DIPLOMARBEIT

MAPPING KPN-BASED APPLICATIONS TO THE NOC-BASED TOMAHAWK ARCHITECTURE

Christian Menard

24. März 2016

Technische Universität Dresden
Institut für Technische Informatik
Professur Compilerbau

Betreuender Hochschullehrer: Prof. Dr.-Ing. Jeronimo Castrillon
Betreuender Mitarbeiter: M.Sc. Andrés Goens

TECHNISCHE UNIVERSITÄT DRESDEN

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur Compilerbau

Aufgabenstellung für die Diplomarbeit

für	**Christian Menard**
Studiengang:	Informationssystemtechnik, Jahrgang 2009
Matrikel-Nr.:	3565258
Thema:	**Mapping KPN-Based Applications to the NoC-based Tomahawk Architecture**

The programming model of Kahn Process Networks (KPN) serves to abstract computation in a way that eases the leveraging of parallelism in an application. It is common in many domains including signal processing and multimedia. This model features a clear separation of computation kernels, which is useful for distributing computation between diverse hardware resources, which may or may not be heterogeneous. There are several frameworks for defining KPNs and mapping them to heterogeneous systems. One such framework is available at the Chair for Compiler Construction. The objective of this Diploma thesis is to use this existing framework and adapt it for the Tomahawk architecture family, developed at the Vodafone chair at TU Dresden. In this architecture, communication is based on a network-on-chip(NoC) which poses new challenges for mapping algorithms. For generating efficient code for the Tomahawk, a good, working model of the chip is needed. Part of this model includes modeling the NoC communication, which has not been done before. Besides, the Tomahawk includes a so-called CoreManager that takes mapping decisions at runtime. An additional, optional challenge consists in enabling dynamic mapping at run-time by using the CoreManager. For this algorithms must be adapted to generate pertinent metadata for the different processes of the network.

The structure of the work in this thesis will be as follows:

1. Familiarization with the Tomahawk 2 architecture and build-chain.
2. Familiarization with Kahn Process Networks and the mapping framework.
3. Prototype implementation of static KPN mapping to the Tomahawk 2, including a NoC, DMA-based FIFO buffer library as well as code generation and process deployment.
4. Extension of the hardware model for taking the NoC into account.
5. Thorough modeling and testing of KPN application mappings for achieving accurate simulations.
6. Concept and implementation of a minimalistic scheduler for the Tensilica cores in the Tomahawk.
7. Integration with the CoreManager for future versions of the Tomahawk.
8. (Optional): Dynamic mapping and migration decisions in the CoreManager using pertinent metadata of the processes.
9. Writing the actual thesis documenting all the above points.

Betreuer:	Andrés Goens, M.Sc. und Prof. Dr.-Ing. Jeronimo Castrillon
1. Prüfer:	Prof. Dr.-Ing. Jeronimo Castrillon
2. Prüfer:	Dr. Emil Matus (Vorschlag)

Ausgehändigt: 3. August, 2015 Einzureichen: 3. Februar, 2016

Prof. Dr.-Ing. habil. Martin Wollschlaeger
Vorsitzender des Prüfungsauschusses

Prof. Dr.-Ing. Jeronimo Castrillon
Verantwortlicher Hochschullehrer

Statement of Authorship

I hereby declare that I have written this thesis independently and have listed all the used sources and means. I understand that attempted fraud will result in the failing grade "not sufficient" (5.0) and in case of recurrence in exclusion from completing of any further examinations and assessments.

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich reiche sie erstmals als Prüfungsleistung ein. Mir ist bekannt, dass ein Betrugsversuch mit der Note „nicht ausreichend" (5,0) geahndet wird und im Wiederholungsfall zum Ausschluss von der Erbringung weiterer Prüfungsleistungen führen kann.

Dresden, den 24. März 2016

Christian Menard

Science is knowledge
which we understand so well
that we can teach it to a computer;
and if we don't fully understand something,
it is an art to deal with it.

— Donald E. Knuth

Abstract

The efficient creation of software for modern and ever more complex MPSoC platforms is an ongoing challenge. For example, the Tomahawk2 is a modern research MPSoC platform but its programming model TaskC lacks portability, flexibility, and usability. A promising alternative is the MAPS compiler framework that provides a complete tool flow for realizing Kahn Process Network (KPN) applications on MPSoC platforms.

This work proposes an extension to the MAPS compiler framework that allows for execution of KPN applications on the Tomahawk2. The work further investigates the mapping of KPN applications to NoC-based architectures using the Tomahawk2 as an example platform. In the process, this work defines a new communication model for MAPS. Benchmarks illustrate that the proposed approach can outperform TaskC applications in certain scenarios. Evaluation further shows that the proposed communication model provides accurate predictions for network traversal times. However, it is also shown that NoC communication only has a small influence on the performance of the average KPN application.

The essence of all beautiful art,
all great art, is gratitude.

— Friedrich Nietzsche

Acknowledgements

First and foremost, I would like to thank Prof. Jeronimo Castrillon not only for giving me the opportunity for writing this thesis but also for coming to Dresden and establishing the Chair for Compiler Construction at TU Dresden. Besides his expertise, I think his warm and positive nature is a welcome addition to the Computer Science Department.

I am also grateful to my supervisor Andrés Goens who accompanied me along the journey of writing this thesis. I appreciate his advice and help that was available whenever needed. Also many thanks goes to Silexica for their interest in my work and for the providence of access to the SLX tool suite (Former MAPS). Without their support, this work would not have been possible.

My gratitude also goes to the Vodafone Chair for providing access to the Tomahawk2 as well as to all the required tools. In particular, my thanks go to Benedikt Nöthen and Mattis Hasler who provided insight on the Tomahawk2 architecture and who where available for questions whenever needed. Also many thanks to Nils Asmussen for maintaining the Tomahawk2 board that I used during my work.

I also want to use this opportunity to thank my dear friends who accompanied me through the beautiful, yet not always easy time of being a student. And even more I want to thank those friends who know and support me for much longer times. I believe its beautiful to have friends with whom you can share everything, take steps in life together, and help each other to grow along the path.

Last but not least, I want to express my deep gratitude to my family who always supported me and my decisions despite all the obstacles that life put in the ways of their own.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADPLL	all-digital phase-locked loop
API	application programming interface
ASIP	application-specific instruction processor
AST	abstract syntax tree
CFG	control flow graph
DAG	directed acyclic graph
DMA	direct memory access
DSP	digital signal processor
FFT	fast Fourier transform
FIFO	first in first out
FIR	finite impulse response
FPGA	field-programmable gate array
GBM	group based mapping
ISA	instruction set architecture
KPN	Kahn process network
MIMD	multiple instruction multiple data
MPSoC	multiprocessor system on chip
NoC	network on chip

PE	processing element
RISC	reduced instruction set computer
SDF	synchronous data flow
SIMD	single instruction multiple data
TAP	test access port
TI	Texas Instruments
TRM	trace replay module
UART	universal asynchronous receiver/transmitter
VLSI	very large scale integration
XML	Extensible Markup Language

1

Introduction

The emergence of ever more complex applications that incorporate multiple algorithms motivated the design of heterogeneous multiprocessor system on chip (MPSoC) architectures [55]. Such architectures provide improved performance by integrating multiple optimized processing elements (PEs) and other specialized hardware components into a single chip. However, the increasing complexity of hardware and software designs creates a gap between available hardware resources and efficient usage of these resource in applications. This gap is known as the software productivity gap [12].

Closing the software productivity gap is an ongoing challenge. Commonly parallel models of computation such as synchronous data flow (SDF) and Kahn process network (KPN) are believed to be the key for efficient MPSoC programming [36]. Their executional semantics and structure of parallel tasks match the semantics and structure of MPSoC platforms. However, often there is a mismatch between the communicational semantics of a parallel model of computation and the available communication resources in an MPSoC platform. Bridging this mismatch can be challenging [37].

The MPSoC Application Programming Studio (MAPS) is a compiler framework that provides a complete tool flow from the definition of KPN applications to the automatic creation of executable implementations for supported target platforms [10]. The framework contains hardware models and implementations of the communicational semantics of KPN for all supported platforms. Based on the hardware model, MAPS automatically assigns KPN components to the available hardware resources of a target platform and generates code for all target cores in order to implement the computational and communicational semantics of an input KPN application.

The Tomahawk2 is a research MPSoC platform designed for signal processing in communications, e.g. 4G [38]. It features eight processing elements, multiple specialized cores, and a network on chip (NoC) communication layer. Currently, developers use the TaskC programming model for creation of applications [4]. TaskC allows for efficient utilization of hardware resources on the Tomahawk2. However, it is not portable and various limitations hinder practical usage. Therefore, this work proposes to use KPN as an alternative model of computation and the MAPS tool flow for automatic implementation of KPN applications on the Tomahawk2 (Chapter 3).

The current hardware model of MAPS does not consider NoC communications. This leads to imprecise predictions of communication times for NoC-based target platforms like the Tomahawk2. Therefore, this work further proposes a NoC-aware communication model in order to improve performance estimation for NoC-based platforms (Chapter 4).

Before Chapter 3 and Chapter 4 discuss the contributions of this work, Chapter 2 gives an overview on MPSoC architectures in general and introduces the tools and concepts that this work builds upon. The solutions that this work proposes are evaluated in Chapter 5 by discussing a set of experiments and their results. Chapter 6 discuses remaining questions as well as ideas beyond the scope of this work and concludes the thesis.

2

State of the Art and Motivation

This chapter discusses the concepts and tools that form the foundation for this work. Section 2.1 gives a brief overview of multiprocessor system on chip (MPSoC) platforms as well as network on chip (NoC) architectures and thereby introduces the software productivity gap. Before Section 2.3 discusses the relevant components of the MAPS compiler framework, Section 2.2 introduces Kahn process network (KPN), which is MAPS' underlying model of computation. The characteristics of the Tomahawk2 architecture are discussed in Section 2.4. Section 2.5 concludes the chapter by motivating the aims of this work.

2.1 Multi Processor System on Chip (MPSoC)

MPSoCs are an important class of very large scale integration (VLSI) systems. They integrate multiple processing elements (PEs), scratchpad memories, caches, memory controllers, I/O devices, and communication layers on a single chip. By using specialized hardware elements, MPSoCs architectures can provide high performance for multimedia and telecommunication applications while keeping energy consumption low. Important applications for MPSoCs include but are not limited to: signal processing in wireless base stations, packet processing in networks, multimedia processing, and cell phone processors [55].

The first known MPSoC is the Lucent Daytona [1] shown in Figure 2.1 on the following page. It was designed for signal-processing on multiple data channels in wireless base stations. The Lucent Daytona is a multiple instruction multiple data (MIMD) architecture consisting of four PEs. A split-transaction bus connects the PEs to I/O devices and the memory interface. Each PE con-

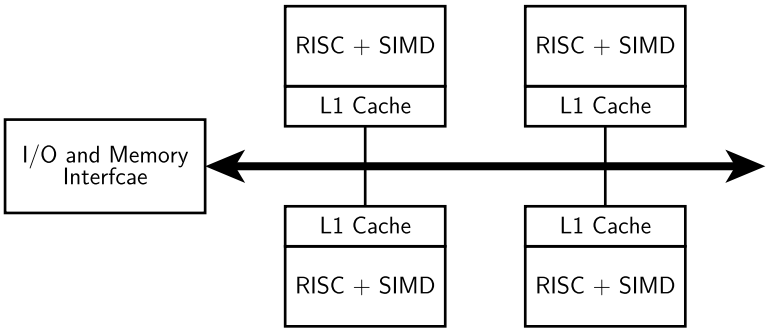


FIGURE 2.1: The Lucent Daytona is the first known MPSoC architecture [1].

sists of a high performance CPU and a reconfigurable level one cache. The CPU architecture is an extended version of SPARC V8 that features a vector coprocessor and additional arithmetic units.

Shortly after the presentation of the Lucent Daytona in early 2000, several other MPSoC designs for a wide range of applications appeared. Early examples include the C-5 Network Processor [16], the Phillips Viper Nexperia multimedia processor [24], the Texas Instruments (TI) OMAP architecture [19] for cell phone processors, and the STMicroelectronics Nomadik cell phone processor [40].

With the highly integrated design of modern MPSoC architectures, an efficient interconnect with high throughput becomes crucial. While early MPSoC designs use crossbar connections and bus hierarchies, modern designs often use a network on chip (NoC) for communication. NoC designs apply methods from networking to the on-chip communication layer. In a NoC addressable endpoints connect to network nodes. Multiple network nodes are interconnected via links and thereby form the network topology. In comparison to conventional bus-based interconnect architectures, NoC architectures provide higher bandwidth, lower latency, better scalability, and more power efficiency [5].

The NoC design space is extensive. It includes design choices like network topology, switching technique, and routing algorithm. Conventional NoC designs often use simple mesh [29] or torus [20] topologies. For large scale networks the hypercube topology or variations like dual cube [32] and metacube [33] are considered to be more efficient.

The switching technique specifies when a node may send messages components to the next node. Common techniques are circuit switching, packet

switching, and wormhole switching [41]. In circuit switched networks, a message transmission locks the whole path until the transmission is complete. This ensures a guaranteed bandwidth but may block other transmissions and, therefore, increases latency. Packet switched networks split messages in fixed size packets. Nodes store incoming packets in buffers and forward them when the desired link is free. Wormhole switched networks further split packets into flits and thereby reduce the required buffer sizes.

NoC routing algorithms are divided into two categories: oblivious and adaptive algorithms [45]. Oblivious algorithms base routing decisions on simple rules that do not depend on the network state. A prominent example is xy-routing in a mesh or torus network, where packets first are forwarded in x-direction until the right y-coordinate is reached and then are forwarded to the destination node in y-direction. Adaptive algorithms base their decisions on the current network state or on its history. As this requires complex network nodes, adaptive algorithms are not commonly used in practical designs.

The development of ever more complex MPSoC architectures with an increasing amount of processing elements of various architectures, with highly specialized components, and various interconnect architectures makes it nearly impossible for software engineers to keep track with new techniques. As also the software complexity rises, a gap between available hardware resources and efficient usage of these resources in software arises. Closing this so called software productivity gap is an ongoing challenge [12].

2.2 Kahn Process Network (KPN)

KPN is the underlying *model of computation* for this work. A model of computation is a fundamental mathematical model that specifies how a computation can progress. A KPN describes a network of concurrent processes that communicate data streams via first in first out (FIFO) channels. Thereby, KPN provides abstraction for communication and task-level parallelism. This makes KPN well suited for modeling streaming applications on MPSoC platforms and a promising tool for closing the software productivity gap.

A KPN is a directed graph in which nodes correspond to processes and edges correspond to communication channels. KPN processes are deterministic and may have an arbitrary control flow (i.e. are Turing complete). Processes communicate by reading/writing atomic data elements (tokens) from/to

channels. KPN channels are unbounded FIFO queues that allow a producing process to send tokens to a consuming process. [27]

A well known characteristic of KPN is its determinism. The behaviour of an application modelled as a KPN is independent of scheduling and timing of communication. In other words, a KPN application always generates the same output pattern for the same input pattern.

At any time, a KPN process either computes or waits for data on one of its input channels [27]. In the best known implementation, a read operation returns a token if there is a sufficient amount of data available. Otherwise, the process that performs the read operation blocks until the data is available [28]. A process may not check if there is data available before performing a read operation. Therefore, process control flow cannot depend on the timing of communication. As KPN channels are unbounded, write operations always succeed.

While KPNs are convenient for modeling signal processing applications, they are hard to analyze because of their use of Turing complete processes. It is well known that the problem of termination is undecidable for Turing machines. In consequence, it is not possible to statically analyze KPNs. Notably, the problems of finding upper bounds for channel capacities and of finding a schedule for a given KPN are undecidable [42].

Models of computation that are more restrictive than KPN are simpler to analyze. The synchronous data flow (SDF) [31] model of computation is a widely used restriction of KPN. SDF processes may only access a known fixed amount of tokens per channel and per iteration. With this restriction it is possible to analyze SDFs statically . A schedule and upper channel bounds can be derived for any given SDF application. However, SDF only models data flow. SDF cannot directly model control flow but can only emulate it using data flow elements. As this is expensive, SDF is only suitable for modeling pure data flow applications. In contrast, the KPN model of computation applies to a broad range of applications and is better suited for scenarios where control flow might be required.

When executing KPN applications on hardware, the limited availability of memory and computational resources needs to be respected. Channels need to be bounded and processes need to share computational resources. As upper channel bounds and a schedule for a given KPN cannot be derived statically, they need to be derived dynamically. For this, Parks proposed an extended model of computation that uses a runtime scheduler [42]. In his extended model of

execution, channels have a fixed capacity. A process that writes to a full channel blocks until there is enough free capacity. As fixed upper channel bounds may introduce artificial deadlocks, the runtime scheduler detects deadlocks and resizes channel buffers as needed. The MAPS compiler framework is based on this extended KPN model.

2.3 MPsoC Application Programming Studio (MAPS)

MAPS is a compiler framework that aims to close the software productivity gap [13]. It provides a complete tool flow for programming MPSoC platforms based on the KPN model of computation. MAPS was first developed as a research project at RWTH Aachen. Now Silexica[1] continues the development of the framework.

The MAPS compiler framework supports a variety of MPSoC platforms including TI (Texas Instruments) OMAP [19], TI Keystone [53], and Adapteva Parallela [3]. The framework provides a method for defining KPN applications, tools for analyzing KPN, hardware descriptions for all supported platforms, implementations of KPN FIFO channels, and tools for the automatic generation of realizations of KPN applications for all supported target platforms. For code generation, MAPS implements a source-to-source compiler that creates C-code. MAPS cannot directly generate machine code for the target platform but relies on the existence of an external C-compiler for machine code generation.

In order to create a realization of a KPN application, MAPS needs to find an assignment of all KPN components (processes and channels) to the available hardware resources (processing elements and memories) of a target platform. Thereby, MAPS distinguishes spatial and temporal assignment. The spatial assignment is called *mapping* and is derived by the *mapper*. The temporal assignment is called *schedule* and is derived by the *scheduler*.

Figure 2.2 on the next page gives an overview of the MAPS compiler framework. Input to the framework are a set of KPN applications with application constraints, an XML description of the target platform, and tool configurations. MAPS analyzes the applications, finds a mapping, and generates code for the target platform. By using a platform description file as input, the framework can easily be configured and can generate mappings for a wide range of platforms. The user provides KPN applications written in a C extension called CPN [47].

[1] https://www.silexica.com/

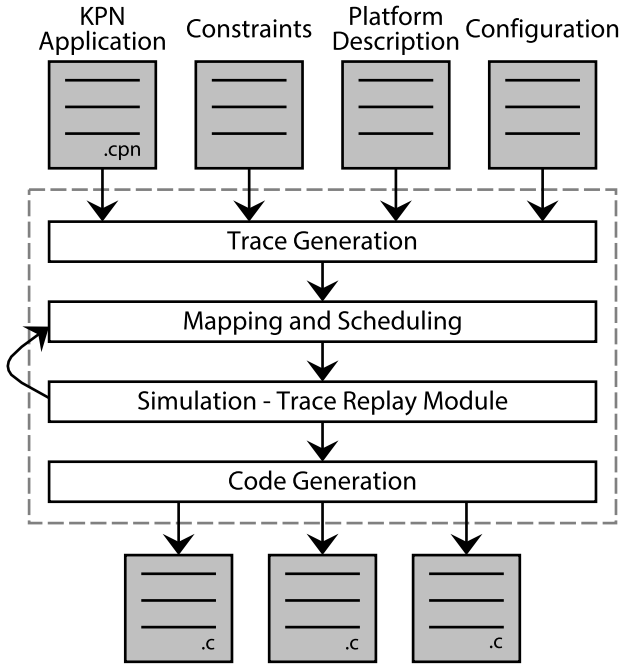


FIGURE 2.2: The MAPS compiler framework provides a complete tool flow for programming MPsoC platforms. It automatically analyzes KPN applications written in CPN, maps them to the target architecture, and generates C code for each target core.

The following sections discuss relevant components of the MAPS compiler framework in more detail. Section 2.3.1 introduces concepts of the CPN programming language, and Section 2.3.2 gives an insight on trace-based analysis of KPNs applications. Section 2.3.3 discusses selected mapping strategies and, finally, Section 2.3.4 introduces code generation.

2.3.1 *C for Process Networks (CPN)*

CPN is the input language of MAPS and allows for portable, structured, and readable description of process networks [47]. It is a C language extension that describes process networks as a set of process templates, channel declarations, and process declarations. For this purpose, CPN introduces a few new keywords which are prefixed by `__PN`.

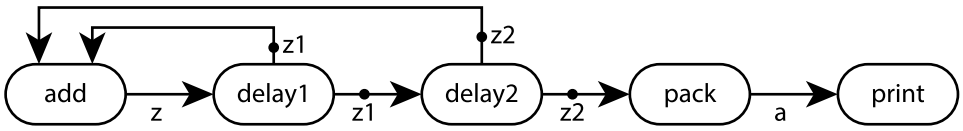


FIGURE 2.3: This KPN represents the CPN application defined in Listing 2.1.

In order to illustrate CPN language features, Listing 2.1 on the following page shows a simple example application written in CPN.[2] This application calculates Fibonacci numbers and prints them in groups of six. Figure 2.3 visualizes the Fibonacci application by showing the corresponding KPN.

The example application is composed of five processes: *add*, *delay1*, *delay2*, *pack*, and *print*. The process *add* consumes one token from each of its two input channels, calculates the sum, and writes the result to its output channel. The processes *delay1* and *delay2* both create a delay by reading one token from the input channel and writing it directly to the output channel. The process *pack* reads three successive tokens from its input channel, packs them into one large token, and writes the resulting token to the output channel. Finally, *print* consumes two of these packs and prints a total of six numbers to the screen.

In CPN, process templates describe the functionality of processes. CPN provides keywords for the definition of KPN process templates (`__PNkpn`) and SDF process templates (`__PNsdf`). The example code defines two KPN process templates *Delay* and *Print* as well as two SDF process templates *Add* and *Pack*. The keywords `__PNin` and `__PNout` define the input and output channels of a process template. CPN also provides an additional keyword `__PNparam` for process parameterization. Within the body of a KPN process template, `__PNin` or `__PNout` statements initiate channel accesses. Code within the body of such statements can access channels as if they were local variables. For SDF process templates the `__PNloop` statement initiates an infinite loop that implicitly accesses all channels.

The declaration of channels in CPN is similar to the declaration of variables in C but requires the additional keyword `__PNchannel`. Valid channel types are all valid C types including structures, unions, and arrays. The assignment operator may be used to place initial tokens on the channel. In the example code in Listing 2.1, line 31 declares three integer channels and line 32 declares a channel where a token is an array of three integers.

[2] The example code in Listing 2.1 is part of SLX as provided by Silexica.

```
__PNsdf Add __PNin(int a, int b) __PNout(int sum) {
  __PNloop { sum = a + b; }
}

__PNkpn Delay __PNin(int i) __PNout(int o) __PNparam(int first) {
  __PNout(o) { o = first; }
  while (1) {
    __PNin(i) __PNout(o) { o = i; }
  }
}

__PNsdf Pack __PNin(int multi:3) __PNout(int array[3]) {
  __PNloop {
    for (int i = 0; i < 3; ++i)
      array[i] = multi[i];
  }
}

__PNkpn Print __PNin(int array[3]) {
  int k = 0;
  while (1) {
    __PNin(array:2) {
      for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 3; ++j)
          printf("%8d ", array[i][j]);
      printf("\n");
    }
  }
}

__PNchannel int z, z1, z2;
__PNchannel int a[3];

__PNprocess add = Add __PNin(z1, z2) __PNout(z);
__PNprocess delay1 = Delay __PNin(z) __PNout(z1) __PNparam(1);
__PNprocess delay2 = Delay __PNin(z1) __PNout(z2) __PNparam(0);
__PNprocess pack = Pack __PNin(z2) __PNout(a);
__PNprocess print = Print __PNin(a);
```

Listing 2.1: This example code illustrates features of the CPN programming language. It defines an application that calculates and prints Fibonacci numbers. Figure 2.3 displays the corresponding KPN.

In order to define the actual process network, process declarations initiate previously defined process templates and connect them to previously defined channels. CPN provides the keyword `__PNprocess` for process declaration. Alongside with an identifier, a process declaration states the process template, all input and output channels, as well as all parameters. The example code in Listing 2.1 declares processes within the lines 34 to 38.

CPN is not a one-to-one representation of the KPN programming model but extends it in various ways. One extension is the support of channels with multiple reader processes. In the example, channels *z1* and *z2* are multiple reader channels as they are read by *delay2* and *add* or *pack* and *add*, respectively. Another extension of CPN over KPN is the support for sliding access windows. Access windows provide access to multiple tokens at the same time without removing them from or writing them to the channel. For instance, line 22 in Listing 2.1 opens an access window of the size of two tokens. The keyword `__PNmove` may be used to advance an access window by a specified number of tokens. Using this keyword, the access window can slide over a series of channel tokens. This feature is especially useful when implementing finite impulse response (FIR) filters.

2.3.2 *Process Traces*

A process of a general KPN may have an arbitrary control flow. Therefore, there is no information on when a process accesses its channels and how many tokens it reads or writes. In contrast, for an SDF graph it is always known how many tokens are produced for each firing and also under which conditions the process fires. As we need information on the occurrence of channel accesses in order to find efficient mappings, analysis of KPN applications is required. MAPS bases its KPN analysis on the evaluation of execution traces derived by profiling the input application using a specific input stimulus.

A process trace is a sequence of segments, where a segment is a path through a process' control flow graph (CFG) [10]. A segment starts and ends with a synchronization event, which is a read access on an input channel, a write access to an output channel, or a time checkpoint. Time checkpoints are needed for applications with timing constraints but are not further discussed in this work.

Based on the information learned from traces, MAPS can derive a schedule and a mapping. However, the behaviour of an application may vary drastically for different input stimuli. Therefore, the input stimuli should represent the average case. In order to derive efficient mappings for multiple scenarios, MAPS could analyze multiple traces and identify regular patterns. However, this is not yet supported.

Based on execution traces and a hardware model, MAPS can estimate the execution time of a specific application using a specific mapping. MAPS provides a performance estimator called trace replay module (TRM) that implements a discrete event simulator. As its name implies, the simulator replays a previously recorded trace. It simulates the timing of resources on the target platform and thereby creates a performance estimation.

2.3.3 *KPN Mapping*

The mapper spatially assigns KPN processes and channels to hardware resources like processing elements and memories. It also applies a bounding algorithm in order to derive upper channel bounds. For both tasks, the mapper uses information gained from trace analysis.

MAPS implements various mapping algorithms. All algorithms generate static mappings that assign all resources at compile time. In contrast, a dynamic mapping postpones decisions to runtime in order to base mapping decisions on the system state.

A simple mapping algorithm is the *Load Balancer*. It assigns processes to processing elements so that the workload is evenly distributed. When a process mapping is found, the algorithm generates a greedy channel mapping by simply selecting the fastest available resource for each KPN channel.

Random Walk is another simple heuristic. The Random Walk algorithm assigns processing elements randomly and evaluates the result using the TRM. This process is repeated for a configurable number of times and then the best observed mapping is selected.

Castrillon, Tretter, Leupers, *et al.* proposed a communication-aware mapping algorithm that operates on groups of similar resources [11]. The group based mapping (GBM) algorithm operates in two phases. The *heterogeneous* phase assigns KPN components to groups of similar hardware components and the *homogeneous* phase assigns KPN components to specific resources of a pre-

assigned group. The algorithm assigns processes and channels simultaneously without any predefined order.

The set of all possible assignments is called *assignment set*. The initial assignment set allows all KPN components to be mapped to any of the available hardware resources. In each iteration, GBM creates a directed acyclic graph (DAG) from process traces and analyzes this DAG to find the critical path. Here, the critical path is the path with longest total execution time. For DAG analysis, the algorithm always assumes the worst possible mapping for the current assignment set.

GBM creates assignment proposals for all nodes in the critical path. A proposal assigns a KPN component to a specific group of hardware components. The algorithm selects the proposal that promises the highest speedup and checks this proposal for feasibility. GBM rejects infeasible proposals until a feasible proposal is found. Then the algorithm reduces the assignment set according to the proposal and recalculates the DAG using the new assignment set. Using the newly formed DAG, the GBM algorithm starts anew by searching for the critical path and creating new proposals. This algorithm is repeated until no further reduction of the assignment set is possible. A more detailed description of GBM can be found in [11].

2.3.4 *Code Generation: cpn-cc*

Code generation in MAPS is handled by *cpn-cc*. cpn-cc is a retargetable source-to-source compiler that takes a CPN-application as well as mapping information as input and generates C-code for a target architecture [47]. The compiler is based on Clang [15], which is the C fronted of the LLVM compiler infrastructure [30]. cpn-cc provides a CPN-aware frontend that creates an extended abstract syntax tree (AST) and transforms this AST until it only contains standard C statements. Figure 2.4 on the following page illustrates the process of code generation in cpn-cc.

Before cpn-cc can operate, a preprocessor removes all C macros and includes all header files. Then the cpn-cc frontend tokenizes and parses the code. Both the tokenizer and the parser are inherited from clang but extended by additional CPN keywords and grammar rules. CPN-aware semantic analysis creates the extended AST from parsed CPN-code. At this point the AST is a

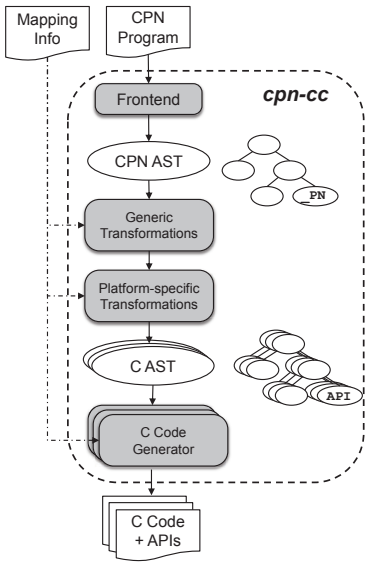


FIGURE 2.4: The code generator creates C code for all target cores. It implements a CPN aware frontend that creates a CPN AST and transforms this AST in order to derive a plain C AST. (Reprinted from [47], © 2013 ACM)

one-to-one representation of the source code and contains all CPN language elements.

In order to generate C code, cpn-cc transforms the CPN AST in several steps until a plain C AST is reached. The AST transformations can be categorized in generic and platform-specific transformations. Generic transformations include for example the transformation of SDF process templates to KPN process templates in order to unify further transformations. Platform-specific transformations replace CPN constructs with C constructs like calls to an application programming interface (API). When the AST transformation is complete, cpn-cc generates C code using Clang's built-in code generator.

2.4 Tomahawk MPSoC

Tomahawk is a research MPSoC family, actively developed at Vodafone Chair for Mobile Communication Systems at TU-Dresden. The first version was presented in 2009 [34]. Currently the second iteration (Tomahawk2) is available in hardware [38] and a new version is in active development.

Besides processing elements of various architectures, the Tomahawk family features two special cores: Core Manager (CM) and Application Processor (App). The Core Manager is responsible for runtime scheduling and map-

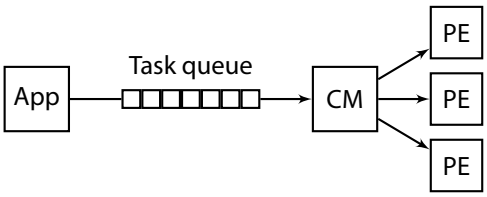


FIGURE 2.5: In the Tomahawk family, the Application Processor (App) issues tasks and the Core Manager (CM) schedules and maps task at runtime.

ping of atomic tasks. The Application Processor executes the application's main thread and issues atomic tasks for execution by sending task descriptors to the Core Manager. The Core Manager reads the descriptors from a queue, checks inter task dependencies, and loads ready tasks to free PEs. Figure 2.5 illustrates this process. By implementing the Core Manager as an application-specific instruction processor (ASIP), the overhead for dependency checks and task initialization can be kept low while maintaining programmability [4].

2.4.1 *Tomahawk2 Architecture*

The Tomahawk2 MPSoC is designed for signal processing applications in communications, most prominently 4G [38]. Figure 2.6 on the following page illustrates the top-level design. The Tomahawk2 features 20 heterogeneous cores connected by a star-mesh NoC. The cores include Application Processor, Core Manager, eight processing elements (PEs) that comprise two cores each, and two programmable application-specific cores: a sphere detection (SD) core and a forward error correction (FEC) core. An I/O-interface connects the Tomahawk2 to a field-programmable gate array (FPGA) and a DDR2 memory interface connects to two 128 MiB memory banks. Each component implements its own all-digital phase-locked loop (ADPLL) and thus the operating frequency of all components can be adjusted individually within a range from 83 to 666 MHz.

The Tomahawk's processing elements are implemented as Duo-PEs. Each Duo-PE comprises two cores of different instruction set architectures (ISAs): a vector digital signal processor (DSP) core and a reduced instruction set computer (RISC) core. The vector DSP provides high-performance 16 bit single instruction multiple data (SIMD) operations. The RISC core is a Tensilica LX4 core [51] and implements a floating-point unit. Both cores share 64 kiB of scratchpad memory—32 kiB for instructions and 32 kiB for data. The two cores

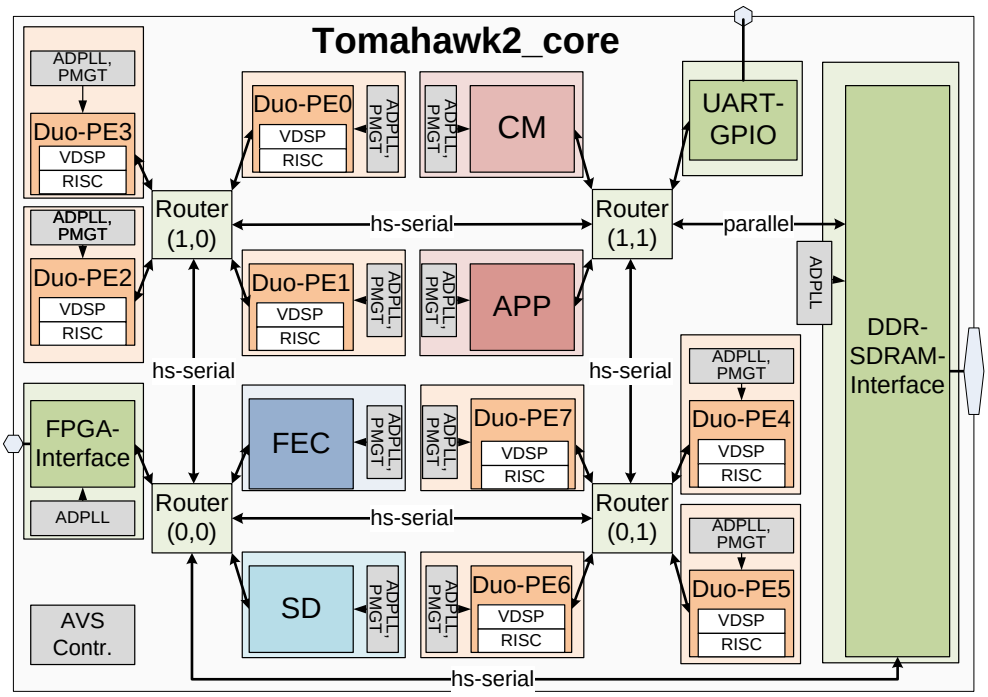


FIGURE 2.6: Tomahawk2 architecture (Reprinted from [38], © 2014 IEEE)

of one Duo-PE cannot operate simultaneously. However, it is possible to switch between cores at all time.

Similar to the PEs, the Core Manager is also based on a Tensilica LX4 core and has 64 kiB of scratchpad memory. However, the Core Manager's instruction set is extended by optimized instructions for task scheduling [4]. The Application Processor is implemented as a Tensilica 570T RISC core and has two 16 kiB caches for instructions and data.

2.4.2 *Intra-Chip Communication*

On the Tomahawk2, all components communicate via a hierarchical xy-routed star-mesh NoC [4]. The star-mesh topology combines characteristics of the classic mesh and star topologies. In a classic mesh network, each router connects exactly one node to the network. In contrast, the Tomahawk's star-mesh network connects multiple nodes to one router. This minimizes the distance between

nodes that are connected to the same router and reduces the total number of routers as well as the maximum number of hops between two nodes compared to a standard mesh topology.

The Tomahawk's routers are packet switched at a fixed packet size of eight bytes. Each one of the four routers has six bidirectional ports. In each cycle, an input port can forward a complete packet to an output port. As all ports operate in parallel, a router can handle up to six packets in one cycle. However, if packets on multiple input ports need to be forwarded to the same output port, the router cannot forward the packets simultaneously but selects one packet at a time using a round-robin strategy. On the Tomahawk2, all routers operate asynchronously. Therefore, data transferred between routers needs to be synchronized, which leads to a delay of two to three cycles for the transfer of one packet between routers.

Each PE has a direct memory access (DMA) controller that handles all NoC communication. It allows for bidirectional accesses to all other modules. In order to issue a transaction, the DMA controller needs to be configured via memory mapped I/O. The direction of communication, local address, remote module ID, remote address, and message size need to be specified. As the NoC can only transport eight byte packets, the message size is restricted to multiples of eight bytes. Furthermore, both local and remote address need to be eight-byte aligned.

2.4.3 *The TaskC Programming Model*

The programming model that is currently used for the Tomahawk architecture is called TaskC. It uses a concept of atomic kernels of computation (tasks) that may read and write data to and from predefined memory areas. As the tasks may be compiled for various architectures, TaskC supports heterogeneous platforms.

In TaskC a task definition specifies the behavior of a task similar to a function definition in C. A custom compiler can derive task descriptors from the task definition. The main thread of each TaskC application runs on the Tomahawk's Application Processor and issues tasks by sending task descriptors to the Core Manager. The Core Manager checks dependencies between tasks and maps ready tasks to free PEs. Figure 2.5 on page 15 illustrates this process of task issuing and mapping.

```
#pragma TASK_BEGIN add_task
#pragma TASK_TYPE some_core
void add_task (int* in1, int* in2, int* out) {
    *out = *in1 + *in2;
}
#pragma TASK_END
```

Listing 2.2: This example code illustrates a task definition in TaskC. The task `task_add` simply adds two integers.

```
#include <task.h>

uint64_t fibonacci[256];

int main() {
  // data initialization
  fibonacci[0] = 1;
  fibonacci[1] = 1;

  for (int i=2; i < 256; i++) {
    task(add_task, IN (&fibonacci[i-2], 8),
                   IN (&fibonacci[i-1], 8),
                   OUT(&fibonacci[i]  , 8));
  }

  taskSync(); // wait for all tasks to terminate

  /* print data */
}
```

Listing 2.3: This example code shows an application written in TaskC. The application performs task calls (`task`) in order to issue parallel computations. This code calculates a sequence of 256 Fibonacci numbers using the task `task_add`.

Listing 2.2 and Listing 2.3 on the facing page illustrate the usage of TaskC by implementing a simple example application. Listing 2.2 defines a task `add_task` that calculates the sum of two integers. Listing 2.3 defines the main function that runs on the Application Processor. It uses the `add_task` to calculate Fibonacci numbers.

The structure of the TaskC Fibonacci application is simpler than the structure of the CPN application in Listing 2.1 on page 10. However, this difference in complexity is not due to a general difference between both programming models. The CPN application is designed to illustrate various features of the CPN language but it could also be implemented just using a single process.

In TaskC, the main thread performs a task call by calling the function `task` with a task definition identifier and a series of `IN` and `OUT` macros as arguments. The macros specify address and size of memory blocks used as input and output for the task. The main thread is responsible for allocating these memory blocks and for providing valid input data.

The Core Manager detects dependencies by comparing input and output memory areas of issued tasks. If the input area of a task T_A overlaps with the output area of a task T_B, T_A depends on T_B. In consequence, T_A may only be executed after T_B finishes computation and all data is written back to RAM. Therefore, in TaskC all dependencies are implicitly defined through overlapping areas of input and output memory. In the example in Listing 2.3 this mechanism ensures that all tasks are processed in sequential order.

TaskC does not allow for explicit definition of dependencies. This has a negative influence on the programmability and on the effort required for performing dependency checks in the Core Manager. The Core Manager always has to perform runtime dependency checks even if the dependencies are static and already known at compile time. In order to avoid the detection of unintended dependencies, the programmer has to handle memory management carefully. Moreover, the programmer has to keep track of intended dependencies manually as they may not be directly visible from the source code. For instance in Listing 2.3, the dependency of all tasks to their predecessor is only visible after inspecting the array indexes.

A further disadvantage of TaskC, is the introduction of two potential bottlenecks. As the Core Manager has to perform dependency checks for each issued task, its performance becomes crucial, especially when using fine grained tasks. For applications with a high communication volume, the memory controller becomes a potential bottleneck. As TaskC does not allow for direct communica-

tion between two tasks via local memory, all communication in an application has to traverse the memory controller.

On the whole, TaskC incorporates special architecture features of the Tomahawk2 (Core Manager and Application Core) and allows for low-level application design. However, there is a lack of abstraction that limits maintainability and portability of TaskC applications. The practicability of TaskC is further limited by the prohibition of global variables and function calls in task definitions.

2.5 Motivation

The limitations of TaskC make this programming model hardly suitable for practical usage. Therefore, the evaluation of alternative programming models for the Tomahawk2 is eligible. One promising alternative is the KPN model of computation. The aim of this work is to provide and evaluate an infrastructure that is capable of executing KPN applications on the Tomahawk2. As the MAPS compiler framework provides most of the required tools, it is chosen as the foundation for this work.

The extension of the MAPS compiler framework for support of the Tomahawk2 as a target platform has a twofold benefit. Besides the benefit from introducing and evaluating an alternative programming model for the Tomahawk2, incorporation of the Tomahawk2 architecture is of interest for the MAPS project. As currently MAPS does not support modelling of NoC-based architectures, this work gives insight on how to model NoCs in MAPS and how NoC architectures influence the execution of KPN applications.

3

Backend

One goal of this work is the extension of the MAPS compiler framework to generate realizations of KPN applications for the Tomahawk2. In order to achieve this goal, a new compiler backend as well as an architectural model need to be created. This chapter introduces a simple backend for code generation for the Tomahawk2. While a platform model that describes the available hardware components is mandatory for code generation, communication costs and interconnect model do not need to be precise. Therefore, Chapter 4 discusses the hardware model separately.

Section 3.1 introduces a simplified model of the Tomahawk2 platform that is used throughout this work. Section 3.2 discusses restrictions of the Tomahawk2 platform as well as early design choices and their influence on the backend. Based on that, Section 3.3 discusses the general design of the new backend. Finally, Section 3.4 points out details of the backend implementation.

3.1 Tomahawk2: A Simplified View

The Tomahawk2 architecture, as discussed in Section 2.4.1 on page 15, has a wide range of hardware components and features. However, the MAPS framework cannot use all hardware components of the Tomahawk2 and consideration of some features would go beyond the scope of this work. Therefore, this section introduces a simplified model of the Tomahawk2 platform as a basis for this work. Figure 3.1 on the next page illustrates this simplified platform model.

The MAPS compiler framework is a source-to-source compiler. It generates C code and relies on a C cross-compiler for machine code generation. However, there is no C compiler for the Tomahawk's vector DSP cores. Therefore, MAPS

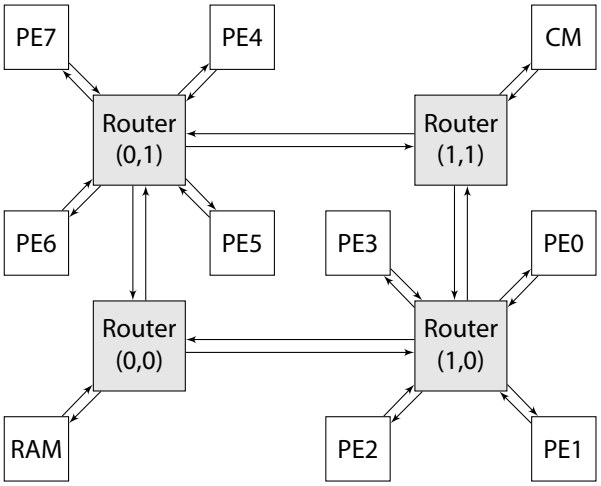


FIGURE 3.1: This simplified model of the Tomahawk2 is used throughout this work. It excludes various features of the Tomahawk2 that are not required for execution of KPN applications.

cannot use the vector DSPs as target cores. A C compiler only exists for the RISC cores of a Duo-PE. Consequently, the simplified model treats the Tomahawk's Duo-PEs as single PEs that implement a Tensilica LX4 core.

The Tomahawk's special purpose cores for sphere detection (SD) and forward error correction (FEC) are not freely programmable. Therefore, they are also excluded in this work. However, one could think of an extension for CPN, which introduces special process templates that are not freely programmable but use special purpose cores.

The simplified model in Figure 3.1 also neglects the Application Processor (App). The Application Processor is only required for execution of TaskC applications and a dedicated core for running the main thread is not required for execution of KPN applications. However, the MAPS backend could consider the Application Processor as a general purpose PE. This is possible as the Application Processor is freely programmable and a C compiler exists. However, the communication capabilities of the Application Processor are limited. It solely operates on global RAM which it caches locally. The Application Core cannot directly access the scratchpad memories of the PEs. The usage of caches makes prediction of NoC communication and execution times difficult. Also cache coherence needs to be considered when communicating via global RAM.

On the whole, utilization of the Application Processor is not required for execution of KPN applications and would introduce additional design challenges. Therefore, this work excludes the Application Processor from the architectural model.

Similar to the Application Processor, the Core Manager is only required for execution of TaskC applications but not needed for execution of KPN applications. However, the Core Manager is used for various tasks throughout this work. As the Core Manager is the only component on the Tomahawk2 that implements a hardware timer, evaluation in Chapter 5 uses the Core Manager for time measurements. Also the Tomahawk2 backend for MAPS uses the Core Manager for allocation of buffers in global memory on startup (see Section 3.4.3). Therefore, the platform model in Figure 3.1 includes the Core Manager.

On the Tomahawk2, ADPLLs allow for individual adjustment of the operating frequency for all components. Incorporation of static or dynamic frequency scaling in order to optimize energy consumption into MAPS would be an interesting problem. However, this would go beyond the scope of this work. Therefore, the simplified platform model neglects ADPLLs and uses the Tomahawk's default ADPLL configuration. In this configuration Core Manager as well as all PEs operate at a fixed frequency of 200 MHz ans the NoC routers operate at 500 MHz.

The resulting model, as shown in Figure 3.1 on the facing page, includes eight PEs, the Core Manager (CM), and global RAM. Each PE and the Core Manager have 64 kiB of local scratchpad memory—32 kiB for instructions and 32 kiB for data. A star-mesh NoC with full duplex links connects all components. All other components, including the FPGA and universal asynchronous receiver/transmitter (UART) interface, are not required for this work and are not included in the platform model.

3.2 Limitations

Currently, the availability of libraries for the Tomahawk2 apart from TaskC is poor. Notably there is no implementation of mechanisms for synchronization and mutual exclusion like locks and semaphores, no multithreading library, no runtime scheduler, and no memory allocator.

The lack of a comprehensive runtime environment puts several restrictions on the mapping of KPNs to the Tomahawk2 and on possible FIFO channel implementations. Most importantly, because of the lack of a runtime scheduler and a multithreading library, it is not possible to map multiple processes to the same PE. The design and implementation of a custom scheduler would go beyond the scope of this work. Therefore, the Tomahawk2 backend presented here only supports one process per PE. Due to the eight PEs of the Tomahawk2, this limits the maximum number of KPN processes to eight.

The usage of Protothreads could be a good alternative to a runtime scheduler. Protothreads are lightweight stackless non-preemptive threads especially designed for systems with limited computational resources [23]. However, Protothreads require all state to be global. This needs to be respected when programming a KPN process and may lead to increased memory usage, as variables with a local scope need to become global and, therefore, take up space constantly.

The Xtensa ISA provides hardware support for mutual exclusion by an optional synchronization feature and an optional read-conditional-write instruction [52]. However, the Tomahawk2 PEs do not implement any of these features. Therefore, mechanisms for mutual exclusion can only be implemented using software solutions like Dekker's algorithm [22] or Peterson's algorithm [43]. Implementations of these algorithms are hard to maintain and introduce an additional overhead. Therefore, the backend design should avoid the need for mutual exclusion when possible.

As there is no run-time allocator for the Tomahawk2, all resources should be allocated at compile time. This can be done by adding global and local variable declaration to the code that MAPS generates. However, on the Tomahawk2 this is only possible for resources stored in a scratchpad memory, as the PEs cannot directly access the global RAM. Therefore, global resources are allocated during initialization at runtime. As this is only required once during startup, a simple greedy allocator is sufficient.

CPN extends the KPN model of computation by support for multi-reader channels and sliding windows. However, these extensions are not required for execution of KPN applications. Therefore, the backend presented here does not support these CPN features in order to keep the implementation overhead low.

```
typedef /* channel structure */ channel_t;

channel_t* create(/* parameters */);
channel_t* open(/* parameters */);
void destroy(channel_t* channel);

void read_begin(channel_t* channel, /* parameters */, void** elem);
void read_move(channel_t* channel, /* paramters */, void** elem);
void read_end(channel_t* channel, /* parameters */);

void write_begin(channel_t* channel, /* parameters */,void** elem);
void write_move(channel_t* channel, /* paramters */, void** elem);
void write_end(channel_t* channel, /* parameters */);
```

Listing 3.1: MAPS internally uses a standard API as an interface between channel library and code generator.

3.3 Design

A MAPS backend for a specific target platform consists of two parts. Firstly, each backend provides a channel library that implements FIFO buffers as well as communication directives and provides abstraction through a standard interface. Secondly, the code generator is part of the backend and provides specific code transformations for each target platform. Basically, the code generator transforms CPN statements to function calls to the channel library.

Listing 3.1 displays the library interface that MAPS uses internally. All functions operate on a data structure that represents the channel state. There are functions for creation and destruction of channels as well as a function for opening an existing channel. The channel library provides three functions each for read and write operations. The *begin* functions initiate a transaction and open a read or write window. The *elem* parameter is used to return a pointer to this window so that the user can read data from or write data to it. The *end* functions close the read or write window. Finally, *move* functions may be used to advance a currently open access window. This is necessary for the sliding window feature of CPN. However, the Tomahawk2 backend presented here does not support this feature and, therefore, does not implement the *move* functions.

The channel library interface as presented in Listing 3.1 is not mandatory and may be adapted as needed. As both the code generator and the channel library need to be newly created for each new target platform, the interface between the two components can be adopted for each platform. However, usage of the standard interface allows for code reuse in the code generator and easy exchange of channel libraries.

According to the limitations discussed in Section 3.2 on page 23, the backend does not support multiple reader channels. Therefore, each channel has exactly one producer and one consumer process. Each process is mapped statically to one PE and each PE can only execute one process. The process and channel mapping is known at compile time. As the Tomahawk2 does not provide any hardware support for locks, the design should be lock-free. Moreover, the Tomahawk2 has limited resources and therefore a good design should keep both the memory footprint and the processing overhead as low as possible.

The mapping specifies the assignment of processes to PEs as well as the assignment of channels to memory resources. In the Tomahawk2 backend, each channel uses a circular buffer to store tokens in FIFO fashion. This buffer may be stored in global RAM or in the scratchpad memory of one PE. When stored in a scratchpad, the buffer can either reside in the consumer's or in the producer's scratchpad.

If the channel buffer is stored in a scratchpad, the process that runs local to the scratchpad can directly access the channel data. However, if the channel data is stored in global memory or in a remote scratchpad, data needs to be transferred via DMA. These transfers require an additional, local, temporary storage that operates as the source for write transfers or as the target for read transfers. This local buffer has to be large enough to hold one token.

Each process keeps a data structure for each channel it uses. This structure stores channel ID, token size, channel size, read pointer, write pointer, and a base pointer that points to the circular buffer. Here a pointer identifies the memory address as well as the module ID of the component that stores the buffer. If the ring buffer is not stored locally, the structure also stores the address of the channel's temporary storage.

Both the read and the write pointer identify the channel state. As both the consumer and the producer process may change the channel state, both processes need to exchange their read and write pointers. In order to do so, each process needs to know where the communication partner stores its read

and write pointers. The location of these pointers is also stored in the channel structure.

The only resources that producer process and consumer process share are read pointer, write pointer, and the actual ring buffer. As consumer and producer operate in parallel, we need to ensure that there are no race conditions when operating on these resources.

Only the consumer process updates the read pointer and only the producer process updates the write pointer. The producer never writes the read pointer and the consumer never writes the write pointer. Therefore, race conditions caused by two processes writing to the same pointer are impossible. The process of updating a pointer is split into address calculation and write back. Therefore, it is not an atomic operation. However, this is not critical as only the process performing the address calculation may update the corresponding pointer. The pointer value cannot change during address calculation. The write back operation itself is atomic as it can use a single NoC packet for writing the pointer. On the whole, this design does not require any locks for the handling of read and writer pointers.

Read and write access to tokens stored in a ring buffer needs to be mutual exclusive. A consumer process must not read partially written tokens and a producer process must not overwrite a token that the consumer is reading. This is guaranteed by only updating the channel state at the end of each read or write operation. This way the consumer only sees tokens that are completely written and the producer only sees empty slots that do not contain any valid data. Therefore, this design allows for the implementation of a lock-free channel library.

3.4 Implementation

The previous section introduced the general design of the Tomahawk2 backend. For a more precises understanding, this section discusses the backend's implementational details. Firstly, Section 3.4.1 discusses the structure of files generated by the backend and the setup used for executing application on the Tomahawk2. Secondly, Section 3.4.2 focuses on code generation in cpn-cc. Finally, Section 3.4.3 discusses details of the channel library implementation.

3.4.1 *Setup and File Structure*

The Tomahawk2 used for testing and evaluation throughout this work is located on a specially designed board. This board connects the Tomahawk2 to periphery like I/O-devices, RAM, and an FPGA interface. It also provides an Ethernet connection to a host PC. The host PC uses this connection to communicate with JTAG test access port (TAP) controllers in order to configure and program the Tomahawk2.

On the host PC, a Python library implements routines for communication with the Tomahawk2 board and provides a high level interface. This way, it is possible to interact with the Tomahawk2 from within a Python script or interactively from within a Python shell. By writing a Python script, chip configuration and programming can be automated. Such a script is also part of the Tomahawk2 backend in MAPS.

A Tomahawk2 application consists of a total of ten binaries—one binary for the Application Processor, one for the Core Manager, and one binary each for each of the eight PEs. The code generator creates code for each target core individually. As the backend does not use the Application Processor, it only generates a placeholder.

The code generated by cpn-cc can be compiled using Tensilica's XTensa C compiler [50]. A Makefile automates this process. It links all binaries against the channel library and incorporates initialization code. The Makefile also automates the execution of compiled KPN applications on the Tomahawk2. It uploads all binaries and the Python script to the host PC. The Python script configures the Tomahawk2 chip, loads binaries to scratchpad memories, and starts application execution. In order to provide basic debugging functionality, the script also interprets a set of opcodes that an application can send via the Tomahawk's FPGA interface.

To allow for application specific configuration and initialization as well as for automated evaluation of results, the user may provide an additional Python script per application that defines two functions. The function `pre_run` is called during initialization and, for instance, may be used to initialize certain memory areas with input stimuli. The function `post_run` is called after the application terminates. This can be used to read, evaluate, and print output data.

3.4.2 Code Generation

In MAPS, code generation is handled by *cpn-cc*. As discussed in Section 2.3.4 on page 13, cpn-cc is based on clang and implements a set of generic and target specific AST transformations. For code generation, cpn-cc selects AST transformations according to the target platform and applies them to the source code. The final transformation, which generates calls to the channel library interface, is highly target dependent and, therefore, needs to be rewritten for each new target. To reduce the implementation overhead, cpn-cc provides a set of configurable classes that handle code generation. However, their flexibility is limited and therefore a partial rewrite may still be required.

The code generator creates code for a single target core. When generating code for a Tomahawk2 PE, the code generator only includes code for processes that are mapped to this PE. As the Tomahawk2 backend only allows one process per PE, the generated code either implements exactly one process or is empty.

When cpn-cc transforms a process for usage on the Tomahawk2, it adds variable declarations for all required resources to the output code. For each channel the process uses, the code generator creates a global declaration for the channel data structure. The code generator checks if the channel is mapped to the local scratchpad or to a remote location. If the channel is mapped locally, the generator adds a declaration for the circular buffer. Otherwise, it adds a declaration for the temporary storage that is required for DMA transfers.

For each PE that has a process mapped to it, cpn-cc inserts initialization code before the actual process code. It adds calls to the *create* and *open* functions of the channel library. In order to exchange the location of channel data structures between consumer and producer, *create* writes channel information to global RAM and *open* retrieves this information. If the information in global RAM is not yet written by a corresponding *create* operation, *open* waits until the data is valid. Section 3.4.3 on the following page explains this mechanism of the channel library in more detail. It is important for code generation that the *create* calls for all channels of one process are placed before the *open* calls. Otherwise processes could deadlock.

The code generator transforms CPN channel accesses, by replacing them with calls to the channel library's *read* and *write* functions. Although, this mechanism is not different for the Tomahawk2 than for other targets, the specific function parameters used in the Tomahawk2 backend make it difficult to reuse

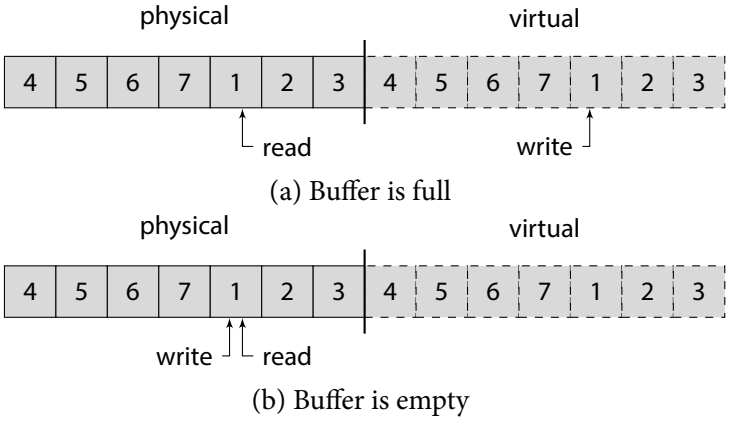
existing code in cpn-cc. Therefore, parts of the channel access transformations where rewritten for this work.

3.4.3 *Channel Library*

Although the backend design as presented in Section 3.3 on page 25 is straightforward, the channel library implementation presents a few challenges. These are the full/empty detection for circular buffers, the initial exchange of information between producer and consumer process, the exchange of read and write pointer during execution, and the data transfer via DMA with eight byte granularity.

When using a circular buffer, it is impossible to decide whether the buffer is full or empty just based on the values of read and write pointer. The read pointer points to the next item that can be read and the write pointer points to the next free slot. In both cases, when the buffer is empty and when the buffer is full, the read and write pointer are identical. Without additional information, it is not possible to determine if the buffer is full or empty. A common solution to this problem is to always keep one free slot. However, this may waste an unacceptable amount of memory for large token sizes. Another common approach is the usage of an additional flag that indicates if the buffer is full or empty. Nevertheless, this flag needs to be evaluated and updated by both the consumer process and the producer process. In order to avoid race conditions, access to this flag needs to be mutual exclusive and, therefore, requires locks. This makes the approach not suitable for the Tomahawk2.

The implementation presented here uses a mirroring technique to distinguish full and empty states [14]. Figure 3.2 on the facing page illustrates this approach. The physical buffer is extended by a virtual buffer that mirrors the content of the physical buffer. Read and write pointer store virtual addresses and need to be translated to physical addresses when accessing tokens. If read and write pointer point to the same virtual address, the buffer is empty. If both pointers point to different virtual addresses but both translate to the same physical address, the buffer is full. As discussed in Section 3.3, address calculation is an uncritical operation and therefore this solution does not introduce any need for mutual exclusion. Unlike other solutions, buffer mirroring also avoids the need for any additional memory. However, this comes at the cost of additional



(a) Buffer is full

(b) Buffer is empty

FIGURE 3.2: The channel library uses buffer mirroring for full/empty detection.

logic for address translation. Nevertheless, address translation breaks down to a conditional subtraction.

Channel parameters like channel and token size are already known in the mapping phase. The code generator respects these parameters when declaring buffers and passes them as arguments to the generated *create* and *open* calls. Moreover, each process knows the location of all its local channel buffers and channel information structures at compile time. However, a process neither knows the location of a buffer that is not stored locally nor does it know where its partner process stores its read or write pointer. This is because the code is generated specifically for each PE and compiled separately. In order to derive the position of buffers that are not stored locally, another linker phase that considers all binaries would be required. However, the effort for writing such a linker would be too large for this work. Therefore, the locations of buffers as well as read and write pointers need to be exchanged at runtime.

In this implementation, a reserved region in global RAM is used for exchange of information during initialization. Figure 3.3 on the next page illustrates the pointers involved in this mechanism. The *create* call of the channel library is responsible for propagating information to the RAM. If the process executing *create* is a producer, it announces the address of its read pointer. Otherwise, it announces the address of its write pointer. If the circular buffer is stored local to the calling process, it also announces the buffer's base address. If the buffer is mapped to global RAM, it is the Core Manager's responsibility to allocate a memory area and to announce the base pointer. For that, the code

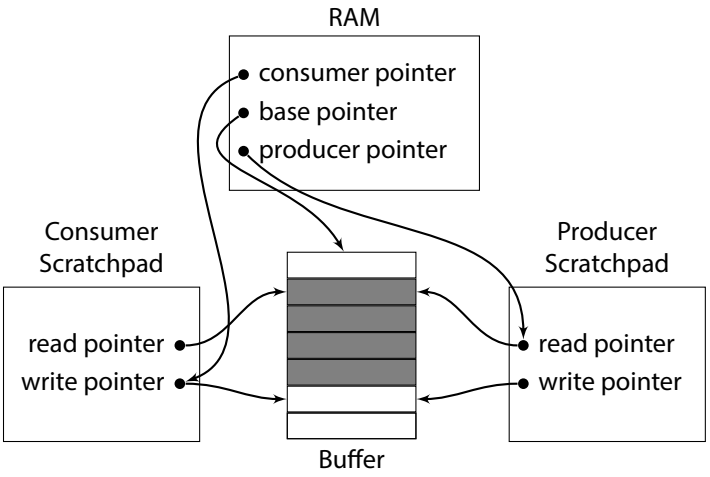


FIGURE 3.3: During initialization, both producer and consumer setup pointers at a known location in global RAM. During execution, these pointers are used to retrieve the position of the circular buffer and to update read or write pointer of the communication partner.

generator also adds *create* calls to the Core Manager's code. The Core Manager then reserves part of a predefined memory area and sets the base pointer accordingly.

A call to the *open* function of the channel library retrieves the information that was previously propagated by a corresponding *create* call. When the *open* function terminates, the process has all information that it requires for handling the opened channel.

In order to keep a consistent state, the two processes using a channel do not only need to exchange data during initialization, they also need to synchronize both read and write pointer during program execution. For that, this implementation uses a simple push strategy. A producer always updates the write pointer of the corresponding consumer after a write operation and a consumer always updates the read pointer of the corresponding producer after a read operation.

An earlier implementation used a lazy pull strategy. A process that wants to check the ring buffer state updates its read or write pointer by reading the corresponding pointer from its communication partner. The pointers do not need to be updated on every operation. For example, if a consumer knows

that there are three tokens in the buffer, it can safely consume the three tokens before updating its write pointer. If a consumer finds the buffer to be empty or a producer finds the buffer to be full, it blocks and permanently pulls the corresponding pointer until its value changes.

The lazy pull strategy has two disadvantages. Firstly, the strategy creates traffic in the NoC when a process blocks on an empty or full buffer. Secondly, the laziness of the strategy makes it difficult to predict execution time of a channel operation. It is not easily predictable when and how often a pointer gets updated. This depends on the buffer size as well as the read and write rates. Therefore, the implementation discussed here uses the pull strategy.

The DMA units on the Tomahawk2 operate with a granularity of eight bytes. Consequently, all data blocks accessed via DMA need to be eight byte aligned. This is uncritical when reading a token from a remote location. The library can simply transfer a properly aligned data block that contains the requested token. However, writing tokens of sizes that are not a multiple of eight bytes leads to problems. Firstly, transfer sizes need to be adjusted to the next multiple of eight bytes. Thereby, we need to ensure that the inserted padding bytes do not overwrite data in the circular buffer. Secondly, it is not easily possible to directly write tokens via DMA as for token sizes that are not a multiple of eight bytes not all tokens are properly aligned.

A simple solution could be to adjust the token size to the next greater multiple of eight. In MAPS, this could be done in an early phase, e.g. before mapping. However, this would require to change the channel type itself in the generated code, which is not easily possible with the current implementation of cpn-cc. Another solution could be to group multiple tokens in packets so that the packet size is a multiple of eight. However, this could introduce artificial deadlocks and, therefore, is not suitable as a general solution.

Figure 3.4 on the following page illustrates the padding mechanism used for this work. This mechanism fits tokens into frames. The frame size is the smallest multiple of eight bytes that is larger than or equal to the token size. When writing a token to a remote buffer, the whole frame gets transferred. Thereby, the frame is written to an eight byte aligned address and the token needs to be properly aligned within the frame. For instance, when the write pointer is `0x14` the frame needs to be written to the address `0x10` and the token has a four byte offset within the frame. According to the position and the size of the token, the frame needs to be filled with padding bytes.

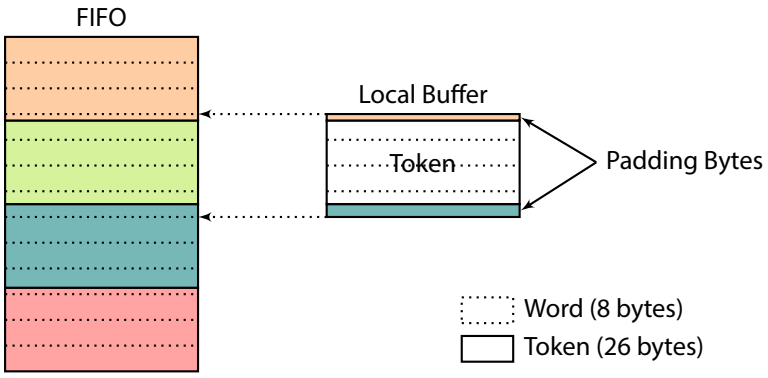


FIGURE 3.4: Due to the eight byte granularity of the Tomahawk2 NoC, writing tokens that have a size that is not a multiple of eight bytes requires padding bytes. The library writes a larger frame and has to fill in the correct padding bytes in order to avoid overwriting valid data.

To ensure that no valid data is overwritten when writing a frame to a buffer, the padding bytes need to be identical with the surrounding bytes in the ring buffer. When a process opens a write window on a channel that is not stored locally, the library calculates the position of the new token in the ring buffer and creates an empty frame in the local temporary storage. If padding bytes in front of the new token are required, the library reads the corresponding word from the ring buffer to the local temporary storage. The same holds for padding bytes that fill in the frame after a token. This ensures, that the bytes surrounding the new token in the temporary storage are equal to the bytes in the actual ring buffer.

In order to fill the new token with data, the producer process gets a pointer to the token in the local storage. As the process operates on local memory, addresses do not need to be aligned. The process only overwrites the token, but leaves the padding bytes untouched. When the process finishes writing the new token, it closes the write window. Then the channel library transfers the whole frame to the proper location within the ring buffer. Thereby, only the actual token changes although the whole frame is overwritten.

Padding a token with data from the buffer comes with a large overhead. Besides the computational overhead, a write operation requires up to two additional data transfers. Therefore, token sizes that are not a multiple of eight bytes should be avoided on the Tomahawk2.

3.5 Conclusion

This chapter introduced a Tomahawk2 backend for the MAPS compiler framework. Both the presented design and implementation respect characteristics of the Tomahawk2 architecture and keep the required memory and processing overhead low. The presented backend allows the MAPS compiler framework to generate realizations of arbitrary KPN applications for the Tomahawk2. However, input application are restricted to a maximum of eight processes. In order to achieve efficient realizations, it is recommended to only use channels with token sizes that are a multiple of eight bytes.

4

Hardware Model

The previous chapter introduced a Tomahawk2 backend for the MAPS compiler framework. This backend is capable of creating realizations of CPN applications for the Tomahawk2 platform. The resulting Tomahawk2 applications can be compiled and executed. However, so far MAPS has no understanding of the characteristics of the platform.

In MAPS, an accurate hardware model is the basis for determination of efficient mappings and for performance estimation. Only with a good understanding of the characteristics of all available hardware components, the mapper can estimate the influence of a mapping decision on the resulting application performance. Therefore, this chapter introduces a hardware model for the Tomahawk2.

MAPS' hardware model comprises a processor model and a communication model. The processor model represents an ISA and is used for prediction of execution times of code segments. The communication model represents the interconnection of all hardware resources and incorporates a cost model for estimation of communication times via a certain connection. With the current interconnect model of MAPS it is not possible to accurately represent NoC architectures. Therefore, an important part of this work is the introduction of a NoC model for MAPS.

Section 4.1 discusses the current communication model of MAPS in detail and introduces changes to the cost model that are required for accurate modelling of the Tomahawk2 architecture. Section 4.2 introduces the new NoC model. In order to create a communication model specifically for the Tomahawk2, we need an understanding of the platform's characteristics. Therefore, Section 4.3 presents a set of benchmarks that measure bandwidth limitations of the platform and communication costs of the channel library. Based on these

measurements, Section 4.4 formulates a Tomahawk2 communication model. Section 4.5 proposes an extension to this model that also considers network congestion. Section 4.6 discusses the processor model of MAPS and its limitations with regard to the Tomahawk2. Finally, Section 4.7 concludes the chapter.

4.1 Communication Model

The communication model of MAPS comprises an interconnect model and a cost model. The interconnect model defines communication resources like local and shared memories. The cost model uses so called *communication primitives* to abstract KPN channel accesses. Communication primitives model a software API that implements the semantics of KPN channels on the target platform. In the case of the Tomahawk2, this is the channel library as described in Section 3.4.3.

4.1.1 *Communication Primitves*

Castrillón defines a communication primitive as a four-tuple CP [12].

$$\mathrm{CP} = \left(\mathrm{PE}_i, \mathrm{PE}_j, S \subseteq \mathcal{CR}, \mathcal{CM}^{\mathrm{CP}}\right) \tag{4.1}$$

This tuple expresses how a process running on PE_i can communicate with a process running on PE_j via a set of communication resources S. This set is a subset of $\mathcal{CR}$, the set of all communication resources that a platform provides. The cost model $\mathcal{CM}^{\mathrm{CP}}$ is a set of functions that map communication volume to a cost value.

In MAPS, the set of communication resources S is called a *hardware channel* and the interconnect model defines the set of all communication resources $\mathcal{CR}$. All possible hardware channels can be derived from the interconnect model. For a certain hardware channel, multiple communication primitives may exist. This allows for usage of multiple channel libraries on the same platform (e.g. one library using programmed I/O and one using DMA). The mapper may choose the communication primitive that fits best in a certain scenario.

In order to create a complete communication model, we need to define communication primitives for every possible combination of PE_i, PE_j, and S. As the number of required definitions may become large easily, MAPS provides

a shortcut in its architecture description. Instead of single communication primitives the architecture description defines groups of communication primitives of a certain type. Thereby, the type is predefined by MAPS and implicitly specifies which communication primitives of all possible communication primitives are included in the group. The cost model is only defined once per group and is identical for all communication primitives within the group. For instance, the *shared memory* type allows communication from all PEs to all PEs but limits the possible communication resources to shared memories. The *local memory* type only includes communication primitives with $PE_i = PE_j$ and limits the communication resources to local memories.

On the Tomahawk2, the mapper has three choices when placing channel buffers. The buffer could be placed in the scratchpad that is local to the consuming PE, in the scratchpad that is local to the producing PE, or in the global RAM[1]. The communication costs for consumer and producer vary depending on the buffer location. If the buffer is mapped to the global RAM, both consumer and producer have to perform data transfers. However, if the buffer is located in the consumer's scratchpad, only the producer has to perform data transfers. The consumer can operate directly on its local memory. Similar, if the buffer is located in the producer's scratchpad, only the consumer has to perform a data transfer and the producer can operate directly on the buffer. As the Tomahawk2 backend only allows one process per PE, direct communication between two processes running on the same PE is not possible.

The dependency of communication costs on the channel mapping needs to be respected by the communication model in order to accurately predict network traversal times. However, the types of communication primitives that MAPS defines do not consider the locality of buffers. Therefore, this work introduces three new types of communication primitives. These are *shared NoC memory*, *producer NoC memory*, and *consumer NoC memory*. Similar to *shared memory*, the *shared NoC memory* type allows communication from all PEs to all PEs via shared memories that are connected to the NoC. The *producer* and *consumer NoC memory* types allow communication via local memories from a PE to all other PEs. Depending on the type, the memory is either local to the producer or to the consumer. Using these new types of communication

[1] In general, the buffer could also be placed in the scratchpad of another PE. However, this case is not mandatory and is not incorporated in this work.

primitives, different cost models can be applied for each of the three different mapping scenarios.

4.1.2 *Cost Model*

The cost model $\mathcal{CM}^{CP}$ is a set of functions that map data volume to communication costs for a certain communication primitive CP. MAPS uses a cost model that consists of three functions:

$$\mathcal{CM}^{CP} = \left\{\mathcal{C}^{CP}(x), \mathcal{T}^{CP}(x), \mathcal{P}^{CP}(x)\right\} \tag{4.2}$$

All three functions map the communication volume x in bytes to a number of cycles. $\mathcal{C}^{CP}(x)$ defines the costs required for consuming a token on a KPN channel and $\mathcal{P}^{CP}(x)$ defines the costs required for producing a token. $\mathcal{T}^{CP}(x)$ models pure transfer costs. For instance, this is useful when modelling DMA transfers. Here, the processor could setup the DMA unit for handling a data transfer. Then the processor is free to perform other operations while the data transfer is handled by the DMA unit.

As the current cost model of MAPS models communication costs only in dependence of transfer volume, this model is not suitable for modelling NoC-based architectures. In a NoC-based architecture, communication costs not only depend on the communication volume but also on the route that a transfer takes. This work proposes an extended cost model that incorporates the number of hops on a NoC route as well as the bandwidth of the slowest link on this route (Equation 4.3). This extended cost model maps communication volume x, number of hops h, and bandwidth limit b to a number of cycles. In this work, the number of hops is the number of routers that a packet traverses on its way.

$$\mathcal{CM}^{CP} = \left\{\mathcal{C}^{CP}(x, b, h), \mathcal{T}^{CP}(x, b, h), \mathcal{P}^{CP}(x, b, h)\right\} \tag{4.3}$$

The proposed parameterization of the cost model allows for definition of communication costs orthogonal to the definition of the interconnect architecture. The dependency between communication costs and network topology can be completely modeled by the cost functions. This is a significant advantage of the extended cost model over MAPS' old cost model. In order to calculate

the communication costs for a certain transfer, the bandwidth limit and the number of hops need to be derived specifically for that transfer.

This cost model does not consider the network load and is thus not suitable for modeling network congestion. A cost model that considers network congestion could add an additional parameter that represents the network load. Then the cost functions could model network congestion depending on the load parameter. However, this work proposes an algorithmic approach as part of the communication model for modelling network congestion. Section 4.5 discusses this approach.

4.2 NoC Model

An interconnect model for NoC architectures should provide an understanding of the network topology and on how nodes in the network communicate. Using the interconnect model, it should be possible to derive the number of hops and the bandwidth limit for a specific transfer in order to parameterize the cost model in Equation 4.3. The model should be capable of modelling the NoC architecture of the Tomahawk2 but also be flexible enough to model other NoC architectures and topologies. Such a model is currently not part of MAPS. This section proposes a new interconnect model for internal representation of NoC-based architectures in MAPS.

In the new model, a NoC is a list of devices, routers, and links. Devices can be shared memories or PEs with a local memory. Links are unidirectional and either connect a device and a router or two routers with each other. Each link has a bandwidth that limits the maximum data rate that the link can handle. Routers operate according to an oblivious deterministic routing algorithm. Switching techniques and input buffers for router ports are not considered by the NoC model.

Using this simple model, it is possible to construct arbitrary network topologies. By modelling single links instead of whole topologies, the model can assign different bandwidths to different links. This is useful for modelling the Tomahawk2 where all components, including the routers, operate asynchronously and the clock frequencies can be adjusted individually for each component.

We can use this model in order to derive routes for arbitrary data transfers as all routers, all links, and the routing algorithm are known. A route simply is a list of links that need to be traversed when transferring data from device A to

```
<Interconnect>
  <Noc Name="noc_ic" GridReference="nocGrid" GridRouting="
    HorizontalThenVertical">
    <Masters List="PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7" />
    <Slaves List="pe0_mem pe1_mem pe2_mem pe3_mem pe4_mem pe5_mem
    pe6_mem pe7_mem global_mem"/>
  </Noc>
</Interconnect>

<NocGrid Name="nocGrid">
  <Mesh Bandwidth="10.15"> <!-- in bytes per cycle -->
    <Router X="0" Y="0">
      <SharedMemoryLink Memory="global_mem" Bandwidth="3.73"/>
    </Router>
    <Router X="0" Y="1">
      <ProcessorLink Bandwidth="7.99" Processor="PE4"/>
      <ProcessorLink Bandwidth="7.99" Processor="PE5"/>
      <ProcessorLink Bandwidth="7.99" Processor="PE6"/>
      <ProcessorLink Bandwidth="7.99" Processor="PE7"/>
    </Router>
    <Router X="1" Y="0">
      <ProcessorLink Bandwidth="7.99" Processor="PE0"/>
      <ProcessorLink Bandwidth="7.99" Processor="PE1"/>
      <ProcessorLink Bandwidth="7.99" Processor="PE2"/>
      <ProcessorLink Bandwidth="7.99" Processor="PE3"/>
    </Router>
    <Router X="1" Y="1">
    </Router>
  </Mesh>
</NocGrid>
```

Listing 4.1: An excerpt of the Tomahawk2 platform description for MAPS that illustrates the definition of the NoC-based interconnect. The NoC is implemented in a mesh topology consisting of four routers. The link bandwidth can be set individually for each device.

device B. Knowing the route, we can derive the number of hops and find the slowest link to derive the bandwidth limit as required by the cost model.

In order to support the definition of the NoC architecture in the platform description of MAPS, this work extends the existing XML description by new tags and attributes. For example, Listing 4.1 on the preceding page shows the XML description of the Tomahawk2 NoC. The `Interconnect` tag is already part of MAPS and defines the general interconnect architecture. In the case of the Tomahawk2, the interconnect is a NoC architecture consisting of eight processing elements, eight local memories, and one global memory.

The network topology is specified using the `NocGrid` tag that is labeled by a name and referenced to by the interconnect definition. This work provides the `Mesh` tag as a shortcut for definition of mesh topologies. Tags for other topologies or irregular structures are not provided by this work but can be implemented easily. The `Router` tag defines routers that are automatically connected according to the topology specified by the surrounding tag. The tags `SharedMemoryLink` and `ProcessorLink` define links that connect a router to a shared memory or to a processing element, respectively. By using the `Bandwidth` attribute, the maximum bandwidth can be set globally, per router, or per link. When MAPS creates its internal representation of the NoC model from the XML description, it sets the bandwidth of each link to the minimum value that applies to this link.

4.3 Measurements

The two previous sections discussed the interconnect model of MAPS and introduced extensions that add support for modelling communication in NoC architectures. In order to define a specific model for the Tomahawk2, we need an understanding of the characteristics of its NoC architecture and of communication costs induced by the channel library. Therefore, this section presents two benchmarks for measurement of communication costs (Section 4.3.1) and link bandwidths (Section 4.3.2).

4.3.1 *Communication Costs*

In order to formulate a cost model for communication on the Tomahawk2, we need an understanding of the timing characteristics of the channel library.

Therefore, this section presents experiments that measure the time required for performing produce and consume operations on KPN channels using the channel library as presented in Section 3.4.3. Each of the experiments covers one of three scenarios. These scenarios are read/write access to a buffer that is located in a remote scratchpad, to a buffer that is located in a local scratchpad, and to a buffer that is located in global RAM.

Experimental Setup

The experiments presented in this section measure the time that is required to perform channel accesses. In order to perform these measurements on-chip, a hardware timer is required. On the Tomahawk2 only the Core Manager implements a hardware timer. Therefore, the Core Manager needs to perform the time measurement. However, in order to examine scenarios that would occur in real applications, where two processes running on different processing elements communicate with each other, the process that performs the channel accesses in question needs to be running on one of the processing elements. Thus a mechanism for controlling time measurement from a processing element is required.

In the experiments presented in this section, processing elements may send *start* and *finish* signals to the Core Manager. When the Core Manager receives the *start* signal, it starts its timer and then waits for the *finish* signal. After receiving the finish signal the Core Manager stores its current timer value and the measurement is complete. The transfer of *start* and *finish* signals introduces an additional delay. However, the delay is approximately equal for both signals. Consequently, the delay gets compensated as it adds to the absolute time of start and finish but not to the time difference between both points.

All the experiments below are implemented as CPN applications where one of the processes performs the measurement. This process under observation sends the *start* signal to the Core Manager, performs the operation in question repeatedly, and then sends the *finish* signal to the Core Manager. The experiments presented here repeat the operation in question 1,000 times in order to derive an average value and thus filter out network jitter.

The experiments use channels with buffers bounded to six tokens. This is the size MAPS uses for most channels in real applications as it is the initial value of the bounding algorithm. A size of six tokens is small enough to fit buffers with a token size up to 4,096 bytes into the Tomahawk2 scratchpad memories.

In order to understand the influence of data volume and number of hops on the communication costs, we need to perform measurements while varying these parameters. The experiments presented here use different mappings in order to vary the number of hops and change the token size within a range from 8 bytes to 4,096 bytes.

The experiments only cover token sizes that are a multiple of 8 bytes. As explained in Section 3.4.3 on page 30, token sizes that are not a multiple of 8 bytes need additional transfers to guarantee consistency. This increases the execution time of channel access operations dependent on a token's position in memory. To keep the cost model simple, it only covers token sizes that are a multiple of 8 bytes. However, in general it would be possible to derive cost functions for arbitrary token sizes.

Experiment 1: Remote Access

The first experiment analyzes channel accesses for buffers that are located in a remote scratchpad. It is based on a simple KPN application that consists of a source process and a sink process connected by a single channel. The processes may be mapped to neighbouring processing elements (e.g. PE0 and PE1), resulting in a distance of one hop, or to processing elements with a three hop distance (e.g. PE0 and PE4). Direct communication between two PEs over a distance of two hops is not possible on the Tomahawk2.

In order to measure the execution time of a remote produce operation, the channel buffer needs to be mapped to the consumer scratchpad. Consequently, to measure the execution time of a remote produce operation, the channel buffer needs to be mapped to the producer scratchpad. Figure 4.1a and Figure 4.1b on the next page illustrate this setup.

Figure 4.2 on the following page displays an excerpt of the measured execution times for remote channel accesses. In order to keep the diagram clear, it only show shows every fourth data point for a range from 8 to 2,048 bytes. As the data points clearly show a linear trend, the diagram also displays a linear function derived from linear regression over all data points.

In the diagram in Figure 4.2, there is no distinction between communication over a one hop distance and over a three hop distance for the remote produce operation. The diagram only displays the values measured for a one hop distance. However, the values measured for a three hop distance are approx-

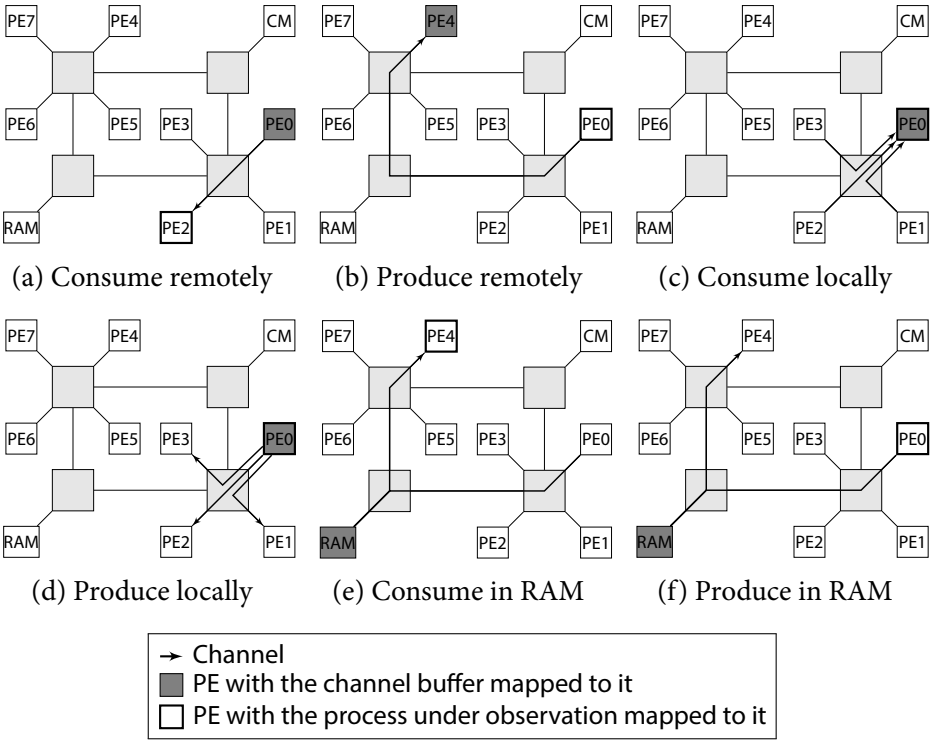


FIGURE 4.1: These experimental setups are used for measurement of communication costs. Each setup applies a different mapping.

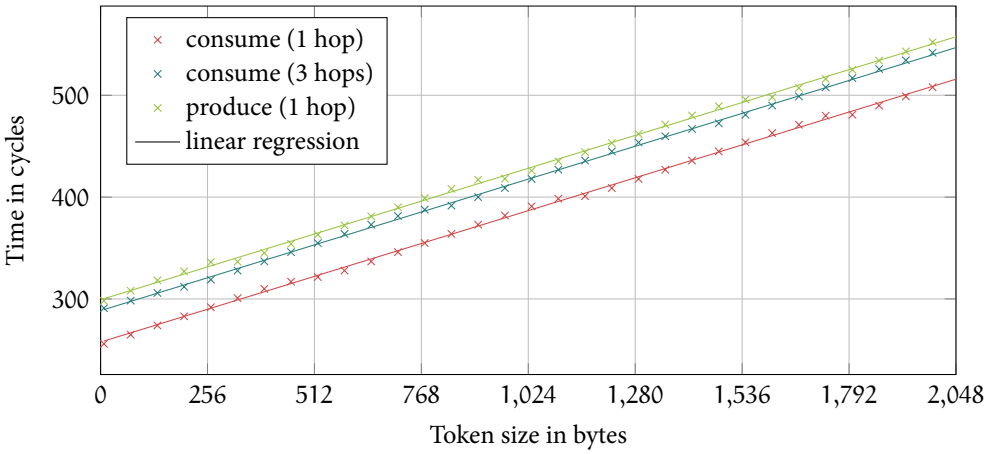


FIGURE 4.2: The measured communication costs for remote channel accesses clearly show a linear trend.

imately identical to the displayed values. The deviation between both values is below 0.02% and can be explained by measurement error.

In fact, the execution time of a remote produce operation does not depend on the number of hops as the DMA unit does not wait for an acknowledge signal when it sends data. The DMA unit simply writes out all data word by word and then the NoC handles the transfer. Therefore, the costs for a remote produce operation only depend on the available bandwidth and the data volume. In contrast, the consume operation reads data via DMA and thus has to wait for a reply. Therefore, the costs for a remote consume operation depend on the number of hops.

Experiment 2: Local Access

The second experiment analyzes channel accesses for buffers that are located in the local scratchpad. The execution times for local produce and consume operations are constant values that are independent of token size and number of hops. This is because a channel access operation that operates locally does not need to transfer any user data. The operation only has to perform one write transfer in order to update the read or write pointer of the communication partner. As this is a DMA write operation, its execution time is also independent of the number of hops.

The experiment uses a setup consisting of four processes and three channels. The process under observation is producer or consumer to all three channels. The other three processes each connect to one of the channels. Figure 4.1c and Figure 4.1d on the preceding page visualize this setup. The process under observation produces or consumes tokens on all three channels in a round-robin pattern. As the communication costs are independent of the data volume for local channel accesses, the experiment uses a fixed token size of 8 bytes.

The usage of three parallel channels ensures that the process under observation always can perform its read or write operation and never has to wait for another process to fill or empty the channel buffer. As local channel accesses do not have to transfer the token, they are always faster than remote operations. However, a remote channel access for a token size of 8 bytes does not need considerably more time than a local access. The process under observation performs three local channel accesses per iteration while the other three processes only perform one remote channel access per iteration. This mechanism ensures

that the process under observation never has to wait and that we only measure the time required for performing a local channel access.

As discussed before, the experiment measures the time over 1,000 iterations. This leads to a total of 3,000 channel access operations executed by the process under observation. By dividing the total measured time by 3,000, we get the average execution time of the operation in question. The experiment results in an average execution time of 164 cycles for a local consume operation and 205 cycles for a local produce operation. Performing the experiment with a mapping that spans the three channels over a one hop distance and over a three hop distance leads to the same results. This verifies that local channel accesses are in fact independent of the number of hops.

Experiment 3: RAM Access

The third experiment analyzes channel accesses for buffers that are located in global RAM. Here, both producer and consumer issue a data transfer to or from RAM. Therefore, the execution time of both operations depends on the transfer size. However, both operations are independent of the number of hops as all PEs have the same distance of two hops to the memory controller. As discussed before, updating read and write pointer is also independent of the number of hops.

The experiment uses an application with a structure that is similar to the application used in the first experiment. Two processes communicate via a single channel. This channel is mapped to the global RAM. Consumer and producer process may be mapped to neighbouring PEs or two PEs with a three hop distance. Figure 4.1e and Figure 4.1f on page 46 show the setup used for this experiment.

Both consumer and producer process perform data transfers from or to the RAM. In order to get accurate results from the time measurement for single operations, we need to ensure that the two processes do not issue interleaving data transfers. Therefore, this experiment uses a modified version of the channel library that implements a mechanism for deactivating the data transfer in produce or consume operations via preprocessor macros. This way, the observed operation can operate normally while the communication partner only performs maintenance tasks (e.g. read and write pointer management). This setup guarantees that the process under observation is the only process that performs data transfers to or from RAM.

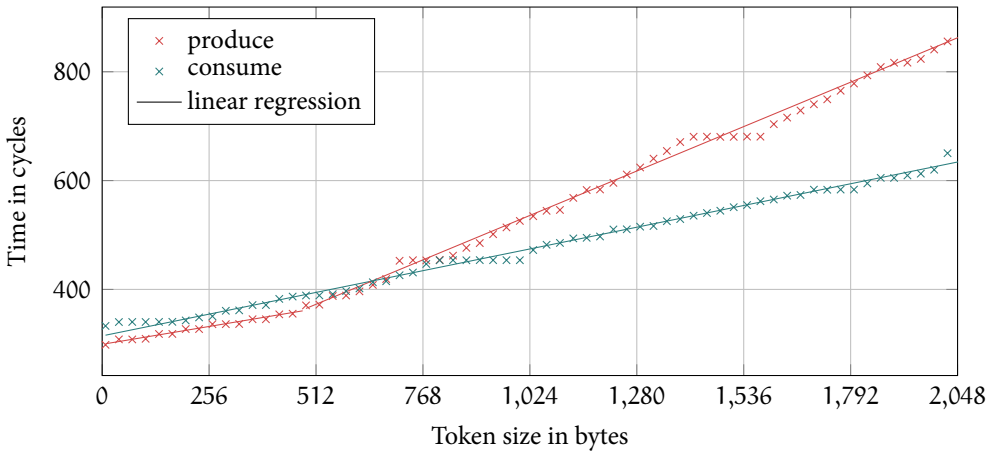
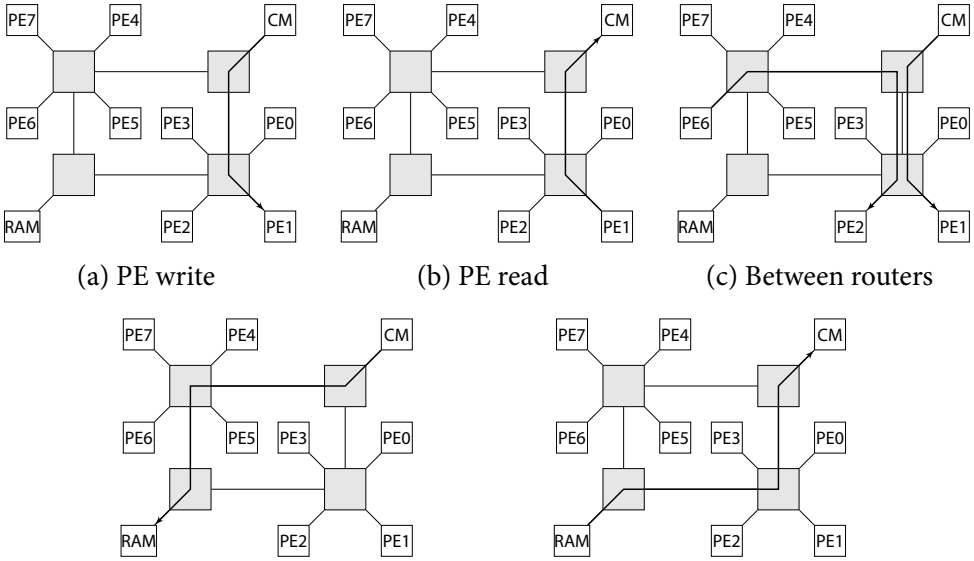


FIGURE 4.3: The measured communication costs for RAM channel accesses do not have a clear linear trend. Nevertheless, the costs are modeled with linear functions.

Figure 4.3 shows the measured execution times of consume and produce operations for token sizes in a range from 8 to 2,048 bytes. Again the diagram only displays every fourth data point. The diagram further shows a linear regression for both operations. As the data points for the produce operation show a sharp bend at about 480 bytes, the communication costs for the produce operation are modeled with two linear functions. However, the diagram shows various steps. Therefore, linear functions are not suitable for accurately modeling the observed behaviour. Nevertheless, at this point a simple model is favourable over a complex model as we do not know yet how accurate the overall communication model will be. In fact, evaluation in Section 5.2 will show that the overall communication model of MAPS cannot accurately model RAM accesses on the Tomahawk2 and thus accuracy of the cost functions is not a concern.

4.3.2 *NoC Bandwidths*

In order to parameterize the NoC model as presented in Section 4.2, we need to measure the bandwidth that each links provides. There are five different types of links on the Tomahawk2: links that connect two routers with each other, links that connect from a PE to a router, links that connect from a router to



(a) PE write (b) PE read (c) Between routers

(d) RAM write (e) RAM read

FIGURE 4.4: The experimental setups for measuring the link bandwidth of all five types of links on the Tomahawk2

a PE, a link that connects from the RAM interface to a router, and a link that connects from a router to the RAM interface. For a complete measurement we need to measure the bandwidth of all five types of links.

In order to measure the bandwidth of a link, we can issue a DMA transfer with a known data volume and measure the time that is required for performing the transfer. With the measured value, we can calculate the data rate of the transfer. This data rate is the bandwidth of the slowest link that was traversed by the transfer.

On the Tomahawk2 the Core Manager is the only component that implements a hardware timer. Therefore, the Core Manager performs the time measurement in this experiment. It issues DMA transfers with a data volume of 8,192 bytes and measures the time until the DMA unit finishes the transfer. As the Core Manager uses an architecture that is similar to the architecture of the processing elements and also implements a similar DMA unit, we can apply the values measured on the Core Manager to the processing elements.

From the Tomahawk2 architecture description [38] we know that the links between routers are configured to higher bandwidth than the links connecting

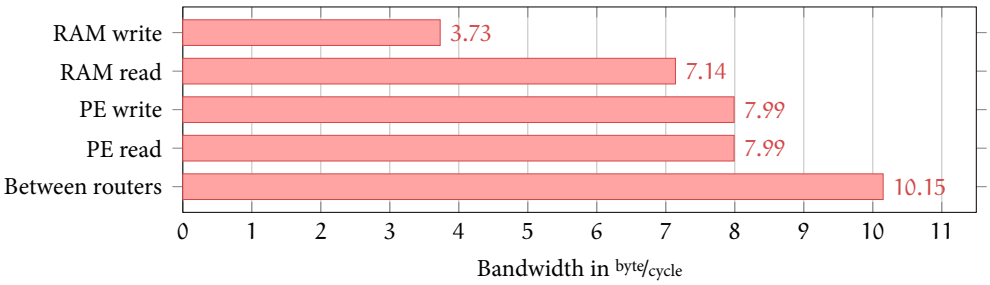


FIGURE 4.5: The measured bandwidths for all five types of links on the Tomahawk2

to a PE or to RAM. By reading/writing from/to RAM or from/to a PE, we can measure the bandwidth of the link that connects this component as this would be the slowest link. However, this setup does not allow for measurement of the bandwidth of links that connect two routers as they are always faster than the link that the Core Manager uses to receive or transmit data.

To fully utilize a link between two routers, another parallel transfer besides the one issued by the Core Manager is required. This additional transfer is issued by a PE that operates independently. As the routers on the Tomahawk2 forward packets in round-robin fashion when two or more packets compete for the same output port, both transfers get the same data rate, which is half of the available bandwidth. Figure 4.4 illustrates the setups used for bandwidth measurement of the different link types.

Figure 4.5 displays the bandwidths as measured using the setup that was discussed above. In order to filter out network jitter, Figure 4.5 displays mean values over 1,000 measurements. The measured bandwidths are the foundation for the formulation of a Tomahawk2 NoC model.

4.4 Model Formulation

With the information gained from measurements in the previous section, we can formulate a communication model for the Tomahawk2 that uses the cost model and interconnect model discussed in Sections 4.1 and 4.2. The model that this section presents, only considers token sizes that are a multiple of 8 bytes. Communication costs for other token sizes would need to considered separately.

In order to model communication cost on the Tomahawk2, three types of communication primitives are required. The *Shared NoC memory* primitive models communication via global RAM, the *producer NoC memory* primitive models communication via a scratchpad that is local to the producer process, and the *consumer NoC memory* primitive models communication via the scratchpad that is local to the consumer process.

As discussed in Section 4.1 the cost model $\mathcal{CM}^{CP}$ is a set of three functions $\mathcal{C}^{CP}$, $\mathcal{T}^{CP}$, and $\mathcal{P}^{CP}$ that model the costs for consuming, transferring, and producing a token using the communication primitive CP. In the proposed cost model, the functions are parameterized by the data volume x, the bandwidth limit b, and the number of hops h. In order to create the Tomahawk2 communication model, we need to define the cost models for the shared NoC memory primitives $\mathcal{CM}^{R}$, the producer NoC memory primitives $\mathcal{CM}^{P}$, and for the consumer NoC memory primitives $\mathcal{CM}^{C}$.

The transfer cost function $\mathcal{T}$ models transfer costs that occur in addition to the costs for processing consume and produce operations. This is useful for modeling DMA transfers, where a produce or consume operation issues a DMA transfer and the operation terminates while the data transfer is still ongoing. However, this is not needed for the Tomahawk2. Although the channel library as presented in Section 3.4.3 uses DMA transfers, there are no transfer costs in terms of the communication model. A produce or consume operation always waits until the data transfer is complete in order to ensure consistency at all time. Therefore, we can set the transfer costs to zero.

$$\mathcal{T}^{R}(x, b, h) = \mathcal{T}^{P}(x, b, h) = \mathcal{T}^{C}(x, b, h) = 0 \tag{4.4}$$

The cost functions for produce and consume operations can be derived from the measurements discussed in Section 4.3.1. As the costs for local consume and produce operations are constant, we can directly define $\mathcal{P}^{P}$ and $\mathcal{C}^{C}$ using the measured values.

$$\mathcal{C}^{C}(x, b, h) = 164 \tag{4.5}$$

$$\mathcal{P}^{P}(x, b, h) = 205 \tag{4.6}$$

From linear regression analysis in Figure 4.2 on page 46 over the data measured for remote channel accesses we get the following equations.

$$\mathcal{C}^{P}_{h=1}(x) = 258 + 0.1256 \cdot x \tag{4.7}$$
$$\mathcal{C}^{P}_{h=3}(x) = 289 + 0.1256 \cdot x \tag{4.8}$$
$$\mathcal{P}^{C}(x) = 299 + 0.1256 \cdot x \tag{4.9}$$

On the Tomahawk2, the number of hops between two PEs is always one or three. The topology does not allow for other values. As the cost for consuming a token remotely depends on the number of hops, we get two equations for $\mathcal{C}^{P}$. However, in order to define the cost model, we need to generalize $\mathcal{C}^{P}$ so that it models communication costs for arbitrary topologies. Under the assumption that the number of hops has a linear influence on communication costs, we can generalize $\mathcal{C}^{P}$ and get the following equation.

$$\mathcal{C}^{P}(x, h) = \mathcal{C}^{P}_{h=1}(x) + \frac{\mathcal{C}^{P}_{h=3}(x) - \mathcal{C}^{P}_{h=1}(x)}{3-1} \cdot (h-1) \tag{4.10}$$
$$= 242.5 + 15.5 \cdot h + 0.1256 \cdot x \tag{4.11}$$

The slope in Equations 4.9 and 4.11 is the inverse data rate for remote produce and consume operation. As the slope of 0.1256 approximately corresponds to the bandwidth of 7.99 $^{\text{byte}}/_{\text{cycle}}$ measured in Section 4.3.2 for reading from and writing to a PE's scratchpad, we can assume that the link bandwidth limits the data rate of the transfer. Therefore, we can generalize Equations 4.9 and 4.11 in order to derive the final cost functions for remote channel accesses.

$$\mathcal{C}^{P}(x, b, h) = 242.5 + 15.5 \cdot h + \frac{x}{b} \tag{4.12}$$
$$\mathcal{P}^{C}(x, b, h) = 299 + \frac{x}{b} \tag{4.13}$$

From linear regression analysis in Figure 4.3 on page 49 over the communication costs measured for channel buffers that are located in global RAM we get the following equations.

$$\mathcal{C}^{R}_{h=2}(x) = 314 + 0.156 \cdot x \tag{4.14}$$

$$\mathcal{P}^{R}_{h=2}(x) = \begin{cases} 299 + 0.128 \cdot x & x \leqslant 480 \\ 209 + 0.319 \cdot x & x > 480 \end{cases} \tag{4.15}$$

On the Tomahawk2, the costs for RAM accesses are identical for all PES as all PEs have a two hop distance to the memory controller. However, the costs are generally not independent of the number of hops and could vary for other topologies. Nevertheless, we have no information on the relation between the number of hops and communication costs for RAM accesses. Therefore, we cannot generalize the equations for an arbitrary number of hops. We could only assume that the relation between number of hops and communication cost for RAM accesses is similar to the relation in Equation 4.12.

The diagram in Figure 4.3 on page 49 clearly shows that the data rate of RAM accesses is not constant but changes in relation to the data volume. The alternating data rate cannot be modeled by a bandwidth limit as this would lead to a constant data rate. An alternative cost model could introduce a service delay in dependence of the token size in order to model the behaviour of the memory controller. However, for now Equations 4.14 and 4.15 are left unchanged and the dependence of communication costs on the number of hops as well as the bandwidth limit are not modeled. Section 5.2 will review this decision.

$$\mathcal{C}^{R}(x, b, h) = 314 + 0.156 \cdot x \tag{4.16}$$

$$\mathcal{P}^{R}(x, b, h) = \begin{cases} 299 + 0.128 \cdot x & x \leqslant 480 \\ 209 + 0.319 \cdot x & x > 480 \end{cases} \tag{4.17}$$

On the whole, the cost model for the Tomahawk2 is defined by the Equations 4.4, 4.5, 4.6, 4.12, 4.13, 4.16, and 4.17.

Besides the cost model, we also need to define the NoC model for the Tomahawk2. The network topology was already discussed in Section 3.1. In order to define the bandwidth of all links, we can use the values measured in Section 4.3.2. Figure 4.6 on the facing page visualizes the resulting model.

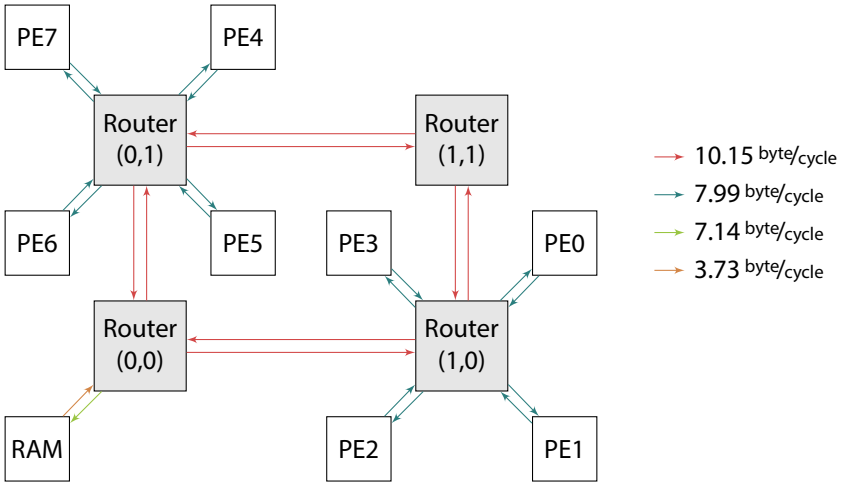


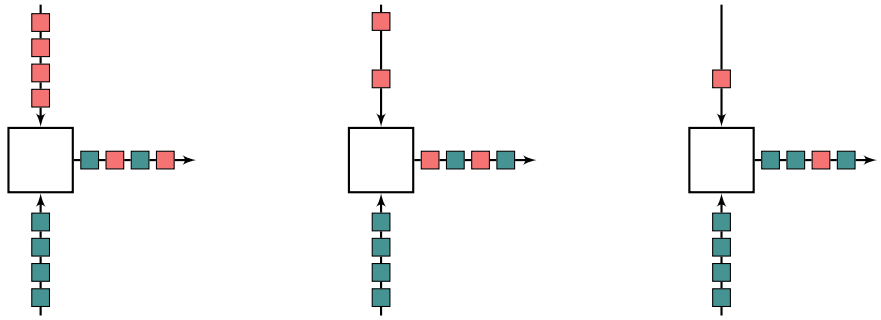
FIGURE 4.6: NoC model for the Tomahawk2 with defined link bandwidths

4.5 Network Congestion

The NoC model as presented in Section 4.2 has a static view on the network. It has no notion of state in the NoC (e.g. buffer utilization or ongoing transfers). However, in a real NoC the state has an influence on transport times. Multiple transfers may compete for the same resources. Contention between multiple transfers may lead to congestion and, therefore, to slower data transfers. An accurate NoC model should also be capable of modeling network congestion. A NoC simulator could be used to derive accurate communication costs in all scenarios but would also increase complexity drastically. Therefore, an analytic approach for calculation of delays in a congested network is desirable.

Most analytical NoC models like those based on network calculus [17], [18] or Poisson processes [39] are designed for evaluation of NoC architectures, but cannot predict transfer times in a certain network state. Dasari, Nikolić, Nélis, *et al.* proposed an algorithm that calculates tight worst case traversal times in a NoC for a known load scenario [21]. This approach is based on a contention-tree [35] that models the influence that multiple ongoing transfers may have on each other. However, the algorithm operates recursively on the tree which can lead to long computation times [21].

In order to keep the implementation expenses and the computational effort required for modeling network congestion low, this work proposes a simple



(a) Two incoming flows that both use the full bandwidth

(b) One incoming flow uses half the available bandwidth

(c) One incoming flow uses a fourth of the available bandwidth

FIGURE 4.7: Contention between two flows can lead to congestion. The slowdown is larger for flows with high data rate and smaller or for flows with low data rate.

algorithm that models congestion in the Tomahawk's NoC. This algorithm models the behaviour of transfers competing for the same output port of a router. In the following, a single data transfer from A to B in the NoC is called a *flow*. A flow transfers a known number of bytes x in a certain time t from A to B. The average data rate of this flow is $\bar{d} = x/t$.

When two or more flows compete for the same output port, the routers on the Tomahawk2 forward packets in a round-robin scheme. When n flows with a high initial data rate d compete for the same output port of a router, all flows get an equal share of the bandwidth b that the target link provides. The resulting average bandwidth for each of the transfers from source to destination is $\bar{d} = b/n$. If one of the flows has a lower initial data rate of d/n, the resulting average data rate for each of the n flows still is $\bar{d} = b/n$. The reason for this behaviour is that the router uses round-robin to decide which packet gets forwarded. As long as all flows send at a data rate of at least b/n, all flows get served by the router with a rate of b/n. From this observation we can conclude, that the slowdown from network congestion is larger for flows that send data at a high rate and is smaller for flows that send data at a low rate. In fact, a flow that sends data at a rate that is less than b/n does not suffer from congestion and is not slowed down by the network. Figure 4.7 illustrates the partitioning of the available bandwidth on an outgoing link for two competing flows.

In order to simulate congestion in a NoC, the NoC model needs to be extended in order to keep track of the network state. The approach presented here extends the NoC model by a list of all flows that are currently active in the NoC. This way, we can model arbitrary combinations of flows that may or may not use the same resources. For each active flow, we know the data volume and can derive the communication cost without congestion from the cost model. This gives us the initial average data rate of each flow.

In order to calculate the additional delay caused by congestion for each active flow, we iterate over all links in the network and use the following algorithm for each link.

1. Sum up the data rate of all flows traversing the link.
2. If the total data rate is equal to or less than the bandwidth of the link, the algorithm terminates. Otherwise, it continues with step 3.
3. Find the flow with the highest data rate that traverses the link and increment its communication costs. This reduces the data rate of the flow.
4. Repeat from step 1.

For this algorithm to work correctly, the order of links that the algorithm analyzes is important. When a flow traverses multiple links, it can first compete with one flow in a link l_1 and later on compete with another flow in a link l_2. As the flow's data rate may be slowed down by congestion in l_1 before reaching l_2, we need to analyze congestion in l_1 first. Therefore, a safe order of links needs to be found according to the network topology.

This simple algorithm allows for simulation of congestion delays for a known network state. However, the algorithm only models congestion that occurs, when two flows compete for the same output port. The NoC model and the algorithm have no notion of the buffer utilization in the network. In a congested network, buffers could fill up and block flows that want to use otherwise unused links. For instance, a flow from PE1 to the RAM on the Tomahawk2 would fill up the input buffer of the south-west router as the memory controller services incoming packets slower than the NoC. Another flow from PE0 to PE4 would be blocked, by the full buffer in the south-west router. Figure 4.8 on the following page illustrates this scenario. The presented algorithm is not able of modeling this kind of congestion. On the Tomahawk2 this scenario may only occur when a flow writes to the RAM. However, for

larger NoC architectures this case could become more relevant and should be considered separately.

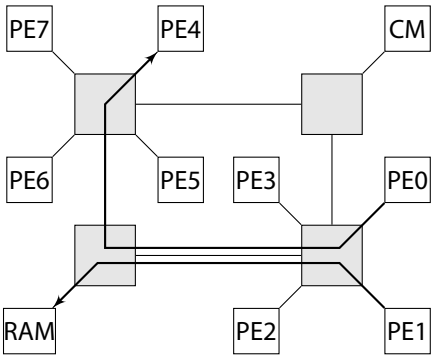


FIGURE 4.8: A flow going to the RAM can fill up the input buffer in the south-west router and thereby block other flows that use the same input port.

4.6 Processor Model

The processor model of MAPS is required for performance estimation. The performance estimator predicts the execution time of code segments by simulating the scheduling of instructions in a virtual processor architecture. This architecture is described by the processor model. The processor model defines all functional units that a processor has, the register files, and how pipelines are structured. The model also defines a mapping of instructions to the number of cycles required for their execution.

Performance estimation in MAPS is based on the LLVM [30] compiler infrastructure and uses a similar approach as the one proposed in [46]. Both approaches use LLVM bytecode for performance estimation. LLVM bytecode is a low-level intermediate representation of a program. It is machine code for a virtual RISC architecture and is executable using the LLVM virtual machine.

The performance estimator predicts execution time based on information learned from profiling. It translates code segments to LLVM bytecode, runs the code in the LLVM virtual machine, and extracts a list of all instructions that where executed. The performance estimator uses this list to simulate the scheduling of instructions in a virtual architecture that is described by the processor model.

To allow for accurate performance estimation for the Tomahawk2, a processor model for the Tomahawk's Xtensa LX4 cores needs to be defined. However,

the specialized Xtensa instruction set does not map well to LLVM bytecode. Xtensa cores are highly configurable and may be implemented as small microcontrollers as well as highly specialized DSP cores [51]. Depending on the configuration, the ISA provides specialized instructions that have no direct representation in the LLVM bytecode. For instance,the `LOOP` instruction allows for zero overhead loops by marking a certain range of instructions for repeated execution. In contrast, LLVM byte code always requires three operations per iteration: increment, compare, and conditional branch.

Another problem when modeling the Tomahawk's processing elements is that they implement the *windowed registers* option [51] of the Xtensa ISA. Using windowed registers, the actual register file is larger than specified by the ISA. Code that is currently executing on a PE only operates on registers that are part of the current window. The call instruction forwards the window by a certain amount of registers. Thereby, the callee can operate on a fresh set of unused registers while the caller's register set is preserved. When the called function returns, the window returns to its old position. By letting the caller's window and the callee's window overlap, the caller can pass parameters to the callee. This way, it is not necessary to preserve registers by pushing them to the stack before calling a function. However, when a call is executed and the register file is full, the whole register file needs to be pushed to the stack and needs to be restored when returning.

The processor model of MAPS only supports simple register files and has no concept for modeling access to windowed registers. This makes it impossible to accurately predict the execution time of call and return instructions as there is no information on the register files state. More importantly, the usage of LLVM bytecode leads to imprecise predictions for specialized instructions of the Xtensa ISA, e.g. the `LOOP` instruction. Altogether, the processor model of MAPS is not suitable for accurately modeling the Xtensa ISA as implemented on the Tomahawk2. During this work, incorrect predictions up to a factor of three where observed. For instance, the execution time predictions for a Sobel filter implementation (see Section 5.4.1) are about 2.6 times larger than the measured execution time.

Modifying the processor model in order to fit better the Xtensa ISA is a complex and time-consuming task. As working time is limited and this work's focus is on modeling communication in a network on chip (NoC) architecture, the Tomahawk2 hardware model uses a generic RISC processor model that is

provided by MAPS. However, this leads to inaccurate prediction of computation times and, therefore, is a limitation of this work.

4.7 Conclusion

This chapter introduced a Tomahawk2 hardware model for the MAPS compiler framework. Thereby, existing models where extended in order to add better support for the Tomahawk2 platform. The proposed hardware model introduces new types of communication primitives, uses an extended cost model that can be parameterized according to the network topology, and provides an interconnect model with support for NoC based architectures. Furthermore, an algorithm for calculating delays caused by network congestion was proposed. The previous section discussed the processor model of MAPS and reached the conclusion that this processor model cannot accurately model the Xtensa ISA, which is a limitation of this work.

5

Evaluation

In order to evaluate the solutions that Chapter 3 and Chapter 4 proposed, this chapter presents a series of experiments that cover various aspects of the proposals. Section 5.1 discusses the setup used for time measurement on the Tomahawk2 and analyzes the measurement error. Section 5.2 evaluates the communication model that this work proposes without considering network congestion. The algorithm for modelling network congestion is evaluated separately in Section 5.3. Section 5.4 compares the performance of applications that are written in TaskC with KPN applications in MAPS. Finally, Section 5.5 reviews mapping strategies for the Tomahawk family.

5.1 Time Measurement

The experiments presented in this chapter measure the total execution time of applications running on the Tomahawk2. Therfore, a method for measuring time is required. On the Tomahawk2, only the Core Manager can perform time measurements as only the Core Manager implements a hardware timer. Time could also be measured externally on the host PC (see Section 3.4.1). However, the communication between the Tomahawk2 board and the host PC would add a large and unpredictable delay. Therefore, time measurement directly on the chip is preferable in order to achieve accurate results.

The time measurement method that this chapter uses is similar to the method discussed in Section 4.3.1. The Core Manager performs the time measurement and signals are used in order to enable communication between Core Manager and PEs. However, in contrast to the time measurement in

Section 4.3.1, the Core Manager not only performs the measurements. Here, it also controls the measurements.

There is a *start* and a *finish* signal. The start signal is a broadcast signal that the Core Manager sends to all PEs when it starts the timer. When booting up, the PEs perform all necessary initialization tasks and then wait for the start signal before they start executing the KPN that the mapping assigned to them. This mechanism ensures that all processes start execution at almost the same point in time and that the time required for booting up and initialization is excluded from the time measurement.

When a process terminates, it sends the finish signal to the Core Manager. The Core Manager stops the timer when it received the finish signal from all eight PEs. Thus the Core Manager measures application execution time from the start of execution of all processes until termination of the last process.

The introduction of signals for the time measurements induces a systematic error. Signal handling and transfers cause an additional delay that is part of the measured time frame. As the PEs start execution only after they received and handled the start signal and the measurement only terminates after the Core Manager received and handled the last finish signal, the delays for both signals add to the total measured execution time of the application.

In order to estimate the measurement error, a simple experiment can be used. This experiment executes a "dummy" application consisting of eight processes which terminate immediately. If we use the time measurement method as described above for the dummy application, we can directly measure the systematic error as we know that the total execution time of the dummy application is zero. By repeating this experiment we can derive an distribution of the absolute error. Figure 5.1 on the next page displays this distribution for a total of 1,000 measurements.

The distribution of the systematic error in Figure 5.1 shows an irregular pattern. There is a smaller group of occurrences at about 800 to 900 cycles and a larger group at about 1,100 to 1,300 cycles. The median error is 1,141 cycles.

In order to compensate the systematic error, the Core Manager automatically subtracts the median value of 1,141 cycles from the measured execution time. However, this only reduces the absolute error but a systematic error remains because of the variation that is shown in Figure 5.1. Based on the error measurement, we can assume the remaining systematic error to be within a range of about -353 to 147 cycles.

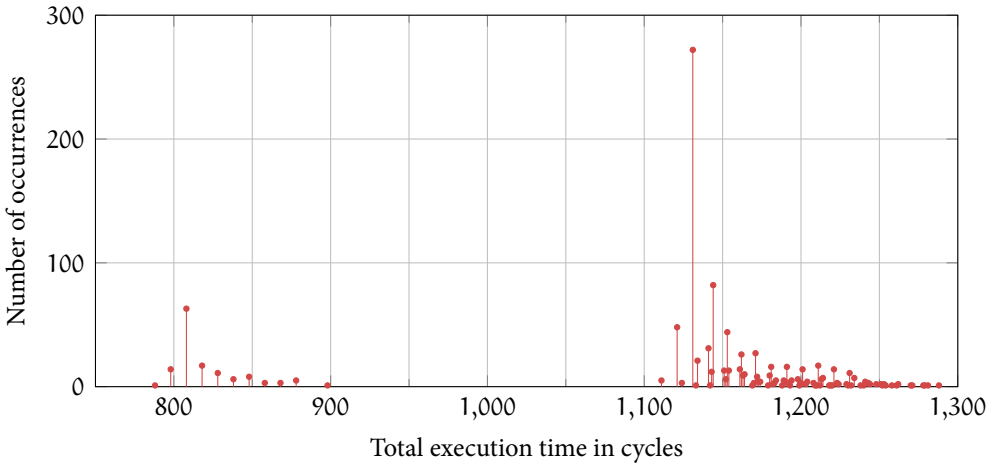


FIGURE 5.1: Distribution of the systematic error over 1,000 time measurements. The median error is 1,141 cycles.

5.2 COMMUNICATION MODEL

Section 4.4 introduced a communication model for the Tomahawk2. In order to evaluate the quality of execution time predictions that are based on this model, this section discusses three experiments that compare the measured execution time of an application running on the Tomahawk2 to the values predicted by the trace replay module (TRM). The first experiment (Section 5.2.1) is based on a simple KPN application with only one channel. The second experiment uses a more complex application with a pipeline structure (Section 5.2.2) and the third experiment is based on random KPN applications (Section 5.2.3).

All three experiments use the Tomahawk2 communication model that was defined in Section 4.4 but do not include the congestion model from Section 4.5. Section 5.3 evaluates the congestion model separately.

5.2.1 *Single Channel Application*

The first experiment is based on a simple KPN application that consists of a source and a sink process connected by a single channel. Neither process performs any computations but only produces/consumes 1,000 tokens in a for loop. The experiment measures the total execution time of the single channel

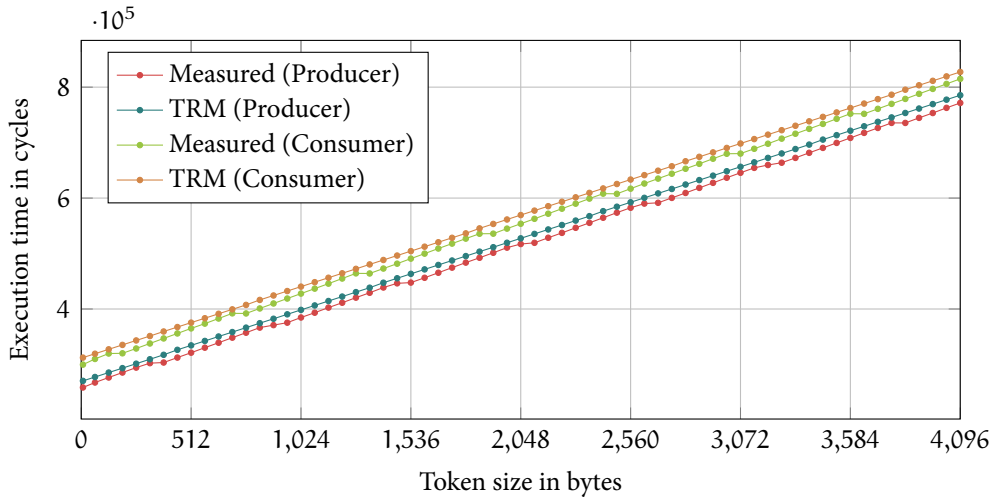
application for token sizes within the range from 8 to 4,096 bytes and for various mappings.

In order to examine all three communication primitives of the communication model, the experiment uses three different mappings. Each of the three mappings places the source and the sink process on neighbouring PEs so that there is a one hop distance between both processes. The three mappings differ in the channel buffer placement. The mappings place the channel buffer in the consumer scratchpad, the producer scratchpad, or in global RAM.

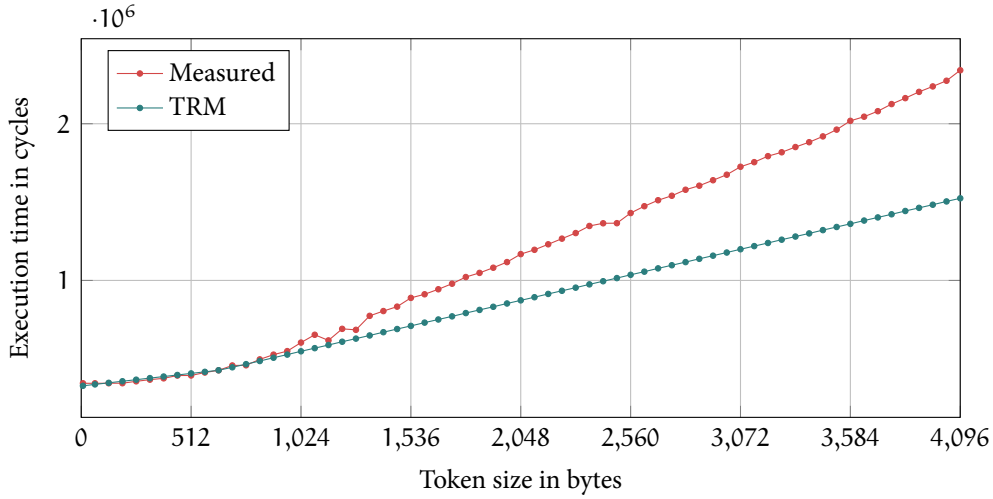
The setup for this experiment is similar to the setup that Section 4.3.1 used in order to measure the communication costs. Therefore, we can expect the measured execution times to match the TRM prediction. Figure 5.2 on the facing page displays the experiment's results.

Figure 5.2a on the next page covers mappings that place the channel buffer in a scratchpad memory. For both the consumer scratchpad mapping and the producer scratchpad mapping, there is a constant offset between simulated and measured execution times. This prediction error can be explained by the insufficient processor model. As discussed in Section 4.6, the Tensilica compiler can optimize for loops using the `LOOP` instruction and thus realizes zero overhead loops. However, the processor model is not aware of this optimization and, therefore, includes the overhead for increment and conditional branch in its prediction. However, as this difference in predicted and actual processing time is constant and the offset between measured and simulated execution times is also constant, we can conclude that the hardware model predicts communication times accurately for a single channel that is mapped to a scratchpad.

Figure 5.2b shows the experiment's results for a mapping that places the channel buffer in global RAM. In contrast to Figure 5.2a, there is a divergence between measurement and simulation. For a token size of 4,096 bytes, the measured execution time is more than 50 percent higher than the simulated execution time. This discrepancy is caused by interleaving memory requests. When both the consumer and the producer process send requests to the memory controller, the controller cannot operate on contiguous memory areas but has to handle interleaving request separately. This increases the service time for interleaving requests drastically. Therefore, the proposed communication model cannot accurately model RAM accesses. The following section will revisit this problem based on a different setup.



(a) Transfer via scratchpad



(b) Transfer via global RAM

FIGURE 5.2: Comparison of measured and simulated execution times for a single channel application. While the transfer time for scratchpad accesses can be predicted accurately, interleaving RAM accesses lead to mispredictions.

5.2.2 *Pipeline Application*

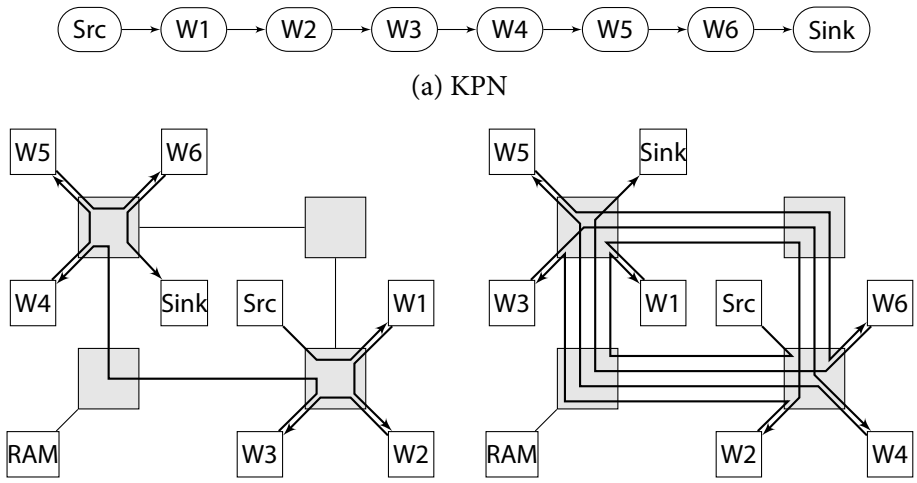
In order to evaluate performance estimation for a more complex application, this section presents an experiment based on a pipeline application. Figure 5.3a on the facing page visualizes the KPN of this application. It comprises six worker processes, a source process, and a sink process. The source process permanently produces tokens and the sink process permanently consumes tokens. In each iteration, a worker process consumes a token, performs computations, and produces a token.

In order to avoid erroneous predictions of computation times caused by the insufficient processor model, the worker processes simulates computations by calling a delay function. The worker process in this experiment use a delay of 1,000 cycles. The CPN keyword `__PNconsume` can be used to define the exact execution time of a process segment. Using this keyword, the experiment ensures that trace analysis uses exactly the delay of 1,000 cycles.

The pipeline application can be mapped to the Tomahawk2 in various ways. In the best case scenario (Figure 5.3b), processes that communicate witch each other are mapped close to each other and each link in the NoC is used by exactly one KPN channel. In the worst case scenario (Figure 5.3c), processes that communicate with each other are mapped to different groups of PEs and multiple KPN channels share the same NoC links.

Figure 5.4 on page 68 displays the experiment's result. Similar to the single channel experiment, this experiment examines three scenarios for the placement of channel buffers. The mapping places all channel buffers in the consumer scratchpads (Figure 5.4a), in the producer scratchpads (Figure 5.4b), or in the global RAM (Figure 5.4c). The diagrams only show TRM values for the best case mapping as the predicted difference between best case and worst case mapping is relatively small and would not be visible in the diagrams.

In the case that all channel buffers are mapped to the consumer scratchpad, the measured best case execution times match the TRM simulation. Similar to the single channel application, there is a constant offset between both values, which can be explained by the processor model's misprediction due to the for loop. The execution time for the worst case mapping only matches the TRM prediction for small token sizes and diverges significantly for token sizes larger than 2,048 bytes. This is caused by network congestion as multiple KPN channels share the same network links. Based on these results, we can conclude



(a) KPN

(b) Best case mapping

(c) Worst case mapping

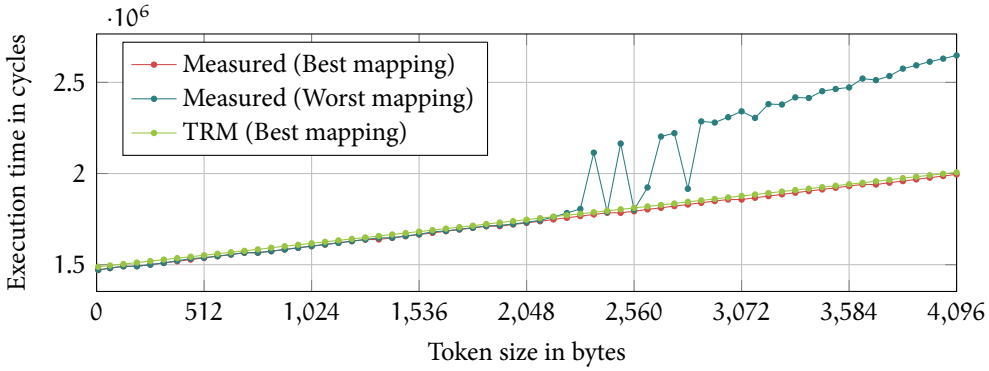
FIGURE 5.3: A pipeline application with two mapping scenarios

that the cost model for the consumer communication primitive accurately predicts transfer times for uncongested networks.

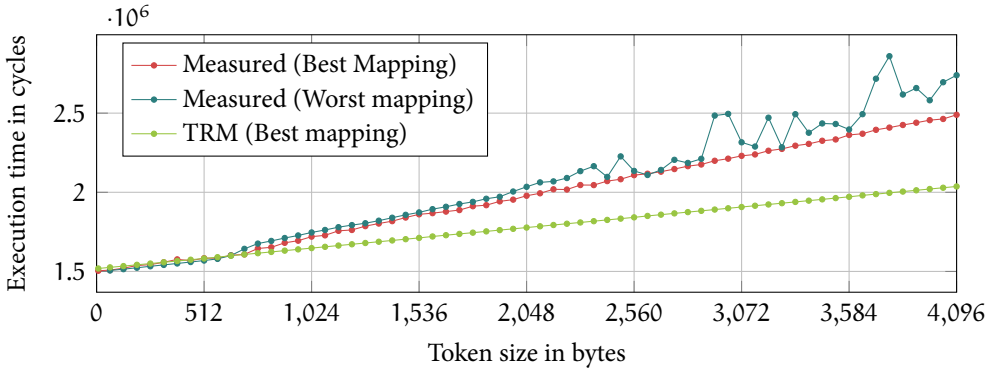
Figure 5.4b on the following page shows a different behaviour. In the case that all channel buffers are mapped to the producer scratchpad, the measured values for both the best case mapping and the worst case mapping diverge significantly from the predicted values. The memory interfaces of the scratchpads on the Tomahawk2 are responsible for this increase of the total execution time.

Although the network links on the Tomahawk2 may operate in full-duplex, the memory interface that connects the scratchpad memory to the NoC cannot handle simultaneous reads and writes. For example, when the memory interface of PE_x handles a read request send by PE_y, this memory interface is busy and cannot accept further requests. This may lead to additional delays and is the reason for the divergence of measured and predicted values in Figure 5.4b.

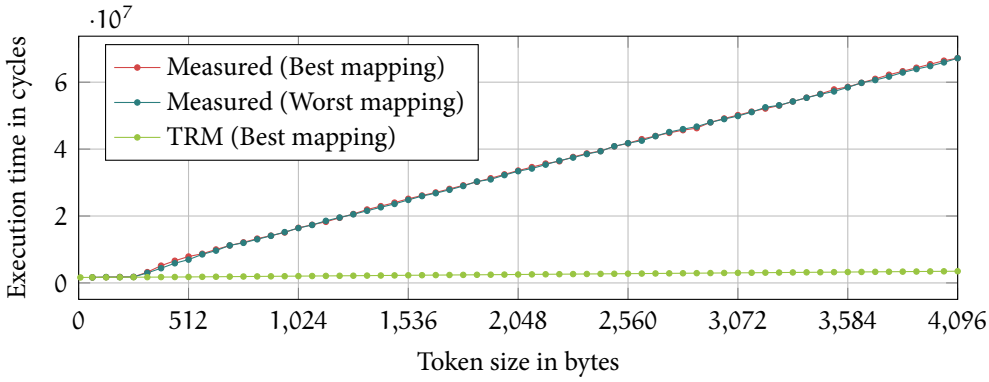
When a worker process W_x finishes producing a token, it immediately consumes a new token. At the same time the next worker process W_{x+1} consumes the newly produced token. As the channel buffer is located in the scratchpad of the producing processes, both consume operations require a data transfer. One transfer writes to the scratchpad of W_x and one reads from this scratchpad. However, as the memory interface can only handle one request at a time, one transfer has to wait, which leads to an additional delay.



(a) Transfer via consumer scratchpad



(b) Transfer via producer scratchpad



(c) Transfer via global RAM

FIGURE 5.4: Comparison of measured and simulated execution times for the pipeline application. The worst case mapping leads to mispredictions due to network congestion (a and b). Interleaving RAM accesses slow down the memory controller and lead to a large runtime overhead (c).

The NoC model as presented in Section 4.2 has no notion of memory interfaces. Therefore, it cannot predict the delay caused by processes waiting for a busy interface. This is a limitation of the proposed hardware model. However, the pipeline application enables and amplifies this effect due to its synthetic structure with identical processes. Other applications might not be affected by this limitations.

Figure 5.4b on the preceding page illustrates measured execution times and TRM prediction for the case that all channels are mapped to the RAM. Naturally, there is no clear difference between the best case mapping and the worst case mapping. As all processes communicate via global RAM, the mapping has no direct influence on the communication.

There is a large difference of more than an order of magnitude between the predicted execution times and the measured values for large token sizes. As already discussed for the single channel application, this discrepancy is caused by interleaving memory requests. As the pipeline application comprises more processes than the single channel application, the number of interleaving memory requests is much higher and thus the effect has a stronger influence on the total execution time.

The proposed communication model does not consider the influence of interleaving RAM requests when estimating communication times. This is a further limitation of the communication model. However, incorporating interleaving into the communication model is not reasonable as the hardware should avoid interleaving. In fact, the NoC provides a burst mode that ensures continues transmission a series of packets without interleaving. This would be similar to a network with optional wormhole switching. However, due to a bug in the Tomahawk2 hardware design, the burst mode is not available. As the interleaving issue will be resolved in future iterations of the Tomahawk chip, creating a specific interleaving model for the Tomahawk2 is not be reasonable.

5.2.3 *Random Applications*

The two previous sections used synthetic benchmarks to analyze performance prediction for communication intensive applications on the Tomahawk2. In order to cover a wide range of possible applications, this section analyzes randomly generated KPNs.

Tretter introduced a random KPN generator for MAPS in his diploma thesis [54]. This generator uses SDF³ [49] for generation of random SDF graphs and generalizes these graphs in order to create KPNs. As a KPN is not only defined by its network structure but also by the behaviour of all processes, the random KPN generator also creates process traces.

As the random KPN generator is incorporated into MAPS, the generated KPNs can only be used internally. In order to create an actual executable application, CPN code needs to be generated. For this purpose, a small script was created as part of this work. This script analyzes the KPN structure and parses process traces of a randomly generated application in order to create CPN code. For each channel access in the process traces the script adds a `__PNin` or `__PNout` statement. Computation times between channel access are implemented by a call to a delay function.

This experiment is based on 2,500 randomly generated KPNs. Each KPN consists of up to eight processes. Each channel has a mean token size of 512 bytes and each process segment has a mean length of 5,000 cycles. The Load Balancer algorithm of MAPS is used to derive a mapping for each application. Then the TRM estimates the total execution time of this application and the application is executed on the Tomahawk2 in order to measure the real execution time.

For each pair of measured and predicted execution times we can calculate the relative error of the TRM prediction. By calculating the relative error for all 2,500 measurements, we get a distribution of the relative error over all analyzed KPNs. Table 5.1 and Figure 5.5 illustrate the characteristics of this distribution.

The results show that most predictions are close to the actual measured value. The relative error is less than 1% for 88% of all analyzed KPNs and is less than 5% for 99% of all analyzed KPNs. However, there are cases where the TRM underestimates the total execution time significantly. The maximum relative error of all analyzed KPNs is 15.3%. This rare but large discrepancy between predicted and measured values is caused by contention on the scratchpad memory interface as described in the previous section.

On the whole, the random KPN experiment shows that the proposed Tomahawk2 hardware model allows for accurate predictions in most scenarios. However, in some cases the hardware model's limitations may lead to significant mispredictions.

Minimum	−0.983%
1st Quartile	−0.338%
Median	−0.191%
Mean	0.186%
3rd Quartile	0.258%
Maximum	15.331%

TABLE 5.1: Characteristics of the relative error distribution for 2,500 TRM performance estimations on random traces

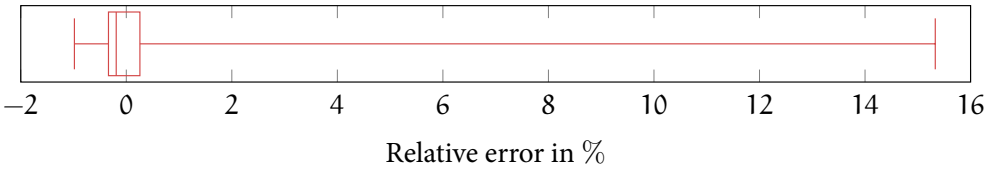


FIGURE 5.5: Box plot of the relative error distribution for 2,500 TRM performance estimations on random traces

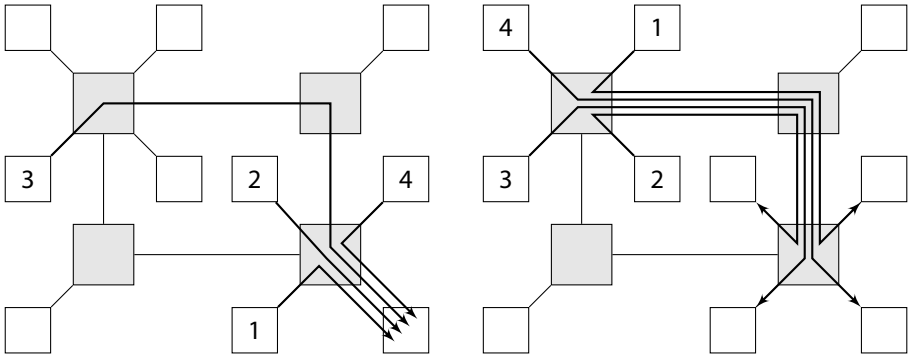
5.3 Network Congestion

The previous section examined the Tomahawk2 communication model without applying the congestion algorithm proposed in Section 4.5. In order to evaluate the congestion model, this section discusses two experiments. The first experiment in Section 5.3.1 examines network congestion on PE links, which are links that connect from a router to a processing element. The second experiment in Section 5.3.2 considers congestion on router links, which are links that connect two router with each other.

Figure 5.6 illustrates the setup for the two congestion experiments. In both scenarios four KPN channels share certain network links. Each channel has a source and a sink process that produces or consumes tokens continuously. Both experiments place all channels in the consumer scratchpad.

We can vary the data rate of each channel by varying the token size. As the channel library has a constant overhead for producing and consuming tokens, a smaller token size leads to a smaller data rate and a higher token size leads to a higher data rate. However, we do not know the exact relation between token size and data rate.

A simple experiment derives the relation between token size and data rate for a single channel. A source and sink process are connected by KPN channel and continuously produce or consume tokens for a known amount of time. We



(a) Congestion on link between router and PE

(b) Congestion on links between routers

FIGURE 5.6: For the congestion measurement, four parallel channels (labeled 1–4) share certain network links.

can measure the average data rate of this channel by counting the number of bytes transferred within the known time frame. Figure 5.7 shows the measured relation.

5.3.1 *PE Link Congestion*

The first experiment examines congestion in a PE link for a varying number of active channels. It uses the setup shown in Figure 5.6a. All active channels are set to the same data rate. For a higher data rate and for a higher number of active channels, congestion becomes more likely. Figure 5.8 shows the experiment's results and compares measured and predicted execution times.

Figure 5.8 illustrates that the total execution time increases, when the combined data rate of all active channels reaches the link bandwidth of 8 byte/cycle and thus network congestion occurs. The diagrams further shows, that the proposed congestion algorithm is capable of predicting the additional delay caused by network congestion for PE links and for channels with equal data rate. However, the predictions are more accurate for highly congested networks.

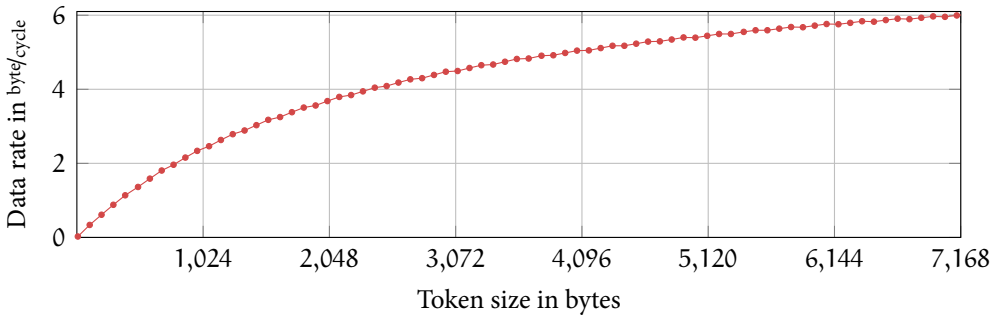


FIGURE 5.7: Relation between token size and average data rate for a single KPN channel

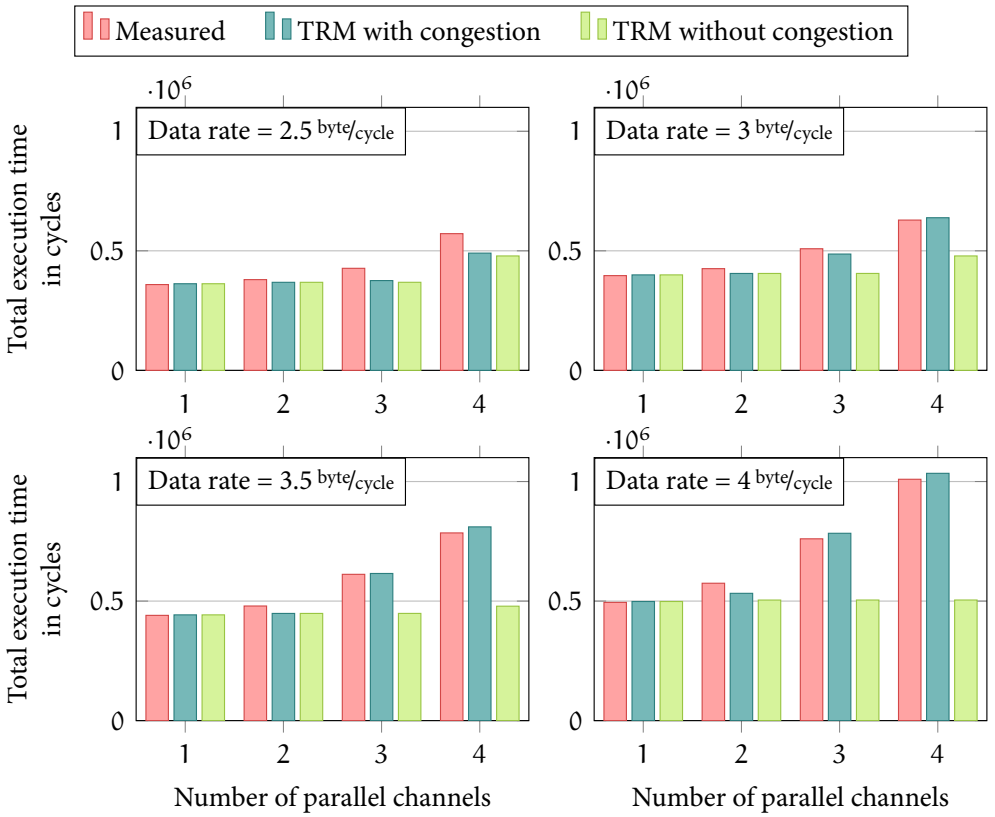


FIGURE 5.8: Comparison of measured execution times and predictions for congestion on a PE link. For high congestion, the congestion model yields accurate predictions.

5.3.2 *Router Link Congestion*

The second experiment examines congestion on links that connect two routers with each other. The experiment uses the setup displayed in Figure 5.6b. In this experiment, all four channels are active at all time. The data rate of three channels is fixed and the data rate of the fourth channel is varied within a range from 0.1 to 6 $^{byte}/_{cycle}$. This setup enables examination of various load scenarios.

Figure 5.9 on the facing page illustrates the experiment's results. The diagrams compare measured execution times with the TRM predictions. From this comparison, we can clearly see that the congestion algorithm accurately predicts communication times in highly congested networks. However, the first diagram (fixed data rate of channel 1–3 is 2 $^{byte}/_{cycle}$) clearly shows that there is an area of low congestion where the congestion algorithm is not effective.

The congestion algorithm only considers the average data rate of active flows. Currently, a flow consists of a computation part (channel library) and a transfer part (DMA). As the computation part is part of the total communication time of a flow, the average data rate of the flow is lower than the actual data rate of the DMA transfer. Therefore, the combined average data rate of two flows may be lower than the link bandwidth, although there is congestion in the real application. This is a limitation of the proposed congestion algorithm.

The two benchmarks that where analyzed in this section are synthetic and specifically designed to cause network congestion. However, such a high congestion will not occur in a real application. As a real application not only performs channel accesses but also performs computations in between accesses, congestion is unlikely to occur on the Tomahawk2. However, in other platforms with large scale networks congestion needs to be considered also for real applications.

5.4 Performance Comparison of KPN and TaskC

KPN and TaskC are two fundamentally different approaches for programming MPSoC architectures. While TaskC uses atomic kernels of computation, KPN describes a network of parallel processes. While MAPS assigns hardware resources statically to KPN applications, TaskC applications are mapped dynamically at runtime. While tasks in TaskC may communicate via arbitrary areas of global memory, KPN processes communicate via FIFO channels.

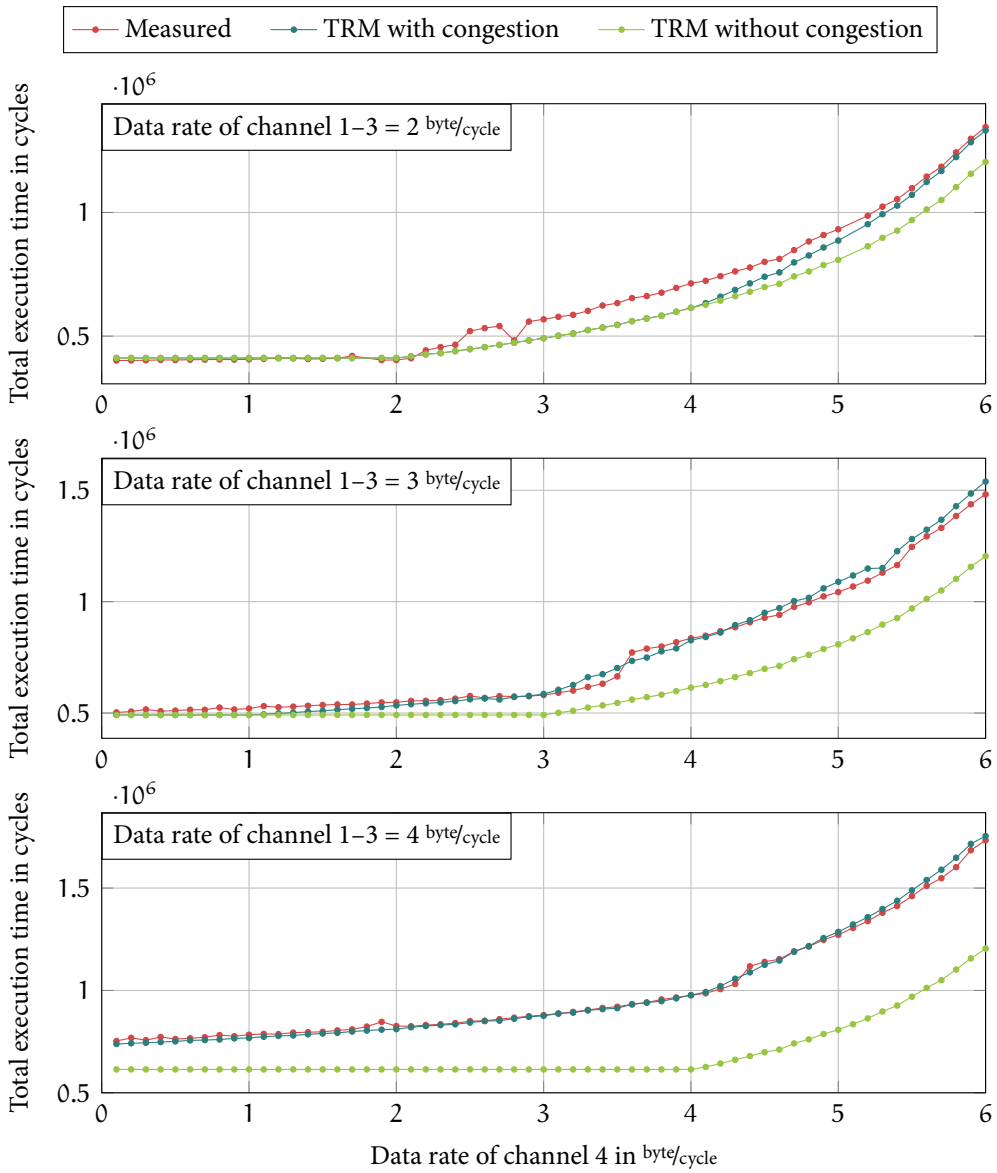
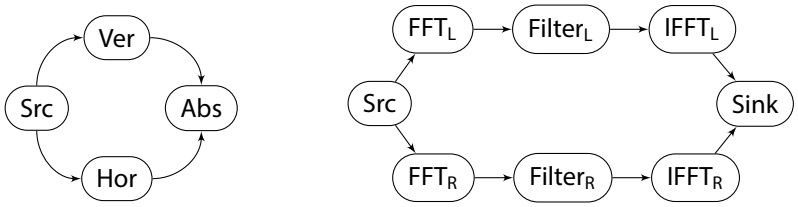


FIGURE 5.9: Comparison of measured execution times and predictions for congestion on a router link. For high congestion, the congestion model yields accurate predictions.



(a) Sobel filter (b) Two channel audio filter

FIGURE 5.10: KPN representation of two applications that are used for performance comparison of KPN and TaskC

This section presents two benchmarks that illustrate the difference in terms of performance between KPN and TaskC. Both benchmarks are based on real world applications. The first application is the *Sobel filter* (Section 5.4.1) and the second application is a two channel audio filter (Section 5.4.2). Figure 5.10 visualizes the KPNs for both applications.

The Sobel filter and audio filter applications where selected as benchmarks, as they are real applications and comply with the limitations of the Tomahawk2 backend. Both applications have a simple structure and can easily be ported to TaskC. Furthermore, the applications can easily be parameterized in order to compare multiple scenarios.

In order to derive a comparable TaskC implementation from a KPN application, all processes are implemented as tasks. Thereby, a task represents one iteration of a KPN process. The main thread that runs on the Application Processor is responsible for deploying tasks according to the process network. It may deploy multiple tasks of the same type, meaning multiple iterations of a KPN process, at the time. For the sake of simplicity, both TaskC implementations use a new area of memory for each data transfer between tasks. This leads to a much higher memory consumption compared to a KPN application. The remaining section uses the term *process* in a wider sense: it represents a KPN process as well as the corresponding tasks in TaskC.

Similar to the execution time measurement for KPN applications, the Core Manager handles the measurement for TaskC applications. In order to exclude initialization from the time measurement, the Core Manager starts its hardware timer when it receives the first task descriptor. The Core Manager stops the measurement after the last task terminated and the Application Processor send a shut down signal.

5.4.1 *Sobel Filter*

The Sobel operator is a simple filter for edge detection in image processing [48]. When applied to an input image, the operator creates an image with emphasized edges. The Sobel operator uses two 3×3 kernels that it convolves with the original image. One kernel approximates derivatives for horizontal changes and the other kernel approximates derivatives for vertical changes. For each point in the image, the approximations of the horizontal and vertical gradient can be combined to a vector. If we set the brightness of each point in the output image to the magnitude of its gradient vector, the output images shows emphasized edges.

Figure 5.10a on the facing page illustrates the KPN implementation. Although MAPS provides an example CPN implementation of a Sobel filter, a new implementation was created for this work. As the example implementation in MAPS uses sliding windows, this application is not compatible to the Tomahawk2 backend and, therefore, a new implementation is required.

Both the TaskC and the KPN implementation process an input image line by line. Therefore, a token is a full image line. A source process generates an image and sends it line by line to the two worker processes. One worker process convolves the generated image with the vertical Sobel kernel and the second process convolves the image with the horizontal Sobel kernel. The output of both processes is sent to a third worker process that calculates the magnitude of the gradient vector.

Figure 5.11 on the next page compares the execution times of both the KPN and TaskC implementation of the Sobel filter for various image sizes. The displayed values are derived from a single measurement and the source process scales the generated image according to the configuration. A change in the image width results in a change of the token size in both applications as a token is one image line. However, changing the picture height only increases the total number of iterations as both implementations generate and process the image line by line.

The results in Figure 5.11 clearly show that the execution time of the KPN implementation is always better than the execution time of the TaskC implementation. This is an interesting result as the TaskC application can utilize all eight processing elements of the Tomahawk2 while the KPN application is limited to four processing elements due to its static mapping. Compared

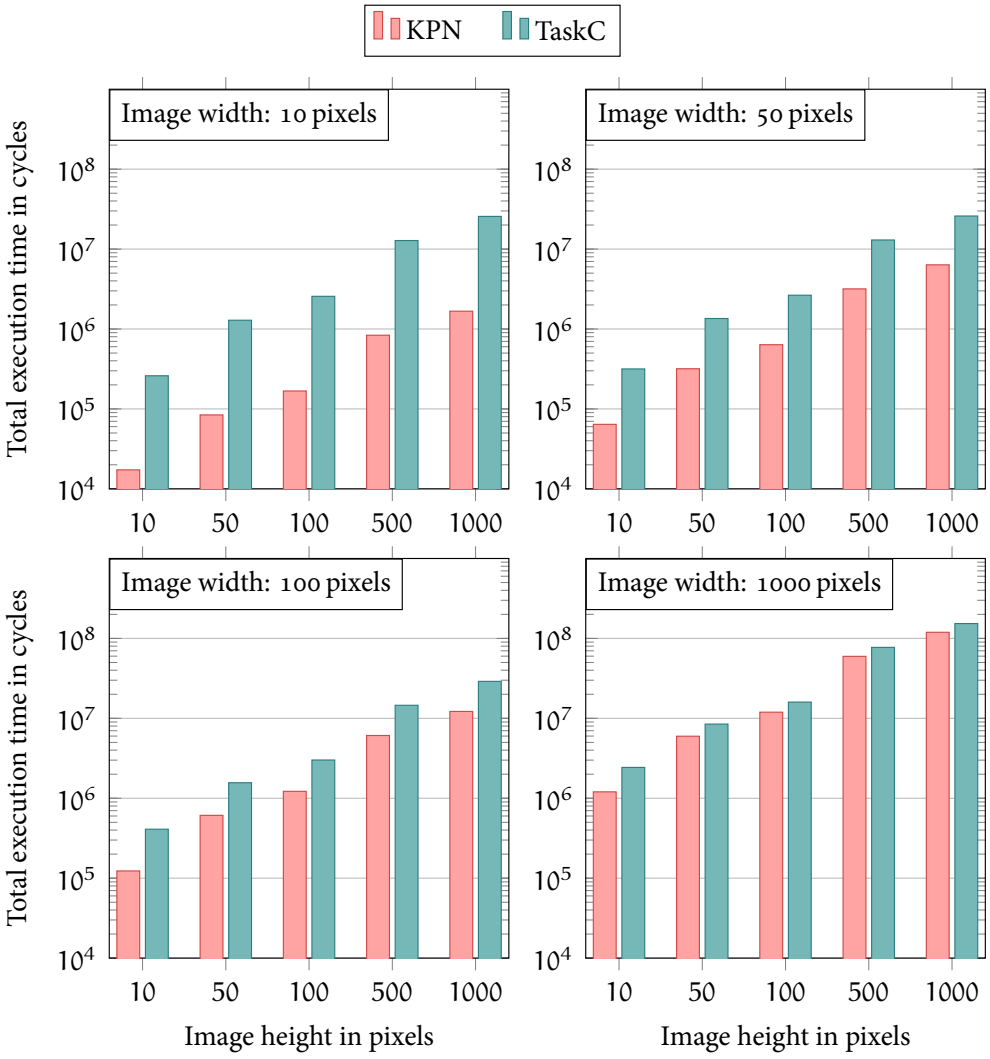


FIGURE 5.11: Performance comparison for the Sobel filter implemented in TaskC and KPN. Although the TaskC application can utilize more PEs, the KPN implementation is always faster.

to the KPN implementation, the TaskC application has the double amount of processing power available but still the overall execution time is higher.

Due to the logarithmic y-axis in Figure 5.11, the diagrams can be used to compare the relative difference of the execution time between both implementations for various image sizes. When the image height changes, the relative difference stays approximately constant. Only in the case of a height of 10 pixels the relative difference is slightly higher than for the other displayed image heights. However, changing the image width has an influence on the relative difference between the two execution times. The relative difference gets smaller for a larger image width. It decreases to a factor of two for an image width of 1,000 pixels. However, for an image width of 10 pixels, the relative difference is more than an order of magnitude.

When increasing the image width, both the token size and the computational effort required for one iteration increase. For a small image width the computational effort for each task is relatively low. In this case, the overhead for dependency checks and task deployment in the TaskC application outweighs the task execution time. For higher image widths, the execution time of single tasks increases while the overhead for task deployment is constant. Therefore, the TaskC implementation of the Sobel filter becomes more efficient for larger image widths.

Possibly the TaskC application could become more efficient than the KPN application for image widths that are larger than 1,000 pixels. However, this experiment is limited to a maximum of about 1,000 pixels due to the memory constraints of the Tomahawk2.

5.4.2 *Audio Filter*

The *Audio Filter* is an example KPN application provided by MAPS. Figure 5.10b on page 76 visualizes the audio filter process network. The audio filter application comprises eight processes that operate on a two channel audio stream. The use of eight processes makes this application well suited as a benchmark for the Tomahawk2 with its eight processing elements.

The audio filter application operates on 1,024 byte data blocks and processes a total of 20 kiB audio data. A source processes creates two audio streams from a stereo source—one for the left audio channel and one for the right audio channel. Data blocks for both channels pass a pipeline of three processes. The

Process	Computation time in cycles (measured)	(rounded)
Src	48,610	50,000
FFT	3,027,372	3,000,000
Filter	153,625	150,000
IFFT	3,013,417	3,000,000
Sink	97	0

TABLE 5.2: Computation times for the audio filter processes derived from pthread traces in MAPS on x86

first process performs an FFT, the second process applies the actual audio filter, and the third process performs the inverse FFT. A sink process unites both audio streams and stores the result.

The audio filter application as provided by MAPS does not fit into the scratchpad memories of the Tomahawk2. In order to avoid the complexity of optimizing the application, this experiment uses an application that has the same temporal behaviour but does not implement the actual computations. The processes of the audio filter application have a rather simple structure. In each iteration they consume a token from the input channel, perform computations for a certain amount of time, and produce a token on the output channel. Knowing the time required for computations, this behaviour can be mimicked using a simple delay. This leads to a flexible benchmark that is inspired by a real application but where certain parameters, like the time required for performing a certain computation, can easily be changed in order to understand their influence on the performance.

In order to derive the computation time for each of the audio filter processes, we can analyze process traces generated by the *pthread* backend of MAPS. Table 5.2 shows the computation times derived from trace analysis on an x86 architecture. As these values originate from analysis on an x86 architecture, they are not representative for other processor architectures. However, the measured computation times can be seen as an estimation of the execution times on other architectures and are a good starting point for this experiment.

Both the TaskC and the KPN application implement delay functions according to the rounded values from Table 5.2. Measurement of the execution time for both applications yields values of 36 million cycles for the KPN implementation and 25 million cycles for the TaskC implementation. These values show, that the TaskC application is more efficient than the KPN implementation.

The audio filter application does not equally distribute the computational effort amongst its processes. The computation time required for performing an FFT or IFFT on a data block is 20 times higher than the computation required for applying the filter. This divergence in computation times between processes leads to the difference in total execution time between TaskC and KPN. The static mapping of MAPS does not allow for efficient usage of the available resources as it does not exploit data parallelism. The filter process always has to wait for the much slower FFT process. Due to the static assignment of one process to one processing element, the processing elements cannot perform other computations while the process is waiting. In contrast, TaskC assigns resources dynamically and, therefore, can execute the FFT tasks in parallel in order to utilize all processing elements.

In order to get a better understanding of the computation times of single process iterations on the application performance, we can vary these times and examine their influence on the overall execution time. Let t_{FFT} be the time required for performing one iteration of the FFT process and let t_{Filter} be the time required for applying the filter to one data block. If we assume that the computation time for FFT and IFFT is identical then the total time T required for processing one token is defined by the following equation.

$$T = 2t_{FFT} + t_{Filter} \tag{5.1}$$

Let ρ be the ratio of computation times in the audio filter application with $\rho = {}^{t_{Filter}}/_{t_{FFT}}$. Then t_{FFT} and t_{Filter} can be calculated as follows.

$$t_{FFT} = \frac{T}{2+\rho} \tag{5.2}$$

$$t_{Filter} = \frac{T\rho}{2+\rho} \tag{5.3}$$

With Equations 5.2 and 5.3 we can calculate the computation times of the audio filter processes in dependence of T and ρ. Therefore, we can measure the performance of the KPN and TaskC audio filter implementations for various values of T and ρ in in order to understand the influence of these parameters on the overall execution time. When changing ρ for a fixed value of T only the distribution of the computational effort amongst processes changes but the overall effort for processing all input data is constant.

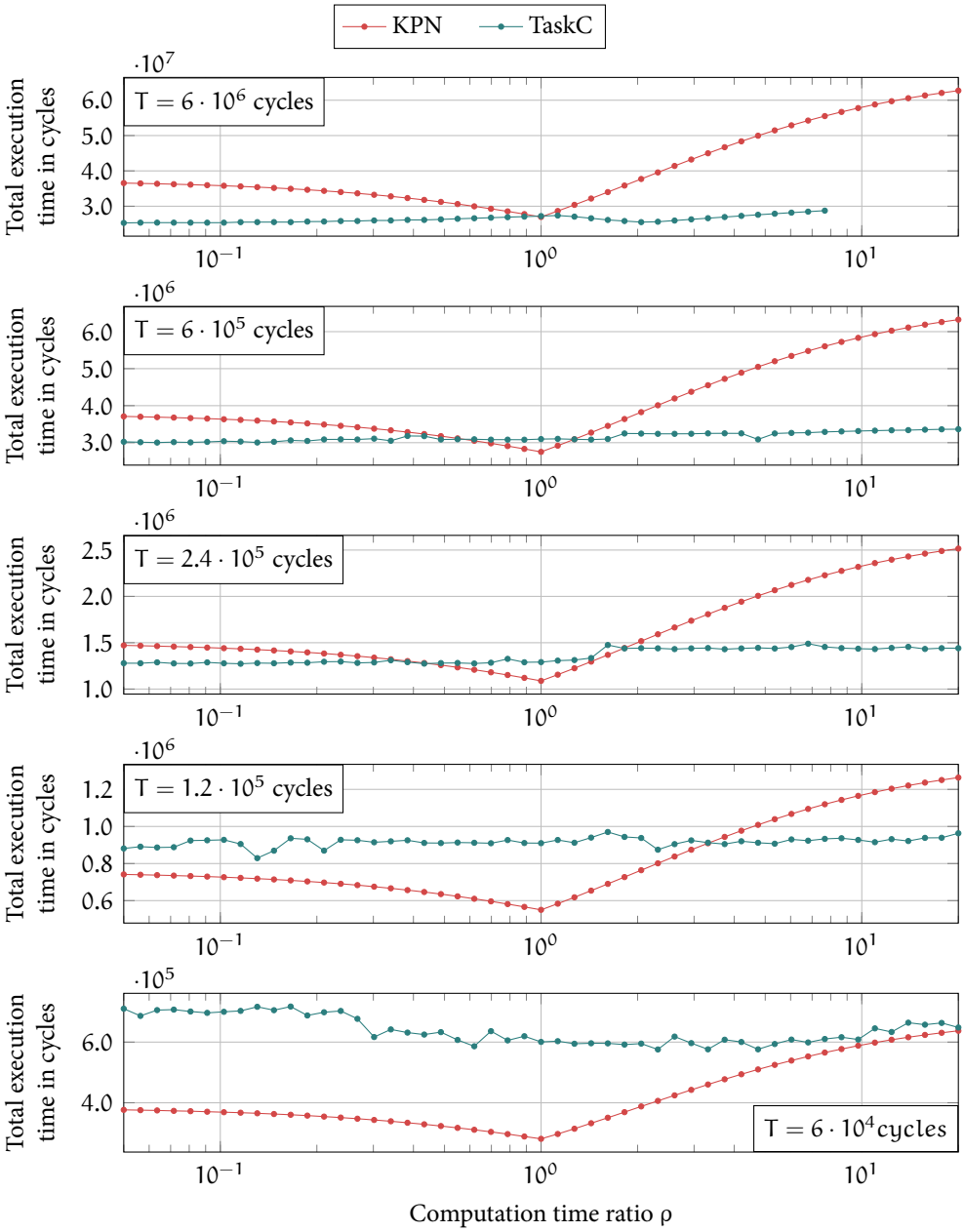


FIGURE 5.12: Performance comparison for the audio filter implemented in TaskC and KPN. The performance of the KPN implementation highly depends on ρ, which is the ratio between the computation length of filter and FFT.

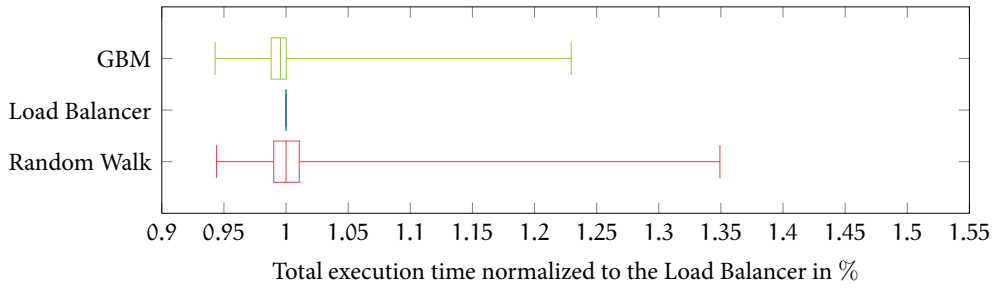
The diagrams in Figure 5.12 show the results of performance measurements that vary ρ within a range from 0.05 to 20. Each diagram displays the results for a certain value of T. The case $T = 6 \cdot 10^6$ cycles and $\rho = 0.05$ approximately reflects the original audio filter application. For $T = 6 \cdot 10^6$ and $\rho > 8$, the execution time of the TaskC application could not be obtained as the application crashes for these values. The reason for these crashes could not be identified within the time frame of this work.

Figure 5.12 verifies that the efficiency of the KPN audio filter realization depends on the distribution of computational effort amongst processes. For an equally distributed computational effort ($\rho = 1$), the overall execution time of the KPN application is minimal. The overall execution time of the TaskC application is approximately constant for all values of ρ. Therefore, we can conclude that TaskC is not affected by unevenly distributed processing times of tasks and that KPN shows best performance on the Tomahawk2 for an evenly distributed workload.

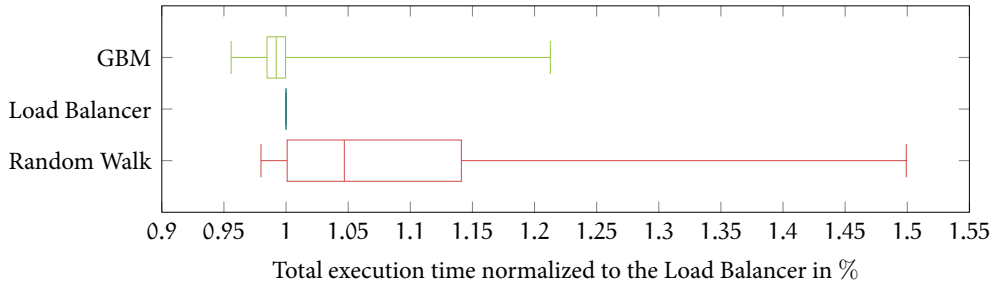
Each diagram in Figure 5.12 shows the measured execution time for a specific value of T. In the case $T = 6 \cdot 10^6$ cycles, the KPN application only reaches the performance of TaskC for $\rho = 1$. However, for smaller values of T and thus smaller computation times of single tasks, TaskC becomes less efficient as the overhead for dependency checking and task deployment is constant. In the case of $T = 6 \cdot 10^4$, the KPN application is faster for all considered values of ρ. Based on these results, we can conclude that KPN is better suited for applications with fine grained processes and that TaskC is better suited for applications with complex tasks.

5.5 Mapping

So far this work discussed code generation and a hardware model for the Tomahawk2 but did not yet consider the influence of mapping decisions on the performance of KPN applications. Therefore, this section compares the resulting application performance for three mapping algorithms and for a set of random KPN applications. The considered mapping algorithms are *Load Balancer*, *GBM*, and *Random Walk* (see Section 2.3.3 for details). While the Load Balancer uses a rather simple strategy, GBM applies a complex heuristic. Random Walk selects the best mapping out of a set of random mappings. In this experiment this set comprises 100 mappings.



(a) Tomahawk2 (Based on 2,500 random KPN applications)



(b) Virtual next generation Tomahawk platform (Based on 400 random KPN applications)

FIGURE 5.13: Box plots comparing the total execution time of three mapping algorithms normalized to the Load Balancer. The more complex GBM algorithm is not significantly better compared to the Load Balancer.

The mapping experiment only compares TRM predictions and does not consider actual execution times. This reduces the effort for performing the experiment significantly but the results are not as representative as they would be for measured values. However, Section 5.2.3 showed that the TRM values correspond with the measured values in most cases.

Figure 5.13a compares the performance of three mapping algorithms for 2,500 random KPNs. In the figure, the total execution time is normalized to the execution time achieved by the Load Balancer. Values that are less than one stand for execution times that are better than the execution time of the Load Balancer and values that are greater than one stand for worse execution times.

The diagram shows that GBM is better than the Load Balancer for most cases. However, the improvement over the Load Balancer is only within the

range up to 5%. In 25% of the cases, GBM even generates mappings that are significantly worse than the Load Balancer mappings. With a median of one, the Random Walk algorithm achieves execution times that are comparable to the Load Balancer. In half the cases, Random Walk is better and in the other half the Load Balancer is better.

In the average case, the discrepancy between GBM, Load Balancer, and Random Walk is rather small. Therefore, the choice of mapping algorithm is not critical for applications running on the Tomahawk2. Due to the regular structure of the Tomahawk2, bad mapping decisions do not necessarily have a strong influence on the application performance.

The next generation of the Tomahawk, could be more interesting for mapping. A chip of the new Tomahawk generation will only comprise four processing elements. However, a PCB will hold multiple connected chips connected by off-chip links. As the off-chip communication is significantly slower than on-chip communication, mapping decisions should have a stronger influence on the application performance.

In order to compare mapping algorithms for the next Tomahawk generation, a virtual platform was created. Figure 5.14 illustrates this platform. It consists of a total of 16 processing elements connected my start mesh NoC that is similar to the Tomahawk2 NoC. However, the links between routers are significantly slower. Their bandwidth is assumed to be 5 $^{\text{Gbit}}/_{\text{cycle}}$.

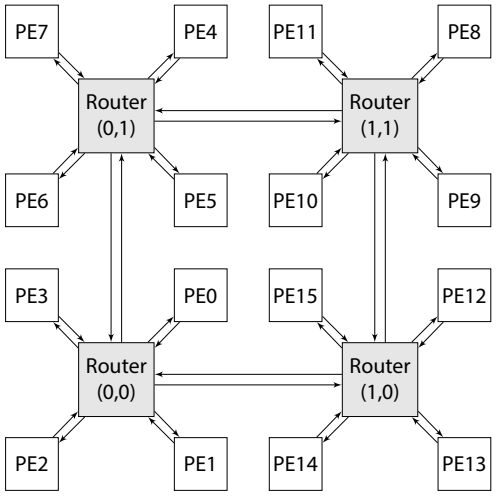


FIGURE 5.14: A virtual platform that represents a possible future generation of the Tomahawk chip.

Figure 5.13b on page 84 compares the performance of the three mapping algorithms for the virtual next generation platform. The measurement for this new platform only analyzes 400 random KPNs. The time required for executing the mapping algorithms for the next generation platform is significantly higher than for the Tomahawk2. Therefore, a larger data set could not be obtained within reasonable time.

The figure shows that Random Walk leads to significantly worse mapping than the Load Balancer for the next generation platform. However, GBM is still not a significant improvement over the Load Balancer and can still lead to significantly worse mappings. This illustrates that a complex mapping heuristic is not required in order to find sufficiently efficient mappings for the Tomahawk2 and for the next generation platform.

6

Future Work and Conclusion

This work presented and evaluated tools for programming the Tomahawk2 MPSoC using the KPN model of computation. Chapter 3 discussed design and implementation of a backend for the MAPS compiler framework. With the newly introduced backend, MAPS is capable of generating code for the Tomahawk2 in order to derive executable implementations of KPN applications.

Chapter 4 introduced modifications to the hardware model of MAPS that are required for incorporating features of the Tomahawk2 platform. Most importantly, a NoC model was created. Furthermore, a specific Tomahawk2 cost model was defined based on various benchmarks.

Chapter 5 evaluated the Tomahawk2 backend and hardware model. Although there are limitations, it was shown that the proposed hardware model can be used to create accurate performance estimations for Tomahawk2 applications. A performance comparison of three mapping algorithms showed that the choice of mapping algorithm only has a small influence on the application performance. Therefore, complex heuristics are not required in order to find efficient mappings for the Tomahawk2.

A comparison of the performance of KPN and TaskC applications, showed that both models have advantages and disadvantages. While TaskC can exploit data parallelism, its dynamic approach leads to a large overhead which makes TaskC unsuitable for applications with fine grained tasks. KPN, on the other hand, keeps the runtime overhead low due to the static mapping that MAPS derives. However, the static mapping prevents runtime adjustments according to the current load scenario. Therefore, statically mapped KPN cannot utilize the available hardware resources as efficient as TaskC.

6.1 Future Work

The achievements as well as the limitations of this work leave a lot of room for future work.

Utilize excluded Tomahawk features

Section 3.1 introduced a simplified model of the Tomahawk2 platform in order to exclude features that are beyond the scope of this work. However, integration of some of these features could be interesting for future work.

The Tomahawk's ADPLLs and power management units allow for frequency and voltage scaling. This can be used to implement power saving mechanisms. Therefore, the Tomahawk2 platform could be interesting for research on power optimization in MAPS. The mapper could statically assign critical processes to fast processing elements and uncritical processes to slower but more power efficient processing elements. A runtime manager that adjusts operating frequencies according to the current load scenario could also be integrated.

The Tomahawk2 implements Duo-PEs that comprise two cores of different ISAs—a DSP and a RISC core. However, as there is no C compiler for the DSP core this work excluded the Duo-PE feature and only considered the RISC cores. Future iterations of the Tomahawk2 platform could implement cores of other architectures. For such an architecture, it would be interesting to derive mapping strategies that include the possibility for switching the processor architecture at compile-time as well as at runtime.

Overcome limitations of the Tomahawk2 backend

Section 3.2 introduced a series of design choices that where made in order to keep the effort for implementing the channel library and code generator within reasonable bounds. However, these design choices led to limitations that should be considered in feature work.

As there is no runtime scheduler for the Tomahawk2, applications that run on the Tomahawk2 are restricted to a total of eight processes—one process per processing element. Future work could consider the implementation of a runtime scheduler in order to allow for multiple processes per processing element. This could improve the utilization of the available computational resources significantly, especially for scenarios with multiple applications. An

alternative to the implementation of a real scheduler could be the usage of Protothreads [23].

The channel library presented in Section 3.3 and Section 3.4 does not support CPN features that are extensions to the KPN model of computation. These are the support for channels with multiple readers and the sliding window feature that enables simultaneous access to multiple tokens. Although these features are not required for KPN applications, the backend needs to implement them in order to be fully compatible to all CPN applications.

The presented channel library puts no restrictions on token sizes. However, the library is most efficient for token sizes that are a multiple of eight bytes. Due to the fixed packet size of eight bytes of the Tomahawk2 NoC, other token sizes require special consideration. The solution presented in Section 3.4 adds additional data transfers that ensure data consistency. This introduces an additional overhead. Future work could implement other techniques like packaging. Packaging could also be interesting for optimizing channels with small token sizes.

Continue work on the NoC model

Evaluation in Section 5.2 showed that the proposed NoC model is capable of modeling NoC communication on the Tomahawk2 accurately in most scenarios. Section 5.3 further showed that the proposed congestion algorithm can predict delays accurately for highly congested networks. However, both models leave room for improvement.

In order to get a better understanding of the qualities and limitations of the proposed communication model, evaluation on other platforms is required. Especially platforms with large scale networks like the Adapteva Epiphany E64G401 [2] are of interest. Also the future generation of the Tomahawk platform with its off-chip network is interesting for evaluation.

Future work could also consider incorporation of more complex NoC models for performance estimation in large scale networks. NoC models that are based on contention trees [35] are able to predict packet traversal times for a known load scenario. This could be useful for performance prediction in the TRM. The contention tree algorithm proposed by Dasari, Nikolić, Nélis, *et al.* is capable of deriving tight worst case bounds for network traversal times in a known load scenario [21]. However, the recursive approach of contention tree algorithms has a high complexity [21].

Besides a NoC-aware communication model, a NoC aware mapping heuristic that considers the network load could be interesting for future work. Such an algorithm could be based on a NoC model that considers the average case characteristic of a network. For instance, stochastic automata networks (SAN) provide efficient average case estimation of transfer times [44]. Cruz proposed a network calculus that uses probabilistic reasoning to characterize network load [17], [18]. Approaches that are based on Poisson processes like the ones proposed in [26] and [39] could also be used.

Develop a dynamic mapping strategy

The design of the Tomahawk platform with the dedicated Core Manager makes this platform interesting for dynamic mapping. Instead of deriving a complete mapping at compile time, the Core Manager could make mapping decisions at runtime. As the Core Manager can base the decisions on the actual load, it can utilize the existing resources more efficiently.

Section 5.5 showed that mapping heuristics do not need to be complex in order to find efficient mappings for the Tomahawk2. Therefore, mapping decisions could be made at runtime without the introduction of a large overhead. Moreover, a hybrid approach could be used. MAPS could derive a static initial mapping that gets adjusted during runtime. Thereby, the runtime mapper can base its decisions on information learned from compile time analysis in MAPS.

In the literature various proposals for dynamic mapping heuristics can be found. For instance, Hölzenspies, Hurink, Kuper, *et al.* investigated runtime mapping of streaming applications onto heterogeneous MPSoCs with the aim of reducing the energy consumption [25]. Carvalho, Calazans, and Moraes proposed heuristics that minimize communication volume and, therefore, avoid network congestion [9]. The spiral mapping heuristic proposed by Benhaoua, Benyamina, and Boulet aims for minimizing data volume, network congestion, and total execution time [7]. The spiral heuristic places tasks of an application grouped in clusters where tasks with a high communication volume between them are placed next to each other. Future work could review the heuristics proposed in the literature and implement a runtime mapping heuristic on the Tomahawk2.

Integrate future developments

The Tomahawk platform and especially its software stack is a work in progress. The Tomahawk backend for MAPS should incorporate new features that will be introduced in the near future.

Currently a Tomahawk development kit as well as a framework for definition of computational kernels and communication channels are under development. A future version of the Tomahawk backend for MAPS should be based on this abstraction layer as it would enable execution of KPN applications on all versions of the Tomahawk platform. There are plans to provide mechanisms for scheduling and migration of computation kernels. This would allow straight forward implementations of dynamic mapping heuristics on the Core Manager.

A future Tomahawk backend could also be based on M3 OS [6]. The microkernel-based system for heterogeneous manycores (M3) is an operating system designed for MPSoC applications. It provides isolation and operating system services on the NoC layer. A dedicated core runs the kernel and configures other cores via a common hardware component, the so called DTU. Depending on the configuration, cores may access operating system services or communicate with other cores via their DTU. The abstraction of communication, providence of operating system services, and the increased security due to isolation make M3 interesting as a target system for MAPS.

6.2 Conclusion

This work contributes a Tomahawk2 backend for the MAPS compiler framework as well as a communication model for NoC-based architectures. Both the backend and the communication model where evaluated on hardware using a Tomahawk2 chip.

The proposed backend enables execution of KPN applications on the Tomahawk2. This introduces KPN as an alternative programming model for the Tomahawk2. As KPN is an abstract and widely used model of computation, it provides better portability than the current programming model TaskC, which was designed specifically for the Tomahawk architecture. The well defined structure of KPN applications as well as the explicit definition of dependencies increase maintainability and usability compared to TaskC. However, due to the dynamic mapping, TaskC allows for better utilization of the available resources. In the future, MAPS can overcome this limitation by supporting

dynamic mapping of KPN applications. Nevertheless, the lower runtime overhead of KPN applications can lead to better performance compared to TaskC, for applications with fine grained processes.

The proposed communication model allows for orthogonal definition of NoC architecture and communication costs. The Tomahawk2 model predicts local and remote accesses to scratchpad memories accurately. However, the Tomahawk2 model only has a limited support for RAM accesses as it does not model the slowdown that interleaving RAM request may cause.

The proposed communication model incorporates an algorithm that models network congestion. This algorithm accurately predicts traversal times in highly congested networks. However, in architectures with small NoCs, like the Tomahawk2, congestion rarely occurs for real applications. The influence of mapping decisions on the application performance is also relatively small for small networks. Nevertheless, as the trend goes towards MPSoC designs that integrate thousands of cores [8], a good NoC model that considers network congestion as well as NoC-aware mapping heuristics are relevant for future applications.

Bibliography

[1] B. Ackland, A. Anesko, D. Brinthaupt, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. O'Neill, J. Othmer, E. Sackinger, K. Singh, J. Sweet, C. Terman, and J. Williams, "A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 3, pp. 412–424, Mar. 2000, ISSN: 0018-9200. DOI: 10.1109/4.826824.

[2] Adapteva, Inc., *E64G401 Epiphany-IV 64-core Microprocessor Datasheet*, 2013. [Online]. Available: http://www.adapteva.com/docs/e64g401_datasheet.pdf (visited on Mar. 20, 2016).

[3] ——, (n.d.). The Parallella Computer, [Online]. Available: http://www.adapteva.com/parallella/ (visited on Aug. 14, 2015).

[4] O. Arnold, E. Matúš, B. Noethen, M. Winter, T. Limberg, and G. Fettweis, "Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3s, 107:1–107:24, Mar. 2014, ISSN: 1539-9087. DOI: 10.1145/2517087.

[5] Arteris, Inc. (2005). A comparison of Network-on-Chip and Busses, [Online]. Available: http://www.design-reuse.com/articles/10496/a-comparison-of-network-on-chip-and-busses.html (visited on Aug. 18, 2015).

[6] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis, "M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores," in *Proceedings of the Twenty-first International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2016, to appear.

[7] M. K. Benhaoua, A. E. H. Benyamina, and P. Boulet, "Heuristics for Routing and Spiral Run-time Task Mapping in NoC-based Heterogeneous MPSOCs," *IJCSI International Journal of Computer Science Issues*, 1st ser., vol. 10, no. 4, p. 1694, Jul. 2013, ISSN: 1694-0814. arXiv: 1312.5764.

[8] S. Borkar, "Thousand Core Chips: A Technology Perspective," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07, New York, NY, USA: ACM, 2007, pp. 746–749, ISBN: 9781595936271. DOI: 10.1145/1278480.1278667.

[9] E. Carvalho, N. Calazans, and F. Moraes, "Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs," in *18th IEEE/IFIP International Workshop on Rapid System Prototyping, 2007. RSP 2007*, May 2007, pp. 34–40. DOI: 10.1109/RSP.2007.26.

[10] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, Feb. 2013, ISSN: 1551-3203. DOI: 10.1109/TII.2011.2173941.

[11] J. Castrillon, A. Tretter, R. Leupers, and G. Ascheid, "Communication-aware mapping of KPN applications onto heterogeneous MPSoCs," in *2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2012, pp. 1262–1267. DOI: 10.1145/2228360.2228597.

[12] J. Castrillón, *Programming Heterogeneous MPSoCs*. Springer International Publishing, 2014, ISBN: 978-3-319-00674-1 978-3-319-00675-8. DOI: 10.1007/978-3-319-00675-8.

[13] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: An integrated framework for MPSoC application parallelization," in *45th ACM/IEEE Design Automation Conference, 2008. DAC 2008*, Jun. 2008, pp. 754–759. DOI: 10.1145/1391469.1391663.

[14] *Circular Buffer*, in *Worl Heritage Encyclopedia*. [Online]. Available: http://www.worldlibrary.org/articles/circular_buffer (visited on Mar. 22, 2016).

[15] Clang team. (n.d.). "clang" C Language Family Frontend for LLVM, [Online]. Available: http://clang.llvm.org/ (visited on Aug. 11, 2015).

[16] C-Port Corp., *C-5 Network Processor Architecture Guide*, May 31, 2001. [Online]. Available: http://www.freescale.com/files/netcomm/doc/ref_manual/C5NPD0-AG.pdf?fsrch=1 (visited on Aug. 18, 2015).

[17] R. L. Cruz, "A calculus for network delay. I. Network elements in isolation," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 114–131, Jan. 1991, ISSN: 0018-9448. DOI: 10.1109/18.61109.

[18] ——, "A calculus for network delay. II. Network analysis," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 132–141, Jan. 1991, ISSN: 0018-9448. DOI: 10.1109/18.61110.

[19] P. Cumming, "The TI OMAPtm Platform Approach to SOC," in *Winning the SoC Revolution*, G. Martin and H. Chang, Eds., Springer US, 2003, pp. 97–118, ISBN: 978-1-4613-5042-2 978-1-4615-0369-9. DOI: 10.1007/978-1-4615-0369-9_5.

[20] W. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 684–689. DOI: 10.1109/DAC.2001.156225.

[21] D. Dasari, B. Nikolić, V. Nélis, and S. M. Petters, "NoC Contention Analysis Using a Branch-and-prune Algorithm," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3s, 113:1–113:26, Mar. 2014, ISSN: 1539-9087. DOI: 10.1145/2567937.

[22] E. W. Dijkstra, "Over de sequentialiteit van procesbeschrijvingen," n.d. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF.

[23] A. Dunkles and O. Schmidt, "Protothreads – Lightweight Stackless Threads in C," Mar. 2005. [Online]. Available: http://dunkels.com/adam/dunkels05protothreads.pdf (visited on Feb. 16, 2016).

[24] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Design Test of Computers*, vol. 18, no. 5, pp. 21–31, Sep. 2001, ISSN: 0740-7475. DOI: 10.1109/54.953269.

[25] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit, "Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multiprocessor System-on-chip (MPSoC)," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08, New York, NY, USA: ACM, 2008, pp. 212–217, ISBN: 9783981080131. DOI: 10.1145/1403375.1403427.

[26] P.-C. Hu and L. Kleinrock, "An Analytical Model for Wormhole Routing with Finite Size Input Buffers," presented at the 15th Intl. Teletraffic Congress, Jun. 1997.

[27] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information processing*, North Holland, Amsterdam, Aug. 1974, pp. 471–475.

[28] G. Kahn and D. MacQueen, "Coroutines and Networks of Parallel Processes," Jan. 1977.

[29] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM on VLSI, 2002. Proceedings*, 2002, pp. 105–112. DOI: 10.1109/ISVLSI.2002.1016885.

[30] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*, Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[31] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987, ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[32] Y. Li, S. Peng, and W. Chu, "Efficient Collective Communications in Dual-Cube," *J. Supercomput.*, vol. 28, no. 1, pp. 71–90, Apr. 2004, ISSN: 0920-8542. DOI: 10.1023/B:SUPE.0000014803.83151.dc.

[33] ——, "Metacube—a versatile family of interconnection networks for extremely large-scale supercomputers," vol. 53, no. 2, pp. 329–351, 2010, ISSN: 0920-8542. DOI: 10.1007/s11227-009-0297-2.

[34] T. Limberg, M. Winter, M. Bimberg, R. Klemm, M. Tavares, H. Ahlendorf, E. Matúš, G. Fettweis, H. Eisenreich, G. Ellguth, and J.-U. Schlüssler, "A Heterogeneous MPSoC with Hardware Supported Dynamic Task Scheduling for Software Defined Radio," presented at the Design Automation Conference 2009 (DAC'09), 2009.

[35] Z. Lu, A. Jantsch, and I. Sander, "Feasibility analysis of messages for on-chip networks using wormhole routing," in *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, vol. 2, Jan. 2005, 960–964 Vol. 2. DOI: 10.1109/ASPDAC.2005.1466499.

[36] G. Martin, "Overview of the MPSoC design challenge," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 274–279. DOI: 10.1109/DAC.2006.229245.

[37] D. Nadezhkin, S. Meijer, T. Stefanov, and E. Deprettere, "Realizing FIFO Communication When Mapping Kahn Process Networks Onto the Cell," in *Proceedings of the 9th International Workshop on Embedded Computer Systems: ARCHITECTURES, Modeling, and Simulation*, ser. SAMOS '09, Berlin, Heidelberg: Springer-Verlag, 2009, pp. 308–317, ISBN: 9783642031373. DOI: 10.1007/978-3-642-03138-0_34.

[38] B. Noethen, O. Arnold, E. P. Adeva, T. Seifert, E. Fischer, S. Kunze, E. Matúš, G. Fettweis, H. Eisenreich, G. Ellguth, S. Hartmann, S. Höppner, S. Schiefer, J.-U. Schlüßler, S. Scholze, D. Walter, and R. Schüffny, "10.7 A 105GOPS 36mm2 heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, Feb. 2014, pp. 188–189. DOI: 10.1109/ISSCC.2014.6757394.

[39] U. Y. Ogras and R. Marculescu, "Analytical Router Modeling for Networks-on-Chip Performance Analysis," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, Apr. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364440.

[40] M. Paganini, "Nomadik®: AMobile Multimedia Application Processor Platform," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '07, Washington, DC, USA: IEEE Computer Society, 2007, pp. 749–750, ISBN: 9781424406296. DOI: 10.1109/ASPDAC.2007.358078.

[41] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005, ISSN: 0018-9340. DOI: 10.1109/TC.2005.134.

[42] T. M. Parks, "Bounded Scheduling of Process Networks," 1995.

[43] G. L. Peterson, "Myths About the Mutual Exclusion Problem," *Inf. Process. Lett.*, vol. 12, no. 3, pp. 115–116, 1981. DOI: 10.1016/0020-0190(81)90106-X.

[44] B. Plateau and K. Atif, "Stochastic automata network of modeling parallel systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, Oct. 1991, ISSN: 0098-5589. DOI: 10.1109/32.99196.

[45] V. Rantala, T. Lehtonen, and J. Plosila, *Network on Chip Routing Algorithms*. 2006.

[46] A. Ray, T. Srikanthan, and W. Jigang, "Practical techniques for performance estimation of processors," in *Fifth International Workshop on System-on-Chip for Real-Time Applications, 2005. Proceedings*, Jul. 2005, pp. 308–311. DOI: 10.1109/IWSOC.2005.94.

[47] W. Sheng, S. Schürmans, M. Odendahl, M. Bertsch, V. Volevach, R. Leupers, and G. Ascheid, "A Compiler Infrastructure for Embedded Heterogeneous MPSoCs," in *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '13, New York, NY, USA: ACM, 2013, pp. 1–10, ISBN: 9781450319089. DOI: 10.1145/2442992.2442993.

[48] I. Sobel, "An Isotropic 3x3 Image Gradient Operator," 2015. DOI: 10.13140/RG.2.1.1912.4965.

[49] S. Stuijk and M. Geilen, "SDF3: SDF for free," *Proceedings - International Conference on Application of Concurrency to System Design, ACSD*, pp. 276–278, 2006, ISSN: 1550-4808. DOI: 10.1109/ACSD.2006.23.

[50] Tensilica, Inc., *Xtensa C/C++ Compiler*, Aug. 2002. [Online]. Available: http://ip.cadence.com/uploads/pdf/xcc_prod_brief_V_final_.pdf (visited on Feb. 16, 2016).

[51] ——, *Xtensa LX Microprocessor Overview Handbook*, 2004. [Online]. Available: http://ip.cadence.com/uploads/pdf/xtensalx_overview_handbook.pdf (visited on Feb. 16, 2016).

[52] ——, *Implementing a Memory-Based Mutex and Barrier Synchronization Library*, Application Note AN07-092-00, Jul. 2007. [Online]. Available: http://ip.cadence.com/uploads/pdf/Mutex_apnote.pdf (visited on Feb. 16, 2016).

[53] Texas Instruments. (n.d.). Keystone Device Architecture, Texas Instruments Wiki, [Online]. Available: http://processors.wiki.ti.com/index.php/Keystone_Device_Architecture (visited on Aug. 14, 2015).

[54] A. Tretter, "Communication-Aware Mapping and Scheduling of KPN Applications onto Heterogeneous MPSoCs," Diploma Thesis, Institute for Communication Technologies and Embedded Systems (ICE) RWTH Aachen University, Nov. 2011.

[55] W. Wolf, A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, Oct. 2008, ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923415.