottom:* The resulting signal $s'(n)$ for $\beta = 0.7$, which is the pitch shift parameter. $s'(n)$ is obtained by scaling the playback speed for each window by β. This shows also the resulting artifacts at the edge of the windows.

Substitution of $k = \frac{k'}{\beta}$ results in

$$s'(n) = \frac{1}{N} \sum_{k'=0}^{\beta \cdot (N-1)} S(\frac{k'}{\beta}) \cdot e^{i2\pi \frac{k'}{N} n}$$

Next step is to determine $S(\frac{k'}{\beta})$:

$$S(\frac{k'}{\beta}) = \sum_{n=0}^{N-1} s(n) \cdot e^{-i2\pi \frac{k'}{\beta \cdot N} n}$$

Substitution of $n = \beta \cdot n'$ results in

$$S(\frac{k'}{\beta}) = \sum_{n'=0}^{\frac{N-1}{\beta}} s(\beta \cdot n') \cdot e^{-i2\pi \frac{k'}{N} n'} \tag{7.2}$$

Setting equation 7.2 into equation 7.1 results in the back transformed function $s(\lfloor \beta \cdot n \rceil)$, with $\lfloor x \rceil$ denoting the round to nearest integer function. The synthesized pitch shifted signal can be determined by

$$s'(n) \approx s(\lfloor \beta \cdot n \rceil) \tag{7.3}$$

$s(n)$ is defined for $n \in \mathbb{N}$, $\beta \cdot n$ has to be rounded if $\beta \in \mathbb{R}$, i. e., if β is chosen to be non-integer. This causes inaccuracies making the determination of $s'(n)$ only an approximation. Moreover, if β is chosen to be larger than 1, the faster playback of a signal may violate the Nyquist theorem. This means that frequencies in the original input signal, i. e., $s(n)$, that are equal or greater than $\frac{\text{sample rate}}{2\beta}$ will cause aliasing. This can be handled by prepending a low pass filter to the pitch shifter. An implementation of a low pass filter is part of our modular audio synthesizer (see Section 7.1.7), but has not been added to this benchmark, because we assume a high sample rate of the input signal compared to its frequencies.

Our implementation achieves the pitch shift of an audio signal by changing the playback speed of the windows. Due to the windowing of the actual input signal, the overall playback speed is maintained. Moreover, this pitch shifter works without DFT and IDFT but only FIFO buffers to cache the samples for windows. This makes it a cheap filter with respect to its computational effort. The downside of this simplification are artifacts that appear at the edge of windows (see bottom of Figure 7.6). We smoothened this issue by overlapping the windows and cross-fading the output signals of the windows in the overlapping range. To this end, our implementation of a pitch shifter has to be considered as a solution for understanding its operating principle.

7.1.7 Modular Audio Synthesizer

The modular audio synthesizer has been constructed as an application of different modules that have been build for audio processing. The collection includes the following modules:

- Various basic arithmetic operations, i. e., adders, volume fader and key note to frequency converter which is based on an exponential function.
- Wave generators of various kinds: sinus, saw tooth, triangle, square and white noise (random). They create a wave with a frequency that is given as input and output the generated samples.
- Envelopes: attack-release (AR) and attack-decay-sustain-release (ADSR) envelopes are used to modify the volume when playing notes. They are triggered by key-press events and output a slowly increasing and decreasing volume value, i. e., the envelopes are used to give synthesized sound a soft appearance.
- Fourier transformation and recursive or infinite impulse response (IIR) filters [147, Chapter 13.5]: generic DFT-IDFT is used to apply filter transformations in frequency domain as explained in Section 7.1.5. A generic implementation of a 2-pole recursive filter is also provided including four derivations: a band-stop, band-pass, low-pass and high-pass filter with variable resonance [164, Chapter 19].

The modularized implementation allows nearly an infinite amount of possibilities to create artificial instruments. The idea of building a modular electronic audio synthesizer originally came from Moog in the early 1960th [2]. Software implementations of this concept have been published in commercial and open-source variants [1, 4].

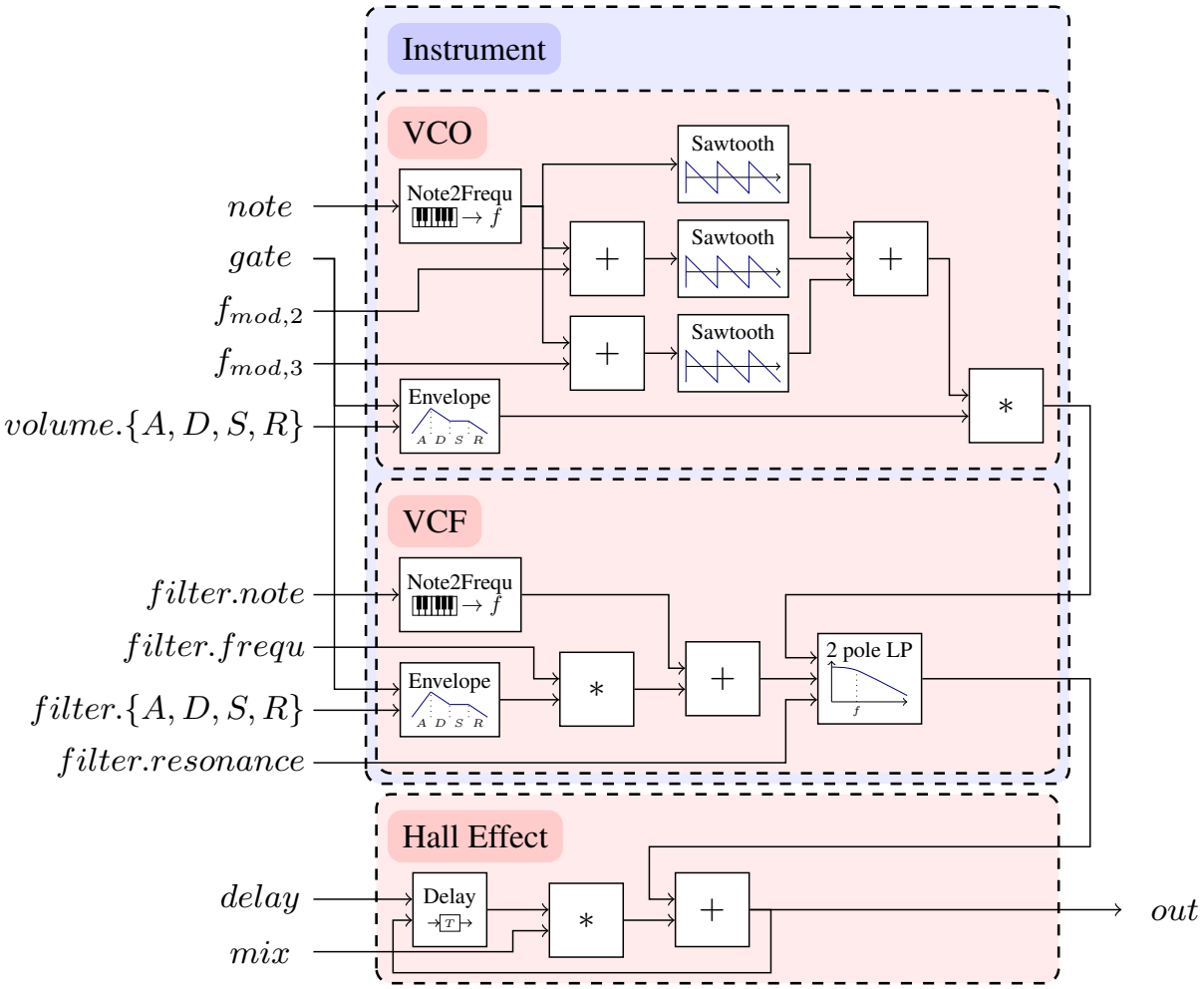


Figure 7.7: Structure of the benchmarked audio synthesizer. It consists of an instrument that can be instantiated several times to play polyphonic sounds, e. g., chords. Each instantiation of an instrument is build of a voltage controlled oscillator (VCO) and a voltage controlled filter (VCF). The hall effect is used to simulate echoes that appear in larger rooms or in big halls. To this end, this benchmark represent a practice oriented application.

The structure of the instrument that was benchmarked is given in Figure 7.7. It represents a voltage controlled oscillator (VCO) with a voltage controlled filter (VCF) to create the actual characteristic of the instrument. Three wave generators in a single VCO with slightly different frequencies give the base sound more natural irregularities and less synthetic character. The VCF is mainly used for its resonance which can be used for instance to form the sound at the beginning, i. e., to improve the characteristics when a key is touched. To create polyphonic sounds, i. e., different notes can be played at the same time, the instrument is instantiated several times. The number of instantiations has been used

to scale the benchmark. At the end of the instruments flow an audio delay gives some atmospheric hall.

The important characteristic of this benchmark for our synthesis tools are: Parallel and independent instantiations of the instrument provide small portions of parallelism, i. e., each instantiation of an instrument consists of a few mathematical operations that are independent of instantiations. The audio delay at the end is the practical application of the delay that was benchmarked in Section 7.1.4.

7.1.8 3D Geometry Transformation

The geometry transformation has been implemented as a streaming application. This benchmark creates a transformation matrix to rotate and translate a sequence of three-dimensional vectors and applies it to a sequence of three-dimensional vectors. The angles and the translation vector are read from the environment. A flag is used to explicitly signalize that a new transformation matrix has to be computed by the program. In each cycle, the program reads a three-dimensional vector from the environment, applies the transformation matrix to that vector and outputs the transformed vector to the environment in the same macro step.

Geometric transformations can be found in many three-dimensional applications, e. g., in ray tracing as described in [163, Section 1.7]. Generally, this is used to transform coordinates between different coordinate systems, e. g., from object coordinate system to world coordinate system to move an object in a modeled three-dimensional environment. Vectors in world coordinates have to be transformed to camera coordinates before it can be displayed using rasterization algorithms.

7.1.9 Image Scaler

The image scaler benchmark is a basic example for image processing. It resizes an image using bilinear interpolation. The intention of this benchmark was to provide a program that contains a lot of parallelism that should be easy to detect for a compiler. Hence, the scaling of the complete image is handled in a single step. In each macro step, the program reads a complete image from the environment, computes the resized image and outputs it to the environment in the same macro step. As a result, a lot of data transports are necessary in each macro step.

The resizing of images by factor s can be done using different interpolation techniques. The easiest one with respect to computation effort is *nearest neighbor interpolation*. For each pixel in the target image, the coordinates (x, y) of the target image are mapped to coordinates $(x', y') = (\lfloor s \cdot x \rceil, \lfloor s \cdot y \rceil)$ in the source image, where $\lfloor a \rceil$ denotes the rounded value of a, i. e., the nearest integer. The color of the corresponding input pixel is copied to the output pixel. Especially, when scaling images of patterns to a non-integer multiple size, this will result in images that appear incorrect and simply look strange.

Bilinear interpolation is another technique that already provides better results at (usually) affordable higher computational effort. The coordinates (x, y) of the output image are

mapped to coordinates $(x', y') = (s \cdot x, s \cdot y)$ in the input image. Let $C(x, y)$ be the color at (x, y) in the source image. $C_{\llcorner} = C(\lfloor x' \rfloor, \lfloor y' \rfloor), C_{\lrcorner} = C(\lceil x' \lceil, \lfloor y' \rfloor), C_{\ulcorner} = C(\lfloor x' \rfloor, \lceil y' \rceil), C_{\urcorner} = C(\lceil x' \rceil, \lceil y' \rceil)$ then define the colors next to the computed coordinates. The color of the output pixel is computed by linear interpolation of these colors. Therefore, the non-integer part of the coordinates which is defined as $(\delta x, \delta y) = (x' - \lfloor x' \rfloor, y' - \lfloor y' \rfloor)$ is used to fade between the given source colors:
$C_{out}(x, y) = (1 - \delta x) \cdot ((1 - \delta y) \cdot C_{\llcorner} + \delta y \cdot C_{\ulcorner}) + \delta x \cdot ((1 - \delta y) \cdot C_{\lrcorner} + \delta y \cdot C_{\urcorner})$.

7.1.10 Mandelbrot Set

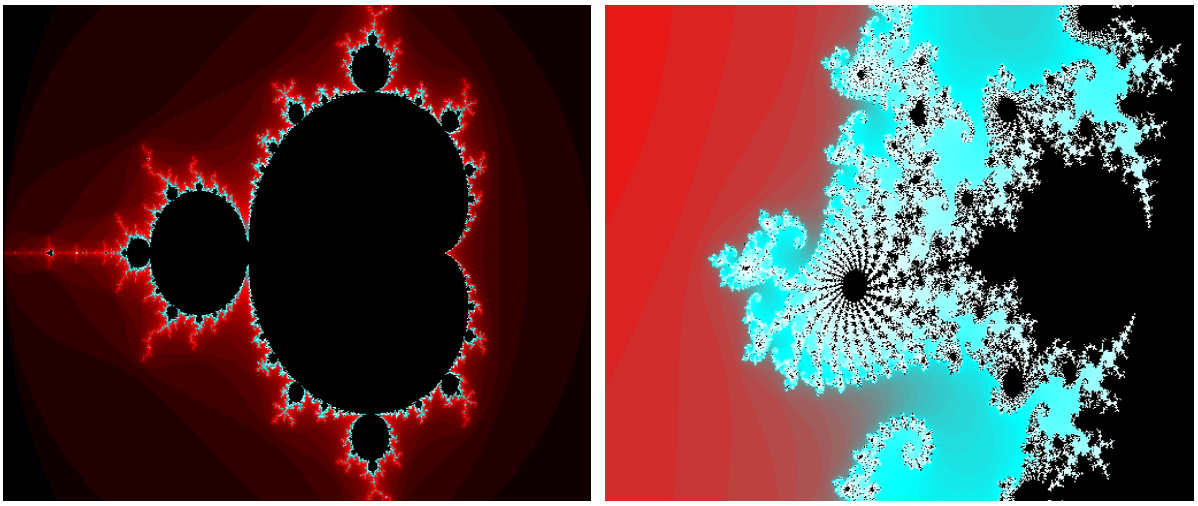


Figure 7.8: The pictures are the result of the *Mandelbrot* set benchmark. The left picture shows the Mandelbrot set for $\{c \in \mathbb{C} \mid -2 \leq \mathrm{Re}(c) \leq 1 \wedge -1 \leq \mathrm{Im}(c) \leq 1\}$, and the right picture shows the Mandelbrot set for $\{c \in \mathbb{C} \mid -0.7445 \leq \mathrm{Re}(c) \leq 0.130 \wedge -0.7427 \leq \mathrm{Im}(c) \leq 0.142\}$.

Fractal geometry is about sets that are quite easily defined by recursive functions, while the actual set usually has complex structures and cannot be described by equations or geometric shapes [72, Introduction]. The theory of chaos [80] also addresses these functions, because they produce values and sets that seem to be irregular, while a closer look unveils that the produced sets often contain some kind of regularities. One of the most famous sets described by such a function is the Mandelbrot set (see Figure 7.8), named after Benoît Mandelbrot. The Mandelbrot set belongs to the more generic Julia sets. They generally describe sets in the complex plane, i. e., sets of complex numbers. A formal definition can be found in [72, Chapter 14]: $J_0(f) = \{c \in \mathbb{C} \mid \text{family } \{f^k\}_{k \geq 0} \text{ is not normal in } c\}$. The basic meaning of this definition is to define a set by deciding for each complex number c, whether the recursive function f applied to c has a specific behavior. In most cases, the specific behavior is defined by boundedness, i. e., $f^k(c)$ must be bounded for all k.

Due to the definition of Julia sets on $\mathbb{C}$, these sets can be visualized in a two-dimensional plane, i. e., images can be produced from these sets. Considering Mandelbrot, Gleick writes:

"... A cataloguing of the different images within it or a numerical description of the set's outline would require an infinity of information. But here is a paradox: to send a full description of the set over a transmission line requires just a few dozen characters of code. ..." [80, Images of Chaos]. The main characteric of the Mandelbrot set is that many figures can be found that are similar but never equal to the one depicted on the left-hand side in Figure 7.8. This effect is often compared to things that appear in nature: Many things, e. g., trees, leaves or clouds, are very similar but never the same.

Formally, the Mandelbrot set M describes the set of all complex numbers $c \in \mathbb{C}$ for which the function

$$z_n(c) = \begin{cases} 0 & n = 0 \\ z_{n-1}(c)^2 + c & else \end{cases}$$

remains bounded. Formally: $M = \{c \in \mathbb{C} \mid \exists s \in \mathbb{R}. \forall n \in \mathbb{N}. z_n(c) \leq s\}$. It is known that s can be set to 2, i. e., if there is a $z_n(c)$ with $z_n(c) > 2$ for a $c \in \mathbb{C}$, the sequence $z_n(c)$ will not converge. Hence, to compute a picture as depicted in Figure 7.8, the program maps each pixel to a complex number c and iterates through the given sequence until $z_n(c) > 2$. To make such a picture computable, the number of iterations is limited, i. e., when n reaches a predefined maximum the corresponding sequence $z_n(c)$ is assumed to converge. Finally, the number of iterations done for a c, i. e., the smallest n with $z_n(c) > 2$ or the maximum value, is mapped to a color.

7.1.11 Height Field Renderer

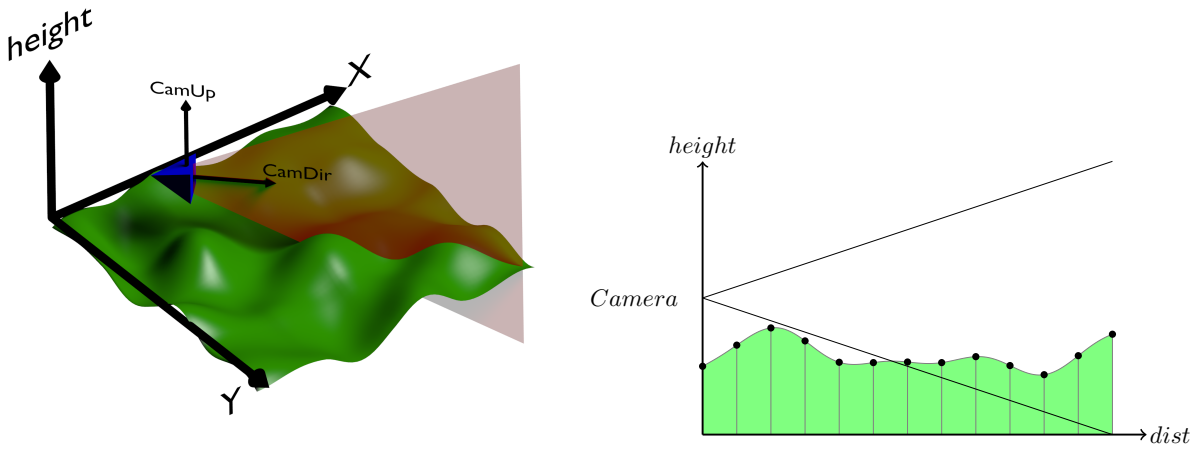


Figure 7.9: *Left*: The rendering algorithm slices for each column the landscape. *Right*: Each slice is scanned from front to back in discrete steps. The landscape height is mapped to its y coordinate in the image.

Different methods have been developed to visualize scalar functions that map two-dimensional vectors, i. e., functions defined as $f : \mathbb{R}^2 \to \mathbb{R}$. Common visualization techniques create a set of three-dimensional vectors by building a discrete set of two-dimensional points

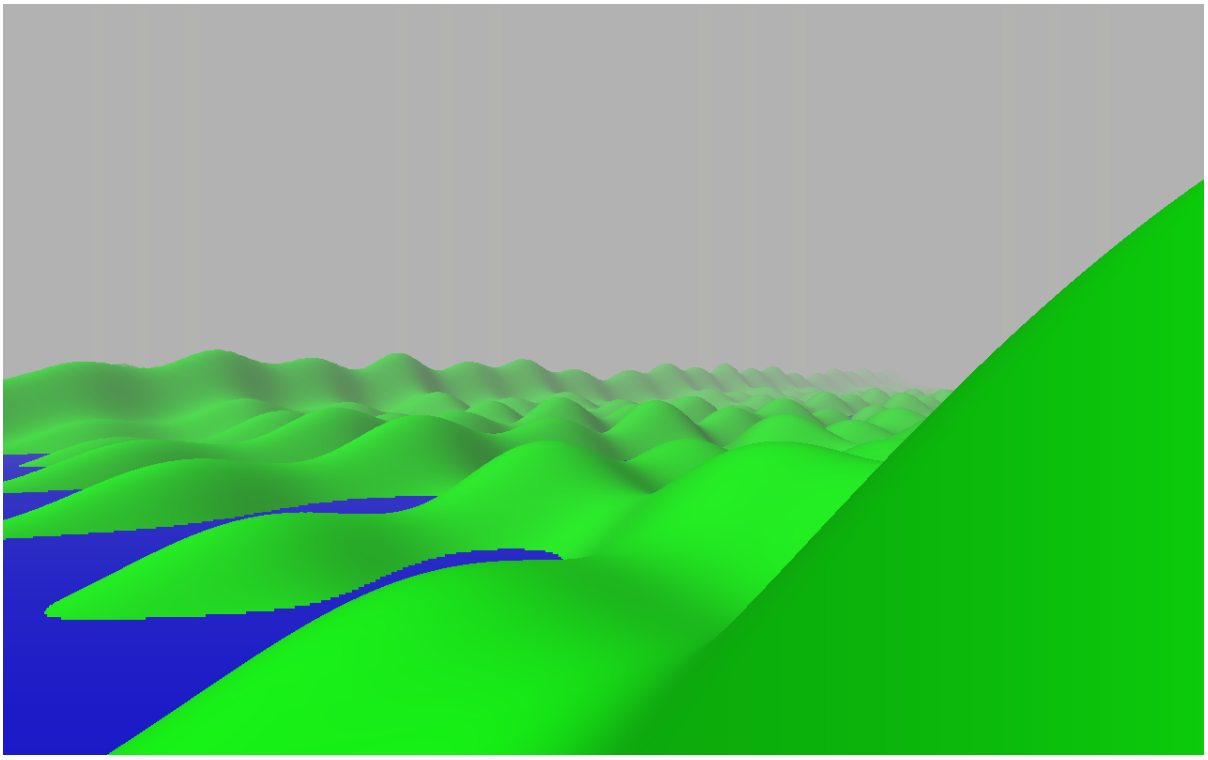


Figure 7.10: Picture rendered with the height field renderer benchmark.

and adding the height information from a given function $f : \mathbb{R}^2 \to \mathbb{R}$. Polygons, particularly quad faces, connect the set of vectors to a surface that represents an approximation of the given height field function. The polygons can then be rendered using graphic libraries, e. g., OpenGL, to create an image of the function.

A sub-field in this aera is the rendering of height fields, i. e., topographical functions of landscapes [180]. Rendering of height fields is used to visualize terrain in various applications, e. g., geographic information systems, video games, etc. Beside previously described rasterization method, these kind of applications focus on interactive fly-through, e. g., for flight simulations.

The rendering mechanism that is used here is a simplified version of [180]. A height field is given as a function that maps (x, y) coordinates to an elevation level which defines the height of the terrain at the given position. Figure 7.11 shows the pseudo code which describes the principle of the rendering mechanism. It has been abstracted from the actual implementation for better readability. Moreover, it makes some limitations to the camera parameters (given in line 2). In particular, it allows only yaw rotation, i. e., rotation about the camera's up vector as depicted on the left-hand side of Figure 7.9. The renderer is triggered by the signal *start* (line 5).

The rendering of an image is split into computations per column. The computations for all columns are independent of each other. Hence, the actual computations for all columns in the picture can be done in parallel (line 6). Each column in the target image is computed

```
 1: typedef tColor = {real r, real g, real b}
 2: typedef tVector2D = {real x, y}
 3: typedef tVector3D = {real x, y, height}
 4: module HeightFieldRenderer(event ?start, (tVector2D → real * tColor) ?field,
 5:                tVector3D cameraPos, real cameraDir,
 6:                tColor[][] !image, event !ready) {
 7:   while(true) {
 8:     immediate await(start);
 9:     for (x = 0..ResolutionX) {
10:       real dist;
11:       tVector2D dir;
12:       nat y = 0; // image coordinates: (x,y), where (*,0)=bottom
13:       let dirx0 = x/(ResolutionX/2) − 1.0;
14:       let diry0 = 1.0;
15:       dir.x = cos(cameraDir) · dirx0 − sin(cameraDir) · diry0;
16:       dir.y = sin(cameraDir) · dirx0 + cos(cameraDir) · diry0;
17:       dist = 1; // draw from front to back
18:       while(y < ResolutionY ∧ dist ≤ maxDist) {
19:         tVector3D pos;
20:         // determine position of the currently scanned point
21:         pos.x = cameraPos.x + dir.x · dist;
22:         pos.y = cameraPos.y + dir.y · dist;
23:         // map height of field at (pos.x, pos.y) to screen height
24:         pos.height = ResolutionY/2
25:                   +(field(p.x, p.y).height − cameraPos.height)/dist;
26:         // draw pixels if current point is visible
27:         while(y < pos.height ∧ y < ResolutionY) {
28:           image[x][y] = field(p.x, p.y).color;
29:           next(y) = y + 1.0;
30:           pause;
31:         }
32:         next(dist) = dist + 1.0;
33:         pause;
34:     }}
35:     emit ready;
36:     pause;
37: }}
```

Figure 7.11: Pseudo code for height field renderer in Quartz syntax. The code is abstracted for readability. The semantics is given in the synchronous model of computation. In addition, the function *field* can be applied several times within a macro step with different parameters without any conflicts or side effects.

by sending a ray that scans the height field. The direction is computed from the camera direction and the parameter x of the column. The ray vector and the height axis define a plane, particularly the slice which is given in three-dimensional and two-dimensional on the left-hand side and right-hand side, respectively, of Figure 7.9).

In a simple ray tracing algorithm, one would send a ray for each pixel in a column and check if it hits the height field. When the height field is hit, the color member of the height field function (see line 1 of Figure 7.11) at that position determines the color of the pixel that was traced. We use a simplified version which considers the steadiness of the height field and the limitations to the camera rotation. Consider the right-hand side of Figure 7.9 which depicts the sliced terrain in ray direction and height axis. Our algorithm computes a position (line 20f in Figure 7.11) using the camera position (line 2), the ray direction (line 11-14) and the distance to the camera (line 16 and 30), which determines the input parameter for the height field function (line 24). The result defines the elevation of the landscape at the given position. This means that we sample the height of the terrain along the ray from front to back. Each elevation level is mapped to a Y coordinate in image coordinate system (line 23f).

If the computed Y coordinate is above the last drawn terrain pixel, we draw a line from the last drawn pixel to the newly computed coordinate (line 26-29). This is explained as follows: The terrain is assumed to be solid, i. e., it has no caves. Hence, a line basically has to be drawn from the computed coordinate to the bottom of the image unless some part is hidden. This may happen because a column in the image is rendered from front to back, i. e., some part of the terrain that is closer to the camera may have already been drawn. Due to the steadiness of the terrain, a pixel is hidden if the adjacent pixel above is also hidden. Hence, it is sufficient to store the last drawn pixel in y (line 9 and 28), i. e., the pixel that is closest to the top of the image. Everything below y will hide any succeeding terrain. As a result, we only have to draw a line from the last drawn pixel at y to the newly computed coordinate $pos.height$. To limit the computational effort for this rendering technique, the distance for each trace is limited to $maxDist$ (line 17). To avoid a choppy look at the horizon of a rendered image, the actual color of a drawn line is faded to gray with increasing distance, thereby simulating fog (not printed in the abstracted implementation).

A rendered image of this benchmark is shown in Figure 7.10. A similar rendering technique has been used in some computer games in the early nineties, e. g., Comanche 1 and 2 from NOVA Logic.

7.1.12 Facility Renderer

Architectural interior rendering is used to walk through buildings, rooms, etc. It allows architects of buildings to get an impression of designed buildings and rooms before they have been built. Most of todays implementations will use commonly available 3d graphics accelerators to render images of a given scene. The implementation of our renderer includes the complete rendering process, and therefore, it does not require any external accelerator

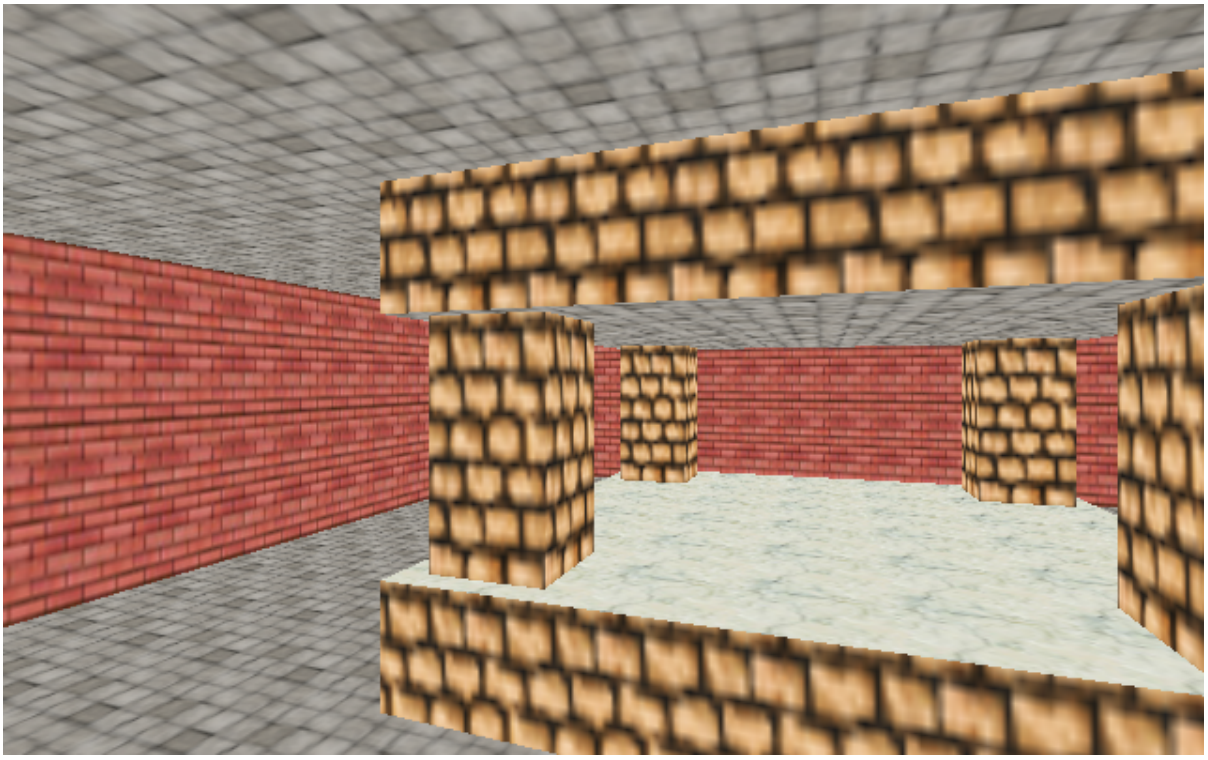


Figure 7.12: Picture rendered with the facility renderer benchmark.

hardware. A rendered image of this benchmark is shown in Figure 7.12. The same rendering technique has been used in some video games, e. g., Doom 1 and 2 from ID software [3].

The principle of the renderer is to do ray casting in a two-dimensional scene description and to map it to three-dimensional space. Similar to the height field renderer, this requires to makes some limitations to the camera rotation to simplify the rendering process. In particular, it allows only yaw rotation, i. e., rotation about the camera's up vector. Moreover, the renderer makes some limitations to the scene: (1) It allows only vertical walls and horizontal floors and ceilings, and (2) it prohibits multi-level structures, i. e., there is at most one floor and one ceiling at a time in top-view of the scene. The scene is given as a two-dimensional floor plan which is described by the data structures given in Figure 7.13. The floor plan (line 8) is defined by a set of lines (line 7) and sectors (line 4). Lines represent walls and enclose sectors which provide information about the height of the floor and the ceiling of a room. Walls may be transparent and connect adjacent sectors with different heights. An upper wall connects the ceilings and a lower wall connects the floors of the adjacent sectors. Without the lower and upper wall an image of a scene may look strange or unreal due to the unconnected floors or ceilings, respectively. Figure 7.15 shows the abstracted algorithm to compute an image of a given scene. Additional functions required by this algorithm are printed in Figure 7.14. The actual implementation has about 500 LOC. Therefore, the printed algorithm was shortened to sketch the basic functionality. For readability, we left the code for texturing which is printed in Figure 7.12, and the

```
1: typedef tColor = {real r, g, b}
2: typedef tVector2D = {real x, y}
3: typedef tVector3D = {real x, y, z}
4: typedef tSector = {real floorHeight, tColor floorCol,
5:                    real ceilingHeight, tColor ceilCol}
6: typedef tSidedef = {tSector sector, tColor upCol, midCol, lowCol}
7: typedef tLine = {tVector2D p1, p2, tSidedef s1, s2}
8: typedef tMap = {tLine lines[], tSector sectors[]}
9: typedef tIntersect = {real alpha, tLine line}
```

Figure 7.13: Structures used in the facility renderer.

```
 1: module Intersections(tVector2D ?p, ?dir, tLine ?lines[],
 2:                      tIntersect !intersection[]) {
 3:   intersection = {}
 4:   forall (l ∈ lines) {
 5:     determine alpha, beta with
 6:         p + alpha * dir = l.p1 + beta * (l.p2 − l.p1)
 7:     if(alpha > 0 ∧ beta ≥ 0 ∧ beta ≤ 1) {
 8:       add (alpha, line) to intersection
 9:       pause
10:     }
11:   }
12: }
13: macro z3dto2d(z, dist) = (z/dist + 1) * (ResolutionY/2)
14: macro fill(x, y, cond, col, dir) =
15:         while(cond ∧ y ≥ 0 ∧ y < ResolutionY) {
16:           image[x][y] = col
17:           next(y) = y + dir
18:           pause
19:         }
```

Figure 7.14: Pseudo code for additional functions and macros required by the facility renderer in Figure 7.15. The code is abstracted for readability. The semantics is given in the synchronous model of computation.

```
 1: module FacilityRenderer(event ?start, tMap ?map,
 2:              tVector3D cameraPos, real cameraDir,
 3:              tColor[][] !image, event !ready) {
 4:   while(true) {
 5:     immediate await(start);
 6:     for (x = 0..ResolutionX) {
 7:       tVector2D dir;
 8:       nat y_cl = ResolutionY // top of image
 9:       nat y_fl = 0 // bottom of image
10:       let dirx0 = x/(ResolutionX/2) − 1.0;
11:       let diry0 = 1.0;
12:       dir.x = cos(cameraDir) · dirx0 − sin(cameraDir) · diry0;
13:       dir.y = sin(cameraDir) · dirx0 + cos(cameraDir) · diry0;
14:       Intersections((cameraPos.x, cameraPos.y), dir, lines, intersections)
15:       sort intersections w.r.t. member alpha
16:       forall(i ∈ intersections) {
17:         l = i.line
18:         dist = i.alpha
19:         s_f = sideFacingCamera(l.s1, l.s2)
20:         s_b = sideAvertedCamera(l.s1, l.s2)
21:         y_cl,front = z3dto2d(s_f.sector.ceilingHeight − cameraPos.z, dist)
22:         y_fl,front = z3dto2d(s_f.sector.floorHeight − cameraPos.z, dist)
23:         y_cl,back = z3dto2d(s_b.sector.ceilingHeight − cameraPos.z, dist)
24:         y_fl,back = z3dto2d(s_b.sector.floorHeight − cameraPos.z, dist)
25:         fill(x, y_cl, y_cl ≥ y_cl,front ∧ y_fl ≤ y_cl, s_f.sector.ceilCol, −1)
26:         fill(x, y_fl, y_fl ≤ y_fl,front ∧ y_fl ≤ y_cl, s_f.sector.floorCol, 1)
27:         fill(x, y_cl, y_cl ≥ y_cl,back ∧ y_fl ≤ y_cl, s_f.upCol, −1)
28:         fill(x, y_fl, y_fl ≤ y_fl,back ∧ y_fl ≤ y_cl, s_f.lowCol, 1)
29:         if(s_f.midCol = transparent) {
30:           fill(x, y_fl, y_fl ≤ y_cl, s_f.midCol, 1)
31:         }
32:     }}
33:     emit ready;
34:     pause;
35: }}
```

Figure 7.15: Pseudo code for the facility renderer in Quartz syntax. The code is abstracted for readability. The semantics is given in the synchronous model of computation. The code for texturing has been removed to improve readability.

handling of one-sided lines. Instead of textures, we define colors of floors, ceilings (lines 4f) and colors of upper walls, i. e., the range between ceilings of adjacent sectors, lower walls, i. e., the range between floors of adjacent sectors, and middle walls, i. e., the range between the upper and lower wall (line 6).

Module Intersections in Figure 7.14 determines those lines in the given set *lines* that are hit by the given ray starting at p and running in direction *dir*. This is accomplished by solving the equation in line 6, which also returns parameters *alpha* and *beta*. The condition given by the if-statement in line 7 must hold for a hit of the given ray and the line. A set containing all hit lines including the corresponding *alpha* parameter are returned. The *alpha* value is required for sorting purposes and conversion of the z coordinates from 3d to 2d, which is done by macro z3dto2d in line 13. Macro fill is used to draw some part of a column with a given color *col*. The fill procedure starts at coordinate (x, y) to set the pixel at the given coordinate to color *col*. Depending on parameter *dir*, y is counted up or down until condition *cond* holds.

The actual computation is done by FacilityRender in Figure 7.15. As already mentioned, the computation of an image is based on a two-dimensional floor plan. The height parameters of sectors are used to scale the map into the third dimension. Similar to the height field renderer, the limitation to the camera's rotation allows to simplify some computations. It is sufficient to sent a single ray for each column of the target image (line 6,line 10-13). Each ray is intersected with the lines of the given scene (line 14). The hit lines are sorted according to their distance to the camera from front to back (line 15). We left the code of the sorting function for a better readability. Algorithms of sorting functions can be found in various literature, e. g., in [161, Chapter 2]. The final step is to render the walls, which is done by iterating through the list of hit lines (line 16ff). For each hit line the floor and the ceiling of the corresponding sector are drawn first (line 25f). Afterwards, the upper and lower walls are drawn (line 27f). Finally, in case that a middle wall is non-transparent, it is also drawn with the corresponding color. Only the middle part of a wall can be transparent, which is always an adjacent range in a column. As a consequence, it is sufficient to store the upper (y_{cl}) and lower (y_{fl}) position to "mark" the remaining visible area of a column. The loop in line 16 continues to iterate through hit lines with larger distance to the camera. The conditions given to the fill macro (line 25-28,30) ensure that further walls are only drawn in the remaining visible area, but not in the area covered by walls that are closer to the camera.

7.1.13 Ray Tracer

Ray tracing is a technique that targets the creation of photo-realistic pictures from a three-dimensional scene description. Photo-realistic means that a computer generated picture should not be distinguishable to a photograph of a real scene. Most techniques rely on the physical laws of rays. One – perhaps the best known – technique to create these pictures is to do so-called backward ray tracing. Tracking rays from the light source to the camera can be a computationally expensive task: It is not known, whether a ray hits the camera. Instead, tracking rays from the camera to a light source is more promising. It provides

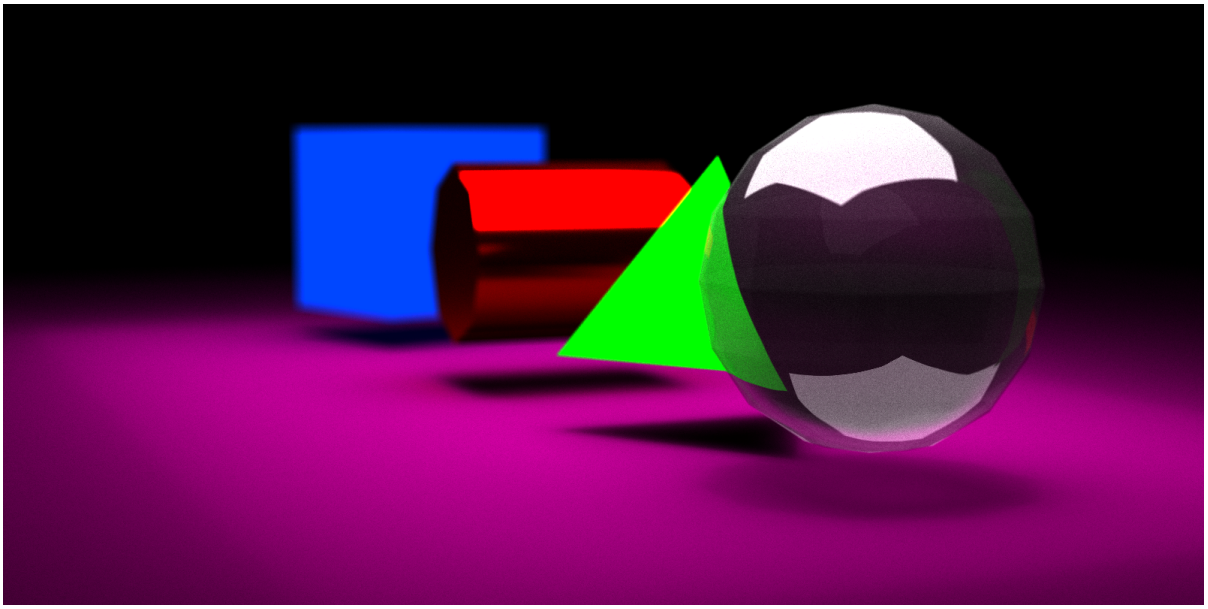


Figure 7.16: Picture rendered with the *Ray Tracer* benchmark. It demonstrates different materials: The sphere has a mostly transparent refracting material, which also reflects the lighting surface at the top of the scene. The tetrahedron has a diffuse green material and the cylinder is reflecting red light. Taking a closer look at the shadow of the blue box reveals that it is emitting blue light. Stochastic ray tracing enables lighting with surfaces, which results in natural soft shadows and depth-of-field which makes objects in the background of this scene softly blurred.

a deterministic model to compute the color for a specific pixel in an image. Objects in a scene are taking into account by applying the law of reflection in case of hit objects. First ray tracers used empirical lighting and illumination models to approximate realistic lighting. They were based on the recursive tracking of an originally single ray for each pixel. Whenever a ray hits an object, the ray may split into several sub-rays: reflection, diffuse illumination and transparency with optional refraction. The sub-rays are further tracked and may recursively split to more sub-rays. A weighted sum of the colors computed by the sub-rays results in the color of the pixel from which the original ray was initiated.

A more precise lighting model considers that an object hit by a ray is not only lit by a single ray, e. g., the reflected ray. It is rather lit by many rays coming from light sources – directly or indirectly, i. e., reflected by other objects. Thereby, the reflection of the ray depends on the material. It may result in a perfect reflection according to the law of reflection (the angle at which a ray is incident on the surface equals the angle at which it is reflected) or in a diffuse reflection, i. e., light from all directions will affect the color. Conversely, computing the color of a ray that hits an object requires to sum up the color of an infinite number of sub-rays. An approximation of that color value can be done using Monte Carlo integration [163, Chapter 11] which is also known as stochastic ray tracing. It determines the color by summarizing the color of a finite number of randomly reflected

rays. In addition, the colors of the reflected rays are weighted with a function depending on the material. This function is called bidirectional reflectance distribution function (BRDF) [163, Chapter 17] and describes the absorption of the material in dependence of the incident and reflected ray. The approach to use a Monte Carlo integration provides several advantages concerning programming and image quality.

Instead of recursively tracking a single ray, the Monte Carlo integration method uses iterative accumulation of weighted rays to compute the color of a single pixel. Thereby, the computation of a single ray starts at the origin of the camera, which is related to path tracing as described by Shirly [163, Chapter 16]. The iterative formulation of the ray tracing algorithm is a promising feature in the context of programming in synchronous languages. Implementation of recursive algorithms in hardware description languages is a non-trivial task [131], while the iterative ray tracing algorithm already has been used by Schmittler et. al. in their SaarCOR ray tracer processor [155].

Many recursive ray tracers use point light sources with the phong shading model to lighten a scene. In contrast, the Monte Carlo integration method allows to illuminate a scene with light-emitting faces, which results in more realistic soft shadows. Therefore, it only has to check whether a ray hits a light-emitting face, which can be easily done in the ordinary collision check. Hence, there is no need for special handling of light sources which is required in ray tracer using phong shading.

Stochastic ray tracing allows effects like depth of field [163, Chapter 14], [36] and motion blur which occur in real photographs. The focus distance is the technical term of the distance from the camera to objects that are in focus, i. e., objects that appear to be sharp in the photograph. It is limited to a range which is called the depth of field. This effect is created by using cameras with lenses. A lens captures light rays emitted from an object at different positions and diffracts it. Afterwards, these rays hit the sensor of the camera, which records the color. Due to physical limitations, only rays coming from the focus distance hit each other on the sensor, which is placed at the focal distance of the lens. Light rays coming from an object that is placed at a closer or farther distance than the focus distance hit different points on the sensor, which finally causes the blurred appearing of the object. The strength of the effect, i. e., the depth of field, is inversely proportional to the size of the area, where light can enter the lens. Reducing the size of that area to an infinite small amount would result in a pinhole camera, where everything is in focus. The physical process of recording an image with a camera takes time. As a result, if an object moves during the recording time, the emitted light of that object will hit different pixels on the camera sensor. Similar to depth of field, this will result in a blurred object.

Depth of field in stochastic ray tracing is achieved by back-tracing rays from different points of a plane which represents the surface of the camera's lens. In particular, when a path is going to be back-traced, a point on that plane is randomly chosen. The ray must run through the focus point, i. e., the direction is implicitly given. The focus point is determined by considering a pinhole camera and computing the point on the ray for the actual pixel at the focus distance. Objects at focus distance will be hit by all rays. In contrast, objects out of focus will only be hit by some rays, i. e., the color of the pixel will be influenced by different objects, which will finally cause the blurry appearance.

Motion blur is not considered in our implementation. It can be implemented by adding speed vector information to the basic geometry. When a ray has to be back-traced, the ray tracer additionally choses a random value for the point of time. The actual geometry is then determined by considering the basic geometry data and, in addition, the time value and the speed vector information.

7.2 Evaluation Platforms

system	synthesis approaches		
	task-based	message passing	OOO
Intel i5-750, 2.66GHz	✓	✓ (PThreads)	✓
Intel 2x Xeon X5450, 3.0GHz	✓		✓
self-build cluster: 4x Raspberry Pi 700 MHz, model B, revision 2, 512MB RAM		✓ (MPI)	

Figure 7.17: List of hardware systems that were used to evaluate our approaches. The right-hand side shows the approaches that have been evaluated on the respective platform.

The programs that were created using our synthesis tools have been evaluated on the architectures given in Figure 7.17. While the Intel i5 computer represents a system that can be found in the desktop domain, the dual Xeon system is a configuration that can be found in server and workstation equipment. The communication between cores on an i5 processor is completely done on the processor, which keeps the communication costs low. In contrast, the dual Xeon X5450 communicates using the front-side bus, while newer versions use the faster Intel quickpath interconnect (QPI). Hence, the communication between the Xeon X5450 processors is quite expensive compared to on-chip communication. To this end, the slow inter-processor communication results generally in lower speedups compared to the single-chip multi-core processor. Both systems are used to evaluate the approaches for SMP systems, i. e., the task-based and OOO execution. In addition, we use the i5 to evaluate the message passing approach in SMP systems using PThreads.

The Raspberry Pi cluster is used to assess the message passing approach and its optimizations. In these systems, communication is done over a network, which is much slower compared to the communication on a single chip. Hence, the execution of our generated applications in this systems will intensify the effect of communication optimizations.

7.3 Results

This section discusses the benchmark results. We will consider the different approaches and different concerns that are related to the respective approach. Table 7.1 shows the results of the translation of SGAs to DPNs. In particular, the first column gives the label of the

benchmarks, the second column the respective (full) name. Column "# SGAs" prints the number of actions contained in the synchronous system description. For all benchmarks, we targeted a DPN size of 8 nodes, i. e., the partitioning targeted the creation of 8 partitions. Since this is only a guide value, the actual size of the generated DPNs may differ and is given in the last column. Figures of the resulting DPNs have been added to the appendix of this thesis (see Appendix A). These figures also show the scheduling dependencies, which are the results of the structuring algorithm from Section 3.1.1. In most applications, we are able to gain parallelism from the structured graph. This is mandatory to enable concurrent execution using the task-based approaches from Chapter 3. In contrast, the DPN interpretation allows concurrent execution of nodes with direct dependencies (e. g., node 0 and node 1 in Figure A.11). Some graphs contain scheduling dependencies without an explicit data dependency between the respective nodes, e. g., node 15 and node 16 of AudioSynth (see Figure A.7). This is caused by scheduling strategies, which ensure correct execution of delayed assignments without carrier variables (see Section 2.1.3).

The computational effort for each node depends on the application, i. e., the overall number of SGAs, and the actual behavior, i. e., the dependencies between the given set of actions.

Table 7.1: Synthesis results

Benchmark	**Description**	**# SGAs**	**# nodes**
ParMatMult(T=8, 16×16)	(parallel) matrix multiplication	517	10
ParMatMult(T=8, 32×32)	(parallel) matrix multiplication	2053	10
SeqMatMult(T=8, 16×16)	sequential matrix multiplication	84	11
SeqMatMult(T=8, 32×32)	sequential matrix multiplication	84	11
LUDecomp(T=8, 16×16)	LU decomposition	3916	7
LUDecomp(T=8, 32×32)	LU decomposition	21132	7
Sqrt(T=8, B=1024)	square root	126	9
Delay(T=8, 200msec)	signal delay	22	8
DFT(T=8)	discrete fourier transform	1312	11
Pitchshift(T=8, 200msec)	pitch shift	67	9
AudioSynth(T=8, C=16)	modular audio synthesizer	9633	13
3D Transform(T=8)	3D geometry transformation	371	10
ImageScale(T=8)	image scaler	10273	5
Mandelbrot(T=8, W=256)	Mandelbrot Set	20785	9
FacilityR(T=8, H=400, W=2)	facility renderer	3195	12
Ray Tracer(T=8, H=4, W=4)	ray tracer	10599	13
Landscape(T=8, H=200, W=4)	height field renderer	950	9
Landscape(T=8, H=800, W=4)	height field renderer	950	9

7.3.1 Task-Based (OpenMP, SmpSS)

For the benchmarks of the task-based approach, we partitioned all benchmarks into 8 tasks. The results of the i5-750 and 2xXeon system are printed in Table 7.2 and Table 7.3, respectively. While some of our benchmarks are speeded up by the task-based execution, others are significantly slowed down, which was to be expected due to the low computational effort in these benchmarks. We start with considering the benchmark results that were made on the i5-750 system.

The parallel matrix multiplication (*ParMatMult*) was benchmarked in two different sizes: 16×16 and 32×32. While the former was slowed down by the task-based parallel execution, the latter was accelerated on the i5-750 system by a factor of 1.91 using OpenMP and 1.81 using SmpSS. Obviously, the break-point for acceleration is between the given benchmark sizes. A similar behavior can be observed for the 2x Xeon system. The smaller

Table 7.2: Runtimes for task-based execution on i5-750. Times are given in seconds, speedups are given in parenthesis. (2): Benchmark was not compiled.

Benchmark	**Seq.**	**OpenMP**	**SmpSS**
ParMatMult(T=8, 16×16)	11.06	12.40(0.89)	13.64(0.81)
ParMatMult(T=8, 32×32)	108.44	56.76(1.91)	59.84(1.81)
SeqMatMult(T=8, 16×16)	0.02	1.99(0.01)	1.06(0.01)
SeqMatMult(T=8, 32×32)	0.01	1.01(0)	0.53(0.01)
LUDecomp(T=8, 16×16)	14.76	20.47(0.72)	12.40(1.19)
LUDecomp(T=8, 32×32)	52.32	50.58(1.03)	39.76(1.31)
Sqrt(T=8, B=1024)	21.67	30.04(0.72)	(2)
Delay(T=8, 200msec)	0.44	6.21(0.07)	6.66(0.06)
DFT(T=8)	15.57	18.44(0.84)	18.75(0.83)
Pitchshift(T=8, 200msec)	0.59	23.41(0.02)	12.93(0.04)
AudioSynth(T=8, C=16)	9.30	11.07(0.84)	7.16(1.29)
3D Transform(T=8)	0.57	7.66(0.07)	12.91(0.04)
ImageScale(T=8)	10.71	13.37(0.80)	7.69(1.39)
Mandelbrot(T=8, W=256)	80.20	71.77(1.11)	51.66(1.55)
Landscape(T=8, H=200, W=4)	0.58	0.74(0.78)	0.90(0.64)
Landscape(T=8, H=800, W=4)	2.92	4.66(0.62)	6.06(0.48)
FacilityR(T=8, H=400, W=2)	233.43	488.54(0.47)	(2)
Ray Tracer(T=8, H=4, W=4)	13.83	10.41(1.32)	8.45(1.63)

variant is not accelerated, while ParMatMult 32×32 gains a speedup by a factor 3.31 using OpenMP and 1.53 using SmpSS. Again, the break point for the acceleration of parallel execution is reached between the given benchmark sizes. The OpenMP version of the parallel matrix multiplication is generally faster than the SmpSS version, which is explained by the scheduling of the nodes. OpenMP uses a static schedule, i. e., sequences of nodes are simply written as a sequence of code, e. g., node 0 and node 1 in Figure A.1 are mapped to a single sequence of instructions. In contrast, SmpSS uses dynamic scheduling and applies its scheduling technique to all nodes. Hence, nodes with sequential dependencies require additional scheduling overhead.

SeqMatMult, *Delay*, *Pitchshift* are considered as theoretical minimal examples to analyze optimizations in Sections 7.3.2.1 and 7.3.2.2. Figures A.2, A.5 and A.8 show that parallelism is available for these benchmarks. However, Figure 7.1 shows that these examples contain only a few actions, which results in a high scheduling overhead in the targeted

Table 7.3: Runtimes for task-based execution on 2x Xeon X5450. Times are given in seconds, speedups are given in parenthesis. (2): Benchmark was not compiled.

Benchmark	**Seq.**	**OpenMP**	**SmpSS**
ParMatMult(T=8, 16×16)	14.39	14.06(1.02)	30.07(0.47)
ParMatMult(T=8, 32×32)	110.63	33.37(3.31)	72.20(1.53)
SeqMatMult(T=8, 16×16)	0.01	3.37(0)	2.68(0)
SeqMatMult(T=8, 32×32)	0.01	1.71(0)	1.34(0)
LUDecomp(T=8, 16×16)	17.71	24.02(0.73)	23.32(0.75)
LUDecomp(T=8, 32×32)	59.59	60.24(0.98)	43.01(1.38)
Sqrt(T=8, B=1024)	18.17	38.40(0.47)	(2)
Delay(T=8, 200msec)	0.56	10.30(0.05)	15.56(0.03)
DFT(T=8)	18.61	20.04(0.92)	28.15(0.66)
Pitchshift(T=8, 200msec)	0.69	41.58(0.01)	30.08(0.02)
AudioSynth(T=8, C=16)	10.41	18.61(0.55)	21.95(0.47)
3D Transform(T=8)	0.64	12.27(0.05)	31.14(0.02)
ImageScale(T=8)	10.53	12.62(0.83)	8.67(1.21)
Mandelbrot(T=8, W=256)	69.92	57.12(1.22)	90.81(0.76)
Landscape(T=8, H=200, W=4)	0.77	1.04(0.74)	1.77(0.43)
Landscape(T=8, H=800, W=4)	4.07	7.09(0.57)	13.03(0.31)
FacilityR(T=8, H=400, W=2)	264.92	634.50(0.41)	(2)
Ray Tracer(T=8, H=4, W=4)	11.72	8.44(1.38)	16.93(0.69)

APIs. As a result, these benchmarks are significantly slowed down by the task-based approach.

The *LU decomposition* shows a different behavior: For both hardware systems, the parallel execution of the smaller version (16×16) using OpenMP leads to a slow down and the execution time of the larger version (32×32) stays close to the sequential execution. In contrast, the generated SmpSS code accelerates the execution of both variants (16×16 and 32×32) on the i5 and the larger variant on the Xeon system. This is due to the structured graph required by OpenMP. In particular, the fork-join execution model restricts available parallelism due to the additional control dependencies (see Chapter 3). In contrast, SmpSS gets full advantage of parallelism due to the data-flow driven execution.

The *Sqrt* and *FacilityR* benchmarks require a C++ compiler because these applications make use of classes (in the sense of object oriented C++ classes). Since SmpSS is restricted to compilation of C code, these applications were only benchmarked for the OpenMP API.

Sqrt provides only few parallelism and its parallel execution results in a slow down due to the overhead. Its use is found in our streaming approaches.

FacilityR and *Landscape* provide concurrency, which is not sufficient to create enough load per partition. This results in programs, which are slower than the sequential execution for both target architectures.

The *DFT* provides parallelism, which is however not enough to amortise the synchronization overhead. This can be observed for both APIs.

AudioSynth, *ImageScale* and *Mandelbrot* gain a speedup using SmpSS for parallel execution. The execution according to the data dependencies seems to allow better load-balancing. The scheduling dependencies of the structured graph for OpenMP code generation correspond to the data dependencies of the DFG used for the SmpSS code generation. We could not spot any difference between the explicit scheduling dependencies and the data dependencies in the generated DPNs (see Figure A.7, A.10 and A.11). We believe that the difference in the execution times is caused by the particular implementation of these APIs.

3D Transform experienced an extreme slow down by the task-based execution. Similar to SeqMatMult, Delay and Pitchshift, this is due to a low computational effort of the benchmark. After the computation of a transformation matrix, the application applies only a 4×4 matrix multiplication to a vector per iteration. This benchmark is intended for stream-based processing and becomes more interesting in our OOO execution.

Ray Tracer provides parallelism and enough computational effort per partition to allow a speed-up using both APIs on the i5-750 and using OpenMP on the Xeon system. The speedup using OpenMP exceeds 1.3 on both target architectures. The speedup using SmpSS reaches a factor of 1.23 on the i5, while the execution on the Xeon system is slowed down. We believe that the scheduling of nodes with sequential dependencies requires additional overhead in the execution using SmpSS. A similar effect has already been observed for *ParMatMult*.

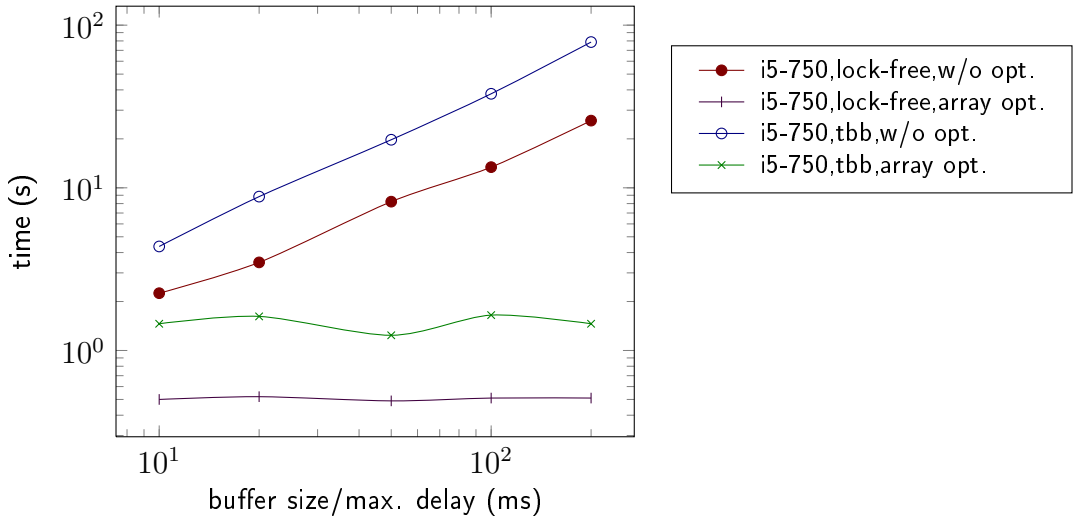


Figure 7.18: Times of the delay benchmark. It represents the idealized benchmark to compare the performance of the array communication reduction approach to the unoptimized message passing approach. All plots show the execution times on the i5-750 system. In particular, Plot —●— and —○— show the benchmarks without the optimization and Plot —+— and —×— using the presented optimization algorithm.

7.3.2 Message Passing

Table 7.2 shows the results of the message passing approach with and without the presented communication optimizations. In general, the execution times of the message passing turned out to be higher than the execution times of the task-based execution. The insertion of buffers to decouple the nodes in a given DPN allows to map each node to a thread, which results in more parallelism on the one side and higher communication costs on the other side. Despite of the higher degree of parallelism, the additional effort to communicate values of variables causes a slow down. To keep the execution times moderate, we reduced the computational load by a factor of 10, e. g., by processing a smaller number of matrices in the matrix multiplication or rendering a smaller number of pictures in the graphical benchmarks.

Beside the implementation of our FIFO queue using the lock-free implementation from Section 6.1, we also tested some benchmarks using the Intel TBB concurrent queue implementation. The results are printed at the bottom of Table 7.2. They clearly confirm the usage of lock-free queues to implement DPN buffers. The lock-free variant provides a significant speedup of the light-weight communication in our DPNs: The speedup factors in non-optimized message passing execution for the given applications are 4.84 for *ParMatMult* (16×16), 5.56 for *ParMatMult* (32×32), 3.97 for *AudioSynth* and over 8 for both *Landscape* variants. The same effect has been observed for the other benchmarks.

7.3.2.1 Array Optimization

The effect of communication costs of arrays can be seen for instance in *Delay*, *AudioSynth*, *Landscape* and *FacilityR* on the i5-750 system.

SeqMatMult, *Delay*, *Pitchshift*, *AudioSynth* and *Landscape* include the processing of large data sets with a low amount of data parallelism at the same time. This allows to reduce the amount of communication using our array communication optimization.

We use *Delay* as a reference benchmark to analyze the approach. We tested different buffer sizes in this benchmark and measured the execution times in dependence of the buffer size. The results for both queue implementations are plotted in Figure 7.18. One can clearly see that the execution times for the unoptimized versions grows with the buffer size. Despite of the constant amount of computational effort, the communication effort increases proportional to the buffer size in the unoptimized versions. In contrast, the execution times for the optimized versions are constant. The figure also confirms the usage of the lock-free queue implementation.

The same effect can be observed for *Landscape*. The execution time increases drastically with the size of the rendered picture. The optimization achieves a better speedup in the execution of the large version (H=800) compared to the execution of the small version of *Landscape* (H=200). The execution time of the optimized version increases with the size of *Landscape*, which is to be expected: The computational effort for larger pictures increases and requires more computations.

FacilityR was expected to provide an improvement, too. However, the application of the array optimization to this benchmark failed due to the memory requirements of the optimization process.

Remaining applications are based on data parallelism. Roughly spoken, they always need a complete communication of their data structures. Hence, they cannot be accelerated by the array communication optimization approach. As a result, the execution time is the same as for the unmodified version.

The results made on the Raspberry Pi cluster generally confirm our results (see Table 7.5). *SeqMatMult* and *Landscape* are significantly accelerated by a factor of 1.9 and 14.75. We used a different partitioning size for the Raspberry Pi cluster to better fit the architecture. The partitioning to larger tasks merged actions writing to the same array for *Delay* and *Pitchshift*. Hence, these benchmarks were not affected by the optimization. *AudioSynth* did not finished in time on the cluster and is therefore not listed.

7.3.2.2 General Optimization

The general communication optimization presented in Section 4.3 is applicable to all kinds of value communication between two nodes. Because the array optimization always leads to the same or a better execution time for each benchmark, we applied the general communication reduction after the array communication reduction. The execution times of the general communication optimization are printed in Table 7.4 for the i5-750 and Table 7.5 for the Raspberry Pi cluster. The speedups refer to the non-optimized version, i. e., the

times in column "array opt.". In addition, we measured the amount of communication reduction of the benchmarks on the i5-750 (see Table 7.6).

The first optimization level (column "gco-1") applies the reduction as described in Section 4.3.1.2, i. e., only if all variables occurring in $\gamma_{\mathrm{wr}}(x)$ and $\gamma_{\mathrm{rd}}(x, n)$ (without existential quantification) are known. The second level (column "gco-2") applies existential quantification to the read and write guards as described in Section 4.3.1.3. The last level (column "gco-3") uses the refinement from Section 4.3.2 which adds transports to get a more precise guard evaluation, which in turn should reduce overall communication.

Table 7.6 clearly shows that the amount of data that has to be sent between nodes in a DPN can be reduced with our approach. The reduction strongly depends on the application and starts at about 1% and reaches up to 74.15%. The effect on the execution time depends on the respective application.

ParMatMult is implemented as a streaming application which computes the result of a complete matrix multiplication per iteration of the program. In each step, the program reads two matrices, calculates their product and sends it to the environment. As a consequence, each action in this program will be involved into the calculation in each step, making communication reduction nearly impossible. The sequential matrix multiplication (*SeqMatMult*) calculates only one element of the result in each step. However, this involves less actions in the program which are still executed in each step. Hence, the scope of optimization in this program is very limited, too.

LUDecomp does a wavefront computation. The amount of computations and the number of modified variables depend on the iteration. Hence, the amount of communication varies and allows a significant reduction by up to 63.47%.

Sqrt, *Delay*, *DFT*, *Pitchshift*, *AudioSynth*, *Landscape* and *Ray Tracer* provide either a high ratio of computation, or they already have been optimized by the array optimization. This limits the general communication optimization for data values. Due to their complexity, they require more control logic compared to the matrix multiplication. The control logic provides more potential for communication reduction. Depending on the ratio of communicated control values to data values, the overall communication reduction ranges between 10% and 26%.

The most significant speedup can be observed for *3D Transform*. This benchmark precomputes a transformation matrix and applies a matrix-vector multiplication to a stream of vectors. Only if the parameter for the transformation change, the transformation matrix has to be recomputed and broadcast to other nodes in the DPN. These parameters are rarely changed which leads to an effective reduction of communication (at least 67%).

Finally, *Mandelbrot* behaves as *ParMatMult*. The application basically changes in each iteration all values in the data structure that describes the state of the current Mandelbrot set computation. In addition, the amount of control logic is low. Hence, the scope of optimizations in this application is limited and reaches a maximum value of 5.96%.

The execution times (see Table 7.4) of our benchmarks depend on (1) the saved amount of data transfers, which may be (2) reduced by the additional effort of evaluating guards that trigger data transfer. A third determining factor for the benchmark times are the

remaining dependencies in a benchmark. Finally, the implementation of the communication influences the communication reduction, too.

The benchmark runtimes obviously do not scale directly to the amount of saved data transfers. This can be explained by Amdahl's law. In our case, this means that the communication affects only a part of the overall execution time, and as a result, the speedup will be slower than the percental communication reduction. Moreover, the conditional communication requires additional effort to evaluate the guards, which may reduce the effect of communication reduction. In some applications, this leads to a slow down in the execution. The light-weight communication using the lock-free queue seems to be faster than the evaluation of guards in *SeqMatMult* (32×32), *Sqrt*, *Delay* and *AudioSynth*. The benchmarks that used the Intel TBB concurrent queue and our MPI benchmarks did not show this effect at all.

In specific cases, our approach can result in a DPN that provides more parallelism: Dependencies in a synchronous DPN may restrict the schedule of nodes to a sequential execution. While nodes in an unoptimized code eventually have to wait for availability of data, the communication reduction of our optimization would temporarily allow to neglect (some) dependencies, thereby allowing to fire nodes earlier. Hence, we do not only reduce communication but also gain more parallelism by allowing early evaluation. This can be observed for *3D Transform* which reaches a speedup of a factor greater than 6 on the i5-750.

The MPI benchmark results printed in Table 7.5 confirm the results on the SMP systems. Similarly, the MPI version of the *3D Transform* benchmark performs best with a speedup of a factor up to 2.81.

Table 7.4: Runtimes for message-passing execution using PThreads on i5-750. Times are given in seconds, the numbers in parentheses denote the speedup of "array opt." to "no opt." execution and the speedup of "gco" to "array opt." execution. (1): Benchmark was not synthesized.

Benchmark	**Optimization Level**				
	no opt.	**array opt.** (speedup)	**gco-1** (speedup to array opt.)	**gco-2** (speedup to array opt.)	**gco-3** (speedup to array opt.)
	PThreads Using Lock-Free Queue				
ParMatMult(T=8, 16 × 16)	13.38	14.31(0.93)	11.14(1.28)	11.36(1.25)	11.18(1.27)
ParMatMult(T=8, 32 × 32)	50.47	50.64(0.99)	41.90(1.20)	41.08(1.23)	41.77(1.21)
SeqMatMult(T=8, 16 × 16)	0.34	0.18(1.88)	0.17(1.05)	0.14(1.28)	0.14(1.28)
SeqMatMult(T=8, 32 × 32)	7.89	3.88(2.03)	3.83(1.01)	4.34(0.89)	4.31(0.90)
LUDecomp(T=8, 16 × 16)	5.43	5.97(0.90)	4.10(1.45)	4.09(1.45)	3.50(1.70)
LUDecomp(T=8, 32 × 32)	13.21	13.38(0.98)	10.26(1.30)	9.93(1.34)	9.34(1.43)
Sqrt(T=8, B=1024)	4.44	3.98(1.11)	5.53(0.71)	6.34(0.62)	5.25(0.75)
Delay(T=8, 200msec)	25.89	0.51(50.76)	0.59(0.86)	0.58(0.87)	0.61(0.83)
DFT(T=8)	3.49	3.39(1.02)	2.93(1.15)	2.91(1.16)	2.64(1.28)
Pitchshift(T=8, 200msec)	5.79	0.99(5.84)	0.63(1.57)	0.61(1.62)	0.67(1.47)
AudioSynth(T=8, C=16)	120.20	25.93(4.63)	32.05(0.80)	34.38(0.75)	34.81(0.74)
3D Transform(T=8)	1.25	1.27(0.98)	0.21(6.04)	0.21(6.04)	0.21(6.04)
ImageScale(T=8)	1.16	1.17(0.99)	(1)	(1)	(1)
Mandelbrot(T=8, W=256)	28.07	27.97(1.00)	28.01(0.99)	29.29(0.95)	27.99(0.99)
Landscape(T=8, H=200, W=4)	8.14	1.83(4.44)	1.69(1.08)	1.64(1.11)	1.73(1.05)
Landscape(T=8, H=800, W=4)	210.82	11.88(17.74)	10.18(1.16)	10.24(1.16)	10.61(1.11)
FacilityR(T=8, H=400, W=2)	908.89	(1)	(1)	(1)	(1)
Ray Tracer(T=8, H=4, W=4)	5.99	6.02(0.99)	5.91(1.01)	6.03(0.99)	5.76(1.04)
	PThreads Using Intel TBB Queue				
ParMatMult(T=8, 16 × 16)	64.78	64.51(1.00)	30.63(2.10)	30.81(2.09)	32.10(2.00)
ParMatMult(T=8, 32 × 32)	280.77	278.38(1.00)	123.07(2.26)	120.75(2.30)	128.92(2.15)
AudioSynth(T=8, C=16)	477.45	187.35(2.54)	178.15(1.05)	187.59(0.99)	182.01(1.02)
Landscape(T=8, H=200, W=4)	67.15	11.84(5.67)	9.55(1.23)	9.52(1.24)	9.30(1.27)
Landscape(T=8, H=800, W=4)	1713.25	90.69(18.89)	70.44(1.28)	68.27(1.32)	69.39(1.30)

Table 7.5: Runtimes for message-passing execution using MPI on Raspberry Pi Cluster. If not explicitly given, time is measured in seconds. The numbers in parentheses denote the speedup of "array opt." to "no opt." execution and the speedup of "gco" to "array opt." execution. (1): Benchmark was not synthesized. (2): Benchmark was not compiled.

	Optimization Level				
Benchmark	**no opt.**	**array opt.**	**gco-1**	**gco-2**	**gco-3**
ParMatMult(T=4, 16 × 16)	9.31	9.16(1.01)	8.90(1.02)	8.97(1.02)	8.60(1.06)
ParMatMult(T=4, 32 × 32)	(2)	(2)	(2)	(2)	(2)
SeqMatMult(T=4, 16 × 16)	1.18h	0.62h(1.90)	0.61h(1.01)	0.57h(1.09)	0.58h(1.06)
SeqMatMult(T=4, 32 × 32)	(3)	17h	16.9h(1.01)	16.7h(1.02)	17.3h(0.96)
LUDecomp(T=4, 16 × 16)	304.19	302.76(1.00)	161.78(1.87)	161.45(1.87)	133.00(2.27)
LUDecomp(T=4, 32 × 32)	(2)	(2)	(2)	(2)	(2)
Sqrt(T=4, B=1024)	411.34	412.42(0.99)	271.50(1.51)	274.32(1.50)	271.00(1.52)
Delay(T=4, 200msec)	41.43	41.42(1.00)	35.13(1.17)	34.66(1.19)	34.70(1.19)
DFT(T=4)	799.07	799.12(0.99)	703.01(1.13)	699.48(1.14)	687.01(1.16)
Pitchshift(T=4, 200msec)	62.63	62.88(0.99)	55.72(1.12)	55.83(1.12)	55.78(1.12)
3D Transform(T=4)	8.86	8.77(1.01)	6.53(1.34)	6.53(1.34)	3.11(2.81)
ImageScale(T=4)	(2)	(2)	(2)	(2)	(2)
Mandelbrot(T=4, W=256)	(2)	(2)	(2)	(2)	(2)
Landscape(T=4, H=200, W=4)	74.8h	5.1h(14.75)	4.65h(1.09)	4.5h(1.12)	4.6h(1.10)
FacilityR(T=4, H=400, W=2)	(2)	(1)	(1)	(1)	(1)
Ray Tracer(T=4, H=4, W=4)	(2)	(2)	(2)	(2)	(2)

Table 7.6: Communication reduction in message-passing execution for T=8. Columns "gco-1", "gco-2" and "gco-3" denote the amount of communication reduction that was achieved using the general communication reduction approach. (1): Benchmark was not synthesized.

	Optimization Level		
Benchmark	**gco-1**	**gco-2**	**gco-3**
ParMatMult(T=8, 16 × 16)	0.33%	0.33%	0.29%
ParMatMult(T=8, 32 × 32)	0.09%	0.09%	0.08%
SeqMatMult(T=8, 16 × 16)	1.68%	2.02%	2.02%
SeqMatMult(T=8, 32 × 32)	0.45%	0.54%	0.54%
LUDecomp(T=8, 16 × 16)	43.10%	43.10%	61.30%
LUDecomp(T=8, 32 × 32)	38.64%	38.64%	63.47%
Sqrt(T=8, B=1024)	11.52%	11.52%	11.52%
Delay(T=8, 200msec)	26.09%	26.09%	26.09%
DFT(T=8)	1.12%	1.12%	16.34%
Pitchshift(T=8, 200msec)	11.33%	11.33%	11.33%
AudioSynth(T=8, C=16)	10.29%	11.21%	10.93%
3D Transform(T=8)	67.27%	67.27%	74.15%
ImageScale(T=8)	(1)	(1)	(1)
Mandelbrot(T=8, W=256)	0.05%	1.40%	5.96%
Landscape(T=8, H=200, W=4)	14.95%	16.22%	16.22%
Landscape(T=8, H=800, W=4)	12.56%	13.17%	13.17%
Ray Tracer(T=8, H=4, W=4)	18.60%	19.06%	19.02%

7.3.3 OOO Execution

Table 7.7 and Table 7.8 show the speedups of our benchmarks on the i5-750 and dual Xeon, respectively. In contrast to [21], we used a different partitioning size for the benchmarks in this thesis. This allows us to compare the OOO approach to the preceding approaches. In general, the different partitioning size affects only a few benchmarks, i. e., the OOO execution leads for most benchmarks to the same behavior.

We start with considering the results that were made on the i5-750 system. In most cases, the OOO approach can speed up the execution compared to in-order execution. A strange effect can be seen in *ParMatMult* (16×16), *3D Transform* (WS=16), *ImageScale* and *Landscape*. All achieve a speedup that is higher than the number of cores. Actually, we can only speculate about this: first, the data alignment may be better in the OOO execution, thereby enabling a better cache behavior in the processor. Second, although the

Table 7.7: Runtimes for OOO execution with and without speculation on i5-750. Times are given in seconds, speedups are given in parenthesis. (2): Benchmark was not compiled.

		OOO		OOO-Spec	
Benchmark	**IO**	**WS=8**	**WS=16**	**WS=8**	**WS=16**
ParMatMult(T=8, 16×16)	3.57	0.58(6.15)	0.56(6.37)	0.61(0.95)	0.61(0.91)
ParMatMult(T=8, 32×32)	12.50	3.26(3.83)	3.26(3.83)	3.29(0.99)	3.24(1.00)
SeqMatMult(T=8, 16×16)	0.17	0.13(1.30)	0.13(1.30)	0.07(1.85)	0.07(1.85)
SeqMatMult(T=8, 32×32)	2.22	1.25(1.77)	1.23(1.80)	0.60(2.08)	1.01(1.21)
LUDecomp(T=8, 16×16)	11.83	11.28(1.04)	14.24(0.83)	11.16(1.01)	6.79(2.09)
LUDecomp(T=8, 32×32)	21.59	20.78(1.03)	25.13(0.85)	22.94(0.90)	23.29(1.07)
Sqrt(T=8, B=1024)	7.87	8.52(0.92)	8.92(0.88)	3.02(2.82)	3.11(2.86)
Delay(T=8, 200msec)	5.01	3.59(1.39)	3.63(1.38)	1.93(1.86)	3.42(1.06)
DFT(T=8)	3.90	3.38(1.15)	3.43(1.13)	1.91(1.76)	1.92(1.78)
Pitchshift(T=8, 200msec)	2.42	2.76(0.87)	2.81(0.86)	1.15(2.40)	0.98(2.86)
AudioSynth(T=8, C=16)	34.51	41.58(0.82)	41.67(0.82)	36.08(1.15)	32.90(1.26)
3D Transform(T=8)	2.20	1.10(2.00)	0.52(4.23)	0.34(3.23)	0.31(1.67)
ImageScale(T=8)	4.13	0.30(13.76)	0.28(14.75)	(2)	(2)
Mandelbrot(T=8, W=256)	24.75	22.76(1.08)	22.77(1.08)	22.06(1.03)	20.66(1.10)
Landscape(T=8, H=200, W=4)	4.89	0.84(5.82)	0.83(5.89)	0.87(0.96)	0.87(0.95)
Landscape(T=8, H=800, W=4)	34.16	6.40(5.33)	6.26(5.45)	8.08(0.79)	8.76(0.71)
FacilityR(T=8, H=400, W=2)	144.54	97.57(1.48)	97.65(1.48)	(2)	(2)
Ray Tracer(T=8, H=4, W=4)	8.00	7.16(1.11)	7.03(1.13)	8.07(0.88)	10.70(0.65)

in-order DPN execution is parallel, it is likely that the communication overhead is larger than the communication overhead of the OOO execution. Despite of the great result, we consider this as a success by accident because the results are not reflected by the dual Xeon system.

Moreover, we used different window sizes for the benchmarks to analyze its effect on the execution time. On the one hand, a larger window size contains more tasks, and therefore, it promises more independent tasks, i. e., more parallelism. On the other hand, the additional memory requirements could result in a reduced cache performance of the processor. Only a few benchmarks have been affected by the given window sizes. While *3D Transform* was able to gain a significant benefit of a larger window size, *LUDecomp* is slowed down. In some additional preliminary benchmarks with a window size of 32

Table 7.8: Runtimes for OOO execution with and without speculation on 2x Xeon X5450: speedups are given in parenthesis. [(2)]: Benchmark was not compiled.

		OOO		**OOO-Spec**	
Benchmark	**IO**	**WS=8**	**WS=16**	**WS=8**	**WS=16**
ParMatMult(T=8, 16 × 16)	5.15	2.77(1.85)	2.87(1.79)	2.45(1.13)	2.22(1.29)
ParMatMult(T=8, 32 × 32)	11.15	5.23(2.13)	5.01(2.22)	4.91(1.06)	5.29(0.94)
SeqMatMult(T=8, 16 × 16)	0.20	0.15(1.33)	0.16(1.25)	0.43(0.34)	0.43(0.37)
SeqMatMult(T=8, 32 × 32)	1.88	1.61(1.16)	1.55(1.21)	7.63(0.21)	8.96(0.17)
LUDecomp(T=8, 16 × 16)	5.50	6.08(0.90)	8.42(0.65)	8.25(0.73)	14.11(0.59)
LUDecomp(T=8, 32 × 32)	8.97	10.03(0.89)	11.64(0.77)	15.02(0.66)	22.99(0.50)
Sqrt(T=8, B=1024)	4.00	4.15(0.96)	4.05(0.98)	4.82(0.86)	5.53(0.73)
Delay(T=8, 200msec)	5.09	5.09(1.00)	5.07(1.00)	13.57(0.37)	20.09(0.25)
DFT(T=8)	4.02	3.57(1.12)	3.58(1.12)	4.90(0.72)	5.46(0.65)
Pitchshift(T=8, 200msec)	4.65	3.19(1.45)	3.13(1.48)	4.99(0.63)	5.78(0.54)
AudioSynth(T=8, C=16)	38.70	35.75(1.08)	36.25(1.06)	75.21(0.47)	136.16(0.26)
3D Transform(T=8)	2.82	2.05(1.37)	1.49(1.89)	3.00(0.68)	1.77(0.84)
ImageScale(T=8)	1.57	0.39(4.02)	0.31(5.06)	0.52(0.75)	0.37(0.83)
Mandelbrot(T=8, W=256)	11.40	10.94(1.04)	11.18(1.01)	21.24(0.51)	24.83(0.45)
Landscape(T=8, H=200, W=4)	3.54	1.80(1.96)	1.74(2.03)	3.11(0.57)	3.18(0.54)
Landscape(T=8, H=800, W=4)	33.96	18.50(1.83)	18.32(1.85)	26.28(0.70)	56.76(0.32)
FacilityR(T=8, H=400, W=2)	110.96	80.74(1.37)	73.92(1.50)	[(2)]	[(2)]
Ray Tracer(T=8, H=4, W=4)	3.24	3.25(0.99)	3.27(0.99)	17.94(0.18)	20.12(0.16)

Table 7.9: Speculation hit ratio for OOO execution with speculation on i5-750.

Benchmark	#Task Executed	#Tasks Speculated ∑	correct	too late	wrong
ParMatMult(T=8, 16 × 16, WS=8)	900037	8099	69.38%	30.29%	0.33%
ParMatMult(T=8, 16 × 16, WS=16)	900069	3355	34.72%	24.23%	41.04%
ParMatMult(T=8, 32 × 32, WS=8)	900048	864	64.70%	34.49%	0.81%
ParMatMult(T=8, 32 × 32, WS=16)	900064	49	69.39%	14.29%	14.29%
SeqMatMult(T=8, 16 × 16, WS=8)	46269	16216	32.90%	9.13%	56.48%
SeqMatMult(T=8, 16 × 16, WS=16)	46295	16753	16.40%	3.13%	79.13%
SeqMatMult(T=8, 32 × 32, WS=8)	348518	32022	27.96%	40.23%	31.34%
SeqMatMult(T=8, 32 × 32, WS=16)	348526	58720	27.10%	10.05%	62.17%
LUDecomp(T=8, 16 × 16, WS=8)	687017	51942	15.37%	21.52%	63.10%
LUDecomp(T=8, 16 × 16, WS=16)	937379	347774	35.74%	5.41%	58.85%
LUDecomp(T=8, 32 × 32, WS=8)	259599	37600	10.45%	10.86%	78.68%
LUDecomp(T=8, 32 × 32, WS=16)	314160	85459	19.44%	4.52%	76.04%
Sqrt(T=8, B=1024, WS=8)	358808	107720	38.68%	9.01%	18.85%
Sqrt(T=8, B=1024, WS=16)	364227	117868	26.20%	9.98%	52.97%
Delay(T=8, 200msec, WS=8)	350037	22891	0.00%	99.98%	0.02%
Delay(T=8, 200msec, WS=16)	350038	50489	2.71%	28.25%	69.04%
DFT(T=8, WS=8)	550023	268659	73.36%	4.95%	21.69%
DFT(T=8, WS=16)	550035	278618	63.74%	4.29%	31.97%
Pitchshift(T=8, 200msec, WS=8)	1000019	92945	34.69%	34.12%	31.18%
Pitchshift(T=8, 200msec, WS=16)	1000088	76391	28.01%	36.61%	35.38%
AudioSynth(T=8, C=16, WS=8)	2160932	78768	0.16%	39.22%	60.62%
AudioSynth(T=8, C=16, WS=16)	2160940	189073	0.10%	22.86%	77.04%
3D Transform(T=8, WS=8)	1000019	43811	45.62%	34.16%	20.21%
3D Transform(T=8, WS=16)	1000055	16055	44.46%	28.81%	26.73%
Mandelbrot(T=8, W=256, WS=8)	467260	160290	52.83%	8.76%	38.41%
Mandelbrot(T=8, W=256, WS=16)	467335	191333	37.15%	6.81%	56.04%
Landscape(T=8, H=200, W=4, WS=8)	516835	46270	35.91%	33.18%	30.91%
Landscape(T=8, H=200, W=4, WS=16)	516923	46407	34.27%	12.23%	53.50%
Landscape(T=8, H=800, W=4, WS=8)	4166635	55354	0.25%	99.71%	0.03%
Landscape(T=8, H=800, W=4, WS=16)	4166723	74848	9.00%	74.12%	16.87%
Ray Tracer(T=8, H=4, W=4, WS=8)	162545	94988	6.82%	3.64%	85.01%
Ray Tracer(T=8, H=4, W=4, WS=16)	162553	126279	2.00%	0.54%	93.23%

(WS=32), we observed that for most benchmarks the performance is decreased, which confirms the negative effect of a large window size on the cache behavior.

The OOO execution on the system with Xeon processors leads also to a speedup for most of the applications. Other benchmarks, in particular *Delay*, *Mandelbrot* and *Ray Tracer*, are nearly unaffected by the OOO execution. *LUDecomp* is the only benchmark that is slowed down. In general, the speedups are smaller compared to the single processor quad core. This version of the Xeon processor communicates using the front-side bus, while newer versions use the faster QPI interconnect. Hence, the communication between these processors is quite expensive compared to on-chip communication. To this end, the slow inter-processor communication makes the scheduling of new tasks expensive and results in lower speedups.

7.3.3.1 Speculation

Our speculative approach that was presented in Section 5.2 was implemented using the speculation method described in Section 5.2.1.2: Depending on the type of the variable the speculation either uses a default value for event variables or a value from the preceding iteration for memorized variables. Since this results in uniquely determined values, we restricted the number of speculations to one.

Results of the OOO execution with speculation are printed in Table 7.7 for the i5-750 system and in Table 7.8 for the dual Xeon system. The speedup of the speculative approach refers to the non-speculative OOO execution. In addition to the runtimes of the benchmarks, Tables 7.9 and 7.10 show the hit ratio statistics for the speculative execution on the i5-750 and dual Xeon system, respectively. Column "#Task Executed" denotes the number of actual tasks that have been executed. The remaining columns show the number of speculative executions. Percentages denote "correct" speculations, "speculations" that have not been finished before the actual execution was started, and "wrong" speculations. The remaining percentages denote the speculations that were canceled due to invalid inputs.

The framework for our speculative OOO execution (OoOSpec) of DPNs in software still provides potential for optimizations. For instance, we used a single task queue which becomes a bottleneck in many-core systems. This can be improved by using work stealing algorithms [35]. We ignored these optimizations so far because the framework is considered as a starting point for further research. Hence, it should be kept readable and simple to enable easy integration of future ideas. As a result, the benchmark results should not be considered as an upper limit for our approach.

The execution time of *ParMatrixMult* (16×16) with OOO-Spec is slowed down compared to the execution time with the OOO approach. Despite of the high degree of parallelism, the hit ratio statistic shows that workers still come to the point where a speculation is triggered. For this benchmark the number of speculations is relatively low and most of these speculations come too late. As a result, this causes a small overhead. Since the OOO approach already achieved a very high speedup, we did not consider the slightly slower execution as an issue. The larger variant of *ParMatrixMult* (32×32) is approximately as fast as the non-speculative variant. The low number of speculations implies that the uti-

lization of the i5 seems to be better for this variant, which results in no significant change of the execution time.

SeqMatrixMult computes only one element of the target matrix per iteration. For that reason, this benchmark does not provide the same trivial parallelism as its parallel implementation. This results in more cores that can do speculative task executions. The hit rate for this application is at least 16% and reaches up to 32.9%. Although all variants are sped up on the i5 (up to a factor of 2.08), the speedup does not scale proportional to the successful speculations. We believe that cache effects in combination with the different matrix and CBS window sizes cause these variances.

LUDecomp (16×16) gains are speedup of 2.09 for the execution with the larger window size ($WS = 16$). The speculative execution of this variant results in a significant higher number of speculations with a higher hit rate. The large version provides tasks with higher computational effort which leads to less idle cores. Hence, the number of speculations for the large version of LUDecomp is generally lower. Moreover, the hitrate is lower, too. This results in a slightly lower execution for $WS = 8$ and a slightly faster execution for $WS = 16$, which has a higher number of successful speculations than the execution for $WS = 8$.

Sqrt results in a good hit ratio ($> 26\%$) with a high speedup (> 2.86). The iterative approximation of the square root of a random number basically allows only two possibilities. This controls most of the actions in this benchmark, which allows a high hit ratio. The probability to make a wrong guess doubles with each iteration step into the future. Thus, the miss rate for the execution with $WS = 16$ increases drastically, but still results in a similar speedup as the execution with $WS = 8$.

Delay, *Pitchshift* and *AudioSynth* strongly depend on the input data, and the speculation results for all applications in a high miss rate. As a result, the smart task selection iteratively reduces the interval for speculative task selection to zero. The speculation is finally deactivated, which leads to a low count of speculative task executions. Hence, one would expect a speedup close to 1.0. At that point, it is not clear why the execution using the speculative approach results in speedups of 1.86 (Delay,WS=8), > 2.40 (Pitchshift,WS=8 and WS=16) and > 1.15 (AudioSynth,WS=8 and WS=16). We believe that the light-weight tasks of these benchmarks result in a high access rate for the single task queue, which results in a bottleneck in non-speculative OOO execution as described at the beginning of this section. The speculative OOO execution uses a different method to get a token from the queue, which may be the reason for a reduction of synchronization overhead and the measured speedup.

DFT allows a high degree of parallelism in each iteration, but not across the iterations. Therefore, the OOO execution is not able to significantly accelerate the in-order execution. Nevertheless, the control-flow for this application is trivial and can be perfectly speculated. The speculative execution achieves a high count of speculatively executed tasks with a high hit ratio, which leads to a speedup of ≥ 1.76.

3D Transform can be accelerated by the speculative approach. Despite of the low number of speculative task executions, the speculation is able to give a significant speedup in the execution of 3D Transform. This is explained by the usage of a transformation

matrix that is rarely changed, and therefore, it does not have to be communicated to other tasks. Hence, tasks depending on this matrix can be started earlier. This confirms the effectiveness of the motivation by false dependencies (see Section 5.2.1.1). Note that the message passing execution with general communication optimization (see Section 7.3.2.2) led to a speedup that was due to the same fact.

The speculation in *Mandelbrot* is quite successful regarding the hit rate. Nevertheless, the initialization phase to start a speculation is quite expensive regarding memory bandwidth. Although the results of the speculations are computed in time, i. e., not too late, the allocation and copy process seem to need additional bandwidth. Thus, the speculative execution is slower than the non-speculative OOO execution, but still faster than the in-order execution.

The speculative execution of the small variant of *Landscape* generator ($H = 200$) results in a speedup of a factor close to 1.0. The overhead for speculation seems to slightly exceed the benefit of successful speculation. In contrast, the speculative execution of the larger variant results in a slow down. The larger data sets used in this benchmark lead to an additional overhead in the initialization phase of a speculation. For that reason, nearly all speculations arrive too late and simply lead to an overhead without any benefit. According to our log files, speculation for Landscape ($H = 800$) has been completely deactivated. A longer execution of the speculative variant will likely result in the same speed as the non-speculative OOO execution.

We get a very high amount of speculative task executions for the *Ray Tracer* benchmark. The miss rate of the speculation is very high, which is likely due to the random numbers used in many computations of the stochastic algorithm. Hence, the speculation introduces a significant overhead without benefit, which results in a slow down of the speculative execution. In contrast to Landscape, the execution did not reach the state to stop speculation. We expect that a longer run would completely deactivate the speculation and end up with a similar execution speed as the non-speculative execution.

On the i5 system, most benchmarks can be further sped up using our speculation approach. The average speedup compared to the OOO execution is 1.71 ($WS = 8$) and 1.59 ($WS = 16$). Except for *Landscape* (H=800) and *Ray Tracer*, the applications are either sped up or stay close to the execution time of the OOO approach. According to the log files of the speculative execution, *Landscape* (H=800) already turned off speculation and *Ray Tracer* is likely to turn it off in a longer run. Hence, these benchmarks would continue their execution with the same speed as the non-speculative version.

In contrast, the speculative execution on the Xeon system results in a slow down for all benchmarks but *ParMatMult*. As already mentioned, the inter-processor connection is slower than the one of the i5. Although a speculation is only triggered for idle cores, the speculative execution of a task still requires memory bandwidth to initialize a structure to store its results. The additional effort to initialize this structure impairs the concurrent execution of actual tasks. We assume that this effect is exaggerated by the higher number of cores that can speculate in the Xeon system. Moreover, this system shows that the hit rate for speculative execution is not a guarantee for a speedup. Handling these issues

remains to be done, for instance by limiting the number of workers that are allowed to execute speculative tasks.

We believe that the performance of the speculation approach may be generally increased by taking locally cached data into account. In particular, if we had knowledge about the cached data in a processor, we could consider this information in the task selection algorithm. Selecting tasks for speculation whose data is already in cache would eventually reduce accesses to the shared memory. Hence, this would leave more memory bandwidth to actual task executions.

In general, the hit ratio statistics for the benchmarks show that the hit ratio is quite high (see Tables 7.9 and 7.10). Note that this is not due to good speculation about input values, but due to dynamically adapting the selection of tasks which prefers tasks with a high hit ratio. The targeted hit ratio is controlled by two threshold values as explained in Section 5.2.4. The adaptive selection always starts with an optimistic initialization, i. e., tasks for speculation are taken from the complete CBS. The adaption is done using speculations and checks of speculations. The more speculations are made for an application, the higher the hit ratio becomes. This can be observed for many benchmarks, e. g., for *DFT* and *Mandelbrot*. Some applications do not allow successful speculations due to unpredictable behavior. Speculative execution of these applications will lead to a non-speculative execution after a couple of miss-speculations.

An expected, but still interesting observation is the relationship between window size and hit ratio. A larger window size allows one to do speculations in a more distant future. As explained in Section 5.2.4, the number of known values in the CBS decreases with the distance to the CBS head. Hence, the probability of bad speculations becomes then higher. This is confirmed by all benchmarks on both systems. Moreover, an inverted effect can be observed for speculations that come too late. The probability density of selecting a task for speculation which will be too late depends on the application, the node and the distance to the CBS head. The selection algorithm which selects a task for speculation, is based on a random selection and has a uniform probability distribution in the range of the CBS size. As a result, if the window size of the CBS is increased, the lower the probability becomes that a task is selected that comes too late, and the higher the probability becomes that input values for a task are missing, i. e., the risk for bad speculation rises.

The re-usability of speculations as described in Section 5.2.3 has not been implemented, yet. This work and its analysis remains to be done. Moreover, an integration into the smart task selection (see Section 5.2.4) is desirable because it will continue the idea of dynamic execution and adapation.

Table 7.10: Speculation hit ratio for OOO execution with speculation on 2x Xeon X5450.

Benchmark	#Task Executed	#Tasks Speculated ∑	correct	too late	wrong
ParMatMult(T=8, 16 × 16, WS=8)	900052	116615	32.46%	17.84%	49.70%
ParMatMult(T=8, 16 × 16, WS=16)	900093	75957	10.75%	6.31%	82.94%
ParMatMult(T=8, 32 × 32, WS=8)	900046	42525	33.98%	33.75%	32.28%
ParMatMult(T=8, 32 × 32, WS=16)	900078	34527	21.82%	12.94%	65.24%
SeqMatMult(T=8, 16 × 16, WS=8)	46269	36911	24.56%	0.11%	73.91%
SeqMatMult(T=8, 16 × 16, WS=16)	46286	36953	12.50%	0.01%	86.10%
SeqMatMult(T=8, 32 × 32, WS=8)	348500	226207	54.58%	8.13%	36.77%
SeqMatMult(T=8, 32 × 32, WS=16)	348517	265902	29.85%	2.09%	67.33%
LUDecomp(T=8, 16 × 16, WS=8)	687117	173089	23.81%	8.62%	67.56%
LUDecomp(T=8, 16 × 16, WS=16)	936494	462942	35.57%	3.31%	61.12%
LUDecomp(T=8, 32 × 32, WS=8)	259564	52807	9.07%	16.65%	74.27%
LUDecomp(T=8, 32 × 32, WS=16)	314069	114282	16.56%	4.95%	78.49%
Sqrt(T=8, B=1024, WS=8)	358925	169642	29.78%	6.09%	59.07%
Sqrt(T=8, B=1024, WS=16)	364358	244948	20.02%	1.99%	76.66%
Delay(T=8, 200msec, WS=8)	350030	116146	12.26%	23.67%	64.06%
Delay(T=8, 200msec, WS=16)	350033	206396	3.39%	6.65%	89.96%
DFT(T=8, WS=8)	550033	352467	68.69%	3.71%	27.60%
DFT(T=8, WS=16)	550037	400701	53.75%	2.11%	44.14%
Pitchshift(T=8, 200msec, WS=8)	1000046	253658	39.86%	9.95%	50.19%
Pitchshift(T=8, 200msec, WS=16)	1000088	365426	17.61%	2.98%	79.42%
AudioSynth(T=8, C=16, WS=8)	2160941	348605	0.17%	13.07%	86.75%
AudioSynth(T=8, C=16, WS=16)	2160947	975634	0.11%	7.48%	92.41%
3D Transform(T=8, WS=8)	1000060	528059	46.69%	8.69%	44.63%
3D Transform(T=8, WS=16)	1000125	279255	30.86%	8.69%	60.45%
ImageScale(T=8, WS=8)	40019	8696	10.83%	72.08%	17.09%
ImageScale(T=8, WS=16)	40056	6893	27.61%	32.71%	39.68%
Mandelbrot(T=8, W=256, WS=8)	467260	271083	32.79%	6.86%	60.35%
Mandelbrot(T=8, W=256, WS=16)	467340	334828	22.08%	2.29%	75.63%
Landscape(T=8, H=200, W=4, WS=8)	516835	118026	33.16%	41.46%	25.38%
Landscape(T=8, H=200, W=4, WS=16)	516923	106172	34.25%	26.40%	39.34%
Landscape(T=8, H=800, W=4, WS=8)	4166635	149226	10.92%	73.77%	15.30%
Landscape(T=8, H=800, W=4, WS=16)	4166723	467269	27.17%	30.78%	42.04%
Ray Tracer(T=8, H=4, W=4, WS=8)	162545	122782	6.04%	2.38%	87.30%
Ray Tracer(T=8, H=4, W=4, WS=16)	162553	132608	1.12%	0.02%	94.97%

Chapter 8

Summary and Conclusions

The manifoldness of parallel hardware architectures and different programming models, and the need to abstract from hardware details to allow programmers to focus on the functionality of a system endorse the model-based approach of development. This is supported by synchronous languages, which abstract from communication costs.

This thesis considered the creation of parallel software from SGAs without any user-interaction. We have shown that the translation of the SGAs to DPNs as a further intermediate format provides (1) a well-known format to create distributed parallel software, and (2) the ability of OOO execution due to the synchronous MoC of the source.

In Chapter 3, we presented a method to derive a structured graph to execute a set of SGAs according to their dependencies. This is necessary to meet the execution model of OpenMP. In general, the task-based execution is very attractive because it already provides an abstraction from the actual available hardware. However, parallelism must be explicitly indicated by the programmer, which makes analysis and resolution of dependencies between parallel code mandatory. A slightly different approach is provided by SmpSS. It allows a more appropriate execution model of SGAs, i. e., the execution of tasks according to their dependencies. The need for the complex structuring algorithm will likely be dispensable because both APIs are going to be merged to OmpSS [8]. Both targets are restricted to SMP systems. The created code for both targets is only capable of exploiting parallelism in a single iteration, i. e., parallelism across several iterations is not considered.

Chapter 4 introduces the translation of SGAs to DPNs to gain more parallelism and more flexibility with respect to the targeted architecture. The basic idea is to decouple tasks from the previous approach by inserting FIFO buffers. Nodes can then be mapped to different processing units, e. g., processors in distributed memory systems. While the approach in Chapter 3 is restricted to SMP architectures, the synthesis of DPNs allows the execution in clusters, e. g., using MPI. The additional flexibility with respect to the target architecture comes with additional communication costs. The original synchronous MoC is reflected in the created DPNs: nodes have to consume and produce exactly one token in each iteration. We addressed this issue in Section 4.2 and 4.3. While the former tackles the reduction of costs introduced by redundant array communication, the latter considers

the general reduction based on the idea to communicate a value only if it is changed and needed.

An execution model for OOO execution is presented in Chapter 5. The idea is inspired by processor design and considers to concurrently fire a node in a synchronous DPN. Varying computational effort of these iterations may lead to out-of-order arrival of results. Instead of reordering the outputs of a node, we centralize all buffers into a global ring buffer (CBS). Out-of-order arrival of tokens may then trigger the iterations of other nodes out-of-order. This provides several advantages: (1) The task-based model allows us to use light-weight tasks, which abstracts from the hardware. (2) We gain more parallelism, particularly over the iterations of a node, and (3) we get a better load-balancing.

In consequence of the inspiration by hardware design, we added speculation to the OOO execution. This is motivated by two different characteristics: Similar to the general communication optimization (see Section 4.3), we assume that a value is not always required for a computation. Hence, an execution may start before all its required inputs are available. Second, a value is not always modified, i. e., the input value for the execution of a task may be left unmodified by the respective producer. This is encouraged by the usage of Boolean variables that are introduced in the compilation of synchronous programs to build the control-flow. This is basically necessary to activate the correct actions in a macro step of a synchronous program.

In Chapter 6, we discussed some general implementation details in parallel programming which have particular uses in this thesis. For instance, DPNs use buffers where the writer and reader of each buffer is uniquely determined. Moreover, writes to and reads from a queue are done in short intervals. This allows us to optimize a concrete implementation to use spinlocks instead of semaphores.

All presented approaches have been implemented and evaluated in Section 7. We have shown that our applications were effectively executed on different parallel architectures, e. g., in cluster architectures with distributed memory. Approaches to reduce communication effort have been successfully applied and improved the execution time.

Our OOO execution improves the parallel execution on SMP architectures compared to task-based in-order execution. Hence, the dynamic and concurrent execution of several iterations of a single node in a synchronous DPN was successful. In addition, the idea of adding speculation to the OOO execution was able to improve some of the applications. The effect of the partitioning strongly depends on many parameters: the application, its partitioning and the target architecture. Therefore, we use a dynamic adaption that basically deactivates the speculation in case of too many miss-speculations. The different results on similar architectures, i. e., Intel x86 architectures, confirm our big goal to provide dynamic execution models.

Beside the presented approaches and their implementations, there remains work to be done. The integration of partitioning tools and their effect on the results remain to be examined. More important, memory architectures, particularly weak memory models, have been roughly considered in this work. They allow the design of more efficient hardware, but require careful programming due to their subtle behavior.

Parallel computing is a wide research area. It is very hard to cover everything in the world of parallel computing. There is a large gap between a convenient and flexible programming model and an implementation for particular hardware. Most of the research and implementations done in this thesis starts from the former side to close this gap. On the other side, APIs provide an abstraction from hardware, which allows a convenient creation of parallel code. To this end, the steadily growing complexity of both, newly designed systems and targeted hardware architectures, and the steadily rising variety in hardware design makes the development of compilers an interesting challenge.

Bibliography

[1] AlsaModularSynth. http://alsamodular.sourceforge.net/.

[2] Analog days: The invention and impact of the moog synthesizer. http://www.moogmusic.com/legacy/bob-moog-timeline.

[3] Doom source code. ftp://ftp.idsoftware.com/idstuff/source/.

[4] NATIVE INSTRUMENTS: Products : Synth line : Reaktor 5 (another wiring-oriented music "programming" system). http://www.native-instruments.com/index.php?id=reaktor5_us.

[5] P. Randive A. Athavale and A. Kambale. Automatic parallelization of sequential codes using s2p tool and benchmarking of the generated parallel codes. http://www.kpitcummins.com/downloads/research-papers/automatic-parallelization-sequential-codes.pdf.

[6] J. Allen, editor. *Software synthesis from dataflow graphs*. Kluwer Academic Publishers, 1996.

[7] M. Amini, O. Goubier, S. Guelton, J. O. Mcmahon, F. Pasquier, G. Péan, and P. Villalon. Par4all: From convex array regions to heterogeneous computing. IMPACT, 2012. http://impact.gforge.inria.fr/impact2012.

[8] M. Andersch, C. Ching Chi, and B.H.H. Juurlink. Using OpenMP superscalar for parallelization of embedded and consumer applications. In J. McAllister and S. Bhattacharyya, editors, *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS)*, pages 23–32, Samos, Greece, 2012. IEEE Computer Society.

[9] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers (T-C)*, 39(3):300–318, March 1990.

[10] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632, October 1989.

[11] A. Aziz, V. Singhal, F. Balarin, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In P.L. Wolper, editor, *Computer Aided Verification (CAV)*, volume 939 of *LNCS*, pages 155–165, Liège, Belgium, 1995. Springer.

[12] S. Balakrishnan and G.S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *International Symposium on Computer Architecture (ISCA)*, pages 302–313, Boston, Massachusetts, USA, 2006. IEEE Computer Society.

[13] P.S. Barth, R.S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. Technical Report CSG-MEMO 327, Computer Science and Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA, 1991.

[14] D. Baudisch, Y. Bai, and K. Schneider. Reducing the communication of message-passing systems synthesized from synchronous programs. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Turin, Italy, 2014. IEEE Computer Society.

[15] D. Baudisch, J. Brandt, and K. Schneider. Dependency-driven distribution of synchronous programs. In M. Hinchey, B. Kleinjohann, L. Kleinjohann, P.A. Lindsay, F.J. Rammig, J. Timmis, and M. Wolf, editors, *Distributed and Parallel Embedded Systems (DIPES)*, pages 169–180, Brisbane, Queensland, Australia, 2010. International Federation for Information Processing (IFIP).

[16] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In *Design, Automation and Test in Europe (DATE)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.

[17] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 11–20, Dresden, Germany, 2010. Fraunhofer Verlag.

[18] D. Baudisch, J. Brandt, and K. Schneider. Translating synchronous systems to data-flow process networks. In S.-S. Yeo, B. Vaidya, and G.A. Papadopoulos, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 354–361, Gwangju, Korea, 2011. IEEE Computer Society.

[19] D. Baudisch, J. Brandt, and K. Schneider. Efficient handling of arrays in dataflow process networks. In *International Conference on Embedded Software and Systems (ICESS)*, pages 1395–1402, Liverpool, United Kingdom, 2012. IEEE Computer Society.

[20] D. Baudisch, J. Brandt, and K. Schneider. Out-of-order execution of synchronous data-flow networks. In J. McAllister and S. Bhattacharyya, editors, *International*

Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS), pages 168–175, Samos, Greece, 2012. IEEE Computer Society.

[21] D. Baudisch and K. Schneider. Evaluation of speculation in out-of-order execution of synchronous data-flow networks. *International Journal of Parallel Programming (IJPP)*, pages 1–44, October 2013. http://dx.doi.org/10.1007/s10766-013-0277-2.

[22] K. Bauer. *A New Modelling Language for Cyber-physical Systems.* PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, January 2012. PhD.

[23] A. Benveniste, P. Bournai, T. Gautier, and P. Le Guernic. SIGNAL: A data flow oriented language for signal processing. Research Report 378, Institut National de Recherche en Informatique et en Automatique (INRIA), Rennes, France, March 1985.

[24] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[25] K. Berlin, J. Huan, M. Jacob, G. Kochhar, J. Prins, W. Pugh, P. Sadayappan, J. Spacco, and C.-W. Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing (LCPC)*, volume 2958 of *LNCS*, pages 194–208, College Station, Texas, USA, 2004. Springer.

[26] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 1998.

[27] G. Berry. The constructive semantics of pure Esterel, July 1999.

[28] G. Berry and L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency (CONCUR)*, volume 197 of *LNCS*, pages 389–448, Pittsburgh, Pennsylvania, USA, 1985. Springer.

[29] S. Bhattacharyya and E.A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *The Journal of VLSI Signal Processing*, 6(3):271–288, 1992.

[30] S.S. Bhattacharyya, G. Brebner, J.W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF-a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News*, 36(5):29–35, December 2009.

[31] S.S. Bhattacharyya and E.A. Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. *Formal Methods in System Design (FMSD)*, 5(3):183–205, December 1994.

[32] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[33] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Static scheduling of multi-rate and cyclo-static DSP-applications. In *VLSI Signal Processing*, pages 137–146, La Jolla, California, USA, 1994. IEEE Computer Society.

[34] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E: Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Notices*, 30(8):207–216, August 1995.

[35] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[36] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface*, GI '07, pages 177–184. ACM, 2007.

[37] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[38] J. Brandt, M. Gemünde, and K. Schneider. Desynchronizing synchronous programs by modes. In S. Edwards, R. Lorenz, and W. Vogler, editors, *Application of Concurrency to System Design (ACSD)*, pages 32–41, Augsburg, Germany, 2009. IEEE Computer Society.

[39] J. Brandt and K. Schneider. Separate compilation for synchronous programs. In H. Falk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, volume 320 of *ACM International Conference Proceeding Series*, pages 1–10, Nice, France, 2009. ACM.

[40] J. Brandt, K. Schneider, and Y. Bai. Passive code in synchronous programs. *Transactions on Embedded Computing Systems (TECS)*, 2014.

[41] J. Bruck, D. Dolev, C. Ho, M. Roşu, and R. Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. In C.E. Leiserson, editor, *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 64–73, Santa Barbara, California, USA, 1995. ACM.

[42] J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, 1995.

[43] J. Buck and E.A. Lee. The token flow model. In L. Bic, G.R. Gao, and J.-L. Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*, pages 267–290, Hamilton Island, Queensland, Australia, 1995. IEEE Computer Society.

[44] S. Burckhardt, R. Alur, and M.M.K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In T. Ball and R.B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 489–502, Seattle, Washington, USA, 2006. Springer.

[45] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In A. Gupta and S. Malik, editors, *Computer Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 107–120, Princeton, New Jersey, USA, 2008. Springer.

[46] L.P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 146(2):61–80, 2006.

[47] L.P. Carloni, K.L. McMillan, and A. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 123–133, Trento, Italy, 1999. Springer.

[48] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 20(9):1059–1076, 2001.

[49] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering (T-SE)*, 25(3):416–427, May/June 1999.

[50] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL)*, pages 178–188, Munich, Germany, 1987. ACM.

[51] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[52] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, USA, May 1989.

[53] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318–331, 2008.

[54] M.H. Cintra, J.F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 13–24, Vancouver, British Columbia, Canada, 2000. ACM.

[55] C.B. Colohan, A.C. Ailamaki, J.G. Steffan, and T.C. Mowry. CMP support for large and dependent speculative threads. *IEEE Transactions on Parallel and Distributed Systems*, 18(8):1041–1054, August 2007.

[56] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[57] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow (extended version). Internal Report LSI-07-13-R, Universitat Politècnica de Catalunya, Barcelona, Spain, 2007.

[58] J. Cortadella, M. Kishinevsky, D. Bufistov, J. Carmona, and J. Júlvez. Elasticity and Petri nets. In K. Jensen, W.M.P. van der Aalst, and J. Billington, editors, *Transactions on Petri Nets and Other Models of Concurrency (Volume 1)*, volume 5100 of *LNCS*, pages 221–249. Springer, 2008.

[59] J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and design of synchronous elastic circuits. In *Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 2006.

[60] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In E. Sentovich, editor, *Design Automation Conference (DAC)*, pages 657–662, San Francisco, California, USA, 2006. ACM.

[61] J.B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[62] J.B. Dennis and D. Misunas. A preliminary architecture for a basic data-flow processor. In *25 Years of the International Symposia on Computer Architecture (ISCA)*, pages 125–131, Barcelona, Spain, 1998. ACM.

[63] J.B. Dennis, D.P. Misunas, and P.S. Thiagarajan. Data-flow computer architecture. Technical Report CSG-MEMO 104, MIT Lab for Computer Science, Cambridge, Massachusetts, USA, August 1974.

[64] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM (CACM)*, 18(8):453–457, 1975.

[65] D.L. Dill. The Murphi verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, New Jersey, USA, 1996. Springer.

[66] A. Dorta, J. Badía, E. Quintana, and F. Sande. Implementing openmp for clusters on top of mpi. In Beniamino Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 148–155. Springer Berlin Heidelberg, 2005.

[67] S. Edwards. Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2):141–187, 2003.

[68] S.A. Edwards. Making cyclic circuits acyclic. In *Design Automation Conference (DAC)*, pages 159–162, Anaheim, California, USA, 2003. ACM.

[69] J. Eker and J.W. Janneck. CAL language report. ERL Technical Memo UCB/ERL M03/48, EECS Department, University of California at Berkeley, Berkeley, California, USA, December 2003.

[70] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow: Model and implementation. In *Asilomar Conference on Signals, Systems and Computers (ACSSC)*, Pacific Grove, California, USA, 1994. IEEE Computer Society.

[71] Y. Etsion, F. Cabarcas, A. Rico, A. Ramírez, R.M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In H. Kim, A. Raman, F. Liu, J.W. Lee, and D.I. August, editors, *Microarchitecture (MICRO)*, pages 89–100, Atlanta, Georgia, USA, 2010. IEEE Computer Society.

[72] K. J. Falconer. *Fractal geometry: mathematical foundations and applications.* Wiley, 1990.

[73] J. L. Flanagan and R. M. Golden. Phase vocoder. *Bell System Technical Journal*, 45:1493–1509, 1966.

[74] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk 5 multithreaded language. In J.W. Davidson and K.D. Cooper, editors, *Programming Language Design and Implementation (PLDI)*, pages 212–223, Montréal, Québec, Canada, 1998. ACM.

[75] G.R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved programs for DSP computation. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 561–564, San Francisco, California, USA, 1992. IEEE Computer Society.

[76] D. Genin, J. De Moortel, D. Desmet, and E. van de Velde. System design, optimization, and intelligent code generation for standard digital signal processors. In *International Symposium on Circuits and Systems (ISCAS)*, pages 565–569, Portland, Oregon, USA, 1989. IEEE Computer Society.

[77] A. Ghamarian, M. Geilen, T. Basten, B. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous dataflow graphs. Research Report ESR-2006-04, Eindhoven, The Netherlands University of Technology, 2006.

[78] A. Girault. A survey of automatic distribution method for synchronous programs. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–20, Edinburgh, Scotland, UK, 2005. unpublished workshop proceedings.

[79] A. Girault and X. Nicollin. Clock-driven automatic distribution of Lustre programs. In R. Alur and I. Lee, editors, *Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 206–222, Philadelphia, Pennsylvania, USA, 2003. Springer.

[80] J. Gleick. *Chaos: Making a New Science*. Open Road, 1988.

[81] R. Guerraoui, T.A. Henzinger, and V. Singh. Software transactional memory on relaxed memory models. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification (CAV)*, volume 5643 of *LNCS*, pages 321–336, Grenoble, France, 2009. Springer.

[82] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, July 1996.

[83] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[84] N. Halbwachs. A synchronous language at work: the story of Lustre. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 3–11, Verona, Italy, 2005. IEEE Computer Society.

[85] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[86] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In D. Bhandarkar and A. Agarwal, editors, *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, San Jose, California, USA, 1998. ACM.

[87] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[88] J.P. Hoeflinger and B.R. de Supinski. The OpenMP memory model. In R. Jhala and D.A. Schmidt, editors, *International Workshops on OpenMP Shared Memory Parallel Programming (IWOMP)*, volume 4315 of *LNCS*, pages 167–177, Eugene, Oregon, USA and Reims, France, 2008. Springer.

[89] L. Huang, G. Sethuraman, and B. Chapman. Parallel data flow analysis for openmp programs. In *Proceedings of the 3rd international workshop on OpenMP: A Practical Programming Model for the Multi-Core Era*, IWOMP '07, pages 138–142. Springer-Verlag, 2008.

[90] Intel. Intel threading building blocks TBB, 2013. `http://threadingbuildingblocks.org/`.

[91] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems (SiPS)*, pages 287–292, Washington, District of Columbia, USA, 2008. IEEE Computer Society.

[92] J. C. Jenista, Y. Eom, and B. C. Demsky. Ooojava: software out-of-order execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 57–68. ACM, 2011.

[93] James C. Jenista, Yong Hun Eom, and Brian Demsky. Ooojava: an out-of-order approach to parallel programming. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 11–11. USENIX Association, 2010.

[94] T.A. Johnson, R. Eigenmann, and T.N. Vijaykumar. Min cut program decomposition for thread level speculation. In C. Chambers, editor, *Programming Language Design and Implementation (PLDI)*, pages 59–70, Washington, District of Columbia, USA, 2004. ACM.

[95] W.M. Johnston, J.R.P. Hanna, and R.J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, March 2004.

[96] H. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.

[97] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, 1974. North-Holland.

[98] R.M. Karp and R.E. Miller. Parallel program schemata. In *Switching and Automata Theory*, pages 55–61. IEEE Computer Society, 1967.

[99] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[100] I.H. Kazi and D.J. Lilja. Coarse-grained thread pipelining – a speculative parallel execution model for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):952–966, September 2001.

[101] P.M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.

[102] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In A. Gupta and P. Manolios, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 19–30, San Jose, California, USA, 2006. IEEE Computer Society.

[103] S.R. Kunkel and J.E. Smith. Optimal pipelining in supercomputers. In *International Symposium on Computer Architecture (ISCA)*, pages 404–414, Tokyo, Japan, 1986.

[104] L. Lamport. Concurrent reading and writing. *Communications of the ACM (CACM)*, 20(11):806–811, November 1977.

[105] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[106] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[107] F. Le Mentec, T. Gautier, and V. Danjean. The X-Kaapi's application programming interface. part I: Data flow programming. Technical Report RT-0418, Institut National de Recherche en Informatique et en Automatique (INRIA), November 2011.

[108] B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, 27(8):27–39, August 1994.

[109] E.A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2(2), 1991.

[110] E.A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[111] E.A. Lee. Computing needs time. *Communications of the ACM (CACM)*, 52(5):70–79, May 2009.

[112] E.A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference (GLOBECOM)*, pages 1279–1283. IEEE Computer Society, 1989.

[113] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers (T-C)*, 36(1):24–35, January 1987.

[114] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[115] E.A. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[116] S.-I. Lee, T.A. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing (LCPC)*, volume 2958 of *LNCS*, pages 539–553, College Station, Texas, USA, 2004. Springer.

[117] X. Li, M. Boldt, and R. von Hanxleden. Compiling Esterel for a multi-threaded reactive processor. Technical Report 0603, Christian-Albrechts-Universität Kiel, Department of Computer Science, 2006.

[118] X. Li, M. Boldt, and R. von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In J.P. Shen and M.R. Martonosi, editors, *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 303–314, San Jose, California, USA, 2006. ACM.

[119] D.J. Lilja. Reducing the branch penalty in pipelined processors. *IEEE Computer*, 21(7):47–55, July 1988.

[120] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *Microarchitecture (MICRO)*, pages 226–237, Paris, France, 1996. IEEE Computer Society.

[121] G. Liu and T. S. Abdelrahman. Computation-communication overlap on network-of-workstation multiprocessors. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1635—1642, 1998.

[122] C. Madriles, P. López, J.M. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez, and A. González. Boosting single-thread performance in multi-core systems through fine-grain multi-threading. In S.W. Keckler and L.A. Barroso, editors, *International Symposium on Computer Architecture (ISCA)*, pages 474–483, Austin, Texas, USA, 2009. ACM.

[123] A. Malik, A. Girault, and Z. Salcic. Formal semantics, compilation and execution of the GALS programming language DSystemJ. *IEEE Transactions on Parallel and Distributed Systems*, 23(7):1240–1254, July 2012.

[124] S. Malik. Analysis of cyclic combinational circuits. In *International Conference on Computer-Aided Design (ICCAD)*, pages 618–625, Santa Clara, California, USA, 1993. IEEE Computer Society.

[125] P. Marcuello and A. González. Exploiting speculative thread-level parallelism on a SMT processor. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *International Conference on High-Performance Computing and Networking (HPCN)*, volume 1593 of *LNCS*, pages 754–763, Amsterdam, The Netherlands, 1999. Springer.

[126] P. Marcuello, A. González, and J. Tubella. Thread partitioning and value prediction for exploiting speculative thread-level parallelism. *IEEE Transactions on Computers*, 53(2):114–125, February 2004.

[127] P. Marwedel. *Eingebettete Systeme*. eXamen.press. Springer, 2008.

[128] J.R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):44–82, January 1982.

[129] P.E. McKenney. Memory barriers: A hardware view for software hackers. http://www.rdrop.com/users/paulmck, June 2010.

[130] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC, pages 267–275, New York, NY, USA, 1996. ACM.

[131] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson. Hardware implementation of recursive algorithms. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 225–228. IEEE, 2010.

[132] A. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi. Dynamic speculation and synchronization of data dependences. In *International Symposium on Computer Architecture (ISCA)*, pages 181–193, 1997.

[133] P.K. Murthy, S.S. Bhattacharyya, and E.A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design (FMSD)*, 11(1):41–70, July 1997.

[134] K.S. Namjoshi and R.P. Kurshan. Efficient analysis of cyclic definitions. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 394–405, Trento, Italy, 1999. Springer.

[135] R.S. Nikhil. Dataflow programming languages. Technical Report CSG-MEMO 333, Computer Science and Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA, 1991.

[136] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[137] A. Pajuelo, A. González, and M. Valero. Speculative execution for hiding memory latency. In *MEmory performance: DEaling with Applications, systems and architecture (MEDEA)*, pages 49–56, Antibes Juan-les-Pins, France, 2004. ACM.

[138] T.M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Princeton University, 1995. PhD.

[139] J. M. Perez, R. M. Badia, and J. Labarta. A flexible and portable programming model for smp and multi-cores. Technical report, 2007.

[140] K. Pingali and Arvind. Efficient demand-driven evaluation. part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(2):311–333, April 1985.

[141] K. Pingali and Arvind. Efficient demand-driven evaluation. part II. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):109–139, January 1986.

[142] M. R. Portnoff. Implementation of the digital phase vocoder using the fast fourier transform. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 24(3):243–248, 1976.

[143] D. Potop-Butucaru. The Kahn principle for networks of synchronous endochronous programs. In *Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS)*, 2003.

[144] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Application of Concurrency to System Design (ACSD)*, pages 67–76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[145] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design (FMSD)*, 28(2):111–130, 2006.

[146] D.B. Powell, E.A. Lee, and W.C. Newmann. Direct synthesis of optimized DSP assembly from signal flow diagrams. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 553–556, San Francisco, California, USA, 1992. IEEE Computer Society.

[147] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007.

[148] J.M. Pérez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *International Conference on Cluster Computing (CLUSTER)*, pages 142–151, Tsukuba, Japan, 2008. IEEE Computer Society.

[149] C.V. Ramamoorthy and H.F. Li. Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102, 1977.

[150] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-level speculation on a CMP can be energy efficient. In *International Conference on Supercomputing (ICS)*, pages 219–228, Cambridge, Massachusetts, USA, 2005. ACM.

[151] S.E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical Report SMLI TR-92-1, Sun Microsystems, Inc., Mountain View, CA, USA, September 1992.

[152] G. Roquier, C. Lucarz, M. Mattavelli, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour. An integrated environment for HW/SW co-design based on a CAL specification and HW/SW code generators. In *International Symposium on Circuits and Systems (ISCAS)*, pages 799–799, Taipei, Taiwan, 2009. IEEE Computer Society.

[153] J. Rumbaugh. A data flow multiprocessor. *IEEE Transactions on Computers (T-C)*, 26(2):138–146, February 1977.

[154] J.E. Savage. *Models of Computation – Exploring the Power of Computing.* Addison Wesley, 3 edition, 1998.

[155] J. Schmittler, I. Wald, and Philipp Slusallek. Saarcor: A hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 27–36. Eurographics Association, 2002.

[156] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[157] K. Schneider, J. Brandt, and T. Schüle. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.

[158] K. Schneider, J. Brandt, T. Schüle, and T. Türk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–19, Edinburgh, Scotland, UK, 2005.

[159] K. Schneider, J. Brandt, T. Schüle, and T. Türk. Maximal causality analysis. In J. Desel and Y. Watanabe, editors, *Application of Concurrency to System Design (ACSD)*, pages 106–115, Saint-Malo, France, 2005. IEEE Computer Society.

[160] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, 2001. ACM.

[161] Uwe Schöning. *Algorithmik.* Spektrum Akademischer Verlag, 2001.

[162] T.R. Shiple. *Formal Analysis of Synchronous Circuits.* PhD thesis, University of California, Berkeley, California, USA, 1996. PhD.

[163] P. Shirley. *Realistic Ray Tracing.* Ak Peters Series. A K Peters, Limited, 2000.

[164] S. W. Smith. *The scientist and engineer's guide to digital signal processing.* California Technical Publishing, 1997. `http://www.dspguide.com/`.

[165] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, SUN Microsystems, Mountain View, California, USA, April 1994.

[166] R.C. Steinke and G.J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5):800–849, September 2004.

[167] A. Stulova, R. Leupers, and G. Ascheid. Throughput driven transformations of synchronous data flows for mapping to heterogeneous MPSoCs. In J. McAllister and S. Bhattacharyya, editors, *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS)*, pages 144–151, Samos, Greece, 2012. IEEE Computer Society.

[168] S. Suhaib, D. Mathaikutty, and S. Shukla. Dataflow architectures for GALS. In A. Girault and R. de Simone, editors, *Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS)*, Nice, France, 2007. Unpublished Participant's Proceedings.

[169] E. Tejedor, M. Farreras, D. Grove, G. Almasi, and J. Labarta. ClusterSs: a task-based programming model for clusters. In *High performance distributed computing (HPDC)*, pages 267–268, San Jose, California, USA, 2011. ACM.

[170] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196. Springer-Verlag, 2002.

[171] J.E. Thornton. Parallel operation in the Control data 6600. In *AFIPS Fall Joint Computer Conference*, number 2, pages 33–40, 1964.

[172] M.A. Thornton, K. Fazel, R.B. Reese, and C. Traver. Generalized early evaluation in Self-Timed circuits. In *Design, Automation and Test in Europe (DATE)*, pages 255–259, Paris, France, 2002. IEEE Computer Society.

[173] J. Tims, R. Gupta, and M. Lou Soffa. Data flow analysis driven dynamic data partitioning. In D.R. O'Hallaron, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR)*, volume 1511 of *LNCS*, pages 75–90, Pittsburgh, Pennsylvania, USA, 1998. Springer.

[174] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[175] K.R. Traub, G.M. Papadopoulos, M.J. Beckerle, J.E. Hicks, and J. Young. Overview of the Monsoon project. Computation Structures Group Memo 338, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, October 1991.

[176] N. Vachharajani, R. Rangan, E. Raman, M.J. Bridges, G. Ottoni, and D.I. August. Speculative decoupled software pipelining. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 49–59, Brasov, Romania, 2007. IEEE Computer Society.

[177] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[178] C. Walshaw and M. Cross. JOSTLE: Parallel multilevel graph-partitioning software – an overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.

[179] M. Wipliez, G. Roquier, and J.-F. Nezan. Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems*, 63(2):203–213, May 2011.

[180] J.R. Wright and J.C.L. Hsieh. A voxel-based, forward projection algorithm for rendering surface and volumetric data. In *IEEE Conference on Visualization*, pages 340–348. IEEE, 1992.

[181] A. Xie and P.A. Beerel. Efficient state classification of finite state Markov chains. In *Design Automation Conference (DAC)*, pages 605–610, San Francisco, California, USA, 1998. ACM.

[182] X. Yuan, R. Gupta, and R. Melhem. An array data flow analysis based communication optimizer. In Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan, and D. Sehr, editors, *Languages and Compilers for Parallel Computing (LCPC)*, volume 1366 of *LNCS*, pages 246–260, Minneapolis, Minnesota, USA, 1998. Springer.

[183] J. Zeng, C. Soviani, and S.A. Edwards. Generating fast code from concurrent program dependence graphs. In D. Whalley and R. Cytron, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 175–181, Washington, District of Columbia, USA, 2004. ACM.

[184] Y. Zhou and E.A. Lee. A causality interface for deadlock analysis in dataflow. In S.L. Min and W. Yi, editors, *Embedded Software (EMSOFT)*, pages 44–52, Seoul, South Korea, 2006. ACM.

[185] C.B. Zilles and G.S. Sohi. Master/slave speculative parallelization. In *Microarchitecture (MICRO)*, pages 85–96, Istanbul, Turkey, 2002. IEEE Computer Society.

Appendix A

DPNs of Benchmark Applications

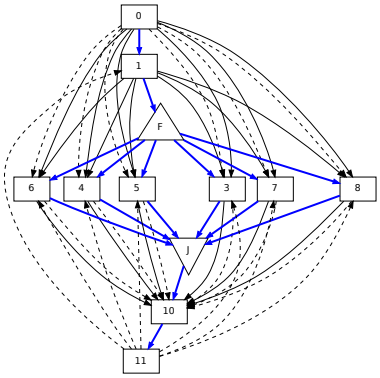


Figure A.1: Generated DPN for ParMatMult (16×16) and ParMatMult (32×32)

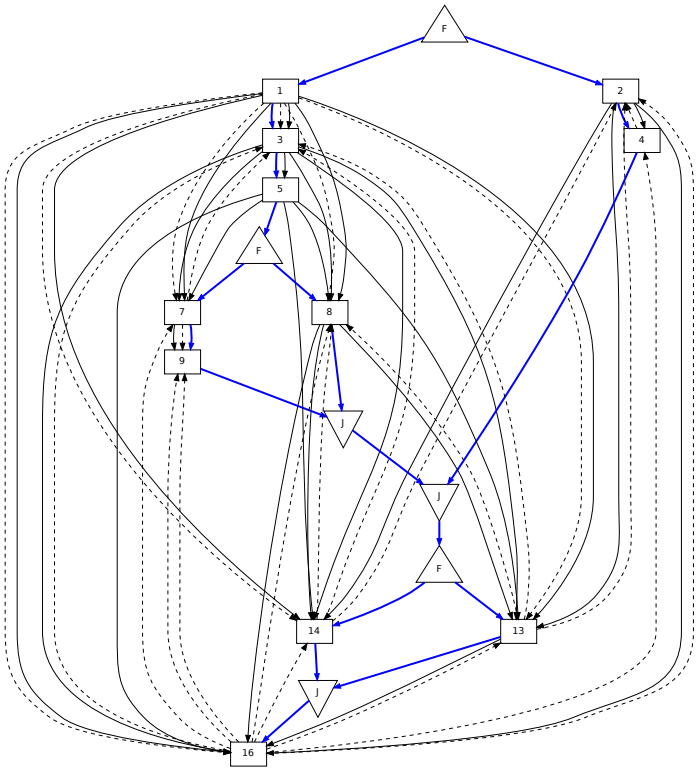


Figure A.2: Generated DPN for SeqMatMult (16×16) and SeqMatMult (32×32)

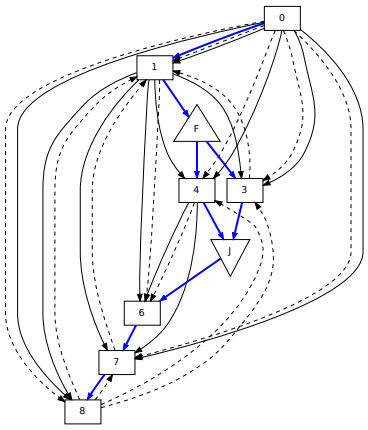


Figure A.3: Generated DPN for LUDecomp (16×16) and LUDecomp (32×32)

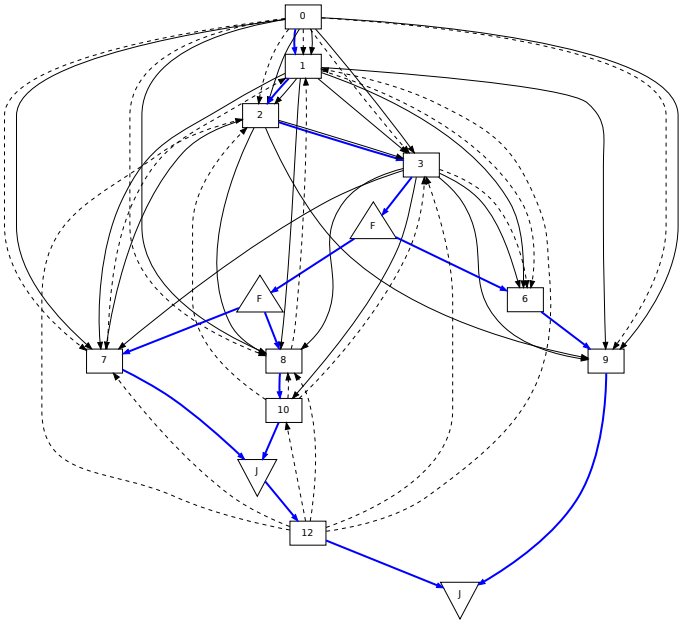


Figure A.4: Generated DPN for Sqrt

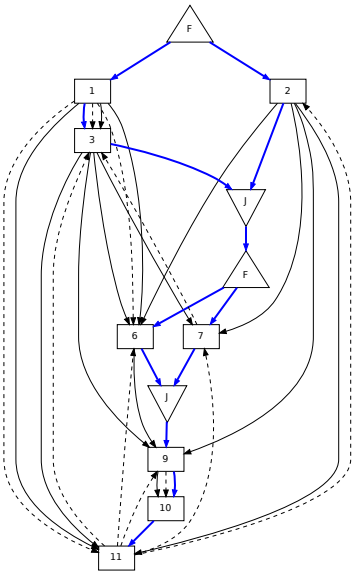


Figure A.5: Generated DPN for Delay

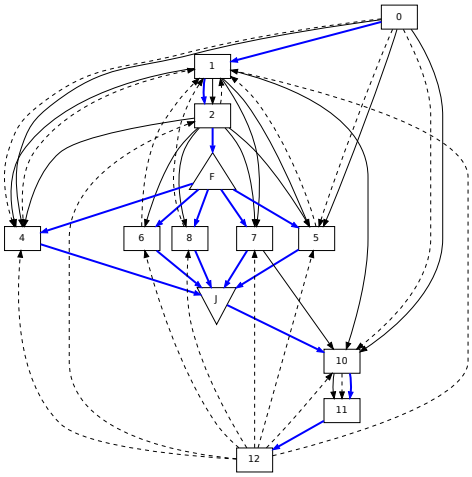


Figure A.6: Generated DPN for DFT

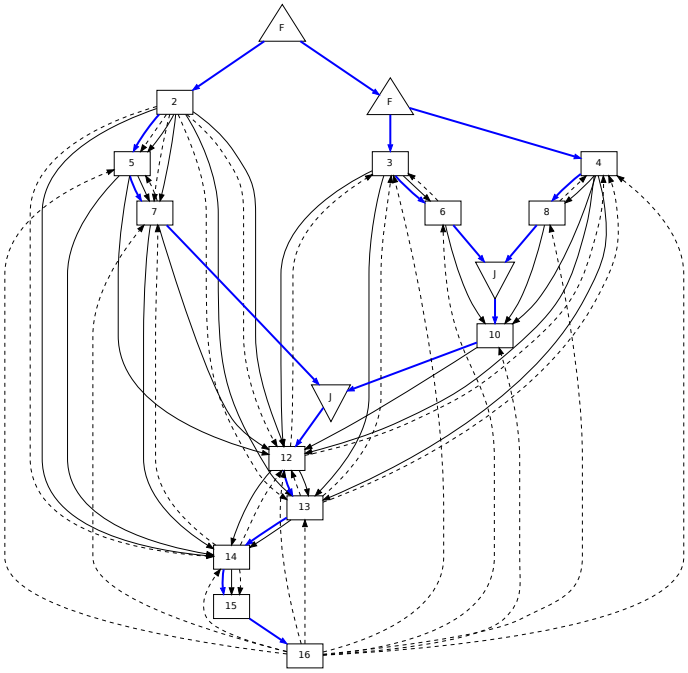


Figure A.7: Generated DPN for AudioSynth

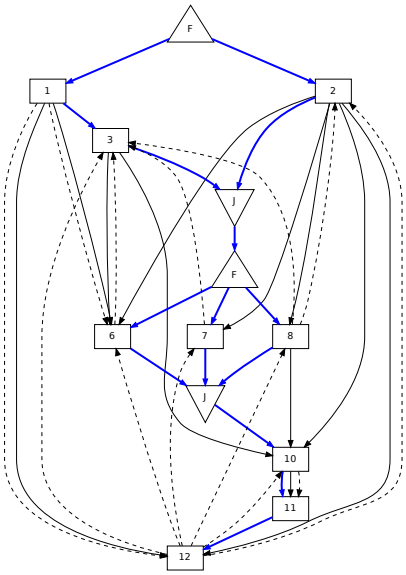


Figure A.8: Generated DPN for Pitchshift

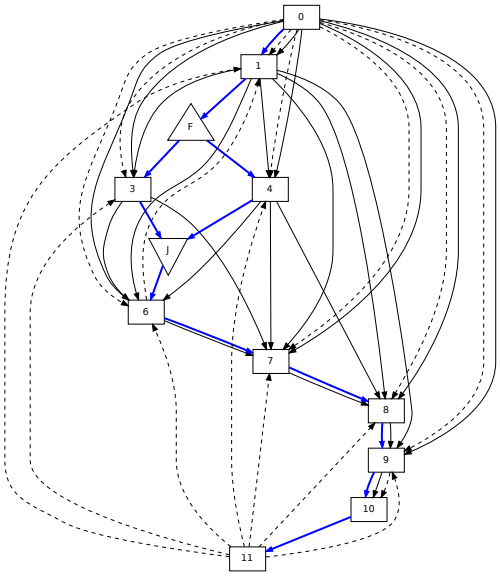


Figure A.9: Generated DPN for 3D Transform

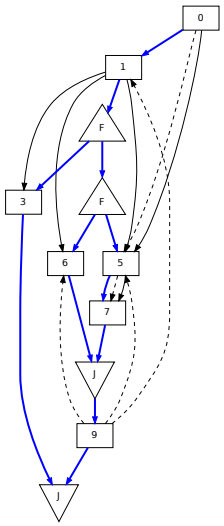


Figure A.10: Generated DPN for ImageScale

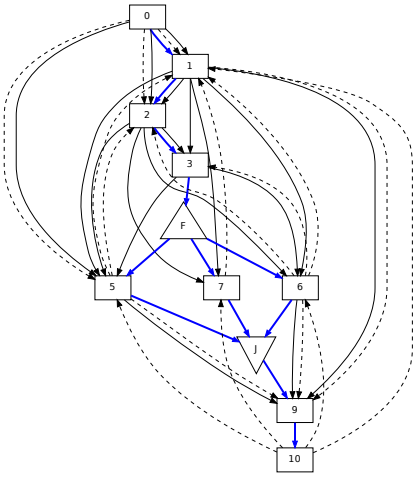


Figure A.11: Generated DPN for Mandelbrot

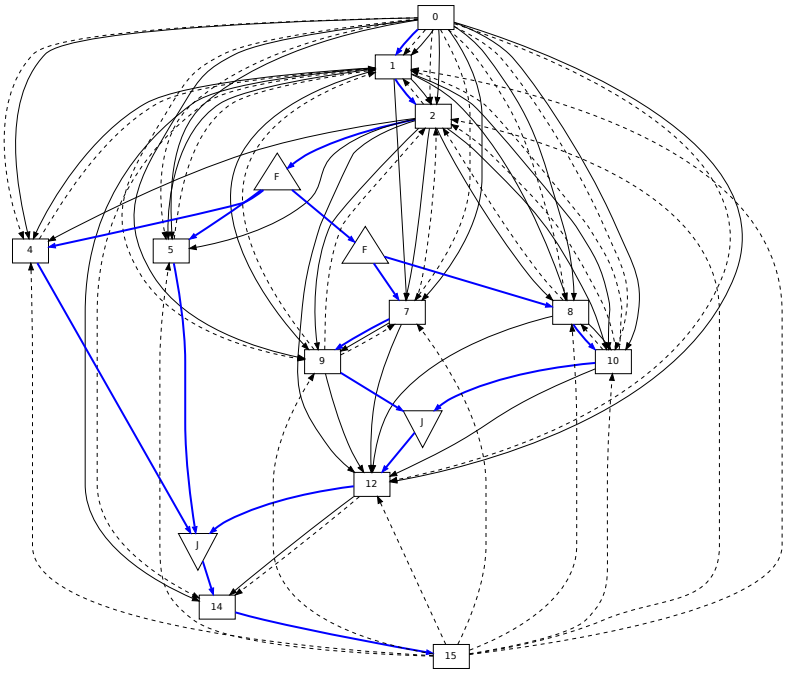


Figure A.12: Generated DPN for FacilityR

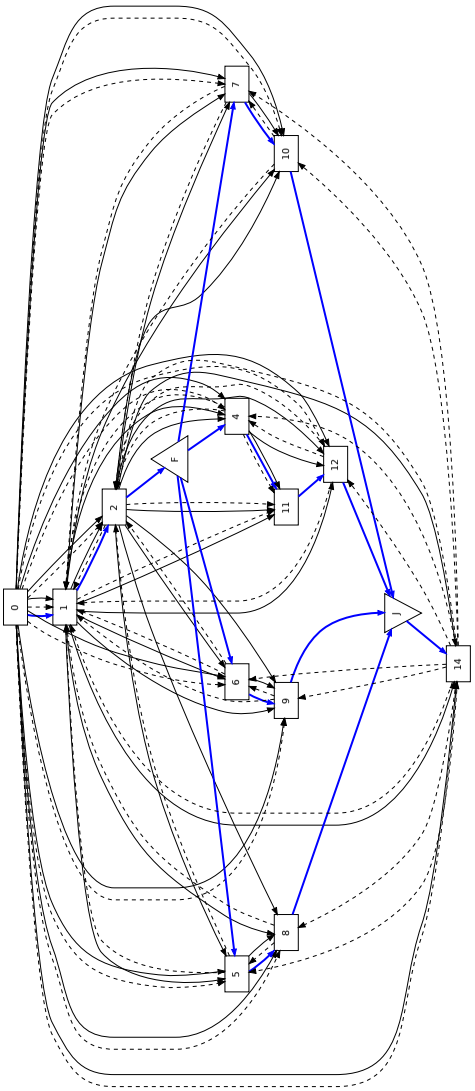


Figure A.13: Generated DPN for Ray Tracer

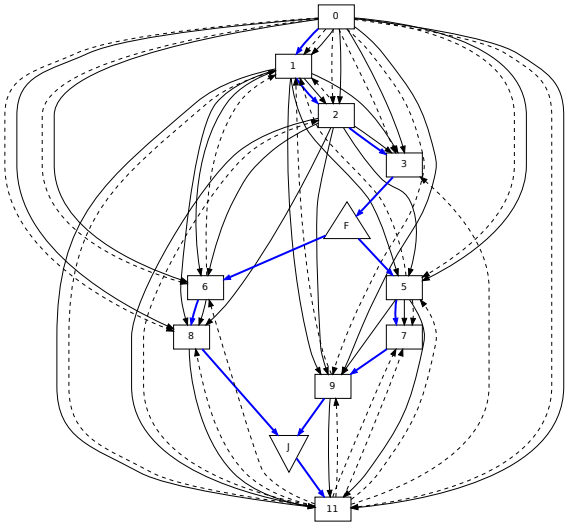


Figure A.14: Generated DPN for Landscape (H=200) and Landscape (H=800)

Appendix B

Curriculum Vitae

Persönliche Daten[1]

Name: Daniel Baudisch

Hochschulstudium

10/2001 – 05/2008	Studium im Fach Technoinformatik an der TU Kaiserslautern
11/2007 – 05/2008	Diplomarbeit mit dem Titel „Synthesis for VLIW Architectures“
05/2008	Abschluss als Diplom Technoinformatiker
seit 05/2008	wissenschaftlicher Mitarbeiter an der TU Kaiserlautern in der AG Eingebettete Systeme im Fachbereich Informatik

[1] für Online-Version gekürzt