Analyzable Dataflow Executions With Adaptive Redundancy

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Angewandte Informatik
der Universität Augsburg

eingereicht von
Christoph Kühbacher, M. Sc.

Analyzable Dataflow Executions With Adaptive Redundancy

Christoph Kühbacher

Erstgutachter:	Prof. Dr. Sebastian Altmeyer
Zweitgutachter:	Prof. Dr. Theo Ungerer
Tag der mündlichen Prüfung:	18. August 2022

Zusammenfassung

Aufgrund steigender Anforderungen an die Leistungsfähigkeit von eingebetteten Systemen finden Mehrkernprozessoren mittlerweile auch in eingebetteten Systemen Verwendung. Autos sind ein Beispiel für eingebettete Systeme, in denen die Verbreitung von Mehrkernprozessoren kontinuierlich zunimmt. Ein Hauptgrund ist, dass es dadurch möglich wird, mehrere Applikationen, für die ursprünglich mehrere Electronic Control Units (ECUs) notwendig waren, auf ein und demselben Chip auszuführen und dadurch die Anzahl der ECUs im Gesamtsystem zu verringern. Der De-facto-Standard AUTOSAR (AUTomotive Open System ARchitecture) wurde jedoch ursprünglich nur im Hinblick auf Einkernprozessoren entworfen und, obwohl der Softwarestack um grundlegende Unterstützung für Mehrkernprozessoren erweitert wurde, sind komplexere Architekturen nicht damit kompatibel.

Die Anforderungen der Softwarekomponenten von modernen Autos sind vielfältig. Einerseits gibt es hochgradig sicherheitskritische Tasks, die beispielsweise die Airbags, das Antiblockiersystem, die Fahrdynamikregelung oder den Notbremsassistenten steuern und andererseits Tasks, die keinerlei sicherheitskritische Anforderungen aufweisen, wie zum Beispiel Tasks zur Steuerung des Infotainment-Systems. Neue Trends wie autonomes Fahren führen zu weiteren anspruchsvollen Tasks, die sowohl hohe Leistungs- als auch Sicherheitsanforderungen aufweisen.

Da die Komplexität eingebetteter Anwendungen, beispielsweise im Automobilbereich, stetig zunimmt, sind neue Ansätze erforderlich. Für komplexe, datenintensive Aufgaben werden in der Regel Cluster-Computing-Frameworks eingesetzt. In dieser Arbeit werden Konzepte solcher Frameworks auf den Bereich der eingebetteten Systeme übertragen. Dazu beschreibt die Arbeit eine Laufzeitumgebung (RTE) für eingebettete Mehrkernarchitekturen. Die RTE folgt einem Datenfluss-Ausführungsmodell, das auf gerichteten azyklischen Graphen basiert. Graphen können in Abschnitte eingeteilt werden, für welche separat mehrere unterschiedlich redundante Schedules mit Hilfe einer Scheduling-Heuristik berechnet werden. Dieser Ansatz erlaubt es, die Redundanz von Teilen der Anwendung zur Laufzeit zu verändern. Alternativ unterstützt die RTE auch Scheduling zur Laufzeit. Zur Erzeugung von Graphen stellt die RTE ein Programmiermodell bereit, welches sich an etablierten Frameworks, insbesondere Apache Spark, orientiert. Damit wurden drei Beispielanwendungen implementiert, die auf gängigen Algorithmen basieren. Konkret handelt es sich um Cannon's Algorithmus, den Cooley-Tukey-Algorithmus und bitonisches Sortieren. Um die Leistungsfähigkeit der RTE zu ermitteln, wurden diese drei Anwendungen mehrfach mit verschiedenen Konfigurationen auf zwei Hardware-Architekturen ausgeführt. Die Ergebnisse zeigen, dass die RTE in ihrer Leistungsfähigkeit mit etablierten Systemen vergleichbar ist und die Laufzeit bei einer sinnvollen Graphaufteilung im besten Fall nur geringfügig beeinflusst wird.

Abstract

Increasing performance requirements in the embedded systems domain have encouraged a drift from singlecore to multicore processors, and thus multicore processors are widely used in embedded systems today. Cars are an example for complex embedded systems in which the use of multicore processors is continuously increasing. A major reason for this is to consolidate different software components on one chip and thus reduce the number of electronic control units. However, the de facto standard in the automotive industry, AUTOSAR (AUTomotive Open System ARchitecture), was originally designed for singlecore processors. Although basic support for multicore processors was added, more complex architectures are currently not compatible with the software stack.

Regarding the software components running on the ECUS of modern cars, requirements are diverse. On the one hand, there are safety-critical tasks, like the airbag control, anti-lock braking system, electronic stability control and emergency brake assist, and on the other hand, tasks which do not have any safety-related requirements at all, for example tasks controlling the infotainment system. Trends like autonomous driving lead to even more demanding tasks in the system since such tasks are both safety-critical and data-intensive.

As embedded applications, like those in the automotive domain, become more complex, new approaches are necessary. Data-intensive tasks are usually tackled with large-scale computing frameworks. In this thesis, some major concepts of such frameworks are transferred to the high-performance embedded systems domain. For this purpose, the thesis describes a runtime environment (RTE) that is suitable for different kinds of multi- and manycore hardware architectures. The RTE follows a dataflow execution model based on directed acyclic graphs (DAGs). Graphs are divided into sections which are scheduled separately. For each section, the RTE uses a DAG scheduling heuristic to compute multiple schedules covering different redundancy configurations. This allows the RTE to dynamically change the redundancy of parts of the graph at runtime despite the use of fixed schedules. Alternatively, the RTE also provides an online scheduler. To specify suitable graphs, the RTE also provides a programming model which shares similarities with common large-scale computing frameworks, for example Apache Spark. Using this programming model, three common distributed algorithms, namely Cannon's algorithm, the Cooley-Tukey algorithm and bitonic sort, were implemented. With these three programs, the performance of the RTE was evaluated for a variety of configurations on two different hardware architectures. The results show that the proposed RTE is able to reach the performance of established parallel computation frameworks and that for suitable graphs with reasonable sectionings the negative influence on the runtime is either small or non-existent.

Danksagung

An dieser Stelle möchte ich mich bei einigen Personen bedanken, die bei der Entstehung dieser Dissertation einen maßgeblichen Einfluss hatten. Zunächst möchte ich mich für die hervorragende Betreuung beim Verfassen der Arbeit bedanken, sowohl bei Prof. Theo Ungerer, als auch bei Prof. Sebastian Altmeyer, der die Betreuung nach Prof. Ungerers Pensionierung übernommen hat. Zu jeder Zeit konnte ich mich darauf verlassen, Unterstützung und konstruktive Ratschläge zu erhalten. Ich möchte mich auch dafür bedanken, dass sie mir die Chance gegeben haben, Teil eines großartigen Teams zu sein und bei zwei wissenschaftlichen Projekten mitzuwirken. Ebenfalls bedanken möchte ich mich bei Prof. Dr. Jörg Hähner, der sich dazu bereit erklärt hat, die Arbeit zu begutachten.

Meinen Kollegen am Lehrstuhl danke ich für die zahlreichen wissenschaftlichen und nicht-wissenschaftlichen Diskussionen und Gespräche. Ein so angenehmes Arbeitsklima findet man selten. Bei Christian Mellwig und Fabian Kempf, mit denen ich im Rahmen eines Projekts eng zusammengearbeitet habe, bedanke ich mich für die zahlreichen fachlichen Diskussionen, die wesentlich zu dieser Arbeit beigetragen haben. Dr. Sebastian Weis danke ich dafür, dass er mir in der Anfangszeit meiner Arbeit am Lehrstuhl immer mit Rat und Tat zur Seite stand.

Bei meiner Familie, vor allem meiner Mutter Ulrike und meinem Bruder Johannes, bedanke ich mich für die bedingungslose Unterstützung in allen Belangen, ohne die ich diese Arbeit nicht hätte verfassen können.

Christoph Kühbacher
Augsburg im August 2022

Contents

1

Introduction

The trend towards increasing performance requirements is not limited to the domain of high performance computing any more and multicore processors are widely used in complex embedded systems today. In a survey from 2020 about industry practice in real-time systems, more than 80% of the respondents stated that the system they are working on contains at least one multicore processor (2–16 cores) and a similar proportion of participants (roughly 80%) indicated their use in real-time embedded projects by 2021 [Ake+20]. According to the survey, the proportion of respondents working on systems which contain manycore processors (16+ cores) is significantly lower (about 15%). However, interest appears to be greater, with 33% of respondents indicating that manycore processors will become established by 2021 and another 14% expecting this by 2029.

1.1 Motivation

About 40% of the participants in the survey work on automotive-related systems. Cars are complex embedded systems consisting of various electronic control units (ECUs) connected via different kinds of buses. A major reason for the increasing use of multicore processors in the automotive domain is to consolidate different software components on one chip and thus reduce the number of ECUs. Tesla is a pioneer

in this field. Back in 2012, the Model S already utilized only 3–4 ECUs [VBP19]. Since the de facto standard in the automotive industry, AUTOSAR (AUTomotive Open System ARchitecture), was originally designed for singlecore processors, an extension of the software stack was required. First support for multicore processors was added in version 4.0 of AUTOSAR [Bec+15]. However, the AUTOSAR multicore extensions were only designed for a simple hardware model in which cores access a shared memory and other peripherals over a shared bus [Bec+15]. To meet the requirements of future automotive applications, there is great interest in extending the existing multicore capabilities and adding support for additional hardware architectures, for example distributed manycores [GC18; Bec+15; UO17].

The software components running on the ECUS of modern cars are diverse and thus have different requirements. First, there are safety-critical tasks like the airbag control, anti-lock braking system, electronic stability control and emergency brake assist. Such tasks may not be computationally complex but have strong requirements regarding reliability and timing predictability. The complete opposite would be tasks controlling the car's infotainment system which gets more performance demanding as technology evolves but is not safety-critical at all. Between these two kinds of tasks, lies broad spectrum of tasks with different performance and safety requirements. The ignition timing control, for example, clearly has real-time requirements, but occasional close deadline misses might be acceptable.

With the continuous expansion of assistance systems and the long-term goal of fully autonomous driving, safety-critical software components are becoming more and more complex. A common computational task in this area is the execution of convolutional neural networks (CNNs) to detect pedestrians, lane lines, street signs and other vehicles from camera images [Tal+20; Luc+16]. CNNs are directed graphs consisting of different types of operation nodes, for example convolution and pooling nodes. Multi- and manycore processors are well-suited for the execution of CNNs since there is enough parallelism resulting from the data parallelism in operation nodes and the possibility to execute nodes concurrently.

Considering the requirements of complex embedded applications, like automotive software components, and modern computational tasks, for example CNNs, this thesis describes a runtime environment (RTE) that is suitable for different kinds of multi- and manycore hardware architectures. A major aspect in the development of the RTE was the transfer of concepts from large-scale computing frameworks, which are commonly used for machine learning and other data-intensive applications, to the high-performance embedded systems domain. Like many of these frameworks, the RTE executes programs in form of directed acyclic graphs (DAGs). Suitable graphs can be specified either directly by defining the nodes and edges or via a functional-style programming model. The benefit of this programming model is that large graphs with enough inherent parallelism for manycore processors can

be created with only a few function calls. In general, the programming model is similar to the programming style of existing big data frameworks but provides a smaller set of functions due to its focus on embedded systems. The RTE's execution model follows a coarse-grain dataflow style, i.e. data passed between the nodes of a graph is usually a larger data structure or collection. Both the type of data passed between nodes and the functions applied to the data are user-defined. Thus, the RTE offers enough expressiveness for today's as well as future embedded systems. An important topic in the embedded systems domain is fault tolerance. Some large-scale computing frameworks which inspired the RTE also contain a fault tolerance mechanism. However, these mechanisms mainly focus on the failure of a node in the compute cluster and the recovery of data. Safety-critical embedded systems have additional requirements, for example the correctness of results, which requires redundant computation. To meet the requirements of modern safety-critical systems, the proposed RTE implements the concept of adaptive redundancy, which allows it is to change the redundancy of parts of the application dynamically at runtime.

1.2 Contribution

The main contribution of this thesis is the transfer of concepts which are successfully used in big data frameworks to the domain of high-performance embedded systems. For this purpose, the thesis comprehensively describes a runtime environment, from the high-level programming model down to the implementation on different hardware architectures. In detail, the individual contributions are:

- A programming model similar to established large-scale computing frameworks that allows to conveniently specify dataflow graphs with enough inherent parallelism to utilize the capabilities of multi- and manycore processors
- A coarse-grain dataflow execution model based on directed acyclic graphs which on the one hand allows to change the redundancy of different parts of an application at runtime and on the other hand to analyze dataflow executions with reasonable effort
- Details of an RTE implementation on two hardware architectures, more precisely a shared-memory architecture and a network-on-chip-based architecture
- Extended versions of an existing DAG scheduling heuristic with support for different redundancy configurations
- Analyzability considerations for dataflow executions in the RTE
- An evaluation of the proposed programming and execution model with three common applications on two existing hardware architectures

1.3 Overview

This thesis consists of eleven chapters. Following this chapter which gives an introduction to the topic, Chapter 2 provides background information on the four concepts the RTE is based on, namely dataflow, functional programming, DAG scheduling and fault tolerance. Chapter 3 shows how the four concepts are combined in the RTE as it gives a high-level overview of the RTE's internal structure. The RTE consists of four parts, a graph construction part based on a functional programming model, an import/export part, a scheduling part and a graph execution part. Subsequent chapters go into the details of the different parts of the RTE. In Chapter 4, the data structures and operations of the functional programming model are introduced. It is important to note that this chapter only describes the programming model from a user's perspective, i.e. the provided operations are considered as functions that are evaluated in a lazy fashion. The chapter also contains three exemplary algorithms expressed in the programming model. How the operations actually construct dataflow graphs is explained in Chapter 5. The chapter also provides more details on the structure of dataflow graphs and shows dataflow graphs constructed by the three example programs from the previous chapter. In Chapter 6, dataflow executions are discussed in more details. This chapter describes reference RTE implementations for two different hardware architectures, a shared-memory system and a network-on-chip-based architecture. Chapter 7 describes extended versions of a common DAG scheduling heuristic and an online scheduling procedure that support the hardware architectures and redundancy configurations covered in Chapter 6. Some considerations about the analyzability of dataflow executions with offline schedules are made in Chapter 8. The chapter also briefly covers different redundancy configurations and transient errors. Details about the performance of dataflow executions on two different platforms are given in Chapter 9. Similar to previous chapters, Chapter 9 also covers the different redundancy configurations. Furthermore, the chapter provides measurements regarding graph construction, scheduling and adaptive redundancy. Chapter 10 describes other approaches which are related to the proposed RTE either in terms of their programming model, dataflow or fault-tolerance. Furthermore, the similarities and differences between the RTE and the related work are discussed. Lastly, Chapter 11 summarizes the results of this thesis and proposes future research opportunities.

2

Background

This chapter provides information on different topics that are important for the proposed programming model and runtime environment. Section 2.1 introduces the concept of dataflow and shows different types of dataflow models, ranging from static dataflow models over dynamic dataflow models to hybrid dataflow/von-Neumann approaches. After that, Section 2.2 describes the functional programming paradigm and some common characteristics, like high order functions and parametric polymorphism. Section 2.3 provides information on scheduling heuristics for directed acyclic graphs (DAGs). These heuristics are applied offline, i.e. before the graph is executed. A common approach to tackle the problem of DAG scheduling at runtime is described in Section 2.4. Lastly, an overview of different concepts related to fault tolerance is given in Section 2.5.

2.1 Dataflow

Today's computer architectures are mostly based on the von-Neumann model. A computer following the von-Neumann model executes programs according to the given control flow. This means that instructions are executed one after the other based on the given instruction order. The von-Neumann model makes it easy for programmers to write sequential programs, but it is also possible to write

parallel applications in this computing model, for example via an abstraction like operating system processes or threads. However, writing parallel programs in the von-Neumann model is much more difficult than writing sequential programs. The reason for this is that programmers have to handle the synchronization of all sequential executions manually. What makes the situation worse is that bugs in parallel von-Neumann programs are often hard to detect and fix [Lu+08]. Examples of such bugs are deadlocks, livelocks, starvation and race conditions. Because of their non-deterministic nature, these bugs may randomly lead to unintended behavior at some point during program execution or may not appear at all.

The dataflow model is an alternative approach to the classic von-Neumann computing model. In contrast to the von-Neumann model's control-flow-based execution, instructions in the dataflow computing model are triggered by the availability of their operands. This means that instructions can be executed as soon as their operands and the required hardware resources are available. As a result, dataflow-based computers are able to fully exploit the inherent parallelism of programs. In this regard, the organization of memory also plays an important role. Since there are no explicit load and store instructions in the dataflow computing model, there is no way to freely access memory locations like in the von-Neumann model.

2.1.1 Emergence of Dataflow Architectures

The first proposal describing the dataflow paradigm was made by Karp and Miller [KM66]. They described a model in which a parallel program is represented by a finite directed graph. Graph nodes represent operations in the parallel program, while edges result from their data dependencies. Edges in the graph act as queues which transfer data from one operation to another. The authors also showed some favorable characteristics of their model. Probably the most important property they proved is *determinism*. This property says that all proper executions of the computation graph lead to the same results. Neither the timing of a graph node execution nor its actual runtime influence the result. Further, the property holds for each graph node as well as the computation graph as a whole. Determinism is not limited to Karp and Miller's model. It is rather a property of the dataflow paradigm in general. Therefore, later upcoming dataflow models are also deterministic. However, there also exist models that support indeterminism when it is explicitly introduced, for example Kosinski's dataflow language DFPL [Kos73]. Other well-known early dataflow models were proposed by Kahn [Kah74] and Dennis [Den74].

What all the dataflow models have in common is that parallel programs are expressed through directed graphs, known as *dataflow graphs*. Although there are subtle differences between the graphs of different models, their core is the same.

Like in Karp and Miller's original model, graph nodes, also called *dataflow actors*, represent instructions, while edges (or *arcs*) represent data dependencies. Nodes in the graph usually do not have an internal state. Data that is transferred from one node to another according to the graph edges is often called a *token*. If all required tokens of an instruction node are available, the node is called *enabled*. An enabled node may *fire*, i.e. consume some input tokens, execute the instruction and create appropriate output tokens. Although dataflow graphs can be created by hand, this procedure is not feasible for larger graphs. Hence, dataflow graphs are often constructed from higher-level code. The construction itself may be done at compile time, decode time or execution time, based on the dataflow system [Yaz+14].

After some dataflow models and languages were proposed, the dataflow principle was also used to design dataflow architectures. These processors can directly execute dataflow programs and therefore differ from typical von-Neumann processors. The first concrete architecture was proposed by Dennis and Misunas [DM74]. Its structure is shown in Figure 2.1 in a simplified form. Dataflow graphs are stored in the processor's memory, which consists of various *instruction cells*. Each instruction cell corresponds to a node in the dataflow graph. Since all instructions in Dennis's model have a maximum of two operands, instruction cells contain three registers, one for the instruction and the result addresses and two for the operands. A cell is enabled when its input operands are available. Enabled cells are fetched by the *arbitration network* and sent to an *operation unit* in form of an *operation packet*. When an operation unit has finished its current instruction, the result is sent as a data packet over the *distribution network*, which sends the data to the correct registers inside the memory. Later proposed dataflow architectures usually have a more com-

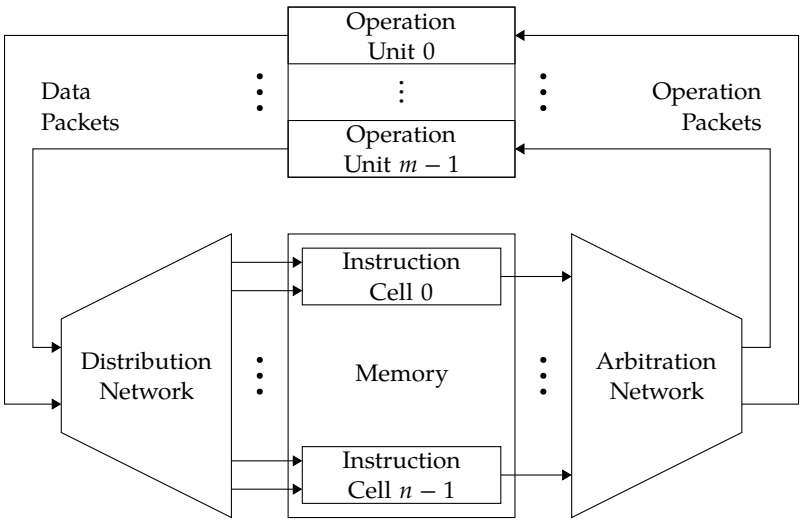


Figure 2.1: Simplified dataflow processor [DM74]

plicated structure than the described simplified architecture. Well known examples for dataflow architectures include the Manchester dataflow computer [GKW85], the MIT tagged-token dataflow architecture [AN90] and the Hughes dataflow multiprocessor [VF85]. What they have in common is the circular organization of the pipeline, which is also visible in Figure 2.1. This structure strongly encourages parallel instruction execution. However, the circular pipeline is the main reason for the poor sequential performance of many dataflow architectures [ABU91; RŠU00]. An instruction can only be executed when all directly preceding instructions in the graph went through the whole pipeline.

2.1.2 Static Dataflow

The original dataflow architecture proposed by Dennis belongs to the category of static dataflow approaches. In such models, graph edges may only contain a single token at a time. Therefore, static models follow the so-called *single-token-per-arc* approach [ŠRU98]. Because of this restriction, the firing of instruction nodes must not only consider incoming edges but also the outgoing ones. In the static model, outgoing edges of a node must not contain data because otherwise the node cannot be executed. As a result, tokens produced by the same actor are always consumed in order of their creation.

The downside of the single-token-per-arc approach is that it prohibits parallel executions of the same instruction node because this would require multiple tokens on the input arcs. This affects, for example, loops in the dataflow graph. Loop iterations cannot be executed completely in parallel even if there is no data dependency between them. The only possibility to introduce some parallelism in loop iterations is via pipelining [ŠRU98]. But besides loop iterations, parallelism in subroutine calls is also limited [Yaz+14]. If subroutines are modeled as independent dataflow graphs, it is not possible to execute the same subroutine multiple times in parallel for the same reason as with loop iterations.

Examples for architectures following a static dataflow model are some of the dataflow architectures based on the work of Dennis [DM74], the DDM1 dataflow processor [Dav78] and the Hughes dataflow multiprocessor [VF85].

2.1.3 Dynamic Dataflow

The described problem of static dataflow architectures that the same instruction node cannot be executed in parallel lead to the development of dynamic dataflow models. In dynamic dataflow models, multiple tokens in the same graph edge can co-exist. But this fact alone is not enough to support parallel instruction node executions. If loop iterations, for example, are executed in parallel, tokens from

different iterations could get mixed up since there is no way to check to which iteration a token belongs. The same problem exists for parallel calls of a subroutine. The solution of dynamic dataflow models is to assign a *tag* to each token [ŠRU98]. In case of parallel loop iterations/subroutine calls, tokens belonging to different iterations/calls have different tags. The presence of tags leads to a different firing rule compared to static dataflow models. An instruction node may fire if for all incoming edges tokens with identical tags are available. Examples for dynamic dataflow architectures are the Manchester dataflow computer [GKW85], the MIT tagged-token dataflow architecture [AN90] and the Monsoon architecture [PC90].

By using tags, dynamic dataflow architectures can exploit more inherent parallelism than static dataflow architectures. However, tags introduce another difficulty. The problem of finding tokens with identical tags, also called *token matching*, is not trivial, and it is relevant for all dyadic (two-operand) instructions. Since token matching is necessary for a lot of the instruction executions, it should be as fast as the execution itself to not become a bottleneck. Dynamic dataflow architectures typically use an associative (or pseudo-associative) memory to support parallel tag comparisons [GKW85]. Unfortunately this type of memory implementation is quite expensive.

Another notable approach to tackle the token matching problem is taken in the Monsoon architecture [PC90]. This approach is called *explicit token store*. The basic idea is to relocate parts of the matching logic from the hardware to the compiler. Token memory is allocated dynamically in blocks, called *activation frames*. An activation frame is allocated each time a procedure is invoked. How the memory in an allocation frame is used is determined at compile time. This approach, however, requires that the storage requirement of a procedure is known at compile time.

2.1.4 Hybrid Dataflow/von-Neumann Approaches

The development of hybrid models had different reasons. One reason is to compensate the difficulties in dataflow architectures, such as the poor sequential performance or the expensive token matching in dynamic dataflow models. Another reason is the desire to create an architecture which is able to execute both dataflow and von-Neumann programs. The variety of hybrid dataflow/von-Neumann models is big. Therefore, this section briefly describes a selection of approaches, based on the taxonomy of Robič et al. [RŠU00].

Threaded Dataflow

One approach is the *threaded dataflow* model, which is relatively similar to pure dataflow models. In the threaded dataflow model, parts of dataflow graphs with a

low degree of parallelism are identified and executed sequentially, i.e. as threads. An advantage of this approach is that the expensive token matching is only necessary for the first instruction of a thread. Further, in a sequential stream of instructions it is possible to keep data in registers instead of writing it back to memory as tokens. This reduces the number of tokens in the system and improves the sequential performance. Especially dyadic instructions profit from the use of registers because these instructions would otherwise require token matching. There are two different kinds of threaded dataflow executions, namely *direct token recycling* and *consecutive execution*. The first technique allows only one instruction per thread at a time. To fully utilize all pipeline stages multiple threads must be executed in an interleaving fashion. In contrast to the former, architectures following the consecutive execution approach execute threads without interleaving. As a result, the token matching in such architectures has to be delayed when a sequential thread is running. [RŠU00; ŠRU98]

Coarse-Grain Dataflow

Another way to combine characteristics of von-Neumann and dataflow models is the concept of *coarse-grain dataflow*. The idea of coarse-grain dataflow is to define actors as sequences of instructions instead of single instructions. These sequential actors can be executed in a von-Neumann manner. An advantage in comparison to the threaded dataflow approach is that von-Neumann processors can be used for actor executions. Therefore, coarse-grain dataflow architectures can also benefit from the development and improvements of standard, control-flow-based processors. A FIFO-buffer in the dataflow pipeline between the token matching stage and the execution stage can also help to improve the overall performance of the system. [RŠU00; ŠRU98]

Complex Instructions

The last approach highlighted in this section is the use of complex machine instructions in dataflow architectures. Vector instructions, for example, allow the dataflow system to process elements of a data structure in blocks rather than individually. This is helpful to reduce the number of nested loops. Further, by splitting a complex instruction into independent sub-operations, it is possible to take advantage of parallelism at the sub-instruction level. The negative effect of the difference in the execution times of standard and complex instructions can be reduced by buffers. It is important to note that, in contrast to conventional dataflow architectures, tokens in this approach do not contain the actual data but only boolean values. The actual data is only accessed in the execution stage of the pipeline. [RŠU00; ŠRU98]

Examples for Hybrid Architectures

Many hybrid architectures were proposed. Notable examples of threaded dataflow architectures are the Monsoon architecture [PT91] and the EM-4 architecture [KSY92]. Monsoon uses the direct token recycling technique, while EM-4 belongs to those architectures following the consecutive execution approach. StarT (also written as *T) [NPA92] and the Threaded Abstract Machine (TAM) [Cul+91] are examples for the coarse-grain dataflow approach. Lastly, examples for architectures that use complex instructions are the Augsburg Structure-Oriented Architecture (ASTOR) [Ung88] and the Stollman Dataflow Machine [GRS89].

2.1.5 Dataflow in Today's Hardware and Software

The advantages of dataflow lead to the use of dataflow concepts in hardware and software. An example for dataflow in hardware is out-of-order superscalar processing, a technique used in nearly all modern high performance processors. Notable software systems utilizing dataflow concepts are big data frameworks. This section briefly describes how modern processors and big data frameworks incorporate the dataflow principle.

Out-of-Order Superscalar Processing

This technique allows a processor to execute multiple instructions per clock cycle by exploiting parallelism in a sequential program at the instruction level. Figure 2.2 gives an abstract overview of the technique. The prerequisite for a parallel execution at the instruction level is that the processor is able to fetch multiple instructions per cycle. Fetched instruction are placed in an *instruction window*. Instructions inside the window are dynamically assigned to a suitable hardware unit as soon as all operands are present. This approach clearly follows the dataflow principle. As shown in Figure 2.2, instructions in the instruction window and their data dependencies form a small dataflow graph. Since the executed program is sequential, finished instructions must be inserted into a *reorder buffer*. The reorder buffer makes sure that instruction results are committed in such a way that the overall result of the parallel execution is equal to the result of the sequential execution demanded by the program. [RŠU00; SS95]

In the described approach, two problems which limit the parallel execution remain. First, instructions may use the same registers, but there is no *true data dependency* between them. This is the case, when two instructions write their result to the same register (*write-after-write*) or when an instruction writes its result to a register which was read by a prior instruction (*write-after-read*). Such artificial

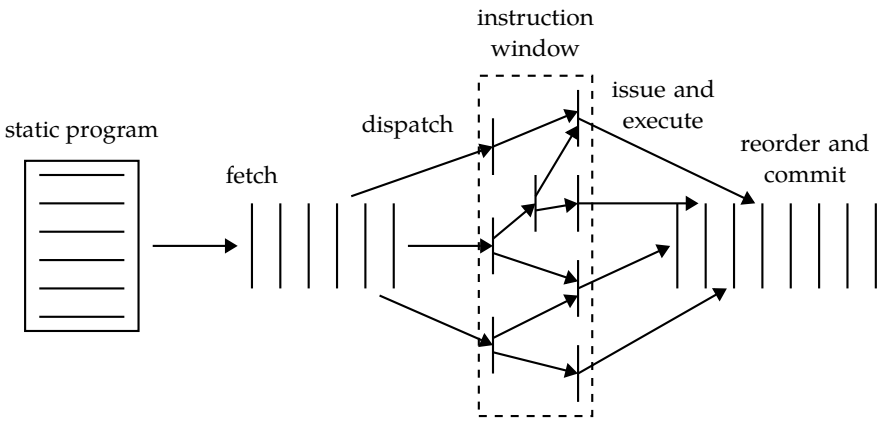


Figure 2.2: Conceptual figure of superscalar execution [SS95]

dependencies can be temporarily removed since they are only important for the reorder buffer and committing stage, not the executing hardware units. The removal of artificial dependencies is done by a hardware unit which renames registers before an instruction is placed in the instruction window. [SS95]

The second problem results from branches. Branches limit the number of instructions that can be fetched so that there are not enough instructions in the window to efficiently exploit parallelism. Out-of-order superscalar processing therefore relies on speculative instruction fetching via branch prediction [SS95]. Today's branch prediction is very accurate. AMD, for example, claims that branch prediction in their Zen 2 architecture exhibits a ~30% lower branch mispredict rate target than in the predecessor architecture by using a two-level prediction hierarchy [Wik]. The first-level predictor in this hierarchy is a fast perceptron-based predictor [JL01], while the second-level predictor is a slower but more accurate TAGE predictor [SM06]. With branch prediction accuracy being above 95% since the 1990s [Cha+94] and still being improved to this day, mispredicts keep getting a smaller and smaller issue for the out-of-order execution despite the costly re-rolling in the pipeline.

Big Data Frameworks

Besides out-of-order superscalar processors, big data frameworks also take advantage of dataflow concepts. These frameworks make it possible to process huge amounts of data on large compute clusters. One advantage of using such frameworks is that users do not have to consider synchronization and work distribution since it is already part of the framework. Another benefit is that big data frameworks like Apache Spark [Zah+12] provide ways to execute self-defined functions on the data. This is often easier and more flexible than, for example, standard SQL queries.

Dataflow in big data frameworks belongs to the category of coarse-grain dataflow models. Data packets traversing through the graph do not contain single values but instead larger chunks of structured or unstructured data. Analogously, actors in the graph usually consist of complex operations executing user-defined functions on their input data instead of single instructions like in traditional dataflow models.

Big data frameworks follow two different approaches, namely batch processing and stream processing. Batch systems process data in chunks, one after the other, while stream-based systems operate on continuously incoming data. Examples for batch systems are Google's original MapReduce [DG04] as well as Apache Hadoop [Shv+10] (an open-source implementation of the MapReduce model), Pig [Gat+09] and Spark [Zah+12]. An example for a stream-based framework is Apache Flink, which emerged from the Stratosphere project [Ale+14].

In the MapReduce model, each computation consists of a map phase which produces intermediate data followed by a reduce step. To realize more complex computations multiple MapReduce steps have to be executed in a row. This leads to reduced performance in the early MapReduce-based systems, for example Hadoop and Pig (which translates programs into MapReduce plans by default) [Sah+15]. In order to increase the performance of complex computations, later frameworks like Apache Spark or Pig with Tez [Sah+15] as back-end follow an approach closer to traditional dataflow models since they translate programs into directed acyclic graphs.

2.2 Functional Programming

Functional programming is a programming paradigm which does not distinguish as strictly between functions and data as the imperative programming paradigm does. Functions can be stored, returned or passed as arguments just like ordinary data. An ability functions do not have is to change the program's global state. In this regard, functions in this paradigm behave like mathematical functions. In general, functional programming shares many concepts with formal systems, especially the lambda calculus [Chu41], which is often considered as its predecessor. The next sections highlight some important concepts in the functional programming paradigm and functional programming languages.

2.2.1 Characteristics of Functions

As the name suggests, functions are the central concept in functional programming. Functional programs consist solely of function applications and compositions. Func-

tions are closer to mathematical functions and show different characteristics than functions in imperative languages. Some of their characteristics are described below.

High-Order-Functions

Because of their important role in functional programming, functions are treated as *first-class values*. This means that functions can be passed to other functions as arguments or can be returned by functions like ordinary data. Functions with parameters or a return value of function type are commonly known as *high-order functions*. [Hud89]

Lambda Expressions

Since functions are treated like values in this programming paradigm, they can be created at runtime and stored in variables or as members of data structures. Therefore, it is not necessary for functions to have a name (although each function has an identity). Such functions are called *anonymous functions* or, with respect to the lambda calculus, *lambda expressions* [SK95, pp. 145–146]. A common use case for lambda expressions is to pass them to high-order functions as arguments, analogous to literals of plain data types.

Currying

In the context of anonymous functions and high-order functions, the concept of *currying* plays an important role. Currying is the process of transforming one function with multiple parameters into a sequence of functions with only one argument. Passing an argument to a curried function returns another curried function (except for the last function in the sequence) which corresponds to an uncurried function with one parameter less. This is sometimes called *partial application*. [SK95, pp. 143–144]

A function returned by partial application of another function could be considered as a specialized version of the latter. Such a specialized function can then be stored in a variable or passed as an anonymous function. Being able to easily create more specialized versions of functions helps to reduce the amount of duplicate code.

Referential Transparency

Another property of functions in this paradigm is that they are free of side effects, i.e. they do not read or modify the global state of the program. All information a function requires must be passed through its parameters. Furthermore, functions are deterministic, which means they return the same value each time they are called with the same arguments. As a result, expressions can be freely replaced by their

values. This is why functional programs are often called *referentially transparent* [Hud89]. Although referential transparency may seem like a restriction from a programmer's view, it has some advantages. First, the lack of side effects leads to a better composability and reusability of code [Hug90]. Second, functions without side effects are usually easier to test, since there is no global state to consider [CH00]. However, complex high-order functions also have the potential to be difficult to test despite being free of side effects. A downside of the functional approach is that I/O, for example, is more difficult to implement in a purely functional fashion [PW93].

2.2.2 Type System

Types in programming languages are used to divide values into different sets. By doing so, it is possible to check whether an argument is suitable for a specific function. Functional programming languages can be statically or dynamically typed. In this regard, they are no different from imperative languages and share the same advantages and disadvantages. In statically typed languages, types can be checked at compile time so that errors are discovered before the program is executed, while dynamic type systems provide more flexibility. Such systems allow, for instance, that a variable is used twice or more with values of different types. Examples for statically typed functional languages are ML [Gor+78], Miranda [Tur85] and Haskell [Hud+07], while Common Lisp [Ste82] and Scheme [Spe+09] belong to the class of dynamically typed languages. Statically typed languages are always *strongly typed*, which means that all expressions are type-consistent. For dynamically typed languages this may not necessarily be the case [CW85]. In the following, statically typed functional programming is discussed in more detail. Type systems of such languages are often based on the Hindley-Milner type system [Hin69; Mil78], a type system for the lambda calculus. The Hindley-Milner type system provides *type inference* and *parametric polymorphism*. These two concept are briefly described in the following sections.

Type Inference

Type inference is the process of determining the type of variables, parameters and expressions without user-specified type annotations. The Hindley-Milner type system provides *complete type inference*, which means that it is guaranteed to infer types, whenever possible, for entirely unannotated programs [PT00]. To achieve this, the system first sets up equations describing type constraints which are then solved in a second step [Mil78].

Complete type inference with the Hindley-Milner type system works for the lambda calculus and some programming languages based on it. For more complex

formal systems or programming languages, however, this approach might not be feasible. Other approaches therefore only use local information (with regard to the syntax tree) to infer types, but they require more manual type annotations in return. [PT00]

The following example from the work of Cardelli and Wegner [CW85] illustrates type inference applied to a function `fun` like it is done in the ML language.

```
fun(x) x+1
```

Type inference works bottom up, i.e. it starts with small expressions, like `x` and `1`. The type of variable `x` is initially not known. Therefore, a new type variable *a*, which represents the type of `x`, is introduced. `1` is easily identified as an integer literal and, therefore, the binary operation as an integer addition. Accordingly, the value of type variable *a* must also be integer. Altogether, `fun` is a function with an integer parameter that also returns an integer.

Parametric Polymorphism

Parametric polymorphism describes the concept of types containing type variables in function declarations. This allows programmers to define functions accepting values of different type. A restriction of parametric polymorphism is that applications of according functions must not depend on actual types, i.e. the actual type instantiations are not important for the functionality [Str00]. (The analogous concept where concrete types are important for the function is called *ad hoc polymorphism* [Str00]. Today, however, it is often referred to as *function overloading*.)

A typical use of parametric polymorphism is in functions modifying data structures, for example lists or queues. Insert functions, for instance, work identically for all types of data values. The only constraint which has to be checked is that the type of the inserted value corresponds to the type of the data values in the list/queue. Accordingly, the type of an insert function would be $\texttt{list}\langle\texttt{a}\rangle \times \texttt{a} \rightarrow \texttt{list}\langle\texttt{a}\rangle$, where `a` is a type variable.

2.2.3 Functional Programming and Dataflow

The functional programming paradigm has some characteristics that makes it well suited for use in dataflow computing. One of these characteristics is that functional programming follows a declarative approach. Programs in this paradigm are expressed through function applications and compositions, not through a sequence of operations. Another property which is beneficial for dataflow computing is referential transparency. Since functions are free of side effects, data is always passed from one function to another. As a result, functions in functional programs do not

depend on the time they are executed. Further, referential transparency ensures that the order of function executions is flexible, with the only restriction being data dependencies between functions. When looking at the properties of functions, it becomes clear that functions in the functional programming paradigm are similar to dataflow actors. Therefore, functional programming languages work well for transformations between high level programs and dataflow graphs. An example for such a transformation from the domain of real-time image processing is proposed by Sérot et al. [SQZ93]

2.3 Offline DAG Scheduling

Section 2.1.5 described that modern big data frameworks often use directed acyclic graphs (DAGs) to describe the dataflow of programs. Each node in such graphs represents a non-preemptive operation that is applied to some data. Arcs between graph nodes are data dependencies. DAG scheduling is the process of assigning graph nodes to processing elements (PEs) with the goal of minimizing the schedule length (or *makespan*). In the context of DAG scheduling, graphs are often called *task graphs* or *macro-dataflow graphs* [KA98]. It is important to note that the DAG scheduling problem occurs in different forms based on the assumptions that are made. Typical assumptions are, for example, that all PEs are homogeneous, all nodes have the same computational cost or that communication costs are zero. In its general form and for most of the assumptions, DAG scheduling is NP-complete [Ull75]. There are only a few restricted cases with polynomial time solutions. Scheduling an arbitrarily structured DAG with uniform weights and no communication costs to two PEs, for example, is possible in almost linear time [Set76].

Because NP-complete algorithms are only feasible with very small problem sizes, many DAG scheduling heuristics have been proposed for the general case as well as for restricted cases. These approaches can be divided into different classes based on various characteristics, for example the number of PEs they require. A widely accepted taxonomy of DAG scheduling algorithms was proposed by Kwok and Ahmad [KA99]. This taxonomy distinguishes between two classes, namely UNC (*unbounded number of clusters*) and BNP (*bounded number of processors*) algorithms. The former class assumes an unlimited number of PEs, while the number of PEs in the latter is assumed to be limited. UNC algorithms use a technique which is called *clustering* because they divide graph nodes into clusters to minimize the overall completion time. The result of such an algorithm is an arbitrary number of clusters. Therefore, further steps are required if the algorithm returned more clusters than there are PEs in the system. BNP algorithms on the other hand assume a fixed number of available PEs from the start and therefore do not need a second step.

Many DAG scheduling algorithms incorporate a technique called *list scheduling*. These algorithms assign priorities to graph nodes and create an ordered list of nodes based on their priority. After that, the algorithms iterate through the list and assign each node to a PE with the goal to minimize the schedule's makespan. [KA99]

The next three sections address list scheduling, UNC and BNP approaches, respectively. In the second and third section, an exemplary algorithm of the respective class is described in more detail.

2.3.1 List Scheduling

List scheduling is a technique which uses a scheduling list to compute a valid schedule from a given DAG. The general procedure of list scheduling is shown in Algorithm 2.1. A list scheduling algorithm first assigns priorities to all nodes in the given graph. The second step is to place all nodes in a scheduling list, ordered by their priority from highest to lowest. After the list was created, the algorithm iterates through the list and successively allocates each node to a PE so that its start time is minimized. [KA99]

Algorithm 2.1: General list scheduling procedure

```
1 routine LIST_SCHEDULING
2     assign priorities to all graph nodes;
3     create an ordered list of graph nodes based on their priority;
4     while the list is not empty do
5         remove the first node from the scheduling list;
6         allocate the node to a PE so that its start time is minimized;
7     end
8 end
```

How node priorities are calculated depends on the specific list scheduling heuristic. Commonly used properties for this are the top level (*t-level*) and bottom level (*b-level*) of nodes. The t-level of a node n is defined as the length of a longest path from an entry node of the graph to n, excluding the node itself. Analogous to the t-level, the b-level of n is defined as the longest path from n to an exit node of the graph. Entry nodes of DAGs are those that have no incoming edges, while exit nodes have no outgoing edges. In the context of DAGs, the path length is defined as the sum of all node and edge weights along the respective path. Consequently, the *critical path* of a DAG is defined as the longest path in the graph. Depending on the graph's structure and weights the critical path may not be unique. [KA99]

It should be noted that the terminology is not uniform in the literature. T-level and b-level are also called *co-level* and *level* [ACD74] or *downward rank* and *upward rank*, respectively [THW02].

Besides the assignment of node priorities, there are also many possible ways to assign graph nodes to PEs. The *heterogeneous earliest finish time* (HEFT) algorithm [THW02], for example, is compatible with different system topologies since it considers different execution times for different PEs and different data transfer speeds between pairs of PEs in the node allocation. Section 2.3.3 describes the HEFT scheduling algorithm in more detail.

The previous description of list scheduling only covered the traditional static list scheduling. However, dynamic approaches also exist. The difference lies in the scheduling list, which in standard list scheduling stays the same during the whole procedure. In dynamic list scheduling, each time a node is assigned to a PE the priorities of all unscheduled nodes are recomputed. General steps of dynamic list scheduling are shown in Algorithm 2.2. The advantage of the dynamic approach is that it has the potential of producing schedules with lower makespan. However, the scheduling gets more complex, which can be a downside of this approach. [KA99]

Algorithm 2.2: General dynamic list scheduling procedure

```
1 routine DYNAMIC_LIST_SCHEDULING
2     while not all nodes scheduled do
3         assign priorities to all unscheduled graph nodes;
4         choose the node with the highest priority;
5         allocate the node to a PE so that its start time is minimized;
6     end
7 end
```

2.3.2 UNC Algorithms

UNC algorithms are clustering heuristics. The main idea behind clustering heuristics is to divide graph nodes into clusters. Such algorithms usually start with each node in a separate cluster. Then, clusters are merged successively whenever the merge reduces the completion time. The number of resulting clusters is therefore only limited by the number of nodes in the graph. If the number of clusters is larger than the number of available PEs, a second step is required. This step merges clusters further until the number of clusters and PEs match. Amongst UNC algorithms, there exist heuristics that duplicate tasks in order to reduce the number of data transfers. These are known as *task-duplication-based* (TDB) algorithms. [KA99]

In the following, the *dominant sequence clustering* (DSC) algorithm proposed by Yang and Gerasoulis [YG94] is described as an example. This non-duplication-based clustering heuristic relies on the *dominant sequence* of a graph. A dominant sequence is a *critical path* of a scheduled or partially scheduled DAG. The DSC algorithm does not require any restriction on the graph. Any DAG with arbitrary node and edge weights can be scheduled. During scheduling, DSC iterates successively over all graph nodes. The algorithm distinguishes between *examined* and *unexamined* nodes. At the beginning, all nodes are unexamined. Nodes are called *free* if all of their predecessors are examined and *partially free* if at least one predecessor is examined. Entry nodes of a DAG are already free at the beginning. The DSC algorithm maintains two ordered list *FL* and *PFL*, which contain the free and partially free nodes, respectively. *FL* is sorted in descending order by the nodes' priorities, which are equal to the sum of their b-level and t-level. *PFL*, on the other hand, is sorted in descending order by the nodes' *p-priorities*. These are computed similar to priorities, with the only difference that for the t-levels only examined nodes are considered. Accordingly, this variation of the t-level is called *pt-level*.

The DSC scheduling heuristic is shown in Algorithm 2.3. At first, the algorithm initializes b-levels, t-levels and the lists *FL* and *PFL*. Then, it enters its main loop which is left when all nodes are examined. With each loop iteration, the heuristic successively merges clusters by setting edge weights to zero. A candidate for the merging is the first node in *FL*, i.e. the node with the highest priority. In Algorithm 2.3 this node is called n_x. By comparing the highest priority and highest p-priority, it is possible to check whether this node is on a dominant sequence. If n_x is on a dominant sequence, the algorithm sets some edge weights (possibly none) between n_x and its predecessors to zero so that the t-level of n_x is minimized. Otherwise, edge zeroing is done under the *dominant sequence length reduction warranty* (DSRW) constraint. The DSRW constraint says that zeroing incoming edges of a free node should have no influence on the reduction of n_y's pt-level if this exact pt-level is reducible by zeroing an incoming DS edge of n_y.

Finding edges that minimize the t-level (whether under the DSRW constraint or not) is not trivial. Since this action is performed in each loop iteration, a low time complexity is desired. DSC therefore uses binary search after sorting all incoming edges to distinguish between the edges which are zeroed and those which are not. Exact details on minimizing the t-level can be found in [YG94].

The time complexity of DSC is $O((e + v) \log v)$ where e is the number of edges and v is the number of vertices of the graph [YG94]. In case the number of clusters is greater than the number of available PEs, a second step for further merging is required. This leads in practice to additional scheduling effort. Nonetheless, in comparison to the NP-complete optimal solution DSC is well suited for practical use.

Algorithm 2.3: DSC scheduling heuristic [YG94]

```
1  routine DSC_SCHEDULING
2      compute the b-level of all nodes and set the t-level of all nodes to 0;
3      initialize the lists FL and PFL;
4      while there are unexamined nodes do
5          let n_x be the first element of FL and n_y be the first element of PFL;
6          if the priority of n_x is greater than or equal to the p-priority of n_y then
               // n_x is on a dominant sequence
7              zero the edges between n_x and its predecessors so that the t-level
                 of n_x is minimized;
8              if there are no such edges, n_x remains in a unit cluster;
9          else
               // no free node is on a dominant sequence
10             zero the edges between n_x and its predecessors so that the t-level
                 of n_x is minimized under the constraint DSRW;
11             if there are no such edges, n_x remains in a unit cluster;
12         end
13         update the priorities of all successors of n_x;
14         update FL and PFL according to the new priorities;
15     end
16 end
```

2.3.3 BNP Algorithms

In contrast to UNC algorithms, BNP heuristics assume that the number of PEs is limited. The majority of BNP algorithms are list scheduling algorithms. Like described in Section 2.3.1, traditional list scheduling BNP algorithms consist of two phases, a task prioritizing phase and a processor selection phase. How these two phases are implemented depends on the specific algorithm. As an example, the two phases of list scheduling in the *heterogeneous earliest finish time* (HEFT) algorithm proposed by Topcuoglu et al. [THW02] are described in the following. With regard to Kwok and Ahmad's taxonomy, HEFT belongs to the special class of *arbitrary processors network* (APN) scheduling heuristics. This class is a subclass of BNP, which contains algorithms that also consider the topology of the underlying hardware architecture [KA99].

The HEFT algorithm uses an abstract model for the system's topology. More specifically, HEFT requires (in addition to a DAG) two tables and one list containing different topology parameters. The first table consists of one estimated execution

time for each combination of PE and task. Each cell in the second table represents the bandwidth between a pair of PEs. Lastly, estimations for the communication startup times of PEs are gathered in a list. The HEFT algorithm is shown in Algorithm 2.4. Like traditional list scheduling algorithms, HEFT first creates a sorted list of nodes. Sorting is based on the *upward rank* of nodes (which corresponds to their b-level). The difference to the b-level in other algorithms is that average values are used for node and edge weights. This is because the actual node and edge weights depend on the assignment of nodes to specific PEs, which is computed in the second phase of the heuristic.

The processor selection phase consists of a main loop which successively iterates over all nodes. For each node, the PE that minimizes the *earliest finish time* (EFT) is selected. The EFT of a node is calculated by adding the execution time to the *earliest start time* (EST). Both the execution time and EST depend on the PE. While the execution time can be determined directly from the first table mentioned above, the EST value is more difficult to compute. To calculate EST, all predecessors of the current node must be considered. Since the scheduling list is by construction topologically sorted, all predecessors already have been scheduled. Therefore, each predecessor is already assigned to a PE and has an *actual finish time* (AFT). The sum of a predecessor's AFT and the appropriate communication time provides a lower bound for the EST. However, just using the maximum of all these lower bounds for EST is not enough since the PE under consideration may be executing another node at this point. Hence, the availability of PEs must also be considered in EST computations.

Algorithm 2.4: HEFT scheduling heuristic [THW02]

```
1  routine HEFT_SCHEDULING
2      compute mean values for all node and edge weights;
3      compute the upward rank for all nodes in the graph;
4      create a sorted list of nodes by nonincreasing order of upward ranks;
5      while the scheduling list is not empty do
6          let n be the first node in the list;
7          remove n from the list;
8          for each PE p do
9              compute the earliest finish time (EFT) of n on PE p;
10         end
11         assign n to the PE with the lowest earliest finish time;
12     end
13 end
```

In summary, the EFT of a node n_i on a PE p_j is computed as follows:

$$EFT(n_i, p_j) = w_{i,j} + \max\left\{avail[j], \max_{n_m \in \text{pred}(n_i)} (AFT(n_m) + c_{m,i})\right\},$$

where $w_{i,j}$ is the estimated execution time of node n_i on PE p_j, $avail[j]$ is the earliest time at which p_j is ready for task execution, $\text{pred}(n_i)$ is the set of predecessors of n_i, $AFT(n_m)$ is the actual finish time of n_m and $c_{m,i}$ is the estimated communication cost for the edge between nodes n_m and n_i.

Another important aspect of the HEFT algorithm is that it uses an *insertion-based scheduling* policy. This means that it looks for idle times on PEs during EFT computations. If a time slot which is long enough for the execution of a node and suitable in terms of data dependencies is found, HEFT inserts this node in the idle time slot. In terms of time complexity, the HEFT algorithm is very efficient. Its runtime is in $O(ep)$ where e is the number of edges in the graph and p is the number of PEs [THW02].

2.4 Online DAG Scheduling

An alternative to the described DAG scheduling heuristics which compute schedules offline is to determine the node mapping and execution order dynamically at runtime. Two well known techniques for scheduling DAGs at runtime are *work stealing* and *work sharing*. This section focuses on work stealing since it is more commonly used. The properties of work stealing are well studied [BL99] and many parallel computing frameworks, for example Cilk [Blu+95] and Intel Threading Building Blocks [Rob11], as well as dataflow-based frameworks, for example DARTS [SZG13] and KAAPI [GBP07], use this technique due to its efficiency.

Graphs in this section have the same properties as in the previous section, i.e. nodes are non-preemptive and all dependencies are data-dependencies. A visualization of the work stealing principle is shown in Figure 2.3. Work stealing is typically implemented with one double-ended queue per PE. In order to execute a graph node, PEs always extract the front element from their queue. When a PE has executed a node, subsequent nodes which are now ready to be executed are inserted also at the front of the PE's queue. In case its own queue is empty, the PE tries to steal a node from the back of another queues. There are different possibilities on how to choose the queue from which a node is stolen. A common, well-studied approach is to choose the queue randomly [BL99]. Regardless of how the queue is chosen, stolen nodes are always extracted at the back of the queue. To initialize the work stealing procedure, all nodes that are ready to be executed from the beginning are inserted into one of the queues.

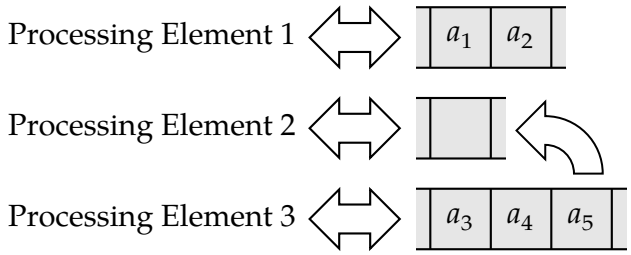


Figure 2.3: Visualization of the work stealing principle

Work stealing is commonly used in practice due to its beneficial characteristics. By finishing the execution of a node, only nodes with a data dependency can become ready. Therefore, inserting new nodes at the front of the queue and also extracting them at the front improves data locality in graph executions. Furthermore, since PEs steal nodes from the back of other PEs' queues, the disturbance of data locality is low. Another benefit lies in the performance of work stealing approaches. The procedure itself is fast and synchronization is easy due to the short critical sections which are limited to queue insertion and extraction operations. As a result, the overhead during graph execution caused by work stealing is small.

2.5 Fault Tolerance

Finding practical ways to utilize computer systems in environments with high safety requirements continues to be an important topic in computer science. To meet the requirements for such systems, mechanisms to prevent system *failure* are indispensable or otherwise human lives may be endangered. Failures are malfunctions that causes the system to not meet certain guarantees like correctness or performance. They always appear in scopes that are visible to users. *Errors* on the other hand appear at a deeper level, i.e. in scopes where they are not observable by users, but they are otherwise similar to failures. When an error is not corrected by the system, it may propagate to an outside scope and become a system failure. Errors are manifestations of *faults* in the system, but not every fault causes an error. Sometimes the effect of a fault is masked or tolerated. Faults can have different causes and can be divided into different categories which can also be used to classify errors. Causes of faults and the different categories of errors are described in the following sections in more detail. [Muk08, pp. 6–8]

2.5.1 Causes of Faults

Knowing the causes of faults which may occur in a safety-critical system is crucial for the design of both the software and hardware. Otherwise, a fault that was not considered may propagate to an outer scope and become a system failure. In the following, some common causes of faults are characterized.

Radiation is a well researched cause of faults in semiconductor devices. Types of radiation that are demonstrably causing faults include alpha particles, high-energy cosmic rays and, in some cases, even low-energy cosmic rays. Alpha particles are emitted by some radioactive materials. Since they do not travel very far, only the contamination of the chip package may become a problem. Cosmic rays may contain different particles, of which neutrons have the highest chance to influence electronic devices. Further, they are more intense at higher altitudes in the atmosphere so that processors in airplanes are more affected than in devices or vehicles on the ground. The effect of radiation is often a flipped bit in a memory cell, register, latch or flip-flop. However, if the intensity is high enough, it is also possible that multiple bits are flipped. [Bau05]

Wearout is another potential cause of faults. There are several physical effects that contribute to the overall aging of transistors. Two notable examples are *negative bias temperature instability* [Wan+10] and *hot carrier injection* [Tak+83]. Wearout affects the threshold voltage of transistors and reduces frequency at which the transistor can be operated [Wan+10]. Therefore, possible faults caused by wearout include timing violations and functional failure of transistors.

Heat is also a factor which affects the operation of processors and memory. More precisely, temperature influences different properties of transistors, for example the individual transistor current, threshold voltage and gate delay. Furthermore, heat makes electronic components more susceptible for different forms of radiation. As a result, high temperature is another factor that may lead to additional faults in a system. [Jag+12]

2.5.2 Types of Faults

Faults are usually divided into three different categories based on their duration. These categories are *transient faults*, *intermittent faults* and *permanent faults*. In the following, the characteristics of each category are pointed out briefly.

Transient Faults are characterized by the fact that they do not persist and occur only once, i.e. not periodically or regularly. Many of the causes described in the previous section can lead to a transient fault. The most notable cause for this type of fault is probably radiation [Con02]. Errors resulting from transient faults are often called *single event upset* (SEU) or *soft errors* [Sor09, p. 3].

Intermittent Faults differ from transient faults in that they occur repeatedly at the same location and at a higher rate. An important aspect of intermittent faults is that they do not appear continuously, which makes them hard to predict. In contrast to transient faults, intermittent faults can be removed by replacing the offending circuit since the cause of such faults is often variability in chip production and wearout. [Con02]

Permanent Faults are, once they occurred, persistent. This type of fault causes hardware units to consistently deliver wrong results rendering them not usable anymore. Like intermittent faults, permanent faults can be removed by replacing the affected hardware units. They are also similar to intermittent faults in their causes. Nevertheless, the distinction between the two types of faults makes sense since it allows to treat them differently. Hardware units with intermittent faults may still be usable if faults do not appear too frequently. [Sor09, p. 3]

2.5.3 Error Models

System developers are rarely interested in the whole process from the physical phenomenon to a resulting fault and lastly to a detected error but rather in the types of errors that may occur and to what extent they occur. Therefore, it is useful to abstract from physical and technical details and create a model describing only the required aspects of faults. This allows developers to design systems that tolerate errors within a set of error models. To be useful, error models should closely represent possible faults in a system, especially the more likely faults. [Sor09, p. 7]

Error models can be classified based on three characteristics, namely the error type, duration and number of simultaneous errors. Categories of error models based on their type are *stuck-at errors*, *bridging errors*, *fail-stop errors* and *delay errors*. Stuck-at errors cover all types of physical faults which cause one bit in a circuit to be stuck at either 0 or 1. Situations where values are coupled, for example because of a short circuit, are covered by bridging errors. The category of fail-stop errors models situations in which a processing element, memory or other component completely stop working. Lastly, the delay error model is used to abstract from faults caused by hardware units which return a value later than expected. Categories

for error models based on the second characteristic, the error duration, correspond to the types of faults described in Section 2.5.2. The last characteristic of an error model, i.e. the number of simultaneous errors, covers which physical faults may occur simultaneously. Although relatively rare, simultaneous faults may have to be considered to meet the safety requirements in highly safety-critical systems. [Sor09, pp. 7–9]

2.5.4 Redundant Execution

Redundancy is an essential concept for error detection in fault-tolerant systems. Without redundant computation and storage, it is impossible to verify whether data is correct or was affected by an error. Redundancy techniques can be classified in different ways. One possible way is to distinguish between *physical redundancy*, *temporal redundancy* and *information redundancy* [Sor09, p. 19]. Another way is the distinction between *software redundancy* and *hardware redundancy* [Avi76].

Physical, Temporal and Information Redundancy

Physical redundancy is achieved by adding redundant modules to a system. The two most common approaches are *dual modular redundancy* (DMR) and *triple modular redundancy* (TMR). In a DMR setting two modules perform the same computation. A comparator then checks whether their results are equal. Under the assumption that the comparator is free of faults, all types of errors can be detected in this way. Only the unlikely case of the same error occurring in both modules is a problem. If DMR does not satisfy the requirements of a safety-critical system, TMR might be an option. TMR involves a third module for computation. Further, instead of a comparator, it features a voter which not only compares the three results but also identifies the correct value. This, of course, only works if two out of three results were computed correctly. Higher grades of physical redundancy, commonly referred to as *N-modular redundancy* (NMR) with N greater than three, are also possible and enhance, for odd numbers of redundant modules, the error detection and recovery capabilities even further. [Sor09, pp. 19–22]

In contrast to physical redundancy, *temporal redundancy* does not require additional modules. Instead, a module performs the same computation twice or more and compares the results afterwards. Temporal redundancy has the advantage that no additional hardware is required, but the runtime of a module is at least twice as long. Pipelining approaches may be a solution for reducing the overall runtime. Lastly, with regard to energy consumption, temporal redundancy behaves similar to physical redundancy. [Sor09, p. 22]

Information redundancy differs from the two redundancy approaches described because it adds redundancy to the data itself instead of multiple computations and duplicate storage. It can be used to detect errors in data transfer and storage but not in computation. Most common are *error-detecting codes* (EDC) and *error-correcting codes* (ECC) which extend data words by additional redundancy bits [KK07, p. 55]. The easiest way to achieve error detection is by adding a parity bit to each word. This allows to detect all single-bit errors, but it is not possible to determine which bit was flipped [KK07, p. 57]. More complex codes can detect more errors and are often able to correct them. Cyclic codes which are based on multiplication and division with *generator polynomials* are an example. Depending on which polynomial was chosen cyclic codes can be used to detect multiple adjacent bit flips [KK07, pp. 67–68].

Hardware and Software Redundancy

Hardware redundancy involves specialized hardware for storage and computation. There are two types of hardware redundancy. In the *static hardware redundancy* approach, redundant modules act as if there was only one non-redundant module. Errors in a module are detected, and error recovery is performed without notifying other modules or the software. Therefore, interfaces between modules can remain the same. Since this type of redundancy is invisible to other modules and the software, it is also called *masking*. Static hardware redundancy can be used against all types of errors. A disadvantage is that redundant components are close to each other and an error may affect both components in the same way. *Dynamic hardware redundancy* differs from the static approach. A redundant module notifies other hardware modules or the software when it detects an error. Error correction is carried out in a separate step. In this context, a viable option for the first step, i.e. the detection of errors, is information redundancy. A minor disadvantage of the dynamic approach is that the design choice has to be made early in the hardware development process. Masking, on the other hand, makes it easier to replace a non-redundant module with a redundant module at a later stage. [Avi76]

Software redundancy achieves fault tolerance by extending the system software. Based on the specific requirements, the granularity can vary from additional programs over program segments to additional instructions. It is possible to realize both error detection and correction in software. An advantage which results from this is that software redundancy can be used to realize fault tolerance on standard hardware with very limited supporting features. But software redundancy concepts can also be combined with dynamic hardware redundancy. The main disadvantage is that it is difficult to ensure that the software works still correctly after an error oc-

curred. Especially errors in memory locations containing program code are difficult to handle in software. [Avi76]

2.6 Summary

This chapter introduced the main concepts on which the proposed runtime environment is built. Dataflow is an alternative to the standard von-Neumann execution model. In the dataflow model, instructions are not triggered by control flow, but instead by their operands. A benefit of the dataflow approach is that it allows to fully exploit the parallelism of applications. Today, dataflow concepts are used, for example, in out-of-order superscalar processors and big data frameworks.

In the functional programming paradigm, programs consist solely of pure functions, i.e. functions without side effects. Furthermore, functions are treated like values and can be passed to functions or returned by functions. Many functional programming languages provide a rich type system with type inference and polymorphic types. Because of its declarative approach and referential transparency, this paradigm is well suited for use in dataflow computing.

Offline DAG scheduling is the process of assigning tasks with data dependencies to PEs. The goal is usually a small makespan of the schedule. Since the DAG scheduling is NP-complete, many heuristics were proposed. These heuristics make different assumptions about the scheduled graph or hardware.

A common method to schedule DAGs dynamically at runtime is work stealing. The basic principle behind work stealing is that PEs steal nodes from the queues of other PEs if their own queue is empty. Since nodes are inserted and extracted at the front of the queue and stolen from the back, data locality in the graph execution is preserved.

The last major concept, fault tolerance, plays an important role in safety-critical systems. Faults can have various causes and appear in different shapes. When the effect of a fault is visible, it becomes an error. Error models help to abstract from physical faults by focussing on the relevant details. To prevent the system from failure, error detection and correction is mandatory. The only way to achieve this is by adding redundancy to the system. There are multiple ways to introduce redundancy so that different error models can be covered.

3

Runtime Environment Overview

The previous chapter introduced the four basic concepts on which the proposed runtime environment (RTE) is based, namely dataflow, functional programming, graph scheduling and fault tolerance. This chapter gives an overview of the RTE's structure and describes how these concepts are combined. Later chapters will then highlight the different aspects of the RTE in more detail.

The RTE aims at bringing parallel applications to safety-critical embedded systems. In this context, the following requirements had an influence on the design of the software architecture:

- Programmability: The programming model should be similar to existing programming frameworks.
- Architecture Support: It should be possible to implement the RTE on different hardware architectures using standard programming languages and compilers.
- Synchronization: The RTE should automatically handle the synchronization on multicore shared-memory architectures
- Data Transfers: On hardware architectures with a network-on-chip, the RTE should automatically handle all data transfers.

- Small Overhead: The RTE should introduce only small overhead in terms of execution time and memory so that it is suitable for embedded multi- and manycore systems with low-power processors and small memories.
- Scalability: Despite the focus on embedded systems, the RTE should also perform well on architectures with high-performance processors and larger memories.
- Analyzability: Program executions should be easy to analyze.
- Redundancy: The RTE should support fault tolerance through redundant execution.
- Adaptive Redundancy: It should be possible to change the degree of redundancy during runtime.

A dataflow approach is well suited to meet most of the described requirements. With regard to programmability, a functional-style programming model allows users to specify suitable programs in a declarative way.

The following sections provide information on the RTE's software architecture. Section 3.1 gives an overview of the RTE's internal structure and shows how the individual components interact with each other. After that, basic information about the different parts of the RTE is provided. Section 3.2 briefly describes the structure of dataflow graphs. In Section 3.3, concepts of the functional programming model are shown. The last three sections cover graph executions on different architectures, scheduling and redundancy, respectively.

3.1 Software Architecture Overview

The proposed RTE uses directed acyclic graphs (DAGs) to model dataflow programs. Users can create graphs either through the programming model and its functional-style operations or by specifying them explicitly in a graph description format so that they can be imported by the RTE. Dataflow graphs are usually scheduled before their first execution and then executed multiple times, often repeatedly. However, there is also the option to switch to online scheduling so that the scheduling step can be omitted. Internally, the RTE consists of four parts, which are responsible for graph construction, import/export, offline scheduling and graph execution/online scheduling. To keep the RTE modular, the different parts only communicate by exchanging dataflow graphs and do not depend strongly on each other. This allows users to replace parts of the RTE with custom implementations or to remove the parts of the RTE that are not relevant for their use cases. Figure 3.1 shows an overview of the RTE's internal structure. The lines connecting the four RTE parts represent interfaces for graph exchange.

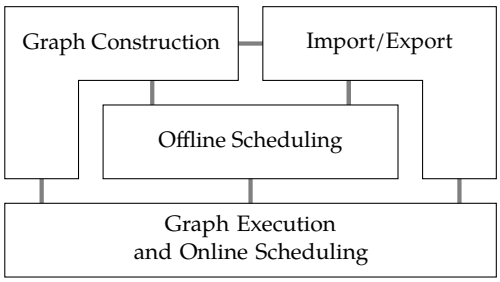


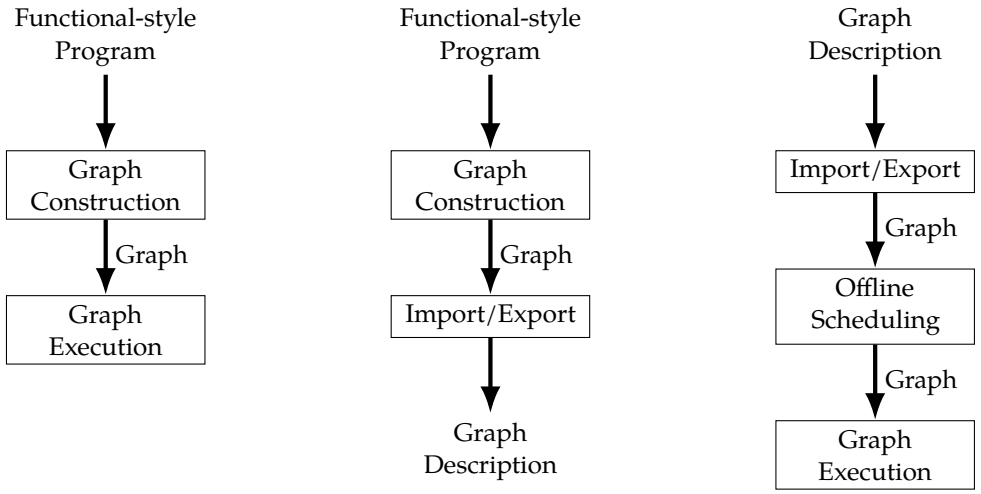
Figure 3.1: Internal structure of the proposed RTE

Graph Construction. The RTE provides a set of functional-style operations to the user. These operations are not executed immediately on the data, but instead build a graph representing the data dependencies. Once its construction is finished, the graph is passed to a different part of the RTE, either to acquire a schedule (via import or by applying a scheduling heuristic) or to be executed directly with online scheduling. Figure 3.2a shows an exemplary use case. In this example, a user has written a program consisting of functional-style operations. When this program is executed, the RTE constructs a dataflow graph. Although the RTE does not restrict their use, it is recommended to make sure that operations are executed on system startup. The constructed dataflow graph is then executed, possibly multiple times or even repeatedly.

Import/Export. Dataflow graphs can be stored externally in a graph description format. Such a graph description explicitly contains information about all nodes, data dependencies and memory requirements. Specifying graphs directly in a description format is possible, but it is usually much easier to use the provided functional-style operations since they automatically compute memory requirements and also some data dependencies. In the proposed RTE implementation, an extended DOT format [GKN15] is used.

Aside from graphs, it is also possible to import schedules which were stored externally in a suitable description format. For a given unscheduled graph, the import/export module reads a pre-computed schedule, assigns it to the graph and passes the scheduled graph to the graph execution module. As with graphs, schedules computed by the RTE can also be exported.

Two use cases illustrating the RTE's import and export capabilities are shown in Figure 3.2b and Figure 3.2c. In the former, a graph created from a RAPID program is passed to the import/export module and converted into a representation in the description format. The latter use case shows the RTE's import functionality. A graph description is read by the import/export module, transformed into a dataflow graph in the main memory and passed to different parts of the RTE.



a) Graph execution b) Graph export c) Import and scheduling

Figure 3.2: Exemplary RTE use cases

Offline Scheduling. The proposed RTE uses DAG scheduling heuristics to compute schedules for graphs received from the graph construction and import/export. An example use case is shown in Figure 3.2c. Scheduled graph can be passed to the import/export model in order to export the computed schedule or to the graph execution model. The RTE implementation provides two variants of the HEFT algorithm (see Section 2.3.3). If this heuristic is not suitable for the desired application, users can easily expand the RTE by adding their own algorithms.

Graph Execution. This part of the RTE receives graphs from other parts of the RTE and executes them in a dataflow fashion. The RTE only provides functions that allow users to start graph executions. Therefore, the decision how often a graph is executed and when a graph should be executed is up to the user. Graphs without an associated schedule can only be executed in online scheduling mode. In the proposed implementation, the RTE supports graph executions with work stealing (see Section 2.4), but as before, it is possible to extend the RTE by additional procedures.

3.2 Dataflow Graphs

Dataflow graphs in the proposed runtime environment are bipartite and consist of actor nodes and partition nodes. Actor nodes (often just called actors in this chapter) contain information about how data is processed during graph execution,

while partition nodes are used to keep track of memory requirements. To support adaptively redundant dataflow executions with fixed schedules, graphs are divided into sections. How the division exactly takes place is a design choice that users of the RTE have to make. The offline scheduling part of the RTE computes an independent schedule for each graph section. More specifically, to support different redundancy configurations, multiple schedules per section are created. During graph execution, the schedule of a section can be changed individually whenever the execution reaches a state between two sections.

Figure 3.3 shows an example graph with two sections. Partition nodes are depicted as filled circles, white boxes represent actor nodes. From the graph's structure, it is easy to see the available parallelism. Besides the incoming and outgoing edges, the simplified graphic representation in Figure 3.3 shows no other node properties. More details on the different nodes follows in Chapter 5.

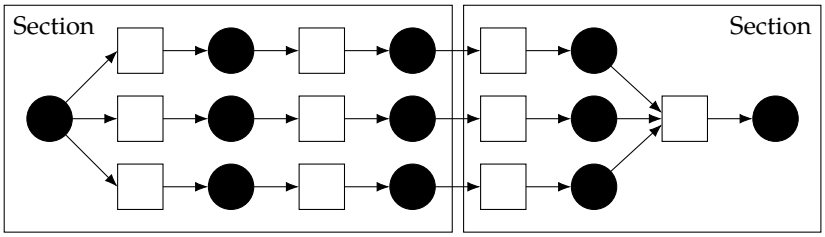


Figure 3.3: Example dataflow graph

3.3 Functional Programming Model

As the previous section briefly described, the programming model provides a set of functional-style operations that are used to construct compatible graphs for dataflow executions. But besides these operations, there are three other important concepts, namely RAPIDs, RAPID functions and contexts. A brief overview on the four concepts of the programming model and their relation to dataflow graphs is shown in Figure 3.4.

RAPIDs (Resilient Analyzable Partitioned Immutable Data structure) are an abstraction for the data in dataflow graphs. Users can treat RAPIDs similar to data collections, with the exception that the actual data cannot be accessed since it is computed later during graph execution. To compensate the lack of access to the data, the programming model allows users to specify how the data is processed in a declarative way. For this purpose, most RAPID operations are high-order functions and require a RAPID function as one of their parameters. But instead of applying the given RAPID functions immediately on the data, RAPID operations construct

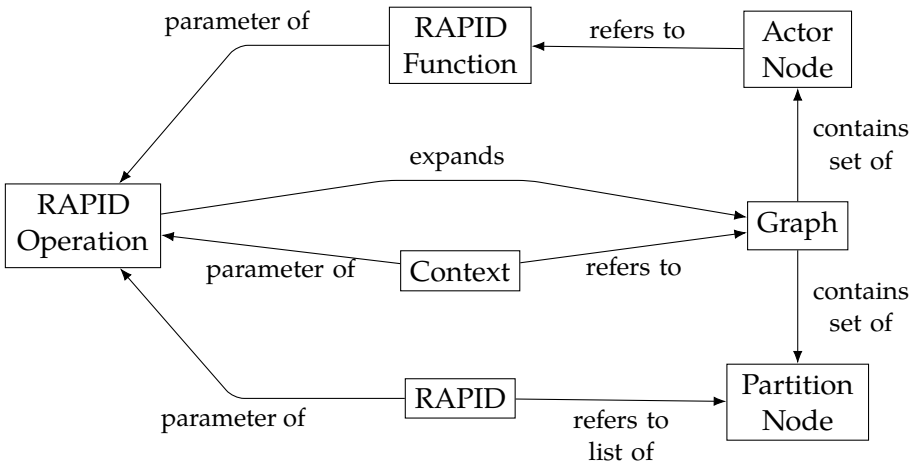


Figure 3.4: Relationship between programming model concepts and graphs

dataflow actors containing references to the respective functions. A user who is not aware of the dataflow internals could consider it a lazy evaluation.

The other important data structure besides the RAPID is the context. Contexts are high-level abstractions over dataflow graphs and schedules. Using contexts is not necessary to write working programs in the programming model but mandatory to utilize the RTE's full functionality. A context contains one reference to a dataflow graph and possibly references to schedules for different redundancy configurations. By passing a context to a RAPID operation, a user can specify which graph is supposed to be expanded by this operation. Further, the context provides methods to utilize the RTE's import/export functionality and to influence how the referenced graph is scheduled.

The programming model shares many characteristics with functional programming languages, for example the use of high-order functions as RAPID operations. Further, RAPID operations are generic functions and RAPIDs are collection-like structures with a generic type. Thus, the programming model incorporates parametric polymorphism. Concrete types can be deduced by the RTE with type inference in most cases. Lastly, all RAPID function have to be pure functions. If a RAPID function could freely access global memory, the execution of a respective actor could lead to unexpected behavior on architectures with multiple distributed memories.

3.4 Dataflow Execution

The proposed RTE follows the coarse-grain dataflow approach described in Section 2.1.4. Actors process partitions, i.e. data collections, rather than single values

and apply functions to the data instead of single instructions. The size of data partitions and extent of RAPID functions is a design choice made by the RTE user. Nonetheless, the underlying hardware architecture, in particular the size of its memories, may set some restrictions. Dataflow in the RTE is static since it is based on DAGs and does not require a token matching mechanism. A benefit of this approach is that graphs can be scheduled with standard DAG scheduling heuristics.

The higher-level portions of the proposed RTE, namely graph construction and import/export, as well as the dataflow graph can be implemented in a hardware-independent way. However, this does not apply to scheduling and dataflow execution. Implementations of these parts of the RTE must consider the underlying hardware to achieve high performance. In the following, a brief overview of the RTE's internal structure on two different hardware architectures is given. More detailed information is provided in Chapter 6.

The first architecture is a standard shared-memory architecture. Figure 3.5 shows an overview. Implementing the RTE on such architectures is straightforward. Each core has full access to the single main memory, which is large enough to store all dataflow graphs, schedules and RAPID functions. Therefore, all cores are able to check which actors are already done and what the next actor according to their schedule is. Ensuring correct synchronization in a parallel program is usually a difficult task on shared-memory architectures. In the proposed RTE, the complexity of synchronization is reduced since both the graph and schedule are read-only during graph execution. Further, partitions get immutable as soon as their data is fully computed. Synchronization is only required when a core marks an actor as done or checks whether an actor is marked accordingly. It is also worth noting that RTE implementations on shared-memory architectures impose the least number of restrictions to the user. Since the shared memory is usually large, partitions

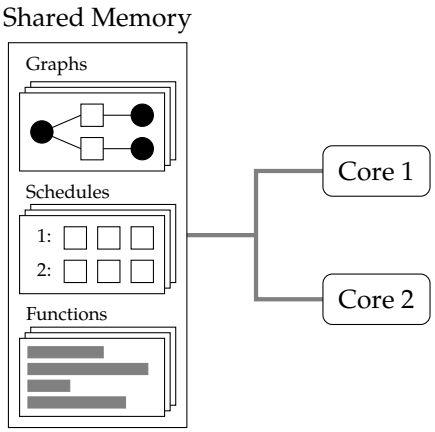


Figure 3.5: Shared-memory architecture overview

can be processed efficiently, regardless of their size. This might not be the case for distributed architectures, for example the one described in the following.

The second architecture is based on a network-on-chip (NoC). A simplified overview is shown in Figure 3.6. The NoC connects multiple tiles which communicate via message passing. Only one tile is connected directly to a memory which is large enough to store graphs and schedules. This tile is called the *driver tile*. All other tiles only store the RAPID functions (which are part of each tile's binary) permanently. During graph execution, the driver tile acts as a coordinator and transmits data and actors from its larger memory to the other tiles. Compute tiles on the other hand only execute actors and transmit new data to other compute tiles or the driver. Local memories of compute tiles are small and can only hold few small partitions. This is a restriction, since data should be divided in a way so that partitions fit in local memories whenever possible. In case an actor processes larger amounts of data, it must be executed on the driver tile. While occasional actor executions on the driver can often be tolerated, frequent data processing on the driver reduces parallelism and leads to poor performance. The RTE does not split large partitions automatically, so it is up to the user to write proper RAPID programs with respect to the hardware architecture. As with the shared-memory architecture, ensuring correct synchronization is rather simple. Graphs are managed by the driver tile, so there is no synchronization required when the status of a node is updated. However, based on how message passing in the underlying hardware architecture is working, synchronization might be required. If communication is done via *direct memory access* (DMA) transfers, for example, the RTE must ensure that two concurrent transfers do not write to the same memory location.

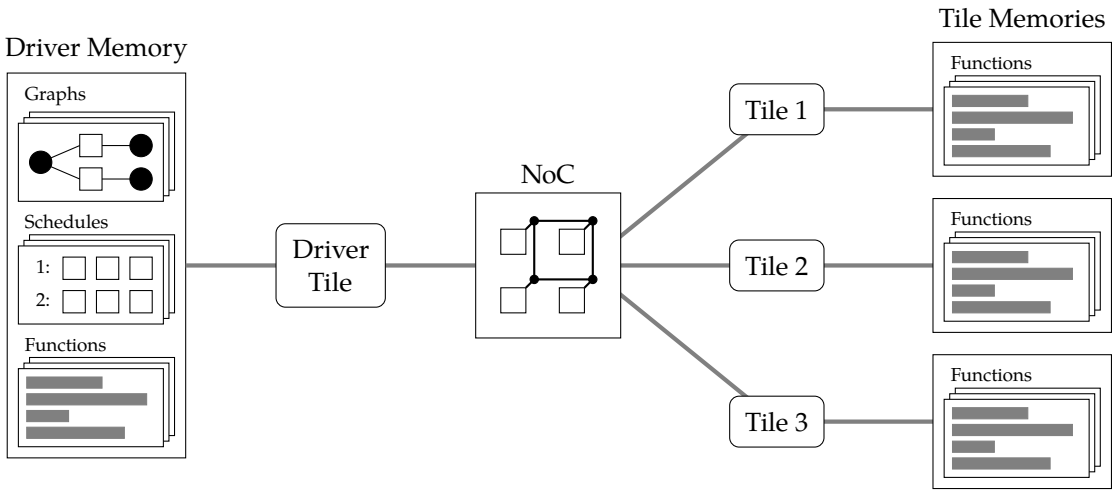


Figure 3.6: NoC-based architecture overview

3.5 Scheduling

The scheduling part of the RTE computes schedules for all graph sections. As with graph execution, it is not possible to implement this portion of the RTE in a hardware independent way (at least not without loss of performance). An appropriate implementation must consider the underlying hardware architecture, not only to compute schedules with a small makespan but also because there may be constraints about which actors can be executed on which hardware components (cores, tiles, ...). An example would be actors processing large amounts of data. On the NoC-based architecture described in the previous section, such actors must be assigned to the driver tile due to the small memory size on compute tiles.

Since the proposed RTE is based on DAGs, arbitrary DAG scheduling algorithms can be used. The HEFT algorithm described in Section 2.3.3, for example, is well suited for many hardware architectures. It should be noted that in the proposed RTE only the ordering and assignment of actors matter. Even though a scheduling algorithm might also compute exact execution times for actors, this information is discarded when the algorithm is finished. The RTE follows a dataflow approach and executes an actor as soon as its arguments are ready and all preceding actors in the schedule are done. Therefore, the exact timing of actor executions may vary in different executions of the same graph.

Besides offline scheduling with DAG scheduling heuristics the RTE also supports online scheduling. In online scheduling mode, the decisions about assignment and order of actors are made during runtime. An actor is scheduled as soon as it gets ready. Since online scheduling takes place during dataflow execution and requires access to the RTE's state in order to make proper decisions, it is closely tied to the RTE's graph execution part.

3.6 Redundancy and Error Model

The proposed RTE supports fault tolerance through redundant actor executions. After an actor is executed redundantly, a comparison actor checks if all results are equal. If the results do not match, error correction depends on the exact number of redundant executions. The RTE supports up to three redundant actor executions. In case of only two redundant actors, the redundant actors are re-executed. When an actor is executed three times, the comparison actor functions as a voter so that no re-execution is required (provided that there is only one error at a time).

Redundant actors and comparison nodes are automatically created during graph construction and import. Further, the RTE always computes three schedules with different redundancy for each graph section. As mentioned in previous sections,

redundancy can be changed during runtime. A change in redundancy is only possible when graph execution reaches a point between two sections and always affects a section as a whole.

Regarding the error model, the RTE's redundancy mechanism focuses primarily on the detection and correction of transient errors in actor executions. Furthermore, an assumption is that errors only affect the data produced by an actor, i.e. errors which prevent a processing element from finishing an actor execution are not considered. Such errors would require a mechanism different from redundancy, for example watchdog timers, which is not the topic of this thesis. An additional assumption is that only one out of two or three redundant actors may produce incorrect data (even though the RTE implementation described in this thesis will probably handle situations with two errors correctly as long as both incorrect results are different). Based on how the comparison actor is implemented, the RTE can detect single- or multi-bit errors. If the results of redundant actors are compared explicitly (for example byte-wise or word-wise), all differences are detected. Comparing checksums of results may not detect all differences but reduces the amount of data transmitted between tiles on NoC-based architectures. As mentioned above, this thesis focuses on the redundancy of standard actor executions. Errors in comparison actors and other parts of the software, for example RTE management code, graph construction and scheduling are not considered, i.e. these software components are considered as error-free. A way to ensure that the RTE is not affected by errors occurring in these parts of the software would be, for example, the use of hardware redundancy.

It should be noted that some chapters in this thesis go beyond the described error model and also discuss methods for graceful degradation when a processing element becomes unusable due to a permanent error. This applies in particular to Chapter 7, where different graceful degradation approaches which are compatible with offline scheduling are described and compared.

3.7 Summary

This chapter presented an overview of the proposed runtime environment. The programming model provides a set of functional-style operations. These operations build a directed acyclic dataflow graph. An alternative way to obtain dataflow graphs is by using the RTE's import functionality, which allows the RTE to create graphs from representations in a graph description format. Constructed graphs can be either scheduled offline or executed immediately with online scheduling. In the former case, the RTE uses a DAG scheduling heuristic which may need to be slightly adjusted depending on the architecture. To support different redundancy config-

urations, the RTE computes multiple schedules. It is possible to switch between these schedules at user-specified checkpoints during graph execution.

The next chapters describe the different aspects of the RTE in more detail. Chapter 4 highlights the programming model from a user's point of view. How dataflow graphs are constructed is the topic of Chapter 5. After that, Chapters 6 and 7 provide more information about graph execution and scheduling on the two hardware architectures introduced in Section 3.4.

4

The RAPID Programming Model

This chapter describes a functional programming model that was designed to leverage dataflow applications on embedded multicore-systems. The programming model is similar to common large-scale cluster computing frameworks, especially Apache Spark. Since one of the requirements was that the corresponding dataflow runtime environment only introduces small overhead in the application binary, the number of provided functions in the programming interface is rather small. But although the programming model offers reduced functionality, it allows users to conveniently implement a variety of typical algorithms in the domain of high performance embedded applications, such as matrix multiplication and fast Fourier transform. Other requirements that had an influence on the programming model's design were good analyzability and easy portability of the dataflow runtime environment as well as fast dataflow executions on different hardware architectures.

Because of the high availability of C/C++ compilers on embedded platforms, the programming model's reference implementation is based on C++14. Although the programming model does not depend on a specific programming language, there are some typical functional language properties that are useful to implement the RAPID programming model, for example type inference and lambda expressions.

Like Apache Spark, the RAPID programming model is centered around special data collections. We call these collections *RAPIDs* as they represent *Partitioned Immutable Data* structures enabling *Resilient* and *Analyzable* program executions. The

associated execution model is a dataflow execution model based on a directed acyclic graph (DAG), which is constructed from RAPID operations and the dependencies between them. Graph construction and dataflow execution are separate steps in a RAPID program. This ensures that graphs can be constructed at system initialization. Furthermore, with a suitable programming language, a construction of dataflow graphs already at compile time is also conceivable.

The rest of this chapter is structured as follows: Sections 4.1 to 4.3 highlight the basic functionality provided by the RAPID programming model, in particular data structures, high-order RAPID operations and the functions which are passed to RAPID operations. Additional concepts of the programming model are described in Section 4.4. After that, Section 4.5 gives insight into the reference implementation of the programming model in C++14. The chapter is concluded with three examples that show how common compute tasks can be implemented efficiently with RAPID operations.

4.1 Data Structures

The two main data structures in the RAPID programming model are *RAPIDs* and *partitions*. From a user's perspective, RAPIDs are immutable collections consisting of one or more partitions which contain data elements. In the implementation of the runtime environment, however, neither RAPIDs nor partitions contain actual data. Both are merely pointer structures pointing to partition nodes in the dataflow graph. Concrete data of partition nodes is computed later during dataflow execution.

In contrast to the resilient distributed datasets (RDDs) from Apache Spark, the order of partitions in a RAPID as well as the element order in a partition is fixed. This eliminates the necessity of key-value-pairs in applications requiring ordered elements. A list containing size information of all partitions inside a RAPID in the correct order is called the RAPID's *partitioning*.

Both data structures, RAPIDs and partitions, are parameterized with the data type of the elements they (conceptually) contain. It is important that all data elements of a type have the same size. For this reason, valid data elements in the RAPID programming model are simple values (for example integers or floating point numbers), fixed-size arrays or structured data elements (for example structs/classes in C++) without dynamically allocated members. This restriction ensures that the size of each partition can be deduced from the type of the respective data elements at graph construction so that the memory requirements of a graph can be determined statically.

4.1.1 RAPIDs

RAPIDs are the main data structure in the RAPID programming model. The functionality a RAPID provides to a user is depicted in Figure 4.1. Types with angle brackets are parameterized types. An important aspect about RAPIDs is that their specification does not include a constructor. RAPIDs can only be constructed from ordinary data collections, like arrays, or from existing RAPIDs through *RAPID operations*. Once created, the user has neither writing nor reading access to the underlying data. Only the RAPID's metadata, like the size or partitioning, can be accessed. The structure also keeps track of the *context* (see Section 4.4) to which it belongs. Further, RAPIDs provide a `persist` member function whose return value is just a reference to the RAPID itself. This function has to be called if the user would like to use a RAPID as input of multiple RAPID operations because non-persistent RAPIDs are consumed by RAPID operations. The reason for this is to allow the system to safely remove unnecessary non-persistent RAPIDs at graph construction time for optimization purposes (see Section 5.3). The RAPID programming model leaves the behavior when a non-persistent RAPID is used multiple times up to the implementatition.

Internally, a RAPID consists of a list of pointers to partition nodes and a pointer to the context the RAPID belongs to. At this point, it should also be noted that RAPIDs can be reassigned. The assignment of a RAPID is only a shallow copy, so the duplicate's pointers point to the same partition nodes in the graph as the original RAPID's pointers. Therefore, using both the original RAPID and the new one in two different RAPID operations requires to call `persist` for at least one of the RAPIDs.

4.1.2 Partitions

Partitions are the second important data structure in the programming model. Like RAPIDs, partitions are pointer structures, which point to partition nodes in the

rapid⟨T⟩
`size(): integer`
`partition_count(): integer`
`partitioning(): partitioning`
`get_context(): context`
`persist(): rapid⟨T⟩`

partition⟨T⟩
`at(index: integer): T`
`data(): memory_address`
`index(): integer`
`partition_count(): integer`
`size(): integer`

Figure 4.1: Functionality provided by RAPIDs and partitions

dataflow graph. They cannot be constructed directly either. Instead, partitions only appear as parameters of partition-based RAPID functions (see Section 4.3.2). Their purpose is to allow users to specify how a RAPID operation shall process data by giving them an interface to access input and output data.

From a user's point of view, each partition belongs to exactly one RAPID. In actual implementations, however, it is possible that different partitions (or RAPIDs) point to the same partition nodes in the dataflow graph. Section 5.2 highlights this in more detail.

Member functions of partitions are shown in Figure 4.1. In contrast to RAPIDs, partitions allow users to access the underlying data through the member functions `at` and `data`. Further, the partition's size and index in the RAPID can be accessed. Lastly, the member function `partition_count` returns the number of partitions in the superordinate RAPID. This function allows a user, for example, to check whether a partition is the last partition in its RAPID.

4.2 RAPID Operations

The RAPID programming model provides a small set of functional-style operations which create new RAPIDs from existing RAPIDs or ordinary data collections. The way a RAPID operation transforms data depends on the type of operation and, for some RAPID operations, on the given *RAPID function* (see Section 4.3). All operations have in common that they do not process data immediately. Instead, RAPID operations build a dataflow graph which is executed at a later point in time. To write working RAPID programs, it is not necessary for users to be aware of this postponed data processing. However, knowledge about the dataflow graph is helpful to write RAPID programs with high performance.

RAPID operations can be divided into three types, namely *initial operations*, *transformations* and *finalization operations*. These three categories are the topic of the following sections. It should be noted that RAPID operations are described from a user's point of view in this chapter, i.e. as if they were actually applied to the data.

4.2.1 Initial Operations

This category contains operations that create a new RAPID from an ordinary container data structure, for example an array. The formal specification of the initial operations is shown in Table 4.1.

Parallelize is an overloaded operation with two variants. These variants differ in the way the input data is split. In the first variant of `parallelize`, data is divided

Table 4.1: Overview of initial operations

RAPID Operation	Argument Types
`parallelize(d,p,i,c): rapid⟨T⟩`	`d: collection⟨T⟩` `p: partitioning`

`parallelize(d,n,i,c): rapid⟨T⟩`	`n: integer`
	`i: identifier`
	`c: context`

`distribute(d,n,i,c): rapid⟨T⟩`	`d: collection⟨T⟩`
	`n: integer`
	`i: identifier`
	`c: context`

into partitions using an explicit, user-defined partitioning. All sizes in the given partitioning have to add up exactly to the input collection's size. Otherwise, an error is thrown. The second variant divides the input data into `n` partitions as evenly as possible for a given `n`.

Distribute creates a RAPID with `n` equally-sized partitions for a given `n`. In contrast to the first initial operation, `parallelize`, all partitions of the resulting RAPID contain the whole input data. This operation is particularly useful in combination with the `zipmap_partitions` operation if the computation of each output data element depends on some global data.

Users may pass a context argument to an initial operation. If no context is specified, the runtime environment's default context is used. This parameter allows users to specify the graph to which the operation belongs. For the other types of RAPID operations, a context parameter is unnecessary since the context can be derived from the data dependencies between RAPID operations.

The identifier argument is also optional. Whenever an identifier is specified, it has to be unique. Passing an identifier to an initial operation makes the corresponding input of the dataflow accessible. This allows a user to change the input data of a graph and start multiple dataflow executions with different data. Section 4.4 provides more details on this topic.

4.2.2 Transformations

Operations from this category create new RAPIDs from already existing RAPIDs. Most of the operations in this category are typical for functional programming languages. An overview of all transformations is given in Table 4.2.

Table 4.2: Overview of transformations

RAPID Operation	Argument Types

`map(r,f): rapid⟨U⟩`
`map(r,g): rapid⟨U⟩`
	`r: rapid⟨T⟩`
	`f: T → U`
	`g: T → U[x]`

`repartition(r,p): rapid⟨T⟩`
`repartition(r,n): rapid⟨T⟩`
	`r: rapid⟨T⟩`
	`n: integer`
	`p: partitioning`

`combine(r,f): rapid⟨U⟩`	`r: rapid⟨T⟩` `f: T[x] → U`
`zipmap(t,f): rapid⟨U⟩`	`t: rapid⟨T1⟩ × ⋯` `f: T1 × ⋯ → U`
`append(l): rapid⟨T⟩`	`l: list⟨rapid⟨T⟩⟩`

`split(r,n): list⟨rapid⟨T⟩⟩`
`split(r,l): list⟨rapid⟨T⟩⟩`
	`r: rapid⟨T⟩`
	`n: integer`
	`l: list⟨integer⟩`

`reorder(r,o): rapid⟨T⟩`
`reorder(r,f): rapid⟨T⟩`
	`r: rapid⟨T⟩`
	`o: list⟨integer⟩`
	`f: integer → integer`

`reduce(r,f): rapid⟨T⟩`	`r: rapid⟨T⟩` `f: T × T → T`
`map_partitions(r,f): rapid⟨U⟩`	`r: rapid⟨T⟩` `f: partition⟨T⟩ → partition⟨U⟩`
`zipmap_partitions(t,f): rapid⟨U⟩`	`t: rapid⟨T1⟩ × ⋯` `f: partition⟨T1⟩ × ⋯ → partition⟨U⟩`
`reorder_partitions(r,f): rapid⟨T⟩`	`r: rapid⟨T⟩` `f: integer → integer`

Map creates a new RAPID from an existing one by applying a function to each element in the given RAPID. Table 4.2 shows two variants of `map`. These two variants differ in the type of the given RAPID function. The RAPID function in the first variant creates one element of type `U` from one element of type `T`. Therefore, the partitioning of the RAPID returned by this variant of `map` matches exactly the given RAPID's partitioning. In the second variant, the given RAPID function creates `x` elements of type `U` for a single input element of type `T`. For a concrete RAPID function, the number of returned elements has to be firm at compile time. As a result, the number of elements in the result RAPID is a multiple of the number of elements in the given RAPID. The correlation also applies to the individual partitions of the result and given RAPID. This second variant of `map` is also known as *flatmap* in other frameworks.

Repartition constructs a new RAPID from a given one with the same elements but a different partitioning. This operation is very similar to `parallelize`, and therefore the two alternative forms of `repartition` correspond to the variants of the former. In the first variant, users may explicitly specify the new partition sizes, whereas they only need to specify the number of partitions in the second one. As with the second variant of `parallelize`, if only the desired number of partitions is given, elements are divided as evenly as possible. Some operations described later in this section have requirements on the given RAPIDs' partitionings. In case an input RAPID is not partitioned appropriately, `repartition` is called implicitly in these operations.

Combine is roughly the opposite of the second variant of `map`. Both operations differ in the type of function they expect as an argument. `Combine` functions create one new data element for a fixed number of input elements. If `x` is the number of elements that the given function requires as input, this function is applied to successive slices of `x` adjacent data elements. Therefore, the element count of all partitions in the given RAPID must be divisible by `x`. If the total number of elements in a RAPID is suitable, but the number of elements in some partitions is not divisible by `x`, `combine` automatically calls `repartition` beforehand.

Zipmap applies a given function element-wise on the given RAPIDs to create a new RAPID. This operations expects a tuple of RAPIDs and a `zipmap` function as arguments. The given function creates one new data element from multiple elements of different type. Most functional programming frameworks provide a zip operation, which creates a collection of tuples from two or more given collections. `Zipmap` represents a combination of a zip operation followed by a map operation. This operation is also known as *zipwith* in some frameworks. `Zipmap` requires at least that all input RAPIDs contain the same number of data elements. If the given RAPIDs' partitionings are not the same, `zipmap` calls `repartition` on all RAPIDs whose partitionings do not match.

Append takes a list of RAPIDs and creates a new RAPID which contains all partitions from the given RAPIDs in the order specified by the RAPID list. This operation does not create new data, nor does it alter the order of elements.

Split is contrary to `append`. It creates multiple RAPIDs, each containing a subset of the input RAPID's partitions. The two variants of `split` allow a user to specify either the partition count of each output RAPID or the number of output RAPIDs.

In the second case, partitions are divided as evenly as possible. Like `append`, `split` does not create elements or alter the element order.

Reorder changes the element order of a RAPID, while it leaves the partitioning and the data itself unchanged. Reordering elements over the boundaries of partitions is permitted. The two variants of `reorder` allow a user to specify the new element indices explicitly as a list or through a `reorder` function which calculates the new indices.

Reduce expects one RAPID and a function as arguments. The given `reduce` function creates one new data element of type `T` from two elements which are also of type `T`. This function is repeatedly applied to data elements in the given RAPID until only one element is left. The single result is encapsulated in a RAPID with one partition. Since the RAPID programming model does not specify in which order the `reduce` function is applied, users should only call `reduce` with associative functions.

Map_partitions, Zipmap_partitions and Reorder_partitions behave similar to `map`, `zipmap` and `reorder`, but operate on whole partitions rather than single elements. `Map_partitions` and `zipmap_partitions` expect RAPID functions as arguments which create new partitions from existing partitions. `Zipmap_partitions` may call `repartition`, if the given RAPIDs' numbers of partitions do not match. The last RAPID operation, `reorder_partitions`, changes the order of partitions in the given RAPID. The new partition indices are calculated with the given function.

4.2.3 Finalization Operations

Operations in this category allow a user to extract data from RAPIDs. The two finalization operations, `collect` and `finalize`, are shown in Table 4.3. These operations require identifiers as one of their arguments. As with initial operations, identifiers must be unique.

Table 4.3: Overview of finalization operations

RAPID Operation	Argument Types
`collect(r,i): context`	`r: rapid⟨T⟩` `i: identifier`
`finalize(r,i): collection⟨T⟩`	`r: rapid⟨T⟩` `i: identifier`

Collect adds an identifier to a RAPID and returns the corresponding context. A user can start the dataflow execution and get the data of the given RAPID later by calling a member function of the context with the same identifier (see Section 4.4).

Finalize is very similar to collect. The only difference is that `finalize` starts the dataflow execution immediately. Therefore, `finalize` returns a collection with all data elements from the given RAPID in the respective order.

4.2.4 RAPID operations example

To give a more practical view on RAPID operations, this section provides a small example program in the C++ reference implementation. The code is shown in Listing 4.1. At first, two C++ vectors are defined. Both are initialized with integer numbers from 1 to 10. Two calls of `parallelize` create the integer RAPIDs `r_1` and `r_2`. In both cases data is divided into four partitions. Since only an integer is provided, data is divided as evenly as possible, i.e. the first two partitions contain three integers each, while the last two partitions only contain two integers each. The given identifiers *input_1* and *input_2* can be used to re-execute the program later with two different integer vectors. With the following `zipmap` call, the user specifies that an element-wise addition is performed on the two specified RAPIDs. Since both RAPIDs have the same partitioning, no implicit repartitioning is required, and the returned RAPID `r` also adopts this partitioning. Lastly, a dataflow execution is started via a `finalize` operation call. For further executions, an identifier for the result (*output*) is given. At the end of the program, the result vector contains all even numbers from 2 to 20.

Listing 4.1: RAPID operations example

```
1  vector<int> vec_1 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2  vector<int> vec_2 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
3
4  rapid<int> r_1 = parallelize(vec_1, 4, "input_1");
5  rapid<int> r_2 = parallelize(vec_2, 4, "input_2");
6
7  rapid<int> r = zipmap({r_1, r_2}, add_function);
8
9  vector<int> result = finalize(r, "output");
10 // result is now {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
```

4.3 RAPID Functions

Most of the transformations specified in the last section expect a RAPID function as an argument. The purpose of RAPID functions from a user's perspective is to specify the behavior of RAPID operations. From an internal view however, RAPID functions affect the behavior of dataflow actors. When a RAPID function is applied to the partitions of a RAPID, the RAPID programming model does not specify an order. This ensures that partitions can be processed in parallel.

Table 4.2 shows simplified specifications of RAPID functions, e.g. `T` → `U` with arbitrary types `T` and `U` for a `map` function. In concrete implementations, however, RAPID functions must be structures which include additional information besides the actual function code.

Figure 4.2 shows the general RAPID function data structure with all possible members. The members in brackets are irrelevant for some specific types of RAPID functions (`map` functions, `zipmap` functions, ...). Each RAPID function object has a type, a name, an execution measurement function and an action function. The measurement function is a user-specified function which returns the estimated runtime of the action function for the given size. Depending on the implementation of the dataflow runtime environment, the execution measurement function may be of type `integer` × `integer` → `integer` or `integer` × `integer` → `real`. These measurement functions are used in the scheduling process (see Chapter 7). Partition-wise RAPID functions need to store an additional size function. Information about this function follows in Section 4.3.2. The exact type of the last member, the action function, depends on the specific type of RAPID function. This member contains the main functionality of a RAPID function. Therefore, the execution of a RAPID function corresponds to an execution of its action function.

operation_function
`type: rapid_function_type`
`name: identifier`
`measure: function`
`[size: function]`
`action: function`

Figure 4.2: RAPID function data structure

RAPID function objects cannot be created directly with a constructor. Instead, users have to define them using the following syntax:

```
rapid_function(name, type, action, [size,] measure);
```

It is recommended to use the function definition syntax only in a static context and with global visibility. This ensures that the RTE has an overview of all RAPID functions already at system initialization. Users have to ensure that RAPID function names are unique. Further, the type in a RAPID function definition must be one of the following:

- `map_t`
- `combine_t`
- `zipmap_t`
- `reduce_t`
- `map_partitions_t`
- `zipmap_partitions_t`
- `reorder_t`
- `reorder_partitions_t`

The other parameters (action, size and measure) depend heavily on the RAPID function's type. These three are usually function-type parameters, only the measurement parameter is of elementary type for some types of RAPID functions. For function-type parameters, users may pass either a function pointer or a lambda expression. To ensure support of different hardware architectures, the given function or lambda expression must be pure, i.e. its output only depends on the respective input. Therefore, it is not allowed for the given function (or lambda expression) to access arbitrary memory locations. The additional size parameter is only necessary for partition-wise functions. More details about the different types of RAPID functions follow in the next sections.

4.3.1 Element-wise Functions

Element-wise RAPID functions are used in `map`, `combine`, `zipmap` and `reduce` operations. These functions are applied individually to the elements in the input RAPID (or tuples of elements from different RAPIDs in case of `zipmap`). The RAPID programming model does not specify an order for the application of element-wise function to the elements in a partition. Therefore, element-wise functions may be executed in parallel to create the result elements.

Further, element-functions do not require the size parameter, since the number of created elements follows directly from the function definition, and the element size follows from the result element type. The execution measurement parameter in element-wise functions expects a function with one parameter. Depending on the implementation, this function may be of type `integer` → `integer` or `integer` → `real` and represents the estimated runtime for a single function execution on the processing element with the given index. To estimate the time it takes to apply the function to a partition, this value is multiplied by the number of sequential function executions during offline scheduling.

Map functions appear in two different forms which differ in the type of their action function. The two possible types are `T` → `U` and `T` → `U[x]`, in which `T` and `U` are types and `x` is an integer greater than one. Therefore, one execution of a `map` function may create exactly one or a fixed number of elements of type `U` from an element of type `T`. In case multiple elements are created, these elements will appear next to each other in the result RAPID.

Combine functions take multiple elements and create one new element. Therefore, their action functions' type is `T[x]` → `U` for types `T` and `U` and an integer `x` greater than one. When applied to a RAPID with n elements, the action function is executed $m = \frac{n}{x}$ times to create a RAPID with m elements.

Zipmap functions also create one element out of multiple elements. But in contrast to `combine` functions, these functions take elements from different RAPIDs, and so their inputs may be of different type. The action function's type is `T1` × `T2` × ⋯ × `Tn` → `U`, in which `T1`, `T2`, ..., `Tn` and `U` are types and `n` is an integer greater than one.

Reduce functions always take two elements and create one new element. All elements must be of the same type. Thus, in case of `reduce` functions, the action function's type is `T` × `T` → `T` for a type `T`. `Reduce` functions are repeatedly applied on the elements of a RAPID until only one element is left. Since the RAPID programming model does not specify the order in which the function is applied on the RAPID's elements, users should only define associative `reduce` functions.

4.3.2 Partition-wise Functions

These RAPID functions take whole partitions as inputs and return new partitions. Only `map_partitions` and `zipmap_partitions` functions fall into this category. Partition-wise functions may create partitions with an arbitrary number of elements, but the RAPID programming model requires that dataflow graphs contain size information for all partition nodes. For this reason, users have to specify a size function as an additional argument in the RAPID function definition. Its exact type is different for the two types of partition-wise functions.

To give the user the possibility to specify the estimated execution time of element-wise functions more precisely, the measurement parameter expects a function of type `integer` × `integer` → `integer` or alternatively `integer` × `integer` → `real` depending on the RTE implementation. This function is supposed to return the

estimated runtime based on the number of created elements in the result partition and the processing element executing the function.

Map_partitions functions create a new partition Q from an existing partition P. Action functions are of type `partition⟨T⟩ → partition⟨U⟩`, in which `T` and `U` are types. As mentioned above, partitions P and Q do not need to contain the same number of elements, so users have to specify a size function in the RAPID function definition. This size function is of type `integer → integer` and returns the number of elements in Q based on the number of elements in P.

Zipmap_partitions functions are similar to `map_partitions` functions. The difference is that `zipmap_partitions` functions have multiple inputs. Action functions are of type `partition⟨T1⟩ × ⋯ × partition⟨Tn⟩ → partition⟨U⟩` with types `T1`, ..., `Tn` and `U` and an integer `n` which is greater than one. Like the action function, the size function also has `n` parameters. Therefore, the type is `integer × ⋯ × integer → integer`. This allows users to define `zipmap_partitions` functions whose result partitions depend on the size of all partitions given as arguments. An example would be a RAPID function creating a partition with an element count equal to the sum of the arguments' element counts. In this example, the size function would be $f(x_1, x_2, \ldots, x_n) = x_1 + x_2 + \cdots + x_n$.

4.3.3 Reordering Functions

The last category of RAPID functions contains reordering functions. These functions are used in `reorder` and `reorder_partitions` operations. All reordering functions take an index and return a new index, but there are additional parameters which are different for the two types of functions.

Since the reordering of elements does not depend on actual data, the time for a single index calculation will be similar for all index values in most cases. Hence, the measurement parameter in the function definition expects a function with one parameter, as in the case of element-wise functions.

Reorder functions calculate new element indices. The action function has, besides the source element index, three additional parameters which give users the possibility to reorder elements depending on various other properties. Additional parameters are the source RAPID's size, the index of the partition which contains the currently processed element and the partitioning of the source RAPID. Therefore, the action function's type is `integer × integer × integer × partitioning → integer`.

Reorder_partitions functions are similar to `reorder` functions and calculate partition indices. But unlike element-based reordering functions, action functions of `reorder_partitions` functions only have one additional parameter. This parameter allows users to change the partition order based on the source RAPID's number of partitions. This causes the action function's type to be `integer` × `integer` → `integer`.

4.3.4 RAPID Functions Example

This section continues the example from Section 4.2.4 and highlights the RAPID function `add_function` in more detail. Listing 4.2 shows the exact definition in the C++ reference implementation. A RAPID function definition always begins with the keyword `rapid_function` followed by a list of arguments enclosed by parentheses. The first argument is the RAPID function's name, in this example `add_function`, which is followed by the type (`zipmap_t`). Third, a lambda expression specifies how the data is processed. A RAPID function with type `zipmap_t` always requires a lambda expression (or pointer to a function) with at least three parameters, one reference representing the result and two or more references to constant values representing the inputs. It is important that a lambda expression passed to the RAPID function definition does not capture any values, as shown in Listing 4.2 by the empty brackets. The lambda expression in the example writes the sum of the two inputs to the output reference. The last argument of the function definition is an integer which can be used by scheduling heuristics. It says that one execution of the given lambda expression takes ten time units. How this value is interpreted depends on the exact scheduling algorithm.

Listing 4.2: RAPID function example

```
1 rapid_function(
2   add_function,
3   zipmap_t,
4   [](int& out, const int& in1, const int& in2) {
5     out = in1 + in2;
6   },
7   10
8 );
```

4.4 RAPID Context

As already mentioned in the previous sections, RAPID operations are not executed immediately, but instead a dataflow graph is built. Contexts represent the high-level concept that allows users to access dataflow graphs to a limited extent. The context data structure can keep track of one dataflow graph and multiple pre-computed schedules so that the graph can be executed with different redundancy configurations.

An overview of the provided functions is shown in Figure 4.3. Users may create any number of contexts with the standard constructor. Contexts which were created like this are initialized with an empty dataflow graph. To add nodes to the dataflow graph, users need to pass the corresponding context to a `parallelize` or `distribute` operation. Other RAPID operations deduce the context from their input RAPIDs. If the contexts of RAPIDs passed to an operation with multiple inputs do not match, an error is thrown.

The RAPID programming model does not forbid contexts with dataflow graphs consisting of multiple unconnected subgraphs. Since graphs are scheduled and executed as a whole, users may take advantage of this to create one graph from two or more independent computations, which are then executed concurrently.

context
`context()`
`actor_count(): integer`
`input_count(): integer`
`result_count(): integer`
`export_graph(l: identifier, i: identifier, f: format)`
`export_schedules(l: identifier, i: identifier, f: format)`
`import(l: identifier)`
`set_input(i: identifier, c: collection⟨T⟩)`
`get_result(i: identifier): collection⟨U⟩`
`move_result(i: identifier, j: identifier)`
`execute()`
`checkpoint()`
`schedule(a: scheduling_algorithm)`
`completed_section_count(): integer`
`set_section_redundancy(i: integer, c: criticality)`
`section_redundancy(i: integer): criticality`

Figure 4.3: Functionality provided by contexts

Chapters 6 and 7 provide more details on dataflow executions and scheduling respectively.

Contexts provide various member functions for users to get information about the dataflow graph. Users may check graph metadata, like the number of actor nodes or number of input nodes in the graph, with the corresponding count functions. Furthermore, users can access the import/export functionality through the context. The `export_graph` function stores a graph in the given graph description format `f` at a location specified by the given identifier `l`. How `l` is interpreted depends on the RTE implementation. As an example, `l` could be interpreted as a file name if the RTE runs on an operating system with a file system. With the second identifier `i`, a name can be assigned to the graph. The C++ reference implementation supports the *DOT graph description language* [GKN15] with some custom node attributes. Exported graphs are compatible with many graph visualization tools since these tools often ignore unknown attributes. Additionally, the context provides an `import` function to construct graphs from a representation in a suitable format. As for the dataflow graph, the static schedules of a context can also be imported and exported. The `import` function automatically detects whether the loaded entity describes a graph or schedule. For schedules, a simple custom format with a syntax similar to exported graphs is used in the reference implementation.

4.4.1 Repeated Dataflow Executions

After a dataflow graph was constructed, users may start the dataflow execution multiple times with different data. For repeated dataflow executions, contexts provide a small set of functions, for example `set_input` and `get_result`. The `set_input` function expects an identifier and a data collection as its arguments. A given identifier must match one of the identifiers used in a `parallelize` or `distribute` operation, otherwise an error is thrown. Another error is thrown if the given collection's size does not match the expected input size. Graph inputs which have not been specified with the `set_input` function keep their data from the previous dataflow execution. If no identifier has been specified in an initial operation call, there is no way to change the data and the initially specified data is used in all dataflow executions.

Repeated dataflow executions may be started either with the `execute` function or the `get_result` function, which additionally returns a collection of result data. As with `set_input`, the latter expects an identifier as its argument. This identifier must have already appeared as an argument in a `collect` or `finalize` operation or otherwise an error occurs. An important property of the RAPID programming model is that graphs are always executed as a whole. This means that not all calls of `get_result` start a dataflow execution. If a context contains multiple results which

are all requested through subsequent `get_result` calls with different identifiers, only the first call starts a dataflow execution. All other calls just return the corresponding data collections. In general, whenever `get_result` or `execute` is called on an already executed graph, dataflow execution is only started once again if some input data changed since the last execution.

4.4.2 Checkpoints

One of the RAPID programming model's design goals was to support fault-tolerant dataflow executions through redundant actor execution. Furthermore, it should be possible to change the degree of redundancy at runtime while the system keeps following pre-computed schedules. However, changing the redundancy of a single dataflow actor makes the graph's schedule obsolete. For example, if the degree of redundancy is changed from a single execution to two redundant executions for some actor, the schedule does not consider the additional actor execution. The insertion of new actors into an existing schedule at runtime has multiple drawbacks. For large graphs with many data dependencies, checking these dependencies and finding an appropriate place to insert an actor is not trivial and the overhead at runtime may get quite large, especially if actor redundancies change often. It is also not feasible to create one schedule for each combination of redundant actors. For n actors and x possible redundancy configurations, the number of schedules would be x^n.

To solve the described problems, the RAPID programming model allows users to divide dataflow graphs into sections. In a graph section, all actors must be executed with the same redundancy and changing the redundancy is only possible for sections as a whole. This restriction leads to a much simpler scheduling. Chapter 7 highlights this topic in more detail.

The first section is always created along with the graph. For additional sections, the graph's corresponding context provides a member function called `checkpoint`. This function requires no arguments. A `checkpoint` function call completes the current section and creates a new one. New actors are always inserted into the section that was created last. To avoid empty sections, `checkpoint` function calls are ignored, if the current section is empty. Users may check into how many sections a graph is divided with the `completed_section_count` function.

Graph sections are referred to by index in order of their creation, with the first section's index being zero. By passing a section index to member functions of a context, users are able to influence dataflow executions of the corresponding graph. The `set_section_redundancy` function allows users to manually specify the redundancy of sections. Changing the section redundancy is only possible for completed sections. Therefore, if this function is called with the currently incomplete

section's index, `checkpoint` is called implicitly. It is also possible to check the current redundancy of a section by passing the section's index to `section_redundancy`.

Whenever a dataflow graph is executed in offline scheduling mode, each non-empty graph section needs a schedule. Because the RAPID programming model only permits complete sections to be scheduled, the `checkpoint` function is called implicitly when a scheduling heuristic is called while the current section is not empty. The offline scheduling process is started implicitly when a not fully scheduled graph is about to be executed in offline scheduling mode. Users may also trigger the scheduling manually with the `schedule` member function. By doing so, it is possible to specify the desired scheduling algorithm (while otherwise the RTE's standard scheduling heuristic is used). Already scheduled sections remain the same, even if they were scheduled with another scheduling algorithm. By calling the `schedule` function right after the creation of each section, users are able to specify different scheduling algorithms for each individual section. The function can also be used to switch between the use of pre-computed schedules and online scheduling. Calling the function with an online scheduling approach does not trigger a scheduling heuristic, but instead marks the graph so that it is executed using online scheduling. More information about online scheduling follows in Chapter 7.

4.4.3 Multiple Contexts

When users implement an application in the RAPID programming model, they have to decide whether to use one single context or a set of contexts. Most programs and algorithms consist of one or more loops. In such cases, users have the choice to either express one or multiple iterations of the outer loop through one context or use a single context containing the whole RAPID program. For dataflow graphs executed in a loop, the context provides the function `move_result`, which can be used to express that the output of a dataflow execution should be an input in the next execution.

An example is depicted in Figure 4.4. The figure shows a generic (non-RAPID) program starting in an initial state. After executing a sequence of instructions, the program enters an intermediate state, which is revisited after each loop iteration. The program then goes into its final state. Programs following this structure may be implemented in the RAPID programming by using three contexts, as shown in the figure. When the RAPID program is executed, the dataflow graphs of `c1` and `c3` are executed once, while the graph of `c2` must be executed n times if it represents one loop iteration. Another possibility would be to build up the dataflow graph of `c2` so that it represents x loop iterations, where n is divisible by x. In this case `c2` would have to be executed only $\frac{n}{x}$ times.

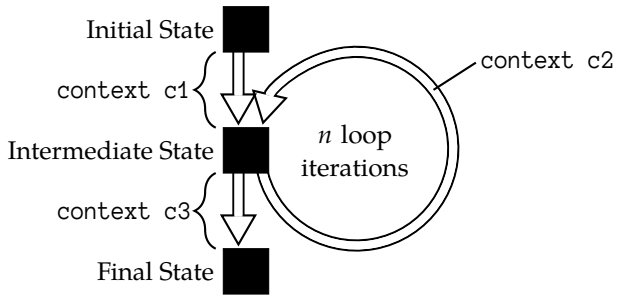


Figure 4.4: Using three contexts to express a program with one main loop

Each approach has its own advantages and disadvantages. Using a single context for the whole program can lead to very large dataflow graphs since RAPID dataflow graphs are acyclic and loops in the RAPID program are therefore automatically unrolled in the graph. This leads to higher memory demand and longer offline scheduling times due to the larger number of data dependencies. The advantage of a single context is that DAG scheduling heuristics have an overview of the whole program and may create schedules with lower makespan. Further, the completion of a dataflow execution is an implicit barrier, so a single context does not lead to barriers in the program execution like multiple contexts do.

4.4.4 Context Example

This section provides an example on how contexts can be used in RAPID programs. Therefore, it extends the small program from Section 4.2.4. The code is shown in Listing 4.3. Besides the two vectors `vec_1` and `vec_2`, a context `cx` is declared. This context is passed to the two calls of `parallelize`. Following RAPID operations can automatically infer the context from their arguments because RAPIDs keep track of the context they belong to. In contrast to previous examples, graph construction is concluded by a `collect` operation, which does not start a dataflow execution. After that, a function of the context is used to specify how the graph is scheduled, in this example with the HEFT scheduling heuristic. The scheduled graph is then executed two times with different data. Dataflow executions are started via the `get_result` calls. In the first graph execution, the two initial vectors are processed. For the second dataflow execution, the result of the first execution is used as the first input, while the second input is not changed, so that the data of `vec_2` is used once again. It should be noted that this example only shows the most common functions the context provides. The code in this example is not optimal, since it calls the `set_input` method in order to use the result of a dataflow execution as the next input. This leads to a copy of the result data. To circumvent the copy, the context provides the `move_result` function, which should be used preferably in such cases.

Listing 4.3: Context example

```
1  vector<int> vec_1 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2  vector<int> vec_2 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
3  context cx;
4
5  rapid<int> r_1 = parallelize(vec_1, 4, "input_1", cx);
6  rapid<int> r_2 = parallelize(vec_2, 4, "input_2", cx);
7  rapid<int> r = zipmap({r_1, r_2}, add_function);
8  collect(r, "out");
9
10 cx.schedule(static_heft);
11
12 vector<int> result = cx.get_result<int>("output");
13 // result is now {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
14
15 cx.set_input("input_1", result);
16 result = cx.get_result<int>("output");
17 // result is now {3, 6, 9, 12, 15, 18, 21, 24, 27, 30}
```

4.5 Characteristics of the C++ Reference Implementation

After the previous sections already showed examples in the C++ reference implementation, this section provides additional details. The RAPID programming model benefits from many features of modern C++ standards. For the reference implementation C++14 was chosen because compilers for embedded architectures are in their development often behind standard x86 compilers and do not support the latest C++ standards. RAPIDs are implemented as class templates and RAPID operations as function templates. This allows the RTE to fully utilize the type deduction capabilities of C++14 so that users almost never have to specify template arguments. Furthermore, the exact type of a RAPID can always be inferred from the RAPID operation returning it. Therefore, users may use the keyword `auto`, whenever they declare a RAPID. The only cases where the exact types must be specified are the definition of RAPID functions and `get_result` function calls.

The reference implementation of the RAPID programming model was designed to serve as a common basis for dataflow environments on different architectures. Therefore, it consequently assigns unique IDs to RAPID functions. The reason behind this is that in systems with distributed memories the same RAPID function

appears in multiple memories at possibly different memory locations. Using unique IDs in the communication between cores with their own local memories ensures that both cores refer to the same function. Since RAPID functions do not have any particular order, they cannot be numbered consecutively. Therefore, the C++ reference implementation applies a hash function on the RAPID function's name (which is always available in the source code) to generate a unique ID which is consistent even over multiple compilations. This is important for distributed hardware architectures in which processors may execute different binaries. *SipHash*, a non-cryptographic hash function, was chosen for this purpose because of its simple implementation and speed [AB12]. As with all hash functions, there is a small chance that two different RAPID functions get the same ID. To avoid errors during runtime, the uniqueness of all IDs is checked at system initialization.

4.6 Example Applications

The RAPID programming model allows users to implement many common applications from the domain of high-performance embedded computing. However, not all programs are equally suitable. Applications that use large complex data structures like graphs or trees are the most problematic. Such structures are usually difficult to split into smaller parts in a sensible way. Using one single partition for the whole data structure is possible, but has the disadvantage that it does not allow parallel modifications. This leaves only such applications which solely need to read from the data structure, for example tree searching algorithms. But on distributed architectures with small local memories, putting a large data structure in one partition might not be possible in the first place.

Since the RAPID programming model was designed with distributed architectures in mind, it is not surprising that applications which are well suited for such architectures also work well with the RAPID programming model. The following sections describe a selection of common tasks in the high performance embedded domain and how they can be implemented in the RAPID programming model, using the C++ reference implementation as an example.

4.6.1 Matrix Multiplication

Matrix multiplication can be implemented in various ways. For an implementation in the RAPID programming model, *Cannon's algorithm* [Can69, p. 22–27] is a good choice. In its original form, Cannon's algorithm is a distributed algorithm for the multiplication of two $n \times n$ matrices on n^2 processors. The main idea behind the algorithm is to shift the rows of the first matrix and columns of the second matrix in

each step, multiply the matrices element-wise and add the result to an accumulator matrix. Implementing the original algorithm in with RAPID operations would lead to very small partitions (one data element per partition), a large dataflow graph and a fine-grain dataflow. To reduce overhead in the dataflow execution and to decrease the graph's size, a common generalization of Cannon's algorithm can be applied. This generalization replaces matrix elements in the algorithm with submatrices, and scalar additions and multiplications with matrix additions and multiplications.

In Algorithm 4.1 the steps to perform the generalized Cannon's algorithm are shown. If X is a matrix, its submatrices are denoted with $X(i,j)$ analog to the matrix elements ($X_{i,j}$). Splitting the matrices into s^2 parts (line 2) corresponds to a `parallelize` operation in the RAPID programming model. But splitting alone does not lead to appropriate partitions. Matrix elements are initially stored in a row-like fashion, which means that partitions contain rows (or parts of rows) instead of square submatrices. Therefore, elements of input matrices have to be rearranged with `reorder` operations. The following initial shifting of submatrices can be done with `reorder_partitions` operations. Block-wise matrix multiplications in the main loop (lines 9 to 13) may be easily expressed with a `zipmap_partitions` operation. The rest of the main loop consists of shifting block rows and columns of

Algorithm 4.1: Generalized Cannon's algorithm

```
1  function MATMUL(A : n × n, B : n × n) : n × n
2      split matrices A, B and C into s² square blocks of size n/s × n/s;
3      initialize all elements of C with zeros;
4      for i from 1 to s do
5          shift block row i of matrix A by i block columns to the left circularly;
6          shift block column i of matrix B by i block rows upwards circularly;
7      end
8      for x from 1 to s do
9          for i from 1 to s do
10             for j from 1 to s do
11                 C(i,j) ← C(i,j) + A(i,j)B(i,j);
12             end
13         end
14         shift block row i of matrix A by 1 block column to the left circularly;
15         shift block column i of matrix B by 1 block row upwards circularly;
16     end
17     return C;
18 end
```

matrices A and B. Since each partition contains one submatrix, shifting the rows or columns block-wise corresponds to reordering the partitions. After the main loop, the result matrix C is stored in a block-wise fashion. Analogous to the initial element shuffling, the row-wise element order can be restored with a `reorder` operation.

Listing 4.4 shows Cannon's algorithm implemented in the RAPID programming model. For optimization purposes, the implementation differs slightly from the descriptions above. First, reordering the elements to blocks of submatrices and shifting these submatrices is done in one step by the `reorder` operations in lines 6 and 7. The second difference is that the RAPID `r` which corresponds to matrix C in Algorithm 4.1 is not initialized with zeros. Instead, the RAPID is initially created by the first block-wise multiplication (line 8). Another important aspect about this example is that using the `persist` function of RAPIDs is required. The matrices must be persisted in each loop iteration since they are used in the block-wise multiplication as well as the following block-wise reordering.

Listing 4.4: Cannon's algorithm implemented with RAPID operations

```
1  int s;
2  std::vector<int> matrix1, matrix2;
3    ⋮ // initialize s, matrix1 and matrix2
4  auto m1 = parallelize(matrix1, s*s, "matrix_1");
5  auto m2 = parallelize(matrix2, s*s, "matrix_2");
6  m1 = reorder(m1, reorder_first).persist();
7  m2 = reorder(m2, reorder_second).persist();
8  auto r = zipmap_partitions({m1, m2}, multiply);
9  for (int i = 0; i < s - 1; ++i) {
10   auto nm1 = reorder_partitions(m1, rotate_left);
11   auto nm2 = reorder_partitions(m2, rotate_up);
12   m1 = nm1.persist();
13   m2 = nm2.persist();
14   r = zipmap_partitions({r, m1, m2}, multiply_add);
15 }
16 r = reorder(r, reorder_reverse);
17 auto result = finalize(r, "output");
```

4.6.2 Fast Fourier Transform

Fast Fourier transform (FFT) is an efficient way to compute the discrete Fourier transform (DFT). It converts a discrete signal from its original domain into a representation in the frequency domain. The most common algorithm to compute

the FFT is the *Cooley-Tukey algorithm* [CT65], which follows a divide-and-conquer approach. In its simplest form, the algorithm can only be applied to sequences whose size is a power of two. The basic idea is to split the input sequence in half, compute the DFT of both halves individually and combine the results of both halves into the full DFT. Applying this method to the two halves recursively leads to algorithm in $O(n \log n)$. It is also possible to implement the algorithm in an iterative fashion. The specification of such an iterative Cooley-Tukey algorithm is shown in Algorithm 4.2. Since the RAPID programming model does not support recursion, this version is preferable as a starting point for an implementation with RAPIDs. In contrast to the original algorithm which implicitly changes the element order with each recursive function call, the iterative version requires to reorder the elements explicitly. Changing the element order only once, either before or after the main loop, is sufficient. The iterative variant in Algorithm 4.2 is the basis of the following RAPID FFT implementation.

Listing 4.5 shows the implementation of the described iterative Cooley-Tukey algorithm in the RAPID programming model. The first `parallelize` operation takes a vector of complex numbers and an integer `p` as its arguments. The given `p` has to be a power of two and must be less than the given vector's size. Since the algorithm is based on the iterative variant, the next RAPID operation is changes the element order. After the reordering, a `map_partitions` operation is used to calculate the FFT for each partition individually. Therefore, the given RAPID function `fft_func`

Algorithm 4.2: Iterative Cooley-Tukey algorithm

```
1  function FFT(S : complex[n]) : complex[n]
2      reorder S by reversing the bits of each element's index;
3      for l from 1 to log(n) do
4          m ← 2^l;
5          for k from 0 to n − 1 step m do
6              for j from 0 to m/2 − 1 do
7                  t ← S[k + j];
8                  S[k + j] ← t + exp(−2πij/m) S[k + j + m/2];
9                  S[k + j + m/2] ← t − exp(−2πij/m) S[k + j + m/2];
10             end
11         end
12     end
13     return S;
14 end
```

Listing 4.5: Fast Fourier transform implemented with RAPID operations

```
1  vector<complex<double>> input;
2  int p;
3    ⋮ // initialize input and p
4  auto r = parallelize(input, p, "input");
5  r = reorder(r, shuffle_bit_reverse);
6  r = map_partitions(r, fft_func);
7  for (int i = 0; i < log2(p); ++i) {
8    auto [r1, r2] = divide_rapid(r, i);
9    auto ri = distribute(vector<int>{i}, p/2);
10   ri.persist();
11   auto r3 = zipmap_partitions({r1, r2, ri}, fft_left);
12   auto r4 = zipmap_partitions({r1, r2, ri}, fft_right);
13   r = merge_rapids(r3, r4, i);
14 }
15 auto result = finalize(r, "output");
```

is equal to a standard iterative Cooley-Tukey algorithm. The main loop (which corresponds to the outer loop in Algorithm 4.2) starts in line 7. At the beginning of each loop iteration, two RAPIDs `r1` and `r2` are created by a function called `divide_rapid`. This function itself is not a RAPID operation, but it internally calls the RAPID operations `split` and `append`. Its purpose is not to create new partitions but to rearrange existing partitions so that the dataflow graph reproduces the typical butterfly pattern of the algorithm's data dependencies. The function also calls persist for the two RAPIDs it returns. In Algorithm 4.2 the loop variables k and j are responsible for creating this pattern as they serve as indices in the computation (lines 7 to 9). The actual computation in the loop is implemented with two `zipmap_partitions` operations. These RAPID operations separately compute interim results for half of the elements. The respective lines in Algorithm 4.2 are lines 8 and 9. Since the FFT computation depends on the loop index, in each iteration an additional RAPID `ri` containing only one integer is required. This RAPID is created by the `distribute` operation in line 9. In the end of the loop, a function `merge_rapids`, which is inverse to the function in line 8, is used to restore the original partition order.

4.6.3 Sorting

Since sorting is a very common task in applications, many sorting algorithms were developed. Sorting algorithms can be divided into two categories, namely

comparison-based and non-comparison-based sorting algorithms. In the following, we focus on comparison-based sorting. A suitable algorithm for an implementation in the RAPID programming model is *bitonic sorting* [Bat68]. In contrast to other common sorting algorithms, like *Mergesort* or *Heapsort*, bitonic sorting is not asymptotically optimal, since the number of required comparisons is in $O\left(n \log^2(n)\right)$, instead of $O(n \log(n))$. However, bitonic sorting is intrinsically parallel and scales very well with an increasing number of processors. As the name suggests, the algorithm is based on bitonic sequences. A sequence of n keys $x_1, x_2, \ldots x_n$ is a bitonic sequence, if $x_1 \leq x_2 \leq \ldots \leq x_i \geq \ldots \geq x_n$ for some i with $1 \leq i \leq n$ or if it is a cyclic shift of such a sequence.

A pseudo-code implementation of bitonic sorting is shown in Algorithm 4.3. This most common form of bitonic sorting requires that the length of the input sequence

Algorithm 4.3: Bitonic sorting

```
1  function BITONIC_SORT(S : key[n]) : key[n]
2      if n = 1 then
3          return S;
4      end
5      A ← BITONIC_SORT(S[1 ... ⌈n/2⌉]);
6      B ← BITONIC_SORT(S[⌈n/2⌉ + 1 ... n]);
7      reverse the element order of B;
8      return BITONIC_MERGE(CONCAT(A, B));
9  end

10 function BITONIC_MERGE(S : key[n]) : key[n]
11     if n = 1 then
12         return S;
13     end
14     m ← n/2;
15     for i from 1 to m do
16         if S[i] > S[i + m] then
17             SWAP(S[i], S[i + m]);
18         end
19     end
20     S_1 ← BITONIC_MERGE(S[1 ... m]);
21     S_2 ← BITONIC_MERGE(S[m + 1 ... n]);
22     return CONCAT(S_1, S_2);
23 end
```

is a power of two. The basic idea behind this algorithm is to sort the first half of a sequence ascending and the second half descending so that the resulting sequence is bitonic (lines 5 to 7 in Algorithm 4.3). Both halves are then merged together to form an ordered sequence. The algorithm only needs to specify how the merge step is performed since the sorting of the two halves can be done via recursion. Merging is also a recursive function. After the merge was applied to the sequence as a whole (lines 15 to 19 in Algorithm 4.3), it is called recursively on the two halves of the sequence.

The recursion of bitonic sorting does not depend on element values but only on the length of the given sequence. Hence, for a given sequence size, it is possible to build a sorting network. Figure 4.5 shows an example network with eight keys in form of a *Knuth diagram* [LB00]. In this diagram, horizontal lines represent element positions in a sequence, with the input sequence being left and the output sequence being right. Vertical arrows originating from black dots are comparators whose arrow tips point to the element position, where the larger element of the two goes. The gray boxes mark the three recursive steps, i.e. the sorting of the upper and lower half and the merge phase afterwards.

Since sorting networks statically specify all data dependencies, they could be considered as dataflow graphs. However, it is not advisable to write a RAPID program which creates a dataflow graph with an explicit edge for each comparison in the sorting network. In such a RAPID program, each partition would only contain a single key. This would lead to a very large graph and fine-grain dataflow, which would increase the overhead during runtime. To avoid overly large graphs, RAPIDs must be divided into less but larger partitions. With bitonic sorting, this is easily possible because of the recursive nature of the algorithm, which leads to recurrent structures in the sorting network. If the bitonic sorting network can sort n keys, valid partition sizes for a RAPID program are all powers of two less than n. Comparisons are then carried out on pairs of partitions through `zipmap_partitions` operations.

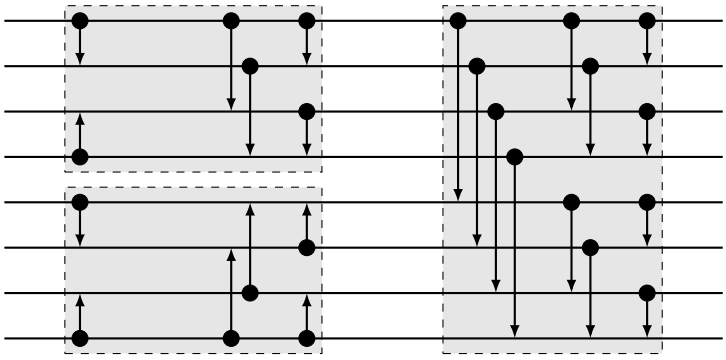


Figure 4.5: Knuth diagram of bitonic sorting with eight elements

If elements are only compared with elements inside their respective partition, a `map_partitions` operation has to be used instead. By following this approach, the overall structure of the RAPID program remains almost the same. Only the resulting dataflow graph resembles a smaller-scaled version of the data-dependencies in the sorting network.

Based on these considerations, an implementation of bitonic sorting with RAPID operations is shown in Listing 4.6. Since the data dependencies in bitonic sorting networks are similar to those in the Cooley-Tukey algorithm, the two RAPID programs also show similarities. As with FFT, the two divide functions (lines 7 and 12) are not RAPID operations themselves but call operations like `split` and `append` in order to produce the RAPIDs `r1` and `r2`. This division is done in a way that applying two `zipmap_partitions` operations to these two RAPIDs leads to the desired butterfly patterns in the dataflow graph. These butterfly patterns are also visible in the sorting net in Figure 4.5. Compared to FFT, bitonic sorting requires more butterfly pattern layers. This is the reason why there is an additional inner loop. Because the inner and outer loop produce similar data dependency patterns in the dataflow graph, the statements in the body of the inner loop closely resemble those in the body of the outer loop. At the end of the outer loop, there is an additional `map_partitions`

Listing 4.6: Bitonic sorting implemented with RAPID operations

```
1  vector<int> input;
2  int p;
3    ⋮ // initialize input and p
4  auto r = parallelize(input, p, "input");
5  r = map_partitions(r, bitonic_inner_1);
6  for(int i = 0; i < log2(p); ++i) {
7    auto [r1, r2] = divide_rapid_1(r, i);
8    auto r3 = zipmap_partitions({r1, r2}, bitonic_lower_1);
9    auto r4 = zipmap_partitions({r1, r2}, bitonic_upper_1);
10   r = merge_rapids_1(r3, r4, i);
11   for(int j = i - 1; j >= 0; j--) {
12     [r1, r2] = divide_rapid_2(r, j);
13     r3 = zipmap_partitions({r1, r2}, bitonic_lower_2);
14     r4 = zipmap_partitions({r1, r2}, bitonic_upper_2);
15     r = merge_rapids_2(r3, r4, j);
16   }
17   r = map_partitions(r, bitonic_inner_2);
18 }
19 auto result = finalize(r, "output");
```

call (line 17). As mentioned above, this RAPID operation is used in the algorithm whenever only elements in the same partition are compared. In contrast to the first use of `map_partitions` in line 5, where all partitions are completely unordered, this second `map_partitions` operation is applied to partially ordered data and thus requires fewer comparisons to sort the elements.

4.7 Summary

This chapter introduced a programming model similar to big data frameworks like Apache Spark, which allows users to write dataflow programs via functional-style operations. In contrast to other programming models, the set of operations is reduced since it targets safety-critical embedded systems. These operations are applied to partitioned collections of data elements called RAPIDs. From a user's perspective, RAPID operations are lazily evaluated functions. Internally, however, operations build a dataflow graph. Via the execution context, users can access the graph to a limited extent. An important functionality of the context is the `checkpoint` function. This function is used to divide graphs into sections. The advantage of multiple sections is that the redundancy configuration of each section can be changed individually. Other member functions of the context allow users to access inputs and outputs of the graph and repeatedly start dataflow executions. The next chapter describes the basic structure of RAPID dataflow graphs and how graphs are built from RAPID operations and their data dependencies.

5

RAPID Dataflow Graphs

Because knowledge of dataflow graphs and their relation to RAPID operations is essential for implementing efficient RAPID programs, dataflow graphs were already mentioned in the previous chapter. This chapter specifies the structure of RAPID dataflow graphs in more detail and describes how dataflow graphs are built from RAPID operations.

It should be noted that RAPID programs are supposed to be executed on standard, control-flow-based processors, while traditional dataflow models are often designed for special dataflow processors. Because some characteristics of these dataflow models would induce overhead if they were applied to standard processors, dataflow graphs in the RAPID programming model are closer to the graphs in modern larger-scale computing frameworks.

The sections of this chapter cover the following topics. Section 5.1 describes the different nodes of a RAPID graph in more detail. Graph nodes are constructed and connected by RAPID operations. The construction rules are specified in Section 5.2. In some situations, nodes can be removed from a graph during construction for optimization purposes. A selection of such optimizations is presented in Section 5.3. Some RAPID operations are already partially executed when the graph is created. How those operations can be handled efficiently is proposed in Section 5.4. After that, Section 5.6 describes characteristics of RAPID dataflow graphs in the C++

reference Implementation. Lastly, Section 5.7 revisits the RAPID applications from Section 4.6 and shows the resulting dataflow graphs.

5.1 Graph Nodes

RAPID dataflow graphs are bipartite and consist of two types of nodes, namely *partition nodes* and *actor nodes*. The basic structure of a RAPID dataflow graph is shown in Figure 5.1. Types marked with a $*$ represent pointer types. Inside the graph structure, partition nodes are stored directly in a set. Actor nodes, on the other hand, are stored inside section structures. Sections additionally contain their schedule and keep track of their redundancy. In contrast to partition nodes, graphs have to store sections in a list rather than a set since sections are accessed by index (see Section 4.4.2). Lastly, graphs contain map data structures for graph inputs and results. These maps are essential for the `set_input` and `get_result` functions from the RAPID programming model, which allow users to access certain partition nodes by their identifier.

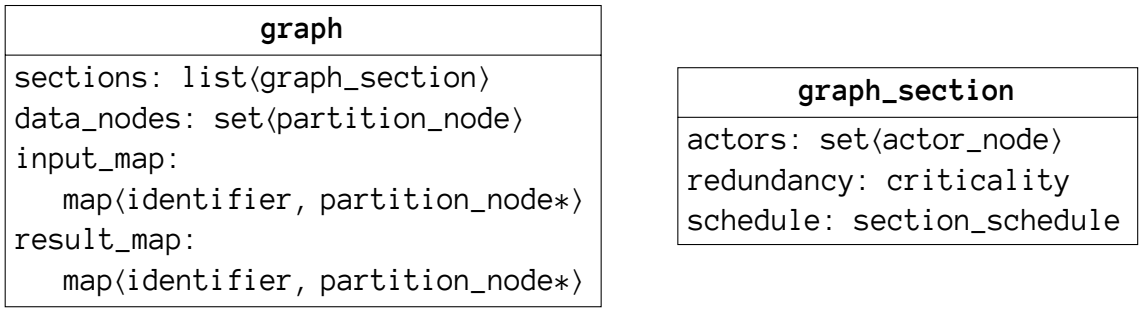


Figure 5.1: Graph and graph section data structures

5.1.1 Partition Nodes

Partition nodes carry data during dataflow executions. Unlike the tokens in traditional dataflow models, data inside partition nodes is not consumed by actor nodes. Instead, data remains in memory to avoid copies, if multiple actor nodes depend on the same data. Partition nodes can be divided into four types. Depending on their type, they may or may not have outgoing edges to actor nodes. In addition to outgoing edges, a partition node can also have an incoming edge which originates from the actor node computing its data during a dataflow execution. The four types of partition nodes are defined as follows:

- **Input partition nodes** represent entry points of graphs. These nodes do not have incoming edges, since their data is not produced by actor nodes. Instead, each input partition has a unique identifier, which allows to replace the contained data for multiple dataflow executions. Identifiers of input partition nodes are specified by users in `parallelize` and `distribute` operations.

- **Output partition nodes** are exit nodes of dataflow graphs and therefore have no outgoing edges. Like input partition nodes, they have a unique identifier, which allows to access its data after a dataflow execution. The RAPID programming model allows users to specify multiple output partition nodes for one graph.

- **Constant partition nodes** have neither incoming edges nor a unique identifier. Data inside constant partition nodes remains the same for all dataflow executions and is only cleared, when the graph itself is deleted.

- **Inner partition nodes** always have incoming and outgoing edges, and their data may be deleted as soon as all dependent actor nodes have been executed. If the dependent actors are redundant, however, the data has to be stored longer, i.e. until the result has been verified.

Figure 5.2 shows the minimum required data members of partition nodes in concrete implementations. Each partition node stores the size of its data, the size of a single data element and a pointer to the actual data. Whether a partition node contains any of the other members shown in Figure 5.2, depends on its type. All partition nodes except output nodes need to keep track of successive actor nodes and store pointers to them in a set. Input nodes and constant nodes do not have incoming edges and therefore do not use the `result_of` pointer. Further, an identifier member is only present in input and output partition nodes.

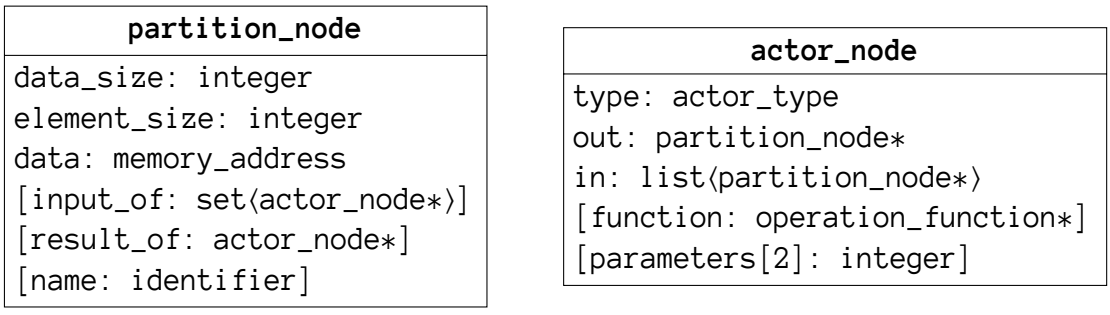


Figure 5.2: Partition and actor node data structures

5.1.2 Actor Nodes

Actor nodes form the second class of nodes in RAPID dataflow graphs. During RAPID dataflow executions, actor nodes take data from partition nodes, run a computation and store new data in another partition node. Figure 5.2 shows which members an actor node structure at least has to include. All actor nodes need to know what their type is, where their incoming edges originate and where their outgoing edge leads to. Incoming edges have to be stored in a list since executed functions of actor nodes do not need to be commutative. An example would be an actor node performing the element-wise division of two integer arrays. For the most part, types of actor nodes are named after the RAPID operation producing them. More specifically, the possible types of actor nodes are:

- `interval`
- `collect`
- `map`
- `map_partitions`
- `combine`
- `zipmap`
- `zipmap_partitions`
- `reduce`
- `reorder`
- `compare`

Actor nodes may additionally store a pointer to a RAPID function and up to two parameters. Whether an actor node stores a pointer to a RAPID function or parameters is determined by its type. In particular, a RAPID function pointer is necessary in `map`, `map_partitions`, `zipmap`, `zipmap_partitions`, `combine` and `reduce` actor nodes. Since partition nodes do not store information about the RAPIDs they belong to, such information (whenever required) is stored in additional member of actor nodes, called `parameters`. More details about the different types of actor nodes and this special member follow in Section 5.2.

5.1.3 Redundant Actor Nodes

As described in Section 4.4, the redundancy of graph sections may be changed during runtime. Redundancy changes affect all actor nodes in a section and may add actor nodes and partition nodes to the dataflow graph or remove them. Figure 5.3 shows an increase of redundancy for a single `zipmap` actor node. Filled circles represent partition nodes. The non-redundant `zipmap` node is replaced by a structure consisting of two equivalent `zipmap` actor nodes, two partition nodes and a comparison actor node.

In the example, the comparison actor node has two incoming edges. If the two incoming edges provide different data, the two previous actor nodes have to be

re-executed. However, RAPID graphs also support comparison actor nodes with three incoming edges. In this case, the comparison node acts as a voter and previous actor nodes do not need to be re-executed if one of its inputs is different.

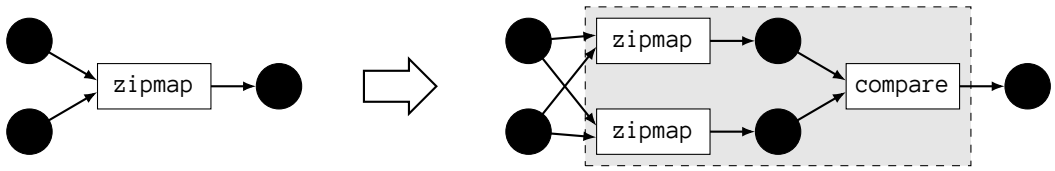


Figure 5.3: Changing the redundancy of an actor

5.2 Graph Construction

RAPID dataflow graphs are constructed from RAPID operations and their data dependencies. Most of the RAPID operations add new nodes to a dataflow graph. Table 5.1 gives an overview of RAPID operations and the types of actor nodes they create. All Operations which are not present in the table (`distribute`, `append`, `split` and `reorder_partitions`) do not add actor nodes to the graph, since they do not perform a computation involving partition data. Such operations can already be executed at graph construction. It is also possible that RAPID operations are partially executed at graph construction. This only applies to `reorder` and only if a RAPID function instead of a list of element indices is provided. In this case, the given RAPID function is called at graph construction to statically compute all new element indices.

Table 5.1: Overview of actor node types

RAPID operation	Types of produced actors
`parallelize`	`interval`
`repartition`	`collect, interval`
`map`	`map`
`map_partitions`	`map_partitions`
`combine`	`combine`
`zipmap`	`zipmap`
`zipmap_partitions`	`zipmap_partitions`
`reorder`	`reorder, interval`
`reduce`	`reduce, collect`
`collect/finalize`	`collect`

RAPID operations must calculate for each partition node they create, how much space in memory for the data is required during dataflow executions. For this, they must know how much space a single data element of type `T` takes up in memory. We assume that this amount of memory can be determined with a special function *sizeof* (`T`). In the following sections, the behavior of RAPID operations with regard to graph construction is described.

5.2.1 Initial and Finalization Operations

Parallelize creates one input or constant partition node and as many interval actor nodes and inner partition nodes as partitions specified in the RAPID operation. Whether an input or constant node is produced depends on whether the user specifies an identifier or not. Figure 5.4a shows the graph nodes and edges created by one execution of `parallelize`. The single input/constant partition node is visualized by the diagonal pattern. This node's data has the same size as the given collection. Inner nodes' data sizes correspond to the element counts specified in the given partitioning multiplied by the size of one element. If instead only an integer is given, a temporary partitioning with partition sizes as equal as possible is calculated. As pointed out in the previous chapter, most RAPID operations (all but finalization operations) return a RAPID containing a list of pointers to partition nodes. In the case of `parallelize`, these nodes correspond to the constructed inner nodes.

Distribute is another operation creating an input or constant partition node. In contrast to `parallelize`, this operation does not divide its input data into smaller partitions. To avoid the overhead of copying data, only a single partition node containing the whole data is created. The returned RAPID contains a list of pointers, but each one of them points to this exact node since there is only one partition node.

a) `Parallelize` b) `Collect` and `finalize`

Figure 5.4: Graphs constructed by initial and finalization operations

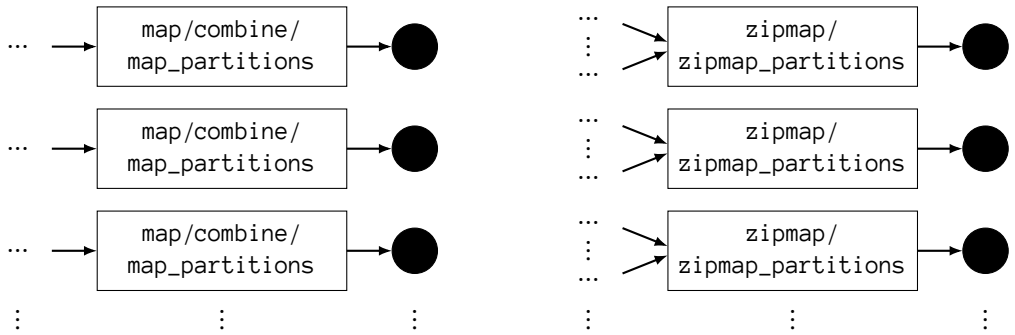
Collect and Finalize are mostly equivalent operations and expand the dataflow graph in the same way. Both create one `collect` actor which gathers the data of the given RAPID's partition nodes and one output partition node. In Figure 5.4b the output node is visualized by the diagonal pattern. The size of its data is equal to the sum of all preceding partition nodes. It is worth noting that the RAPID programming model allows multiple incoming edges of the `collect` node to originate from the same partition node. In such cases, the same data will appear multiple times in the graph output.

5.2.2 Mapping Transformations

Map constructs one `map` actor node and one inner node for each partition node pointer in the given RAPID. New nodes and edges are shown in Figure 5.5a. The memory demand for data in a new inner node is $\frac{\texttt{x} \cdot s \cdot sizeof(\texttt{U})}{sizeof(\texttt{T})}$, where the action function's type is `T` → `U[x]` and the memory demand for data in its preceding partition node is s. Incoming edges of actor nodes do not need to be pair-wise different. In case of a preceding `distribute` operation, for example, all incoming edges originate from the same partition node.

Combine operations produce one actor node and one inner node for each partition node pointer in the given RAPID (Figure 5.5a). Similar to `map`, the additional memory demand for each new inner node is $\frac{s \cdot sizeof(\texttt{U})}{\texttt{x} \cdot sizeof(\texttt{T})}$, if the action function's type is `T[x]` → `U` and the size of the preceding partition node's data is s.

Map_partitions also expands the graph in the same way as `map`. But since it is not an element-wise operation, the RAPID function's associated size function must



a) `Map`, `combine` and `map_partitions` b) `Zipmap` and `zipmap_partitions`

Figure 5.5: Graph nodes and edges created by various transformations

be taken into account to calculate the amount of memory required for data inside partition nodes. The additional memory demand for each new partition node is $sizeof(\texttt{U}) \cdot \texttt{size}\left(\frac{s}{sizeof(\texttt{T})}\right)$, where `size` is the RAPID function's size function, s is the data size of its preceding partition and $\texttt{partition}\langle\texttt{T}\rangle \rightarrow \texttt{partition}\langle\texttt{U}\rangle$ is the action function's type.

`Map_partitions` nodes belong to the category of actor nodes which require the `parameters` member. The two integers are used to store the index of the input partition node inside the superordinate RAPID and the number of partitions belonging to this RAPID. These values must be stored because users may specify `map_partitions` functions that use this information (there are corresponding members in partition structures, see Section 4.1.2).

Zipmap is similar to the previous transformations in the way it expands the dataflow graph. Figure 5.5b shows that the only difference lies in the additional incoming edges of actor nodes. Required memory for partition nodes can be calculated in the exact same way as in the case of `map`. It does not matter which one of the preceding partition nodes is chosen for the size calculation, since all preceding nodes of a `zipmap` node contain the same number of data elements.

Zipmap_partitions constructs new graph nodes and edges analogous to `zipmap` (see Figure 5.5b), but the memory requirements of partition node data is calculated similar to `map_partitions`. The difference is that the size function has to be called with size information of all preceding partition nodes in the correct order.

For the same reason as with `map_partitions` actor nodes, these nodes also need to store two integers in their `parameters` member. Two parameter fields are sufficient because every input node of a `zipmap_partitions` actor node has the same index inside its superordinate RAPID.

5.2.3 Other Transformations

Repartition behaves like a `collect` operation followed by a `parallelize` operation (with a different partitioning than before). Therefore, the graph expansion of a `repartition` operation looks very similar to the graph snippets shown in Figure 5.4. New graph nodes and edges are depicted in Figure 5.6. Instead of an input, output or constant partition node, `repartition` constructs an additional inner node, which connects the `collect` actor node with all `interval` actor nodes. The memory demand of partition nodes is calculated exactly like in `collect` and `parallelize`. It should be noted that `repartition` temporarily narrows the dataflow and therefore

reduces the possible parallelism. To achieve higher performance, users should reduce its usage to a minimum.

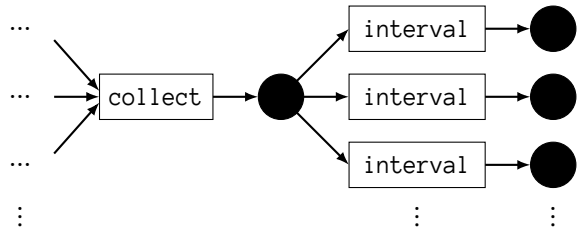


Figure 5.6: Graph nodes and edges created by `repartition`

Reduce constructs multiple actor nodes of the same name and one `collect` actor node, as shown in Figure 5.7. A `reduce` actor node applies its associated RAPID function to the elements inside the preceding partition node until only one element is left. As a result, each partition node following a `reduce` node only requires additional memory for one element. If the action function's type is $\mathtt{T} \times \mathtt{T} \rightarrow \mathtt{T}$, the required amount of memory is *sizeof*(`T`). Since all `reduce` actor nodes in Figure 5.7 use the same RAPID function, the single partition node before the right `reduce` node requires memory in the amount of $n \cdot$ *sizeof*(`T`), where n is the number of preceding partition nodes.

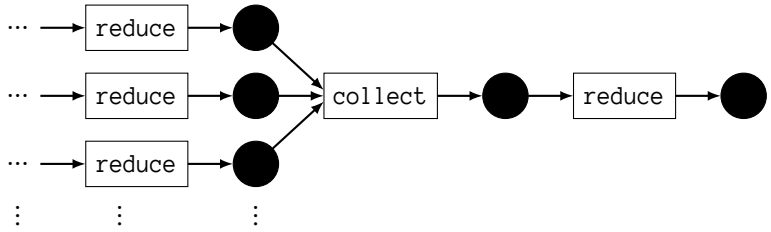


Figure 5.7: Graph nodes and edges created by `reduce`

Reorder creates one actor node of corresponding type and multiple `interval` actor nodes. An according graph snippet is shown in Figure 5.8. The `reorder` node has incoming nodes from one constant partition node (crosshatched in Figure 5.8) and multiple other nodes constructed by previous RAPID operations. The purpose of this single constant node is to store the new element indices, so the amount of required memory for its data is the number of elements in the given RAPID multiplied with *sizeof*(`integer`). If a user specified a `reorder` function instead of an index array, this function is used to calculate all new indices at graph construction.

Data inside the single inner node following the `reorder` node requires the same amount of memory as the data within the given RAPID. Besides the single `reorder` node, the operation creates multiple `interval` nodes, which restore the original partitioning. Similar to `repartition`, `reorder` narrows the dataflow temporarily and should not be used frequently in a RAPID program.

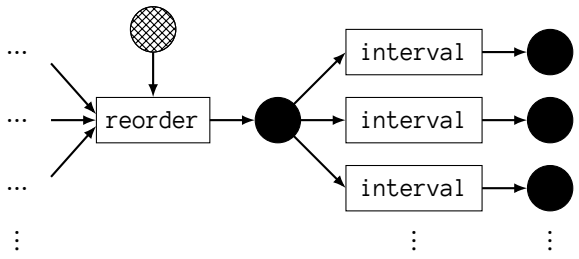


Figure 5.8: Graph nodes and edges created by `reorder`

Append and Split are two of the types of RAPID operations that do not expand the dataflow graph. Both operations create RAPIDs which point to already existing partition nodes. Figure 5.9 shows an example. From left to right, the figure illustrates a `split` operation which divides a large RAPID into small RAPIDs with two partitions each. The other direction represents an `append` operation which returns a single RAPID with pointers to all given RAPIDs' partition nodes. Since no actor nodes are created, both operations do not lead to additional costs at runtime. Instead, `append` and `split` affect the following transformations. An important aspect about `append` is that the RAPID programming model does not prohibit to call the operation with multiple references to the same RAPID. In such cases, the created RAPID will have multiple pointers to some partition nodes in the graph.

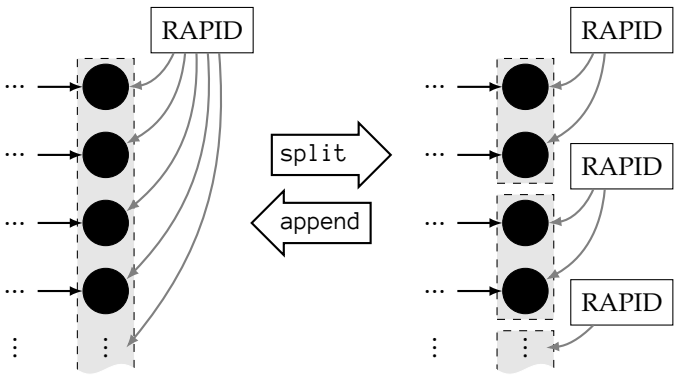


Figure 5.9: Visualization of `Append` and `split`

Reorder_partitions does not expand the dataflow graph either. The operation only returns a new RAPID with a different order of partition node pointers. Like with `append` and `split`, calling `reorder_partitions` affects the graph expansion of following RAPID operations. An example is shown in Figure 5.10. The graph snippet in the left half of the figure was created by a `map` operation, which was called with a RAPID consisting of four partitions. The right side shows the same partition nodes and the same `map`, but instead of calling `map` directly, the partition order is changed before. This causes `map` actor nodes to have different incoming edges. The first actor node now has the second partition node as its input, the second actor node depends on the fourth partition node, etc.

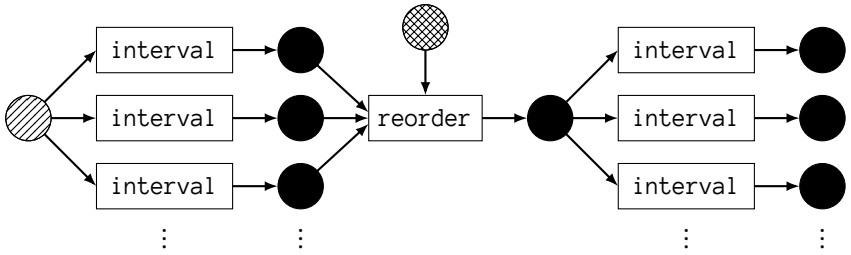
a) Situation without `reorder_partitions` b) Situation with `reorder_partitions`

Figure 5.10: `Reorder_partitions` affecting a following `map` operation

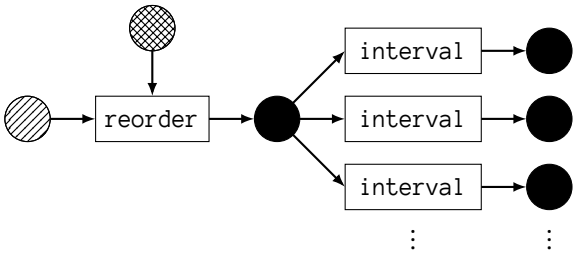
5.3 Graph Construction Optimizations

The previous section already mentioned that some RAPID operations reduce parallelism in the dataflow. Specifically, this applies to `repartition` and `reorder`. Both of these operations usually involve merging data with a `collect` or `reorder` node and splitting data with `interval` actor nodes (see Figure 5.6 and Figure 5.8). However, there are some situations, where splitting or merging data is unnecessary and can be avoided. One such situation is a `reorder` or `repartition` operation appearing directly after `parallelize`. In this case, `interval` actor nodes produced by `parallelize` are unnecessary (since the data is immediately merged again) and may be removed. An example graph built by a `parallelize` operation and a following `reorder` operation is shown in the upper half of Figure 5.11. The bottom half shows, how the graph looks, after all excess nodes were removed. In case of `repartition`, the removal of nodes is similar. The difference is that the `collect` node can also be removed, whereas the `reorder` actor node must remain.

The second situation that allows to prune the dataflow graph is when a `reorder` or `repartition` operation appears directly before a `collect` (or `finalize`) operation.



a) Graph after a `parallelize` operation followed by `reorder`



b) Graph after removal of unnecessary nodes

Figure 5.11: `Reorder` optimization example

These specific combinations of RAPID operations lead to a `collect` actor node preceded by multiple `interval` nodes. Since the single `collect` node cancels out the preceding actor nodes, all of them may be removed from the graph.

Lastly, dataflow graphs can be pruned if the corresponding RAPID program contains successive `reorder` and `repartition` operations. The removal of nodes is analogous to the previous cases, with the only difference that pruning happens inside the graph instead of in the front or rear.

As shown in Figure 5.11, graph pruning involves amongst other things the removal of partition nodes. An important aspect is that most of the removed partition nodes appear in the pointer lists of RAPIDs. The removal of nodes makes these RAPIDs invalid. Graph modifications like this are the reason why RAPIDs can be used only once as an argument by default. Users have to call the `persist` member function of a RAPID in order to use it multiple times in RAPID operations (see Section 4.1.1). This member function marks the according partition nodes as persistent and thereby prevents them from being removed from the graph.

5.4 Temporary Dataflow Graphs

This chapter already stated that some RAPID operations are already executed completely or partially at graph construction. It is possible that such a RAPID operation

computes large amounts of data and therefore slows down graph construction. In the proposed programming model, this only applies to `reorder`, which computes the new element indices already at graph construction. An easy and architecture-independet way to efficiently compute data at graph construction is by building a small temporary graph and starting a dataflow execution. This way, all cores of the underlying hardware can be utilized. An example for such a temporary graph is shown in Figure 5.12. The graph's constant partition node (leftmost circle) is initialized with the partitioning of the RAPID that is supposed to be reordered. Based on the partitioning, new element indices are computed in parallel with special `reorder_compute` nodes. After a dataflow execution, the temporary graph's output partition is converted to a constant partition node and transferred to the actual graph. The rest of the temporary graph can be discarded.

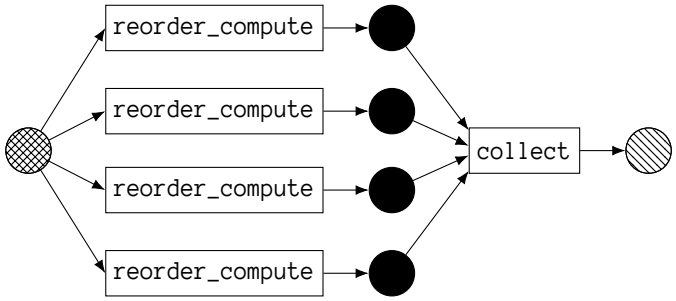


Figure 5.12: Exemplary temporary graph for `reorder`

5.5 Import and Export of Dataflow Graphs

The previous sections described how dataflow graphs are built by RAPID operations. But as mentioned in Chapters 3 and 4, graphs can also be created from representations in a suitable description format. To support an import and export of graphs, it should be easy to reverse this process. A way to achieve this is by using a direct transformation, in which each node and each edge in the RAPID dataflow graph has a corresponding entity in the graph representation. This includes that all required node attributes from Figure 5.2 are present and that all actor nodes are part of a graph section.

The graph import and export functionality implemented in the reference implementation uses an extended *DOT graph description format* [GKN15] and represents such a direct transformation. A small example is shown in Listing 5.1. The code in this listing describes a minimal graph consisting of one actor node with two incoming edges and one outgoing edge. After the graph name is defined in the first line, all partition nodes are listed. These nodes have several custom attributes,

Listing 5.1: Small graph in extended DOT format

```
1  digraph example_graph {
2    n_1 [uid=1, psize=256, esize=8, ptype="input", pname="in_1"];
3    n_2 [uid=2, psize=256, esize=8, ptype="input", pname="in_2"];
4    n_3 [uid=3, psize=256, esize=8, ptype="output", pname="out"];
5    subgraph 0 {
6      n_4 [atype="zipmap", afunc="add_function"];
7    }
8    n_1 -> n_4 [paramindex="0"];
9    n_2 -> n_4 [paramindex="1"];
10   n_4 -> n_3;
11 }
```

for example a unique ID, partition size and element size. It is followed by a list of all actor nodes. These are divided into subgraphs, a standard DOT concept we utilize in order to distinguish between the different graph sections. Like partition nodes, actor nodes also have different non-standard attributes, for example a type and RAPID function attribute. Starting in Line 8, all node dependencies are listed. The custom attribute *paramindex* is used to ensure the right order of incoming edges. DOT-specific IDs can be chosen arbitrarily, since they are only used to identify dependencies between nodes. In this example IDs *n_1* to *n_4* were chosen. It should also be noted that it is possible to add DOT-specific attributes in favor of a better graph visualization, for example the *label* and *shape* attributes. These DOT-specific attributes are ignored by the import implementation.

5.6 Dataflow Graphs in The C++ Reference Implementation

This section continues the topic of Section 4.5 and describes characteristics of RAPID dataflow graphs in the C++14 reference implementation. Similar to the implementation of the RAPID programming model, graphs and graph construction rules are implemented in a hardware-independent fashion. Dataflow graphs consist of double linked nodes since many scheduling heuristics benefit from navigation in both direction. Further, actor nodes include pointers to their redundant actor nodes and the corresponding comparison node. Because all redundant actor nodes and comparison actor nodes are already created during graph construction, switching between redundancy configurations at runtime is fast.

To reduce the size of the binary, partition nodes are implemented as ordinary classes, not as class templates. As a result, partition nodes do not carry type information of the associated data. The type of the elements in a partition node is restored whenever the data is accessed by a RAPID function during dataflow executions.

5.7 Example Dataflow Graphs

This section revisits the RAPID applications described in Section 4.6 and shows resulting dataflow graphs. These dataflow graphs are created from the respective RAPID applications according to the rules described in Sections 5.2 and 5.3. Example graphs in this section are depicted without redundant actor nodes. Furthermore, illustrations of exemplary dataflow graphs do not show different sections, since we assume that these graphs were constructed without checkpoints.

5.7.1 Dataflow Graph of Cannon's Algorithm

In the RAPID implementation of Cannon's algorithm (see Listing 4.4), the number of submatrices into which the input matrices are divided can be varied. Different divisions lead to smaller or larger dataflow graphs. The smallest possible graph is depicted in Figure 5.13. It is easy to see that matrices are split into four blocks. For this division, the main loop has to be run twice. In the graph, these two loops correspond to the two layers of `zipmap_partitions` actor nodes. Splitting the input matrices into more submatrices leads to both wider and deeper graphs with more `zipmap_partitions` layers.

Cannon's algorithm is a good example for the graph optimization techniques since all `reorder` operations used for this algorithm are either at the beginning or right before the final `collect` operation. All `interval` actor nodes before the upper `reorder` nodes and all `interval` and `collect` actor nodes after the bottom `reorder` nodes were removed from the graph. Besides `reorder` operations, the RAPID implementation of Cannon's algorithm also involves `reorder_partitions` operations. These operations are not directly visible in Figure 5.13. Only their impact can be seen in the rather complicated data dependencies between the two layers of `zipmap_partitions` nodes.

The small difference between the RAPID program and Algorithm 4.1, which is described in Section 4.6.1, can be observed in the upper part of the graph. In favor of a smaller graph, the creation of partition nodes for the result matrix C and the initialization with zeros was left out. Instead, partition nodes between the two layers of `zipmap_partitions` nodes are the first partition nodes in the graph that correspond to matrix C.

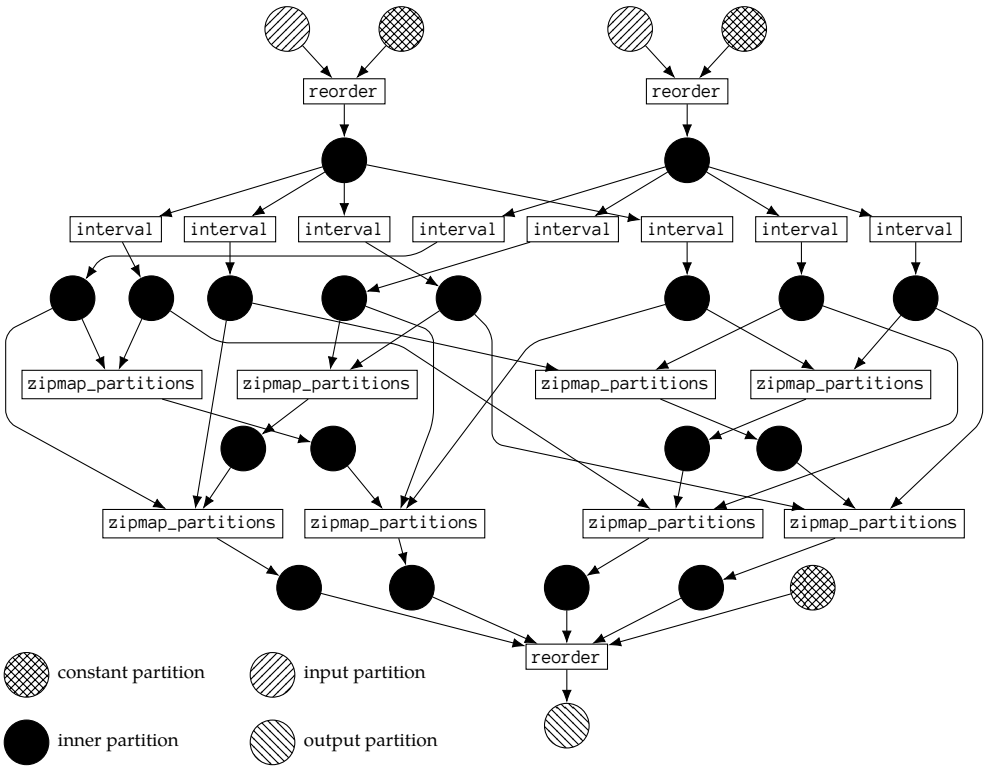


Figure 5.13: Dataflow graph of Cannon's algorithm

5.7.2 Dataflow Graph of the Fast Fourier Transform

A possible dataflow graph constructed by the FFT RAPID program specified in Listing 4.5 is depicted in Figure 5.14. The figure clearly shows a butterfly pattern. This pattern is typical for the data dependencies in FFT algorithms. Similar to the graph of Cannon's algorithm, the more partitions the input is divided into, the deeper the graph becomes. The FFT graph's depth is logarithmic with respect to the number of partitions the input vector is divided into. In this example, the input is divided into four partitions. The middle part of the graph shows the two constant nodes which contain the loop index. In theory, one constant node would be sufficient and subsequent loop index nodes could be created at runtime with `map` actor nodes. However, this would lead to additional data dependencies and therefore a more complicated scheduling. Not only in this case but in general, it is often preferable to compute as many values as possible already at graph construction time. The graph also shows that this RAPID program, like the implementation of Cannon's algorithm in the previous section, benefits from graph optimization. In the upper part of the graph, all `interval` nodes before the `reorder` node were removed.

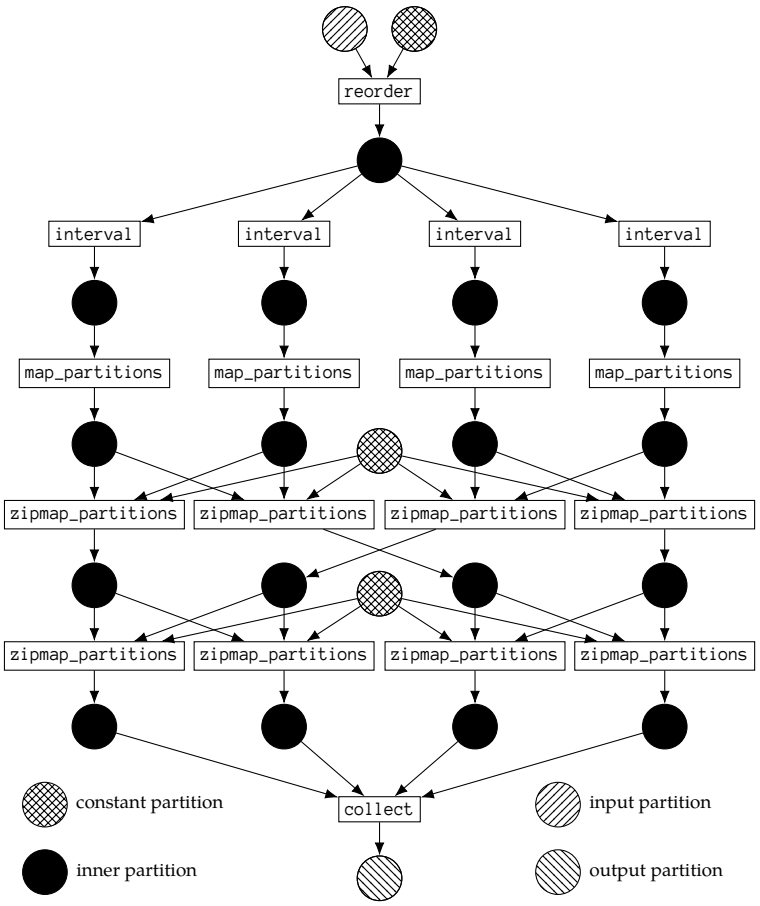


Figure 5.14: Dataflow graph of the FFT algorithm

5.7.3 Dataflow Graph of Bitonic Sorting

An example graph for bitonic sorting created by the RAPID program from Listing 4.6 is shown in Figure 5.15. It is easy to see the similarities between RAPID dataflow graphs for bitonic sorting and FFT, since both graphs contain butterfly patterns. The main difference is that (for equal partitionings) bitonic sorting graphs are much deeper because of the additional inner loop in the RAPID program (see Listing 4.6). Further, FFT graphs only contain `map_partitions` actor nodes in the beginning of the graph (see Figure 5.14), whereas bitonic sorting graphs contain `map_partitions` actor nodes on multiple occasions inside the graph. These nodes sort the elements inside the given partition entirely, whereas `zipmap_partitions` actor nodes only compare two partitions element-wise. This means that in contrast to the `zipmap_partitions` nodes `map_partitions` nodes contain multiple steps of parallel comparisons.

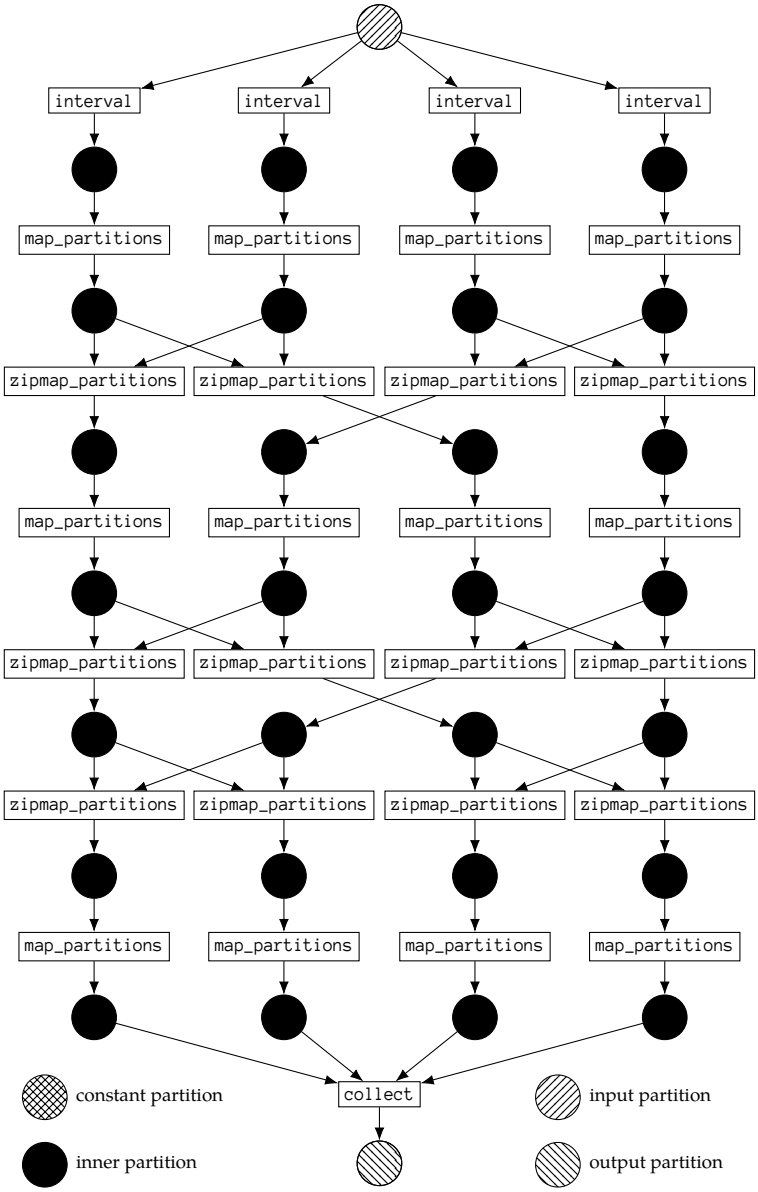


Figure 5.15: Dataflow graph of bitonic sorting

5.8 Summary

In this chapter, RAPID dataflow graphs were described in greater detail. Dataflow graphs are bipartite and consist of partition nodes and actor nodes. The former carry data during dataflow executions, while the latter contain information about how this data is computed. The redundancy of actor nodes in the graph may be changed by replacing an actor node with a structure of multiple redundant actor

nodes and a comparison actor node. Graphs are successively constructed by RAPID operations according to the specified construction rules. A few RAPID operations narrow the dataflow and should be used as rarely as possible. In some cases, it is possible to remove partition nodes and actor nodes from the dataflow graph at graph construction for optimization purposes. How dataflow graphs can be actually executed on different hardware architectures is the topic of the next chapter.

6

Dataflow Execution on Different Hardware Architectures

The previous chapters introduced the RAPID programming model and RAPID dataflow graphs, and described their reference implementation in C++14. As already mentioned, the programming model's reference implementation is hardware independent and can serve as a basis for dataflow runtime environments (RTEs) on various architectures, in particular those with distributed memories. The implementation abstracts from the dataflow execution and only covers the construction of dataflow graphs based on RAPID programs. In general, the interface between the RAPID programming model and a compatible dataflow RTE on a specific hardware architecture is small. It only consists of functions for graph execution (with and without online scheduling), offline scheduling and memory allocation.

In this chapter two dataflow RTEs for different hardware architectures based on the RAPID programming model are proposed. The focus lies on graph execution. Scheduling is the topic of Chapter 7. Since this chapter focuses more on the execution of actor nodes and less on the structure of graphs, actor nodes are often referred to as *actors* and analogously partition nodes as *partitions*.

The structure of this chapter is as follows: The first section introduces a dataflow RTE for shared-memory architectures, while the second section describes how RAPID dataflow graphs can be executed on hardware architectures based on a

network-on-chip (*NoC*). Both sections describe the basic routines for the execution of graph sections and point out possible optimizations in the RTE.

6.1 Shared-Memory Dataflow Runtime Environment

Shared-memory architectures can be utilized in many application areas. Their use ranges from low-power embedded systems up to high-performance desktop computers. Shared-memory systems consist of multiple processing cores which access one main memory. Most of the time, all cores are homogeneous, have full access to any memory location and no core is privileged in terms of access latency.

The implementation of a RAPID-compatible dataflow runtime environment on shared-memory architectures is rather simple since each core can freely navigate through the dataflow graph and has direct access to the fixed schedule or online scheduling data structures. This means that each core is able to check on its own which actor is the next one to execute according to its schedule and whether all the required data is available. When multiple cores concurrently access shared data, there is often a risk that race conditions might occur. To avoid race conditions, the proposed shared-memory dataflow runtime environment uses special data structures which are described in the following section.

6.1.1 Runtime State Data

Runtime state data covers different values that are important during dataflow executions. The proposed runtime environment encapsulate these values as members in a data structure of type `runtime_data`, which is shown in Figure 6.1. Every partition node in a RAPID graph exclusively owns one `runtime_data` object. Since multiple cores may concurrently modify runtime values of the same node, each access to a member of `runtime_data` must be atomic. Because all members are integer or boolean values, there are no mutexes or other locks required if the hardware supports atomic operations.

The first member, `is_complete`, is a flag which is set when partition data is present. This flag can be used to check whether an actor is ready to be executed. When actors are executed redundantly, however, it is not sufficient to store only one bit for the partition's state. Partition data may be present, but the subsequent comparison actor has not yet declared the data as correct or has already declared it as incorrect. These two states are stored in the second field `redundancy_state`.

Remaining data members are primarily used to accelerate graph executions. The first of these two, `delete_counter`, is used for memory management purposes. Whenever an actor is finished, the delete counter of all preceding partitions is

runtime_data
is_complete: atomic⟨boolean⟩
redundancy_state: atomic⟨integer⟩
delete_counter: atomic⟨integer⟩
ready_counter: atomic⟨integer⟩

Figure 6.1: Shared-memory runtime state data

decremented. If the counter reaches zero, data inside this partition is no longer needed and the memory can be freed. Therefore, the delete counter has to be initialized with the number of outgoing edges of the partition node. To avoid race conditions, which could lead to freeing the memory multiple times, the decrement has to be implemented via an atomic fetch-and-subtract instruction. The advantage of such a counter is that it offers a fast way to check whether memory can be freed. Without a counter the executing core would have to navigate through the graph in order to check whether memory can safely be freed whenever an actor is finished. However, this could lead to high overhead. In the worst case, the core would have to iterate nearly through the whole graph just to check if a single partition is no longer required (for example if nearly all actors depend on this one partition).

The last member, `ready_counter`, is used to determine whether an actor is ready to be executed. Analogous to the other members, one ready counter is stored for each partition, although this counter conceptually belongs to an actor. Since each actor node in a RAPID dataflow graph has exactly one outgoing edge to a partition node, the ready counter of this particular node is used for the actor. The ready counter of an actor is decreased, whenever the computation of an input is finished. When the counter reaches zero, the actor is ready for execution. This means that for each actor the ready counter must be initialized with the number of incoming edges, excluding edges from input partition nodes and constant partition nodes. Besides the acceleration of graph executions, the ready counter serves another purpose. In online scheduling mode, the scheduling routine has to be invoked whenever an actor gets ready. It is important that an actor only gets scheduled once. This can be ensured by implementing the decrement with a fetch-and-subtract instruction. Synchronization would be much more difficult if the system had to check the `is_complete` flags of all inputs instead of a single ready counter.

6.1.2 Supported Redundancy Configurations

As already mentioned in Section 5.1.3, the proposed RTE was designed to support redundant actor executions. Based on this, the proposed shared-memory dataflow

runtime environment supports five different redundancy configurations, which are shown in Figure 6.2. The first possible configuration is no redundancy at all. This configuration clearly achieves the highest performance, but it lacks the ability to detect errors in the computation of data. The four redundant configurations are combinations of the number of redundant executions and the way redundant actors are executed. Executing actors three times has the advantage that results can be compared by a voter and a single error does not cause a rollback, but it clearly results in more computing time than executing the actor twice.

If redundant actors are executed on the same core, scheduling gets easier since the two (or three) redundant actors and their comparison node can be scheduled as one actor with longer runtime. However, the downside of this approach is that errors which occur consistently, for example due to a faulty core, cannot be detected.

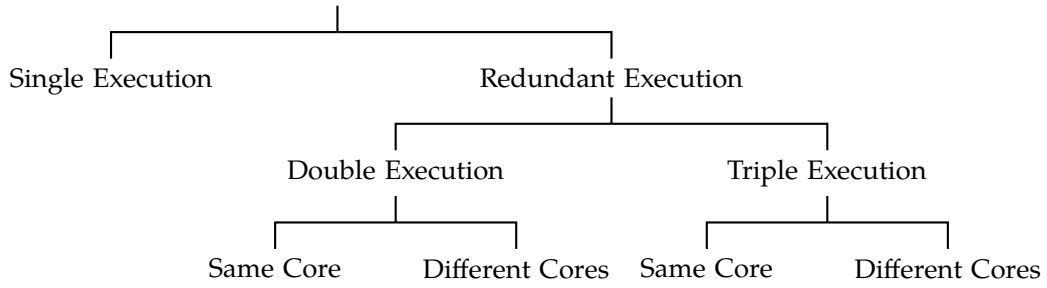


Figure 6.2: Redundancy configurations in the shared-memory RTE

6.1.3 Graph Execution

Whenever a RAPID graph is executed, the first step is to run an initialization routine which correctly sets the initial runtime values of all partitions in the graph, as described in Section 6.1.1. After the initialization phase is done, all cores enter the main graph execution loop. As mentioned in previous chapters, graph sections are always executed separately. A barrier ensures that cores do not enter the next section immediately while they are working on the current section. Changing the redundancy configuration of a section may change the graphs structure and therefore may require a different schedule. Barriers between sections ensure that the next section's redundancy configuration can always be changed safely.

Graph Execution with Offline Scheduling The main graph execution loop of one section in offline scheduling mode is shown in Algorithm 6.1. In every iteration of the outer loop, the core waits until the next actor according to the schedule is ready, i.e. its inputs are complete. All further steps depend on the section's redundancy

configuration. In case of no redundancy or redundant execution on the same core, the actor is executed without additional synchronization (lines 20 and 22). The remaining, more complicated case relates to redundant actor executions on different cores. In this case, the core executes the actor (line 8) and then either waits for the comparison actor (line 16) or executes the comparison actor itself (line 11). This process is repeated until the comparison actor is successful. In case of double executions, this requires that both results match, while in case of triple executions, at least two results must be equal.

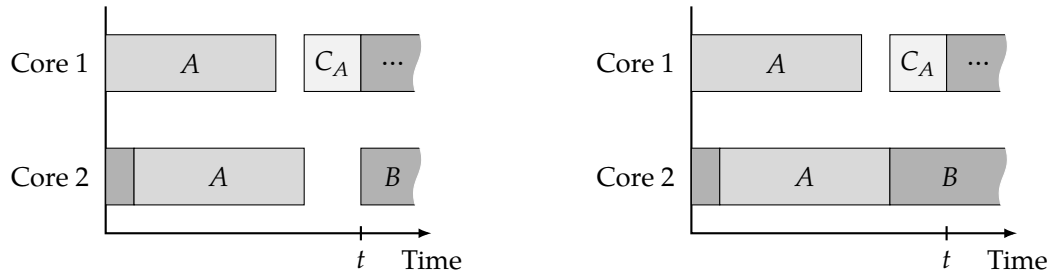
The described section execution is easy to implement and analyze, but the performance of redundant actor executions on different cores may be reduced due to

Algorithm 6.1: Shared-memory section execution with offline scheduling

```
1  function EXECUTE_SECTION_FIXED_SCHEDULE(s : section_schedule)
2      a ← s.BEGIN();
3      while s not done do
4          wait until a ready;
5          if s redundant on different cores then
6              c ← a.COMPARISON_ACTOR();
7              repeat
8                  EXECUTE_ACTOR(a);
9                  if c = s.NEXT(a) then
10                     wait until c ready;
11                     EXECUTE_ACTOR(c);
12                     if c successful then
13                         a ← s.NEXT(a);
14                     end
15                 else
16                     wait until c done;
17                 end
18             until c successful;
19         else if a redundant on same core then
20             EXECUTE_ACTOR_REDUNDANTLY(a);
21         else
22             EXECUTE_ACTOR(a);
23         end
24         a ← s.NEXT(a);
25     end
26 end
```

waiting for the comparison actor to complete (line 16). Therefore, the proposed shared memory dataflow runtime environment provides the option to execute such redundant actors more optimistically. To illustrate this, an example with two cores that execute a redundant actor A is shown in Figure 6.3. Core 2 starts with the execution of A a little later. The comparison actor C_A is executed on core 1. On the left side of the figure, the pessimistic actor execution (without errors) is shown. The second core waits until the comparison has been successful before proceeding with the execution of actor B. In optimistic mode, core 2 does not wait until the first core has executed the comparison actor and instead proceeds with its schedule. This is under the assumption that B does not depend on the result of A. Otherwise, the second core would wait nonetheless because the optimistic execution is not speculative and actors are only executed with verified data. It should be noted that the waiting for comparison actors in pessimistic mode is an additional dependency between actors which has to be considered by offline schedulers because otherwise a non-executable schedule might be computed.

The execution mode influences the RTE's behavior in case an error is detected. In pessimistic mode, re-execution of A starts on both cores at time t. If identical computations take roughly the same time on all cores, core 1 does not have to wait long for the re-execution of C_A. In optimistic mode on the other hand, core 2 must finish the execution of actor B to re-execute A. Therefore, core 1 must likely wait some time until the comparison actor can be re-executed. But under the assumption that errors occur much less frequent than correct computations, this runtime mode effectively reduces idle times.



a) Pessimistic Execution b) Optimistic execution

Figure 6.3: Shared-memory execution modes with offline scheduling

Graph Execution with Online Scheduling Graph execution with online scheduling slightly differs from the previously described execution based on a fixed schedule. The routine in Algorithm 6.2 has one parameter, which represents an abstract scheduling object. Since it is possible to implement the methods of this object in

different ways, the RTE is compatible with various online scheduling algorithms. At the beginning of the main loop, the core first waits until there is work to do or until the execution is finished. The `get_work` call in line 5 makes the RTE compatible with work stealing approaches. For other scheduling procedures, for example work sharing approaches, this method can be implemented as an empty function. When the core leaves the waiting loop because there is work to do, an actor is extracted from the schedule (line 8). As before with static scheduling, there are three cases for different redundancy configurations. The non-redundant and locally redundant

Algorithm 6.2: Shared-memory graph execution with online scheduling

```
1  function EXECUTE_SECTION_ONLINE_SCHEDULING(s : scheduling_struct)
2      loop
3          while no work to do and execution not done do
4              wait some time;
5              s.GET_WORK();
6          end
7          if execution done then return ;
8          a ← s.POP();
9          if a redundant on different cores then
10             EXECUTE_ACTOR(a);
11             c ← a.COMPARISON_ACTOR();
12             if c ready then
13                 EXECUTE_ACTOR(c);
14                 if c successful then
15                     s.SCHEDULE(c.READY_SUCCESSORS());
16                 else
17                     s.SCHEDULE(c.PREDECESSORS());
18                 end
19             end
20         else if a redundant on same core then
21             EXECUTE_ACTOR_REDUNDANTLY(a);
22             s.SCHEDULE(a.READY_SUCCESSORS());
23         else
24             EXECUTE_ACTOR(a);
25             s.SCHEDULE(a.READY_SUCCESSORS());
26         end
27     end
28 end
```

cases (lines 20 and 23) are straightforward. After the actor is executed, a dynamic scheduling routine is called. This routine schedules all subsequent actors that are now ready. In case of redundant actor execution on different cores, the core that finishes a redundant actor last executes the corresponding comparison actor. If the comparison is successful, subsequent actors are scheduled like in the other cases. However, if the comparison fails, the redundant actors are scheduled again (line 17).

6.1.4 Copy Avoidance

Section 5.2 described that some RAPID operations (more precisely `append`, `split` and `reorder_partitions`) do not modify data at runtime and are already executed at graph construction. Similar to such operations in a RAPID program, `interval` and `collect` actors in the graph do not compute new data.

An `interval` actor only extracts a portion of data from its input partition and stores it in its output partition. This copy can be avoided by allowing the input and output partitions of an `interval` actor to share memory as shown in Figure 6.4. The input node maintains a pointer to a continuous block of memory (gray area), where its data is stored. This exact memory is then reused for the actor's output partition, which contains only a portion of the whole data (area with diagonal pattern). Sharing the memory between the two partitions reduces not only number of copy instructions but also the amount of required memory.

Sharing memory is also possible for `collect` actors. However, avoiding copies entirely like with `interval` actors is not always possible. `Collect` actor nodes have multiple incoming edges from not necessarily distinct partition nodes. So whenever some partition data is collected multiple times, this data must be copied to obtain a contiguous block of data.

It should be noted that the described procedure of copy avoidance is advantageous under the assumption that all cores are able to access any memory location equally fast, but it may not lead to a better performance in all cases. On *Non-Uniform Memory Access* (NUMA) architectures, for example, it can be beneficial to copy data to a memory location with faster access to accelerate following computations.

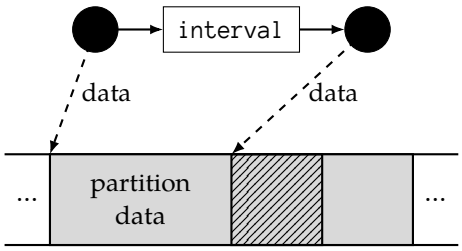


Figure 6.4: `Interval` actor copy avoidance

6.2 Network-on-Chip-based Dataflow Runtime Environment

Hardware-architectures based on a NoC usually require special programming. Cores only have small local memories and cannot access memories of other cores directly. Instead, data is sent over the NoC via explicit send and receive instructions. In the development of the RAPID programming model NoC-based, possibly clustered, hardware architectures were considered from the beginning. Therefore, an efficient runtime environment can be developed without larger pitfalls. On such architectures, the RAPID programming model greatly simplifies programming. It allows users to fully utilize the underlying hardware in a high-level fashion, i.e. without the need to use low-level send or receive instructions. In case of a clustered NoC-based architecture, the runtime environment also partially takes care of the synchronization between cores inside a cluster. An abstract NoC-based system is described in the next section, and all concepts described in the following sections are based on this architecture.

6.2.1 Hardware Architecture Overview

The basic hardware architecture which serves as the basis for the following sections is shown in Figure 6.5. It consists of multiple tiles, which are connected through a NoC of arbitrary topology. Each tile contains a network adapter, a small tile-local memory and multiple cores. The tile-local memory is shared amongst all

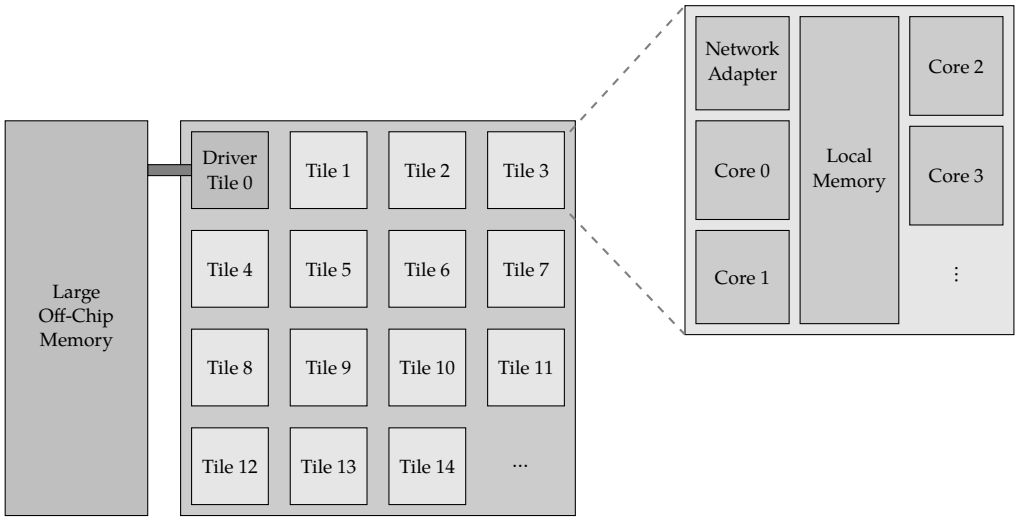


Figure 6.5: Abstract clustered hardware architecture

cores of a tile. To exchange data between tiles, two ways of communication are supported. First, tiles may communicate via sending and receiving short messages with the size of a few integers. It is assumed that the hardware supports blocking and non-blocking variants of send and receive instructions. The second way is by asynchronously transferring data directly into other memories or requesting data directly from other memories. Besides the standard tiles, one tile additionally has access to a larger off-chip memory, for example a DDR-memory. This tile is called the *driver tile*.

6.2.2 Compute Tile Dataflow Execution

Tile-local memories are small, and so both the dataflow graph and fixed schedules are usually too large to fit. Therefore, only a few partitions and the current actor are stored in local memories of compute tiles. Since the small memory is supposed to contain as much partition data as possible during runtime, the binary for compute tiles is rather minimalistic. It consists of a small main loop, code for message passing and the different types of actors as well as RAPID functions. All compute tiles are initialized with the same binary containing all RAPID functions. This ensures that after, for example, switching to online scheduling, any tile is able to execute any actor. Furthermore, small portions of memory are required for communication buffers and to store a list of available partitions. The largest part of the tile-local memory, however, is reserved for actual partition data. This contiguous part of memory is divided into n equally sized chunks, where n is configurable before compilation.

Algorithm 6.3 shows the compute tiles' dataflow execution loop. It is assumed that an array called *partitions* has already been initialized before the compute tile enters the loop. This array has one entry for each of the n chunks of memory and stores information about the partition whose data is currently available in the respective chunk. Inside the loop, the tile first waits for a message from the driver tile. When a message is received, the further steps depend on the type of message. Types of messages are:

- *partition message*
- *delete message*
- *clear message*
- *exit message*
- *actor message*

If the received message is a *partition message*, partition information is extracted and the partition is added to the array of partitions. Besides partition information, the message also contains a target tile ID. This ID belongs to one of the tiles that have the partition's data in their tile-local memory. In case this ID matches the driver tile's

Algorithm 6.3: Compute tile graph execution

```
1  function EXECUTE_COMPUTE_TILE()
2      loop
3          wait for message from driver tile;
4          m ← GET_MESSAGE();
5          if m is partition message then
6              p ← m.GET_PARTITION();
7              c ← m.GET_CHUNK();
8              partitions[c] ← p;
9              t ← m.GET_TARGET_TILE();
10             if t ≠ driver tile then
11                 r ← m.GET_REMOTE_CHUNK();
12                 transfer partition data from chunk r on tile t to local chunk c;
13                 wait until transfer done;
14                 notify driver tile;
15             end
16         else if m is delete message then
17             c ← m.GET_CHUNK();
18             CLEAR_CHUNK(partitions, c);
19         else if m is clear message then
20             CLEAR_ALL(partitions);
21         else if m is exit message then
22             return;
23         else if m is actor message then
24             p ← m.GET_PARTITION();
25             c ← m.GET_CHUNK();
26             partitions[c] ← p;
27             a ← m.GET_ACTOR();
28             EXECUTE_ACTOR(a);
29             notify driver tile;
30         end
31     end
32 end
```

ID, the compute tile is done since the driver tile already transferred the partition data into the right chunk. Otherwise, the compute tile must initiate a transfer from the target tile's local memory into its own tile-local memory. Information about the remote chunk which contains the required partition data can be extracted from the

message. After the transfer is done, the driver tile is notified about the successful transfer via a small notification message.

Another type of message from the driver tile is the *delete message*. This message is the counterpart to a *partition message*. After receiving this type of message, the compute tile clears the corresponding entry in the partition array. Since memory management is done via simple equal-sized chunks, no memory freeing routine is required. Transfers or computations can simply overwrite previous data in chunks.

Further, there are small messages for initialization and termination purposes. *Clear messages* are similar to *delete messages*. When such a message arrives, the partition array is reset to an empty state. It is important to always clear tile-local memories between graph executions or otherwise the runtime environment could process wrong data. Via another short message, namely the *exit message*, the driver tile allows a compute tile to exit the main loop. This is required to shut down the runtime environment.

Lastly, the driver tile sends *actor messages* to assign work to a compute tile. Here, the first step is to extract partition information from the message. The extracted partition represents the actor's output partition. After the actor was extracted from the message, the compute tile starts the actor execution. Based on the redundancy configuration, this actor may be executed redundantly. In the end, the driver tile is notified about the completed actor execution via a small message.

6.2.3 Driver Tile Dataflow Execution

The driver tile is responsible for managing dataflow graphs and schedules since it is the only tile with direct access to the large off-chip memory. However, because accessing the tile-local memory is faster than accessing the off-chip memory, only large data structures, i.e. dataflow graphs, schedules and partition data, are stored in the off-chip memory. Everything else, for example code and communication buffers, is stored in the driver's tile-local memory. Analogous to the partition array described in the previous section, the driver tile stores one partition array per compute tile. This allows the driver tile to keep track of all tile-local memories. Knowledge about tile-local memories is important for the driver to check whether a tile is ready to execute the next actor in its schedule and, therefore, whether an actor can be sent to the tile.

As in the shared-memory implementation, each partition node in the dataflow graph is linked to some runtime state data. The concrete attributes are depicted in Figure 6.6. Two members, `is_complete` and `delete_counter`, have pendants in the shared-memory implementation. The two other attributes keep track on which tiles a partition is available. This enables a faster lookup method than iterating through

runtime_data
is_complete: boolean
delete_counter: integer
holding_tiles: bitvector
in_off_chip_memory: boolean

Figure 6.6: Clustered architecture runtime state data

all partition arrays. Since runtime data members are only accessed by one core, there are no atomic operations are required.

The procedure of executing a section is shown in Algorithm 6.4. Variables cn and $dn[i]$ represent the next actor in the schedule of the driver tile and compute tile i, respectively. These variables are initialized with the first actor of each schedule (lines 2 to 6). After that the driver enters the main loop (lines 7 to 17) and stays in the loop until all schedules (driver tile schedule and compute tile schedules) are fully processed. Inside the loop, the driver repeatedly checks if any ongoing transfer is now complete (line 8), sends actors to compute tiles (line 11) and executes actors on its own (line 15). It is important to note that the check whether an actor is ready

Algorithm 6.4: Driver tile section execution with offline scheduling

```
1  function EXECUTE_DRIVER(cs : compute_tile_schedules, ds : driver_schedule)
2      nc ← number of compute tiles;
3      for i from 0 to nc − 1 do
4          cn[i] ← cs[i].BEGIN();
5      end
6      dn ← ds.BEGIN();
7      while actors in cs or ds remaining do
8          (dn, cn) ← POLL_NOC_EVENTS(cs, cn);
9          for i from 0 to nc − 1 do
10             if compute tile i has work to do and is idle and cn[i] is ready then
11                 EXECUTE_ON_COMPUTE_TILE(cn[i], i);
12             end
13         end
14         if driver tile has work to do and dn is ready then
15             (dn, cn) ← EXECUTE_ON_DRIVER_TILE(ds, cs, dn, cn);
16         end
17     end
18 end
```

only ensures that required partition data is available somewhere and, in case of redundant actor executions, that the result of the previous actor in the schedule has been verified. The check for partition data being in the correct memory takes place in the respective functions(`execute_on_compute_tile` and `execute_on_driver`). Further, the variables *cn* and *dn* in Algorithm 6.4 are reassigned via the functions `poll_transfer_events` and `execute_on_driver`. Under the assumption that all tiles (including the driver) are able to execute comparison actors, both variables may be modified by both functions since an unsuccessful comparison actor leads to a rollback in the schedule of up to three tiles (also including the driver).

6.2.4 Driver Tile Routines for Actor Execution

Algorithm 6.5 highlightes the procedure `execute_on_compute_tile` from Algorithm 6.4 in more detail. This function initiates the communication with a compute tile in order to execute an actor on this specific tile. If there are not enough free chunks on the target tile to send a partition or an actor, an evict routine is called. The evict function chooses a partition in the local memory of the given compute tile and sends an appropriate *delete message*. If there is no partition on the tile which is not required any more, a partition is transferred into the off-chip memory before the message is sent. Depending on the redundancy configuration, up to three chunks must be available on the target tile before the actor can be transferred. In case there are enough free chunks on the tile, the driver checks whether all required partitions are in its local memory. If this is not the case for a partition, the driver either starts an asynchronous data transfer (line 8) or sends a *partition message* (line 11) telling the target tile which other tile has the required partition in its tile-local memory (see Section 6.2.2). In case all requirements are met, the given actor can be transferred to the compute tile via an *actor message* by calling the according function (line 16).

As shown in Algorithm 6.4, actors may also be executed on the driver tile. All actors whose input or output data does not fit into a chunk of memory on the compute tiles definitely must be executed on the driver tile. Executing actors on the driver is unfavorable since the driver is unable to respond to notifications from compute tiles and to initiate asynchronous transfers during its own actor execution, which likely causes additional idle time on compute tiles. For most types of actors, an execution on the driver can be avoided by writing RAPID programs in which RAPIDs are divided into more but smaller partitions. However, `interval`, `collect` and `reorder` actors are always executed on the driver. For the former two types of actors, copy avoidance like described in Section 6.1.4 can be applied. Therefore, only `reorder` actors cause additional load on the driver tile.

Algorithm 6.6 shows the procedure that is called when the driver itself executes an actor. Similar to the procedure of actor execution on a compute tile (Algorithm 6.5),

Algorithm 6.5: Routine for actor executions on compute tiles

```
1  function EXECUTE_ON_COMPUTE_TILE(a : actor, c : compute_tile)
2      if not enough free chunks on tile c then
3          EVICT_PARTITION(c);
4      else
5          foreach required partition p of a do
6              if p not in local memory of tile c then
7                  if p in off-chip memory then
8                      SEND_DATA(p, c);
9                  else
10                     t ← FIND_TILE_WITH_PARTITION(p);
11                     SHIFT_PARTITION(p, c, t);
12                 end
13                 return;
14             end
15         end
16         SEND_ACTOR(a, c);
17     end
18 end
```

Algorithm 6.6: Actor execution routine on the driver tile

```
1  function EXECUTE_ON_DRIVER_TILE(ds : schedule, cs : schedules, dn : actor,
     cn : actors) : (actor, actors)
2      foreach required partition p of dn do
3          if p not in local memory of tile c then
4              t ← FIND_TILE_WITH_PARTITION(p);
5              REQUEST_PARTITION(p, t);
6              return dn;
7          end
8      end
9      EXECUTE_ACTOR(dn);
10     if dn is comparison actor and not all values match then
11         return ROLLBACK(ds, cs, dn, cn);
12     else
13         return (ds.NEXT(dn), cn);
14     end
15 end
```

the driver first checks whether partition data of all required partitions is in the off-chip memory and otherwise starts an asynchronous transfer (line 5). If all requirements are met, the actor is executed and a pointer to the next actor in the schedule is returned. In case the actor was a comparison actor and not all values match, a rollback routine which returns the new pending actors is called (line 11). This rollback routine requires all currently pending actors and all schedules as its arguments. The reason for this is that any tile may be involved in a redundant actor execution and therefore any of the pointers to pending actors may need to be modified. In this function, the driver tile also sends *delete messages* to all compute tiles whose data is considered as incorrect. When an actor is executed on three different tiles and one result does not match, the driver only sends a *delete message* to the corresponding tile.

6.2.5 Driver Tile Event Polling

The last remaining function from Algorithm 6.4 is `poll_noc_events`. Algorithm 6.7 shows how this function is implemented. There are basically two types of events that have to be polled repeatedly. On one hand, the driver needs to check if any of its asynchronous data transfers is complete, on the other hand there may be notification messages from compute tiles in its message queue. Notification messages from compute tiles can be either *actor done messages* or *transfer done messages*. Compute clusters send notifications of the latter type when they completed a data transfer from the local memory of another compute tile into their own local memory.

Events of asynchronous transfers are handled in lines 3 to 12. If the asynchronous transfer was a transfer from the off-chip memory into a tile-local memory, the driver sends a *partition message* to the corresponding tile to indicate that partition data is now in its tile-local memory. Then, the driver updates its knowledge about partition memories. In the second case, the asynchronous transfer was from a tile-local memory into the off-chip memory. Here, it is sufficient to update the partition's runtime data by marking it as available in the off-chip memory.

The message queue is checked in line 13. If there is a *transfer done message* in the queue, the driver only updates the corresponding partition's runtime data and its knowledge about the tile's local memory. In case of an *actor done messsage*, the driver also updates the corresponding memory array and runtime data and reassignes the pointer to the current actor (lines 23 to 24). There is also the possibility that the actor was a comparison actor and not all values match. In this case, the driver has to call the rollback routine (see Section 6.2.4) which updates the pending actors and sends *delete messages* to all tiles with incorrect or potentially incorrect data.

Algorithm 6.7: Polling routine on the driver tile

```
1  function POLL_NOC_EVENTS(ds : schedule, cs : schedules, dn : actor,
     cn : actors) : (actor, actors)
2      nc ← number of compute tiles;
3      for i from 0 to nc do
4          if asynchronous data transfer to compute tile i complete then
5              p ← partition, whose data was transferred;
6              SEND_PARTITION(p, i);
7              UPDATE_MEMORY_ARRAYS_AND_RUNTIME_DATA(p, i);
8          else if asynchronous data transfer from compute tile i complete then
9              p ← partition, whose data was transferred;
10             UPDATE_RUNTIME_DATA(p);
11         end
12     end
13     while message queue mq is not empty do
14         m ← mq.EXTRACT_MESSAGE();
15         t ← m.GET_TILE();
16         if m is transfer done message then
17             p ← m.GET_PARTITION();
18             UPDATE_MEMORY_ARRAYS_AND_RUNTIME_DATA(p, t);
19         else if m is actor done message then
20             if cn[t] is comparison actor and not all values match then
21                 (dn, cn) ← ROLLBACK(ds, cs, dn, cn);
22             else
23                 UPDATE_MEMORY_ARRAYS_AND_RUNTIME_DATA(cn[t], t);
24                 cn[t] ← cs[t].NEXT(cn[t]);
25             end
26         end
27     end
28     return (dn, cn);
29 end
```

6.2.6 Data-Parallel Actor Execution

The previous sections only described how actors are sent to compute tiles and explained that some actors are executed on the driver tile. Since each tile, possibly including the driver tile, contains multiple cores which are able to access the whole tile-local memory or, in case of the driver tile, the off-chip memory, there is another

level of parallelism the runtime environment can exploit. To utilize all cores of the clustered architecture, actors are executed in a data-parallel fashion. For most types of actors, this is easily possible because partitions usually contain more elements than there are cores on a tile and elements are processed individually. Figure 6.7 shows an example with a `map` actor which is executed in parallel by the four cores of a tile. The actor applies a RAPID function doubling its argument to all elements in the left partition. As shown in the example, the runtime environment distributes the elements as evenly as possible on all available cores.

It is important to note that the described data-parallel approach only works for element-wise actors. In partition-wise actors, the processing of an element may depend on other elements. To utilize as many cores as possible during the execution of such actors, the runtime environment provides a function that allows users to manually specify parallelism. This function represents a parallel for-loop. The declaration of this function is as follows:

```
parallel_for(start: integer, end: integer, step: integer,
             loop_function: integer → void)
```

The given function `loop_function` is executed once for each loop iteration. Since the loop is executed in parallel, there are no guarantees about the order of iterations. Analogous to the parallel execution of element-wise actors, loop iterations are distributed as evenly as possible to cores. It should be noted that, when using the parallel for-loop function, it is up to the user to ensure correct synchronization between the cores on a tile.

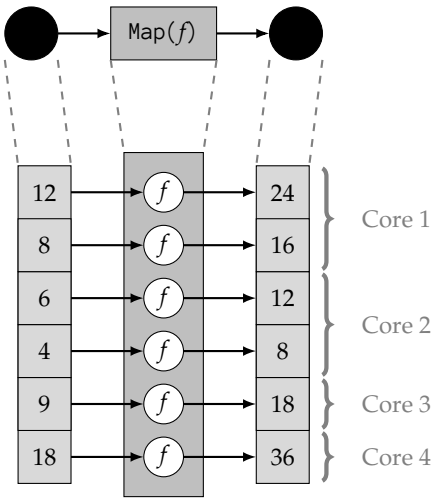


Figure 6.7: Parallel execution of a `map`-actor with RAPID function $f(x) = 2x$

6.2.7 Runtime Memory Management

When an actor is supposed to be executed on a compute tile, it might happen that there are not enough free chunks in the tile-local memory. If this is the case, one or more partitions in the local memory must be deleted. Since it is the only tile that is able to check data dependencies between actors and schedules, the driver tile determines which partitions are deleted. For this, an eviction policy is used. Different eviction policies influence the number of data transfers and therefore the time graph executions take. The runtime environment's eviction policy is as follows:

- Find a partition that is no longer needed
- Else find a partition whose data is already in the off-chip memory
- Else look one actor ahead in the schedule and find a partition which is not required by this future actor
- Else choose the first partition the pending actor does not depend on

The driver tile iterates through the corresponding memory array and tries to find a partition which is not needed anymore because all actors depending on this partition were already executed. If there is no such partition, the driver iterates through the array again to find a partition whose data is already in the off-chip memory. Such partitions have the advantage that a small *delete message* is sufficient and no data transfer is required. In case no suitable partition is found, the driver tries to keep the number of transfers low by looking at the tile's schedule. It therefore picks the actor right after the currently pending actor and iterates through the memory array a third time in order to find a partition this actor does not depend on. This prevents partitions that are required soon from being replaced. When an appropriate partition is found, its data must first be transferred into the off-chip memory before a *delete message* can be sent to the tile. Otherwise, the first partition without a connection to the currently pending actor is chosen.

Another aspect the driver has to consider are data transfers between compute tiles. If a tile t is receiving partition data from another tile u, the driver must not overwrite the affected chunk on tile u because otherwise t would receive wrong data. Therefore, the driver marks chunks with an active transfer as locked in its memory arrays. Partitions with data stored in locked chunks are then ignored in the eviction routine. However, this could lead to situations in which no partition on a tile can be deleted since the only possible choices are locked. When this happens, the driver has to wait until one of the transfers have finished proceeding with the eviction routine.

6.2.8 Additional Runtime Characteristics and Improvements

The dataflow runtime environment on clustered architectures supports the same execution modes as the already described shared-memory implementation. Redundant actors can be executed either on the same tile or different tiles. In contrast to the shared-memory runtime environment, redundant actor execution on the same tile has to detect some forms of permanent errors. On clustered architectures, each cluster contains multiple cores so that a faulty core can be detected by comparing its results with the results of other cores. Further, redundant execution on the same tile has the advantage that the amount of transferred data does not increase in comparison to a non-redundant execution.

Previous sections focused on actor execution with pessimistic redundancy (in case redundant actors are executed on different tiles). Analogous to the shared-memory implementation, an optimistic redundancy mode can also be implemented in the runtime environment for clustered architectures. Optimistic redundancy, of course, does not affect actors which are only executed on the driver tile.

Online scheduling is implemented analogous to the shared-memory runtime environment. In this mode the driver tile assigns actors dynamically to the tiles based on the online scheduling policy (work stealing in the proposed RTE implementation). The driver tile itself is also included in the process of online scheduling, i.e. it is possible for the driver to assign actors to itself.

The described runtime environment treats comparison actors as ordinary actors with input and result partitions. Whenever actors are executed redundantly on different tiles, all resulting partitions have to be sent to one of the executing tiles in order to compare the results. These transfers can be avoided if comparison actors are executed indirectly, for example by only comparing checksums of the data. Each tile is able to calculate the checksum of its computed data independent of other tiles and, therefore, checksum calculations can be executed in parallel. The subsequent comparison of checksums is fast and can be executed on the driver. To avoid idle times on compute tiles, comparison actors can be removed from the driver's fixed schedule since it is easily possible to include these short comparisons in the polling routine. Further, it is possible to keep the number of messages low by including the checksums in the payload of notification messages about finished actor executions.

Another way to reduce the number of messages in the NoC is to mark those partitions that are only required by a single actor. If such a marked partition is sent to a compute tile, the tile can safely delete the partition after the corresponding actor was executed or the result data of this actor was confirmed in case of a redundant actor. Based on the dataflow graph, this small optimization has the potential to greatly reduce the number of required delete messages.

6.2.9 Runtime Environment on Existing Hardware Architectures

Examples for existing hardware architectures that meet all assumptions from Section 6.2.1 are the *Kalray MPPA-256 Andey* [Din+14] and its successor, the *Kalray MPPA2-256 Bostan* [Sai+15]. Both include 16 compute tiles containing 16 cores each, a two megabyte tile-local memory and a NoC interface. In addition, there are two IO subsystems with 4 cores each, which have access to larger DDR memories and two IO subsystems with access to Ethernet interfaces. With regard to IO subsystems, Bostan differs from Andey since one DDR subsystem is coupled with one Ethernet subsystem so that there are only two larger IO subsystems in total. On both MPPAs, any IO subsystem with access to a DDR memory may be used as the driver tile. The NoC topology of both MPPAs is a bi-directional 2D torus in which IO subsystems have a wider NoC interface than compute tiles. This is beneficial for dataflow executions because it allows multiple simultaneous transfers between the driver tile and compute tiles at high speed. Especially the beginning of dataflow executions, when data must be initially sent to the compute tiles, and the completion, when the results are collected, benefit from parallel transfers.

6.3 Summary

In this chapter, two dataflow runtime environments based on the RAPID programming model were proposed. The first implementation targets shared memory architectures. On such architectures all cores can freely access the dataflow graph, offline schedules and online scheduling data structures. Since dataflow graphs are not modified during dataflow executions, synchronization is easy. The runtime environment supports five redundancy configurations, offline scheduling with optimistic or pessimistic execution modes and online scheduling.

On clustered architectures there is one driver tile and multiple compute tiles. Only the driver tile can access dataflow graphs and schedules. Therefore, the driver has to coordinate the dataflow execution. This includes keeping track of all tile-local memories. Compute tiles only execute actors that were transferred to them by the driver. The implementation on clustered architectures supports the same redundancy and scheduling modes as the shared memory runtime environment. The next chapter builds on the proposed runtime environments of this chapter and describes how offline and online scheduling can be implemented.

7

Scheduling

Scheduling in the proposed runtime environment (RTE) involves the assignment of actors to processing elements (PEs) and the order in which the actors are executed on the respective PE. Since scheduling is an important aspect in the RTE, some basic information about this topic was already provided in previous chapters. The proposed RTE is able to execute graphs either by following fixed schedules, which are computed before graph execution (offline), or in a more dynamic fashion with all scheduling decisions being made during graph execution (online). Using fixed schedules is the preferred way to execute graphs in the proposed RTE. The main benefit is the better analyzability compared to online scheduling. A disadvantage is that it is more difficult to react to unexpected events during runtime, like component failures. Performance-wise, none of the approaches is clearly superior. Provided that the actor runtimes and transfer costs are estimated realistically, common DAG scheduling heuristics are able to produce schedules which utilize the available hardware very well.

An important aspect which separates the proposed RTE from similar approaches is its support for adaptive fault tolerance. At first glance, adaptive fault tolerance contradicts the use of fixed schedules. Actor re-executions resulting from transient errors and redundancy changes during runtime are events which affect graph executions but cannot be predicted precisely. The previous chapters already mentioned that the RAPID programming model allows users to divide graphs into sections for

the purpose of specifying checkpoints during graph execution at which redundancy changes can occur. This chapter gives a more detailed insight into this topic and other characteristics of scheduling with regard to fault tolerance.

It should be noted that in this chapter we use the term *PE* for the hardware entities considered in the scheduling process. This means that PEs can be either cores, in case of shared-memory architectures, or tiles, in case of clustered architectures. Furthermore, special characteristics of hardware, for example memory constraints in the clustered architecture, can make hardware-dependent modifications of the standard scheduling approaches necessary.

The rest of this chapter is structured as follows. Section 7.1 shows which properties the RTE must provide to be compatible with a variety of scheduling heuristics. Extensions of standard DAG scheduling heuristics which are required to support concepts like different redundancy configurations and graph sections are discussed in Section 7.2. The HEFT algorithm serves as an example on how these modifications can be implemented. As mentioned above, the proposed RTE is also able to execute graphs in a more dynamic fashion with online scheduling. Section 7.3 provides details on this topic. Lastly, Section 7.4 describes possible ways to realize graceful degradation in case of faulty PEs. Since online scheduling approaches can easily react to the omission of a PE, the focus of this section lies on graceful degradation with offline scheduling.

7.1 Scheduling Prerequisites

In order to make appropriate scheduling decisions, it is important that scheduling heuristics are able to estimate the execution time of actors as well as data transfer times. Further, it is mandatory to consider the underlying hardware architecture and additional constraints due to redundancy in the scheduling process since otherwise non-executable schedules may be the result.

7.1.1 Properties Used in the Scheduling Process

There are many properties related to the dataflow graph and the underlying hardware architecture which can be used to implement a variety of scheduling algorithms. This section provides an overview of properties used by scheduling heuristics in the reference implementation for the two hardware architectures introduced in previous chapters. RTE implementations for other architectures may have to consider additional properties in order to produce reasonable schedules.

- **Number of elements in a partition**: Each partition node in the graph stores information about the memory requirements of its data and the size of a single

data element. This information can be used by the scheduling algorithm to estimate the runtime of an actor or the transfer latency of a partition.

- **Runtime estimation of RAPID functions**: Users can pass runtime estimations to the definition of RAPID functions. These estimations are functions whose exact type depends on the type of RAPID function. Scheduling algorithms use the estimation functions to compute runtime estimations of dataflow actors. The next section provides more details on this topic.

- **Data transfer rates**: In order to compute reasonable schedules it is important for scheduling algorithms to estimate the time it takes to transfer data from one PE to another. In the proposed RTE, transfer times are estimated from the number of transferred bytes and a constant bandwidth. Other typical properties which may be required for other hardware architectures are, for example, transfer initialization times or different bandwidths for different pairs of PEs.

- **Available processing elements**: Another important property is the set of PEs that scheduling algorithms are supposed to consider. For maximum performance this set should contain all PEs in the system. However, it may be useful to limit the set of considered PEs to a subset, for example if an external program is supposed to run next to the RTE on the hardware architecture or if some PEs should serve as spare components.

- **Memory constraints**: The NoC-based architecture we assume contains one small local memory for each tile and one large off-chip memory which is only accessible directly by a special driver tile. Therefore, scheduling algorithms have to make sure that actors processing large partitions are assigned to the driver tile. Shared-memory architectures do not require such a constraint.

- **Constraints because of redundancy**: Some redundancy configurations can introduce additional constraints. An example would be the restriction that each comparison actor must immediately follow one of the corresponding redundant actors in the schedule, which is necessary for the pessimistic redundancy mode. Section 7.2.2 provides more details about scheduling with regard to different redundancy configurations.

The described properties provide enough information to implement many DAG scheduling heuristics. The RTE's reference implementation uses the HEFT scheduling algorithm described in Section 2.3.3. Additionally, the insertion-based policy of HEFT can be disabled in the RTE. This makes the scheduling process faster but may produce schedules with larger *makespans*, i.e. longer schedules. For some

redundancy-related features, the insertion-based policy must be disabled to ensure a correct graph execution. The following sections will highlight this in more detail.

Actor re-executions due to errors are currently not considered by the implemented HEFT algorithm. This has two reasons. First, errors are quite rare under normal circumstances, and scheduling algorithms should therefore compute schedules which perform well when no errors occur. Second, errors are not accurately predictable, and the number of re-executions per error varies since there are different types of errors. Transient errors can be resolved by a single re-execution. Permanent errors, on the other hand, require multiple re-executions to be identified and are more expensive to resolve. This makes it difficult to estimate re-executions in the scheduling process.

7.1.2 Runtime Estimation of Dataflow Actors

The process of estimating an actor's execution time depends on the type of the actor. In particular, estimating the runtime differs for actors applying partition-wise RAPID functions and those applying element-wise functions. Further, actors that do not apply a RAPID function at all have to be treated specially.

Element-wise RAPID functions are applied to elements or tuples of elements (in case of `combine` and `zipmap`) individually. Therefore, the first step is to determine a runtime estimation for a single function application by passing the index of the PE which is supposed to apply the function to the user-provided estimation function. Then, the runtime of a corresponding actor can be estimated by multiplying this value with the number of function applications. The latter is in many cases equal to the number of elements in the actor's input partitions. In fact, the only exception is `combine`, which processes n elements from the same partition in a single RAPID function call for a given n.

The procedure described so far only holds for shared-memory architectures. For clustered architectures another step is required since actors are executed in a data-parallel manner. Therefore, the number of function applications has to be divided by the number of cores in the corresponding tile (with rounding up to the next integer).

Partition-wise RAPID functions differ from element-wise functions as users have to specify manually how the elements in a partition are accessed. In contrast to element-wise functions it is, for example, possible to iterate over all elements in a partition multiple times in different order. To avoid duplicate code, partition-wise RAPID functions are not bound to firm partition sizes but can be applied to arbitrary

large partitions. For a better runtime estimation of a corresponding dataflow actor, the definition of a partition-wise function expects an estimation function with an additional parameter. With this estimation function, a user specifies how long one execution of the RAPID function takes depending on the processing element executing the function and the number of elements in the input partitions. During scheduling, all partition sizes are known, so scheduling algorithms can execute the user-specified estimation functions for the actors to compute a concrete value representing the actor's runtime.

Similar to the element-wise functions, the data-parallel actor execution on distributed architectures has to be considered for an accurate runtime estimation. Since users have to manually specify the parallelism inside a partition-wise function, it is also up to the user to provide an appropriate estimation function.

Actors not applying a function are estimated by scheduling algorithms without user-provided values or functions. These actors only copy the data of partitions from one memory location to another, for example to change the element order or to create a contiguous sequence of values. Thus, an estimation depends on the number of copied bytes. Like described in Section 6.1.4, it is often possible to avoid copy operations for many actors that do not apply a function. If such optimizations are applied, the execution of corresponding actors only takes a very short constant amount of time. It is important that scheduling algorithms are aware of the implemented optimizations in order to estimate this type of actors accordingly.

7.2 Offline Scheduling

For graph executions with fixed schedules, the proposed RTE aims at bridging the gap between analyzability and fault tolerance. On one hand, the RTE should be able to dynamically react to unpredictable events, like actor re-executions and redundancy changes, but on the other hand, it should be possible to determine the worst case execution time of graph executions without large overestimation. Some aspects about offline scheduling in the proposed RTE were already mentioned in Chapter 6. Graph executions according to fixed schedules do not follow exact timings. Instead, the RTE only considers information about the assignment of dataflow actors to PEs and the order in which the PEs execute these actors. The actual time an actor is executed at runtime depends on the availability of data.

This approach is a trade-off between completely static schedules (with exact timings), which are rigid and do not allow actor re-executions, and graph executions with online scheduling, which are difficult to analyze statically. A static actor assignment and execution order can be used in the analysis of a graph execution,

while the lack of exact timings allows the RTE to insert additional actor executions into the schedule at runtime whenever an error is detected and a re-execution is required.

In the course of this section, scheduling under different redundancy configurations and on different hardware architectures is examined. Some configurations and architectures require a modification of the standard DAG scheduling heuristics. Such modifications depend on the specific algorithm. In the following, the HEFT algorithm serves as an example. Since the NoC-based architecture considered in this thesis has memory constraints which are not covered by standard HEFT, Section 7.2.1 describes how the HEFT algorithm can be made compatible. Section 7.2.2 then discusses the different redundancy configurations.

7.2.1 Scheduling under Memory Constraints

In the considered NoC-based architecture each compute tile only has direct access to a small local memory (with, for example, a few hundred kilobytes), which contains a statically allocated memory region for partition data. This region is divided into a fixed number of uniform slots. Thus, there are two scenarios preventing actors from being executed on compute tiles. The first scenario concerns actors with too many inputs. If the number of input partitions plus the actor's single output partition is higher than the number of partition slots, the actor must be executed on the driver. In practice, this usually applies to `interval`, `collect` and `reorder` actors. For the former two types of actors, copy avoidance strategies like those described in Section 6.1.4 can be applied so that only `reorder` actors generate workload on the driver. The second scenario covers actors which either require or produce an oversized partition. Since partition sizes depend on the application program, this might be the case for any type of actor.

Modifying HEFT to consider memory constraints is rather straightforward. Since the dataflow graph contains information about the memory requirements for all partitions as well as all data dependencies, actors that have to be executed on the driver tile can be identified easily. Algorithm 7.1 shows the modified HEFT algorithm for the assumed NoC-based architecture. The additional lines compared to standard HEFT are highlighted. A single conditional block checking the number of inputs and memory requirements in the outer loop is sufficient because compute tiles are homogeneous in the considered architecture. For more heterogeneous architectures with differently sized local memories additional conditional blocks in the inner loop (lines 12 to 14) would be required.

Algorithm 7.1: HEFT scheduling on the NoC-based architecture

```
1  function HEFT_NOC-BASED(s : graph_section)
2      compute mean values for all node and edge weights in s;
3      compute the upward rank for all nodes in s;
4      create a sorted list of nodes by nonincreasing order of upward ranks;
5      while the scheduling list is not empty do
6          let n be the first node in the list;
7          remove n from the list;
8          if n has too many inputs or n processes too much data then
9              compute the earliest finish time (EFT) of n on the driver tile;
10             assign n to the driver tile;
11         else
12             for each compute tile t do
13                 compute the EFT of n on compute tile t;
14             end
15             assign n to the tile with the lowest EFT;
16         end
17     end
18 end
```

7.2.2 Support for Different Redundancy Configurations

The adaptive fault tolerance concept requires that actors can be executed in at least two different redundancy configurations. The RTE supports five different redundancy configurations. An overview of available configurations was already given in Section 6.1.2 but for convenience they are listed below:

- Non-redundant execution
- Double execution on different PEs
- Triple execution on different PEs
- Double execution on the same PE
- Triple execution on the same PE

Redundancy over multiple PEs

Different redundancy configurations require their own schedule since the number of actors varies. To compute schedules with redundancy over multiple PEs, scheduling algorithms must be altered. Otherwise, the heuristics could assign redundant

actors to the same PE. For the HEFT scheduling heuristic, such a modification is easy. Algorithm 7.2 shows the extended algorithm for the case of two redundant actor executions. The general principle behind this extension also works for three redundant executions per actor. As before, additional lines are highlighted. By default, HEFT iterates over all graph nodes and considers all PEs as possible candidates for actor assignments. In order to generate suitable redundant schedules, it is necessary to check for each actor whether one of its associated redundant actors is already assigned to the currently considered PE so that this PE can be excluded temporarily from the scheduling process. Line 12 in Algorithm 7.2 contains the corresponding check. Similarly, the modified HEFT heuristic assigns comparison actors always to a PE which also executes one of the redundant actors (line 14).

The modified HEFT always schedules redundant actors and their associated comparison actor in direct succession (line 10). When HEFT is applied to a graph

Algorithm 7.2: HEFT scheduling with redundant executions

```
1  function HEFT_REDUNDANT(s : graph_section)
2      compute mean values for all node and edge weights in s;
3      compute the upward rank for all nodes in s;
4      create a sorted list of nodes by nonincreasing order of upward ranks
       ignoring all redundant and comparison nodes;
5      while the scheduling list is not empty do
6          let n be the first node in the list;
7          remove n from the list;
8          let n' be the redundant node belonging to n;
9          let n_c be the comparison node belonging to n and n';
10         for x in [n, n', n_c] do
11             for each PE p do
12                 if x is n' and n was assigned to p then
13                     proceed with next PE;
14                 else if x is n_c and both n and n' were not assigned to p then
15                     proceed with next PE;
16                 end
17                 compute the earliest finish time (EFT) of x on PE p;
18             end
19             assign x to the PE with the lowest EFT;
20         end
21     end
22 end
```

without the insertion-based policy, this ensures that each comparison actor is the direct successor of one of the redundant actors in the schedule. Despite the modification, actors are still scheduled in a topological order so that correct schedules are produced. Furthermore, redundant actors and comparison actors can be skipped when the scheduling list is created.

With regard to the different redundancy configurations, it is also useful to adjust the EFT computation. As described in Section 2.3.3 the EFT is usually computed with the formula:

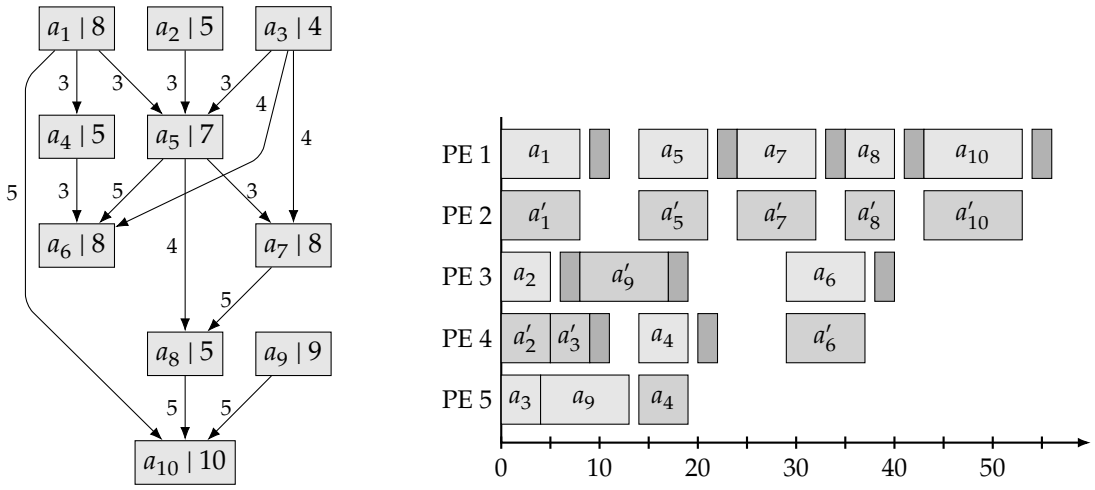
$$EFT(n_i, p_j) = w_{i,j} + \max\left\{avail[j], \max_{n_m \in pred(n_i)}(AFT(n_m) + c_{m,i})\right\}$$

This formula only considers direct predecessors for communication costs. In redundant sections, the direct predecessors of non-comparison actor are comparison actors. Therefore, it makes sense to also consider the preceding actors of the comparison actors since the corresponding PEs performed the actual computation and have the data in their local memory or cache.

It should be noted that the described modified HEFT algorithm in Algorithm 7.2 is only suitable for the shared-memory architecture. In order to get an appropriate algorithm for the NoC-based architecture, the concepts from Algorithm 7.1 and Algorithm 7.2 have to be combined. The scheduling heuristic must also consider optimizations like the dynamic execution of checksum-based comparisons on the driver tile which was described in Section 6.2.8.

Another important aspect is related to the two execution modes our RTE supports. In pessimistic mode, a PE waits until the results of the current actor have been compared before it continues with the next actor. Thus, it is important to run the extended HEFT scheduler without the insertion-based policy because otherwise a cyclic dependency, i.e. a deadlock, during graph execution could occur. In optimistic mode where PEs do not have to wait there is no such restriction, and it is possible to use HEFT with the insertion-based policy.

An example schedule computed with the modified HEFT algorithm is shown in Figure 7.1. On the left side, a DAG with node and edge weights is depicted. In favor of a more concise illustration, redundant and comparison nodes are excluded from the figure. In analogy to the hardware architectures which are supported by the RTE's reference implementation, all PEs in this example are identical so that no computation cost matrix is required. Further, we assume that duplicate actors have the same node weight as their pendant, comparison nodes have a weight of 2 and edges between redundant nodes and comparators have a weight of 1 (for example because only checksums instead of actual data are transferred and compared). The redundant schedule created by the described variant of HEFT is shown on the right. Its makespan is 56. The small dark boxes in the schedule represent the execution of



a) Example graph with node and edge weights

b) Redundant schedule generated with modified HEFT heuristic

Figure 7.1: Redundant schedule example

comparison nodes. In this example, each comparison node in the schedule compares the result of its predecessor node (which is why the schedule is unambiguous even though comparison nodes are not labelled). In this example, the modified HEFT has successfully identified the critical path $(a_1, a_5, a_7, a_8, a_{10})$ and assigned the nodes in the correct order to the first and second PE.

Redundancy on the same PE

For redundant executions on the same PE, it is possible to use standard DAG scheduling heuristics. Redundant actors and the following comparison or voting actor are executed without interruption by other actor executions or transfers and can thus be considered as one longer lasting execution in the scheduling process. In contrast to redundant execution over different PEs, these redundancy configurations have the advantage that fewer transfers are required. But despite the reduced number of transfers, redundancy on the same PE may not be beneficial to the performance of graph executions. An example is shown in Figure 7.2. This schedule is suitable for the graph in Figure 7.1a and consists exclusively of redundant actor executions on the same PE. In this case, the schedule's makespan is 86, and adding more PEs would not help to lower the makespan. The critical path in the example graph is dominant, and there is not much parallelism due to the data dependencies. Therefore, the makespan of the redundant schedule in Figure 7.1b (which is 56) is lower despite the additional transfers. Redundancy on the same PE can only improve the

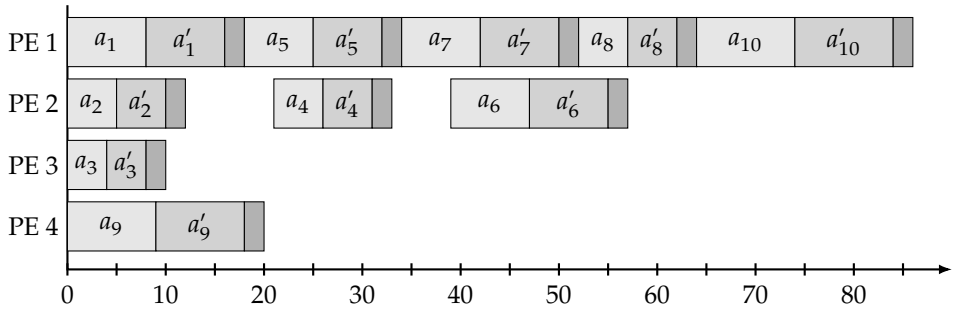


Figure 7.2: Schedule for the graph from Figure 7.1a with redundant actor executions on the same PE

performance if the degree of parallelism in the graph is high enough to utilize the available PEs.

Since schedules with redundancy on the same PE are generated with standard heuristics, it is possible to use non-redundant schedules for the respective redundancy configurations. The downside of this approach is that the non-redundant schedule is possibly less suited for a redundant execution than a separately computed schedule because of the difference in the ratio of transfer to computation. However, this approach is chosen in the RTE's reference implementation since it lowers the overall scheduling workload and redundancy across multiple PEs is the preferred form of redundancy in the RTE.

7.2.3 Simultaneous DAG Executions

The proposed RTE considers graphs as sets of nodes and edges and executes actor nodes in the order provided by the schedule. It is not required that graphs are weakly connected, i.e. DAGs may consist of multiple unconnected parts. This makes the RTE compatible with applications involving multiple simultaneously executed DAGs. To schedule unconnected DAGs, the HEFT implementation provided by the reference implementation uses the same standard process that is commonly used to schedule graphs with multiple entry or exit nodes. Before the actual HEFT scheduling heuristic is applied, the (unconnected) graph is temporarily extended by two nodes with a weight of zero as shown in Figure 7.3. The first node has outgoing edges to the entry nodes of all independent subgraphs, while the second node has incoming edges from the exit nodes of the subgraphs. Analogous to the new nodes, the additional edges have a weight of zero. Since the resulting graph represents a weakly connected DAG, the actual HEFT scheduling heuristic can be used without a further modification.

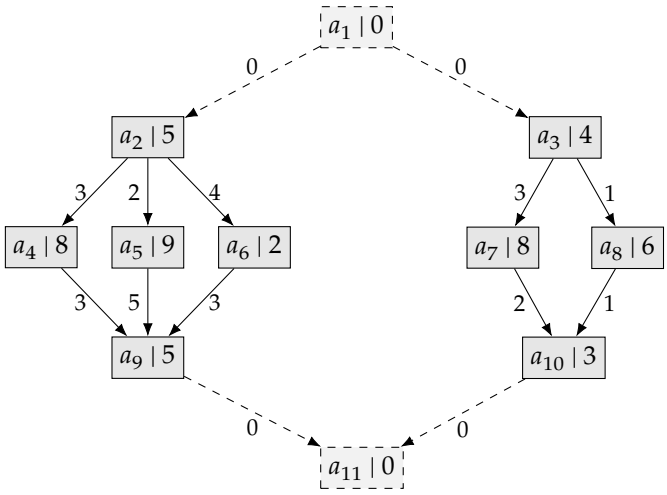


Figure 7.3: Composition of two DAGs by introducing new entry and exit nodes

The described method is the easiest way to extend scheduling so that arbitrary heuristics are able to schedule multiple graphs. A disadvantage of this approach is that it does not consider any fairness aspects in the scheduling process. There are, however, other more complicated methods which can be applied to scheduling heuristics and which do consider fairness aspects. Zhao and Sakellariou compared six different approaches (including the basic approach described above), which were applied to two scheduling heuristics (with one being HEFT), with regard to fairness and makespans [ZS06]. The authors define fairness via differences in slowdowns of the individual graphs resulting from the simultaneous execution. Since the proposed RTE is highly customizable, a user can easily extend a scheduling routine implementation in case an application requires fairness in a simultaneous graph execution.

7.2.4 Impact of Graph Sections

As mentioned in previous chapters, graphs can be divided into different sections which are scheduled independently. These sections are executed one after another with barriers in between. The barriers serve as checkpoints to switch to a different redundancy configuration, i.e. to a different schedule. It is important to choose the section boundaries and the order in which sections are executed in a way that the graph remains executable. The `checkpoint` function from the RAPID programming model guarantees a correct sectioning, but this has to be ensured externally when the RTE's graph import functionality is used. If a graph consists of multiple sections, it is possible to check whether the sectioning is correct by looking at the dependencies between sections. There is a dependency between two sections a and b (with

$a \neq b$) if and only if there is a data dependency between an actor node from a and an actor node from b. A correctly sectioned graph must not contain any cyclic dependencies between sections, i.e. the dependencies between sections must also form a DAG. Otherwise, it would not be possible to specify an order in which sections are executed. In the following, the DAG consisting of sections as its nodes and section dependencies as its edges is called *Meta-DAG*. After verifying that the sectioning of a graph forms a correct Meta-DAG, the execution order of sections (given by their indices) must be checked. To be correct, the order of sections must follow the dependencies between sections, i.e. it must represent a topological sort of the Meta-DAG.

Graph sections are always DAGs on their own. Therefore, the described scheduling techniques can be applied to individual sections without any modifications. Scheduling sections independently clearly has an impact on the performance of graph executions. This has two reasons. First, the more sections a graph is divided into, the less information about the overall graph structure scheduling heuristics can use. Second, sections prevent scheduling algorithms to optimize the execution order of actors over section boundaries. These two factors lead to a high probability that the execution of a graph with many small sections takes longer than the execution of the same graph with fewer but larger sections.

7.3 Online Scheduling

Even though graph executions following fixed schedules are preferred in the RTE, there is also support for online scheduling. In online scheduling mode, decisions about the assignment and execution order of actors are made during graph execution. The focus here is solely on adaptability and performance. If application has requirements regarding the analyzability of graph executions, it is advisable to utilize statically computed schedules instead since online scheduling approaches are usually more difficult to predict.

In the RTE's reference implementation, dynamic scheduling follows the *work stealing* approach described in Section 2.4. For standard, non-redundant dataflow executions on shared-memory architectures, there are no major differences between the work stealing implementations in the proposed RTE and other frameworks. But as for the HEFT algorithm, implementations on other hardware architectures require some extensions to the standard work stealing procedure. Further, the different redundancy configurations the RTE supports lead to additional modifications of the standard routine.

7.3.1 Work Stealing on Different Hardware Architectures

As described in Section 2.4, work stealing is usually based on double-ended queues. On shared-memory architectures, each core has access to all scheduling queues and the dataflow graph. Thus, cores can identify actors which are ready to be executed and insert them into their queue by themselves. Since other cores may access the queue concurrently due to the stealing of elements, synchronization between cores is required. In many proposals, double-ended queues are implemented as lock-free data structures. This has the advantage that it allows simultaneous access on both ends of the queue. However, since critical sections are rather short, the RTE's reference implementation uses standard queues protected by locks. Proper synchronization is also required to identify which actors are ready to be executed because otherwise an actor could be inserted by multiple cores. In the reference implementation this is realized through the `ready_counter` (see Section 6.1.1). Each actor has its own `ready_counter` which is initialized with the number of predecessors in the graph. The counter is decreased atomically with a fetch-and-add instruction each time one of the preceding actors has been executed. This ensures that only one core detects the counter reaching zero.

In case of the considered clustered architecture, there is no synchronization required. Since compute tiles do not have access to the dataflow graph, the driver is the only tile which can identify if an actor is ready. As a result, the driver tile has to manage all queues and run the work stealing procedure on behalf of the other tiles. The driver has also a special role in the scheduling process since it is responsible for the execution of actors processing large partitions. Therefore, actor nodes must not be stolen from the driver's queue. Further, to ensure that the driver can focus on managing NoC communication and other driver-specific tasks, the driver itself should not steal actors from the queues of other PEs. Instead, actors which have to be executed on the driver must be identified as soon as they are ready and inserted in the driver's queue. So for this special tile, the work stealing aspect has to be undermined in favor of an approach more closely to work sharing.

7.3.2 Redundancy and Fault Tolerance

The different redundancy configurations also require modifications in the work stealing procedure. As with offline DAG scheduling heuristics, there are no modifications necessary for configurations with redundant actor executions on the same PE. Algorithms 7.3 and 7.4 show modified insertion and stealing routines for the case of two redundant actor executions. Corresponding functions for three redundant actor executions are analogous.

Algorithm 7.3: Work stealing insertion routine with redundant actors

```
1  function INSERT_REDUNDANT(a : actor, p : PE)
2      insert a at the front of p's queue;
3      if a is comparison actor then return;
4      s ← empty set;
5      l_p ← list of predecessor actors of a;
6      for actor a_p in l_p do
7          l ← list of PEs which have executed a_p or one of its redundant actors;
8          insert all PEs from l into s;
9      end
10     remove p from s if present;
11     if s is not empty then
12         p_r ← random PE from s whose queue contains the least amount of
             actors;
13     else
14         p_r ← random PE which is not p and whose queue contains the least
             amount of actors;
15     end
16     insert a's redundant actor at the front of p_r's queue;
17 end
```

The insertion routine in Algorithm 7.3 first inserts the given actor a into the PE's queue, just like in the case of standard work stealing. For comparison actors there is nothing else to do since actors of this type have no redundant actors. Otherwise, a PE for the redundant actor has to be chosen. To preserve data locality, the routine iterates over all predecessors of a and inserts all PEs that have executed a predecessor into a temporary set s. It is important to note that these predecessors may have a different redundancy since they are not necessarily in the same graph section. This is considered in line 7. After the set of possible candidates has been identified, the routine chooses the one whose queue contains the least amount of actors (line 12). If the queue containing the least amount of elements is not unique, a random one of them is chosen. There is also the chance that s is empty. This can only happen if a has no predecessor actors or all predecessors were executed non-redundantly by p itself. In this case, the procedure is similar. The only difference is that all PEs besides p are considered as possible targets (line 14). Lastly, the redundant actor is inserted into the corresponding queue (line 16).

Stealing (Algorithm 7.4) also differs from the non-redundant case since it has to consider which actors are in the thief's queue and which actor it has already

Algorithm 7.4: Stealing routine with redundant actors

```
1  function STEAL_REDUNDANT(p : PE)
2      l ← ordered list of all PEs with non-empty queue;
3      shift the elements in l in a cyclic manner x times for a random x;
4      for p_c in l do
5          a ← last element in p_c's queue;
6          a_r ← a's redundant actor;
7          if a_r is not in p's queue and a_r was not executed by p then
8              move a into p's queue;
9              return;
10         end
11     end
12 end
```

executed. For the given thief p, the routine first creates a list l of all PEs whose queue is not empty. This implicitly ensures that p's own queue is not considered for stealing. Elements in the list are then shifted x times in a cyclic manner for a random x. Because of the shift, the RTE iterates over the list beginning with a random element (line 4). In each iteration, the procedure checks whether the last element in the corresponding queue is suitable for being stolen. This is the case if its redundant actor is not in p's queue and was not executed by p. As soon as an appropriate actor is identified, it is stolen and the procedure is exited (lines 7 to 10). Because only the last element in a queue is considered for stealing, there is a chance that the described routine does not find an actor for stealing even though at least one of the queues contains a suitable actor. However, this has no drastic effect on the performance since PEs do not only receive work by stealing but also through the scheduling of redundant actors (see Algorithm 7.3).

With regard to redundancy and fault tolerance, online scheduling has some advantages over offline scheduling. Redundancy changes and component failures are unpredictable events and therefore difficult to bring in line with fixed schedules. As described in previous sections, redundancy changes always affect all actors in a section to keep the number of schedules reasonable. In online scheduling mode, however, no such restriction is required and the RTE could be extended quite easily to support redundancy changes of individual actors. This would remove the necessity of barriers between sections since no replacements of the active schedule are required. However, graph sections and the barriers in between are still useful since they allow the RTE to switch from online scheduling to fixed schedules when a barrier is reached.

7.4 Graceful Degradation in Case of Component Failure

Permanent failure of an actively used component can be difficult to handle. This section discusses component failure from a general scheduling perspective. Impacts of malfunctioning PEs not related to scheduling would be, for example, the loss of data. The focus of this section lies on faulty PEs, i.e. PEs that, from an arbitrary point during graph execution, do not respond anymore or regularly produce incorrect data and are therefore considered as malfunctioning. For the clustered architecture, we assume that permanent faults do not affect the driver tile.

If a graph is executed using the work stealing approach described in the previous section, adding graceful degradation is straightforward. The work stealing procedures can simply ignore malfunctioning PEs in the scheduling process. However, if analyzability is a requirement and the RTE executes graphs using statically computed schedules, it is more difficult. The main problem is that faulty PEs cannot execute actors anymore but appear in schedules and thus have a workload assigned to them. So in order to ensure progress in the system, other PEs have to take over the workload of a faulty one. There are multiple ways to add graceful degradation despite the use of fixed schedules, each with benefits and drawbacks. In the rest of this section, some possible ways are described.

7.4.1 Spare Processing Elements

One possible solution is to have spare PEs in the system. Spare PEs can replace actively used PEs that have become faulty. An advantage of this approach is that the schedules can still be used in case a PE fails. Furthermore, if the replacement PE is identical to the faulty PE, the performance of the system stays the same. Following this approach it is easily possible for the system to cope with multiple faulty PEs during its lifetime as long as enough spare PEs are available. The disadvantage is that not all functional PEs in the system are continuously utilized and thus either the performance is lower than in other approaches or the hardware costs are higher.

7.4.2 Spare Schedules

A different solution is to use all available PEs in the system by default and provide additional schedules that require only a subset of all PEs. If a permanent error in some PE is detected, the system can change to a different schedule in which no actors are assigned to the faulty PE. Besides the advantage that the available hardware can be fully utilized, there are also disadvantages. Hardware constraints

can make the number of required spare schedules quite large, especially if all redundancy configurations should be still available after some PE became faulty. On relatively homogeneous systems, like the considered shared-memory or NoC-based architecture, PEs are interchangeable. Thus, only one set of spare schedules is enough to cope with the failure of an arbitrary PE. In heterogeneous systems, however, it may not be possible to run every actor on every PE, so multiple sets of spare schedules may be required to cover all cases of failing PEs.

7.4.3 Rescheduling at Runtime

Another possibility is to run the scheduling algorithm whenever a PE becomes unusable. Advantages compared to the approaches considered so far are that the available hardware is always fully utilized and no additional memory is required. The disadvantage is that rescheduling at runtime greatly increases the recovery time in case of a permanent error. How long the process of rescheduling takes also depends on the hardware architecture. A prerequisite for DAG scheduling heuristics is the ability to navigate through the graph. On the assumed clustered architecture, only the driver tile has access to the off-chip memory containing graphs and schedules. So for this hardware architecture, most of the computing power cannot be used in the scheduling process.

7.4.4 Modification of Existing Schedules

The availability of different schedules due to the adaptive fault tolerance capabilities in the proposed RTE allows a fourth solution. In the available redundant schedules, each actor is executed by at least two different PEs. Therefore, if one PE fails, new schedules can be obtained by removing actors from existing schedules. More specifically, when a permanent error is detected, a non-redundant schedule can be created from a schedule with two executions for each actor by processing the following steps:

1. Remove the faulty PE from the schedule
2. Remove all comparison actors from the schedule
3. Remove duplicate actors evenly from the schedule

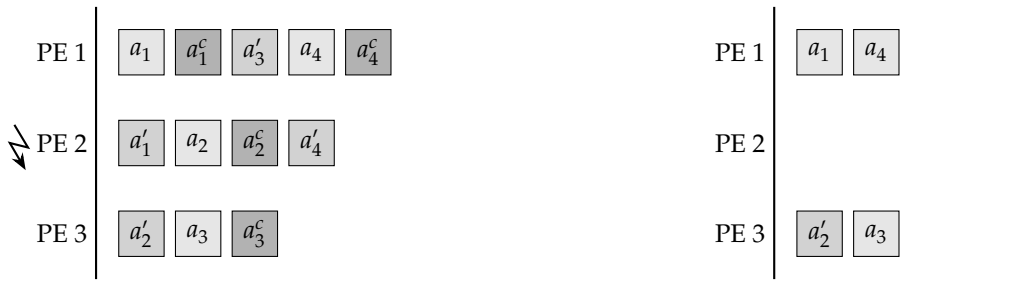
In step 3, duplicate actors should be removed from the schedule evenly, i.e. the work should be divided as evenly as possible between the functional PEs.

An example is shown in Figure 7.4. Since dataflow executions do not depend on exact timings, only the actor assignment and order are depicted. The left side of the figure consists of a redundant schedule where each of the four actors (a_1 to a_4) is

executed on two different PEs. Out of the three PEs, the second one is considered as faulty by the system. To obtain a new schedule from the existing one, all actors assigned to the faulty PE as well as all duplicate actors and comparison actors assigned to other PEs are removed. This leads to the schedule in Figure 7.4b which does not utilize the faulty PE and has no redundant actor executions anymore.

Analogous to this example, it is possible to modify a schedule with three redundant actor executions in order to obtain a schedule with two redundant executions per actor. In case the results of redundant actors are compared dynamically (see Section 6.2.8), the procedure can be applied as described above. However, if comparison actors are part of the schedule, some extra considerations are required. The reason for this is that comparison actors are necessary for the graph execution but might be assigned to the faulty PE. These comparison actors have to be moved to other PEs which is not trivial. Moving such an actor directly after one of the two remaining redundant actors can lead to a cyclic dependency and thus a non-executable schedule. There are two possible solutions to solve this issue. The first solution is to check all data dependencies when moving a comparison actor. This, however, leads to a more complex reconfiguration and thus increases the reaction time when a permanent error is detected. The second solution is to compute the schedule with three redundant actor executions in a way so that moving comparison actors is easy. In case of HEFT, this can be accomplished by assigning redundant actors and their comparison actors always in direct succession and by placing actors always at the end of the PE's schedule (i.e. run HEFT without the insertion-based scheduling policy).

Modifying existing schedules has the same advantages as rescheduling, i.e. the hardware is always fully utilized and there are no increased memory requirements due to additional schedules. Further, the modification of existing schedules is faster than computing entirely new schedules by running the scheduling algorithm. Modifying an existing schedule like shown in the example only requires iterating



a) Schedule before modification, PE 2 is faulty b) Pruned schedule without PE 2

Figure 7.4: Modification of a redundant schedule as a result of a faulty PE

through the schedule once. An implementation of the HEFT algorithm, for example, has to iterate through the graph at least twice and consider each PE as a candidate for each actor.

However, a disadvantage is that the available degree of redundancy is reduced when a PE becomes faulty, which may be unacceptable in highly critical environments. Furthermore, it is likely that the makespan of a modified schedule is higher compared to a schedule which was computed by a reasonable scheduling algorithm if both schedules utilize the same number of PEs.

7.4.5 Comparison of Graceful Degradation Approaches

The previous sections described various approaches to realize graceful degradation in case of a faulty PE. For graph executions with work stealing, adding support for graceful degradation is straightforward. Regarding graph executions following pre-computed schedules, graceful degradation can be realized in different ways. The advantages and disadvantages of each approach are summarized in Table 7.1. Spare PEs and spare schedules are simple ways to ensure graceful degradation. Main drawbacks of the former are that, at least as long as no permanent error occurs, the hardware is not fully utilized and that heterogeneous systems will likely require more than one spare PE. Spare schedules on the other hand allow the system to fully utilize the available hardware but permanently occupy some amount of memory. The more distinct types of PEs a system contains, the more spare schedules are required in order to cope with arbitrary failing PEs. In contrast to spare PEs and spare schedules, rescheduling at runtime has no drawbacks as long as no permanent

Table 7.1: Comparison of graceful degradation approaches with offline scheduling

Approach	Advantages	Disadvantages
spare PEs	no additional memory required, very fast reconfiguration	hardware not fully utilized, expensive for heterogeneous systems
spare schedules	hardware fully utilized, very fast reconfiguration	higher memory requirements (especially on heterogeneous systems)
rescheduling at runtime	hardware fully utilized, no additional memory required	slow reconfiguration
modification of existing schedules	hardware fully utilized, no additional memory required, fast reconfiguration	lower performance after permanent error, decrease in redundancy after permanent error

error occurs. The disadvantage lies in the slow error recovery since running a scheduling heuristic takes much longer than activating a spare PE or switching to a spare schedule. Modifying existing schedules also has no drawbacks under normal circumstances. In comparison to rescheduling, the reconfiguration effort is much smaller. The downsides are a reduced performance in case a permanent error occurs since the modified schedules usually have a higher makespan than computed schedules and a reduction in the available degree of redundancy, which might be unfavorable in environments with high safety requirements.

7.5 Summary

This chapter highlighted the different aspects of graph scheduling in the proposed RTE. To give users the opportunity to implement a variety of scheduling techniques, the RTE provides access to different properties of graph nodes and the underlying hardware architecture. Standard scheduling heuristics must be modified to be able to generate redundant schedules and to consider the memory constraints of the clustered architecture the RTE targets. The previous sections described extensions of HEFT as an example.

Besides graph execution following pre-computed schedules, the RTE also supports graph executions with online scheduling. In the reference implementation, this kind of graph execution is based on work stealing, a technique commonly used in parallel computing frameworks and dataflow systems. As for the DAG scheduling heuristics, work stealing has to be extended to support the different hardware architectures and redundancy configurations.

The last part of this chapter discussed graceful degradation from a scheduling perspective. While extending work stealing is easy, graceful degradation is more complex to implement when graph executions follow fixed schedules. In this regard, four different approaches, each with its benefits and drawbacks, were discussed.

As mentioned at the beginning of this chapter, by supporting fixed schedules, the RTE bridges the gap between analyzability and fault tolerance. The next chapter provides information on the analyzability of graph executions when fixed schedules are used.

8

Analyzability

This chapter focuses on the analyzability of dataflow executions with varying degrees of redundancy on different types of hardware architectures, in particular the two types of hardware architectures introduced in Chapter 6. Similar to previous chapters, this chapter highlights the topic from a software perspective without going into the hardware details. A general assumption in this chapter is that it is possible to determine an estimation for the worst-case execution time (WCET) of sequential code on any core with sufficient accuracy. For the shared-memory runtime environment (RTE), this assumption also applies to actor executions as they are always executed sequentially. On clustered architectures, however, the RTE can make use of parallelism within actor executions. A necessary assumption for this chapter is that a sufficient WCET estimation can be determined nonetheless. For element-wise actors this assumption is realistic since all cores in the executing tile act in a data-parallel fashion, with only one fork at the beginning and one join at the end of the actor execution. In between, each core exclusively processes a set of elements. Therefore, each core can be analyzed individually for the most part. Partition-based actors give the user greater control over the parallelism inside the actor and allow a much more complicated synchronization of cores. It is up to the user to ensure the analyzability of such actors.

Since graph executions with online scheduling target maximum flexibility and do not have any properties that would simplify the analysis, this chapter only covers

dataflow executions according to fixed schedules although an analysis of dataflow executions with online scheduling might also be possible.

The rest of this chapter is structured a follows. Section 8.1 focuses on standard graph executions without redundant actor executions. Aspects described in this section are essential for an analysis regardless of the hardware architecture. While the introduced formula is adequate for basic graph executions on shared-memory architectures, additional factors must be considered in the presence of a network-on-chip (NoC). Section 8.2 elaborates a formula that takes these aspects into account. Sections 8.3 and 8.4 focus on the analyzability of redundant dataflow executions and graph executions with possibly occurring faults. In contrast to previous sections, no concrete formulas are provided since the analysis is similar and the formal notation would be cumbersome. Instead, these two sections only describe the different aspects which must be considered in an analysis informally. Lastly, Section 8.5 gives a short conclusion on the analyzability of dataflow executions.

8.1 Basic Dataflow Executions

The most basic case is a graph execution without actor re-executions or redundancy changes. Graphs consist of one or more sections, which again consist of multiple actors. Following this hierarchy, WCET estimations of graph executions can be broken down to the WCET estimation of their sections, which can be again broken down to the analysis of individual actors. The general relation between the WCETs of graphs, sections and actors becomes clear with the small example graph in Figure 8.1. There are only two sections with only two data dependencies directed in the same way between them. But even if there were more sections and the dependencies between them were more complex, sections are always executed one after another in the order of their index, and the WCET estimation of a graph can thus be computed by summing up the WCET estimations of sections. For sections, however, estimating the WCET is more difficult. Actors are executed in parallel and so it is necessary to determine the longest path inside a section in terms of WCET

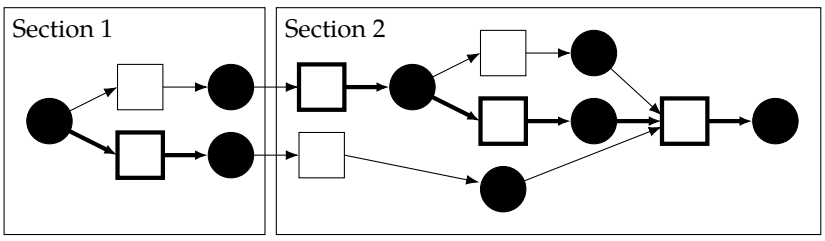


Figure 8.1: Example graph where the longest path is highlighted in each section

estimation. Such a longest path does not only need to consider data dependencies but also the execution order of actors given by the schedule. Furthermore, longest paths do need to connect to each other at section boundaries (such as in Figure 8.1). The next three sections describe the relation between the WCETs of graphs, sections and actors more thoroughly in a top-down fashion. It should be noted that in all following sections, the abbreviation *WCET* and similar terms like *finish time* always refer to proper estimations since the real WCET cannot be determined in almost all practical settings.

8.1.1 Worst-Case Execution Time Estimation of Graph Executions

For the estimation, an arbitrary graph g with n sections $s_1, \dots, s_n$ is considered, where the WCET for one execution of g is denoted with W^g and the WCET of s_i with W_i^s for $i = 1, \dots, n$. Since each section has its own schedule and is executed individually, the WCET estimation for one execution of the whole graph can be computed by adding up WCETs of all sections. Between two section executions, there is a portion of sequential code, for example to read error and performance counters, make a decision about the upcoming section and change the redundancy configuration accordingly. In the following, the WCET for the intermediate code between section s_{i-1} and s_i is denoted with W_i^r. Lastly, graph executions contain sequential initialization and termination routines with WCETs W^b and W^t, respectively. Altogether, this leads to the following formula:

$$W^g = W^b + W_1^s + \sum_{i=2}^{n} \left(W_i^r + W_i^s \right) + W^t$$

If a graph is not divided into sections, i.e. there is only one section, the large sum disappears so that only the WCET estimations for the initialization and termination routines and W_1^s remain.

8.1.2 Worst-Case Execution Time Estimation of Section Executions

The formula to compute the WCET of a graph depends on the WCETs of all sections in the graph. Each section in a graph consists of a set of actors. The execution time of a section is equivalent to the time from the beginning of the first actor execution to the end of the last actor execution. Thus, if the start time of the first actor execution is defined to be at $t = 0$, the section WCET is equivalent to the worst-case finish time of the last actor execution.

Let s be an arbitrary section with m actors $a_1, \ldots, a_m$ and F_i^a the worst-case finish time of actor a_i. With the considerations from above, a formula for the WCET of s is:

$$W^s = \max_{1 \leq i \leq m} F_i^a$$

Due to the data dependencies, the worst-case finish time of an actor depends on the worst-case finish time of its preceding actors. Therefore, the problem of computing the WCET of a section corresponds to the problem of finding a longest path in the graph. For directed acyclic graphs, this problem can be solved in linear time by finding a topological sort and successively annotating each node with the length of the longest path ending at that node. The topological ordering ensures that the longest paths of preceding nodes are known whenever a new node is processed. For computing the WCET of a section, the general procedure is the same. The only difference is that node annotations do not represent longest paths but correspond to worst-case finish times.

8.1.3 Basic Worst-Case Finish Time Estimation

To determine the worst-case finish time of an actor two parameters must be available, its worst-case start time and its WCET. As mentioned in previous sections, a general assumption in this chapter is that actor WCETs can be determined with appropriate tools and are thus known. The second important factor, the worst-case start time, depends on the worst-case finish time of the actor's predecessors since an actor can only be executed if the required data is present. An additional dependency may result from the schedule because an actor can only be executed when the execution of its predecessor in the schedule has been finished. In the example graph section depicted in Figure 8.2, there is such an additional dependency (visualized by the dashed arrow) between a_4 and a_3 caused by the schedule. With regard to the overall procedure described in the previous section, it is important to note that the topological sort which determines the order in which finish times are computed must also consider the additional dependencies resulting from the schedule.

To estimate the worst-case start time S^a of an actor a, it is necessary to determine the maximum of the worst-case finish time of all predecessors in the section (including the predecessor according to the schedule). If Pred^a is the set of predecessors of a, the worst-case start time can be estimated with the following formula:

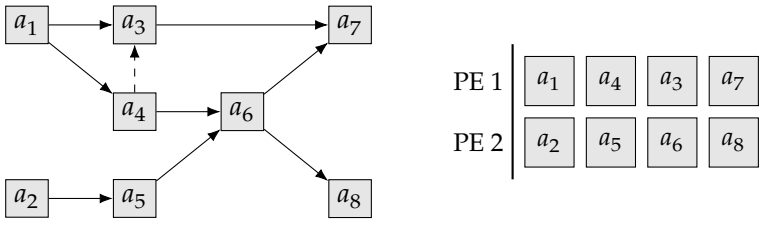
$$S^a = \max_{d \in \text{Pred}^a} F^d$$

To compute a's worst-case finish time F^a, the worst-case execution time W^a on the processing element (PE) specified by the schedule must be added to the worst-case

start time. It is important to consider optimization techniques, for example the copy avoidance technique described in Section 6.1.4, for the corresponding types of actors. If such an optimization is implemented, the WCET of some actors can be greatly reduced. Further, W^a should also consider any necessary preparations for the execution of actor a. Altogether, the resulting formula for F^a is:

$$F^a = S^a + W^a = \max_{d \in \text{Pred}^a} F^d + W^a$$

An example is shown in Figure 8.2. For a better overview, only the actor nodes are shown in the graphical representation of the section. Actors a_1 and a_2 do not depend on the data of other actors in this section and do not have a predecessor in the schedule. Therefore, both actors are executed immediately at the beginning of the section execution. Since the starting time is zero for these two actors, their worst-case finish time is equal to their respective WCET. Actor a_3 is the only actor where the predecessor according to the schedule (actor a_4) is not a predecessor according to the data dependencies. Because of this additional dependency, the worst-case finish time of a_3 is $F_3^a = F_4^a + W_3^a = 14$.



Actor	W	F
a_1	5	5
a_2	8	8
a_3	2	14
a_4	7	12
a_5	3	11
a_6	5	17
a_7	4	21
a_8	6	23

Figure 8.2: Example section (excluding data nodes) with schedule for two PEs, actor WECTs and worst-case finish times

8.2 Dataflow Executions on the Network-on-Chip-based Architecture

The estimation in the previous section may be sufficient for shared-memory architectures, but it is less suitable for NoC-based architectures due to additional factors that need to be considered. Since different NoC-based architectures usually show their own specific characteristics, an analysis is usually specific to a certain architecture. As mentioned in the beginning of this chapter, the abstract architecture specified

in Section 6.2.1 is the basis for the considerations in this section. In contrast to a shared-memory based system, three additional aspects must be considered. First, partition and actor transfers are an essential part of graph executions in the system. Second, local memories are small so that there are situations where partitions have to be displaced from tile-local memories. And lastly, actor executions can take place on compute tiles or the driver. Since these two types of tiles fulfill different roles in graph executions, it is clearer to use separate formulas for both cases.

Because the WCET estimation for graphs and their sections is the same for the two hardware architectures, only the worst-case finish times of actors is covered in the following sections. The basis for the considerations is an arbitrary actor a which is executed on a tile t_x. To cover the general case, a only depends on the results of other actors (and not on data already available at the beginning of the section). An additional assumption that a is not an initial actor in the schedule and its worst-case execution time is W^a. The following section describes some prerequisites regarding the considered hardware architecture before the actual execution of actors is discussed.

8.2.1 Additional Assumptions

To specify a formula for the estimation of actor worst-case finish times on the NoC-based architecture, some additional assumptions regarding the hardware architecture have to be made:

- There are p compute tiles and one driver tile available in the system. The driver tile is denoted with t_0 and the compute tiles with $t_1, \ldots, t_p$.

- Tile-local memories have a limited size, and thus the number of partitions which can be stored simultaneously is limited.

- The time it takes to transmit x bytes of data from tile t_i to t_j in the worst case is $D^x_{i,j}$. For the driver tile, data transfers always involve the off-chip memory. Further, the size of partitions sent over the NoC is bound in the RTE. Thus, the time to transfer such a partition can be denoted with $D_{i,j}$. In the following sections, $D_{i,j}$ is always used to estimate data transfers since it simplifies the formulas and each occurrence can be easily replaced by a suitable $D^x_{i,j}$ in a concrete analysis.

- Since control and status messages are small, one worst-case time is used in the following sections for all messages regardless of their size. The time it takes to transfer a message from t_i to t_j in the worst case is denoted with $M_{i,j}$.

- There is only one active data transfer per tile at a time. Additionally, a tile can initiate one message transfer at a time but receive arbitrary many messages concurrently. Furthermore, the NoC itself does not limit the overall number of concurrent transfers.

- An upper bound for the execution time of sequential code between two transfers or between a transfer and an actor execution on tile t_i is R_i.

For a better overview, Table 8.1 lists all variables introduced so far and describes their meaning. The table also contains all symbols which will become relevant in the following sections.

Table 8.1: Overview of all symbols used in this chapter

Symbol	Meaning
a, d	actors
g	graphs
s	sections
W^x	WCET of x, where x may refer to a graph, section, actor or sequential code outside of sections
F^a	worst-case finish time of actor a
S^a	worst-case start time of actor a
Pred^a	predecessor actors of a including the predecessor from the schedule (if existent)
t	tiles in the NoC-based system, t_0 is the driver
$D_{i,j}$	upper bound for the time it takes to transfer partition data from t_i to t_j
$M_{i,j}$	upper bound for the time it takes to transfer a small message from t_i to t_j
R_i	upper bound for sequential code between two transfers or a transfer and an actor execution on t_i
$P_{i,j}$	upper bound for the time it takes to transfer a partition (including metadata, actual partition data and acknowledgements) from t_i to t_j
Γ^a	upper bound for the time it takes to transfer all partitions required for the execution of actor a to the proper tile including partition displacements
Φ^a	upper bound for the potential delay of a partition transfer required for the execution of actor a due to actor executions on another compute tile
Ψ	upper bound for the potential delay of a partition transfer due to an actor execution on the driver tile
Ω^a	upper bound for time it takes to transfer actor a from the driver to the compute tile including all potential delays

8.2.2 NoC Transfers

The protocol for transferring partition data is always initiated by the driver tile since it is the only tile that is able to check the graph and schedule. Partition transfers take place in several ways. The first possibility is a partition transfer from the off-chip memory to the local memory of a tile t_x (with $x \neq 0$). In this case, the driver first starts a data transfer from the off-chip memory to the tile's local memory. After the data transfer is complete, the driver tile sends a notification message containing metadata, for example the partition size, to the compute tile. Therefore, an upper bound for the times this procedure takes in total is:

$$P_{0,x} = D_{0,x} + R_0 + M_{0,x}$$

Besides data transfers between the off-chip memory and local memory, there are also transfers between two local memories. A data transfer between two tile-local memories is initiated by the compute tile after receiving a message from the driver about the exact location of the required partition. Thus, three transfers are required in total, namely a message transfer from the driver to the compute tile, a data transfer between two local memories and the transfer of a short acknowledgement message back to the driver tile after the data transfer is complete. If t_y is the compute tile on which the desired partition is available, an estimation for the time is:

$$P_{y,x} = M_{0,x} + R_x + D_{y,x} + R_x + M_{x,0}$$

Lastly, there are data transfers from tile-local memories to the off-chip memory. Depending on whether the transferred partition is supposed to be removed from the local memory or it should remain there as a copy, the estimation is different. In the latter case the communication protocol is easier since there are no notification messages required and $D_{x,0}$ alone is suitable as an upper bound. The other case requires an additional message that tells the tile to delete the partition. Hence, in this case the estimation is:

$$P_{x,0} = D_{x,0} + R_0 + M_{0,x}$$

However, not all message transfers in graph executions are related to a partition transfer. This includes, for example, messages about finished actor executions or delete messages for partitions which are already present in the off-chip memory. Further, a message is required to transfer the relevant information about an actor from the driver to a compute tile. A suitable estimation for these transfers is either $M_{x,0}$ or $M_{0,x}$ depending on the direction.

8.2.3 Actor Execution on Compute Tiles

The general assumption for this section is that in order to execute a on t_x l partitions have to be removed from the tile-local memory and k partitions have to be transferred to the tile-local memory. Summing up the estimation for all transfers leads to total communication costs of:

$$\Gamma^a = \sum_{i \in T} \left(P_{i,x} + R_0\right) + l\left(P_{x,0} + R_0\right),$$

where T is a multiset containing k tile indices, one for each required partition in the memory of the respective tile. The left part is an estimation for partition transfers into the local memory of t_x, while the right part covers partition displacements.

It is difficult to determine when the required partition transfers may start exactly. A generally suitable estimation is the worst-case start time described in Section 8.1.3 since all required partitions have been computed at this time:

$$S^a = \max_{d \in \mathrm{Pred}^a} F^d$$

In practice, some partitions may be transferred earlier in case the data is available and a's predecessor according to the schedule has already been executed.

Tile t_x, however, is likely not the only tile with a pending transfer. Because of the assumption that the NoC interface does not support parallel data transfers, adding Γ^a to S^a is not enough. In the proposed RTE, the driver treats all actors equally regardless of the tile they are running on, and data transfers are initiated in the order in which they become pending. In the worst-case all compute tiles (except t_x) have a pending transfer between the local memory of t_x or the off-chip memory and their own local memory, so t_x may have to wait for the transfers of all other tiles to finish. Altogether, a general upper bound for this kind of delay is:

$$\Phi^a = \sum_{i=1, i \neq x}^{p} \left(\max\left(P_{x,i}, P_{0,i}\right) + R_0\right)$$

The value of the maximum in this formula depends on whether transfers between two tile-local memories or between a tile-local memory and the off-chip memory are slower on the hardware architecture. It should be noted that the two formulas above represent the absolute worst case. If the concrete graph and schedule are available, there is a chance that delays can be predicted more accurately, and thus the overestimation is less drastic. It is also important to note that such delays may occur for each transfer required for the execution of a.

Another factor which has to be considered are actors executed on the driver tile and the potentially associated transfers. Thus, each transfer required for the

execution of a could be delayed by either a transfer or an actual actor execution on the driver tile. If both cases should be considered, a possible upper bound for the delay is:

$$\Psi = \max\left(W^d, D_{y,0}\right) + R_0,$$

where d is the actor in the driver's schedule with the longest WCET and y is the index of the tile with the lowest data transfer rate.

Lastly, a message from the driver with details about a to initiate the actor execution and a notification message afterwards is required. Again, the first message may be delayed by transfers of other tiles. Only message transfers must be considered in this case because of the assumption that each tile is able to send a message even if a data transfer is ongoing. Therefore, one message transfer for each other tile has to be taken into account in the worst case. In combination with the actual execution of a, estimated by W^a, this leads to:

$$\Omega^a = \sum_{i=1, i\neq x}^{p} \left(M_{0,i} + R_0\right) + M_{0,x} + R_x + W^a + R_x + M_{x,0}.$$

Considering all the factors discussed above, a general upper bound for the finish time of actor a (which is executed on tile t_x) is:

$$F^a = S^a + \Gamma^a + \Omega^a + (k+l)\left(\Phi^a + \Psi\right)$$

8.2.4 Actor Execution on the Driver Tile

Actor executions on the driver tile can be estimated similarly. The biggest difference is of course that the driver tile actor executions cannot delay each other and thus only transfers between the off-chip memory and other tiles must be considered. Further, the driver tile has direct access to the off-chip memory so that no displacements are required and thus no displacement-related delays can occur. However, an actor execution itself can be delayed by data transfers with compute tiles. Hence, the resulting formula is

$$F^a = S^a + \Gamma^a + W^a + (k+1)\,\Phi^a,$$

with a slightly modified formula for Φ^a, where all compute tiles are considered, and a slightly modified Γ^a without displacements:

$$\Phi^a = \sum_{i=1}^{p} \left(P_{0,i} + R_0\right)$$

$$\Gamma^a = \sum_{i \in T} \left(P_{i,0} + R_0\right),$$

where T is a multiset consisting of one tile index for each required partition in the memory of the respective tile.

8.2.5 Worst-Case Finish Time Example

To illustrate the formulas from the previous sections, the example graph from Figure 8.2 is revisited, and the worst-case finish time of actor a_6 is determined. The formulas specified in the previous sections require some values to be known. Figure 8.3 provides an overview of all important values. For convenience, the figure also shows the dependencies between actors and the schedule which are the same as in Figure 8.2. The only difference is that Figure 8.3 explicitly shows that the driver tile t_0 does not execute actors from this graph section.

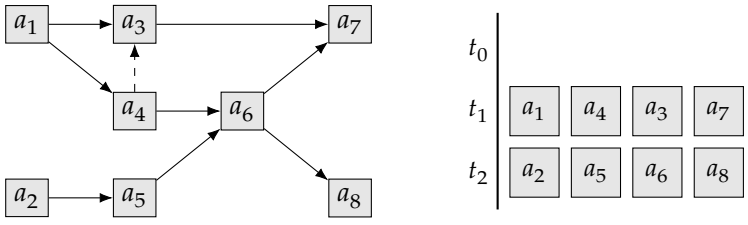
Actor a_6 is executed on tile t_2 and has two predecessors, a_4 and a_5. The latter is also executed on t_2, while the former is executed on t_1, requiring the result of a_4 to be transferred from t_1 to t_2. In this example, it assumed that there is enough memory available on t_2 for not only this partition but also the result of a_6 so that no partition has to be moved to the driver tile. The time to transfer one partition from t_1 to t_2 is:

$$P_{1,2} = M_{0,2} + R_2 + D_{1,2} + R_2 + M_{2,0} = 1 + 1 + 4 + 1 + 1 = 8$$

Since only one partition has to be transferred and no partitions have to be displaced ($l = 0$), the computation of Γ_6^a is quite short:

$$\Gamma_6^a = P_{1,2} + R_0 = 8 + 1 = 9$$

As described in previous sections, Γ_6^a represents an upper bound for the total transfer costs required for the execution of a. Due to the constraints given by the assumed architecture, transfers related to other actor executions might cause a delay. In this



Variable	Value
F_4^a	36
F_5^a	32
W_6^a	15
$D_{i,j}$	4
$M_{i,j}$	1
R_i	1

$i, j \in \{0, 1, 2\}$

Figure 8.3: Example section (excluding data nodes) with schedule and overview of values required to compute F_6^a

example there is one possible scenario that could cause such a delay. Tile t_1 might need to move a partition to the off-chip memory in order to make space for the result of actor a_3 because the memory may contain partitions from previous sections. For the purpose of a more detailed example, let this be the case. The partition transfer between t_0 and t_1 occupies the NoC interface of t_1 so that the partition transfer from t_1 to t_2 is delayed. Hence, the delay value for this actor is:

$$\Phi_6^a = P_{1,0} + R_0 = D_{1,0} + R_0 + M_{0,1} + R_0 = 4 + 1 + 1 + 1 = 7$$

The second delay value Ψ which is related to actor executions on the driver tile is irrelevant in this example since the driver's schedule is empty. Lastly, we need to determine an upper bound for the transfer of a_6 itself. In the assumed architecture a tile is able to start a message transfer, although a data transfer is ongoing. Thus, the transfer of a_6 from t_0 to t_2 may be delayed only by another message transfer, for example the transfer of actor a_3 from t_0 to t_1. Along with the actual actor execution and the required transfers, this results in:

$$\Omega_6^a = M_{0,1} + R_0 + M_{0,2} + R_2 + W_6^a + R_2 + M_{2,0} = 1 + 1 + 1 + 1 + 15 + 1 + 1 = 21$$

Summing up all values and adding the maximum worst-case finish times of the predecessors gives the desired upper bound for the worst-case finish time of a_6:

$$F_6^a = F_4^a + \Gamma_6^a + \Omega_6^a + \Phi_6^a = 36 + 9 + 21 + 7 = 73$$

8.3 Redundant Dataflow Execution

Since the proposed RTE supports different redundancy configurations, each section must be analyzed multiple times. For shared-memory architectures where no transfers have to be considered, estimations for all configurations can be determined as described in Section 8.1. The only aspect that has to be taken into account is whether the RTE is configured to operate in optimistic or pessimistic redundancy mode (see Section 6.2.8). In pessimistic mode, a PE has to wait until the result of its previously executed actor has been verified. This leads to an additional implicit dependency for some actors. If the RTE is configured to operate optimistically, there are no additional dependencies.

On the NoC-based architecture each redundancy configuration requires a different number of transfers. The next two sections therefore focus on redundant section executions on this architecture.

8.3.1 Redundant Actor Execution on the Same Tile

Estimations for redundant actor executions on the same tile are similar to the standard case as long as no re-execution is required. But since two or three unoccupied partition slots in the tile-local memory are necessary to store the results, there is a high chance that more data transfers take place due to additional displacements. A small difference is that the second and possibly third actor execution follow immediately after the first one with no intermittent transfers in between. This also applies to the comparison actor. Only if the comparison was successful, a message about the completed actor execution is sent to the driver tile. Thus, the combination of two or three redundant actors and their associated comparison actor could be considered as single piece of code with an appropriate WCET estimation in the analysis.

8.3.2 Redundant Actor Execution on Different Tiles

The second type of redundancy involves redundant actor executions on different tiles. As long as no errors occur, an upper bound for the worst-case finish time of redundant actors can be determined similarly to that of non-redundant actors. It is important to consider potential optimizations, in particular those regarding comparison actors. Such an approach was described in Section 6.2.8. With this optimization, comparison actors must be treated specially in an analysis because their execution corresponds to a single comparison of checksums on the driver. Compute tiles include the checksums in the payload of messages sent to the driver after a finished actor execution. As a result, the execution of comparison actors does not require any previous data transfers. Furthermore, since checksums are computed immediately after the result of an actor is available, checksum computations must be included in the WCET estimation of actor executions on compute tiles.

8.3.3 Actor Re-Executions

In case an error is detected, redundant actors have to be re-executed. An analysis has to distinguish between the different redundancy configurations. For redundant executions on the same PE, re-executions are transparent to other PEs and can be treated as delays in the estimation on an actor. This is also possible on the clustered architecture since a tile does not initiate a transfer as long as all results differ.

In case of redundancy over different PEs, the RTE supports two different execution modes. The pessimistic mode was designed with the intention to keep the overestimation for re-executions smaller. On shared-memory architectures, the re-execution can be treated as a delay on all PEs involved in the actor execution. On

the clustered architecture, additional message transfers are required when the driver is informed by another tile about a failed comparison or has executed the failed comparison itself. In this case, the driver needs to send messages to all involved tiles in order to initiate the deletion of the invalid data and the re-execution. Other transfers are not necessary since all partitions required for the actor executions are still available in the tile-local memories. In optimistic mode, however, PEs proceed with other (unrelated) actors even though their previous result was not yet verified. This makes an analysis more complex because re-executions may be delayed by an actor execution. On the clustered architecture, additional data transfers may be required because partitions may have already been displaced from the local memories when the comparison is complete. This might be the case for all tiles involved in the redundant actor execution. Furthermore, based on the schedule, it might be difficult to estimate how far a tile has proceeded in its schedule. In the worst-case it must be assumed that all required partitions have been replaced.

Regardless of the redundancy configuration, the WCET estimation of a section becomes more expensive if one or more arbitrary re-executions are considered. Based on the schedule it can be difficult to determine the actor whose re-execution prolongs the section execution the most. If a section contains x actors, x section WCETs must be computed in the worst case to find this actor. More re-executions of arbitrary actors may further increase the number of WCETs that have to be computed in order to find the combination of actors whose re-executions cause the highest delay. For y re-executions the number of possible combinations and therefore an upper bound for the number of WCETs is x^y. In practice, if the number of combinations is too large, a possible approach is to split a graph into smaller sections. This leads to more sections, but it reduces the number of combinations significantly.

8.3.4 Changing the Redundancy Configuration at Runtime

Adding redundancy changes to the analysis is rather straightforward. Redundancy changes can only occur between two graph sections, and so the corresponding WCET estimations can be determined regardless of section WCETs. Changing the redundancy of a single actor takes only a small amount of time so that a single WCET for the different types of actors and redundancy configurations is sufficient. Because redundancy changes are applied to each actor in a section sequentially, the WCET can be computed by multiplying the WCET of a single redundancy change with the number of actors in the section.

Redundancy changes during runtime have an impact on the execution of a dataflow graph because they add or remove redundant actors and comparison actors. This has to be considered in the WCET analysis. In the worst-case, the

redundancy configuration is changed for each section to the one with the largest WCET. Based on the requirements it might be beneficial to restrict the number of redundancy changes in the analysis to a realistic value in order to get a tighter upper bound for the WCET of a graph.

8.4 Permanent Faults

Chips contain different components which can potentially show a malfunction. The previous chapter described possible strategies to handle permanent faults in PEs, i.e. failing cores in case of shared-memory systems and compute tiles in case of a clustered architecture. Section 8.4.1 briefly discusses the influence of these procedures on the analyzability of graph executions. For the latter type of architecture, the failure of a single core does not lead to an entirely broken tile. Therefore, Section 8.4.2 provides some additional information about the influence of faulty cores on an analysis. Other components, however, are essential for a proper functioning of the tile. A defect in the tile's NoC interface or local memory, for instance, can make the entire tile unusable.

The consideration of malfunctioning components has a major influence on the analysis since it has a permanent impact on the execution of graphs. On heterogeneous architectures, the number of possible executions increases sharply when multiple permanent faults are taken into account. In contrast, on mostly homogeneous architectures like the two architectures described in this thesis where PEs are interchangeable, the number of combinations which have to be considered is more limited, and thus the analysis is less expensive.

8.4.1 Malfunctioning Processing Elements

Section 7.4 described different strategies to continue with dataflow executions in case a PE becomes unusable. On the NoC-based architecture, there is a chance that the local memory of a faulty tile is not accessible anymore and its content is lost. To simplify the analysis, it is beneficial to always restart the current section entirely whenever a PE becomes faulty, regardless of the hardware architecture. This requires to keep all partitions which are between two or more sections in the off-chip (or shared memory) memory as long as dependent sections have not been executed. Otherwise, it would be necessary to re-execute the entire graph in case partition data was lost.

To add the notion of faulty PEs to an analysis, two additional factors have to be considered, namely the reconfiguration and re-executing of the current section. Based on the strategy (see Section 7.4), subsequent sections might be executed

with altered schedules. This applies to all strategies except for spare PEs. In case of rescheduling at runtime and altering schedules the outcome of the respective strategy has to be determined in the analysis. Although in contrast to spare schedules the effort is significantly higher, both approaches are deterministic and an analysis is possible. If spare PEs or schedules are chosen, it is sufficient to add a minimal additional reconfiguration time to the WCET estimation.

On the clustered architecture, restarting a section additionally involves a memory reset on each compute tile. To perform a memory reset, a message from the driver is necessary. Thus, the reconfiguration WCET has to include an estimation for the respective broadcast. After the reset, the content of each tile-local memory from the first beginning of the section has to be restored. These additional transfers must also be added to the WCET estimation.

8.4.2 Malfunctioning Cores on the NoC-based Architecture

On the considered clustered architecture, each tile contains multiple homogeneous cores. Therefore, if one of the cores becomes faulty, the affected tile is able to execute actors with decreased performance. In this case, the current schedule can still be used, and a restart of the section is not required. The analysis only has to consider the potential restart of the currently executed actor.

Another aspect to take into account is that the impact of a reduced number of cores depends on the type of actor. Element-wise actors are executed on the tile in a data-parallel fashion, and elements are split as evenly as possible among the cores. For partition-wise actors it is up to the user to define the parallelism. In the best case, all other cores can take over some of the faulty core's work, and the performance drop is linear, while in the worst case the share of work cannot be divided, and one of the remaining cores has to handle it entirely.

8.5 Conclusion on the Analyzability of Graph Executions

As the previous sections described, it is possible to compute WCETs for standard graph executions, redundant graph executions and error cases. For the discussed cases, the RTE's design ensures that each functional PE is always able to make progress in a predictable amount of time. If the RTE is able to identify permanent faults, it is also possible to compute an upper bound in case a PE becomes faulty. The more faults and redundancy changes are considered, the more costly the analysis of a graph execution becomes. Thus, for large graphs it might be necessary to make

a compromise about which kinds of dynamic events are considered in the analysis and to which degree.

The analysis of an application executed on a NoC-based architecture is more complex because the different types of transfers must be considered. Since the displacement of partitions from tile-local memories is deterministic, it is possible to identify the memory content after each actor execution and therefore to determine the required transfers at design time. To simplify the analysis, previous sections made various assumptions regarding transfers. Real hardware architectures are often less restrictive and may allow, for example, multiple concurrent data transfers involving the off-chip memory. This can help to reduce the delays that have to be considered in the analysis so that the general overestimation is reduced.

8.6 Summary

In this chapter, the analyzability of dataflow executions was discussed. The WCET computation for a whole graph execution can be broken down to the WCET computation of graph sections and sequential code executed between sections. To determine the WCET of a section, the worst-case finish time of each actor in the section has to be computed. Two main factors must be taken into account here. First, since most actors depend on the results of other actors, the worst-case finish time of preceding actors has to be considered in the analysis. Additional dependencies result from the order of actors in the schedule. The second factor is the actual actor execution which is expressed in the analysis via the actor's WCET.

On the clustered architecture, the analysis must also consider data and message transfers. The number of transfers of each type that are required for an actor execution can be determined from the graph and schedule. In this regard, there may also be delays due to restrictions in the hardware, for example a limited amount of concurrent transfers.

Besides standard executions, it is also possible to analyze graph executions under events like redundancy changes and faults. There is no real upper bound regarding the number of events that could be considered in the analysis of an application. However, if the number of redundancy changes and faults is too large, the analysis may become unfeasible in practice.

The next chapter shows how the RTE's reference implementation performs on different hardware architectures with different redundancy configurations for the three example applications described in Chapter 4.

9

Evaluation

Chapter 4 described three exemplary RAPID programs which represent commonly used algorithms. In this chapter, the three applications are used to investigate the performance of the RTE's reference implementation. A characteristic of these applications is that they provide the necessary parallelism for distributed architectures and manycore processors. Thus, the applications themselves are not a limiting factor, and the influence of the runtime environment (RTE) on the execution time can be determined more precisely. As described in previous chapters, the RTE is able to execute graphs in many possible ways. The goal of this chapter is to cover all features of the RTE. However, the following sections will not provide results for all possible RTE configurations since the number of combinations is too large for the scope of this thesis.

Section 9.1 first covers the two hardware architectures used in the evaluation and provides information on their internal structure as well as how the RTE and benchmark applications were compiled. This is followed by additional details on the three benchmark applications. Dataflow execution times for a selection of non-redundant configurations are presented in Section 9.2. The results show the influence of different partitionings, varying numbers of graph sections and the supported scheduling techniques on the execution time. How the performance of graph executions under the different redundancy configurations compares to these results is the topic of Section 9.3. After that, Section 9.4 puts the execution time of

graph construction and scheduling on the two architectures into perspective, and Section 9.5 provides results of experiments related to the RTE's adaptive redundancy features.

9.1 Evaluation Hardware and Application Overview

The proposed RTE was evaluated on two different hardware architectures which match the abstract hardware architectures described in Chapter 6. More precisely, benchmark applications were run on a standard x86 multicore processor and the Kalray Bostan massively parallel processor array (MPPA). The benchmark applications that were used in the course of the evaluation are RAPID programs for Cannon's algorithm, the Cooley-Tukey algorithm and bitonic sort.

9.1.1 Shared-Memory Architecture

The shared-memory system used in the evaluation was a standard x86 personal computer equipped with an Intel Core i7-7700 CPU and 32 GB of memory. The operating system running on the machine was based on GNU/Linux with Kernel version 5.8. All benchmarks and the RTE were compiled with GCC 10.2.0 and with all optimizations available for the processor (compiler flags `-O3` and `-march=native`). The minimum required language standard to compile the RTE's reference implementation is C++14, but since the compiler on this architecture supports more recent standards, the RTE and applications were compiled according to the C++17 standard (`-std=c++17`).

The x86 RTE implementation features all optimizations described in previous chapters and is based on POSIX threads. In the initialization routine, the RTE starts a fixed number of threads which is configurable before compilation. For the rest of the RTE's runtime, threads are then maintained in a thread pool. Since the processor in the evaluation hardware consists of four cores with Hyper-Threading (Intel's simultaneous multithreading implementation), the RTE was compiled with eight threads for all experiments.

9.1.2 Clustered Architecture

To evaluate the execution of dataflow graphs on a clustered manycore architecture, a Kalray Bostan MPPA [Sai+15] was utilized. The internal structure of the MPPA is shown in Figure 9.1. This manycore processor consists of sixteen compute tiles and two I/O tiles. Each compute tile consists of sixteen compute cores and one resource management (RM) core which all share the same very long instruction

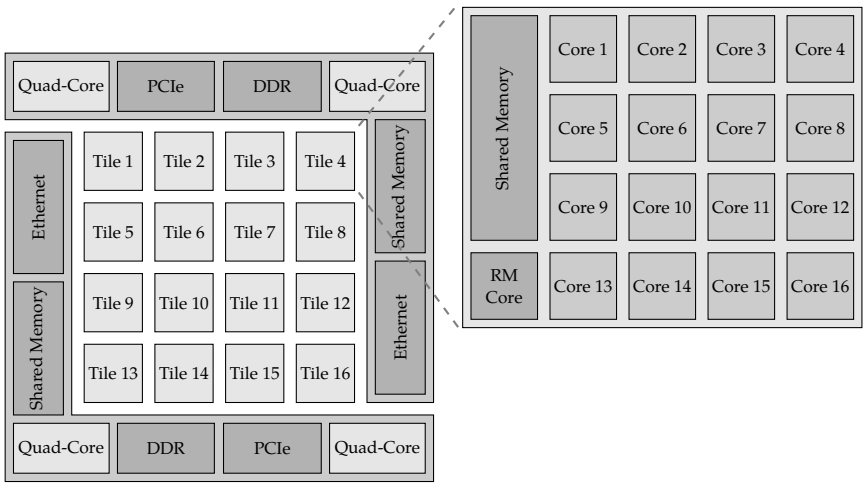


Figure 9.1: Kalray Bostan processor architecture (left part of the figure from [Sai+15], slightly modified)

word (VLIW) core architecture. Further, all cores on the tile are able to access a multi-banked local static memory (SMEM) with a capacity of 2 MB. Each of the I/O tiles consists of two quad-core CPUs, a shared SMEM, PCIe and Ethernet interfaces as well as a DDR3 memory controller. DDR memories in the MPPA used for this evaluation had a capacity of 2 GB. One quad-core in the I/O tile has direct access to the respective DDR memory. To run any kind of computation on a tile other than the driver, data has to be transferred from the DDR memory to the respective local memory over the network-on-chip (NoC). The topology of the NoC is a 2D torus consisting of 32 nodes. Each compute tile is connected to one node and each I/O tile to eight nodes. Four of these connections per I/O tile can be used to transfer data between the DDR memory and local memories.

It is often stated (for example in [Sai+15]) that the Kalray Bostan MPPA usually operates between 400 MHz and 800 MHz. However, the development board used in the evaluation did not allow a stable execution above 540 MHz, not even for small dataflow graphs. To ensure stability, all benchmarks were run at 500 MHz. A few benchmarks were additionally run at 400 MHz to observe the influence of different clock speeds and for the purpose of comparability with Kalray's OpenCL framework. For the evaluation, one I/O tile and all sixteen compute tiles were utilized. The I/O tile functioned as the driver and executed the main program. In contrast to the shared-memory implementation, the RTE for the Kalray MPPA runs bare metal without an operating system. Furthermore, all optimizations described in previous chapters were implemented. The benchmark programs and the RTE were compiled with a Kalray-specific GCC 4.9.4 which supports the required C++14 features to compile the RTE (compiler flag `-std=c++1y`). The optimization level was set to `-O3`.

9.1.3 Benchmark Applications

The benchmark applications that were used in the evaluation are implementations of Cannon's algorithm, a method for calculating the matrix multiplication, the Cooley-Tukey fast Fourier transform (FFT) algorithm and bitonic sort. Details on how the three algorithms can be expressed in the RAPID programming model were described in Section 4.6. Each benchmark was implemented to run on datasets of different sizes and with configurable partitionings. The three applications were specifically chosen for two reasons. On the one hand, all three benchmarks are based on algorithms which are commonly used in practice. On the other hand the applications differ in their computational complexity and focus on different types of instructions, namely integer arithmetic, floating-point arithmetic and load/store instructions.

Each configuration that was evaluated was run ten times, and input values were computed randomly for each benchmark execution. RAPID programs for Cannon's algorithm and bitonic sort are based on integers. The Coley-Tukey algorithm implementation, in contrast, uses complex numbers where the real and imaginary part are double-precision floating-point numbers. It should also be noted that there are some restrictions regarding the input datasets of the three benchmark applications. Cannon's algorithm is only compatible with square matrices, and the FFT and bitonic sort implementations require input vectors with a size equal to a power of two.

To measure the execution time, the `steady_clock` from the C++ chrono library was used. This clock is recommended for measuring time intervals since it is monotonic, i.e. its value cannot decrease as physical time moves forward, and the time between ticks is constant. The resolution of this clock depends on the hardware architecture and C++ library implementation. For the two hardware architecture used in the evaluation, the resolution is one nanosecond.

9.2 Non-Redundant Dataflow Execution

This section focuses on non-redundant graph executions and shows the influence of different RTE configurations and graph properties. Section 9.2.1 provides runtime measurements for dataflow executions of non-sectioned graphs on the two hardware architectures. The experimental results from this section represent the baseline to which the execution times of other configurations are compared. Therefore, this section covers all benchmarks, input sizes and hardware architectures. But since the number of possible combinations is large, most subsequent sections focus on a particular benchmark or input size.

9.2.1 Standard Graph Executions

In this section, execution times for the three benchmark applications and two hardware architectures are provided. All executions followed a fixed schedule which was computed by the RTE's implementation of the heterogeneous earliest finish time (HEFT) heuristic. It is important to note that all provided values only include the actual graph execution. Graph construction and scheduling are not part of the execution times. How long these two procedures take is the topic of Section 9.4.

To put the results in a general context, all benchmarks were also implemented in a commonly used computing framework which is available on the respective platform. On the x86 architecture, the three dataflow applications are compared to OpenMP versions of the algorithms. These reference benchmarks were implemented as the standard iterative versions of matrix multiplication, FFT and bitonic sort, with the outermost suitable loop being executed in parallel. In order to obtain reference execution times on the Kalray MPPA, Kalray's OpenCL framework was used. For a fair comparison, the data was arranged in a way that is advantageous for Kalray's OpenCL paging mechanism. In case of the OpenCL matrix multiplication benchmark, the matrices were enlarged slightly (e.g. from 2000×2000 to 2048×2048), and the second matrix was transposed so that memory pages contain full rows or columns. The OpenCL implementation of this benchmark is actually not completely fair since the preparation of input matrices is done by the host system while the dataflow matrix multiplication benchmark prepares the input matrices itself. In the experiments, the paging mechanism of Kalray's OpenCL implementation did not handle the transposition of the second matrix very well, and thus including it in the benchmark would make the comparison less meaningful.

Execution times of the benchmarks on the x86 platform are shown in Figure 9.2. To eliminate rare cases of excessively large or small execution times, benchmarks were run ten times for each input size and the graphs show the respective average values. The figure shows that the execution of RAPID dataflow graphs is faster than the execution of the respective OpenMP implementation for two of the three benchmarks. In this regard it is important to note that the number of partitions into which the input data is divided matters, and for each execution an optimal partitioning with regard to the size of the input data was chosen. An advantage of small partitions is that they fit in the data cache. However, dividing data into many small partitions leads to large dataflow graphs consisting of short actors. The drawback of such short actors is that the overhead introduced by the RTE is more significant. Section 9.2.2 provides more information on this topic.

Bitonic sort is the only benchmark where the execution of the dataflow graph takes longer than the execution of the OpenMP program. A major reason for this lies in the design of the programming and execution model. To re-execute a redundant

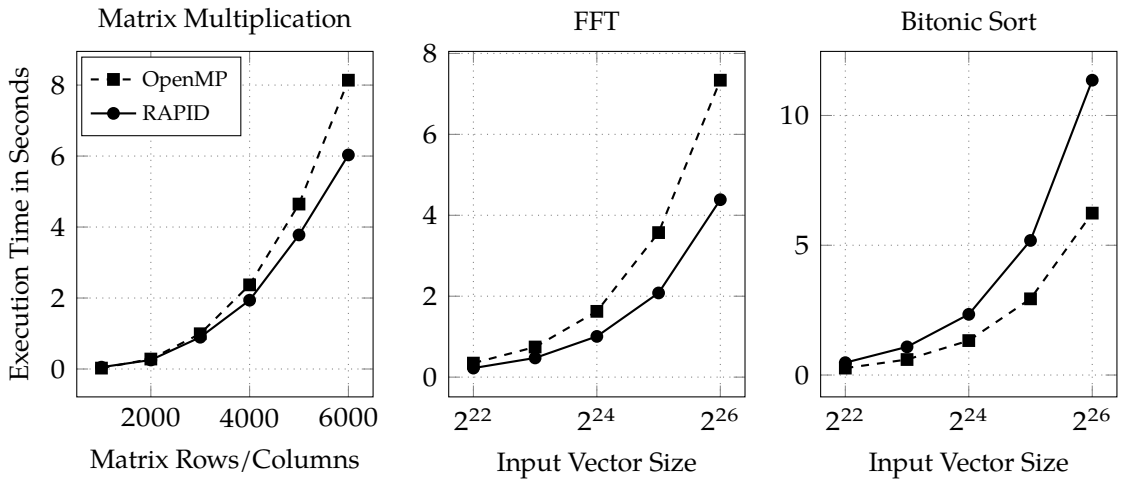


Figure 9.2: Execution times of benchmarks on x86

actor in case of an error, the actor's inputs must still be available. As a consequence, actors are not able to work in-place on their input data. With regard to bitonic sorting, this means that an actor always has to copy all elements from the input partition to the output partition. The OpenMP program, on the other hand, has to write less data since two elements can often stay in place after they were compared.

Experimental results for the Kalray MPPA are shown in Figure 9.3. Similar to x86, to achieve the best performance, partition sizes were chosen as close to the size of partition memory slots as possible. For this configuration, all benchmark application were executed with 400 MHz and 500 MHz since the Kalray OpenCL documentation does not specify at which frequency programs are executed (only that the maximum OpenCL frequency is 600 MHz) or how to configure the hardware to execute an OpenCL program with the desired clock frequency. It is most likely that OpenCl programs are executed at the default frequency, i.e. 400 MHz.

The OpenCL benchmarks outperforms the dataflow RTE consistently in the FFT benchmark, for small matrices in the matrix multiplication benchmark and for large input vectors in the bitonic sort benchmarks. There are multiple reasons for these performance differences, mostly related to the RTE's redundancy mechanism. One reason is that data is immutable in the dataflow execution model and actors only create new data. This is an essential aspect of the redundancy mechanism and allows the system to re-execute actors in case of a faulty execution. The OpenCL FFT and bitonic sort benchmark applications, however, work in-place. Bitonic sort, for example, benefits from an in-place execution as it compares pairs of elements and only needs to swap them when they are in the wrong order. A second reason is that, in favor of easier rollbacks, dataflow actors are limited one output. This reduces the

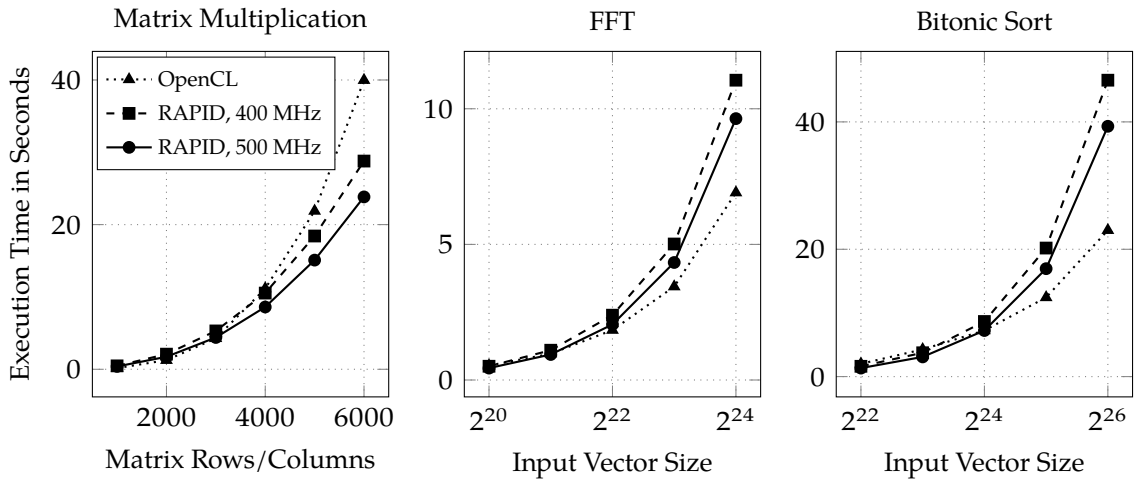


Figure 9.3: Execution times of benchmarks on Kalray

performance of the dataflow FFT benchmark since in favor of a smaller dataflow graph and easier scheduling, some computation is repeated. Another factor lies in the use of fixed schedules. The RTE's HEFT implementation estimates the execution time of actors and data transfer times according to user-defined measurement functions and the amount of processed or transferred data. Especially transfer times are difficult to predict accurately since the exact timing of transfers at runtime can vary, so it is difficult to determine offline which transfers will occur in parallel. Section 9.2.4 shows that using work stealing instead of fixed schedules can lead to increased performance in some cases.

Compared to the x86 architecture, execution times on the Kalray MPA are consistently higher. However, these results must be put into perspective since the two architectures are typically used in different scenarios. Based on the clock frequency, the Kalray MPPA has a typical power consumption of 10-20W [Sai+15] and targets embedded applications, while the Intel CPU was primarily designed for use in desktop computers. The algorithm which performs the worst on the MPPA compared to the x86 architecture is FFT. In this application, the execution time is roughly ten times as high as on the x86 processor. Without detailed information about processor internals it is difficult to determine the exact reason for this. Since the implementations of Cannon's algorithm and bitonic sort used in the evaluation are based on integer arithmetic, it is possible that the high amount of floating-point computations in the FFT benchmark is a significant factor. Further, the experiments of this and following sections suggest that non-sequential direct accesses to the DDR memory are quite slow on the Kalray architecture. However, such accesses are required to reorder the data elements before the actual FFT computation can take place. While

the matrix multiplication also requires a reordering of the data elements, the impact is significantly lower due to the higher computational complexity of the algorithm.

In contrast to the respective graph in Figure 9.2, the FFT graph for the Kalray MPPA only provides execution times for input vectors with up to 2^{24} elements. The reason for this is that the RTE utilizes only one of the two I/O tiles and the 2 GB of DDR memory are not sufficient for graph executions with larger input vectors.

When comparing the graphs for the two different frequencies, it is noticeable that increasing the frequency by 25% leads to a reduction in the execution time by roughly 17% for Cannon's algorithm, 13% for FFT and 15% for bitonic sort. Cannon's algorithm benefits the most from an increased frequency since it is the most expensive algorithm out of the three with a complexity in $O(n^3)$. This higher overall complexity directly affects the complexity of individual actors instead of leading to overly large graphs and the associated runtime overhead. Thus, in comparison to the other algorithms the influence of actual computation on the overall execution time is higher.

9.2.2 Impact of Different Partitionings

The previous section already showed that the number of partitions the input data is split into has an influence on the execution time of applications. This section focuses on the influence of different partitionings on the execution time for the Cooley-Tukey FFT algorithm. The input vectors that were used for the benchmark execution on x86 and Kalray consisted of 2^{25} and 2^{23} elements, respectively. The measured execution times are shown in Figure 9.4. As before, this section only considers graph execution, and all results represent average execution times.

For the x86 shared-memory architecture, the optimal number of partitions is 512. Since each complex number in the input vector occupies 16 bytes of memory (two 64 bit floating-point numbers), this division leads to partitions with a size of 1 MB. Most actors in the graph are `zipmap_partitions` actors which read two partitions of this size, and there are always two such actors which process the same data and can be executed concurrently by two different cores. In the best case, the four physical cores of the Intel Core i7-7700 process the data of four partitions, i.e. 4 MB of data, at a time. Adding the data created by the four concurrently executed actors leads to 8 MB in total, which is also the size of the shared L3 cache. Therefore, this partitioning allows the system to efficiently use the processor cache. For fewer but larger partitions, the cache cannot be utilized in such an efficient way, and thus the execution times are much higher. Smaller partitions on the other hand also allow an efficient use of the cache. However, the more partitions the input vector is split into, the more actors are required to process these partitions, and the overhead for navigating through the graph and for the synchronization of cores increases.

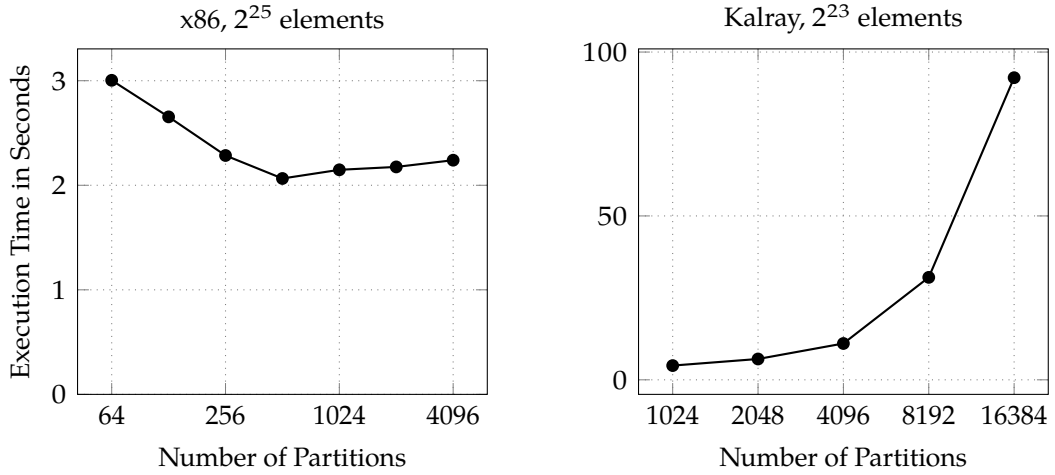


Figure 9.4: Execution times of FFT for different partitionings

Compared to the optimal partitioning, this leads to longer execution times, but the effect is much less significant than in case of larger partitions.

Figure 9.4 also shows execution times for the Kalray MPPA. In comparison with x86, the graph for the Kalray MPPA has a different shape. This is a consequence of the limited memory on the compute tiles. As mentioned in previous chapters, local memories are divided into fixed slots. For this evaluation, the RTE was configured so that each local memory contains eight slots with a size of 160 KB each. Therefore, splitting the input vector into less than 1024 partitions is not possible since the partitions would be too large. As the graph shows, 1024 is the optimal number of partitions. In this case, each partition is 128 KB in size. If the input vector is split into smaller partitions, the graph becomes larger, and more data and message transfers are required. Further, slots are filled to a lesser degree, and memory in the tile-local memories is wasted. As a result, overly small partitions affect the execution time much more than in the shared memory system.

9.2.3 Impact of the Number of Graph Sections

Similar to the partitioning, dividing a graph into multiple sections has an influence on the execution time. This section provides experimental results for Cannon's algorithm. Two 6000×6000 matrices were used as input data on the x86 architecture and two 4000×4000 matrices on the Kalray MPPA. On the x86 architecture, input matrices were split into 900 and on the Kalray architecture into 400 square blocks. This leads to graphs with 28803 and 8803 actor nodes in total, respectively (excluding all the redundant actor nodes created by the RTE). From the 28803 (8803) actors, two are responsible for rearranging the memory of the input matrices, 1800 (800) for

splitting them into blocks and one for reordering the elements of the output matrix. The remaining 27000 (8000) actors are `zipmap_partitions` actors and perform the actual computation. These actors were divided evenly into sections.

As mentioned in Section 4.4.2, graph sections can only be specified via the context's `checkpoint` function in the RAPID programming model. This way of creating sections ensures that sectioned graphs are always executable, but the number of possible sectionings is limited. For the graph considered in this section, the 27000 (8000) `zipmap_partitions` actors are arranged in 30 (20) layers consisting of 900 (400) actors each. There are no data dependencies between the actors of a layer. Since each layer is constructed by a `zipmap_partitions` operation, section boundaries can only be between two layers. With this in mind, the graph was divided into 2, 5 and 10 sections consisting of 15 (10), 6 (4) and 3 (2) layers each, respectively.

The results are shown in Figure 9.5. As a reference, execution times for the graph without a sectioning (which is equal to a graph with one section) are also included in the figure. For this benchmark, dividing the graph into sections does not lead to significantly higher execution times. On the x86 architecture, there is no performance drop at all. There are two main reason why the execution times are almost the same. First, the schedules do not have large gaps since actors are homogeneous and the work can be divided evenly across all processing elements. Second, even though sections could be scheduled in parallel, the RTE was configured so that sections are scheduled sequentially. This allows the HEFT scheduling heuristic to make use of the mapping of previous sections and therefore produce a schedule which attaches to the previous schedules quite seamlessly.

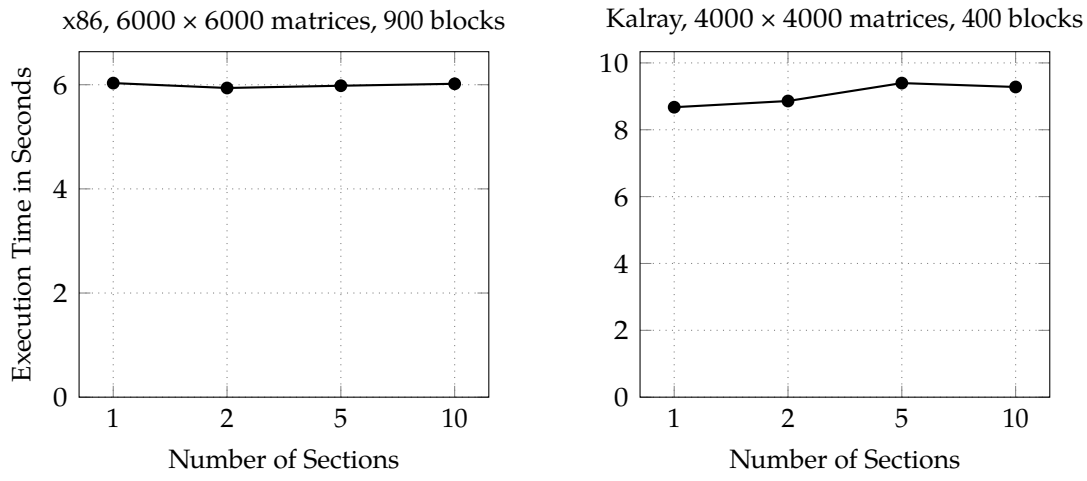


Figure 9.5: Execution times of the matrix multiplication benchmark for different numbers of sections

9.2.4 Online Scheduling

In addition to the use of fixed schedules, the RTE also supports graph executions based on work stealing. To compare the performance of such graph executions with graph executions according to fixed schedules, all three benchmark applications were additionally executed in online scheduling mode. As before, only the graph execution times were measured. Figure 9.6 shows the results for the x86 architecture. In two cases the work stealing approach slightly outperforms the statically computed

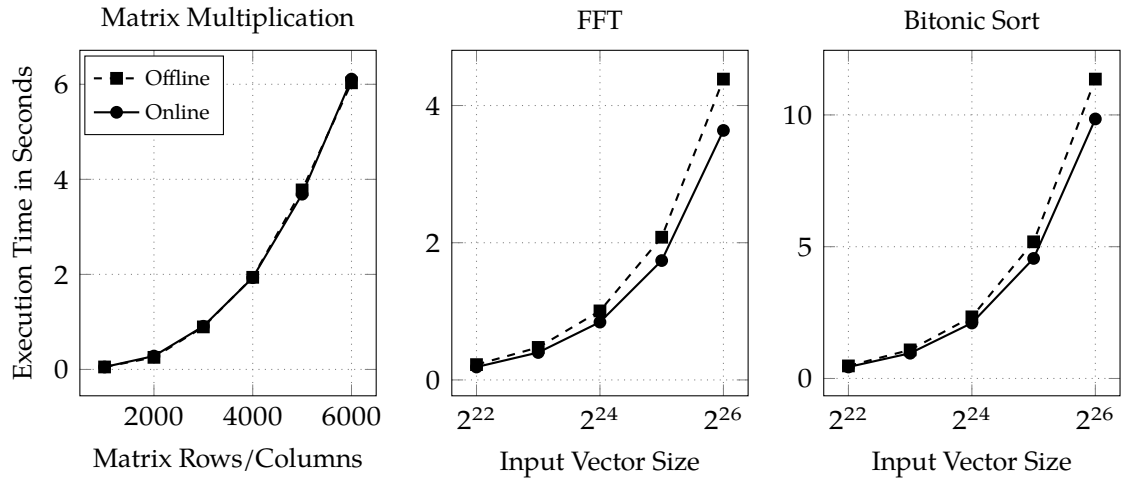


Figure 9.6: Dataflow execution times with fixed schedules (offline) and work stealing (online) on x86

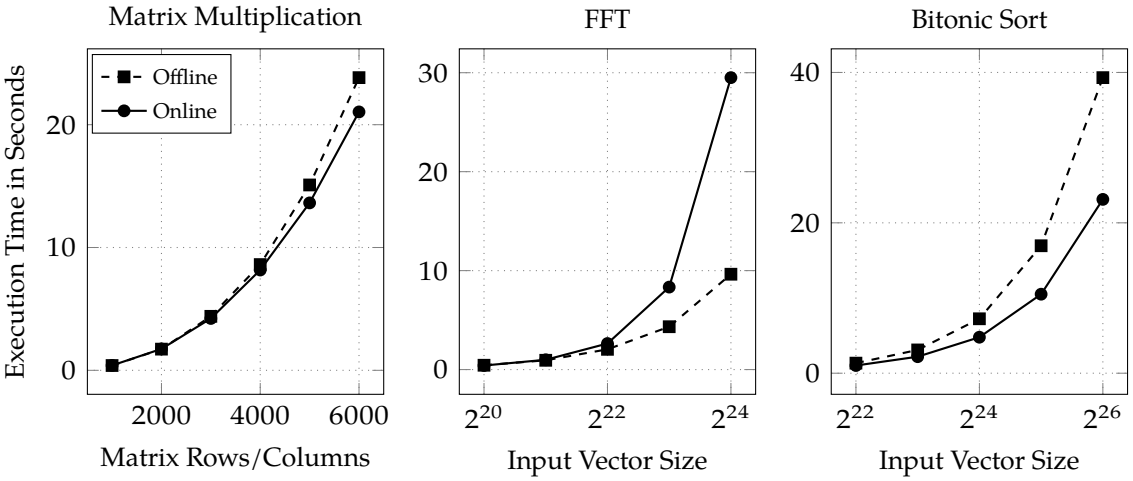


Figure 9.7: Dataflow execution times with fixed schedules (offline) and work stealing (online) on Kalray

schedule since cores are not bound to a fixed actor mapping and order, so the cores can be utilized more efficiently. For Cannon's algorithm, both approaches lead to nearly identical execution times.

Results on the Kalray MPPA are different to those on x86 as Figure 9.7 shows. For Cannon's algorithm and bitonic sort, the execution times are lower when the RTE operates in online scheduling mode. The Cooley-Tukey FFT algorithm, however, shows that there are cases where a statically computed schedule leads to a better performance. Work stealing does not consider data dependencies and transfers, which have a big impact on the graph execution time on NoC-based architectures. Further, the number of stolen actors can also have an influence on the performance because it is likely that additional transfers are required when a random actor is stolen from another PE. Table 9.1 shows how many actors are stolen on average for the three benchmark applications with various input sizes. Because these numbers alone are not very meaningful, the table also puts them in relation with the respective total number of actors in the graph (excluding the redundant actors created by the RTE). It is noticeable that the percentage of stolen actors decreases for larger graphs. In the matrix multiplication benchmark the relative number of stolen actors is the lowest, with only 0.3% of all actors being stolen for the largest graph, while in the FFT benchmark it is the highest, with 5.2% for the largest graph.

Table 9.1: Number of stolen actors on the Kalray MPPA

	Input Size	Partitions	Actors in Total	Average Stolen	Percent
MatMul	1000^2	25	178	10.4	5.8%
	2000^2	100	1203	31.9	2.7%
	3000^2	225	3828	48.8	1.3%
	4000^2	400	8803	65.5	0.7%
	5000^2	625	16878	82.7	0.5%
	6000^2	900	28803	91.6	0.3%
FFT	2^{20}	128	1154	331.8	28.8%
	2^{21}	256	2562	459.0	17.9%
	2^{22}	512	5634	674.2	12.0%
	2^{23}	1024	12290	1050.1	8.5%
	2^{24}	2048	26626	1382.5	5.2%
Bitonic	2^{22}	128	4737	1339.5	28.3%
	2^{23}	256	11777	1812.3	15.4%
	2^{24}	512	28673	2317.6	8.1%
	2^{25}	1024	68609	2987.7	4.4%
	2^{26}	2048	161793	4166.8	2.6%

9.3 Redundant Dataflow Execution

After the previous sections provided information about standard dataflow executions, the following sections focus on redundancy. As before, the results from Section 9.2.1 act as a baseline. For better comparability, all redundant graph executions used the same partitionings as in Section 9.2.1 and followed a fixed schedule which was computed by the RTE's implementation of the HEFT scheduling heuristic.

The proposed RTE executes only those actors redundantly that perform an actual computation. This means that all actors which only copy data from one memory location to another, i.e. `interval`, `collect` and `reorder` actors, were only executed once in the experiments. Since `interval` and `collect` actors in the three benchmark applications are affected by the optimization strategies described in Section 6.1.4 and are thus, regardless of the redundancy configuration, in most cases not executed at all, only `reorder` actors are executed non-redundantly. Depending on the amount of reordering an application requires and how fast the hardware is able to reorder data elements in memory, it is possible that executing a graph in a redundant configuration requires less than twice (or three times) the baseline execution time.

9.3.1 Redundant Actor Execution on Different Processing Elements

This section covers graph executions with redundant actor executions on different processing elements. As described in previous chapters, the RTE supports two modes for such types of redundancy configurations. In this section, the results for pessimistic actor executions are shown. Thus, the extended HEFT scheduling heuristic had to be applied without the insertion-based policy (see Section 7.2.2).

Results for the x86 architecture are shown in Figure 9.8. As described in the previous section, due to the fact that the expensive `reorder` actors are not executed redundantly, the times for redundant execution are not quite twice or three times as long in case of FFT. This can also be observed for Cannon's algorithm with small matrices. For larger matrices, however, the normalized execution times are higher. The reason for this is that the computing actors perform a standard matrix multiplication which is quite time-consuming for large matrices due to the cubic runtime. Therefore, the reordering is less significant, and the normalized execution time is higher. Bitonic sort graphs do not contain any `reorder` actors, and thus it is not surprising that executing the graph redundantly takes more than twice (or three times for triple execution) as long as the baseline execution time.

On the Kalray architecture (Figure 9.9), execution times are different because there are additional influencing factors. One important aspect are data and message

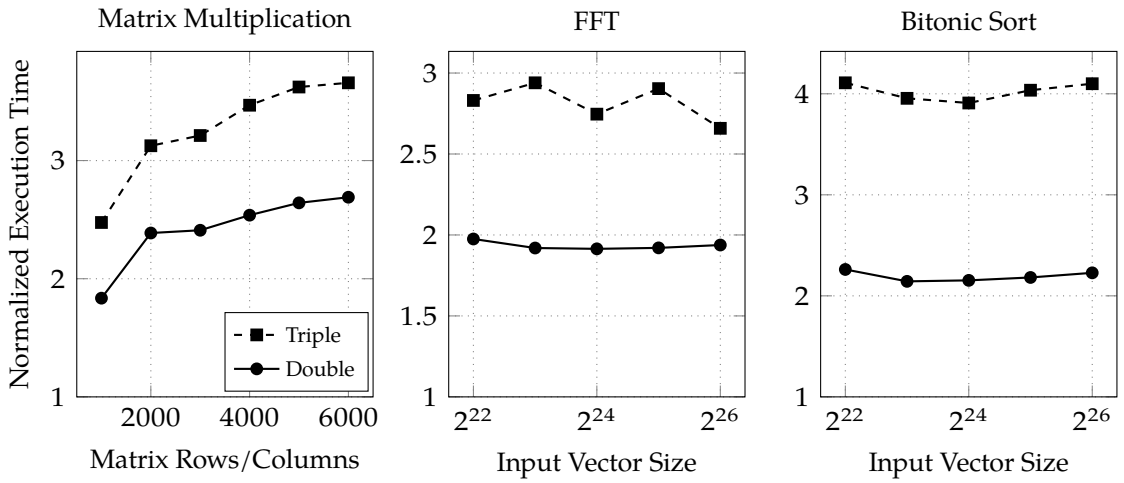


Figure 9.8: Dataflow execution times with redundancy over different PEs on x86 normalized to non-redundant execution times

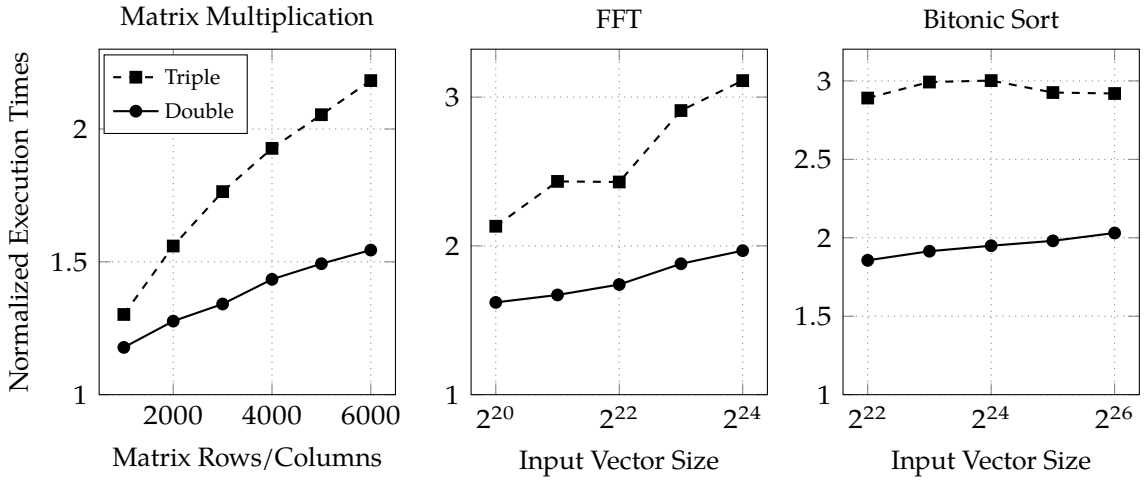


Figure 9.9: Dataflow execution times with redundancy over different PEs on Kalray normalized to non-redundant execution times

transfers between the tiles. Redundant actor executions on different tiles clearly require more communication than non-redundant executions since each actor must be transferred from the driver to multiple tiles, and consequently partitions must be present in additional local memories. Depending on how well transfers and actor executions can occur in parallel, it is possible that redundant graph executions take not quite twice (or three times) as long as the baseline even though there are no `reorder` actors in the graph. The second difference between the x86 and Kalray architecture is the I/O subsystem. Only cores in the I/O subsystem have direct

access to the DDR memory which contains the dataflow graph and schedule. These cores are also responsible for the execution of `reorder` actors in the benchmarks because the involved partitions are too large to fit into a local memory. Therefore, the fact that the I/O subsystem executes only non-redundant code in the benchmarks is another reason for relatively short redundant execution times. The results for the matrix multiplication (see Figure 9.9) are an example. As mentioned in Section 9.2.1, accessing the DDR memory in a non-sequential fashion from the I/O cores is quite slow, and thus the I/O subsystem is a potential bottleneck. However, there seems to be no other way to reorder large amounts of data or navigate through a large graph and schedule on this platform.

9.3.2 Redundant Actor Execution on the Same Processing Element

Redundant actor execution on the same processing element is especially beneficial on the Kalray platform because the number of transfers is lower compared to redundancy across different tiles. However, in comparison to the non-redundant execution, the number of transfers is possibly higher. The reason for this is that a redundant actor execution on a tile requires multiple memory slots for the results and hence more partition displacements are required. However, since PEs do not have to wait for comparisons, there is also a performance benefit on x86.

Figure 9.10 shows results for the x86 architecture. Execution times are lower compared to redundancy over different PEs, but otherwise the shape of the graphs

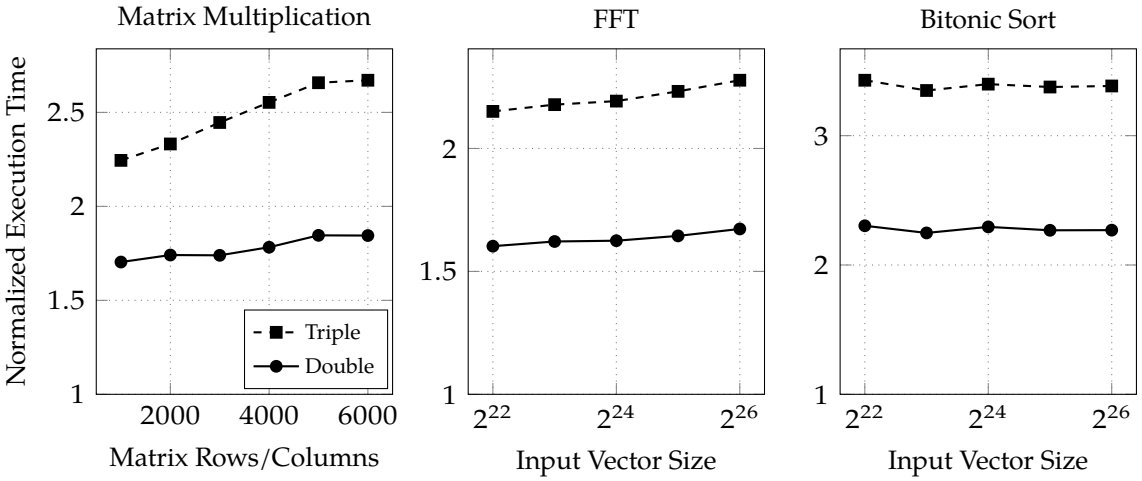


Figure 9.10: Dataflow execution times with local redundancy on x86 normalized to non-redundant execution times

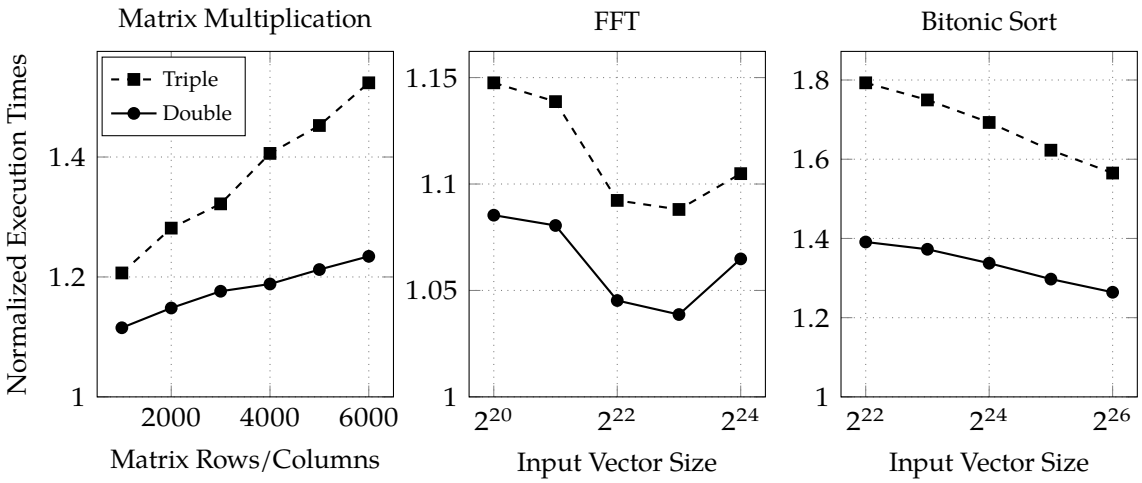


Figure 9.11: Dataflow execution times with local redundancy on Kalray normalized to non-redundant execution times

is similar to those in Figure 9.8. As before, bitonic sort graphs do not contain `reorder` actors, and thus the execution times are higher than for the other benchmarks.

The results for the Kalray architecture shown in Figure 9.11 reveal that there is a significant overhead in the dataflow executions since the redundant execution times are very low compared to standard graph executions. FFT is an extreme case where redundant execution increases the execution time only by 15% at most. As before, the `reorder` actor is a significant factor for this. Furthermore, since almost all transfers involve the I/O subsystem, it is possible that the preparation of transfers which requires navigation through the graph and schedule on the I/O cores is a bottleneck. Another possibility is that the I/O subsystem's NoC interface or the connection to the DDR memory limits the performance. However, it seems that transfers and the rearrangement of data elements do not only have a large influence on the execution times in the proposed RTE but also in OpenCL because otherwise the performance difference between the two frameworks would be higher.

9.3.3 Optimistic and Pessimistic Redundancy Modes

With regard to redundant actor executions on different PEs, the RTE provides two different modes. Section 9.3.1 only showed results for the pessimistic mode. The considerations in Chapter 8 also focused on the pessimistic execution mode because optimistic execution increases the complexity in the analysis of error cases. However, executing graphs in optimistic mode leads to a higher performance. This section exemplarily shows the performance benefit for the x86 architecture. Fig-

ure 9.12 shows the difference of both modes with regard to execution time. In this figure, the dataflow execution times in optimistic mode are normalized to the corresponding execution times in pessimistic mode. Since all values are smaller than 1, optimistic mode consistently increases the performance of dataflow executions for all benchmarks and redundancy configurations. It is also noticeable, that the performance benefit heavily depends on the specific graph and schedule. The biggest performance increases were measured for the FFT benchmark with three redundant executions per actor. For one FFT graph (the one with 2^{23} input elements), the average execution time with optimistic redundancy was only 77% of the average execution time with pessimistic redundancy.

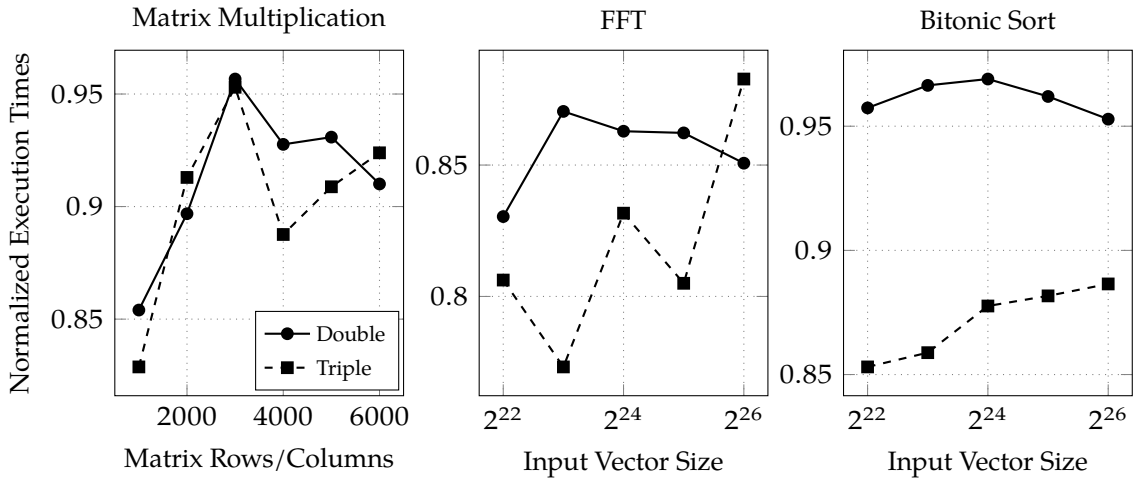


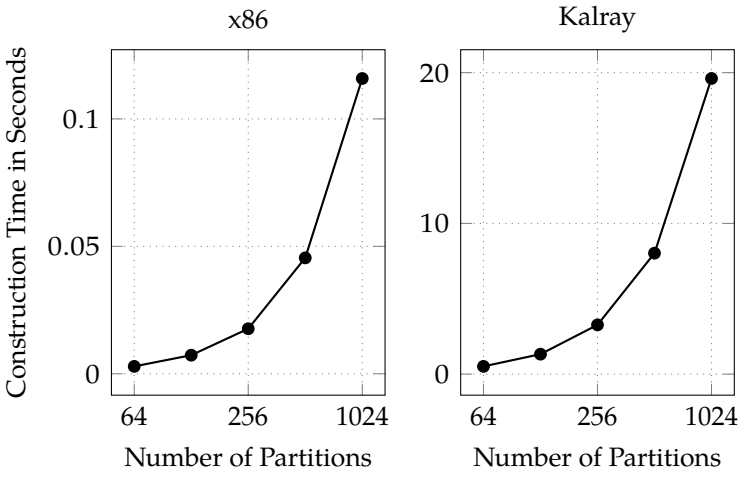
Figure 9.12: Dataflow execution times with redundancy over two or three PEs on x86 in optimistic mode normalized to the corresponding execution times with pessimistic redundancy

9.4 Graph Construction and Scheduling

Previous sections provided execution times for the three benchmarks executed in various RTE configurations. Since the execution time of graph construction and scheduling are probably less important for a productive use of the RTE than the execution time for actual graph executions, the focus in the RTE implementation was primarily on the optimization of graph executions. Nonetheless, the following two sections show how the RTE performs for these tasks. As before, there will not be experimental results for all possible combinations of benchmarks and configurations but rather for the most interesting aspects.

9.4.1 Graph Construction

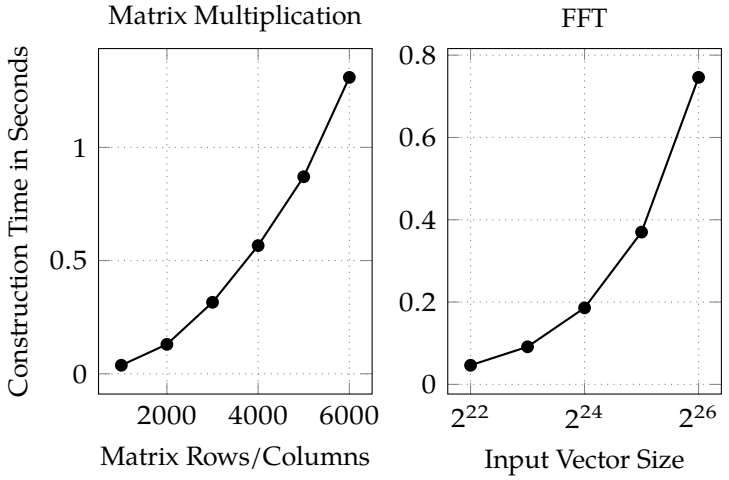
How long the construction of a graph from a RAPID program takes depends on two aspects. The first and most obvious factor is the number of nodes in the resulting graph. To investigate the impact of the number of nodes, the RAPID implementation of bitonic sort was run with different partionings. For this experiment, the number of elements in each partition is irrelevant because only the number of partitions has an influence on the size of the graph. Results for different partitionings on x86 and Kalray are shown in Figure 9.13. The figure also contains a table which lists the corresponding number of constructed nodes for each partitioning including all redundant actor nodes, comparison actor nodes and data nodes. Therefore, these numbers are between seven and eight times as high (three redundant actors and one comparison actor for all actor types except `interval`, `collect` and `reorder` plus the required data nodes) as the numbers in Table 9.1. The diagrams show that construction time scales with the number of actors in the graph. Graph construction on the Kalray platform is significantly slower than on the x86 platform. The difference between the two architectures is much larger than for dataflow executions. This is not surprising since RAPID programs are sequential C++ code, and thus graphs are built mostly sequentially. Furthermore, graphs are always constructed in the DDR memory because they are usually quite large. Altogether, the longer graph construction times on the Kalray platform are a result of the low single-core performance and the slow non-sequential access to the DDR memory on this architecture.



Partitions	Nodes in Total
64	14467
128	37123
256	92675
512	226307
1024	542723

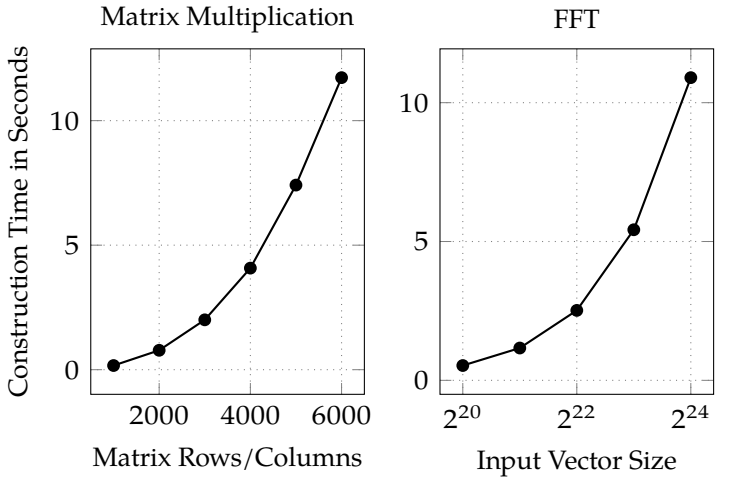
Figure 9.13: Graph construction time and number of constructed nodes for bitonic sort with different partitionings on x86 and Kalray

The second factor is what type of RAPID operations are used in the RAPID program. Some types of operations may require complex computation during graph construction. In the current state of the programming model only `reorder` falls into this category. However, extensions of the programming models with similar operations are conceivable. The `reorder` operation computes new element indices at graph construction. As a consequence, graph construction times also depend on the number of elements whenever at least one such operation is present. Cannon's algorithm and FFT are examples for this. In Figures 9.14 and 9.15, measurements for



Input Size	Nodes in Total
1000^2	1111
2000^2	8411
3000^2	8411
4000^2	65611
5000^2	65611
6000^2	219611
2^{22}	3724
2^{23}	8461
2^{24}	18958
2^{25}	41999
2^{26}	92176

Figure 9.14: Graph construction times and number of constructed nodes for matrix multiplication and FFT on the x86 architecture



Input Size	Nodes in Total
1000^2	1111
2000^2	8411
3000^2	27911
4000^2	65611
5000^2	127511
6000^2	219611
2^{20}	8461
2^{21}	18958
2^{22}	41999
2^{23}	92176
2^{24}	200721

Figure 9.15: Graph construction times and number of constructed nodes for matrix multiplication and FFT on the Kalray architecture

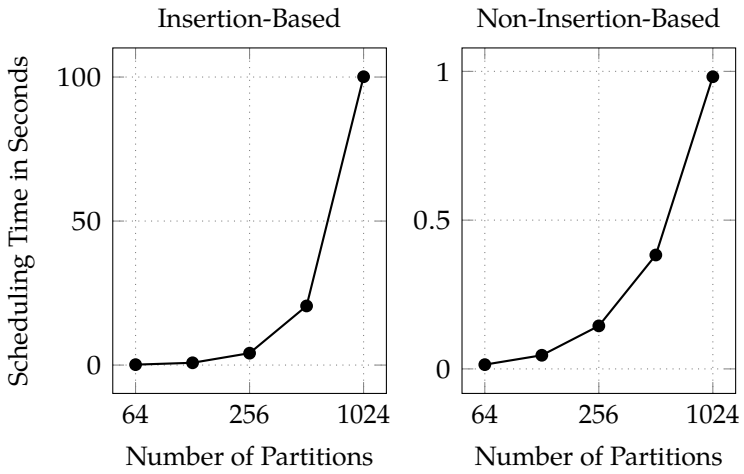
these two algorithms are shown. It clearly stands out that graph construction times for these two benchmarks are significantly higher than for bitonic sort when graphs of similar size are compared, especially on the x86 architecture. Since the index computation (which is equally complex for all partitionings) is mostly responsible for the graph construction time, the optimal partitioning for each input size was used. Nonetheless, the two figures also show a table with information about the number of partitions for a better comparison with Figure 9.13. As before, these numbers include all constructed redundant actor nodes, comparison actor nodes and data nodes.

9.4.2 Scheduling

After the previous sections covered dataflow execution and graph construction, this section provides execution times for the extended HEFT scheduling heuristic described in Chapter 7. Since the scheduling performance is less important for the RTE than the performance of dataflow execution, the section focuses on bitonic sort graphs and only presents measurements for non-sectioned graphs. The choice fell on bitonic sort since its graphs are the largest for the input data sets used in the experiments (see Table 9.1). With regard to sectioning, the scheduling of a sectioned graph has the potential to take less time, since it may be possible to schedule some sections in parallel. In theory, all sections could be scheduled in parallel, but in order to reduce graph execution times, it is beneficial to consider the data dependencies between sections in the scheduling process and only schedule independent sections in parallel.

How long the execution of the scheduling routine takes depends on multiple factors. The first and most obvious factor is the number of nodes in the graph. However, the dependencies between nodes are also important since the HEFT heuristic iterates over the predecessors of each node. In this regard, the graphs for the three benchmark applications are similar, since most actor nodes depend on two or three data nodes. The third factor is the number of processing elements. Except for some actors on the Kalray architecture, which have to be executed on the driver tile, HEFT considers each processing element as a possible candidate for each actor node. Lastly, the RTE's HEFT implementation can be run either with or without the insertion-based policy (see Section 2.3.3). Based on the graph structure, the search for suitable gaps in the schedule can have a huge impact on the execution time of the scheduling heuristic.

Results for the x86 architecture are shown in Figure 9.16. It should be noted that the RTE always computes three schedules in order to support the different redundancy configurations. Thus, each time in the figure represents the sum of three executions of HEFT. The total number of scheduled actor nodes for each partitioning

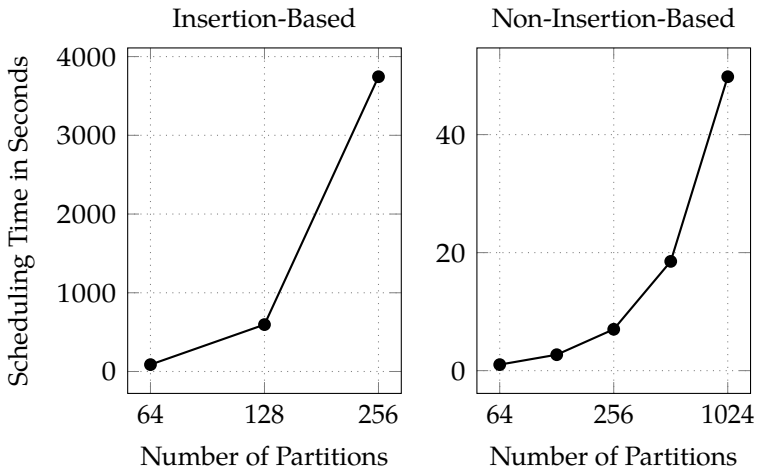


Partitions	Actors
64	14531
128	37251
256	92931
512	226819
1024	543747

Figure 9.16: Scheduling time and number of scheduled actors for bitonic sort graphs scheduled with the two variants of HEFT on x86

is shown in the table on the right. For each type of actor that supports redundancy eight actors are scheduled in total. Such actors are executed once in a non-redundant graph execution and two or three times in redundant graph executions, i.e. they have to be scheduled six times in total. The two remaining actors are the comparison and voting actor required in redundant graph executions. For each actor that does not support redundancy, on the other hand, only three actors are scheduled in total. The number of actors in the graph excluding redundancy-related actors can be found in Table 9.1 for comparison. Figure 9.16 shows that the difference between HEFT with and without the insertion-based policy is quite large so that the former may become unfeasible for larger graphs with a similar structure.

Figure 9.17 shows measurements for the same graphs on the Kalray architecture. Scheduling on the Kalray takes significantly more time than on x86. This has two reasons. First, the Kalray has sixteen compute tiles and one driver tile compared to the eight threads (due to the four physical cores with Hyper-Threading) which have to be considered in the scheduling process on the Intel CPU. Second, the HEFT scheduling heuristic is a single-threaded algorithm, and the Kalray's single-core performance is rather low, especially when the DDR memory is involved. Furthermore, the considered bitonic sort graphs only consist of one section. If the graphs had multiple sections, it would be possible to schedule them in parallel. However, since scheduling requires access to the graph, it is limited to the IO subsystem and thus only four sections at most can be scheduled in parallel. In contrast to Figure 9.16, the graph for insertion-based HEFT does not show results for 512 and 1024 partitions. For dataflow graphs this large, the insertion-based HEFT heuristics becomes infeasible on the Kalray. If a user wishes to use insertion-based



Partitions	Actors
64	14531
128	37251
256	92931
512	226819
1024	543747

Figure 9.17: Scheduling time and number of scheduled actors for bitonic sort graphs scheduled with the two variants of HEFT on Kalray

HEFT, it is advisable to compute the schedule on a different hardware architecture, for example x86, and load it into the Kalray's DDR memory via the RTE's import functionality.

9.5 Adaptive Redundancy

This section provides experimental results related to the adaptive redundancy methods described in previous chapters. For this purpose, two experiments were conducted. The first experiment shows the overhead caused by redundancy changes during runtime on the x86 and Kalray architecture, while the purpose of the second experiment was to determine the quality of schedules created by the modification approach described in Section 7.4.4.

9.5.1 Redundancy Changes

Since the evaluated RTE uses the same graphs and schedules for non-redundant graph executions and those with redundancy on the same PE, switching between these configurations causes nearly no overhead. For the other redundancy configurations, however, the system needs to switch to a different schedule and restructure the graph in order to handle redundant actors and comparison actors correctly. Therefore, this section focuses on redundancy changes involving configurations with redundancy over different PEs.

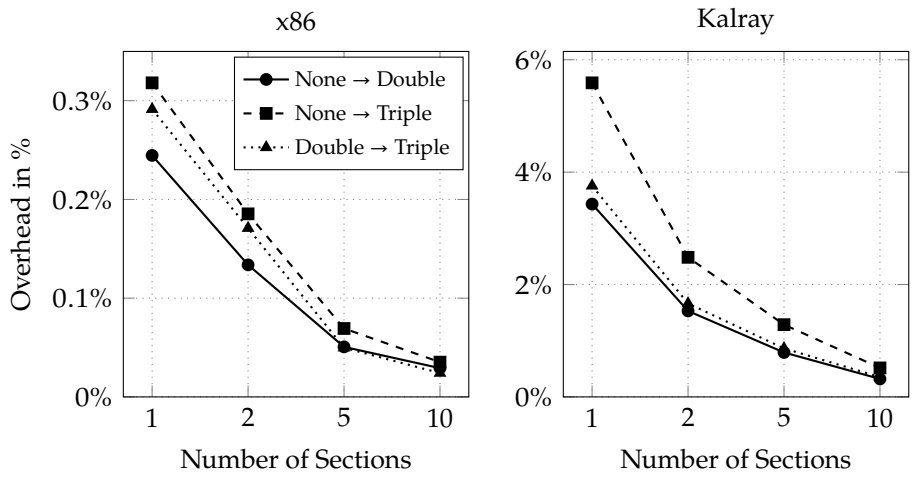


Figure 9.18: Overhead caused by one change of the redundancy configuration for the matrix multiplication with different numbers of sections

In the RTE implementation, all redundancy-related actors are created along with the standard actors, and thus the system only has to modify the edges between actors. The left side of Figure 9.18 shows how much overhead a single redundancy change for the matrix multiplication application on the x86 architecture would cause. The execution times from Section 9.2.3 were used as a baseline. Since the difference in the graph structure between no redundancy at all and three redundant executions per actor is the largest, the reconfiguration time is the longest in this case. The shape of the three curves is also quite as expected. Because the section boundaries in the three benchmark applications were chosen so that the sections are as equal in size as possible, the overhead for a redundancy change in case of two sections is about halved compared to the situation with one section (i.e. changing the redundancy of the entire graph). Overall, the overhead of a redundancy change on the x86 architecture is very low.

While the shape of the curves is similar, the overall numbers are much higher on the Kalray architecture. This is due to the structure of the Kalray Bostan architecture. In contrast to transfers of larger portions of data between the DDR memory and a local memory, random accesses to the DDR memory, which are required to restructure graphs, are rather slow. However, using the DDR memory to store graphs is the only option because the local memories on the chip are too small.

Figure 9.18 shows only the overhead for an increase in the level of redundancy. Further experiments showed that on Kalray, a decrease in the redundancy level takes less time than an increase. Differences of 10% to 34% between redundancy level increases and decreases were measured, with the difference being lower for a

redundancy change of two levels. In contrast to the Kalray architecture, redundancy increases and decreases take a similar time on x86.

9.5.2 Schedule Modification

The purpose of the last experiment was to determine the effects of schedule modifications as described in Section 7.4.4. To estimate how big the difference between the makespan of modified schedules and computed schedules is, the two types of schedules were compared for various graphs. In this regard, it is important to note that it is difficult to make a general statement on the quality of modified schedules. The main problem here is basically the same as for comparisons between different DAG scheduling heuristics and lies in the construction of suitable DAGs. A thorough comparison requires a wide variety of graphs, and thus different algorithms for generating random DAGs were proposed. However, it is difficult to estimate how realistic randomly generated DAGs are, i.e. to which degree the results from experiments with randomly generated graphs apply to task graphs of real applications. A description and comparison of random DAG generation approaches was published by Canon et al. [CSH19].

For the experiment, the classic Erdős-Rényi method was used. In this approach, a random upper-triangular adjacency matrix is generated by setting the corresponding entries with a firm probability p to a non-zero value. Based on their comparison of random graph generation methods, Canon et al. came to the conclusion that edge probabilities between 0.05 and 0.15 lead to the most interesting graphs [CSH19]. Therefore, a probability of $p = 0.15$ was chosen for the experiments. Node and edge weights were generated from a uniform integer distribution with nodes having higher weights on average. Furthermore, all comparison nodes and all edges from redundant nodes to comparison nodes had very small weights (for example because only two checksums are compared). An overview of all parameters that were used for the generation of random graphs is shown in Table 9.2.

Table 9.2: Overview of parameters used for graph generation

Parameter	**Value**
Number of Nodes	500
Edge Probability	0.15
Possible Node Weights	$\{10, 11, \ldots, 50\}$
Possible Edge Weights	$\{5, 6, \ldots, 15\}$
Comparison Node Weight	1
Comparison Edge Weight	1

The HEFT scheduling heuristic was used to schedule the generated graphs for different numbers of PEs. Since all hardware architectures covered in this thesis are relatively homogeneous, the schedule modification mechanism was only evaluated for homogeneous PEs. For each number of PEs the following steps were executed 100 times:

1. Generate a random graph using the Erdős-Rényi method
2. Compute a schedule A for x PEs using the HEFT scheduling heuristic
3. Compute a redundant schedule B for $x + 1$ PEs using the extended HEFT heuristic that assigns redundant actors to different PEs
4. Choose a random PE and create a non-redundant schedule C for x PEs by modifying schedule B
5. Determine the makespans M_A and M_C of schedules A and C, respectively
6. Compute the relative makespan M_{avg} as follows: $M_{avg} = \frac{M_C}{M_A}$

The procedure for the experiment with schedules with three redundant executions per actor was analog.

Figure 9.19 shows the results. The upper and lower curves represent the smallest and largest observed relative makespan respectively, while the curve in the middle shows the average of all 100 relative makespans for each number of PEs. The left side shows the normalized makespans of non-redundant schedules resulting from the modification of redundant schedules. An observation is that the described modification mechanism works best when there are at least eight PEs in the system

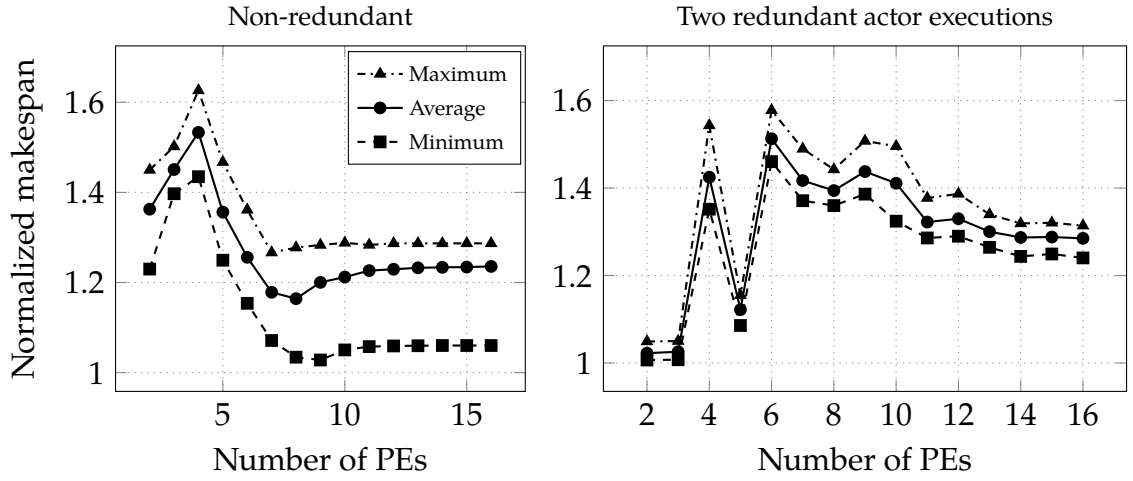


Figure 9.19: Makespans of modified schedules normalized to makespans of schedules computed with HEFT

(note that the figure shows the number of PEs after a permanent fault). Less than eight PEs lead to a larger difference since the impact of the modification is higher. For smaller numbers of PEs, up to 63% larger makespans were observed. The makespan of the best schedule was about 3% larger than the corresponding HEFT schedule. For 12 or more PEs, the modified schedules have a 23% larger makespan on average.

The right side shows the same curves for result schedules with two redundant actor executions, i.e. schedules with three executions per actor were modified. For 14 or more PEs, the difference between modified and HEFT schedules is 29% on average. Most noticeable is the low difference in the setting with 5 PEs. In this case, the initial schedule before modification was computed for 6 PEs. Since it is a multiple of three, this number of PEs is beneficial for a schedule with three redundant executions per actor. Using HEFT to compute a schedule with two redundant executions per actor for a system with 5 PEs on the other hand leads to a relatively poor result, and thus the difference between the modified schedule and the HEFT schedule is rather small.

As described in Section 7.4.5, an advantage of schedule modification is the faster error recovery compared to running a scheduling heuristic at runtime. In order to evaluate the difference in recovery time between schedule modification and re-scheduling, the time it takes to modify the schedule and the time it takes to compute a new schedule with HEFT were measured for different numbers of processing elements on x86. Figure 9.20 shows times for schedule modification normalized to the times of HEFT scheduling. Since schedule modification is in contrast to HEFT

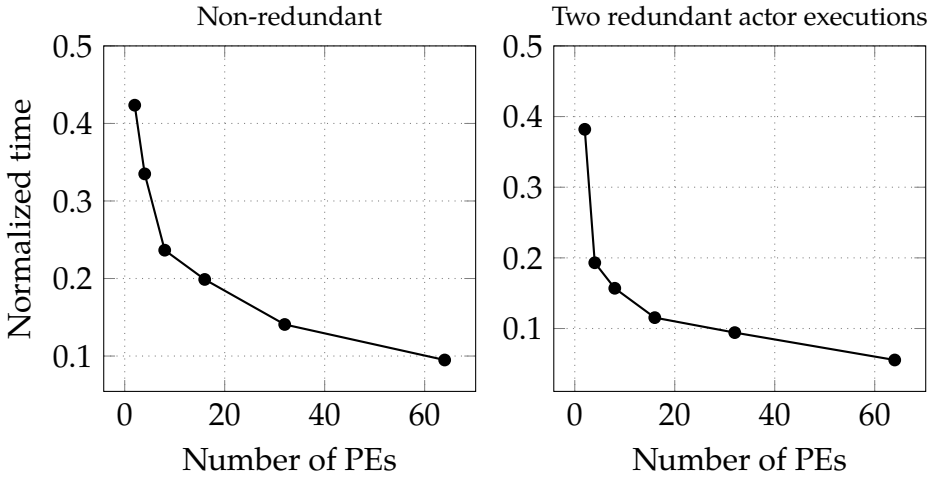


Figure 9.20: Time for schedule modification normalized to the HEFT scheduling time for different numbers of PEs on x86

rather independent of the number of PEs, the performance benefit increases with the number of PEs. For 64 PEs, the modification process takes less than 10% (and less than 6% for a result schedule with two redundant executions per actor) of the time for computing a new schedule with HEFT.

9.6 Summary

This chapter provided execution time measurements for the three RAPID applications described in Section 4.6 under various RTE configurations. The two hardware architectures used in the experiments were a standard x86 personal computer and a Kalray Bostan MPPA. A large portion of the chapter discussed dataflow executions since the execution time of graph construction and scheduling routines are less important in practical use. Varying different properties of the graph or RTE settings may or may not have a large influence on the graph execution time. For the three benchmark applications, the number of partitions into which the input data is divided had a much greater impact than the number of graph sections. The use of online scheduling based on work stealing was in all cases slightly beneficial on the x86 architecture but lead to a lower performance for FFT on the Kalray architecture. Because only computation (and neither memory restructuring actors nor data transfers) is executed redundantly, redundant graph executions often take less than twice (or three times) the baseline execution time. The chapter also covered graph construction and scheduling. Since these two procedures require a high single-core performance, it is advisable to rather use the RTE's import/export functionality on the Kalray and create graphs and schedules on an x86 machine. In the last part of the chapter, two aspects related to the adaptive redundancy features of the proposed RTE were evaluated. Redundancy changes only cause a small overhead in graph executions, especially when the graph is divided into smaller sections. Schedule modification leads to higher makespans compared to schedules created with a proper scheduling heuristic. However, the procedure is more suitable for graceful degradation compared to re-scheduling due to the lower complexity and thus lower error recovery times.

10

Related Work

To meet the requirements of different application areas and to cover a variety of hardware architecture, numerous dataflow systems have been developed. Many dataflow systems are designed for use in large computer clusters, but there are some approaches that are usable in smaller systems. Further, since fault tolerance is an important aspect for not only small embedded systems but also large computer clusters, suitable dataflow systems with built-in fault tolerance have been proposed. Some dataflow approaches include fault tolerance mechanisms from the beginning while other dataflow systems were extended in later proposals. This chapter addresses related approaches from the domain of dataflow-based frameworks. A selection of systems is described, and similarities and differences to the proposed programming model and runtime environment (RTE) are discussed.

Section 10.1 shows computing frameworks for large-scale distributed systems. Even though these frameworks target big data applications and are less suitable for smaller systems, there are some interesting similarities to the proposed RTE, for example in the programming model and the use of dataflow graphs. After that, frameworks that are suitable for smaller system, like single workstations or embedded systems, are discussed in Section 10.2. Some big data frameworks from Section 10.1 already exhibit fault tolerance mechanisms up to a certain degree, but there are also dataflow approaches with special focus on fault tolerance. A small selection of such approaches is shown in Section 10.3.

10.1 Computing Frameworks for Large-Scale Distributed Systems

The emergence of big data applications in recent years has lead to the development of frameworks which are able to run computations on large-scale distributed systems. Since dataflow is an efficient way to handle the required parallelism, many of these frameworks are based on the dataflow principle. This section provides information about four big data frameworks, which are related to the proposed RTE because of their programming model and dataflow characteristics. What these frameworks have in common is that they allow users to write parallel programs by providing a set of high-level operations. The framework then handles communication and work distribution automatically.

10.1.1 Apache Spark

The latest version of Apache Spark [Zah+12] supports multiple programming models. This section focuses on the original programming model based on data structures called *resilient distributed datasets* (RDDs). From a user's point of view, RDDs are partitioned read-only collections of data elements. Internally however, RDDs do not contain data at all times but instead keep track of their *lineage*, i.e. how their data can be computed from the data of other RDDs. Users specify the lineage of RDDs via *transformations*. Besides transformations, the programming model provides *actions* to start the graph execution. The lineage of all RDDs involved in a Spark program forms a directed acyclic dataflow graph, called *lineage graph*. When such a lineage graph is executed, the Spark RTE distributes the work to the nodes in the cluster. To improve performance, Spark tries to make use of data locality whenever it is possible. Another important aspect of lineage graphs is that they consist of stages. Each stage contains as many transformations with only narrow dependencies as possible. The difference between narrow and wide dependencies is shown in Figure 10.1. Transformations with wide dependencies lead to an expensive shuffling at runtime and therefore such operations should only be used when they are necessary. Apache Spark also provides a fault tolerance mechanism. The error model of Spark only considers node failures. If a node fails, lost partitions of this node are recomputed from the previous checkpoint. The programming model provides functions that allow users to specify which RDDs in the lineage graphs are checkpoints.

The biggest similarity between approach from this thesis and Apache Spark is the programming model, with RAPIDs serving the same purpose as RDDs. But since the RAPID programming model targets small embedded systems, the number

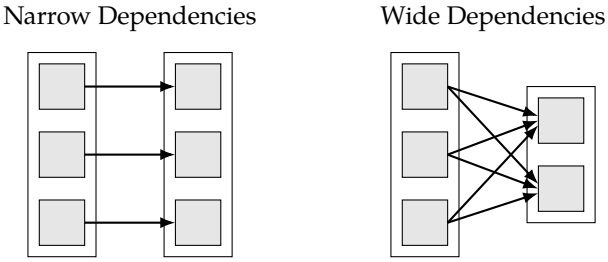


Figure 10.1: Examples of narrow dependencies and wide dependencies [Zah+12]

of provided operations is reduced, especially the number of operations with wide dependencies. Only `reorder` and `repartition` belong to this category. Another similarity is the use of directed acyclic graphs as dataflow graphs. However, lineage graphs differ from RAPID dataflow graphs in their structure because they are based on RDDs. RAPID dataflow graphs, on the other hand, are based on partitions, and it is not always possible to reconstruct the exact RAPID program that lead to a given graph.

A difference between the two approaches is that the proposed RTE does not only support online scheduling like Spark but is also able to execute graphs according to fixed schedules. Further, it allows users to specify the online scheduling policy or offline scheduling algorithm. Another major difference lies in the fault tolerance aspect. Apache Spark only considers node failures, while the proposed dataflow RTE can execute graph nodes redundantly and is therefore able to detect errors in computations. The concept of checkpoints is also quite different in both approaches. While checkpoints in Spark are used to recompute data in case of a node failure, checkpoints in the RAPID programming model are used to specify the granularity of the adaptive redundancy.

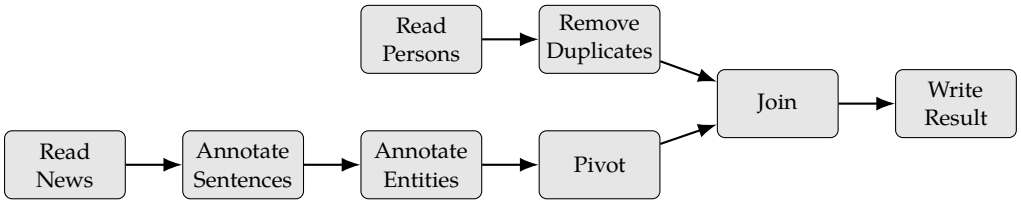
10.1.2 The Stratosphere Project

The Stratosphere project [Ale+14] is the origin of Apache Flink. Its software stack consists of three layers, namely the *Sopremo* layer, *PACT* layer and *Nephele* layer. These layers provide different programming models, each with a different level of abstraction. The programming model of the Sopremo layer is the most declarative and therefore high-level one, while the Nephele layer provides the most low-level functionality out of the three. Programs in the Sopremo layer are expressed through *Meteor* scripts. The functionality is similar to other query languages since it provides, for example, selection, projection, join, group and union operators. Meteor scripts are transformed into Sopremo plans by the Meteor parser. Sopremo plans are DAGs

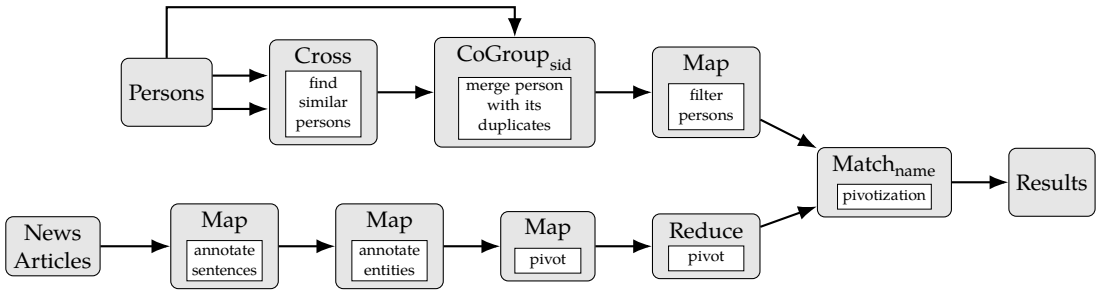
whose nodes are Sopremo operators. Edges in the graph result from the variables in the Meteor script.

A program assembler then translates the Sopremo plans into PACT programs, which are also DAGs. PACT programs are usually larger than the Sopremo plans they are created from because each operator in the Sopremo plan is translated into one or more operators in the PACT program. These operators correspond to the second-order functions PACT provides, for example *map* and *reduce*. Second-order functions require a dataset and a first-order *user-defined function* (UDF) as arguments. Therefore, an operator consists of a PACT second-order function and concrete UDF instance. Further, it contains information on how datasets are partitioned into *parallelization units* (PUs). During graph execution, the UDF of an operator is applied to each PU independently. Figure 10.2 shows an exemplary Sopremo plan and the PACT program returned by the assembler. Inner boxes in the PACT program graph represent UDFs. This example illustrates on multiple occasions that one operator in the Sopremo plan is usually translated into multiple operators in the PACT program. The *Remove Duplicates* operator in Figure 10.2a, for example, corresponds to three operators in Figure 10.2b, namely *Cross*, *CoGroup* and *Map*.

In the next step, a PACT program is compiled into a Nephele job graph by Stratosphere's optimizer. The optimization of a PACT program consists of multiple phases. After the optimization is complete, the job graph is submitted to the Nephele ex-



a) Sopremo plan example



b) Corresponding PACT program

Figure 10.2: Stratosphere graph examples [Ale+14]

ecution engine, which spans it to the execution graph. The difference between a Nephele job graph and an execution graph is that the latter contains a node for each parallel instance of a node in the former. Execution graphs are the representation of a program that is finally executed on the computer cluster.

Stratosphere provides a fault tolerance mechanism which is predicated on log-based rollback recovery. Similar to Apache Spark, intermediate results are *materialized* by the system, and lost data is recomputed from the last materialized results when an error occurs. However, the rollback system in Stratosphere is more complex than in Apache Spark. Since Stratosphere supports streaming (in contrast to Spark which uses the *blocking operator model*), operators can be executed if their input data is only partially available. In case the execution of an operator fails, dependent operators may have been started already and must be considered for the rollback.

The proposed dataflow RTE shares several similarities to the Stratosphere project. While it does not provide an abstraction layer equivalent to Sopremo, the RAPID programming model is similar to the PACT programming model. RAPID operations correspond to second-order PACT functions and RAPID functions represent the counterpart to the UDFs in PACT. But as mentioned in the comparison with Apache Spark, the RAPID programming model provides a reduced functionality since it also targets smaller embedded systems.

Other similarities lie in the use of DAGs as dataflow graphs and the construction of graphs. One RAPID function usually leads to multiple nodes in the RAPID dataflow graph. This is quite similar to the graph transformations in Stratosphere. The more a graph is transformed while it passes through the layers of Stratosphere, the more it gets expanded. After all transformations are complete, an execution graph in Stratosphere contains nodes for all parallel instances of nodes in previous graph revisions. Therefore, execution graphs are closest to RAPID dataflow graphs.

With regard to fault tolerance, the error model of Stratosphere is similar to Apache Spark. Thus, the similarities and differences described in the comparison with Apache Spark also apply to Stratosphere.

10.1.3 Thrill

Thrill [Bin+16] is a framework with the goal to improve the performance of big data applications by running native binaries on the nodes in the computer cluster. In contrast, both Apache Spark and Flink run bytecode in the Java Virtual Machine. The programming model of Thrill is similar to Apache Spark. It is based on a data structure, called *distributed immutable array* (DIA) and provides various operations, like *map* and *reduce*. A difference is that DIAs are based on arrays, i.e. data structures with a fixed element order, whereas RDDs in Spark are based on sets. Like in the other discussed frameworks, operations in Thrill build a directed acyclic dataflow

graph. The execution of programs in Thrill, however, works differently than in other frameworks since every node in the computer cluster executes the same binary. In contrast to Apache Spark and Flink, Thrill does not provide fault tolerance.

With regard to the importance of element order, DIAs are similar to RAPIDs. Therefore, operations in the RAPID programming model are closer to Thrill operations than Spark operations. In Spark, for example, the zip operation requires key-value-pairs. A difference, however, is that users are not able to explicitly specify a partitioning in Thrill. DIAs are always distributed evenly among the workers by the RTE. As a result, Thrill's programming model does not provide partition-wise operations like the RAPID programming model does. Data elements in Thrill do not have to be of fixed size. It is only required that suitable serialization and deserialization methods are available. Thus, RAPID programs using partition-wise operations can also be expressed in Thrill. A selection of operations is shown in Table 10.1. For most of the listed operations an equivalent RAPID operation exists, sometimes with a different name, for example *Generate* (`parallelize`), *Window* (`combine`) and *AllGather* (`finalize`). Exceptions in the table are *Filter* and *Sort*. For these Thrill operations, no corresponding RAPID operations exist.

A similarity to the dataflow system of this thesis is the use of C++ to achieve a high performance. But, in contrast the RTE implementation for NoC-based architectures, Thrill does not have a special driver node in the computer cluster. All nodes in Thrill execute the same binary. In the proposed NoC-based RTE implementation, compilation returns two binaries, one for the driver tile and one for the compute tiles to overcome the limited memory of compute tiles on the target architecture.

Table 10.1: Selection of Thrill operations [Bin+16]

Operation	**User Defined Functions**
`Generate(n): [0,…,n-1]`	`n: DIA size`
`Generate(n,g): [A]`	`g: unsigned → A`
`Map(f): [A] → [B]`	`f: A → B`
`FlatMap(f): [A] → [B]`	`f: A → list⟨B⟩`
`Filter(f): [A] → [A]`	`f: A → bool`
`Sort(c): [A] × [A]`	`c: A × A → bool`
`Concat(): [A] × [A] × ⋯ → [A]`	
`Zip(z): [A] × [B] × ⋯ → [C]`	`z: A × B × ⋯ → C`
`Window(k,w): [A] → [B]`	`k: window size` $w: A^k \to B$
`Execute()`	
`AllGather(): [A] → list⟨A⟩`	

`[A]` is a shorter notation for `DIA⟨A⟩`.

10.1.4 TensorFlow

A computation framework that has gained much importance in recent years is Google's TensorFlow [Aba+15; Aba+16]. The focus of TensorFlow is different to the other discussed frameworks since it lies on machine learning, but its dataflow approach is quite similar. TensorFlow supports various operations that are commonly used in the implementation of neural networks, for example *matrix multiplication*, *convolution* and *sigmoid*. A computation in TensorFlow is described by a directed graph. To create graphs, TensorFlow provides frontends in different programming languages, like C++ and Python. Graphs consist of nodes representing the instantiation of TensorFlow operations. Since TensorFlow supports heterogeneous systems, it distinguishes between *operations* and *kernels*. Operations represent an abstract, high-level concept, while kernels are concrete implementations for devices like CPUs, GPUs or hardware accelerators. Besides operations, *tensors* are the second important concept in TensorFlow. Formally, tensors are multi-dimensional arrays whose exact type can often be inferred at graph construction. During graph execution, tensors flow along the data edges of the graph. Besides data edges, TensorFlow graphs may also contain edges which specify how control information is exchanged between nodes during graph execution. Since operations in TensorFlow are not necessarily stateless, such control dependencies can, for example, be used to enforce the right order of node executions.

There are some notable similarities and differences between the proposed framework and TensorFlow. Both TensorFlow and the RAPID programming model encourage users to build a graph once and execute it many times with different data. A difference is that graph nodes in TensorFlow can involve state, so that they behave differently in subsequent graph executions. In RAPID dataflow graphs, data nodes have to be declared as input nodes and their data has to be changed between graph executions to achieve a similar effect.

A major characteristic that separates TensorFlow from the other discussed frameworks and from the proposed RTE is the use of potentially cyclic graphs as program representations. This leads to the concept of dynamic dataflow described in Section 2.1.3. For this purpose, TensorFlow provides a small set of control flow nodes, for example *switch* and *merge* nodes, which can be used to skip subgraphs during execution. Consequently, a token matching mechanism is necessary. Tags implemented in the TensorFlow RTE are conceptually very similar to tags in the MIT Tagged-Token machine.

Because of the potentially cyclic graphs, node placement is more difficult in TensorFlow. By supporting only acyclic graphs, the proposed RTE can use standard DAG scheduling heuristics. In TensorFlow, node placement requires a simulated execution of the graph based on a greedy heuristic.

TensorFlow provides a mechanism, which is similar to the fault tolerance in Apache Spark, to handle node failures during the training of neural networks. In TensorFlow, users have to explicitly specify checkpoints via the provided checkpointing library. It is important to note that the creation of checkpoints and the neural network training are executed concurrently. Since checkpointing in TensorFlow involves no synchronization by default, checkpoints may not be consistent. However, there are workarounds for the case that consistent checkpoints are required.

10.2 Graph-Based Frameworks Suitable for Smaller Systems

The frameworks described in previous sections target large computer clusters and are well suited for big data computations. Using their reference implementation on single machines would lead to unnecessary overhead. Other graph-based frameworks are better suited for small systems. This section provides information about some approaches regarding this topic.

10.2.1 TensorFlow Lite

TensorFlow Lite [Goo; Lee+19] is a machine learning framework targeting mobile devices and embedded systems. In comparison to standard TensorFlow its functionality is limited. Only a subset of all TensorFlow operators is supported by TensorFlow Lite. The reason behind this is to keep the size of binaries low. To reduce the binary size even further, the training of neural networks is not supported. TensorFlow Lite currently only supports pre-trained networks. The workflow is usually as follows: Users first create a dataflow graph using the programming interface of standard TensorFlow. This graph represents a neural network and is trained on a high-performance architecture. The trained model is then exported to a file and converted into a TensorFlow Lite-compatible format. In TensorFlow Lite, the file is imported and executed arbitrarily often with different input data.

Similar to TensorFlow and TensorFlow Lite, the RAPID programming model also gives users the possibility to import and export graphs. In the proposed RTE, graphs can be built on the developer's workstation, for example, and exported for use in the target embedded system. The training of neural networks is expensive and thus TensorFlow Lite-compatible models are trained before they are used in the target system. Since the RAPID programming model does not focus on machine-learning but general-purpose computing, there is no training phase in general. However, offline scheduling also has the potential to be expensive. Therefore, the RTE described in this thesis provides an import function for schedules.

10.2.2 DARTS

The Delaware Adaptive Run-Time System (DARTS) [SZG13] is an implementation of the *codelet model*. Codelets are non-preemtive collections of machine instructions and represent the *scheduling quantum* in the codelet execution model. The system expects codelets to behave like pure functions, but it does not forbid to specify codelets that modify the global state. Codelets are similar to dataflow actors. The difference to the latter lies in the firing. A codelet can fire as soon as all *events* are met. Not only data dependencies can serve as events but also other conditions with regard to, for example, shared resource requirements, bandwith or power. A *codelet graph* (CDG) consists of codelets that are linked together and act as producers and/or consumers. Result data of a codelet can be used by multiple subsequent codelets. The codelet model also introduces the concept of *threaded procedures* (TPs), which represent containers for codelet graphs. Multiple TPs can be linked together to form a larger dataflow graph. Instances of TPs are always bound to *clusters*. In the codelet model, clusters are parts of chips which are connected through arbitrary interconnects and consist of multiple cores and memories. Each cluster must contain exactly one *scheduling unit* (SU), which is a special core responsible for the scheduling of TPs. All other cores in the cluster are *compute units* (CUs). In contrast to SUs, CUs run a micro scheduler which manages the scheduling of codelets. In DARTS, a work-stealing scheduling approach is used for the distribution of TPs over clusters. Furthermore, a central queue distributes the codelets to the CUs inside a cluster.

The codlet model targets different kinds of multicore architectures, from single-processor systems up to large computer clusters. DARTS is an implementation of the codelet model which is suitable for single nodes with possibly multiple processors. However, it does not explicitly target embedded systems.

Further, codelets are very similar to RAPID functions in the RAPID programming model. Instances of codelets correspond to actor nodes in RAPID dataflow graphs. In both DARTS and the RAPID programming model, users can divide the graph in TPs and sections, respectively. However, the purpose of this division differs in both systems. While TPs in the codelet model are used in favor of hierarchical scheduling, graph sections are an important concept for the adaptive redundancy concept of the proposed RTE.

Another difference is that the codelet model allows users to specify loops in dataflow graphs. To reduce complexity in the RTE, only three types of loops are allowed, namely a serial loop, a TP-parallel for-each loop and a codelet-parallel for-each loop. The latter two require that all loop iterations are independent of each other, while the first assumes that each iteration depends on the previous one.

10.2.3 Concurrent Collections

Another programming model with a similar focus is the Concurrent Collections (CnC) language family [Bud+10]. A CnC specification is a graph consisting of instances of *step collections*, *data collections* and *control collections*. Step collections represent computation and are therefore comparable to procedures. Consequently, instances of step collections correspond to procedure invocations. In CnC, a control collection *prescribes* a step collection. This means that a control collection instance can cause a step collection instance to execute with the former as an input. Steps also read and write instances of data collections. The dependencies between the three kinds of nodes form a CnC graph. An example graph is shown in Figure 10.3. The graph models an application that takes a set of strings, splits them into words and puts the words in uppercase. Inputs are provided from the outside of the CnC application, called the *environment* (env). Similarly, results are passed to the environment again. It should be noted that CnC is a coordination language, i.e. CnC specifications are required to assemble the steps which are implemented in a separate programming language.

Having different kinds of nodes in the graph is a concept in both RAPID and CnC. However, there are no control nodes in RAPID dataflow graphs. CnC is based on dynamic dataflow, and control collections are required to forward tags in the CnC graph. RAPID graphs are more static and therefore do not require tags.

Another similarity is the data-parallel execution of step collections in CnC. Elements of data collections are possibly processed in parallel based on the implementation. This is comparable to the data-parallel execution of actors in the RTE implementation for the Kalray MPPA described in this thesis. As with the RAPID programming model, it is possible to develop implementations of CnC for hardware architectures with shared memory as well as distributed memories.

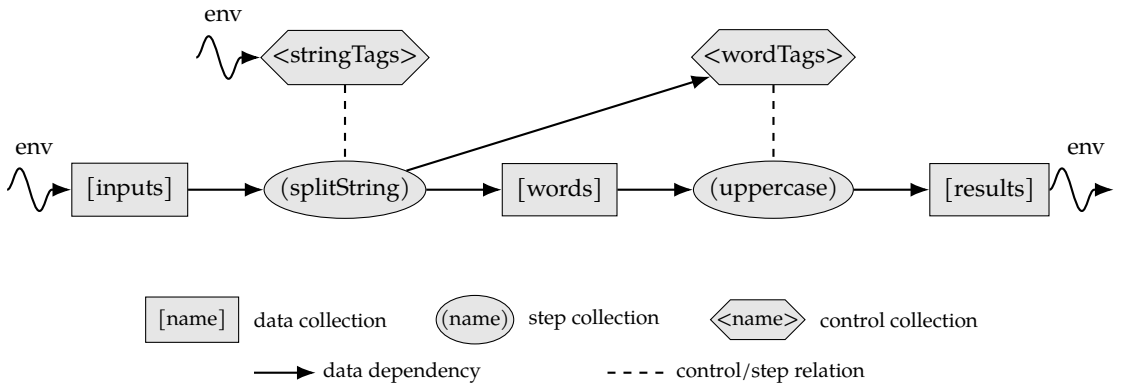


Figure 10.3: CnC graph example [Bud+10]

10.2.4 TeamPlay Coordination Toolchain

The last approach addressed in this section is the coordination toolchain from the TeamPlay project [Roe+20]. This project targets embedded systems based on off-the-shelf heterogeneous multicore processors. Major part of the TeamPlay toolchain is a compiler for the specially developed coordination language. This language is used to specify the *components* of an application. Components refer to functions specified in a different language (usually C) and can have various properties regarding, for example, deadlines or security. The entirety of the components in an application forms a DAG, in which edges correspond to FIFO queues connecting to the input and output ports of components. Conceptually, components are stateless, but it is possible to model state information by introducing state ports. These special ports only allow self loops in the graph. Each component can have an arbitrary number (including zero) of input, output and state ports.

The coordination language puts the focus especially on the non-functional properties of code execution. More precisely, the three considered non-functional properties are energy, time and security (ETS). While time and energy consumption vary for different hardware architectures, security is a property affecting the implementation of components. An abstract component is shown in Figure 10.4. This example actually shows the most general form of component, also called a *multi-version component*. The inner boxes visualize the different versions, each with its own name, code and ETS-contracts.

The most important part of the toolchain is a coordination compiler which runs various analyses on a given coordination program, computes a suitable scheduling policy and generates code in the target programming language. Once the compilation of the coordination program is finished, the application can be compiled a second time with a standard compiler, for example a WCET-aware C-compiler. Based on the application requirements, the coordination compiler has different options regarding the execution of generated programs. Components in the gener-

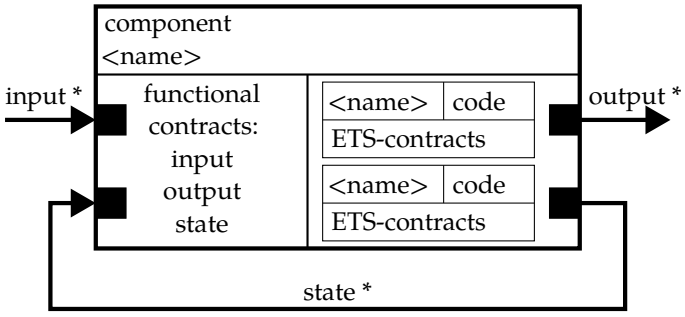


Figure 10.4: Abstract component in TeamPlay [Roe+20]

ated program may either be executed at statically determined time slots (where the required data is guaranteed to be present) or data-driven similar to the execution of actors dataflow-based frameworks.

While the coordination language and its compiler-based toolchain is quite different to the RAPID programming model, the use of DAGs as the main representation for programs is similar. A feature that separates the TeamPlay toolchain and the proposed RTE from many other approaches is its support for offline scheduling. However, fixed schedules generated by the coordination compiler are fully static with exact timings whereas statically computed schedules in the proposed RTE only specify the actor assignment and order. Further, both approaches have a different focus regarding program executions. While the coordination toolchain in the TeamPlay project aims at involving ETS-properties into program executions, the focus of the RTE described in this thesis lies in the support for adaptive redundancy.

10.3 Fault-Tolerant Dataflow Approaches

Section 10.1 already described that many big data frameworks provide a fault tolerance mechanism. However, since fault tolerance is a secondary goal in these frameworks, it is limited to node failures in the computer clusters. This section highlights some dataflow-based systems which focus on fault tolerance. Some systems utilize redundancy and can therefore detect errors in computation, storage or data transfers. Further, the described approaches provide error recovery through rollback mechanisms.

10.3.1 Fault-Tolerant Dataflow in a Teradevice System

Weis et al. describe fault tolerance in a teradevice dataflow system [Wei+16]. A tiled multiprocessor is assumed as the underlying hardware architecture for this type of system. In the processor, nodes are connected through a 2D mesh-based interconnection network. Each node consists of multiple cores, memory and a node manager, which are connected via a crossbar. The node manager consists of a *distributed thread scheduling unit* (D-TSU) and a *distributed fault detection unit* (D-FDU). Further, each core contains a *local thread scheduling unit* (L-TSU) and a *local fault detection unit* (L-FDU). The L-FDU gathers information about errors in the core and periodically sends health messages from the core to the D-FDU. These messages include, among other things, information about errors in caches, the bus unit and load-store unit. The D-FDU then analyzes the information and makes predictions about the reliability of cores. If a core is considered faulty, the D-FDU transmits this information to the D-TSU which will then adjust the scheduling

accordingly. Other important tasks of the D-FDU are the monitoring of the D-TSU as well as the monitoring of other nodes' D-FDUs. This ensures that as many errors as possible can be detected. Redundancy is introduced via double execution of dataflow threads. The D-TSU manages the coordination of redundant thread executions. Redundant threads can be scheduled normally, with the only additional constraint that two redundant threads must not be executed on the same core. The L-FDUs compute 32-bit signatures of the thread results. Checksums are then compared by the node's D-FDU. If error recovery actions must be taken, the D-TSU can restart dataflow threads. The rollback mechanism is rather simple due to the side-effect-free execution of threads in the coarse-grain dataflow approach.

Although the described system represents a hardware-based fault tolerance approach, there are similarities to the proposed RTE. First, both systems run on similar tiled hardware architectures. However, the RTE described in this thesis targets embedded manycore processors, and thus the execution of dataflow actors is different. While dataflow actors are assigned to tiles and executed in a data-parallel manner, teradevices have bigger tile-local memories than the embedded manycore architectures, like the Kalray MPPA, so that threads can be mapped to individual cores. While the mapping can be determined offline or online in the proposed dataflow system, the fault-tolerant teradevice system always uses an online scheduling approach.

Redundancy is another aspect with some interesting similarities and differences. In both systems, redundant executions of nodes in the dataflow graph are used to detect errors in computations. The degree of redundancy in the described system is limited to double execution of dataflow threads. The redundancy approach of the proposed RTE additionally provides triple execution of actors and the switching between different redundancy configurations at runtime. However, the fault tolerance approach in [Wei+16] has the benefit that uses special hardware units. Therefore, it has the potential to detect faults that are very difficult or maybe impossible to detect in software.

10.3.2 Dataflow Error Recovery

A software-based approach for fault-tolerant dataflow is Dataflow Error Recovery (DFER) [Alv+14]. Like for most dataflow fault tolerance approaches, redundancy is established by adding redundant nodes to the dataflow graph. Additional commit nodes are responsible for comparing the results of redundant executions. If the system detects that a computation produced wrong results, the respective graph node has to be re-executed. The execution of nodes in DFER is speculative, i.e. subsequent nodes are executed with possibly uncommitted values. In case of a fault, this can lead to a *domino effect* since nodes may have already been executed with

incorrect data. Further, whenever redundant nodes are executed with uncommitted data, it does not make sense to compare their results for errors. If an error occurs, not only the first commit node but also subsequent commit nodes might detect the error, and multiple re-executions would be initiated in the worst case. Therefore, DFER introduces additional dependencies called *wait edges* between commit nodes. Wait edges ensure that commit nodes are executed in an appropriate order so that nodes are only re-executed once per error. In some areas, the latency introduced by the described domino effect in case of an error might be undesirable. Therefore, DFER provides the possibility to insert additional edges between a commit node and all nodes that process the corresponding data. These additional dependencies cause subsequent nodes to wait until the required data has been compared. However, the additional dependencies in the graph may lead to reduced performance.

The redundancy mechanism in DFER with its redundant nodes and commit nodes is very similar to the redundancy approach of RAPID. A difference is that the RTE presented in this thesis does not provide speculative execution in favor of better analyzability. Instead, the RTE provides an adaptive redundancy mechanism that can be used to enhance performance in exchange for a reduction in the degree of redundancy arbitrarily during runtime.

10.3.3 Systematic Event Logging and Theft-Induced Checkpointing

Jafar et al. describe two different fault tolerance approaches [Jaf+05] for the Kernel for Adaptive, Asynchronous Parallel Interface (KAAPI) [GBP07]. KAAPI consists of a dataflow programming model and runtime system. Its programming model is based on the Athapascan interface [Gal+98] and represents a slight extension of C++. Dataflow graphs in KAAPI are built dynamically at runtime from KAAPI programs. These graphs consist of tasks and their dependencies. Internally, the framework itself is also implemented in C++ and uses a custom variant of the POSIX thread interface. There are two types of threads in KAAPI, user threads (or KAAPI threads) and kernel threads. KAAPI sees kernel threads as virtual processors that execute user threads. The latter are responsible for the execution of the tasks in the dataflow graph. An important property of user threads is their non-preemptive nature. With regard to task scheduling, a work-stealing approach was chosen.

One of the fault tolerance techniques described in [Jaf+05] is *Systematic Event Logging* (SEL). The idea behind SEL is to log all modifications of the dataflow graph during runtime, i.e. all additions and deletions of graph nodes. When a processor fails, the log is used to rebuild the associated subgraph from the checkpoint file. Since checkpoints are created on each graph modification, this approach allows the

runtime system to re-execute single dataflow tasks. The other described fault tolerance technique is *Theft-Induced Checkpointing* (TIC). In TIC, checkpointing is done periodically or by the theft of a task (with regard to the work-stealing scheduling policy). Checkpoints created on task thefts are called *forced* checkpoints. Only the virtual processor from which a task was stolen creates such a checkpoint. Normal checkpoints, on the other hand, are created by all virtual processors periodically. Recovery in TIC is similar to SEL, but since not every graph modification is logged, there may be more task re-executions required than in SEL.

KAAPI and the two described fault tolerance mechanisms heavily differ from the RAPID programming model and the associated RTE. The programming model of KAAPI does not follow a functional style like the RAPID programming model. Further, graphs in KAAPI dynamically grow and shrink during runtime, whereas RAPID graphs are fully constructed before their execution. While KAAPI's approach has the advantage that less memory is required (since the graph never exists as a whole), periodic checkpoints have to be created in order to allow re-executions of graph nodes.

10.4 Summary

In the last two decades, many dataflow frameworks have been proposed. This chapter presented a selection of dataflow systems and highlighted the similarities and differences to the proposed RTE. For a better overview, Table 10.2 lists all approaches and compares them with the proposed RTE in terms of their target hardware architecture, programming model, scheduling and fault tolerance.

Many modern big data frameworks are based on the dataflow principle. These frameworks provide a programming model which is used to build dataflow graphs. Graphs are then executed on large computer clusters. While Spark, Flink and Thrill only support DAGs, dataflow graphs in TensorFlow may contain cycles. Some big data frameworks have built-in fault tolerance mechanisms. Spark, Flink and TensorFlow can detect node failures in the cluster and continue with the dataflow execution. Lost data is recovered by re-executing the corresponding graph nodes.

There are also frameworks based on the dataflow principle which are suitable for smaller systems. In their core, these dataflow systems are similar the big data frameworks that were discussed before. The biggest difference is their reduced functionality. TensorFlow Lite, for example, does not provide a programming model by itself. Instead, it only consists of an RTE which can execute graphs created with standard Tensorflow. DARTS, Concurrent Collections and the coordination toolchain developed in the TeamPlay project are other DAG-based systems that can be implemented efficiently on smaller-scale hardware. All of them provide some

kind of programming model or coordination language which is used to specify dataflow graphs.

Lastly, there are dataflow proposals which directly focus on fault tolerance. In contrast to the fault tolerance mechanisms provided by big data frameworks, which can only recover from node failures, some of these approaches also utilize redundancy to detect errors in computations. The teradevice dataflow system and DFER are two examples. KAAPI, another dataflow framework, follows a rather unique aproach and builds dataflow graphs dynamically during dataflow executions. Its logging-based extensions SEL and TIC create checkpoints to recover from process failure but are not able to detect errors in computations.

Table 10.2: Comparison of the proposed RTE with related work

Framework	Target Hardware Architecture	Programming Model	Scheduling	Fault Tolerance
Spark [Zah+12]	computer clusters	functional-style	online job scheduler	based on checkpoints
Stratosphere [Ale+14]	computer clusters	multiple models with different levels of abstraction	online job scheduler	based on checkpoints
Thrill [Bin+16]	computer clusters	functional-style	online, balances work equally	no fault tolerance
TensorFlow [Aba+15; Aba+16]	computer clusters	high-level, declarative	offline node placement, otherwise online	checkpoints during training
TensorFlow Lite [Goo]	mobile devices, embedded systems	none (executes trained models from files)	online	no fault tolerance
DARTS [SZG13]	multicore and manycore systems	object-oriented	hierarchic online scheduling	no fault tolerance
Concurrent Collections [Bud+10]	multicore and manycore systems	coordination language	online (implementation-dependent)	no fault tolerance
TeamPlay Toolchain [Roe+20]	embedded systems	coordination language	offline (currently) or online (planned)	planned
Teradevice Dataflow Redundancy [Wei+16]	manycore systems	based on coarse-grained dataflow threads	hierarchic online scheduling	redundant execution with hardware-based fault detection
DFER [Alv+14]	multicore and manycore systems	based on TALM [Mar+10]	not specified	redundant execution and commit nodes
SEL & TIC [Jaf+05]	cluster and grid architectures	based on Athapascan [Gal+98]	online	based on logging and checkpoints
proposed RTE	(embedded) multicore and manycore systems	functional-style	offline or online	adaptive redundancy

11

Summary and Future Work

11.1 Summary

This thesis presented a runtime environment (RTE) that is suitable for use in complex embedded systems which utilize multi- and manycore processors. The RTE consists of a functional-style programming model and a dataflow execution model based on directed acyclic graphs. The programming model is similar to commonly used parallel computing frameworks, especially Apache Spark. Its main data structure is a partitioned collection of data elements called RAPID. To construct and process RAPIDs, the programming model provides a set of RAPID operations. From a user's perspective, RAPID operations are lazily evaluated high-order functions. By passing suitable RAPID functions to them, users can specify how the data is processed.

Internally, RAPID operations build a directed acyclic dataflow graph which consists of data nodes and actor nodes. After a graph was built, it can be executed arbitrarily often. Alternatively, it is possible to export the graph in an extended DOT graph description format. The graph can then be imported on a different system. Since the DOT format is human-readable, it is also possible to specify graphs directly in this format. A special property of graphs in the proposed RTE is that they can be divided into sections. This allows the RTE to change the redundancy configuration of parts of the graph individually. The division into sections is specified by the user,

either by calling special checkpoint functions in a RAPID program or by defining them directly in the DOT representation of the graph.

One of the requirements of the execution model was that it is well-suited for various hardware architectures. Consequently, this thesis described possible RTE implementations for two hardware architectures, a shared-memory and a network-on-chip-based (NoC-based) architecture. On the latter, local memories are too small for a graph or schedule. Only a special driver tile has direct access to a large off-chip memory which contains all graphs and schedules. Since only the driver has global knowledge, its main task is to coordinate transfers and actor executions on the compute tiles. However, if an actor requires a large amount of memory, it must be executed on the driver. On the shared-memory architecture, all cores have access to graphs and schedules, and thus there is no global coordination required. Each core is able to check whether all required partitions for its next actor are available.

Scheduling is very similar on the two architectures. The only major difference is that a scheduler must consider the driver tile and map all actors which process large amounts of data always onto this tile. This thesis presented two commonly used scheduling techniques that were modified to support graph sections with different redundancy on both architectures. The first scheduler is an extended version of the HEFT scheduling heuristic which computes an actor mapping and order before the graph is executed. The second scheduler is based on work stealing and determines the actor mapping and order at runtime. In order to handle redundant actors correctly, the scheduler has to deviate from the work stealing concept in some cases.

Graph executions according to fixed schedules can be predicted with reasonable effort. However, the analysis of a graph execution gets more costly the more details are considered. While the consideration of data and message transfers is mandatory for an accurate analysis on the NoC-based architecture, the number of redundancy-related events (for example redundancy changes or actor re-executions) considered in the analysis can be chosen depending on the application requirements.

To determine the performance of the proposed execution model on different architectures, the RTE was implemented on a standard x86 shared-memory system and the Kalray Bostan platform. In addition, three benchmark applications were implemented in the RAPID programming model to evaluate different graph properties and RTE configurations. These applications are Cannon's algorithm, fast Fourier transform and bitonic sort. The results show that the influence of different graph properties and RTE settings varies. For the three applications, the partitioning that was chosen for the input data had a much greater impact than the number of graph section. Using online scheduling was beneficial in most cases. Only for the FFT benchmark on the Kalray architecture the performance was lower. Regarding redundancy, graph executions often take less than twice (or three times) the baseline execution time because only actual computation is executed redundantly.

11.2 Future Work

Although the presented RTE supports different architectures and offers a variety of options to execute graphs with different redundancy, there is room for further development. An important aspect that deserves a closer look is redundancy. This thesis only described the basics of the adaptive redundancy concept, i.e. how the redundancy configuration at runtime can be changed while maintaining an appropriate predictability. How the RTE could make a decision to change the redundancy configuration is still an open question, especially since the supported redundancy configurations are two-dimensional (execution on same vs. different processing elements and double vs. triple execution) and can be adjusted in different ways. Furthermore, RTE management code is currently executed without redundancy. One possible extension would be to utilize hardware redundancy to increase the fault tolerance of RTE-internal routines. On NoC-based architectures, one lockstep processor per tile would be sufficient for this.

Besides transient faults which can be corrected by either voting or re-executing actors, there are also permanent faults. This thesis briefly discussed possible ways to react to a failing hardware component on a theoretical level, but the actual RTE implementation does not support graceful degradation at the moment. The addition of different mechanisms to handle failing hardware units could further increase the versatility of the proposed RTE.

Another possible extension of the RTE would be the ability to determine graph sections automatically instead of the currently required manual division. However, an entirely automatic sectioning would reduce the flexibility. Users should be able to specify the adaptive redundancy requirements of an application, so the sectioning routine can determine the number of sections and section bounds accordingly.

Additional multi and manycore architectures may also be worth studying. The evaluation in Chapter 9 showed that the I/O tiles are possibly a limiting factor for graph executions. Thus, the successor to the Kalray Bostan MPPA, the Kalray Coolidge MPPA [Din19], is an interesting architecture for the dataflow-based execution model. This processor consists of five tiles connected through a NoC. The tiles themselfs are similar to the compute tiles of the Kalray Bostan MPPA. However, a major difference between the Coolidge and its predecessor is that the former does not contain I/O tiles. Instead, the five tiles can access the DDR memory via an AXI fabric. If the bandwidth and latency of the AXI fabric are appropriate, an RTE implementation on this architecture has the potential to utilize the Kalray Coolidge MPPA more efficiently than the Bostan MPPA. Furthermore, an RTE for this architecture could overcome the disadvantages of large partitions which are present in the RTE on its predecessor.

11.3 Conclusion

This thesis presented a concept to leverage multi and manycore processors in embedded systems. The proposed RTE uses a dataflow-based execution model which allows to change the redundancy of parts of the application at runtime. By using fixed schedules and restricting redundancy changes to certain points during runtime, dataflow executions can be analyzed with reasonable effort. Regarding performance, experimental shows that the proposed RTE is on par with existing parallel computation frameworks that focus on high performance and thus provide no redundancy-related features. At the moment, the RTE's redundancy capabilities only cover the execution of dataflow actors. In a practical setting, the RTE internals should be fault-tolerant as well. A possible way to achieve this with only minimal modification of the RTE is the use of hardware redundancy. Furthermore, with the addition of a mechanism for determining at runtime when the redundancy configuration should be changed, the RTE can fully utilize the adaptive redundancy concept.

Bibliography

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. "SipHash: A Fast Short-Input PRF". In: *Progress in Cryptology - INDOCRYPT 2012*. Springer Berlin Heidelberg, 2012, pp. 489–508. ISBN: 978-3-642-34931-7. DOI: 10.1007/978-3-642-34931-7_28.

[Aba+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015. URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

[Aba+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[ABU91] Arvind, Lubomir Bic, and Theo Ungerer. "Evolution of Data-Flow Computers". In: *Advanced Topics in Data-Flow Computing*. Prentice Hall, 1991, pp. 3–33. ISBN: 978-0-13-006503-2.

[ACD74] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. "A Comparison of List Schedules for Parallel Processing Systems". In: *Communications of the ACM* 17.12 (Dec. 1974), pp. 685–690. ISSN: 0001-0782. DOI: 10.1145/361604.361619.

[Ake+20] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. "An Empirical Survey-based Study into Industry Practice in Real-time Systems". In: *IEEE Real-Time Systems Symposium (RTSS)*. 2020, pp. 3–11. DOI: 10.1109/RTSS49844.2020.00012.

[Ale+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. "The Stratosphere Platform for Big Data Analytics". In: *The VLDB Journal* 23.6 (Dec. 2014), pp. 939–964. ISSN: 1066-8888. DOI: 10.1007/s00778-014-0357-y.

[Alv+14] Tiago A. O. Alves, Sandip Kundu, Leandro A. J. Marzulo, and Felipe M. G. França. "Online Error Detection and Recovery in Dataflow Execution". In: *20th International On-Line Testing Symposium (IOLTS)*. IEEE Computer Society, July 2014, pp. 99–104. DOI: 10.1109/IOLTS.2014.6873679.

[AN90] Arvind and Rishiyur S. Nikhil. "Executing a program on the MIT tagged-token dataflow architecture". In: *IEEE Transactions on Computers* 39.3 (Mar. 1990), pp. 300–318. DOI: 10.1109/12.48862.

[Avi76] Algirdas Avižiens. "Fault-Tolerant Systems". In: *IEEE Transactions on Computers* 25.12 (Dec. 1976), pp. 1304–1312. ISSN: 1557-9956. DOI: 10.1109/TC.1976.1674598.

[Bat68] Kenneth E. Batcher. "Sorting networks and their applications". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). ACM Press, 1968, pp. 307–314. ISBN: 978-1-4503-7897-0. DOI: 10.1145/1468075.1468121.

[Bau05] Robert C. Baumann. "Radiation-induced soft errors in advanced semiconductor technologies". In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316. DOI: 10.1109/TDMR.2005.853449.

[Bec+15] Matthias Becker, Dakshina Dasari, Vincent Nélis, Moris Behnam, Luís Miguel Pinho, and Thomas Nolte. "Investigation on AUTOSAR-Compliant Solutions for Many-Core Architectures". In: *Euromicro*

Conference on Digital System Design. 2015, pp. 95–103. DOI: 10.1109/DSD.2015.63.

[Bin+16] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-performance algorithmic distributed batch data processing with C++". In: *IEEE International Conference on Big Data (Big Data)*. Dec. 2016, pp. 172–183. DOI: 10.1109/BigData.2016.7840603.

[BL99] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *Journal of the ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234.

[Blu+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *ACM SIGPLAN Notices* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340. DOI: 10.1145/209937.209958.

[Bud+10] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. "Concurrent Collections". In: *Scientific Programming* 18.3–4 (Aug. 2010), pp. 203–217. DOI: 10.3233/SPR-2011-0305.

[Can69] Lynn Elliot Cannon. "A Cellular Computer to Implement the Kalman Filter Algorithm". PhD thesis. USA: Montana State University, 1969.

[CH00] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ACM SIGPLAN Notices* 35.9 (Sept. 2000), pp. 268–279. ISSN: 0362-1340. DOI: 10.1145/357766.351266.

[Cha+94] Po-Yung Chang, E. Hao, Tse-Yu Yeh, and Yale Patt. "Branch classification: a new mechanism for improving branch predictor performance". In: *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*. 1994, pp. 22–31. DOI: 10.1109/MICRO.1994.717404.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Vol. 6. Annals of Mathematics Studies. Princeton, New Jersy, USA: Princeton University Press, 1941.

[Con02] Cristian Constantinescu. "Impact of deep submicron technology on dependability of VLSI circuits". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, pp. 205–209. DOI: 10.1109/DSN.2002.1028901.

[CSH19] Louis-Claude Canon, Mohamad El Sayah, and Pierre-Cyrille Héam. "A Comparison of Random Task Graph Generation Methods for Scheduling Problems". In: *Euro-Par 2019: Parallel Processing*. Springer International Publishing, 2019, pp. 61–73. ISBN: 978-3-030-29400-7. DOI: 10.1007/978-3-030-29400-7_5.

[CT65] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19 (1965), pp. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1.

[Cul+91] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 164–175. ISBN: 0-89791-380-9. DOI: 10.1145/106972.106990.

[CW85] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". In: *ACM Computing Surveys* 17.4 (Dec. 1985), pp. 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042.

[Dav78] Alan L. Davis. "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine". In: *Proceedings of the 5th Annual Symposium on Computer Architecture*. ISCA '78. New York, NY, USA: Association for Computing Machinery, 1978, pp. 210–215. ISBN: 978-1-4503-7400-2. DOI: 10.1145/800094.803050.

[Den74] Jack B. Dennis. "First version of a data flow procedure language". In: *Programming Symposium*. Springer Berlin Heidelberg, 1974, pp. 362–376. ISBN: 3-540-06859-7. DOI: 10.1007/3-540-06859-7_145.

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[Din+14] Benoît D. de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. "Time-critical computing on a single-chip massively parallel processor". In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.110.

[Din19] Benoît Dupont de Dinechin. "Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore Processor". In: *Proceedings of the 56th Annual Design Automation Conference*. DAC '19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3323473.

[DM74] Jack B. Dennis and David P. Misunas. "A Preliminary Architecture for a Basic Data-Flow Processor". In: *Proceedings of the 2nd Annual Symposium on Computer Architecture*. ISCA '74. New York, NY, USA: Association for Computing Machinery, Dec. 1974, pp. 126–132. ISBN: 978-1-4503-7366-1. DOI: 10.1145/642089.642111.

[Gal+98] François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mathias Doreille. "Athapascan-1: On-Line Building Data Flow Graph in a Parallel Language". In: *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. PACT '98. USA: IEEE Computer Society, 1998, pp. 88–95. DOI: 10.1109/PACT.1998.727176.

[Gat+09] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. "Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience". In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1414–1425. ISSN: 2150-8097. DOI: 10.14778/1687553.1687568.

[GBP07] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. "KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors". In: *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. PASCO '07. London, Ontario, Canada: Association for Computing Machinery, 2007, pp. 15–23. ISBN: 978-1-59593-741-4. DOI: 10.1145/1278177.1278182.

[GC18] André Göbel and Denis Claraz. "A Multi-Core Basic Software as Key Enabler of Application Software Distribution". In: *ERTS 2018*. 9th European Congress on Embedded Real Time Software and Systems. Toulouse, France, Jan. 2018. URL: https://hal.archives-ouvertes.fr/hal-02156255.

[GKN15] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. Jan. 2015. URL: https://www.graphviz.org/pdf/dotguide.pdf (visited on 01/07/2020).

[GKW85] John R. Gurd, Chris C. Kirkham, and Ian Watson. "The Manchester Prototype Dataflow Computer". In: *Communications of the ACM* 28.1 (Jan. 1985), pp. 34–52. ISSN: 0001-0782. DOI: 10.1145/2465.2468.

[Goo] Google LLC. *TensorFlow Lite*. URL: https://www.tensorflow.org/lite/ (visited on 08/11/2020).

[Gor+78] Michael Gordon, Robin Milner, L. Morris, Malcolm Newey, and Christopher Wadsworth. "A Metalanguage for Interactive Proof in LCF". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: Association for Computing Machinery, 1978, pp. 119–130. ISBN: 978-1-4503-7348-7. DOI: 10.1145/512760.512773.

[GRS89] Elvira Glück-Hiltrop, Matthias Ramlow, and Ute Schürfeld. "The sto//mann data flow machine". In: *Parallel Architectures and Languages Europe*. Ed. by Eddy Odijk, Martin Rem, and Jean-Claude Syre. Springer Berlin Heidelberg, 1989, pp. 433–457. ISBN: 978-3-540-46183-8. DOI: 10.1007/3540512845_55.

[Hin69] Roger Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 0002-9947. DOI: 10.2307/1995158.

[Hud+07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. "A History of Haskell: Being Lazy with Class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, pp. 12-1–12-55. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856.

[Hud89] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM Computing Surveys* 21.3 (Sept. 1989), pp. 359–411. ISSN: 0360-0300. DOI: 10.1145/72551.72554.

[Hug90] John Hughes. "Why Functional Programming Matters". In: *Research Topics in Functional Programming*. USA: Addison-Wesley Longman Publishing Co., Inc., 1990, pp. 17–42. ISBN: 0-201-17236-4.

[Jaf+05] Samir Jafar, Thierry Gautier, Axel Krings, and Jean-Louis Roch. "A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing". In: *Euro-Par 2005 Parallel Processing*. Springer Berlin Heidelberg, 2005, pp. 675–684. ISBN: 978-3-540-31925-2. DOI: 10.1007/11549468_74.

[Jag+12] Srikanth Jagannathan, Zachary Diggins, Nihaar Mahatme, Thomas D. Loveless, Bharat L. Bhuva, Shi-Jie Wen, Richard Wong, and Lloyd W. Massengill. "Temperature dependence of soft error rate in flip-flop

designs". In: *IEEE International Reliability Physics Symposium (IRPS)*. 2012, SE.2.1–SE.2.6. DOI: 10.1109/IRPS.2012.6241927.

[JL01] Daniel A. Jiménez and Calvin Lin. "Dynamic branch prediction with perceptrons". In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 2001, pp. 197–206. DOI: 10.1109/HPCA.2001.903263.

[KA98] Yu-Kwong Kwok and Ishfaq Ahmad. "Benchmarking the task graph scheduling algorithms". In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. 1998, pp. 531–537. ISBN: 0-8186-8404-6. DOI: 10.1109/IPPS.1998.669967.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors". In: *ACM Computing Surveys* 31.4 (Dec. 1999), pp. 406–471. ISSN: 0360-0300. DOI: 10.1145/344588.344618.

[Kah74] Gilles Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Information Processing 74, Proceedings of the 6th IFIP Congress*. Stockholm, Sweden: North-Holland, 1974, pp. 471–475.

[KK07] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 978-0-12-088525-1.

[KM66] Richard M. Karp and Raymond E. Miller. "Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing". In: *SIAM Journal on Applied Mathematics* 14.6 (1966), pp. 1390–1411. ISSN: 0036-1399. DOI: 10.1137/0114108.

[Kos73] Paul R. Kosinski. "A Data Flow Language for Operating Systems Programming". In: *Proceeding of ACM SIGPLAN - SIGOPS Interface Meeting on Programming Languages - Operating Systems*. New York, NY, USA: Association for Computing Machinery, 1973, pp. 89–94. ISBN: 978-1-4503-7379-1. DOI: 10.1145/800021.808289.

[KSY92] Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi. "A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation". In: *Future Generation Computer Systems* 7.2 (1992), pp. 199–209. ISSN: 0167-739X. DOI: 10.1016/0167–739X(92)90007–X.

[LB00] Jae-Dong Lee and Kenneth E. Batcher. "Minimizing Communication in the Bitonic Sort". In: *IEEE Transactions on Parallel and Distributed Systems* 11.5 (May 2000), pp. 459–474. ISSN: 2161-9883. DOI: 10.1109/71.852399.

[Lee+19] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. *On-Device Neural Net Inference with Mobile GPUs*. 2019. DOI: 10.48550/ARXIV.1907.01989.

[Lu+08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 329–339. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346323.

[Luc+16] Andre Luckow, Matthew Cook, Nathan Ashcraft, Edwin Weill, Emil Djerekarov, and Bennie Vorster. "Deep learning in the automotive industry: Applications and tools". In: *IEEE International Conference on Big Data (Big Data)*. 2016, pp. 3759–3768. DOI: 10.1109/BigData.2016.7841045.

[Mar+10] Leandro A. J. Marzulo, Tiago A. O. Alves, Felipe M. G. França, and Vítor S. Costa. "TALM: A Hybrid Execution Model with Distributed Speculation Support". In: *22nd International Symposium on Computer Architecture and High Performance Computing Workshops*. Oct. 2010, pp. 31–36. ISBN: 978-0-7695-4276-8. DOI: 10.1109/SBAC-PADW.2010.8.

[Mil78] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4.

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 978-0-08-055832-5.

[NPA92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. "T: A Multithreaded Massively Parallel Architecture". In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA '92. Queensland, Australia: Association for Computing Machinery, 1992, pp. 156–167. ISBN: 0-89791-509-7. DOI: 10.1145/139669.139715.

[PC90] Gregory M. Papadopoulos and David E. Culler. "Monsoon: An Explicit Token-Store Architecture". In: *SIGARCH Computer Architecture News* 18.2SI (1990), pp. 82–91. ISSN: 0163-5964. DOI: 10.1145/325096.325117.

[PT00] Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100.

[PT91] Gregory M. Papadopoulos and Kenneth R. Traub. "Multithreading: A Revisionist View of Dataflow Architectures". In: *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ISCA '91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, pp. 342–351. ISBN: 0-89791-394-9. DOI: 10.1145/115952.115986.

[PW93] Simon L. Peyton Jones and Philip Wadler. "Imperative Functional Programming". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 71–84. ISBN: 0-89791-560-7. DOI: 10.1145/158511.158524.

[Rob11] Arch D. Robison. "Intel® Threading Building Blocks (TBB)". In: *Encyclopedia of Parallel Computing*. Boston, MA, USA: Springer US, 2011, pp. 955–964. ISBN: 978-0-387-09766-4. DOI: 10.1007/978–0–387–09766–4_51.

[Roe+20] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. "Towards Energy-, Time- and Security-Aware Multi-core Coordination". In: *Coordination Models and Languages*. Springer International Publishing, 2020, pp. 57–74. ISBN: 978-3-030-50029-0. DOI: 10.1007/978–3–030–50029–0_4.

[RŠU00] Borut Robič, Jurij Šilc, and Theo Ungerer. "Beyond Dataflow". In: *Journal of Computing and Information Technology* 8.2 (June 2000), pp. 89–101. DOI: 10.2498/cit.2000.02.01.

[Sah+15] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1357–1369. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742790.

[Sai+15] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît D. de Dinechin. "The shift to multicores in real-time and safety-critical systems". In: *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Oct. 2015, pp. 220–229. DOI: 10.1109/CODESISSS.2015.7331385.

[Set76] Ravi Sethi. "Scheduling Graphs on Two Processors". In: *SIAM Journal on Computing* 5.1 (Mar. 1976), pp. 73–82. DOI: 10.1137/0205005.

[Shv+10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System". In: *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-65697-3.

[SM06] André Seznec and Pierre Michaud. "A case for (partially) TAgged GEometric history length branch prediction". In: *Journal of Instruction-level Parallelism - JILP* 8 (Jan. 2006).

[Sor09] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009. ISBN: 978-1-59829-954-0. DOI: 10.2200/S00192ED1V01Y200904CAC005.

[Spe+09] Michael Sperber, R. kent Dybvig, Matthew Flatt, Anton Van straaten, Robby Findler, and Jacob Matthews. "Revised^6 Report on the Algorithmic Language Scheme". In: *Journal of Functional Programming* 19.S1 (Aug. 2009), pp. 1–301. ISSN: 0956-7968. DOI: 10.1017/S0956796809990074.

[SQZ93] Jocelyn Sérot, Georges Quénot, and Bertrand Zavidovique. "Functional programming on a dataflow architecture: Applications in real-time image processing". In: *Machine Vision and Applications* 7.1 (Dec. 1993), pp. 44–56. ISSN: 1432-1769. DOI: 10.1007/BF01212416.

[ŠRU98] Jurij Šilc, Borut Robič, and Theo Ungerer. "Asynchrony in Parallel Computing: From Dataflow to Multithreading". In: *Journal of Parallel and Distributed Computing Practices* 1 (1998), pp. 1–33.

[SS95] James E. Smith and Gurindar S. Sohi. "The microarchitecture of superscalar processors". In: *Proceedings of the IEEE* 83.12 (1995), pp. 1609–1624. DOI: 10.1109/5.476078.

[Ste82] Guy L. Steele. "An Overview of COMMON LISP". In: *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*. LFP '82. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1982, pp. 98–107. ISBN: 0-89791-082-6. DOI: 10.1145/800068.802140.

[Str00] Christopher Strachey. "Fundamental Concepts in Programming Languages". In: *Higher-Order and Symbolic Computation* 13.1–2 (Apr. 2000), pp. 11–49. ISSN: 1388-3690. DOI: 10.1023/A:1010000313106.

[SZG13] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. "An Implementation of the Codelet Model". In: *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg, 2013, pp. 633–644. ISBN: 978-3-642-40047-6. DOI: 10.1007/978-3-642-40047-6_63.

[Tak+83] Eiji Takeda, Yoshinobu Nakagome, Hitoshi Kume, and Shojiro Asai. "New hot-carrier injection and device degradation in submicron MOSFETs". In: *IEE Proceedings I - Solid-State and Electron Devices* 130.3 (1983), pp. 144–150. DOI: 10.1049/ip-i-1.1983.0026.

[Tal+20] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev. "Compute Solution for Tesla's Full Self-Driving Computer". In: *IEEE Micro* 40.2 (2020), pp. 25–35. DOI: 10.1109/MM.2020.2975764.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274. DOI: 10.1109/71.993206.

[Tur85] David A. Turner. "Miranda: A Non-Strict Functional Language with Polymorphic Types". In: *Functional Programming Languages and Computer Architecture*. Nancy, France: Springer Berlin Heidelberg, 1985, pp. 1–16. ISBN: 978-3-540-39677-2. DOI: 10.1007/3-540-15975-4_26.

[Ull75] Jeffrey David Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80008-0.

[Ung88] Theo Ungerer. "ASTOR - An architecture of special purpose processing units with distributed control and message passing". In: *Microprocessing and Microprogramming* 23.1 (1988), pp. 227–232. ISSN: 0165-6074. DOI: 10.1016/0165-6074(88)90360-2.

[UO17] M. Urbina and R. Obermaisser. "Efficient Multi-core AUTOSAR-Platform Based on an Input/Output Gateway Core". In: *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2017, pp. 157–166. DOI: 10.1109/PDP.2017.85.

[VBP19] Hrvoje Vdovic, Jurica Babic, and Vedran Podobnik. "Automotive Software in Connected and Autonomous Electric Vehicles: A Review". In: *IEEE Access* 7 (2019), pp. 166365–166379. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2953568.

[VF85] Rex Vedder and Dennis Finn. "The Hughes Data Flow Multiprocessor: Architecture for Efficient Signal and Data Processing". In: *SIGARCH Computer Architecture News* 13.3 (June 1985), pp. 324–332. ISSN: 0163-5964. DOI: 10.1145/327070.327290.

[Wan+10] Wenping Wang, Shengqi Yang, Sarvesh Bhardwaj, Sarma Vrudhula, Frank Liu, and Yu Cao. "The Impact of NBTI Effect on Combinational Circuit: Modeling, Simulation, and Analysis". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18.2 (2010), pp. 173–183. DOI: 10.1109/TVLSI.2008.2008810.

[Wei+16] Sebastian Weis, Arne Garbade, Bernhard Fechner, Avi Mendelson, Roberto Giorgi, and Theo Ungerer. "Architectural Support for Fault Tolerance in a Teradevice Dataflow System". In: *International Journal of Parallel Programming* 44.2 (Apr. 2016), pp. 208–232. ISSN: 1573-7640. DOI: 10.1007/s10766-014-0312-y.

[Wik] WikiChip. *Zen 2-Microarchitectures-AMD*. URL: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2 (visited on 05/15/2020).

[Yaz+14] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. "Hybrid Dataflow/von-Neumann Architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2014), pp. 1489–1509. DOI: 10.1109/TPDS.2013.125.

[YG94] Tao Yang and Apostolos Gerasoulis. "DSC: scheduling parallel tasks on an unbounded number of processors". In: *IEEE Transactions on Parallel and Distributed Systems* 5.9 (Sept. 1994), pp. 951–967. DOI: 10.1109/71.308533.

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI '12. San Jose, CA, USA: USENIX Association, 2012, pp. 15–28.

[ZS06] Henan Zhao and Rizos Sakellariou. "Scheduling multiple DAGs onto heterogeneous systems". In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. IPDPS'06. IEEE Computer Society, 2006. ISBN: 1-4244-0054-6. DOI: 10.1109/IPDPS.2006.1639387.

List of Figures

List of Tables

List of Code Listings

List of Algorithms