Speculative Execution of Data Flow Process Networks

Stefan Willenbrock

of the Department of Computer Science

August 1, 2012

Technische Universität Kaiserslautern

First Reviewer: Prof. Dr. Klaus Schneider
Second Reviewer: Dipl.-Technoinform. Daniel Baudisch

Dies ist eine leicht abgewandelte Version der originalen Masterarbeit zur Veröffentlichung auf den Seiten der Embedded Systems Group der Technischen Universität Kaiserslautern.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit mit dem Thema

Speculative Execution of Data Flow Process Networks

selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Kaiserslautern, den 1. August 2012

Stefan Willenbrock

Danksagung

Ganz besonders Bedanken möchte ich mich bei Dipl.-Technoinform. Daniel Baudisch für seine ausdauernde Unterstützung bei der Erstellung meiner Masterarbeit. Besonders für die Unterstützung bei der F# Programmierung möchte ich ihm danken.
Meiner Familie danke ich auch für Ihren Rückhalt und die immer währende Unterstützung während meines Studiums der Informatik.

Abstract

Data-flow process networks (DPNs) are a powerful model of computation (MOC) with inherent concurrency. Process networks are nets of nodes that work independently of each other only triggered by the availability of data on their input streams.

This (MOC) with its parallel characteristics and well defined data-dependencies is meant for the description of highly concurrent software. With the shift from sequential to parallel processing a few years ago the demand for efficient usage of multiprocessor systems has increased.

In related work from Baudisch et al. [BBS12] it has been shown that the out-of-order scheduling from processors applied to DPNs could also increase the throughput of applications described by data-flow graphs. In this work I evaluate the suitability of another approach for more parallelism applied in DPN, that is known from the field of multiprocessors, speculative execution.

Zusammenfassung

Datenflussnetzwerke (DPNs) sind ein ausdrucksfähiges Berechnungsmodell mit innewohnender Nebenläufigkeit. DPNs sind Netze aus Knoten welche unabhängig voneinander arbeiten und nur durch die Verfügbarkeit von Daten auf ihren Eingangs- und Ausgangströmen aktiviert werden.

Dieses Berechnungsmodell mit seinen nebenläufigen Eigenschaften und genau bestimmten Datenabhängigkeiten ist besonders geeignet für de Beschreibung von stark parallelisierte Software. Mit dem Wechsel von sequentieller zu paralleler Verarbeitung vor einigen Jahre haben sich die Vorderungen nach effizienter Ausnutzung von Multiprozessorsystemen noch erhöht.

Mit der zugehörigen Arbeit von Baudisch et al. [BBS12] wurde gezeigt, dass der aus Prozessoren bekannte Ansatz der out-of-order Ausführung angewendet auf DPNs auch den Durchsatz von durch DPNs beschriebenen Anwendungen steigern kann. In dieser Arbeit evaluiere ich die Verwendbarkeit eines anderen Ansatzes angewendet auf DPN, bekannt aus Multiprozessorsystemen, die spekulative Ausführung.

Contents

1 Introduction **1**
- 1.1 Shift in computing to multi-core systems 1
- 1.2 Shipments of multi-core processors 1
- 1.3 Scalability of multithreaded architectures 2
  - 1.3.1 Amdahl's law 2
- 1.4 Introduction to Data Flow Process Networks 2

2 Fundamentals **4**
- 2.1 Data Flow Process Networks 4
  - 2.1.1 DPN in software modelling 4
- 2.2 Synchronous Data Flow Graphs 5
  - 2.2.1 Synchronous Model 5
  - 2.2.2 Atomic and Large Grain SDFG 5
  - 2.2.3 Homogeneous SDF graphs 6
- 2.3 Terminology 6
- 2.4 Speculative Out-Of-Order SDFG Execution 6
  - 2.4.1 Static vs. Dynamic Scheduling 7
  - 2.4.2 Blocked HSDFG Scheduling 8
  - 2.4.3 Out-Of-Order HSDFG Execution 8
  - 2.4.4 Speculative HSDFG Execution 10
- 2.5 Averest Framework 10
- 2.6 Quartz 10
- 2.7 Weak Memory Model 12
  - 2.7.1 Cache and Memory Interaction 13
  - 2.7.2 MESI Cache Coherence Protocol 13
  - 2.7.3 Introduction of Memory Barriers 14

3 Related Work **17**
- 3.1 Out-Of-Order Execution Approaches 17
  - 3.1.1 Data Scalar Architectures 17
- 3.2 Compiler Based Speculation Approaches 18
  - 3.2.1 Speculative Execution of Alternative Program Paths . . . 18
  - 3.2.2 Program Dependence Graph Based Speculation 19
- 3.3 Parallelism Enhancing Hardware Architectures 19
  - 3.3.1 Simultaneous Multi-Threading 20
- 3.4 Speculative Hardware Architectures 20
  - 3.4.1 The Trace Processor 20
  - 3.4.2 The Microprocessor Architecture for Java™ 21

3.4.3 Reference Idempotency Analysis . . . 22
3.5 Task Based Scheduling approaches . . . 22
3.5.1 The StarSS programming paradigm . . . 22
3.5.2 Intel® TBB . . . 23
3.5.3 Cilk . . . 24
3.5.4 Work-Stealing Deques . . . 24

4 Implementation **25**
4.1 Implementation Goals . . . 25
4.2 Conceptual Overview . . . 26
4.3 Speculative Execution Design . . . 27
4.3.1 Central Buffer Station . . . 28
4.3.2 Task Queue . . . 29
4.3.3 Worker Threads . . . 29
4.3.4 Implementation Requirements . . . 30
4.4 Detailed Implementation Description . . . 31
4.4.1 Central Buffer Station . . . 31
4.4.2 The Weak Memory Model and Atomicity . . . 31
4.4.3 The Synthesized Task Functions and Task Data . . . 34
4.5 Worker Threads . . . 39
4.5.1 Task Token Selection . . . 40
4.5.2 CBS Head Removal . . . 41
4.5.3 Task Execution . . . 42
4.5.4 Updating Data Input Dependencies . . . 42
4.5.5 Updating Data Output Dependencies . . . 43
4.5.6 The Speculation of Self Scheduled Tokens . . . 44
4.6 Implementation Discussion . . . 44
4.6.1 Thread-Based and Task-Based Scheduling . . . 45
4.6.2 Global and Worker Based Scheduling . . . 46
4.6.3 Centralized Buffer Storage for FIFO Buffers . . . 48

5 Benchmarks **50**
5.1 The Benchmark Environment . . . 50
5.2 The Measuring Approach . . . 50
5.3 The Matrix multiplication benchmarks . . . 52
5.4 The Pitchshift benchmarks . . . 55
5.5 Speedup . . . 61
5.6 Processor Core Usage . . . 63

6 Conclusion **64**

7 Future Work **67**

Chapter 1

Introduction

1.1 Shift in computing to multi-core systems

The demanding requirements of today's computing that appear anywhere from embedded systems to the large data centers empowering the *so-called* cloud computing make it necessary that the present computing power meet these demands.

One way to meet these demands would have been to scale up the processing power of the underlying processor architecture. But there exists practical evidence today, that there is a limit to that approach. The limit of ever increasing clock-speeds was reached during the development of Intel's NetBurst architecture [SGC01].

One of the main design decisions had been to lay out the architecture for very high clock speeds. This design principle made it necessary to increase the pipeline depth considerably. More pipeline stages have the effect of lowering the instructions per cycle count (IPC) [SGC01]. This performance decrease per cycle was intended to be caught up by the much higher CPU clock.

Even though more processors of the NetBurst architecture had been scheduled by Intel, early in 2004 [Fly04] canceled these new processors for new dual-core processors. Industry experts assumed that these problems had been due to power and thermal constraints [Fly04].

This marked a change, from where on simply executing the same sequential software did not lead to a compulsory speed up. Software needed to be designed for these new multi-threaded architectures to take advantage of their provided computational power. As this step had already been envisioned years before the discontinuation of the NetBurst architecture, it still marked the point in computer history where no one could hope to longer look away from the discipline of implementing multi-threaded applications.

1.2 Shipments of multi-core processors

In the recent years there have been numerous releases of multiprocessor CPUs, which are used in embedded systems. In figure **??** you see a list of example processor systems that have just doubled their core count in a fairly small time. There exists a trend to a higher core count in parallel processors.

Processor	Release Date
Apple A4 (1-core)	January 2010
Apple A5 (2-core)	March 2011
Apple A5 (1-core)	March 2012
Apple A5X (2-core)	March 2012
Nvidia Tegra (1-core)	February 2008
Nvidia Tegra 2 (1-core)	January 2010
Nvidia Tegra 3 (4-core)	November 2011

Table 1.1: Releases of common processor systems in the recent past.

1.3 Scalability of multithreaded architectures

1.3.1 Amdahl's law

From that point in time there was a shift from increasing CPU clock frequency to higher core per CPU count. Nobody had really proved that this was the answer to today's computing needs. This is even more considerable since back in 1967 Amdahl [Amd67] tried to defend the validity of the single processor (single core) approach. But he already [Amd67] was acknowledging that the problems he stated in his paper justifying his statement would have been maybe overcome in future times.

The most important contribution of Amdahl [Amd67] had done was a formula that could have been derived from his literal descriptions:

$$S = \frac{1}{r_s + \frac{r_p}{n}} \tag{1.1}$$

S is the speedup compared to a sequential program by utilizing n processors.

In that formula Amdahl divided the overall computational load in sequential r_s and parallelable effort r_p. The sequential effort was "data householding" related and according to Amdahl "The nature of this overhead appears to be sequential so that it is unlikely to be amenable to parallel processing techniques" [Amd67].

The conclusion of Amdahl was, that there exists a rather low speedup potential by using multithreaded architechtures. Even with a high processor count the execution time cannot get smaller than the time needed for r_s.

The problem with his view was, that he considered a fixed problem size. Another standpoint would be to assume that multi-threaded processors could do more work at the same time. In this thesis the additional workload is added by using speculation. In case of good speculation results the original task of the processor could even be cut short, thus improving the computation time.

1.4 Introduction to Data Flow Process Networks

Data-flow process networks (DPN) [Lee89] are a powerful model of computation and are closely related to the ideas based on Kahn process networks [Par95].

The simple and very general concept behind DPNs carries at its heart independent and data-driven execution. The main ingredients for high concurrency.

The DPN data model consists of process nodes that fire without any global scheduling only triggered by the neighbors they depend on. Process nodes need for their computation input data. So called data-tokens that are exchanged between nodes by FIFO buffers.

A FIFO buffer is always a one-to-one connection between nodes. The producer and the consumer of a FIFO buffer are unique. In the general form of DPNs the number of tokens that a node produces for a certain buffer is not determined.

DPNs are intended to operate on infinite data streams that are read from and written to the environment. But how this works is not part of the computational model which is concerned with how often a node fires and how many data tokens are produced each time. As an extension to that model one can think that there are designated reader and writer processes, which will in every execution read data (not data tokens) from the environment. This has no impact on the number of produced data tokens, but let's say maybe the data within the tokens is affected.

This infinite interaction with the environment was very familiar with digital signaling processing applications (DSP) but spread later as well into other areas.

Such another area where DPNs are common due to their continuous operations are embedded systems. But more to embedded systems and the application of DPNs therein later.

As the DPN model is very powerful, which means that it is very general, almost no guarantees are given for the execution of it. One aspect is the unboundedness of buffers in general DPNs. The buffer size in the theoretical model is not specified and possibly of infinite size.

For any practical application this is very unfortunate since the analyzability of certain aspects is desirable. For example freedom from unbounded memory and deadlocks should be verifyable by checking the DPN.

This is in fact possible by restricting the DPN model [Lee91] to the more restricted Synchronous Data Flow graphs [LM87].

Chapter 2

Fundamentals

2.1 Data Flow Process Networks (DPN)

2.1.1 DPN usage in software modelling methods

Data Flow Process Networks (DPN) are a common paradigm that is used widely in different modeling approaches not only for digital signaling applications (DSP) but also for embedded systems in general, especially for concurrent implementations [YYWW93].

To give the reader an understanding why it is relevant to work with DPNs in terms of software synthesis for embedded systems we show some of the methods and tools where they are used.

A very known modelling approach is *structured analysis / real time* (SA/RT) that exists in different adaptations, for example the semi-formal approach from [HP88, HP93] and the more formal one by [Har87]. Other similar approaches are ROOM [SGW94] and Octopus [Awa96].

Common to all these methods is the usage of DPNs as a method for describing the requirements of an embedded software system. The data flow graph that is used can be incrementally refined, to consider the more precise aspects of the system.

These methods are often supplemented by *computer-aided software engineering* tools (CASE) that support the modelling, model checking (extended syntactical checking) and simulation of the model. The simulation in case of *Teamwork* [tea95][1] a CASE tool for [HP88, HP93] the simulation is restricted to the creation of a process activation protocol. Other tools like *STATEMATE* [HLN+88], that follows the approach of [Har87], allow full simulation (execution) of the DPN because of the well-defined semantics of the programming constructs.

Similar capabilities are supported by the tool Averest[2], developed by the Embedded Systems Group of the University of Kaiserslautern. Unlike the aforementioned CASE tools Averest doesn't support the modelling process itself,

[1] *Teamwork* of Cadre Technologies Inc. was sold to Sterling Software, which was itself discontinued in 2000. It seems that Teamwork is no longer offered by any company.

[2] `www.averest.org`

instead it expects a specification in the Quartz format, that is translated into an intermediate representation, that can be simulated, verified and synthesized directly or over an transformation step. The DPN that is transformed from the intermediate format is another representation of the specification of the Quartz format. More details about Averest can be found in 2.5.

2.2 Synchronous Data Flow Graphs

2.2.1 The Synchronous Model

In the introduction to DPNs synchronous data flow graphs (SDF) have been mentioned as a restricted DPN model.

One prominent property of SDF graphs and similar restricted DPNs, that make them relevant is consistency. When a DPN is consistent, it never fires so often that the number of tokens produced would need unlimited buffer sizes. This is equivalent to the guarantee of bounded memory. Therefor the buffer are called bounded.

Consistency [Lee91] means that no dead-locks can happen during the execution of a DPN due to empty buffers. A deadlock in a DPN can happen when at some point nodes produce less data tokens than are needed to keep the DPN running. The input buffers of dependent nodes will run empty and the DPN execution halts.

Combined these two aspects result in the analyzable property, that in the long run, a DPN only produces as many tokens as it will consume. This property is the definition of consistency [Lee91].

The restriction within SDF graphs that enables consistency for DPNs [LM87] is that the number of data samples produced or consumed by each node on each invocation is specified.

Every input and output arc of a node has a number, this number specifies the amount of tokens produced or consumed by each node. This number can be specified before runtime and is thus independent of the data.

So far I had omitted one aspect of data flow graphs, delayed arcs. The term delay can be understood as an offset within the arc between producer and consumer node. This means that the token read by the consumer node was not produced within the same iteration of the SDF. Instead it is stored within the buffer on the arc and stems from a previous execution of the producer arc. In case the iteration is the first one, a so-called initial token has to be placed within the buffer.

2.2.2 Atomic and Large Grain Synchronous Data Flow Graphs

In Lee et al. [LM87] interprocessor communication time has been ignored. Process nodes were very simple. The implemented functions were atomic and most likely indivisible operations of a programmable processor.

Since the execution time of the nodes is short, the overhead associated with the communication will have a huge impact on the overall performance of the SDF execution and will probably consume most of the processor time.

A larger granularity would reduce this overhead. The associated paradigm striving for more computational complexity within process nodes are large grained data flow graphs [YYWW93].

2.2.3 Homogeneous SDF graphs

A special kind of SDF graphs are homogeneous SDF graphs (HSDFG). Each node in a HSDFG is only producing a single token on each arc in each firing. For the whole iteration of the SDF this means, that all nodes only have to fire once, before the next iteration begins. This is a simplification that was very helpful implementation-wise.

2.3 Terminology

Process nodes of the DPN execute certain process functions when fired. During the synthesis process 2.5 these process functions are translated from guarded actions to procedural C code. These resulting functions are no longer part of the data flow model of computation with its process nodes and buffers. Therefor, I refer to these synthesized functions as *task functions* or simply *tasks*. These will be the more common term in my thesis but can be seen as analogs to the process nodes and buffers.

When I mention nodes in general, usually computation nodes are referred.

2.4 Scheduling of SDFGs with Out-Of-Order and Speculative Execution

In section 2.2 and 1.4 the theory behind SDFGs has been presented. Everything that is known so far about the execution of SDFGs is its data driven nature, which means that the firing of a node is dependent on the presence of a data token on the node's input edges.

Since the synthesis target is run on top of the *von Neumann* CPU architecture, I cannot rely on architecture given task scheduling like it is part in the data flow architecture. Therefor, the scheduling, i.e. the assignment of task executions onto processor resources (e.g. multiprocessor cores), has to be implemented manually.

Since *out-of-order* and additionally speculative execution will be supported in the synthesis target, these concepts are presented in the following. To have a clear understanding of the benefits each concept brings on his own, I will start with a basic approach to scheduling and add the more advanced concepts later. This happens by utilizing an example SDF graph which is shown in figure 2.1. The effects of the different scheduling mechanisms will be shown by example using a hypothetical dual-core multiprocessor system.

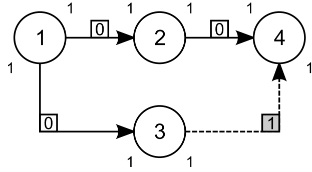

Figure 2.1: A homogeneous SDFG. The dashed edge from node 3 to 4 indicates a delayed dependency.

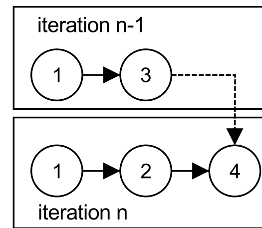

Figure 2.2: The precedence graph for the SDFG in 2.1. Because of delayed dependencies there are tasks of two iterations part of the precedence graph.

In figure 2.1 a SDF graph is shown with an input reader node (1) and an output reader node (4). The solid lines present data-dependencies within the same macrostep (or iteration). The dashed line is an input data-dependency upon the previous macrostep, which is further clarified in figure 2.2. As this graph is a homogeneous SDF graph (HSDFG), which means all nodes produce and consume per firing exactly one token on each arc. This is also indicated by each '1' between the node and the incoming and outgoing side on each arc. The values in the boxes present the buffer size. The buffer sizes are not relevant for our *dynamic* scheduling approach. The reason why the buffer on the delayed edge reserves place for one data-token is that the token from the last execution has to be stored till the next macrostep. This is not the case for immediate edges (within the same macrostep), where the data-token can be consumed immediately, at least in the theoretical model.

The precedence graph in figure 2.2 shows the execution order for the SDFG in figure 2.1. The precedence between the nodes has to be fulfilled by any schedule of the graph. This means, when the input data-tokens should be available for a particular node firing, another node with precedence over that node has to fire first.

2.4.1 Static vs. Dynamic Scheduling

There exist two approaches when it comes to scheduling tasks that are ready for execution. There is the dynamic scheduler, a *runtime supervisor* which decides which nodes can be fired next. The biggest downside is that it requires additional computational effort, and impacts therefor negatively the overall performance of the synthesis target.

The second approach that is available for SDF graphs is static scheduling (already mentioned in 2.2). At compile-time a static schedule for each processing resource (e.g. a multiprocessor core) is determined. The runtime overhead by the additional supervisor disappears that way and any communication between the processing units (e.g. data-exchange) is part of the compiled code.

While a static schedule can handle tasks with fixed execution times it cannot react to changes of the execution time of the same task. A static schedule is always executed in the same way.

A scenario where a task returned faster than another one, whose successor nodes are foreseen to be executed next, would lead to wasted processor cycles. The successor(s) of the faster task cannot be scheduled, because all available processor cores are determined to execute the successors of the slower node. This is how it is determined by the static schedule for each individual core.

Reaction to this runtime dependent behavior, which can be caused by using different hardware platforms with different performance characteristics, can only happen by a dynamic scheduler. In the previous scenario the successor nodes of the faster task would have been scheduled first. Only if any cores would have been available when the slower task has finished, at least partially his successors could have been scheduled immediately as well. Otherwise they would have been postponed to a later point.

In any way, the wait time in between would have been eliminated, which would benefit the overall throughput of the application.

2.4.2 Blocked HSDFG Scheduling

In [LM87] and [LD87] the most basic scheduling mechanism for SDF graphs has been introduced. The work by Messerschmitt et al. [LM87] introduced SDF graphs for the first time. Their scheduling mechanism *blocked scheduling* presents a bottom line of what can be achieved scheduling wise.

The term "blocked" comes from the execution of macrosteps in blocks. These blocks contain all node executions. Before another macrostep or block can be started, all nodes of the previous macrostep have to be executed.

The advantage is that parallel processor cores can be utilized by this scheduling technique. The downside is that the strict precedence order between the task executions leaves cores without workload. Especially at the end of a block the cores run empty. An example of such a blocked schedule can be seen in figure 2.3.

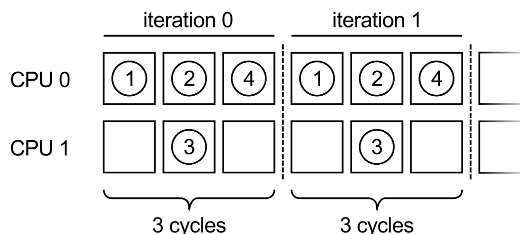

Figure 2.3: Blocked schedule wasting parallel ressources.

2.4.3 Out-Of-Order HSDFG Execution

Out-of-order execution is a concept derived from processors. Most of the time these out-of-order processors implement the Tomasulo algortihm of out-of-order (OOO) execution. To understand the positive effect, first a description of a processor without OOO support should follow.

Because of the introduction of pipelining instruction words are divided in smaller operands, the pipeline stages. Such a pipeline stage is mapped to functional units within the processor. A pipeline stage might depend on data, that

has been produced by other instructions. If such a data dependency between a pipeline stage and a pipeline stage of a predecessor instruction or even the same instruction is detected, the functional unit will halt the execution and wait until the needed data has been produced.

The Tomasulo Algorithm

These waits can be avoided by the Tomasulo algorithm. By the Tomasulo concept a reservation station is added to the processor. All pipeline stages of the instruction stream are loaded into the reservation station. When the data-dependencies by a pipeline stage are fulfilled the functional unit executes the instruction stage. Since these execution might happen in a different order than the instruction stream had intended, the produced data can be ready in a point of time where other functional units expect the availability of data from a previous execution step.

To maintain the data ordering a reorder buffer was added. In random order finished data can be fetched from the reorder buffer and used by the functional units. In that way the computation seems to happen still in order from the outside which is important. Because the waits of the functional units have been minimized the overall performance of the processor increases.

Out-Of-Order in SDF Graphs

Processor instructions, pipeline stages, functional units and the reorder buffer had been translated by Baudisch et al. [BBS12] to process networks.

The whole instruction word can be seen as equivalent to one iteration of the DPN or SDF. This is very natural because every macrostep (iteration) of a SDF needs only one execution of every task within the graph. Exactly like pipeline stages of instruction words.

The out-of-order execution in SDF has the following components:

Task Queue Ready tasks in the SDF are scheduled onto to the task queue. The task queue can be accessed from different execution threads which remove these tasks from the queue and execute them. New tasks that should be executed are inserted by the execution threads when they have finished execution of a task.

Reorder Buffer The reorder buffer stores execution results of tasks, the data that was produced, as long as no task has consumed these values. The buffer has to be able to hold data of the same task of different SDFG iterations.

Worker Thread The worker thread pops tasks from the task queue and executes them. The worker stores the results and then schedules tasks onto the queue that were waiting for the produced data.

An effect of this data-driven execution is that iterations of the SDF might finish before their predecessors have finished. Since the iterations of the SDF read and write data to the environment the ordering of the SDF is important. Therefor those tasks that are responsible for the environment interaction have still to be executed in order.

The core aspect of OOO is that tasks might be executed before their counterparts in a previous macrostep. The reason why this is even possible is that tasks might not consume the same time in every execution step.

2.4.4 Speculative HSDFG Execution

The Out-Of-Order approach showed that it is possible to increase parallelism by executing tasks out-of-order.

Another concept of processors is the speculative execution. In processors the speculation focuses on control-flow-speculation. This means that future instructions are executed before it is clear if the control-flow will even invoke them. This is not a concept we can translate to SDFGs. But the idea of speculations could be translated to multiprocessor systems.

The idea is to speculate over data flows. Since tasks in a SDF already specify their dependencies explicitly it is possible to determine what input data is missing. The concept of SDF data flow speculation will be shown later, we will no further go into the details here.

2.5 The Averest Framework

In my attempt to increase the concurrency of reactive systems I employ the *Averest* framework[3].

Averest is intended to support the development and design flow of reactive systems. The reactive system of choice has to be specified first in the imperative synchronous language called *Quartz* [Sch09]. Using this specification as input Averest supports the developer with following tools:

A Quartz to AIF translator A compiler translating the Quartz specification into the Averest Intermediate Format (AIF)

AIF Programmming Library An API that enables the developer to transform the AIF representation of the design into any target representation. This transformation is called synthesis.

An overview about the design flow is shown in figure 2.4. Because the imperative synchronous language Quartz and AIF have certain properties that are important for translation process into an executable system, I will give in the following a little introduction into why Quartz is used as a foundation. This will give the reader an understanding about the special properties that make Quartz determined to the specification and representation formats for highly concurrent systems.

2.6 The Synchronous Language Quartz

Synchronous languages [BB91, BCE+03, Hal98] are a special model for describing reactive systems. There are numerous reasons synchronous languages are employed over other types of languages. The main reasons are:

[3] Averest is developed by the *Embedded Systems Group* of the University of Kaiserslautern. `http://www.averest.org/`

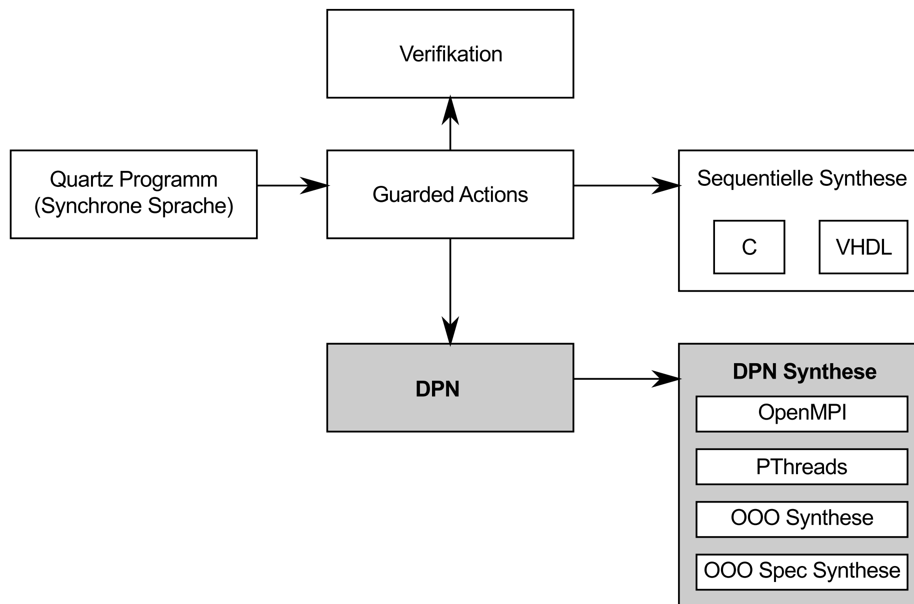

Figure 2.4: The Averest design flow

High-level language: As hardware platforms change from time to time and reactive systems tend to be used for several years to diminish development costs, low-level programming is not appropriate anymore. Common languages like C tend to be too hardware specific and impose a certain portability barrier.

Hierarchical design: The specification can be hierarchically refined which makes it possible to exchange subcomponents when possible. This modularity guarantees that small changes in the design only have local impact.

Concurrency: The specification of concurrent components and their composition is possible. The constraints of the components and their concurrent behavior caused by inter-component signals can be formally described. The actual behavior can be inferred through a fixpoint computation using operational semantics.

Determinism: Determinism is an essential property but can be heavily affected by concurrency. Quartz concurrent components have well-defined and deterministic behavior.

Verification: The language constructs have precise logical and temporal semantics that make it possible for automatic verification tools to prove behavioral and temporal correctness of the system. The temporal correctness makes synchronous languages especially appealing for real-time systems.

The perfect synchrony of Quartz is the most important aspect. Perfect synchrony means that all statements are executed at once. No time is consumed

at all. To partition the time domain into successive execution steps a special pause statement was introduced.

The perfect synchrony makes it necessary to analyze if the statements of an interval can be executed in a way that every variable is assigned exactly once. This way data races and non-determinism are avoided.

The writing of variables is dependent on input values the statements that execute them are called *guarded actions*. The guarded action checks whether in the given time interval the variable should be written or not. Within the whole interval this leads to a schedule between the guarded actions which has to fullfill the property that any variable has to be written once.

2.7 Introduction to the Weak Memory Model

The implementation of the speculative execution mechanism is targeted at modern multi-threaded CPU architectures. These architectures implement the weak memory model which is concerned with the write order of variables between multiple threads. In order to gain sensitivity for the issues that had to be dealt with during the implementation, a gentle introduction into that matter is presented here. To ensure the deterministic behavior a thorough understanding of that model is mandatory.

The *memory hierarchy* is used in computer architecture to reduce the memory access latency. It divides the memory vertically into several levels which differ in their characteristics. Higher memory levels enable shorter response times than lower level memory structures. There is a tradeoff between response time and capacity so the faster memories will only hold a small amount of data.

Layers of the Memory Hierarchy		
1.	Registers	
2.	Cache	
3.	Main Memory	
4.	Magnetic Disc	
5.	Tape	Optical Disc

Table 2.1: The memory hierarchy after [TG97].

The traditional memory hierarchy is specified in [TZ86, TG97] and divided in five levels (see table 2.1). The matter of this section will focus on the second and third level, the *cache* and the *main memory* level.

Together the cache and main memory represent the same set of data. The CPU cores always access this data by going directly through the caches. The fourth and fifth layer can be addressed explicitly from the programmer's view. Even the register level can be accessed directly in Assembler or C code.

Since the faster smaller caches cannot hold the same amount of data as the main memory, only a small subset can be held within them. The subset of data that is chosen from the memory is the data most recently being requested from the CPU.

2.7.1 Cache and Memory Interaction

This recent data is organized in cache lines. It is a copy of a sequential memory area. This block of data is transferred between cache and the main memory. Usually each CPU core only interacts with its own assigned cache. The following events upon cache access lead to interaction with the main memory.

Cache Miss: When the CPU requests a data item for the first time it is missing from the cache. So the CPU core is stalled for some cycles while the data is loaded from the memory and stored in the cache.

Capacity Miss: A cache miss which requires to delete one cache line from the cache, even when the cache is not full yet. The reason for this is the usage of hash buckets for the implementation of the cache. A hash bucket is mapped to a certain subset of cache lines, so a cache miss already happens when the hash bucket is assigned to a cache line that is also mapped to the bucket but another cache line that is mapped to the same bucket is requested.

Communication Miss: Before a cache line is written to memory it has to be invalidated from the other caches. When an invalidated item is accessed by the CPU this results in a communication miss and makes it necessary to reload the line from the memory.

2.7.2 The MESI Cache Coherence Protocol

The mismatch between the shared memory of the secondary level and the individual caches of the internal memory level makes it necessary to ensure data consistency between the caches and the main memory as well. Otherwise *lost updates* and *stale data* could be the result. To ensure consistency the *Modified Exclusive Shared Invalid* (MESI) protocol was introduced.

The idea of the MESI protocol is to enable caches to negotiate with each other, which cache has the right to write to a certain cache line. Additional to the write permission, the latest state of a cache line has to be communicated between those caches as well.

1. As long as a cache line is not modified by any thread, it can be kept in several caches at the same time only for read access.
2. For modification a cache line can be assured to be a unique copy by invalidating all copies of that cache line from other caches.
3. Reading of cache lines, which can happen from memory or another cache line.

The MESI cache coherence protocol adds a 2 bit state tag to each cache line (within the cache). The 4 possible states will be presented in the following.

Modified: Recently this cache line had been written in the bucket by the cache. No other cache owns the same cache line. The cache has the responsibility to write the cache line back to memory, before it is invalidated from the cache.

Exclusive: The cache line is owned by the cache which has not yet written to it. Data in memory is up-to-date. No need for writing back.

Shared: No writing to this cache line is allowed because other caches can own the same memory position. There is no need to write back because memory is recent.

Invalid: Cache line has no data, therefor it can be written.

To complete the picture an overview about the protocol messages follows. The messages or requests are communicated between the caches by a shared bus.

Read: Message contains the physical address that is going to be read by the CPU core.

Read Response: The requested data delivered from the cache holding it.

Invalidate: Notify the other caches about the cache line that should be invalidated.

Invalidate Acknowledge: All caches receiving an invalidate request have to send an acknowledgment.

Read Invalidate: Notify the other caches about the cache line that should be invalidated and requests data by transmitting a physical address. The cache owning the requested data item has to send an read response additional to the invalidate acknowledgment.

Write Back: The physical address and the data that is written back to memory is transferred to other caches with this message. Other CPU cores that want to read this data can capture it.

A more in depth discussion of the MESI protocol than the following introduction can be found in [CSG99, McK10]. A figure representing the MESI protocol as state machine is given in [McK10] as well.

2.7.3 Introduction of Memory Barriers

In the cache architecture previously described and the MESI protocol CPU designers had identified two distinct cases where CPU stalls could be avoided. The architecture was extended in order to speed up the cache behavior. This changes had an impact on the ordering of memory writes which affects the programmer writing multi-threaded applications. In summary these changes lead to the new weak memory model.

The two cases which lead to changes in the architecture and resulted in the weak memory are presented in the following. The changes are presented as well in connection to the implications for the programmer.

Introduction to Store Buffers

When a CPU core wants to write to a cash line that is owned by another CPU, then an *invalidate* request is send to the other caches. The CPU core has to wait several cycles until all caches have responded with an acknowledgment.

The solution for this is the addition of a *store buffer*. The store buffer enables writes of cache lines immediately, even if they are owned by another cache. The store buffer holds the written values so long until the acknowledgments by all caches have arrived and the value can be written to the cache.

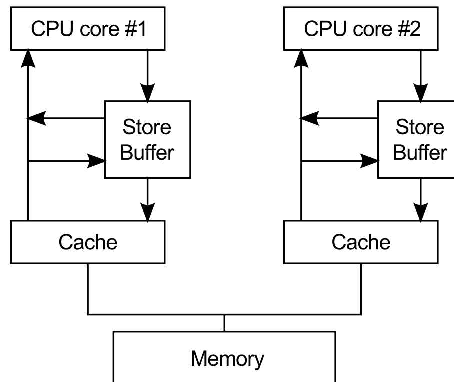

Figure 2.5: Addition of the store buffer

The store buffer and the cache now have two copies of the same cache line. To circumvent the issue that other caches read the stored and duplicated cache line from the store buffer that is probably out-of-date, *store forwarding* has been put in place.

When a cache line is duplicated between store buffer and cache, store forwarding ensures that other caches trying to read that cache line will see the probably more recent value in the store buffer.

Another problem that is unsolvable architecture wise is still untouched. In case another cache receives the invalidate message to late, it is possible that the same cache line is read in this cache before it has been invalidated.

The solution to this problems needs the programmer to act. It has to be accepted that writes are seen in order by other caches.

Write memory barriers are the only way to ensure for other threads some ordering between the writes. When the programmer adds a write memory barrier into the code this means the store buffer is flushed, before any further writes to the CPU cache happen.

This means, if a value that is written *after* a write memory barrier is read by another thread, this thread can be sure that all previously written values are also visible for him.

Introduction of the Invalidate Queues

The processing of invalidate requests cuts a certain amount of CPU time from the overall CPU time. CPU designers found out that these lost CPU cycles can be saved by introducint the *invalidate queue*.

Invalidate queues are attached to every cache like in figure 2.6. They accept the invalidate requests ahead of time, by sending immediately an acknowledgment. The invalidate request is stored in a queue.

It is guaranteed that the invalidate request will be processed before any further messages will be sent by the CPU regarding the cache line.

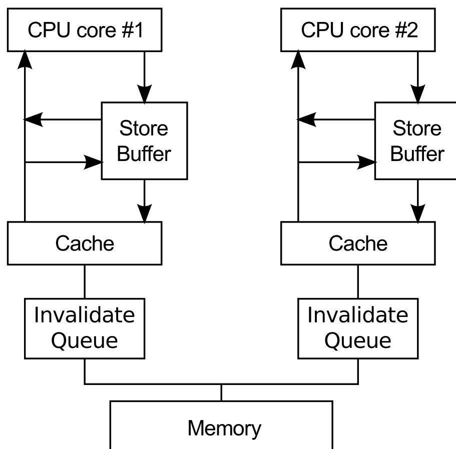

Figure 2.6: Addition of the invalidate queue

The problem with the invalidate queue is that already invalidated cache lines are still present and valid in the cache. Other threads might already have altered this cache line. Invalidate queues don't guarantee that invalidate requests are executed at all, unless the affected cache line makes it necessary to communicate with other caches. So the refresh of these cache lines must be enforced differently.

The solution to this problem are *read memory barriers*. When the programmer inserts a read memory barrier explicitly in his code this flushes all invalidate requests from the queue and executes them. After a read memory barrier all values are up-to-date in the cache.

Chapter 3

Related Work

My topic of research is focused on data-flow speculation for DPN. Since this is a specific area of research, not a lot of relevant related work is available. But since the basic elements of my work, namely the throughput optimization of programs by speculation and the efficient usage of parallel hardware architectures, are also relevant in other areas especially compiler improvements and processor architectures there exist many similar efforts in regard to speculation or parallelism exploitation of software.

3.1 Out-Of-Order Execution Approaches

3.1.1 Data Scalar Architectures

Since in this thesis the out-of-order execution of DPNs has been extended, I want to show another processing architecture different from the data driven DPN model or the control flow driven shared memory architecture, which already implements out-of-order execution. It is possibly interesting to think about an extension for speculation for this architecture as well.

Data scalar architectures [BKG97] try to improve on the memory and cache latency which is part of shared memory architectures.

Data scalar computing is targeted at special hardware platforms where each processor executes the same program and is attached to a very large cache which can hold a large portion of the data that is part of the program. All processors caches combined can hold the whole program data. All the processor are connected by a bus which can be used to communicate computation operands and results.

Because every memory location is owned by only one processor, which is the only processor able to access it in a fast manner, the processor which owns the data that is currently executed is the lead processor. The lead processor is the processor which runs slightly ahead of all the other processors and sends the operands and results of the computations in computation order over the shared bus. The computation order corresponds to the instruction order within the program.

The other processors which listen on the bus can read the operands, compute results in parallel or simply copy the results when they are send from the lead

processor. It is not necessary that the computation of the following processors is finished, since the result can be read from the bus at a later point. Because the operands and results are send over the bus in computation order of the program, no addressing of the cache (memory) locations of the operands or results is necessary.

Following processors can overtake the lead processor when the data which is used by the computation is owned by one of them. The lead processor stalls the execution till the processor which catches up has overtaken the old lead processor.

In [BKG97] this data scalar architecture is adapted to utilize out-of-order execution. This means that every processor can executes those operands first, whose data is available to its cache. This has the effect that the stream of result data is generated more quickly and thus the throughput increases.

Since the data comes out-of-order over the shared bus it becomes necessary to attach tags to the data values to indicate to which instruction they belong.

This is another form of out-of-order execution which is called by the authors *data threading*. Its benefits had been shown in benchmarks which lead to possible speedups of 100% in certain benchmarks.

The main difference of this architecture is the way memory is accessed and handled. The bus is only used for forwarding of computation operands and results. No requests have to be executed on the bus.

The difference in the out-of-order execution between the data-scalar architecture and the DPN execution lies in the way how the data is partitioned. In the data-scalar architecture the CPU caches decide how big the data of can be, that is assigned to a single execution unit. In the DPN model a partitioner decides during synthesis how many and how large the tasks should be.

3.2 Compiler Based Speculation Approaches

3.2.1 Speculative Execution of Alternative Program Paths

An improved form of control flow speculation had been developed by Unger et al. [UUZ98].

In [UUZ98] Unger et al. describe a technique for speculative execution of alternative execution paths. This is a form of control flow speculation which was not part of this work but relates to the concepts of speculation and thus should be mentioned here.

The idea when speculating alternative program paths is to utilize processor cycles better, which are left idle by a lack of instruction level parallelism. This lack of instruction level parallelism can be caused when usual branch prediction mechanisms fail. To enable more instruction level parallelism branches within the code are used to schedule the following program paths onto different threads. This makes it possible to fill the processor pipeline more efficiently.

The throughput is increased when the already executed program paths are actually selected by the control flow. Several instruction results can be already available at the point of time when the branch condition had been evaluated.

The goal is different than my approach that tries to increase the use of several processor cores. In [UUZ98] a higher CPU pipeline usage had been targeted.

The work is founded on an adapted compiler that decides during compilation time which program paths should be executed in parallel. This differentiates it from this thesis' approach as well, since I try to exploit parallelism at runtime.

Hwu et al. [Hwu98] worked on a similar approach but the above presented work from Unger et al. [UUZ98] represents the more recent state of research in that area.

3.2.2 Program Dependence Graph Based Speculation

Bernstein et al. [BR91] adapted the IBM XL family of compilers to incorporate a control flow speculation mechanism which considered the data dependencies between basic blocks.

A program is first translated into a forward control flow dependence graph. The procedure for this is presented in [CHH89]. Instructions between basic blocks can be shifted without any speculation if the basic blocks are equivalent to each other, which means the predecessor basic block dominates the successor basic block and the successor basic block postdominates the predecessor basic block.

The move of an instruction from the successor basic block to the predecessor basic block is speculative when the successor basic block is not the postdominator of the predecessor basic block. In such a case a control flow speculation happens.

The data flow dependencies between instructions are described by data dependence graphs as well (now of the data flow kind). These data flow dependencies are considered when moving instructions between the basic blocks. For correctness of that approach the register liveness (variables that are potentially read before their next write) in each basic block had to be considered also to determine a speculative execution.

Considering the control flow and data flow dependencies together a scheduling mechanism for processor instructions had been defined. This is different from my approach where data flow dependencies are considered between DPN nodes. Also only instructions are executed speculatively between basic blocks, not all instructions can be scheduled speculatively.

3.3 Parallelism Enhancing Hardware Architectures

The goal in this thesis was to enable more parallelism and a better utilization of multi-threaded processors platforms. The target processor platform in my work is the shared memory processor. The shared memory processor with a fixed set of multiple cores makes it necessary that each processor core is issued from a separate thread. Is the thread level parallelism not high enough to employ all cores some processor resources are left unused.

Some processor architectures try to resolve this issue by a more dynamic approach to processor resource allocation to achieve a higher throughput.

3.3.1 Simultaneous Multi-Threading

Lo et al. [LEL+97] discuss simultaneous multi-threading (SMT), another processors architecture. This is not a speculation approach but it tries to go new ways by combining thread-level and instruction-level parallelism. Both forms of parallelism can be interchanged with each other in the SMT processor to some degree to compensate limited parallelism in the program code of the one or other kind.

The idea of the SMT architecture is to make all functional units inside the processor available to all threads. This way multiple threads, independent from the actual number, can compete for the allocation of functional units.

The way this is achieved is by fetching multiple instructions from several threads in each cycle. After instruction and register renaming, instructions from multiple threads can be scheduled onto the instruction queues. Inter thread register dependencies had been resolved by the register renaming step already.

By this mechanism it became less important how many actual threads are available, because the thread level parallelism is "transformed" to instruction level parallelism inside the instruction queues.

3.4 Speculative Hardware Architectures

3.4.1 The Trace Processor

An interesting processor architecture is the trace processor presented in [SV97] by Smith et al.

The ideas of the trace processor incorporate the following principles.

- The trace processor tries to overlook a larger portion of the issued program instructions to exploit more instructions that can be executed in parallel. Therefor a large instruction window is necessary to make more instructions visible to the processor.

- The processor or the hardware architecture should be able to overlook the program as a whole and should be able to partition it in several units which can be assigned to parallel resources simultaneously.

- The integration of control flow speculation is already mandatory since it is widely implemented by branch prediction in most processors. In order to overcome the restrictions of data dependencies, data value speculation should be possible as well.

These principles are realized by the trace processing paradigm. The trace processing paradigm employs instruction fetch hardware that unwinds the program code into traces. A trace is sequence of instructions and predicted conditional branches. A trace resembles a certain path in the program, predicted by a control flow speculation mechanism, spanning over multiple basic blocks of the program.

A trace is the elementary unit of execution in the trace processor. The traces are stored within the trace cache [RBS96]. From the trace cache the trace fetch unit subsequently reads traces and schedules them onto the parallel trace execution units of the processor.

The goal is to reuse a traces several times, by that the trace results are already available when the trace is reused. If a reuse of a trace happens, the result of that trace execution is available within one clock cycle.

The control flow speculation that is usually implemented only as branch prediction, which is already applied in the trace processor, is also applied to traces. To execute several traces at one time in the trace execution units, a speculation about future traces is necessary.

The data value speculation comes into the picture when input values for traces should be guessed. This is necessary to make the trace executions independent from each other and to schedule several, usually sequential, traces at once.

The initial approach to data value speculation for the trace processor, which was presented in [LS96], has been to guess constant values. This is very similar to the guessing approach that is applied in this work. Even more advanced speculation mechanisms had been considered for the trace processor but haven't been published yet.

3.4.2 The Microprocessor Architecture for Java™

MAJC™ supports static and dynamic *branch prediction* to increase instruction level parallelism.

In *static* branch prediction the compiler is responsible to decide whether a branch is likely to be taken. This is also known as *branch steering* by the compiler.

Dynamic branch prediction is the hardware controlled approach, where the processor decides based on a history table of previous branch outcomes whether to take a branch or not. This history table only contains results of branches which were decided to be difficult to predict by the compiler.

As loads from memory to the processor are time consuming, MAJC™ employs *load speculation* to overcome this issue. "The MAJC™ architecture permits software to "speculatively load" the contents of memory to a register before it is certain that the results of the load will be used."[1] When the execution of a memory load is certain, the register holding the prefetched value is assigned to the actual target register of the load instruction. This makes the memory value available when it is needed without the memory latency for loading it.

Another speculative technique applied by the MAJC™ architecture is *Space Time Computing* (STC) which is equivalent to thread-level-speculation. The name comes from letting speculative threads execute future *time* instruction streams in parallel on separate processors. The number of speculatively executed threads depends on the number of processors on the chip. If there are two processors, two threads in total will be executed, execute the non-speculative (head) thread and one additional speculative thread. The *space* part is due to the separate speculative heap, which is reserved for every speculative thread. It keeps the values of all memory references accessed by the instruction stream that is speculatively paralleled.

[1] [SUN, MAJC™ architecture manual, p. 13]

3.4.3 Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution

[KlOE+01] targets recent multithreaded architectures [SUN, OKP+01, SBV95, SCZM00, HWO98] that allow thread-level-speculation.

In thread-level-speculation code sections with data dependencies in between them will also be executed in parallel without enforcing the correct execution order of the memory references. Therefore at the end of a speculative thread the actual execution order has to be checked against the data dependencies. In case of data dependence violations the speculative thread has to be restarted.

To prevent any memory corruption by speculative threads with data dependence violations, multi-threaded architectures use hardware *speculative storage* "to buffer uncertain data, track data dependencies and roll back incorrect executions [KlOE+01, p. 1]." So far *all* memory references (data) of a speculative thread had to be stored in the limited speculative storage.

With memory *reference idempotency* a new program property was introduced that helps to reduce the amount of references stored in the speculative storage. Idempotent references don't have any data dependencies to memory references of other parallel executed threads. Therefore they don't have to be duplicated in the speculative storage and can be kept in the non-speculative storage.

[KlOE+01] proposes 'hardware-only speculation' (HOSE) and 'compiler-assisted speculation' (CASE). In HOSE all memory references of the speculative threads go into the speculative storage.

The CASE approach enables bypassing of the speculative storage by detecting idempotent references at compile time and keeping them in the non-speculative storage.

Experiments in [KlOE+01] have shown, that 60% of all memory references in non-parallelizable code (by non-speculative execution) can be marked as idempotent.

3.5 Task Based Scheduling approaches

3.5.1 The StarSS programming paradigm

The StarSS paradigm is a new programming model trying to exploit concurrency on concurrent architectures. StarSS is not limited to a single platform and adaptions of this approach exist for SMP architectures [PBL08,BPAL09,PBL10, BHLP08], the Cell architecture [BPBL06] and others.

StarSS consists of a source to source compiler and a runtime. The source to source compiler takes regular sequential C or Fortran code and translates it into source code that can be compiled and linked to the StarSS runtime.

StarSS can detect automatically functional parallelism that can be extracted from sequential programs. It's relying on annotations from the developer in the source code to help the StarSS runtime to detect data dependencies between different function invocations. The annotations are therefor targeted at the parameters of the functions that should be paralleled to indicate whether they are input or output parameters (pointers).

The important aspect of StarSS is that the data flow dependencies are detected at runtime. StarSS tries to build a graph where each node represents a

function that is executed in parallel. The node edges represent the data dependencies between the functions.

This is different to the DPN execution mechanism described in this thesis where data flow dependencies are explicitly defined already at compile time. At runtime when the StarSS runtime library finds a function that can be paralleled, a task is scheduled onto a worker thread.

In contrast to the also task based DPN execution mechanism, where worker threads exist during the whole runtime and receive computational load by executing a task object, a global scheduler is part of StarSS which assigns tasks to threads.

To reduce the communication overhead StarSS tries to exploit as much locality as possible, this means that the communication between the nodes due to data dependencies is reduced by scheduling tasks with many dependencies on each other onto the same worker thread. The data can be kept within one thread, instead of communicating it to other ones.

Like other task based scheduling approaches, StarSS has no speculation support. But since it is very similar to the out-of-order execution, it could be possibly extended by one.

3.5.2 Intel® Threading Building Blocks (TBB)

Intel® TBB [KV07] is a C++ template library that is intended to abstract from directly managing threads.

It provides a set of carefully optimized generic parallel algorithms and concurrent containers, that are also used by the implementation discussed in this thesis.

Besides these efficient data-structures Intel® provides a *work-stealing* task scheduler. The task scheduler maintains internally a thread-pool, where each thread is connected to a set of tasks ready to be processed by the task scheduler thread.

The ready tasks are stored in a deque data-structure, which is the so called work-stealing deque. A deque is a double ended queue where tasks can be added on one side and pushed from both sides.

The side for adding and removing tasks is assigned to the thread so it is able, while processing, to add new tasks to the queue and to remove them again, when it is ready for execution. When other threads' ready queues become empty they can steal tasks from the other end of the queue.

Without unveiling too many design and implementation details at this point, there are indeed quite some commonalities between Intel® TBB's task scheduler , which also relies on thread pools and a task-based scheduling approach. A differentiating fact is the usage of work-stealing deque. In this work stealing from other threads is not used, instead worker threads actively schedule tasks to them self and only as a last resort a global task queue is accessed.

The work-stealing task scheduler of Intel® TBB is supplemented by a *scalable memory allocator*. Since memory allocation is quite costly and often the bottleneck in multi-threaded applications this addition is actualy of quite some interest for the speculation mechanism implemented by this work. Due to time

constraints Intel® TBB's memory allocator didn't make it into the implementation.

3.5.3 Cilk

Cilk [BJK+95] is another work-stealing task-based scheduling mechanism, that tries to abstract from a direct interaction with threads by the programmer and instead focuses on the "critical path" that is optimized by the programmer and then efficiently executed by Cilks internal work-stealing enabled thread pool.

A differentiating aspect is though, that Cilk comes as an extension to the C language unlike Intel's ® TBB although Intel® TBB is the direct successor of Cilk. By preprocessing done by Cilk the language is translated to C before being compiled.

3.5.4 Work-Stealing Deques

Work stealing is a popular concept implemented by many applications using internal thread pools and task scheduling. [BL, Blumofe et al.] that the expected execution time of a well-structured application running on top of a work-stealing enabled thread pool would run within $T_1/P + O(T_\infty)$, where T_1 is the minimum serial execution time of the multi-threaded computation and T_∞ is the minimum execution within an infinite set of processors.

A modern and recent implementation of a work-stealing deque can be found in [CL05]. It is a lock-free work-stealing deque, which stores the elements in a cyclic array that can grow when it overflows. The algorithm is only limited by the range of integer indexes and the memory size.

Chapter 4

Implementation

This section will present the system for SDFG execution which I had the chance to work on. My task was to extend it with the capability for speculative SDFG execution. This thesis builds on the efforts from [BBS12, Baudisch et al.] and [Bla11, Blatner].

The implementation presented here is most similar to [BBS12, Baudisch et al.] and is in fact an extension of the same code base.
In order to still give the reader a complete and self-contained understanding on how the developed system works, the implementation is discussed as a whole. I will mention the differences that have been made for the additional speculation capabilities in the detailed implementation description. This also makes it easier to distinct [BBS12, Baudisch et al.] from my own efforts.

This chapter starts with a conceptual overview 4.2 on how speculative execution should work in the context of SDFGs. This broad idea will then be refined to a design discussion 4.3 of the developed software.
The purpose of the former section is to present all components as a whole and their relationships to each other. This gives a broad understanding, without loosing the reader in implementation details. The functionally of each those components will be described as well in an abstract manner, enough to actually understand how everything connected to each other.
In section 4.4 the implementation aspects that are noteworthy and have an impact on the performance will be shown in form of algorithms. Aspects that can be trivially understood already from 4.3 will be avoided. Nevertheless this work should be verifiable so the possibility of re-implementation is given by the level of detail.

For terminology issues subsection 2.3 should be considered.

4.1 Goals for the Implementation

The heart of this thesis is the question whether a speculation mechanism as a whole can increase the throughput of an application defined by an SDFG.

The actual speculation of data-flow computations was secondary. I could find a straight forward method described in 4.4.3 that speculates task functions of SDFGs for relevant applications reasonably well. The amount of correct speculations was high enough. Further improvements of that method would be too likely application specific and were for that reason not considered.

The overall design was more important and had to deal with the following questions:

Q.1 Can speculative execution increase the processors load on multiprocessor systems?

Q.2 How is the CPU load distributed between non-speculative and speculative execution?

Q.3 Is the speculative execution counter-productive? Does the speculative execution block resources for the non-speculative execution?

Q.4 Can the speculative execution decrease the total execution time? Or is the speculation time even increasing?

Q.5 Does the communication overhead between the threads diminish the positive effect of more concurrency? Is the communication overhead higher than the computation speedup?

The speculation method deals more with the following question:

Q.6 How many total correct speculations canbe achieved?

The speculation rate in (**Q.6**) can be seen as a given parameter that is influenced by our speculation method (described in further detail in 4.4.3) but is accepted as not directly influencable.

4.2 A Conceptual Overview

The issue sketched with scheduling SDF graphs and executing them in section 2.4 has always been that at any time too few tasks were ready to be executed. Therefor processor cores have been left idle. The reason is that unfulfilled input data-dependencies hinder the dynamic scheduler from executing more tasks. Even with out-of-order execution [BBS12], which has shown speedup potential over task-based (explained in 4.6.1) in-order execution for applications with much parallelism, processor nodes have been left with no load.

Speculative Execution should make it possible to schedule those tasks that have open data-dependencies left and execute the speculatively. From a SDF graph perspective, those empty input arcs are “filled” with speculated values. Together with those available input data on other arcs a task will be scheduled. The non-speculated values on the input arcs will not be removed by the speculative task execution. Instead of this they can be used later on for a non-speculative execution of the task. Important is that the result of the

speculative execution is stored in an intermediate buffer, that keeps it until the time has come to check the speculation for correctness.

The speculation depth in our approach is one. This means, that once a speculative result has been scheduled the result is not used for any further executions. This should limit the overhead that is part of the speculative execution. Processor resources should not be overfilled with speculative executions since this would have a negative impact on the availability of processor time for non-speculative task executions.

The speculated values will not be inserted into the input arcs or the buffer structures that implement them. Instead they will be produced by a method that writes them directly into the intermediate buffer for the speculation result, from where it is read by the task function. The task function is executed like for a non-speculative execution. The only difference is that the data is read from and written to an intermediate buffer.

The values themselves that will be speculated are the absence reactions of the synchronous guarded actions for the input data that is consumed by the task. For event variables this would have been the default value of the data-type or in case of a memorized variable its the value from the previous macrostep.

Then the speculation result is stored in the aforementioned intermediate buffer. The speculation result will not be requested until all the input data dependencies for the speculated task have been resolved and an actual task is about to be scheduled.

The intermediate buffer is accessed in order to check if speculation results are available. If one speculation result is found, its speculated input values are compared with the actual input which had been produced by predecessor tasks up to this point. If all input values match the guessed ones, the speculation result is immediately available. The throughput of the over SDFG execution has been increased.

If the value does not match the speculation result has to be dismissed and a non-speculative execution has to be started.

The overall goal of the speculation is to execute more tasks in parallel and make execution results earlier available, which should lead to an improvement in the overall execution time.

4.3 The Design of the Speculative Execution Mechanism

The goal of this section is to introduce the components of the SDFG execution mechanism that was implemented in this work. All components and relevant data structures are introduced by name to clarify the necessary terminology for the following section. The reader should receive a broad understanding about how speculative execution was implemented and integrated with the execution mechanism that was already implemented by [BBS12, Baudisch et al.].

In figure 4.1 all main components of the speculative execution mechanism are presented and the functional dependencies between them. I will go through

all the components that are presented in figure 4.1 in the next sections. Some of the non-trivial components are discussed in more depth later in the detailed implementation description (section 4.4).

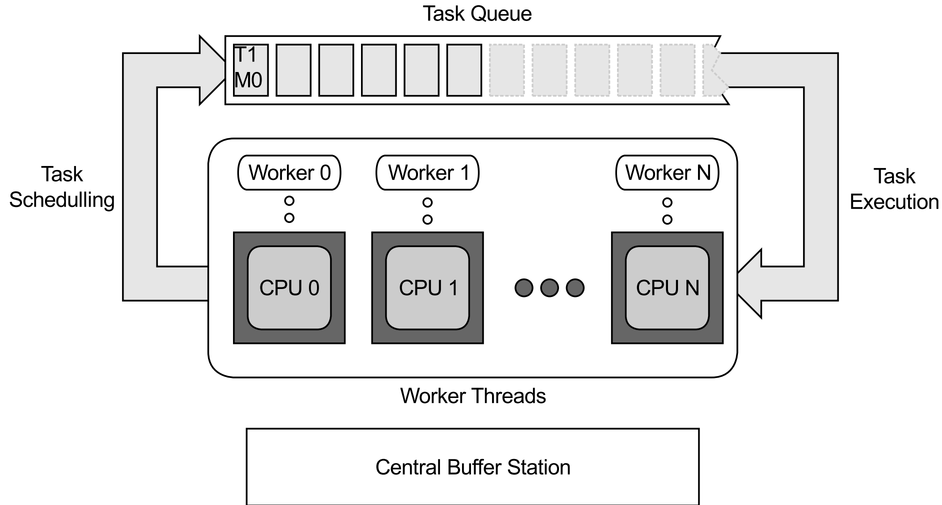

Figure 4.1: The components of the thread-based speculative execution with out-of-order support and speculative execution

4.3.1 Central Buffer Station

Out-Of-Order Central Buffer Station

Since the out-of-order execution of a SDFG is analog to several SDFG executions (macrosteps) in parallel, the buffer data that is written within these macrosteps has to be stored.

Therefor the `SystemState` data type is used. It contains fields for all scalar, tuple and array variables that can be written inside a macrostep. Every field of this data type can only be read and written by one task.

The buffer data of the `SystemState` are stored together with the exeution state of each task of every scheduled macrostep inside the central buffer station (CBS). The CBS is a ring buffer that is organized in rows and columns. Each column can hold the data of a whole macrostep. Each column of the CBS stores also a pointer to the corresponding `SystemState`. The rows divide the macrostep columns into several fields, one for each task (process node) of the SDFG.

Inside the field input and output data-dependencies are stored predecessor and successor tasks. Most important is the number of open input data dependencies, which decides when a task can be executed.

Speculative Central Buffer Station

The CBS needed some adjustments to be able to store data that results from the speculative executions. Therefor a new data structure was introduced, the

`SpecInfo` object. It contains data specific to the speculative execution of a task. The following data is stored within the `SpecInfo` object.

Macrostep Number The number of the macrostep to which the speculatively executed task belongs.

Task Number The number of the speculatively executed task.

Executed Input Tasks The state of the input dependencies during speculation time. The input data dependencies which had to be guessed for the speculative execution are encoded in this data item.

Previous, Current and Next Speculative System State Before the speculative execution happens, the `SystemState`s are copied to store the guessed data values and the execution result apart from the `SystemState`s of the non-speculative state.

Inside the CBS for every field speculation queues had been instantiated. The elements these queues can take are the `SpecInfo` objects that come from the speculative executions. When a task is ready for non-speculative execution the speculation queues are checked for correct speculation results.

4.3.2 Task Queue

The scheduling of tasks happens by pushing task tokens onto the global task queue or by assigning them to worker local variables. These task tokens which contain the relevant data (macrostep number, task number, CBS index) about the task that should be scheduled are implemented by the `TaskInfo` data type. The global task queue can store in every slot one `TaskInfo` object.

The task queue that was part of the out-of-order execution was thread safe and supported the usual queue operations (enqueue, dequeue, etc.). The queue invocations had been blocking calls. In order to implement an algorithm which is able to access queues without the need to permanently wait on them, these queues had been replaced with non-blocking implementations which support both, the blocking and non-blocking invocation of a queue operation.

4.3.3 Worker Threads

Out-Of-Order Worker Threads

In the out-of-order implementation the worker thread was concerned with the scheduling and execution of tasks. When the worker thread had no current task token assigned, it was waiting at the global task queue for the arrival of new task tokens.

After the worker thread had executed a task it will update the open input data dependencies of tasks within the CBS. If new tasks become ready they can be scheduled onto the task queue.

An optimization in the implementation had been the possibility for worker threads to **self schedule** a task token. One out of all tasks that became ready after an execution is only assigned thread local to the worker thread itself.

The advantage of this approach is to avoid the contention at the task queue when several worker threads access it and the synchronization overhead that is

connected to it. It should be noted that it is possible that a task continuously self schedules task onto itself without ever requesting task tokens from the task queue.

Speculative Worker Threads

With the speculative execution the worker threads used the possibilities of the non-blocking queue operations. In order to avoid an overboarding amount of speculations, in each iteration of the worker loop first tries to execute a self scheduled task, if that is not available the task queue is accessed non-blocking to check whether a non-speculative task can be executed. If there is no task ready a self speculated task token can be executed. If all fails the worker thread is forced to wait on the task queue.

Speculative task tokens are only scheduled locally to the worker thread. Since after a speculative task execution no data dependencies in the CBS are update no new non-speculative tasks can be executed. This means that after every non speculative task execution the worker has to wait on the task queue.

The speculative task execution results are stored in the aforementioned `SpecInfo` object and pushed onto the speculation queue in the CBS.

The normal task execution has changed as well. Whenever a task is about to be executed non-speculatively the speculation queue for this task is checked for speculation results. If a `SpecInfo` object can be found that has matching guessed input values to the actually used non-guessed input values the task does not have to be executed. The result values can be copied from the `SystemState`s into the non-speculative `SystemState`s of the CBS.

4.3.4 Implementation Requirements

The speculation mechanism can easily degrade the performance because the speculative execution has a much higher overhead than the non-speculative execution due to the creation of `SystemState`s.

There are a few measures that try to keep the negative performance effects low. These are listed in the following.

Late Speculations The result of a speculative execution is only used when a non-speculative task execution is about to be executed. Therefor the speculative execution result has to finish before the non-speculative execution starts. A speculative execution result that is not used causes only unnecessary overhead.

Speculation Overhead Speculative executions have a high overhead because of the duplication of the system state. Since the high computational effort for speculations should only lead to a higher CPU load but should at best not compete with non-speculative executions for processor time, the amount of speculations executed should be limited.

Misspeculation Avoidance The amount of open input data dependencies of executed speculations has influence on the likelihood of misspeculations. Too many open data dependencies generate worthless speculations since the guessed input data is probably wrong.

4.4 A Detailed Description of the Speculation Mechanism

4.4.1 Central Buffer Station

A necessary step for designing a mechanism for the execution of SDF graphs is the development of a single or a set of data structures that implement or replace the FIFO (first in - first out) buffers of the SDF graph.

This is a decision that has to be carefully thought through, because it has 1. performance implications for the synthesized code 2. out-of-order execution and speculation have to be considered if they should be applied to the data structure(s) 3. additional buffers are needed for storing and accessing speculation results

In the following I will present the design of a centralized buffer structure called *central buffer station* (CBS). This centralized buffer station was already part of the work done in [BBS12] and was extended by support for speculative execution.

4.4.2 The Weak Memory Model and Atomicity

As described in the introduction chapter to memory barriers the *ordered* access to data in weak memory architectures is not guaranteed. Instead the programmer has to enforce a certain ordering between the accessed values by employing memory barriers.

The CBS provides information about the global execution state of the SDF graph. This is information divided among many variables, which have some order relation on each other.

A common scenario that is affected by this is setting a flag, imagine a hypothetical "ready flag" that is set after assorted data has been written. It is assumed but not guaranteed by the weak memory model, that the data has been written when another thread sees this flag enabled. To enforce this intended relationship in state between flag and the assorted data, a store fence must have been invoked *after* the data was written and *before* the flag was set.

This implies no modifications to the CBS, but to procedures that operate on it. Whenever there is a read or write memory barrier used within a worker thread, it has to do with the ordering of the read and written data.

Another issue is the atomic modification of values within the CBS. The CBS has many counters, that can be accessed by different worker threads simultaneously. This makes it necessary to have atomic read-modify-write operations. A fast implementation is provided by the atomic concurrent container of the Intel®Threading Building Blocks, which is used within the CBS.

System State

The execution of SDF graphs happens in iterated macrosteps, which consist of a single firing of all tasks within that step. All the buffer data that was produced

during such a macrostep execution should now be collected within one data-structure. The circumstance that the synchronous language is translated to an SDF graph that is homogeneous (HSDFG) makes it easy to construct such a data structure. Since every FIFO buffer is only written once per macrostep it is enough to store a single data token per buffer. For that purpose the structure *system state* has been created that owns fields of a proper data-type for all buffers.

Because of delayed arcs in the SDF graph, that rely on the data-tokens produced in the previous macrostep, always the previous macrostep have to be available as well. This means we have to store at least two system states side-to-side.

Additional to the buffer data, system state stores state data about the execution state *within* the macrostep.

Random Access

Because of out-of-order execution that is also employed by this work random access to each macrostep is necessary. As out-of-order execution is the execution of tasks before their counterpart in the previous macrostep, it is important to make several macrosteps available at once. Therefor the data-structure has to be tabular or array like, which makes it possible to access a range of macrosteps by index. For constant access times, the data-structure should be flat.

Ring Buffer

From the last paragraph it is already known that multiple macrosteps should be available at once. Since there can be an uncertain amount of macrosteps for each SDF graph execution, the data structure storing its data should make indefinitely long executions possible. Therefor the central buffer station is organized as one large ring buffer [1]. Each buffer slot can hold the buffer data for one macrostep in the execution of the SDF. Therefor the buffer size determines the amount of macrosteps that can be loaded into the central buffer station at once and is referred to as the *macrostep window size*.

The macrostep window size is an important attribute for the out-of-order execution and the speculation as well, because 1. the amount of tasks that can be executed out-of-order is limited to the tasks that are loaded within the central buffer station 2. the speculative execution can only schedule speculative executions for tasks that are loaded into the central buffer station. For the speculative execution is the macrostep window size in general also an upper bound to restrict the amount of speculated tasks and therefor also a restriction for the overhead imposed by the additional speculation. In this particular implementation the speculation is actually limited by the algorithm, which is designed to speculate only after a non-speculative execution.

Task dependencies

Within the macrostep slot execution data for all tasks are stored. For every task there are two counters `taskInDeps` and `taskOutDeps` that describes the

[1] A fixed size buffer where the current write position jumps in circular fashion over the data structure. Old values will be overwritten by new ones. The fixed buffer size determines how many elements can be held at once.

number of open input-data and output-data dependencies.

Open input-data dependencies are tasks that have an outgoing arc *to* this particular task and haven't been fired yet.

Open output-data dependencies are tasks that have an incoming arc *from* this particular task and haven't been fired which means that they haven't consumed the input data yet.

Every time a task fires that is an input data-dependency to other tasks, these tasks are looked up in the CBS and `taskInDeps` is decremented. The same happens for output data-dependencies an their task.

The purpose of `taskInDeps` is to identify the point in time when all input data is available and a non-speculative execution can be started. Initially when this value is written for all tasks of the macrostep and additional '1' is added to this value. This is an scheduling protection which is added and explained later. For the speculation this value is useful, because it shows the number of open input-data. This makes it possible to decide whether the task should be scheduled or not.

`taskOutDeps` is an indicator whether its task and the data it has produced is still necessary for other tasks. This is the case as long these output data-dependencies haven't been executed.

Macrostep Removal

In the beginning of the execution just after the CBS has been created the first macrosteps are loaded till the CBS is filled. For the whole execution, there should be as many macrosteps as the macrostep window size permits.

Since SDF graph execution proceeds and new macrosteps need to be loaded, the old macrosteps have to be removed at some point to create space in the ring buffer. It has to be determined at which point in time it is safe to remove a macrostep.

Since the current buffer content (system state) is stored with each macrostep and the macrostep's tasks have access to the previous macrostep by delayed arcs, a macrostep has to wait for the execution of all output data-dependent tasks of the all tasks of the macrostep. The variable `tasksToWaitFor` is initialized with the number of tasks with output data-dependencies. Every time a task of the macro step has no open output-data dependencies any more (`taskOutDeps == 0`) this counter is decreased. When it has reached 0 it is safe to remove the macrostep with its obsolete system state.

In-Order Macrostep Removal

To guarantee in order macrostep removal, initially `tasksToWaitFor` equals the number of tasks with output data-dependencies + '1'. Only when a macrostep in the CBS becomes the oldest one, or how [BBS12, Baudisch et al.] call it the *head element* it is decremented by '1'. This ensures that the head element is removed first.

Initial State and Abortion

In the context of SDF graphs data-dependencies to a previous macrostep in the initial state had been resolved by assuming the presence of data-tokens on the

input arcs. Because we translate our graphs from synchronous guarded actions these initial tokens would correspond to default values. The SDF graph execution mechanism does exactly that an assumes for these input data-dependencies the default value.

The abortion of and SDF graph execution is triggered by a stop condition. A pointer to this condition is handed to the environment whenever a task is executed. This way the environment can determine when the execution should stop.

Speculation Queues

To store the results of the speculations the speculation queue was added to the CBS. It contains tokens of the new data-type `SpecInfo`.

A instance of this type contains three pointers to copies of the previous, current and next macrostep of the speculated task. This is necessary since delayed arcs of the SDFG would write to the next state, while absence reactions of the task function can access the previous macrostep. These task queues exist for every task of the macrostep.

In an initial approach to scheduling even multiple speculations of the same task had been allowed. To limit that the speculation counter was introduced.

Speculation Counter

To stop multiple speculations of the same task the speculation counter was added to the CBS. Multiple speculations are inefficient when a queue is used that is accessed in first-in first-out manner. The older speculation is always processed first. The following speculation is superfluous overhead.

4.4.3 The Synthesized Task Functions and Task Data

The Averest tool that was used in this thesis transforms the synchronous program into the corresponding SDFG. The transformation into a SDFG is not part of the Averest framework itself. It's an extension developed by Dipl.-Technoinform. Daniel Baudisch and was already utilized in [Bla11,BBS12]. The SDFG transformation during the synthesis provides the following:

1. A data structure describing the DPN graph and the data-flow between its nodes.
2. An *partitioner* that splits the guarded actions into task functions, that represent process node functions of the DPN. The number of task functions that should be created can be given as parameter. This influences the computational load of each task function.
3. Data structures that represent the guarded actions and their subexpressions.
4. Helper functions to translate the guarded actions into C code.

Based on this DPN extension to the Averest tool a part in the synthesis chain was added, to write out the synthesized code. For the non-speculative task execution, the task speculation and the speculation value check we need three methods and additional meta-data.

The Task Function Invocation

The extension of the out-of-order execution made it necessary to adapt the task function representing the DPN behavior and in addition create two more functions, the value speculation function that derives guessed input data for non fired input nodes and the check value function that compares the guessed data with the actual one when all input data is ready for a non-speculative execution.

The tasks functions are invoked from the execution mechanism by a function parameter that is part of an array. The identifier of the DPN node corresponds to the array index where the corresponding task functions are stored.

All three functions share the same set of parameters. Since the guessed input data and the speculation results are stored within copied system states these methods must have been extended to operate over them. Like the non-speculative system states the speculative ones are passed by a pointer to the task function. The non-speculative and speculative system states share the same synthesized data-type that contains fields for every variable that can be read or written during a macrostep of the DPN.

The set of parameters are listed and explained in the following.

Macrostep Number The macrostep is necessary data for the writer and reader hooks of the task behavior function. Dependent on how the environment communication and the implemented application works the macrostep needs eventually to be known. Only some of the benchmarks 5 actually use this parameter.

Although implementation wise this parameter has no important meaning it also denotes a so called reference macrostep number from which the previous, current and next macrostep number can be derived.

For the task behavior function the macrostep number is always the one of the non speculative or speculatively executed task's macrostep.

In case of the value speculation function the macrostep number is the macrostep of the task for which result values should be guessed. This task could be in the same or even previous macrostep of the speculatively executed task due to absence reactions of memorized variables inside the speculatively executed task function.

The check value function's macrostep parameter number is always of the task that had been speculated and should no be checked for correctly guessed input data.

Init Flag This flag indicates whether the task is executed in the first macrostep. Memorized input variables would normally rely on the stored value from the previous macrostep. In the first macrostep this is not available so the default value of Quartz is used as absence reaction. In the computational model of SDFGs there had been initial tokens on the arcs for this case. They were necessary to make the SDFG computable, because there was no possibility to generate tokens without any previous firing of a producing node. This parameter is equal to `macrostep == 0`.

Stop Flag The stop flag itself can be set by the environment to indicate when the communication with it should stop. The execution mechanism will react to this by no further scheduling any tasks.

Previous, Current and Next System State These parameters point to the system state objects that hold the buffer data for the previous, current and next macrostep of the task that is non-speculative executed.

Previous, Current and Next Speculative System State These parameters point to the system state objects that hold the buffer data for the previous, current and next macrostep of the task that is non-speculative executed.

The Task Behavior Function

The task functions are derived from the DFG node data objects that are provided by the partitioner during the synthesis process. In the following DFG node will be used as the data object equivalent to the DPN node that is produced by the partitioner. The DFG nodes are divided into nodes with environment communication (reader and writer nodes) and behavior nodes. The data that is read and written in a behavior nodes can range from the scalar variables to tuples and arrays. In order to refer to variables and single elements of arrays and tuples I will use the term *cell*.

The behavior nodes contain a behavioral description in form of a mapping from a boolean expression, called the fire guard, to a DFG action. This mapping should not be confused with the guarded actions of the synchronous language. The DFG action actually represents all possible actions, including guarded actions and absence reactions, that can be executed within a DPN node. The fire guard determines which of these actions is executed, based on the input data that is consumed by the DPN node.

For every variable that is written inside the DFG node, the behavioral description contains both, the guarded action that is used to write it and the absence reaction, in case the value is not written in the referred macrostep.

The behavioral description is translated in to C code by using conditional statements, where the condition contains the boolean fire guard and the body of the statement contains the DFG action.

A bit care has to be taken when dealing with the absence reaction of memorized values. Per definition the absence reaction for a cell is the value of the data cell from the previous macrostep. In the initial macrostep a previous macrostep doesn't exist. For these cases the theoretical model of DPN incorporates initial tokens on the data edges. In this implementation the absence reaction for memorized values is just the same as for event variables, a default values are chosen.

The task function is used in the speculation mechanism for both, the execution of a non-speculative DPN node execution and a speculative one. In speculative executions the speculated input cells can cause exceptions during runtime.

For example a division by 0 error can happen when a 0 is speculated and is used in division as divisor. Another example would be the speculation of a array index. This speculated value can exceed the actual array size and cause a array out of bounds error.

For the detection of these issues the implementation contains already assertions, that report these kinds of errors. An error handling mechanism is still

missing though. This circumstance limits the type of benchmarks that could be run.

A proper error handling routine could have been implemented in several ways. I want to make some suggestions at this point and show the issues with that.

The division by 0 error is the most easily removed error. By adapting the speculation method 0 values can simply dismissed when an assertion detects its usage as a divisor and is replaced by another value not equal 0. This replacement must uniformly for the guessed cell in the whole behavioral code. A single deviation in the error handling for a cell would be not consistent.

The error handling for guessed array indexes out of bounds would work in a similar way. In addition the error handling procedure had to check first which value would be valid for the given array size, which might be runtime dependent, and then make a replacement for the guessed array index throughout the whole DPN node firing.

The Value Speculation Function

The intention for the value speculation function was that it will be able to guess values for the result data that is produced by a non fired input DPN nodes. To generate this data the speculation mechanism invokes the value speculation function of the non fired input DPN node. Because of delayed writes (next) the non fired input DPN node can be in the previous macrostep of the task that should be speculatively executed.

Since the non fired input DPN node can own memorized variables the absence reaction requires it to access the previous macrostep of the non fired input node. For a non fired input node in the previous macrostep of the DPN node which should be speculatively executed, this would make it necessary to access a macrostep that is two iterations away from the macrostep of speculatively executed task. This shows that it should be necessary to hold this macrostep in the CBS. For a simpler implementation one can neglect that, at the cost of a lesser strict value speculation.

The value speculation function has to produce for the non fired input data node result values. The parameters of the value speculation function that are used to write these result values are the current and next speculative macrostep.

The data is read from the system states that are passed as non speculative previous and current macrostep.

The values speculation function is synthesized from the same behavioral description of the DFG node like the one of the task behavior function. The mapping (the behavioral description maps fire guards to DFG actions) is filtered for absence reactions. The fire guards that are key elements of this mapping are dismissed for the absence reactions. The method body of the value speculation function contains only assignments of absence reactions to cells.

The Value Check Function

The value check function is only synthesized for behavior nodes. Nodes with environment communication should not be speculated so a value check function

for their input parameters is note necessary. Instead for reader and writer nodes a function with an empty method body is generated.

The value check function is invoked when a task should be non speculatively executed and a speculation result for that task is present. Depending on the values of the input data to the task function only a subset of the input data is read. As previously described, an important of that function was to compare only those input values that are actually read by the task function. Otherwise values that had been guessed wrong but are not read would cause the speculation result to be dismissed.

The synthesis of that mechanism is based on a guard map that is part of the DFG node data object. The guard map is a tuple of two mappings. In the first mapping all cells that can be read of the task function are mapped to a the guards of actions that read that cell. In the second mapping all cells that could be written are mapped to the guards of actions that write to that cell. For the value check function only the read mapping is important.

The semantics of the read map is the following: If one of the guards that is mapped by the cell evaluates to true under the given input values, the cell is read.

For every cell in the read mapping a disjunction of all mapped guards is created. In the value check function the cells will only be compared for guessed and non-guessed values if the disjunction evaluates to true.

This is an important optimization. At runtime not all input values of the task are compared, only the ones that are actually read will be checked. This reduces the misspeculation rate, since values guessed wrong which are not read cannot affect the computation result.

If no comparison fails the values are copied from the speculative `SystemState`s to the non-speculative ones in the CBS.

The Task Meta-Data

The task functions depend on incoming data. The availability of this input data is bound to the previous execution of the tasks that produce these data. To provide the information which tasks depend on each other they are stored in arrays for the, referring the tasks by their task number.

For data holding purposes in the CBS the outgoing data dependencies are important. This is redundant data but is stored for each task, so the lookup of this data depndency is faster than searching through all task's meta-data.

Note that both, the incoming and outgoing data dependencies are separated for the different macrosteps they refer to. For the incoming data dependencies there is a list for the current and previous macrostep while for the outgoing dependency there are lists for the current and next macrostep.

For the special reader and writer task functions, which can also carry computational load besides just accessing the environment, in-order execution is

important. To guarantee that these tasks are executed before their counterparts in the previous macrostep a data-dependency is added to the previous macrostep's reader or writer node.

Since the execution of the task function is part of the speculation process and the reader and writer nodes contain the access hooks to the environment, the speculation of these tasks would cause more read and writes to the environment by the additional speculations which would produce false data.

To circumvent this problem a flag was introduced as meta-data to each task to disable speculative execution for reader and writer tasks. This flag is checked by the speculation mechanism and prevents the speculative execution.

It would have been possible to remove the computational load from the reader and writer nodes to gain speculative execution also for these computations.

Since this would have made it necessary to modify the partitioner which was not scope of this work, this idea was not implemented.

Another option was to add two exclusive reader and writer nodes that include the read or write hook for communicating with the environment. These hooks have to be removed from the nodes that had been foreseen as reader and writer nodes by the partitioner. The data-dependencies need to be updated. The following modifications of the graph are necessary:

1. Remove the data-dependency from the old reader and writer tasks to their equivalent in the previous macrostep for in-order execution.

2. Add a data dependency for in-order execution to the new reader and writer tasks to their counterparts in the previous macrostep.

3. Add a data-dependency from the new reader to the old reader in the same macrostep.

4. Add a data-dependency from the old writer task to the one in the same macrostep.

The second alternative did not need any modifications on the partitioner. The issue with this second method is that these two tasks need additional scheduling in the current implementation, which would influence the runtime behavior.

4.5 Worker Threads

In this section I want to describe in some detail the implementation of the worker thread. The worker thread is responsible for the execution and scheduling of tasks. The worker thread contains the necessary logic to decide when a task should be speculatively executed. First the algorithm is described in its entirety in abbreviated C++ code. The underlined method invocations refer to relevant functional aspects of the algorithm.

```
TaskInfo currentToken       = NULL;
TaskInfo selfScheduledToken = NULL;
TaskInfo speculativeToken   = NULL;
bool run                    = true;

while (run) {
    if (selfScheduledToken != NULL) {
        currentToken = selfScheduledToken;
    } else if (speculativeToken != NULL) {
        currentToken = tryPopTaskQueue();
        if (currentToken == NULL) {
            speculateSpeculativeToken();
        }
    } else {
        currentToken = popTaskQueue();
    }
    if (currentToken.taskNumber == STOPPER_ID) {
        run = false;
    } else if (currentToken.taskNumber == REMOVE_HEAD_ID) {
        removeCBSHead();
    } else {
        if (!currentToken.stop) {
            executeTask();
        }
        updateOutputDataDependencies();
        updateInputDataDependencies();
    }
}
```

Listing 4.1: The basic algorithm of a worker thread

4.5.1 Selecting a Task Token for Execution

This description covers the code from line 7 to line 16 of algorithm 4.1.

Worker threads become assigned to a task function by receiving a corresponding task token. This task token (implemented by the `TaskInfo` object) can be assigned to worker thread by the worker itself or by the global task queue.

The task token (`TaskInfo` object) contains only the information about the task (macrostep and task number) and the index where it can find the corresponding task in the CBS.

Speculative executions are also scheduled by a task token. But no task tokens for a speculative execution are pushed onto the global task queue. Speculative tokens are only scheduled by the worker thread for itself. The amount of executed speculations is limited and it will be described in the next sections how it is.

The Previous Approach

The worker thread has been enabled to self schedule one task when it discovers that the task is ready for execution. Self scheduling means that one task can be scheduled for execution in the next worker loop iteration.

The selection of a self scheduled token happens later in the algorithm by assigning a `TaskInfo` token object (task token) to the thread local variable

`selfScheduledToken`. The details on how a task token is selected are shown in section 4.5.4.

It should be emphasized that only one task token can be self scheduled. Since more than one task can become ready for execution, the remaining tasks will be scheduled onto the global task queue.

The benefit of self scheduling is that the global task queue is circumvented. Too many accesses to the task queue can content it and would lead to longer waits due to the synchronization overhead that is created by accessing the concurrent implementation of the global task queue.

This self scheduling happens as long as the worker finds tasks that are ready for execution. Only when a task does not find any ready tasks the task queue is accessed. The task queue is accessed usually by the blocking method call `popTaskQueue()`. This method does not return as long as no task token is available. When a task token has been returned to the worker thread, the algorithm continues with the execution of the corresponding task. This is described in the following sections.

It is important to note that the worker thread is able to continuously self schedule task tokens for itself as long as they become ready after every execution of the preceding task. This makes it possible that the task queue is almost never accessed for the retrieval of a new task token. The synchronization overhead mentioned earlier is avoided.

The Speculation Extension

The scheduling mechanism of the out-of-order execution should be extended for self speculated task executions. The discussed code segment deals with the high overhead of speculative executions (see 4.3.4).

Too limit the amount of speculatively executed tasks and thus the overhead, the speculation should be limited to the scenario where no non-speculative tokens are available to the worker thread.

Since the task queue had been replaced for the speculation mechanism with a concurrent queue implementation which support blocking and non-blocking queue operations, the worker thread is able to pop a task token via the non-blocking `tryPopTaskQueue()` invocation. The method returns immediately and passes the a task token as return value when ready task tokens are available on the task queue. If this is not the case, `NULL` will be returned.

This capability is used when a self scheduled `speculativeToken` is available to the worker. As an alternative to the non-speculative execution the speculative execution of the task token in `speculativeToken` is started.

Depending on the availability of the self scheduled token the task queue is invoked via a blocking or a non-blocking pop request. When there is no such token available the worker thread will wait until new non-speculative tasks are available on the task queue.

4.5.2 CBS Head Removal

This explanation refers to line 20 of algorithm 4.1.

```
void removeHead(int index) {
    cbs[index].init = false;
    for (int i = 0; i < numberOfTasks; i++) {
        cbs[index].specCount[i] = 0;
        cbs[index].taskStatus[i] = "Pending";
        inDeps = numberOfPreviousInBounds
            + numberOfCurrentInBounds + 1;
        cbs[index].taskInDeps[i] = inDeps;
        outDeps = numberOfCurrentOutBounds + numberOfCurrentOutBounds;
        cbs[index].taskOutDeps[i] = outDeps;
    }
    cbs[index].taskForWait = numTasksToWaitFor + 1;
    cbs[index].macrostep = cbs[indexOfTail].macrostep + 1;
    cbs[index].stop = cbs[indexOfTail].stop;
    SFENCE();
}
```

Listing 4.2: The remove head procedure of the CBS

If a task token has been received which carries the `REMOVE_HEAD_ID` as task number the current macrostep in the head of the CBS can be removed. The procedure is actually a overwriting of all entries within the old head column of the CBS. After this procedure the old head index carries the latest macrostep. The column which owns the latest macrostep is called the CBS tail. The next column in the ring buffer (note the ring structure of the CBS) becomes the CBS head index.

Note the additional one to the taskOutDeps value in line 7. This is the scheduling protection. For details refer to 4.4.2.

A the end a `SFENCE()` is invoked to flush all written values from the store buffer of the worker thread running CPU core down to the shared cache. Otherwise other worker threads would use stale data, when they try to execute tasks from old CBS head index.

4.5.3 Task Execution

In this section I describe the `executeTask()` procedure in line 23.

The algorithm first checks the speculation queue whether speculation results are available. All values are popped from the queue and it is checked if they belong to the right macrostep.

If the macrostep of the executed task matches the speculation result's macrostep number the result is checked. It is checked whether the input data of the speculation had been guessed correct. This happens by the synthesized function value check function (see 4.4.3) that was created as part of the synthesized code. The data is directly written into the system state of the executed macrostep of the executed task within the CBS if the data matches. Otherwise a non-speculative execution is started by task behavior function (see 4.4.3).

4.5.4 Updating Data Output Dependencies After a Finished Task

This procedure happens inside the worker loop after an execution of a real task. This code is not executed when a task with a stopper ID or a remove head ID was popped. This section refers to line 26 of algorithm 4.1.

This method handles the input task dependencies within the CBS. For the according data within the CBS see 4.4.2.

It also sets the counter that determines when a macrostep should be removed from the CBS (see 4.4.2).

This subprocedure is executed for the current and previous macrostep within the CBS. If no previous macrostep exists (in the initial state) the procedure is only performed for the current macrostep.

1. Get all tasks of the current and previous macrostep that the executed task's input depends on.
2. For every task that is an input dependency decrease the `taskOutDeps` value by one.
3. If a task has reached 0 also decrease the `taskForWait` value of the related macrostep.
4. If the `taskForWait` has reached 0 remove the macrostep from the CBS. This happens indirectly by scheduling a task with the `REMOVE_HEAD_ID`.

4.5.5 Updating Data Input Dependencies After a Finished Task

This procedure happens inside the worker loop after an execution of a real task. This code is not executed when a task with a stopper ID or a remove head ID was popped. This section refers to line 25 of algorithm 4.1.

This method handles the output task dependencies within the CBS. For the according data within the CBS see 4.4.2. In this code section also new speculative and non speculative task tokens are scheduled. Non-speculative tokens are scheduled when some of the data dependent tasks have no open input dependencies. Speculative task tokens are scheduled only thread local. By checking the open input data dependencies of data dependent tasks the algorithm finds out whether a task has a number of open input data dependencies that lies in between a certain interval. The value `TRIGGERSPEC_MINDEPS` determines the lower and `TRIGGERSPEC_MAXDEPS` determines the upper bound.

`TRIGGERSPEC_MINDEPS` ensures that not tasks will be executed speculatively that will soon receive their ope input data. If too few of this open input data dependencies exist a task is executed non-speculative before the speculative execution of the same task could finish. This measure targets the problem discussed in section 4.3.4.

`TRIGGERSPEC_MAXDEPS` ensures that only tasks will be executed that have a limited amount of open input data dependencies. If that would not be the case, the worker thread would probably schedule tasks whose input values are likely to be guessed wrong. This measure refers to the problem presented in section 4.3.4.

The subprocedure presented in the following is executed for the current and next macrostep of the previously executed task.

1. This algorithm keeps maintains a local integer variable `minDeps`.

2. Execute the following procedure for all tasks of the current and next macrostep that wait for the executed task to finish.
   (a) For every task that is an input dependency decrease the `taskInDeps` value by one.
   (b) If `taskInDeps` has not reached 0 but is within the range from `TRIGGERSPEC_MINDEPS` to `TRIGGERSPEC_MAXDEPS` and the number of open input data dependencies is smaller than the value in `minDeps` the task is assigned to `speculativeToken`. `minDeps` is update with the number of open input data dependencies of the task token of `speculativeToken`.
   (c) If the `taskForWait` has reached 0 remove the macrostep from the CBS. This happens indirectly by scheduling a task with the `REMOVE_HEAD_ID`.
3. When the procedure is finished maybe the task for `speculativeToken` has been updated. By assigning always the smallest number of open input data dependencies to `minDeps` as a comparison value the task with the minimum number of open input data dependencies is found.

4.5.6 The Speculation of Self Scheduled Tokens

When the worker decides to speculatively execute a token that was self scheduled he invokes a sub procedure. This procedure executes the following steps. This section refers to line 12 of algorithm 4.1.

1. The method checks which task should be speculated. Then the system state of the previous current and next macrostep of the task are executed. This is actually a costly procedure. The method needs to allocate new memory for these copies.
2. The input data dependencies of the task are checked. If a input has no data available then a function is invoked that produces this values. This function is synthesized for every task by the synthesis tool. The input value speculation function produces default values for all the data tokens that would be written in a normal execution of a task and sets them to a default.
3. When for every missing data value the value speculation has been invoked the procedure starts to execute the thread by the normal task execution function. But instead using the system states of the CBS is uses the cloned system states.
4. The speculation result is wrapped into a `SpecInfo` object and pushed on the speculation queue in the CBS.

4.6 Discussion of the Implementation Approach

In the implementation of my work certain programming paradigms and models had been applied. In this section I want show the reasoning behind these decisions. In the following I will discuss the chosen paradigms and models and their alternatives by highlighting their advantages and disadvantages.

4.6.1 Thread-Based and Task-Based Scheduling

Influenced by the related work for task-based scheduling presented in section 3.5, the already task-based approach to out-of-order execution by [BBS12] and the more in-depth knowledge I could gain about shared memory architectures and concurrency in section 2.7 lead me to the decision to integrate speculative SDF graph execution into the task-based programming paradigm as well.

Before a more precise explanation of task-based scheduling follows and the design approach for the task-based speculation method follows, some information about the alternative approach is given (thread-based scheduling). Its weaknesses have to be known to understand why task-based scheduling had been chosen.

Thread-Based Scheduling

The common approach when it comes to concurrency in software has always been *thread-based* scheduling. In thread-based scheduling for every code path that should run in parallel a new thread is invoked.

Although threads are known as "lightweight processes", their creation itself has some overhead and is by far more expensive than sequential processing.

Thread Overhead through Context Switches The operating system usually provides a set of kernel threads. Each physical or logical processor core that exists in hardware is usually assigned by the operating system to a kernel thread. A kernel thread can execute kernel-internal functions or execute some user threads instantiated by user-space software. Since there are usually more user threads created than kernel-threads are available, the operating system has to switch between them to give each user thread some processor time.

The overhead of this switch comes from the additional data that is maintained with every thread. There is the stack pointer that has to be saved from the processor registers to memory first, before the stack pointer of the next user-thread can be loaded. Processor registers have to be flushed, which means bad cache performance during or after the context switch. There are scheduling properties, signals and other thread specific data that has to be maintained as well during a context switch.

The Problem of Good Multiprocessor Utilization When every task is executed within a new thread additional effort is necessary to exactly execute that number of threads at any time that utilizes the number of processor cores most efficiently. Too few threads would lead to under-utilization of processor cores, too many threads would degrade the performance due to many context switches.

This is actually a quite difficult task. Since threads have their own stack and communication via shared memory has not only to deal with the atomicity, visibility and ordering[2] but any synchronization in between processor cores has additional overhead as well. The point of time, when a thread has finished its task is not known to other threads and difficult to observe in the global

[2] The main concepts to ensure determinism in concurrent programming.

system state without any additional synchronization. So a new thread cannot be executed that simple when another one has finished.

Since the handling of new threads is so difficult and resource intensive, it had been decided to do it not at all.

Task-Based Scheduling

In task-based scheduling the creation and tear-down of new threads is avoided by the creation of a thread pool. The number of threads within the thread pool is matched against the actual number of physical or logical processor cores and the kernel threads that run on them. The question is how should tasks that are ready for execution be scheduled onto the different threads. There have been two major designs considered during the work for this thesis to enable out-of-order and speculative execution. They are both discussed in section 4.6.2.

The task-based programming model has also been applied by others, like Cilk [BJK+95], OpenMP and Intel Thread-Building Blocks' own task scheduler [KV07] implementation.

4.6.2 Comparison of Global and Worker Based Scheduling

The Global Task Scheduler Approach

A common design for general task based execution is the division between worker threads and a global scheduler. The scheduler has to serve the worker threads with new tasks, when they are ready for execution. The scheduler is the only entity that knows about the data dependencies between the tasks. The workers are only concerned with executing them.

In figure 4.2 all components of this programming model are shown. Further details about their role can be found in the following.

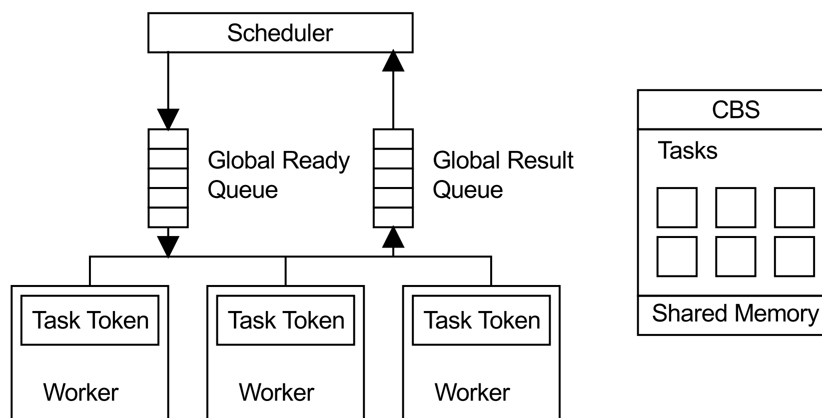

Figure 4.2: The task-based global scheduler approach. A scheduler pushes ready tasks onto a "ready queue". The worker threads pop tasks from the "ready queue" and write the results into the "result queue".

Worker Thread The worker threads in the global scheduling approach are restricted to receiving new task tokens from the ready queue. Self scheduling or the scheduling of new task tokens on the result queue is not foreseen.

Before new task tokens can be scheduled the result tokens (tokens of executed tasks) have to be checked by the global scheduler. So the worker thread has to push a result token onto the the result queue.

Global Scheduler The global scheduler is the only component within the design that has a global knowledge about the execution state and scheduling is its only purpose. It is notified by the result queue about finished tasks and can decide whether a new task should be pushed to the ready queue.

The speculation should only be initiated through this scheduler. The reason for this is the knowledge of the global scheduler about the sum of all recently scheduled speculations. This makes it possible to keep a certain ration between non-speculative and speculative executions. This would even be an advantage over the current speculation approach where the amount of speculation is influenced by the algorithm but no actual ratio can be specified.

Criticism of the Global Scheduling Approach One of the major downsides is that the global scheduler may become easily the bottleneck of the whole approach. When the task and ready queue cannot be processed fast enough the worker threads run out of task tokens. This would lead to a system load where one processor core is heavily utilized and the other ones have only partial load. Because scheduling is not inexpensive its computational overhead should be distributed. The worker based scheduling model implements that idea (see section 4.6.2).

At a first glance the global scheduler with its global statistics about how many tasks have been speculatively executed seems appealing. This enables the the scheduler to hold a certain ratio between non-speculative and speculative execution. But when the scheduling happens not fast enough, the additional increase in processors utilization cannot be reached and this has been the main motivation for speculative execution.

Work Stealing Extension to the Global Task Scheduler Approach An alternative to this approach are the work-stealing algorithms and data-structures [BL]. They combine the global task scheduler with the option for every worker to steal tasks from other worker threads. This usually happens when a worker has run out of tasks. The idle worker thread accesses the work-stealing queues of the other worker threads and steals a not executed task item from it. The downside here is that the bottleneck of the global scheduler still exists if the single task executions are very fast and the scheduler cannot schedule new threads that fast. The advantage of the work-stealing algorithms are the load distribution between the worker threads.

The Worker-Based Task Scheduler Approach

The main critique to the global scheduling approach has been that the single scheduler serving the task and ready queues leads to contention.

The obvious conclusion is to remove the scheduler and distribute the scheduling work between all worker threads. Every worker thread is now able to schedule new tasks. As scheduling is also concerned with out-of-order execution and speculation, this has to be integrated into the worker as well.

The two scheduling queues that had been part of the previous design are no longer necessary. The ready queue in the previous image has been removed. Only the task queue still contains tokens of tasks that should be executed. Now these tokens are pushed directly by the worker threads.

What remains now is the ready queue, the worker threads and the central buffer station in shared memory for task execution data.

4.6.3 Replacing Decentralized FIFO Buffers by Centralized Buffer Storage

Since SDFGs and dataflow computation models are directly implemented in hardware by the dataflow architecture [DM74], the execution naturally reaches the best performance there.

In the *datflow model* no central storage is used. There is nothing like a global memory that is accessed by all computing threads. As described in section 2.2, it uses decentralized FIFO buffers instead.

From the theoretical standpoint of the dataflow model this has two major advantages:

1. The bottleneck of a central memory is avoided. Read and write accesses by the threads have to be ordered to ensure deterministic behavior. This enforcement poses a performance penalty.
2. Since the FIFO buffers decouple all execution units from each other (by avoiding all needs for ordered memory access) the computation is highly asynchronous. This asynchronous computation makes more concurrency possible.

These advantages are actually realized in the data flow architecture. [DM74] describes how buffers are realized as instruction cells and operation queues that feed the computation units asynchronously. A form of computation freed from all kinds of ordered memory access needs.

But since the target of our synthesis is the von Neumann architecture the data tokens cannot be stored in the efficient hardware realization of the dataflow architecture.

Therefore we have to accept the restrictions that the von Neuman architecture gives us. The first one is that a central memory is accessed anyway so we don't have the benefit of point 1. As the high concurrency (point 2) in data flow architectures is enabled by the *non-blocking* memory access of a decentralized memory we have to resemble this as closely as possible on top of a von Neumann CPU.

Therefor in the implementation of the *centralized buffer station* the performance penalties imposed by ordering the accesses of concurrent threads to the same memory had been reduced by avoiding mutual exclusion as much as possible end rather utilizing cache coherence protocols by careful usage of memory barriers [Mck09] and the Intel TBB library.

Chapter 5

Benchmarks

5.1 The Benchmark Environment

For the evaluation of my thesis I chose a head-to-head comparison between the implementation of [BBS12, Baudisch et al.] and my extension of his synthesis tool for speculative execution on equal benchmarks.

The benchmark environment was a PC of the Embedded Systems Group with these specifications:

- Dual processor system with two Intel Xeon DP X5450 quadcore cpus running at 3.00GHz
- 4GB RAM
- GCC 4.7.1
- EGLIBC 2.12.1
- Linux 2.6.35
- mono 2.10.8

The system used was a headless Linux server (without X running). During my experiments the system ran with no heavy user load. This was important, because other processes would compete with my benchmarks for processor time and would degrade the performance of my tests.

5.2 The Measuring Approach

The measuring approach included running the test programs head-to-head several times by changing the parameters over time. The parameters that were altered included the test application input data and the parameters for the (speculative) out-of-order execution. The application specific parameters will be described in the discussion of the benchmark results. First I will present the parameters for the (speculative) out-of-order execution.

Task Number: The task number defines in how many tasks the data process network will be divided by the partitioner. The partitioner is a part of the synthesis process.

Window Size: The macrostep window size determines how many macrostep can be at the same within the central buffer station. This method has an impact on the amount of out-of-order executions that can be scheduled and the amount of speculations that can happen at the same time.

In my benchmarks I measured the total runtime of the test application by using the GNU time command. In my benchmarks I measured the real-, user- and system time. All three measurements are interesting because they give some insight into the runtime behavior of the speculation.

Real Time The real time measurements show the so called wall-clock time. This is the actual execution time like it would have been measured by a stopwatch. Real time is a difficult concept for computers.

For my work this value means if my implementation was actually successful with the goal to increase the overall runtime by throughput optimization with the help of speculations.

User Time The user time is a processor time based on the cycles that had been spent while the test was actually running on a processor core. This time also adds up all the processor times by all threads of the application. Not part of the user time is the time that is been spent with context switches and operating system functions.

System Time The system time is the overhead. It is the sum of the time kernel threads had spent with kernel procedures.

5.3 The Matrix multiplication benchmarks

The matrix multiplication is an example application where the iterations of the SDF graph have only a few dependencies to the next macrostep. This is beneficial for the out-of-order execution because it means that more tasks can be executed out-of-order. The tasks of the next macrostep are not that dependent on the execution of the previous one. It is interesting to see that the speculation had actually in a few cases a positive effect.

Real Time This benchmark showed already interesting results. On the average the performance actually decreased. For nearly all tests there have been slower total execution times. The execution times are longer in the speculative execution than they are in the non speculative one usually except in a few cases. But sometimes when the speculated input data of the tasks were good the execution times actually were smaller than the time the out-of-order execution needed.

User Time The user time is actually quite high. The speculative mechanism only executes tasks with a low number of open data-dependencies. Brute force speculation, which is the speculative execution of tasks that have many open data dependencies, is not applied. This means that the speculation itself is quite expensive. The speculation is only invoked after a non-speculative execution has finished which limits the number of speculative executions.

System time This time is actually lower than the time of the of the out-of-order execution. Therefor we didn't introduce many context switches since our execution mechanism used the cores longer. This is possible because the wait happens less often in the speculative version.

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	0,02	0,03	0,04	8032 (36)	8038 (47)	
16	16	0,02	0,04	0,03	8028 (30)	8042 (60)	
16	32	0,02	0,03	0,04	8044 (50)	8075 (64)	
32	8	0,02	0,01	0,04	7017 (176)	7024 (181)	
32	16	0,02	0,01	0,04	7026 (53)	7042 (150)	
32	32	0,02	0,05	0	7051 (100)	7082 (197)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	0,03	0,09	0,01	8026 (213)	8032 (110)	84.26 % (2865)
16	16	0,03	0,05	0,05	8028 (130)	8043 (124)	84.26 % (2885)
16	32	0,03	0,06	0,04	8044 (257)	8075 (164)	84.26 % (2830)
32	8	0,02	0,07	0,01	7018 (178)	7026 (103)	94.82 % (2816)
32	16	0,02	0,06	0,02	7030 (130)	7045 (91)	94.83 % (2787)
32	32	0,03	0,04	0,04	7046 (182)	7077 (111)	93.82 % (2628)

Table 5.1: Matrix Size 4 Comparison

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	0,04	0,05	0,1	12065 (64)	12035 (7)	
16	16	0,04	0,04	0,12	12153 (281)	12059 (18)	
16	32	0,04	0,1	0,05	12330 (265)	12110 (7)	
32	8	0,02	0,05	0,02	8042 (42)	8031 (12)	
32	16	0,03	0,03	0,05	8098 (76)	8056 (7)	
32	32	0,03	0,04	0,04	8211 (151)	8108 (12)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	0,04	0,08	0,08	12066 (85)	12039 (7)	99.09 % (988)
16	16	0,04	0,1	0,06	12153 (185)	12059 (6)	98.60 % (997)
16	32	0,04	0,06	0,1	12330 (368)	12110 (7)	96.98 % (992)
32	8	0,02	0,02	0,07	8043 (93)	8034 (21)	98.79 % (994)
32	16	0,03	0,04	0,06	8098 (128)	8056 (14)	98.50 % (1000)
32	32	0,03	0,04	0,06	8210 (218)	8104 (14)	96.52 % (976)

Table 5.2: Matrix Size 8 Comparison

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	0,1	0,15	0,22	20114 (139)	20053 (10)	
16	16	0,09	0,15	0,18	20265 (247)	20070 (1)	
16	32	0,1	0,18	0,14	20569 (454)	20121 (2)	
32	8	0,1	0,16	0,19	20113 (136)	20045 (13)	
32	16	0,09	0,17	0,17	20265 (402)	20072 (15)	
32	32	0,07	0,25	0,09	20569 (912)	20117 (11)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	0,09	0,25	0,11	20113 (121)	20050 (2)	99.10 % (998)
16	16	0,1	0,24	0,18	20265 (441)	20072 (14)	98.39 % (992)
16	32	0,09	0,21	0,13	20065 (534)	20054 (31)	96.60 % (941)
32	8	0,1	0,18	0,24	20113 (148)	20044 (10)	99.30 % (996)
32	16	0,09	0,24	0,12	20265 (386)	20071 (6)	98.39 % (994)
32	32	0,09	0,26	0,07	20569 (572)	20116 (4)	96.97 % (989)

Table 5.3: Matrix Size 16 Comparison

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	0,13	0,41	0,06	17096 (172)	17047 (35)	
16	16	0,14	0,43	0,06	17224 (352)	17082 (35)	
16	32	0,14	0,42	0,09	17480 (762)	17119 (52)	
32	8	0,23	0,62	0,29	36209 (371)	36069 (7)	
32	16	0,17	0,54	0,42	36081 (850)	36059 (92)	
32	32	0,23	0,59	0,25	37049 (2431)	36130 (16)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	0,14	0,56	0,12	17096 (210)	17052 (30)	98.99 % (991)
16	16	0,14	0,52	0,13	17225 (632)	17085 (73)	97.81 % (961)
16	32	0,14	0,51	0,14	17480 (1084)	17123 (98)	96.03 % (958)
32	8	0,18	0,6	0,3	36209 (709)	36071 (41)	91.41 % (512)
32	16	0,16	0,51	0,4	36081 (792)	36056 (41)	57.27 % (110)
32	32	0,15	0,59	0,14	37049 (3752)	36137 (49)	95.91 % (930)

Table 5.4: Matrix Size 32 Comparison

5.4 The Pitchshift benchmarks

The pitchshift application is an example application where the iterations of the SDF graph have many dependencies into the next macrostep. This is is problematic for out-of-order execution. Many of the tasks of the next macrostep have open data-dependencies to the previous one. This should be benchmark where the advantages of the speculative execution shows. This actually happened in a few cases very obvious.

Real Time On the average the performance still was not up to the one of the out-of-order execution. But since this benchmarks performs longer the few cases where the results had been faster are very clear.

User Time The user time is again very high. For this careful implementation in terms of speculations this is a sign that there are some improvement areas.umber of speculative executions.

System time The system time showed the same behaviour than in the last example.

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	40,87	76,1	58,09	17000058 (111757)	17000053 (69971)	
16	16	40,77	74,96	57,9	17000067 (118198)	17000064 (69516)	
16	32	40,78	75,43	57,75	17000059 (117695)	17000058 (70751)	
32	8	14,47	83,16	14,7	23000111 (1237956)	23000108 (448664)	
32	16	15,47	83,29	20,47	23000302 (1363629)	23000303 (590609)	
32	32	15,36	81,95	21,39	23000622 (1373701)	23000623 (602637)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	23,75	130,73	21,42	17000056 (287005)	17000054 (85621)	71.36 % (2866729)
16	16	18,42	116,9	14,04	17000202 (323107)	17000192 (77510)	65.79 % (2394221)
16	32	18,7	117,01	15,67	17000417 (333486)	17000416 (82168)	65.97 % (2425839)
32	8	12,20	87,18	3,47	23000114 (1012447)	23000112 (203259)	50.11 % (887316)
32	16	14,67	101,31	6,48	23000273 (779393)	23000267 (204963)	52.01 % (1750788)
32	32	14,48	100,78	6,02	23000590 (788235)	23000586 (201531)	50.92 % (1674864)

Table 5.5: Pitchshift Benchmark Comparison with Window Size 100

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	44,29	75,74	58,27	17000028 (43919)	17000028 (24784)	
16	16	39,15	80,28	55,14	17000036 (111395)	17000035 (48637)	
16	32	31,43	83,48	54,08	17000482 (214128)	17000482 (80922)	
32	8	23,18	99,77	42,43	23000107 (1233500)	23000100 (737465)	
32	16	22,97	98,81	42,16	23000297 (1264599)	23000297 (758847)	
32	32	22,91	98,61	42,45	23000607 (1262976)	23000604 (758957)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	26,96	166,69	20,88	17000087 (330672)	17000084 (97864)	26.97 % (2370970)
16	16	25,56	163,64	19,054	17000212 (330312)	17000209 (89287)	30.37 % (2445410)
16	32	25,88	165,11	19,98	17000433 (329352)	17000432 (88749)	30.97 % (2462488)
32	8	25,14	168,3	14,71	23000136 (520810)	23000137 (177208)	27.39 % (2574945)
32	16	24,33	161,07	14,71	23000280 (550170)	23000278 (192744)	28.95 % (2592788)
32	32	24,59	159,99	16,41	23000593 (555736)	23000592 (195111)	30.22 % (2680963)

Table 5.6: Pitchshift Benchmark Comparison with Window Size 200

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	34,35	98,16	64,17	17000097 (128934)	17000095 (72892)	
16	16	32,85	100,95	64,97	17000212 (121393)	17000209 (68502)	
16	32	32,72	100,87	63,92	17000500 (126218)	17000500 (70578)	
32	8	33,56	116,28	67,57	23000131 (794589)	23000131 (467496)	
32	16	33,52	116,76	66,47	23000290 (807560)	23000288 (475290)	
32	32	33,52	118,42	65,16	23000611 (828924)	23000611 (491905)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	37,56	253,73	19,6	17000092 (232478)	17000092 (59804)	58.74 % (3016983)
16	16	38	251,05	24,42	17000193 (248319)	17000186 (62653)	55.13 % (2787082)
16	32	38,82	254,01	25,54	17000430 (244494)	17000427 (61736)	54.84 % (2778579)
32	8	38,87	262,42	19,83	23000116 (280040)	23000114 (101861)	46.41 % (2679476)
32	16	38,93	252,42	26,10	23000290 (333609)	23000289 (123623)	46.87 % (2667523)
32	32	39,15	255,42	25,06	23000610 (306760)	23000610 (106667)	48.35 % (2714771)

Table 5.7: Pitchshift Benchmark Comparison with Window Size 400

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	49,09	123,02	82,01	17000100 (20838)	17000099 (11643)	
16	16	49,05	122,52	82,97	17000215 (20086)	17000216 (11093)	
16	32	48,74	121,96	84,58	17000492 (7842)	17000492 (4040)	
32	8	50,51	142,48	91,31	23000130 (176162)	23000127 (99599)	
32	16	50,23	143,15	90,72	23000320 (167185)	23000322 (93064)	
32	32	50,03	143,6	89,53	23000619 (175090)	23000617 (96924)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	62,55	395,45	47,99	17000082 (169887)	17000076 (43915)	37.03 % (2416772)
16	16	65,42	406,56	54,11	17000200 (181538)	17000197 (46332)	38.32 % (2375427)
16	32	66,79	422,08	45,29	17000430 (186761)	17000427 (48875)	38.98 % (2425154)
32	8	65,84	418,48	52,14	23000121 (131138)	23000121 (33072)	34.37 % (2515606)
32	16	69,03	424,24	59,28	23000281 (144106)	23000276 (33394)	36.14 % (2507990)
32	32	70,15	440,42	48,72	23000605 (153748)	23000605 (36294)	35.78 % (2502674)

Table 5.8: Pitchshift Benchmark Comparison with Window Size 800

Out-of-order							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	
16	8	73,92	152,38	91,17	17000097 (11023)	17000094 (7756)	
16	16	73,3	151,7	89,76	17000203 (15769)	17000200 (11075)	
16	32	73,19	151,78	89,44	17000433 (15235)	17000431 (10627)	
32	8	81,45	183,67	114,43	23000123 (25889)	23000123 (15098)	
32	16	81,23	182,42	117,31	23000288 (26408)	23000288 (15434)	
32	32	80,82	184,81	115,27	23000611 (23973)	23000611 (14150)	
Out-of-order Speculation							
Task Number	Window Size	Real	User	Sys	Sched. (OO Sched.)	Exec. (OO Exec.)	Spec. (OO Spec.)
16	8	113,97	641,76	157,51	17000098 (69451)	17000095 (13201)	30.44 % (2367940)
16	16	122,88	715,02	141,38	17000203 (82458)	17000200 (16728)	33.39 % (2319993)
16	32	128,45	809,01	94,25	17000477 (84053)	17000471 (18141)	34.16 % (2340492)
32	8	123,1	690,69	173,31	23000118 (34976)	23000116 (13951)	28.23 % (2514555)
32	16	127,97	733,61	158,49	23000278 (46958)	23000276 (17076)	31.78 % (2544197)
32	32	135,92	852	100,52	23000603 (51860)	23000603 (18921)	31.97 % (2546063)

Table 5.9: Pitchshift Benchmark Comparison with Window Size 1600

Task Number	Window size	Speed Up	Executed	OOO	Spec.	Corr. Spec.
16	8	66,67 %	8026	110	2865	84,26 %
16	16	66,67 %	8028	124	2885	84,26 %
16	32	66,67 %	8044	164	2830	84,26 %
32	8	100 %	7018	103	2818	94,82 %
32	16	100 %	7030	91	2787	94,83 %
32	32	100 %	7046	111	2628	93,82 %

Table 5.10: Matrix 4x4 speed up

Task Number	Window size	Speed Up	Executed	OOO	Spec.	Corr. Spec.
16	8	100 %	12039	7	988	99,09 %
16	16	100 %	12059	6	997	98,6 %
16	32	100 %	12110	7	992	96,98 %
32	8	100 %	8034	21	994	98,79 %
32	16	100 %	8056	14	1000	98,5 %
32	32	100 %	8104	14	976	96,52 %

Table 5.11: Matrix 8x8 speed up

5.5 Speedup

The out-of-order and the speculative out-of-order execution was finally compared for the total speedup. When the speed up was compared the real time value of the out-of-order execution was divided by the value for the speculative out-of-order execution.

Besides the results for the speedup the result tables for the matrix multiplication (5.10, 5.11, 5.12) and the pitchshift benchmark (5.13, 5.14, 5.15) contain the number of total executed tasks (col. 4), the number of out-of-order executed tasks (col. 5), the number of speculatively executed tasks (col. 6) and the number of correct speculations (col. 7).

For the matrix multiplication the table data show almost no speedup in any benchmark. Even though the benchmark section for the matrix multiplication had some real times values in the speculative out-of-order execution which were lower than the ones in the usual out-of-order execution benchmarks. This is due to rounding error, in general the speedup that could be measured in the matrix multiplication benchmarks was so small that the advantage was neglectable.

The pitchshift benchmarks showed some more promising performance gains. Even though in some benchmarks were performance decreases up to 15 % shown, the performance gains were usually larger. In some benchmarks a performance increase up to 220 % percent was shown.

The reason for the more outstanding results of the pitchshift benchmarks were the high amount of data dependencies between the tasks of the macrosteps. These data dependencies limited the amount of parallelism in the non-speculative out-of-order execution. The speculative execution was able too exploit more parallelism because not all input data had to be available.

The matrix multiplication had different characteristics. The data dependencies between the macrosteps are low so the non-speculative out-of-order execution was already able to exploit a lot of parallelism.

Task Number	Window size	Speed Up	Executed	OOO	Spec.	Corr. Spec.
16	8	66,67 %	17052	30	991	98,99 %
16	16	66,67 %	17225	73	961	97,81 %
16	32	66,67 %	17123	98	958	96,03 %
32	8	100 %	36071	41	512	91,41 %
32	16	100 %	36056	41	110	57,27 %
32	32	100 %	36137	49	930	95,91 %

Table 5.12: Matrix 32x32 speed up

Task Number	Window size	Speed Up	Executed	OOO	Spec.	Corr. Spec.
16	8	172,08 %	17000054	85621	2866729	71,36 %
16	16	221,34 %	17000192	77510	2394221	65,79 %
16	32	221,34 %	17000416	82168	2425839	65,97 %
32	8	118,61 %	23000112	203259	887316	50,11 %
32	16	105,45 %	23000267	204963	1750788	52,01 %
32	32	106,08 %	23000586	201531	1674864	50,92 %

Table 5.13: Pitchshift 100

Task Number	Window size	Speed Up	Executed	OOO	Spec.	Corr. Spec.
16	8	164,28 %	17000084	97864	2370970	26,97 %
16	16	153,17 %	17000209	89287	2445410	30,37 %
16	32	121,45 %	17000432	88749	2462488	30,97 %
32	8	92,20 %	23000137	177208	2574945	27,39 %
32	16	94,41 %	23000278	192744	2592788	28,95 %
32	32	93,17 %	23000592	195111	2680963	30,22 %

Table 5.14: Pitchshift 200

Task Number	Window size	Speed Up	Executed	OOO	Spec.	Corr. Spec.
16	8	91,45 %	17000092	59804	3016983	58,74 %
16	16	86,45 %	17000186	62653	2787082	55,13 %
16	32	84,29 %	17000427	61736	2778579	54,84 %
32	8	86,34 %	23000114	101861	2679476	46,41 %
32	16	86,10 %	23000289	123623	2667523	46,87 %
32	32	85,62 %	23000610	106667	2714771	48,35 %

Table 5.15: Pitchshift 400

Task Number	Window size	OOO Utilization	Spec. Utilization
16	8	172,08 %	550,44
16	16	221,34 %	634,64
16	32	221,34 %	625,72
32	8	118,61 %	714,59
32	16	105,45 %	690,59
32	32	106,08 %	695,99

Table 5.16: Pitchshift 100

Task Number	Window size	OOO Utilization	Spec. Utilization
16	8	171,01 %	618,29
16	16	205,06 %	640,22
16	32	265,61 %	637,98
32	8	430,41 %	669,45
32	16	430,17 %	662,02
32	32	430,42 %	650,63

Table 5.17: Pitchshift 100

5.6 Processor Core Usage

Task Number	Window size	OOO Utilization	Spec. Utilization
16	8	285,76 %	675,53
16	16	307,31 %	660,66
16	32	308,28 %	654,33
32	8	346,48 %	675,12
32	16	348,33 %	648,39
32	32	353,28 %	652,41

Table 5.18: Pitchshift 100

Chapter 6

Conclusion

The results showed that speculative execution can improve performance in applications quite a bit. Especially in benchmarks a performance increase could be shown where the out-of-order execution was not able to create a lot of parallelism. For example in the pitchshift benchmark, there are many data dependencies between the tasks from one macrostep to the tasks of the same or next macrostep. The high amount of data dependencies made it difficult for the out-of-order execution to schedule tasks in parallel.

Since the speculative out-of-order execution does not require it to have all data-dependencies resolved for execution it can create much more parallelism at once than the normal out-of-order execution could do. Even when the speculation rate averaged on a very low value (e.g. 30% in the pitchshift benchmark), some throughput optimization could have been achieved.

This shows that the primary goal of this work, the better utilization of parallel processor resources, could actually be achieved with speculation when the existing parallelism enhancing measures did not enhance the situation sufficiently.

Not only did the multiprocessor load increase, the higher load also translated into better throughput. This way in certain cases it was possible to gain a 220% speedup. In terms of load usage and throughput optimization the approach showed its potential in certain benchmarks.

Even though the speculation method was not the focus of this work it was very helpful to have in each benchmark at least a certain percentage of correct speculation results. This was the foundation for the throughput improvements that were shown in the benchmarks.

In the end of my work efforts were started to develop a "reference speculation". The idea of that approach was to execute every benchmark in within the speculative execution mechanism in two iterations. In the first iteration for every task and every macrostep the result data is calculated and stored within a large result array of `SystemState` objects. This way the execution result for every task has been precomputed. The second run represents the actual benchmark with speculation. The difference in the second run opposed to a normal speculative out-of-order execution is that the input data for the speculation will be fetched from the result array. The value speculation function will use precomputed data instead of the absence reaction of the `SystemState` variables.

Unfortunately it was not possible to finish the implementation during this thesis. The rate of successful speculations was nevertheless high enough, so reasonable benchmarks were possible which even showed the improvements that were made by the speculative execution mechanism. Even though the implementation of the reference speculation was not finished in this work it still should be presented here. The idea is an interesting one, which can be reevaluated in future work.

Even with the quite simple guessing approach to input values the speculation rate was exceptionally good. Since the speculation method applied absence reactions, which would either evaluate to a default value or the value of a system state variable from the previous macrostep, this would guess in general values with little or no variation. These values were not the values that were generated as results of a matrix multiplication or did they correspond to pitchshifted audio samples. The reason for the highly successful speculation rates must be owed to one optimization that was integrated into the check value function. Because only task input values are compared that are actually read by the task function, it is likely that in cases were the speculation rates were very high (up to 98%) the guessed input values have not been read. This encourages even further the usage of speculation in DPN execution. When open input data dependencies limit the actual parallelism that is available to the application by normal out-of-order execution and these open input data will not even be read during the computation it is only right to speculatively execute the task. The scenario where input values were guessed that are not used as input values is an important use case for the DPN data speculation.

Actually the implementation unveiled some weaknesses as well. The user time and system load was a bit high for the few speculations that were executed additionally in parallel. It is possible that the memory allocations within the speculation algorithm cause this load. The copying of system states is quite expensive because of the new allocated memory. The `new` operator in C++ is implemented in a thread safe manner that leaves improvement potential for the memory allocation in a multi-threaded context.

For any ongoing efforts this part of the implementation should be improved first, to single out bad performance due to memory allocations. A solution to this problem is the avoidance of memory allocation within the speculation (this should not be done at all) and the use of a more efficient memory allocator. This could already improve the execution times. It is possible that the workers that were busy with the speculation could have been blocked for more out-of-order and normal executions. When the speculations perform faster this would decrease the contention of workers.

There are even more ideas to speculation already discussed which will be shown in the next chapter, which means hopefully even better results can be gathered in the future.

A major design aspect of this work had been the limitation of the total speculation amount. The speculations were limited the most by the fact that speculations could only be issued after a non-speculative execution. Several speculations were considered harmful, because they would block worker threads for non-speculative task executions. The design goal was that speculations are

only executed when the worker cannot be used in a better (non-speculative) way. This seemed to be the right thing to do, because nevertheless had been the processor core utilization high enough (close to full usage of all cores) and the performance improvements (up to 220%) showed that the speculation still paid off, even though it was restricted.

To sum things up it could be said that in data flow speculation high successful speculation rates are possible due to the fact that only a subset of the communicated data between the nodes is actually used for the computation. This is a fact that was exploited by my implementation. High speculation rates that are based upon the actual guessing of correct input values are unlikely, because of the large value range of the input data. Control flow speculation whose value range is limited to the binary decision between branch taken and branch not taken have a higher chance to guess the outcome of a branch condition correctly.

In retrospective it was right to be not too concerned about the value speculation. Unused open input data dependencies in DPN were a limiting factor for a high amount of parallelism. It showed that speculative execution was the right way to tackle this problem.

Chapter 7

Future Work

As already mentioned in the conclusion there might exist a some performance issues regarding memory allocation.

An idea is to decrease the memory allocations in my implementation. The objects that were instanciated were of the `SpecInfo` type. A pool of these objects can be created that is always reused for speculations. It is only necessary to overwrite the old data in the `SpecInfo` object. This is by far more efficient than the allocation of a new object.

It should be tried to employ the Intel® TBB scalable memory allocator. The scalable memory allocator is a more efficient implementation of a memory allocator and optimized for multi-threaded processors than the memory allocator that is currently available in C++.

Combined these two approaches could reduce the overhead when the issue with performance was related to memory allocation.

A more complex effort would be to stop deleting old speculation results. Since the speculation result is nothing else than a computation with speculated input data it represents a correct computation result in itself. This computed data could be reused when it is stored and accessed later, when a normal execution for a an equal set of inputs will be executed.

The problem with this approach is that it is not clear how the computation results should be stored and accessed.

A possible solution would be to store the results in a hash map. The key of the hash map is calculated based on the input results and the task number. These to parameters identify every distinct computation.

The key is mapped to hash buckets that will store the result data. Within a hash bucket several computation results can be stored at the same time, all the computations which have the same key because of the hash calculation are gathered in the same hash bucket.

The read operations for hash maps are usually constant. When many results are stored within the same hash bucket linear search would be necessary. Write operations are also constant, only affected by the hash calculation of the key. Therefor the overall added overhead should be small.

The hash map would not only be accessed when a speculation is about to be executed. For any non-speculative execution computation results could be reused. The partitioning of tasks could have a major impact on the performance here. If the task functions are too large, the variation between the input values is too high, so a reuse of computations becomes unlikely. But with the right partitioning this could be another promising approach for throughput optimization of DPN executions.

Bibliography

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. 1.3.1, 1.3.1

[Awa96] Maher Awad. *Object-Oriented Technology for Real Time Systems: A Practical Approach Using OMT and Fusion.* Prentice Hall Computer, facsimile edition, 3 1996. 2.1.1

[BB91] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991. 2.6

[BBS12] D. Baudisch, J. Brandt, and K. Schneider. Out-of-order execution of synchronous data-flow networks. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS)*, Samos, Greece, 2012. IEEE Computer Society. (document), 2.4.3, 4, 4.2, 4.3, 4.4.1, 4.4.2, 4.4.3, 4.6.1, 5.1

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003. 2.6

[BHLP08] Rosa M. Badia, José R. Herrero, Jesus Labarta, and Josep M. Perez. Parallelizing dense and banded linear algebra libraries using smpss, 2008. 3.5.1

[BJK+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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995. 3.5.3, 4.6.1

[BKG97] Doug Burger, Stefanos Kaxiras, and James R. Goodman. Datascalar architectures. In *ISCA'97*, pages 338–349, 1997. 3.1.1

[BL] Robert D. Blumofe and Charles E. Leierson. Scheduling multithreaded computations by work stealing. 3.5.4, 4.6.2

[Bla11] D. Blatner. Out-of-order execution of data-flow process networks for streaming applications. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, December 2011. 4, 4.4.3

[BPAL09] R.M. Badia, J.M. Perez, E. Ayguade, and J. Labarta. Impact of the memory hierarchy on shared memory architectures in multicore programming models. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 437 –445, feb. 2009. 3.5.1

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. 3.5.1

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 241–255, New York, NY, USA, 1991. ACM. 3.2.2

[CHH89] R. Cytron, M. Hind, and W. Hsieh. Automatic generation of dag parallelism. *SIGPLAN Not.*, 24(7):54–68, June 1989. 3.2.2

[CL05] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM. 3.5.4

[CSG99] David E. Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 1 1999. 2.7.2

[DM74] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, December 1974. 4.6.3, 4.6.3

[Fly04] Laurie J. Flynn. Intel halts development of 2 new microprocessors. *The New York Times*, May 2004. 1.1

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems - a tutorial and commented bibliography. In *In Tenth International Conference on Computer-Aided Verification, CAV'98, Vancouver (B.C.), LNCS 1427*, pages 1–16. Springer Verlag, 1998. 2.6

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987. 2.1.1

[HLN+88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and a. Shtul-Trauring. Statemate: a working environment for the development of complex reactive systems. In *Proceedings of the 10th international conference on Software engineering*, ICSE

’88, pages 396–406, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. 2.1.1

[HP88] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing Co Inc.,U.S., new ed edition, 1 1988. 2.1.1

[HP93] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategien für die Echtzeit - Programmierung*. Hanser Fachbuchverlag, 1993. 2.1.1

[HWO98] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGPLAN Not.*, 33:58–69, October 1998. 3.4.3

[Hwu98] Wen-mei Hwu. Technology outlook: Introduction to predicated execution. *Computer*, 31:49–50, January 1998. 3.2.1

[KlOE+01] Seon Wook Kim, Chong liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP01*, pages 2–11, 2001. 3.4.3

[KV07] Alexey Kukanov and Michael Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(04), November 2007. 3.5.2, 4.6.1

[LD87] Edward Ashford Lee and David. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36:24–35, 1987. 2.4.2

[Lee89] Edward Ashford Lee. Scheduling strategies for multiprocessor real-time dsp. pages 1279–1283, 1989. 1.4

[Lee91] E.A. Lee. Consistency in dataflow graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(2):223 –235, apr 1991. 1.4, 2.2.1

[LEL+97] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15:322–354, August 1997. 3.3.1

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *COMPCON*, pages 310–315, 1987. 1.4, 2.2.1, 2.2.2, 2.4.2

[LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society. 3.4.1

[Mck09] Paul E. Mckenney. Memory barriers: a hardware view for software hackers, 2009. 4.6.3

[McK10] P.E. McKenney. Memory barriers: A hardware view for software hackers. http://www.rdrop.com/users/paulmck, June 2010. 2.7.2

[OKP+01] Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *IN PROCEEDINGS OF THE 2001 INTERNATIONAL CONFERENCE ON SUPERCOMPUTING*, pages 368–380. ACM Press, 2001. 3.4.3

[Par95] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California, Berkeley, 1995. 1.4

[PBL08] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008. 3.5.1

[PBL10] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Handling task dependencies under strided and aliased references. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM. 3.5.1

[RBS96] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pages 24 –34, dec 1996. 3.4.1

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995. 3.4.3

[Sch09] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009. 2.5

[SCZM00] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM. 3.4.3

[SGC01] Dave Sager, Desktop Platforms Group, and Intel Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001. 1.1

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling (Wiley Professional Computing)*. Wiley Professional Computing, 4 1994. 2.1.1

[SUN] SUN. Majc architecture tutorial. 1, 3.4.3

[SV97] James E. Smith and Sriram Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *Computer*, 30:68–74, September 1997. 3.4.1

[tea95] *Cadre Technologies Inc.*, 1995. 2.1.1

[TG97] Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization*. Prentice Hall International, 4th edition, 10 1997. 2.1, 2.7

[TZ86] Wing N. Toy and Benjamin Zee. *Computer Hardware/Software Architecture*. Longman Higher Education, 1 1986. 2.7

[UUZ98] Andreas Unger, Theo Ungerer, and Eberhard Zehendner. A compiler technique for speculative execution of alternative program paths targeting multithreaded architectures. In *MICROPROCESSORS AND MICROSYSTEMS 24 (2000) 175-190 THREADED ARCHITECTURES, IN: PROCEEDINGS OF THE YALE MULTITHREADED PROGRAMMING WORKSHOP*, 1998. 3.2.1

[YYWW93] Steven D. Young, Steven D. Young, Robert W. Wills, and Robert W. Wills. Performance analysis of a large-grain dataflow scheduling paradigm, 1993. 2.1.1, 2.2.2