THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Models and Methods for Development of DSP Applications on Manycore Processors

Jerker Bengtsson

School of Information Science, Computer and Electrical Engineering
HALMSTAD UNIVERSITY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden, June 2009

Models and Methods for Development of DSP Applications on Manycore Processors

Jerker Bengtsson
ISBN 978-91-7385-288-3

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2969
ISSN 0346-718X

Department of Computer Science and Engineering
Chalmers University of Technology
Technical Report No 62D.

Contact Information:

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Telephone: +46 (0)31 772 1000
Fax: +46 (0)31 772 3663
URL: http://www.ce.chalmers.se

School of Information Science,
Computer and Electrical Engineering
Halmstad University
Box 823 SE-301 18 Halmstad, Sweden
Telephone: +46(0)35 16 71 00
Fax: +46(0)35 12 03 48
URL: http://www.hh.se/ide

Printed by Chalmers Reproservice
Göteborg, Sweden, Maj 2009.

Abstract

Advanced digital signal processing systems require specialised high-performance embedded computer architectures. The term high-performance translates to large amounts of data and computations per time unit. The term embedded further implies requirements on physical size and power efficiency. Thus the requirements are of both functional and non-functional nature.

This thesis addresses the development of high-performance digital signal processing systems relying on manycore technology. We propose building two-level hierarchical computer architectures for this domain of applications. Further, we outline a tool flow based on methods and analysis techniques for automated, multi-objective mapping of such applications on distributed memory manycore processors. In particular, the focus is put on how to provide a means for tunable strategies for mapping of task graphs on array structured distributed memory manycores, with respect to given application constraints. We argue for code mapping strategies based on predicted execution performance, which can be used in an auto-tuning feedback loop or to guide manual tuning directed by the programmer.

Automated parallelization, optimisation and mapping to a manycore processor benefits from the use of a concurrent programming model as the starting point. Such a model allows the programmer to express different types and granularities of parallelism as well as computation characteristics of importance in the addressed class of applications. The programming model should also abstract away machine dependent hardware details. The analytical study of WCDMA baseband processing in radio base stations, presented in this thesis, suggests dataflow models as a good match to the characteristics of the application and as execution model abstracting computations on a manycore.

Construction of portable tools further requires a manycore machine model and an intermediate representation. The models are needed in order to decouple algorithms, used to transform and map application software, from hardware. We propose a manycore machine model that captures common hardware resources, as well as resource dependent performance metrics for parallel computation and communication. Further, we have developed a multi-functional intermediate representation, which can be used as source for code generation and for dynamic execution analysis.

Finally, we demonstrate how we can dynamically analyse execution using abstract interpretation on the intermediate representation. It is shown that the performance predictions can be used to accurately rank different mappings by best throughput or shortest end-to-end computation latency.

Keywords: parallel processing, manycore processors, high-performance digital signal processing, dataflow, concurrent models of computation, parallel code mapping, parallel machine model, dynamic performance analysis.

Sammanfattning

Avancerade inbyggda signalbehandlingssystem kräver ofta specialiserade och högpresterande datorplattformar. Med högpresterande menas att stora datavolymer och beräkningsintensiva algoritmer måste processas inom ett begränsat tidsintervall. Inbyggda system har generellt oftast krav på att understiga en viss fysisk storlek och att de måste vara energieffektiva.

Denna avhandling fokuserar på användning av flerkärniga processorer för konstruktion av högpresterande signalbehandlingssystem. Vi föreslår en datorarkitektur, i form av en två-nivå hierarki, för denna typ av signalbehndlingssystem. Vidare presenterar vi ett verktygsflöde baserat på metoder och analystekniker för automatiserad mappning av sådana tillämpningar på flerkärniga processorer med distribuerat minne. Mer specifikt, vi fokuserar på flexibla strategier för kravstyrd mappning av beräkningsgrafer på flerkärniga processorer organiserade i en två-dimensionell matrisstruktur. Vi förespråkar automatiskt eller manuellt styrd reglering av mappingsdirektiv, där beslut baseras på återkoppling av prognostiserade exekveringsegenskaper.

Automatiserad parallellisering, optimering och mappning på en flerkärnig processor kan förenklas avsevärt genom att utgå ifrån en lämplig parallell programmeringsmodell. En sådan modell tillåter en programmerare att uttrycka olika typer och olika kornigheter av parallellism samt beräkningskarakteristik, typiska för de typer av tillämpningar som adresseras. En sådan programmeringsmodell måste också abstrahera bort processorspecifika hårdvarudetaljer. En analys av basbandsprocessning i WCDMA radiobasstationer, som presenteras i denna avhandling, pekar ut dataflödesmodeller som mycket lämpliga för att programmera denna typ av tillämpningar, samt som exekveringsmodeller för program mappade på flerkärniga processor.

För att kunna konstruera portabla utvecklingsverktyg krävs det en lämplig modell av flerkärniga processorer samt en mellanrepresentation. Processormodellen behövs för att kunna utveckla maskinoberoende algoritmer för transformering och mappning av program. Vi föreslår en flerkärnig processormodell som fångar typiska hårdvaruresurser, såväl som dess grundläggande, parallella beräkningsoperationer. Vidare har vi utvecklat en multifunktionell mellanrepresentation, vilken kan användas som utgånspunkt för kodgenerering och för dynamisk exekveringsanalys.

Slutligen, denna avhandling visar ytterligare hur vi kan prognostisera dynamiska programegenskaper vid exekvering på flerkärniga processorer, genom att tillämpa abstrakt tolkning av mellanrepresentationen. Vi demonstrerar hur resultatet av dessa prognoser kan användas för att rangordna programmappningar enligt bäst periodisk genomströmning av data eller kortast svarstid.

Acknowledgements

There are several people who more or less have had some influence on my work and the decisions I have made during the very crooked path leading to completion of this thesis.

My main supervisor Bertil Svensson (*Duracell^{TM} Inside*) for his encouragement, experienced advices and never fading enthusiasm and positiveness. Not to forget, for all the time (weekends and evenings) spent reading papers, reports and my theses produced during these years.

My assistant supervisor, Verónica Gaspes for providing constructive criticism, concrete inputs and suggestions, and for being a good source for moral support and advices in issues of non-functional character in life and work.

Professor Edward Lee, for hosting and inspiring me greatly and giving me very valuable advices and suggestions during my very fruitful visit at University of California at Berkeley. Also, thanks to Man-kit "Jackie" Leung for discussing and providing solutions to problems that popped up during my first close encounter with Ptolemy and the code generator framework during my stay in Berkeley.

A large deal of this work has been done in various forms of cooperation and discussions with with Ericsson AB. I am very greatful to Anders Wass who got the leading role in the Ericsson relay race and hosted me and spent valuable time discussing and supporting me during and after my stays in Kista. The baton was later handed over to Henrik Sahlin and Peter Brauer who continued guiding me through the LTE jungle, which greatly helped me when struggling with how to find an academic entry point to real problems, finding the right levels of abstraction and to obtain valuable insights into the industrial world. These industrial connections have provided me with a great deal of insights and experiences which can only be read between the lines in this thesis.

Professor Marina Papatriantafilou, for being actively committed as external advisor in my support committee. Despite the limited room for meetings, there has always been a positive and friendly attitude, and a good use of this limited time leading to useful outcome.

Roland Thörner, who have absolutely nothing at all to do with this thesis, but who is a very nice bloke and friend in his best years having a rarely good sense of humor.

Finally, last but not least, my wife and constant brother in arms Hoai. Con kiên cõng con voi. Anh yêu em.

The work presented in this thesis has been funded by research grants form the Knowledge Foundation and Ericsson AB.

Contents

Lists of Appended Papers

This thesis is based on the work contained in the following papers:

Paper A Johnsson, D., Bengtsson, J., and Svensson, B. (2004). Two-level reconfigurable architecture for high-performance signal processing. In *Proc. of Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, pages 177-183, Las Vegas, USA.

Paper B Bengtsson, J. (2006). Baseband processing in 3G UMTS radio base stations. Technical Report IDE0629, School of Information Science, Computer and Electrical Engineering, Halmstad University, 2006.

Paper C Bengtsson, J., and Svensson, B. (2006). A configurable framework for stream programming exploration in baseband applications. In *Proc. of The 11th Int'l Workshop on High-Level Programming Models And Supportive Environments (HIPS'06) in conjunction with IPDPS (IPDPS 2006)*, Rhodes, Greece.

Paper D Bengtsson, J. and Svensson, B. (2008). A domain-specific approach for software development on manycore platforms. *ACM Computer Architecture News, Special Issue: MCC08 - Multicore Computing 2008*, 39(6):2-10.

Paper E Bengtsson, J., and Svensson, B. (2009). Manycore performance analysis using timed configuration graphs. To appear in *Proc of. Int'l Symp. on Systems, Architectures, Modeling and Simulation (SAMOS IX 2009)*, Samos, Greece.

Other Related Publications

Bengtsson, J. (2006a). *Efficient Implementation of Stream Applications on Processor Arrays.* Licentiate Thesis, Chalmers University of Technology, March 30, 2006.

Bengtsson, J., Gaspes, V., and Svensson, B. (2007). Machine assisted code generation for manycore processors. *Real-time in Sweden, Västerås (RTIS 2007).*

Bengtsson, J. and Svensson, B. (2008a). A set of models for manycore performance evaluation through timed configuration graphs. Technical Report IDE0856, School of Information Science, Computer and Electrical Engineering, Halmstad University, 2008.

Bengtsson, J. and Svensson, B. (2008b). Methodologies and tools for development of signal processing software on multicore Platforms . *Workshop on Streamings Systems in conjunction with 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41).*

Bengtsson, J., and Svensson, B. (2008c). A domain-specific approach for software development on manycore platforms. In *Proc. of First Swedish Workshop on Multicore Computing (MCC08).*

Chapter 1

INTRODUCTION

1.1 Microprocessor evolution: multicore versus manycore

The microprocessor industry has long been able to deliver increased performance of processors incited by Gordon Moore's predictions. This has to a large extent been possible through technology scaling of transistors and wires, continuously increased clock frequencies and adding more specialised functional units around a centralised processor architecture. However, an undesirable side effect of the technology scaling and the clock frequency race has been a rapid increase in power dissipation, which started to push cooling technology to its limit in the first years of 2000 [Carmean and Hall, 2001]. In 2005, Intel and AMD made a significant change in their directions on the chase for more power efficient performance, starting from a single core architecture and then doubling the amount of cores per semiconductor process generation. The term multicore has become highly associated with this concept. There are many arguments for why this rather narrow minded design shift will unlikely be the ideal solution for dramatic improvements in power performance [Asanovic et al., 2006]. Sustainable and evolutionary solutions for parallel processing must be sought from a broader perspective. As a response to this, the term *manycore* was coined at the University of California at Berkeley to distinguish new and innovative parallel processor architectures from the more limiting core doubling convention.

1.2 High-performance digital signal processing

High-performance signal processing systems, such as advanced radar systems, has long required computer architectures capable of delivering performance largely exceeding the performance of general purpose computer systems. Parallel embedded computer architectures were early been a must to manage the processing requirements in such systems, and future generations will keep pushing these requirements higher [Åhlander, 2007]. Another similar example is the signal processing required in radio base stations (RBS). In practice, an RBS is a highly advanced parallel computer system. A state-of-the-art platform for 3G WCDMA RBSs is typically designed using the latest ASIC and FPGA technologies to maximise the capability of serving as many number of concurrent lines as possible [Zhang et al., 2003]. Further, each RBS site typically contains many such parallel platforms.

Although there are many dissimilarities between these two examples of high-performance signal processing systems, there is at least one very important common denominator: development complexity and manufacturing cost. Radar systems are typically produced in smaller series, which makes the cost per system for ASIC development very high. In the telecommunication industry, the production series are much longer, which therefore makes the ASIC cost per system much lower compared to radars. However, the cost of developing ASICs increases dramatically for each semiconductor technology generation. Furthermore, new and more complex functionality keeps being added to each system generation. Thus, reducing manufacturing costs is one argument for industry's interest in using more commercial-of-the-shelf (COTS) manycore hardware.

Taking wireless telecommunication industry as example, the trend is that a growing part of the baseband platforms are implemented using programmable hardware technology. An RBS typically has a life expectancy counted in decades. Using programmable technology enables performance upgrading and forward-compatibility; new standardised functionalities and improved algorithms can be integrated after system roll-out. Moreover, different customers need different system solutions to meet their specific site requirements and have the ability to modify the network with respect to communication. This puts further requirements on platform scalability.

1.3 Scope of the thesis

This thesis addresses a few of the many problems related to system and software development for using manycore technology in embedded computer architectures for high-performance signal processing applications. We focus on array structured, software cached, distributed memory manycore processors. We believe that such machines provide good hardware scalability and good means for predictable timing.

We address a certain class of high-performance digital signal processing systems. We translate the term high-performance to real-time constrained processing of large amounts of data and computation intensive algorithms. We focus on embedded systems, thus there are further requirements on physical size and power efficiency. Therefore, the requirements are of both functional and non-functional nature. Applications that fit within this class are, e.g. baseband processing in radio base stations and signal processing in modern radar systems. Applications not belonging to this class are, e.g. weather simulations and scientific computations, since there typically is no strict requirements on response time.

Multi-processors have been widely explored in research since the 1980s and early 1990s. There has also been a great deal of research on parallel programming models. We have focused on finding solutions for manycore programming and code mapping based on this earlier work; more specifically, we have investigated methods and techniques based on dataflow models of computation.

The problem of mapping task graphs to a parallel processing hardware is well studied, and many solutions for automation of that problem exist. Given such a mapping, we focus on analysis techniques for the prediction of run-time properties of such task graphs on the category manycore targets addressed.

1.4 Problem description

We are interested in performance efficient DSP task graph mapping on highly parallel, generic manycore hardware. We are especially interested in self-timed task graph mappings, which put minimal requirements on run-time overhead for execution on manycore hardware [Lee and Ha, 1989]. To find solutions to this overall problem, we were motivated to address the following questions:

- **What are the trade-offs concerning system development using different paradigms of manycore processors?** Fine-grained manycores with reconfigurable interconnections theoretically offer a larger area performance compared to more coarse-grained manycores. However, reducing core hardware implemented functionality means that a certain part of the hardware resources must be used to configure corresponding functionality through software. Examples of such functionality can be memory address and cache logic and configuration of data paths (switching/routing functionality). How does this strategy of moving certain functionality to software - normally implemented in hardware - impact on practically achievable area performance?.

- **What are the typical computation characteristics and processing requirements in embedded high-performance DSP systems?** Embedded high-performance DSP systems typically contain large amounts of logical parallelism. These kinds of embedded systems are usually also associated with a set of non-functional constraints, for example real-time constraints. Furthermore, the hardware resources must be shared for many concurrent processing tasks. To be able to determine requirements on programming models, target hardware and mapping strategies for such hardware, it is necessary to analyse typical algorithm characteristics, logical parallelism etc. from a system perspective.

- **What is a suitable parallel model of computation for DSP applications?** Automated mapping of application software requires well-defined parallel models of computation. Fewer and fewer are those who still believe that the non-concurrent, shared memory programming model offered by the C language is a good starting point for automated mapping to parallel hardware. A suitable parallel model of computation must offer concurrency but also provide means for manycore hardware independence.

- **What is a suitable machine abstraction for manycore processors?** Tools and application software must be portable. This requires suitable intermediate representations for modular tool building. Further, the target embedded systems are also associated with constraints of a non-functional nature. Thus, the optimisation of DSP task graphs to manycore hardware is a complex, multi-objective task with many trade-offs. To deal with optimisation coupled to non-functional constraints, we believe that such intermediate representations need to capture time and offer means for analysis of dynamic execution costs.

1.5 Research goals and approach

The overall goals of this thesis work are:

1. to investigate trade-offs in signal processing implementation using different paradigms of manycore technology;

2. to investigate models, methods and techniques for computer assisted mapping of signal processing task graphs on manycore hardware; and

3. to develop techniques for analysing and predicting dynamic execution costs of DSP task graphs on manycores.

The research was initiated by studying and evaluating emerging manycore architectures [Bengtsson and Lundin, 2003]. To be able to analyse system design trade-offs related to the choice of hardware technology, we later conducted a study on parts of the physical layer processing in 3G radio base stations. For the parts of the system studied, we found the synchronous dataflow model of computation to offer a good match with the system and development requirements and with the manycore hardware addressed. Considering industrial requirements on tool and software portability, we further addressed methods and techniques for automated software mapping on parallel hardware. To abstract manycore hardware and to predict dynamic execution costs related to non-functional properties, we identified the need for a new manycore machine model providing realistic execution cost predictions and a multi-functional intermediate representation as an important research topic.

1.6 Contributions of the thesis

The contributions of this thesis are as follows:

- **Hierarchical processor architecture for high-performance signal processing**
  We propose a hierarchical manycore processor architecture for multi-dimensional signal processing platforms. We have investigated some of the trade-offs related to different core granularities by comparing two paradigms of existing manycore processors: a coarse-grained array of tightly coupled RISC cores and a fine-grained reconfigurable array of single instruction ALUs. In the latter, both control flow and data operations must be mapped spatially using one or several of ALUs. Given a certain chip area and assuming the same technology, our comparisons show that the finer grained array has a potential peak performance of almost a factor of 7 higher than the coarse-grained processor. Furthermore, we studied the impact on performance when large amounts of ALU resources need to be allocated for algorithm control flow. Our experiments shows that, even if using up to as much as 75 percent of the ALUs for algorithm control flow, it would still be competitive with the peak performance of the coarse-grained alternative.

- **Processing requirements and characteristics in WCDMA baseband**
  We provide a comprehensive study of the complete set of processing functions specified for 3G WCDMA downlink baseband. In the study, we make an analysis of different types of potential parallelism. We discuss the functionality and characterise the intra-algorithm computations and data representations. On the basis of the study, we find that stream-oriented models of computation constitute a very good match for describing the task and the pipeline parallelism that we identify on the function level in the WCDMA processing chain. On the intra-algorithm level, we also identify and discuss potential mapping on SIMD and MIMD parallel hardware. Moreover, we conclude that the requirements on instruction-level computations are mostly of a logical, rather than an arithmetical, nature.

- **Modelling framework for SDF based languages**
  We provide a framework specialised for the modelling of stream-oriented languages based on the synchronous dataflow model of computation. We used the StreamIt language in order to reason about implementation of WCDMA downlink processing using stream-oriented languages. On the basis of the WCDMA study, we identify a number of weaknesses related to expressiveness in the StreamIt language and propose language extensions

related to data types and instruction level computations. Further, we introduce the notion of a control mode - for periodical reconfiguration of actors - and control streams - for distribution of actor reconfiguration parameters. We demonstrate language improvements by modelling an experimental language called *StreamBits*. In particular, we demonstrate how we can extend the StreamIt language with syntactic means to express computations on variable length bit-string data types.

- **Models for manycore performance evaluation**
  We developed a set of models to be used as a part of a manycore mapping and code generation tool. Manycore processors are described using a machine model that captures essential performance measures of array structured, tightly coupled manycore processors. Moreover, we developed a timed intermediate representation for manycore targets in the form of a heterogeneous dataflow model. We show how the intermediate representation is constructed for abstract interpretation, given a model of the application (SDF) and a specification of the machine as input. Thus, the use of the timed intermediate representation is two-fold: 1) we can by means of abstract interpretation obtain feedback about the run-time behaviour of the application and 2) we can use this IR as source for code generation to parallel targets.

- **Rank based feed back tuning using performance predictions**
  We outline a design flow for iterative tuning of dataflow graphs on manycores using predicted performance feed back. The tool has two purposes: 1) to provide means for early estimates of application performance on a specific manycore and 2) to provide means for a programmer or an auto-tuner to tune mapping decisions on a manycore, based on feed back of predictions of a mapped application's dynamical behaviour. We evaluate the accuracy of the predictions calculated by our tool by making comparisons with measurements on the Raw processor. We show that we can fairly accurately predict both on-chip and off-chip communication costs. Furthermore, we show with our experiments that the tool's predictions can be used to correctly rank parallel mappings, by highest throughput and shortest end-to-end latency, when tuning an application implementation for such non-functional constraints.

1.7 Outline of the thesis

The thesis consists of two parts: a summary and a set of five appended papers. The six following chapters in the summary of the thesis are briefly summarised below.

- Chapter 2 summarises our work on investigating a domain specific computer architecture for embedded high-performance digital signal processing. We outline and motivate our proposal of a two level hierarchical architecture. Further, we make an analysis of area performance trade-offs associated with the choice of manycore structure and its core granularity.

- Chapter 3 presents a study of downlink baseband processing in 3G radio base stations. We summarise the function flow and its functional requirements, the inter as well as intra-function characteristics and we identify potential sources of logical parallelism. Further we discuss potential mapping of baseband use cases on common types of parallel processors.

- Chapter 4 introduces stream processing and dataflow models of computation in particular. Motivated by the baseband study in Chapter 3 and the type of manycore architectures discussed in Chapter 2, we especially focus on the synchronous dataflow model of computation. Furthermore, we briefly discuss data types and language constructions that we find useful for implementing baseband processing in languages based on the SDF model of computation. Finally, we end the chapter by discussing related work on stream-oriented languages.

- Chapter 5 presents our work on models for abstracting manycores and DSP applications. We further propose a manycore intermediate representation suitable for code generation and analysis of non-functional properties of SDF graphs when mapped on a particular manycore. The chapter is finalized with related work on techniques and methods for mapping task graphs on parallel processors. The chapter is ended by discussing further related work.

- Chapter 6 contains conclusions and suggestions for further work.

Chapter 2

HIERARCHICAL ARCHITECTURE FOR EMBEDDED HIGH-PERFORMANCE SIGNAL PROCESSING

This chapter summarises our work on investigations of a domain specific computer architecture suited for embedded high-performance digital signal processing. We motivate and outline our proposal of a two level hierarchical architecture. Further, we focus on analysing area performance trade-offs associated with the choice of computation structure and its granularity in the lower abstraction level of our proposed architecture, as described in [Paper A].

2.1 Embedded high-performance DSP systems

We will first discuss some general system requirements that must typically be considered for a computer architecture for the targeted application domain. We discuss these requirements related to two examples of concrete applications within this domain: signal processing in RBS and in radar systems.

Parallelism Embedded high-performance signal processing applications typically consist of several types and granularities of parallelism. From a system perspective, modern radars are developed to be capable of operating in different modes and using multiple parallel antenna inputs. In an RBS, the goal is to maximise the number of concurrent user channels having different requests on the types of services. The signal processing in a radar or an RBS constitutes pipelined function flows, exposing task, data and pipeline parallelism. Each function can further contain different amounts of fine-grained instruction level parallelism. We aim for an architecture that is flexible and that can be dimensioned for heterogeneous parallelism of variable amounts.

Non-functional properties Both radar systems and radio base stations are real-time systems. Thus, one important type of non-functional constraints associated with such systems is time. New data are continuously streamed into the system by a certain periodicity (throughput) and its output must be produced within a certain time (end-to-end latency). A sufficient mapping of a function flow must also fulfil the specified system timing requirements. We must be able to offer appropriate parallel mapping strategies in order to process function flows of different structures and with different computation loads, with respect to some given timing requirements.

Scalability A set of, often computationally demanding, functions is applied on multi-dimensional arrays of data collected from multiple antenna streams. Embedded high-performance systems are typically built using boards with multiple chips and even using multiple boards. The sizes and the dimensions of the data shapes processed by the function flows typically vary depending on for example the radar task or the number of connected users and the specific services each is requesting from the wireless network. Thus, a domain specific computer architecture should be scalable in order to enable the designer to dimension a system for different requirements.

System reconfiguration A system must be able to dynamically adapt to periodically changing computation requirements. From a service scheduling point of view, the number of concurrent users and different types of service requests in an RBS can change frequently by a certain (often very short) periodicity. In practice, the size and the structure of the function flows and the workload for each function change. Since these changes are non-deterministic, the resource allocation must be done dynamically. In conclusion, the computer architecture should be able to allow for fast reconfigurations (dynamic resource allocation) of the processing resources to handle varying structures of the function flows and varying workloads of the functions.

2.2 A hierarchical manycore architecture

In [Paper A], we propose a mesh structured computer architecture using a two level hierarchical abstraction: a macro level structure and a micro level structure, see Figure 2.1. A mesh structure is easily scalable for different sizes of parallel structures, both on chip level as well as on the board level. Furthermore, the two level hierarchy provides an abstraction for both homogeneous and heterogeneous multiprocessor systems.

The macro level is a mapping abstraction for application and function level parallelism. A macro node can very well abstract, for example, a programmable manycore structure as well as a hardware implemented function accelerator.

The inside of a macro node constitutes the micro level structure. Depending on the type of micro level structure, the architecture allows exploitation of further task, pipeline, data and instruction level parallelism, in order to compute the mapped functions as efficiently as required.

In our work, we have primarily focused our investigations on micro level structures in the form of homogeneous manycore technology. This choice offers a highly flexible mapping space and also simplifies programmability and code mapping.

2.3 Reconfigurable micro level structures

In the most recent decade there has emerged a variety of different kinds of parallel and so called reconfigurable processor architectures. There is no well defined taxonomy for categorisation of such processors. Neither is there a good

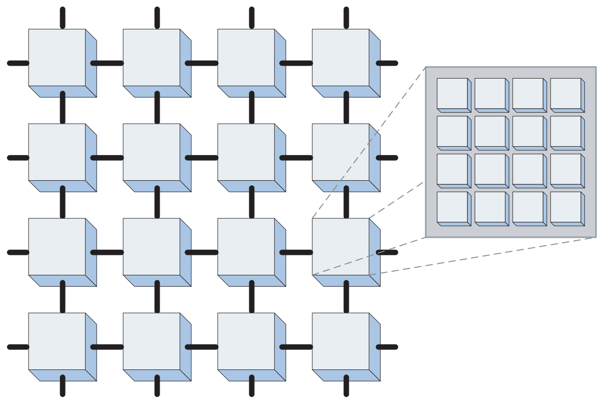


Figure 2.1: A two level manycore hierarchy. Macro level nodes can be either specialised cores, dsp or, as illustrated by the figure, a micro level manycore structure.

definition, in our opinion, of what clearly distinguishes a ”reconfigurable” processor from a ”programmable” processor. Instead, such processors are usually coarsely compared by the granularity of the processing elements (PE), how the PEs (or cores) are networked to form a parallel computing structure and how computations and the mapping of programs are done [Mangione-Smith et al., 1997].

The probably most explored and also most fine-grained of (re)configurable structures are field programmable gate arrays (FPGA). However, such fine-grained bit-level structures tend to require large amounts of logic to implement arithmetic operations, such as multiplication, on word length data. This issue has been addressed by the FPGA industry through embedding specialised word level arithmetic units, such as multipliers, in the fine-grained FPGA logic. However, the reconfiguration times in FPGAs are very long and therefore not fitted for system requirements of fast run-time reconfiguration.

Word level reconfigurable manycore processors represent one class of interesting manycore processors for signal processing functions. Interestingly, research investigations have shown that it is possible to achieve performances comparable with FPGAs for many applications requiring bit-level computations [Wentzlaff and Agarwal, 2004].

Many parallel processors have been designed in the form of an array structure, where the PEs (cores) are interconnected via a k-ary n-cubical network (that is, a network of n dimensions and k cores in each dimension). Considering wire densities in VLSI implementations of such networks, it was shown by William Dally in the early 1990s that low dimensional n-cubical networks yield lower communication latencies and higher hot spot throughput [Dally, 1990]. Research has later also suggested that it can be possible to build and efficiently utilise two dimensional array structures with thousands of coarse-grained cores [Moritz et al., 2001].

On the basis of these research findings and the characteristics of the application domain, which we described in Section 2.1, we have focused our interest on micro level structures in the form of mesh structured manycores.

2.4 Evaluation of a reconfigurable micro level structure

Regarding the type and design of micro level nodes, the main issue is to choose a manycore structure that offers a mapping space flexible enough for exploitation of different types and levels of potential parallelism within different functions. There are several aspects to consider, for example:

- What are the area performance trade-offs with respect to core granularity?
- What are the functional requirements at the micro level nodes?
- What is the cost for implementing control flows and address logic?

In [Paper A] we mainly addressed these three aspects to get some indicative answers. We chose to compare two different categories of existing manycore structures.

The first one, the XPP array architecture from PACT shown in Figure 2.2, is a homogeneous MISD (multiple instruction stream single data stream) array processor consisting of word length ALUs, which offers a spatial reconfigurable mapping space for algorithms [Baumgarte et al., 2001]. Using simple cores without instruction or data memory naturally enables a large amount of cores to be stamped out on a chip and thereby offers a large amount of instruction level parallelism. Similar to FPGAs, the array of cores must be reconfigured to switch from one algorithm to another. Further, each type of ALU operation is performed within one clock cycle. The output from a computing core is available for its nearest neighbours in the proceeding clock cycle. The XPP array has on-chip data memory distributed over a set of memory elements. These elements can also be combined to form larger logical memories, offering a larger address range when needed. The array has no dedicated controller logic to handle memory. Thus, memory read and write operations have to be implemented using one or a set of ALUs.

The other, the Raw micro processor, shown in Figure 2.3, is a more coarse-grained MIMD (multiple instruction stream multiple data stream) array of

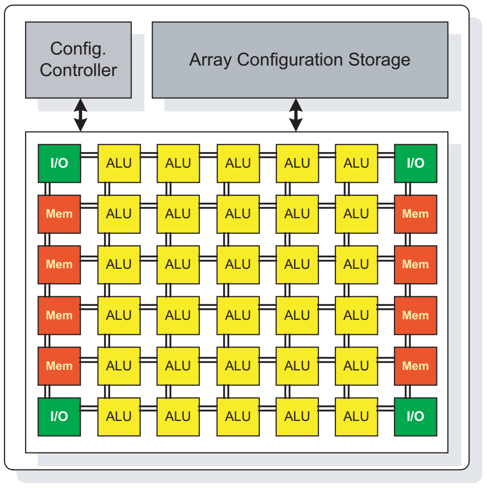


Figure 2.2: The figure illustrates the XPP dataflow processing array, which is a reconfigurable MISD type of manycore.

fewer (16) cores, compared to the XPP array, but offers both temporal and spatial mapping of algorithms [Taylor et al., 2002]. The cores are MIPS processors with a slightly modified instruction set. Each core has instruction memory and a local (private) data cache. Moreover, Raw cores have floating point units. The cores are tightly coupled via four physical on-chip scalar operand networks - two statically and two dynamically routed - mapped via the register files. This means that the network can be treated as source and destination operands of an instruction (hence, the name scalar operand networks). The static network routers are programmable, so that static communication paths can be set up between cores. The dynamic networks are wormhole routed message passing networks, for core and memory communications of less deterministic nature.

2.4.1 Evaluating area performance

We have studied a specific implementation of the XPP array architecture - the XPP-64A - and the first prototype implementation of Raw. It is difficult to make a fair area performance comparison between Raw and XPP-64A since they are of different types of architecture and are implemented in different process technology. For example, large parts of the area for each Raw tile is used for the floating point unit and for larger local data caches (Raw has in total 512 Kb compared to XPP's 12 Kb) and the processors are implemented in different silicon processes. Raw cores use 32 bit word length, while XPP-64As ALUs have 24 bit word length. Furthermore, the Raw processor and the

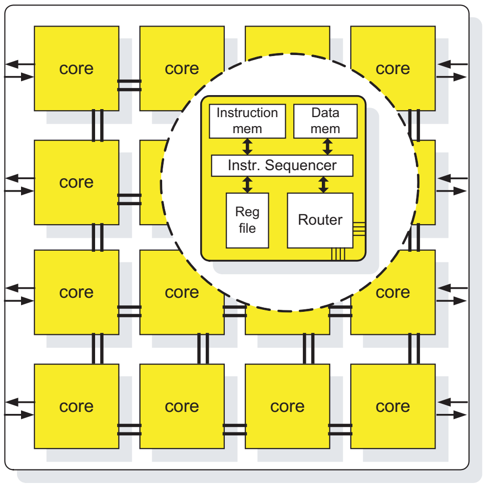


Figure 2.3: The figure illustrates the Raw micro processor, which is a MIMD type of manycore array.

Processor	**Cores**	**Area** (mm^2)	**Lithography** (μm)
Raw	16	250	0.15
XPP	64	32	0.13
XPP Scaled	320	250	0.15

Table 2.1: Comparison of instruction level parallelism per area for Raw with XPP and the scaled XPP.

XPP-64A run at different clock frequencies. It should also be mentioned that Raw has not in the first hand been implemented with the aim to maximise the number of cores. However, we still find it very interesting to make a coarse estimation of instruction level parallelism and performance per area to be able to reason about trade-offs between core complexity (size), performance and resource utilisation.

In the first part of our study, we evaluated area performance. Table 2.1 shows the number of cores on XPP-64A, the chip area and the lithography before and after up-scaling it to the same chip area and lithography used for Raw. We denote the scaled XPP array XPP Scaled. In our estimates, we calculated with a linear scaling from $13\mu m$ lithography up to $15\mu m$, which was used for the first implementation of the Raw prototype. The result is naturally a much larger spatial computation space, offering 320 parallel instructions compared to the 16 of Raw.

Processor	**Frequency** (MHz)	**Peak Perf.** $(GOPS)$
Raw	225	3.6
XPP	64	4.1
XPP Scaled	64	24.5

Table 2.2: Comparison of peak performance for Raw with XPP and the scaled XPP.

Further we compared peak performances for Raw, XPP-64A and XPP Scaled, see Table 2.2. Due to the uncertainty regarding how fast it is physically possible to clock XPP Scaled, we chose to use the same clock frequency, 64 MHz, as is documented for the XPP64-A. However, earlier and larger implementations of the XPP array have been clocked at least at 100 MHz. It should also be mentioned that PACT has aimed for a low clock frequency for the XPP-64A in order to provide a low power performance ratio. Similarly, for the Raw prototype, the reported clock frequency was 225 MHz.

2.4.2 Evaluating resource utilisation

An obvious trade-off, by making cores simpler as in the XPP architecture, is that some amount of cores has to be used to implement algorithm control flows and memory control logic. In the second part of our study, we evaluate the resource utilisation ratio between control flow and data computations and what impact this ratio has on reachable performance. To evaluate resource utilisation, we implemented a Radix-2 FFT on the XPP-64A. FFT is an important class of algorithms used in DSP. The Radix-2 was chosen because it has a fairly complex dataflow pattern, requiring a large amount of control flow [Proakis and Manolakis, 1996].

The Radix-2 FFT is logically computed in $log\ n$ stages. We implemented an FFT module on the XPP-64A array, optimised for a stream throughput of one complex sample per clock cycle. This module can either be spread out in n instances, to create pipelined computation of the FFT, or as a single instance, where the n stages are iteratively computed by a single FFT module. Figure 2.4 shows a high level block schematic of the implemented Radix-2 FFT module and how the utilised ALU resources are distributed in the form of control flow and of data computations.

A considerable part, 76% of the resources used, is used to implement algorithm control flow and memory management (address generation, double buffer synchronisation etc.). The remaining 26% used for computations corresponds to the butterfly computations. This relation in resource utilisation might seem highly inefficient, but it is important to bear in mind that ordinary micro processor program code requires portions of the code to do address calculations,

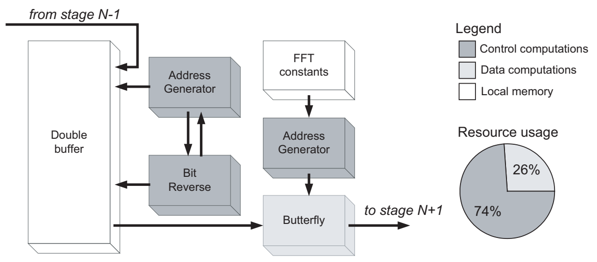


Figure 2.4: The block diagram to the left illustrates the implemented FFT module. The diagram to the right shows the amount of cores used for control flow and data flow respectively. Local memory was needed to be used for both double buffering, in order to relax synchronisation between FFT modules when mapped in a pipelined fashion, as well as for storing pre-calculated FFT constants

control flow (loops and conditional statements etc.) and synchronisation. Turning back to our calculations presented in Table 2.2, we can quickly establish that using only 15% of the XPP Scaled's resources still means that the XPP array is competitive compared to the peak performance of a more traditional CMP (chip multi processor) architecture (3.6/24.5). Even if it would probably be easier in practice to come close peak performance on Raw than on XPP, a part of the computations would still be temporal control flow, which will naturally decrease the throughput. Furthermore, our FFT implementation is capable of computing one complex valued sample per clock cycle, which, if not impossible, would at least be very difficult to achieve on a CMP due to longer communication latencies and a much more limited spatial mapping possibility.

2.5 Implications for the further work

Many algorithms in the DSP domain require less complex control flow compared to the FFT algorithm and can be mapped with good resource utilisation, thus reaching high performance. Other algorithms will need to utilise run-time reconfiguration when using micro level structures such as the XPP array. Johnsson et al. addressed the run-time reconfiguration aspects by analysing the requirements on speed of run-time reconfigurability from the perspective of a radar application [Johnsson et al., 2005]. This study concluded that reconfigurable array processors, like the PACT XPP, have the potential of managing the reconfiguration times as required in the radar use cases.

In our opinion, a more serious issue with manycore processors, such as the XPP and Raw, is that the programming complexity rapidly increases with the number of cores. The implementation of the Radix-2 FFT required hardware-near programming (using the native XPP assembly language) in order to optimise the dataflow (throughput) with respect to maximum I/O capacity. Further, the spatial placing and routing of the algorithm in whole was needed to be done fully manually. This low level programming proved to be very time consuming, error prone and very difficult in terms of debugging. Moreover, for industry, it would be strategically very risky to introduce such tight dependencies to a specific processor architecture. This is one of the main reasons why industry has been unwilling to use commercially available manycore technology.

The results of the experiments conducted in this study further motivated us to focus our research on parallel models of computation and programming methodologies for machine independent application development for array structured manycores.

Chapter 3

ANALYSIS OF WCDMA BASEBAND PROCESSING

This chapter is based on the contents of [Paper B], which is a study of one concrete, complex and relevant, industrial application: baseband processing of the WCDMA downlink in 3G radio base stations. The study has three main goals and contributions: 1) to summarise and provide a complete overview (abstraction) of the functional requirements in downlink baseband processing, 2) to characterise function level characteristics (such as data dependencies), intra function characteristics (such as data representation and instruction level computations) and, finally, 3) to identify potential exploitation of parallelism. The chapter will summarise the more important contents of the study and a give concluding discussion of the matching potential with stream/dataflow models of computation.

3.1 WCDMA and the UTRAN architecture

There are several 3G enabling technologies, such as EDGE, CDMA 2000 and WCDMA. The wideband code division multiple access (WCDMA) radio technology is the universal standard chosen by the 3GPP standardisation organisation for 3G mobile networks [Holma and Toskala, 2004]. The WCDMA technology constitutes the core of the UMTS terrestrial radio access network (UTRAN) architecture shown in Figure 3.1. The UTRAN architecture is built upon one or several radio network subsystems (RNS). An RNS is a sub-network comprising a radio network controller (RNC) and one or several Node B's. Node B is the terminology used by the 3GPP for a radio base station (RBS). At service requests from users, the RNC is responsible for setting up physical radio channels provided by the Node B.

3.1.1 The RBS

The RBS[1] implements the lowest layer of the UTRAN layers, i.e. the physical layer and the radio. The functionality of the RBS can generally be described as a mapping procedure of logical channels from higher layers (L2 and above) to the physical radio channels (L1). In the downlink (from the RBS to the user equipment), data frames from higher layers are encoded, multiplexed and modulated before radio transmission. In the uplink (from the user equipment to the RBS), physical channels are demodulated, de-multiplexed, decoded and mapped onto higher layer frame structures. More briefly, an RBS can be viewed as the modem in wireless telecommunication networks.

[1]In the rest of the thesis we will use the term RBS when referring to Node B.

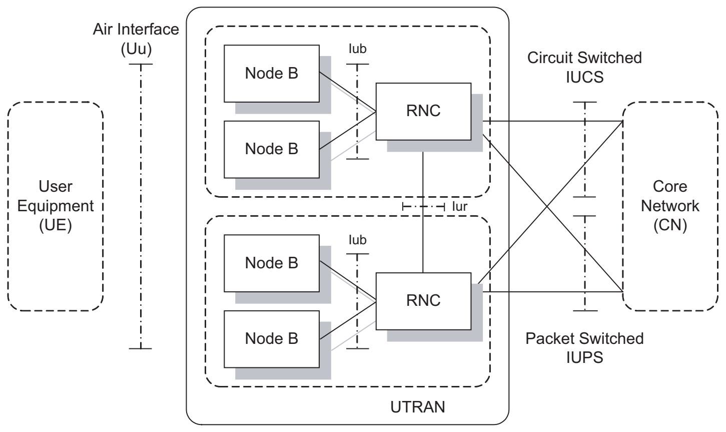


Figure 3.1: Utran.

3.1.2 Downlink transport channel multiplexing

The baseband processing of the downlink in an RBS constitutes a pipelined function flow, see Figure 3.2. There are several types of transport channels: control, shared and dedicated user channels. The analysis in our study is limited to the processing of user dedicated transport channels (DCHTRCH). Furthermore, the processing is performed with different processing rates at different stages in the baseband: symbol rate and chip rate. Symbol rate corresponds to the rate of information bits, i.e. each information bit in the user data streams corresponds to one symbol. At the chip rate, each information bit (symbol) has been spread out on a longer code bit sequence. Our study covers only the symbol rate functions, i.e. we do not cover functionality such as code spreading, modulation and the radio.

3.2 Downlink processing analysis

A single user can be allocated one or several dedicated transport channels (as illustrated by the 1 to n branches in the abstract task graph in Figure 3.2), depending on the type of service requested. In the mid stage in the graph, the user channels are multiplexed into a single composite transport channel (CC-TRCH). Then, depending on the required bandwidth, the composite channel is segmented and mapped on a number of physical channels (the 1 to m output branches in the figure). The structure of the task graph for each individual user is static during the service session.

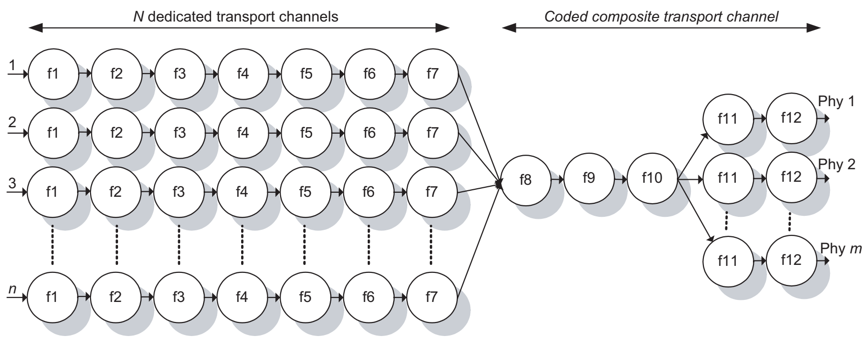

N dedicated transport channels
Coded composite transport channel
1
2
3
n
f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
Phy 1
Phy 2
Phy m

Figure 3.2: Abstract task graph describing the symbol rate function flow in WCDMA downlink. The input of N user transport channels is multiplexed (at f8 in the figure) and mapped to M physical channels (at f10).

ID	**Function**
f1	Cyclic redundancy check (CRC)
f2	Block concatenation and segmentation
f3	Channel coding
f4	Rate matching
f5	First DTX insertion
f6	First interleaving
f7	Frame segmentation
f8	Channel multiplexing
f9	Second DTX insertion
f10	Physical channel segmentation
f11	Second interleaving
f12	Physical channel mapping

Table 3.1: The table lists the types of downlink functions corresponding to the graph in Figure 3.2. These functions are described in [Paper B].

3.2.1 Types of parallelism

To avoid confusion about what we mean with certain types of logical parallelism in an application, we start by making a few definitions of such types. We refer to the abstract program implementing the downlink processing as the task graph. Nodes in the task graph correspond to functions, having a private address space, and edges represent the data dependencies between the functions (communication). The functions are considered to be infinitely repeated. We make the following definitions of logical program parallelism, using a function as the basic unit of computation:

Task parallelism. Two functions that are on separate branches in a task graph, in a way such that the output of one function never reaches the input of the other, are said to be *task parallel*.

Data parallelism. Any function that can be instantiated in multiple copies, such that no data dependency exists between the instances, is said to be *data parallel*.

Pipeline parallelism. Chains of functions having a producer consumer dependency are said to be *pipeline parallel*.

3.2.2 Real-time characteristics

An RBS is a real-time system. The correctness of the functionality is not only dependent on the logical correctness of its computations but also on at which point in time the system is able to consume and produce input and output data. In the case of an RBS, it means that certain processing requirements must be fulfilled to manage air and RNC interface compatabilities and to provide a certain level of quality-of-service.

The system's input data rate is determined by the transmission time interval (TTI), see Figure 3.3 and the size and number of transport blocks carrying payload data. The output must be produced with respect to the given radio frame rate (10 ms in WCDMA), at which information is transmitted over the air. Services such as voice transmissions naturally put requirements on computation end-to-end latencies for user comfort. Furthermore, since re-transmission requests are handled by higher layers, there is naturally also a requirement on end-to-end computation latency, in order to meet a certain quality-of-service (effective bandwidth) for varying radio conditions.

Remarks on the real-time aspects Considering mapping of different task graphs on parallel hardware, we see a need to explore different parallel mapping strategies, allowing optimisation not only with respect to workload (number of concurrent users and services) but also with respect to the given timing requirements.

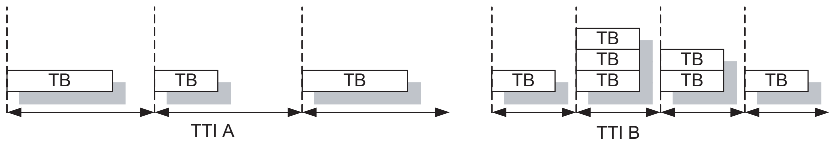


Figure 3.3: The transmission time interval (TTI) is always fixed during a transmission. For transmission of single transport blocks (TB), the length of transport blocks can be variable, as can be seen to the left in the figure. For transmission of transport block sets, all transport blocks must be of the same length, as can be seen to the right.

Dynamic	Transport block size	0-5000 bit
	Transport block set size	0-200000 bit
Semi-static	Transmission time interval	10,20,40,80 ms.
	Channel coding type	no coding, convolution, turbo
	Code rates (convolution)	1/2 or 1/3
	CRC size	0,8,12,16,24

Table 3.2: Transport format attribute options used to configure the baseband functions at the sender and the receiver.

3.2.3 Parameter configuration

The processing functions are individually configured using a set of transport format attributes. Table 3.2 shows these parameters. The dynamic attributes can be altered each TTI and are mainly used for configuring the functions to adjust to the current data to be processed. However, all transport blocks belonging to the same transport block set within the same TTI must be of the same size. We will make use of this information when discussing mapping of the logical parallelism on certain types of parallel hardware. The semi static attributes are configured once when setting up a service session. These parameters mainly configure the mode of functions and the TTI.

Remarks on Parameter configuration It is important to consider the configuration of parameters from at least two aspects. Firstly, considering a distributed software implementation on parallel hardware, there will be a need for a suitable representation of configuration data, and especially how to efficiently deal with distribution of the configuration data. Secondly, different configurations of functions in parallel transport channels indicate a limitation on certain data parallel hardware (SIMD).

3.2.4 Function level parallelism

When studying the abstract task graph for the downlink, as was shown in Figure 3.2, two types of coarse-grained parallelism are naturally exposed: task parallelism and pipeline parallelism. All data dependencies between functions are of the producer consumer type. The channel multiplexing function (node f8 in Figure 3.2) constitutes the first logical point of synchronisation in the downlink task graph. Before the multiplexing function, each branch of the graph (nodes f1 to f7) can be computed in a task parallel way, and the functions within each branch can be computed in a pipeline parallel way. The functions processing the composite transport channel (nodes f8 - f10) are pipeline parallel. After physical channel mapping (node f10), the functions (nodes f11-f12) can be mapped task parallel (if several physical channels are used) and pipeline parallel within each physical channel flow.

Remarks on task and pipeline parallelism On an abstract level, both potential task and pipeline parallelism are naturally exposed. To further determine potential data parallel mappings of the downlink task graph, it is necessary also to analyse intra-function data dependencies.

3.2.5 Intra function characteristics

In our analysis we mainly consider logical parallelism exposed in the specification of the standard. To examine potential intra algorithm parallelism, we need to analyse the computation characteristics of the standardised processing functions. For a more detailed analysis of the functions, the reader is referred to [Paper B]. Here we give a short summary of the characteristics.

Data dependencies The data streaming through the functions have the form of logically serial streams of binary information symbols. The functions specified for the downlink functions are therefore, to a large extent, performing bit serial computations. This means that there is in general no obvious fine-grained data parallelism exposed within the algorithms. However, most of the functions allow bit parallel mapping and computation of the data using word length data types. None of the symbol rate functions has data dependencies between iterations, i.e. between the processing of consecutive TTIs. This means that most of the task parallel functions in the downlink computation graph potentially also, from a logical point of view, are data parallel.

Instruction level computations The computations on the data are primarily of a logical nature. The functions performing different kinds of data coding, such as CRC, convolution and turbo coding, are dominated by logical arithmetic and shift operations. Computing such functions bit serially using bit parallel hardware is not an efficient usage of the hardware. Therefore, software implemented solution of such algorithms typically make use of pre-calculated look-up tables whenever possible. Bit serial calculations are thus transformed to bit parallel masking and memory reads and writes. Other functions mainly reshape the block representations of the data streams, for example the block concatenation and the segmentation functions. Further, another category is the interleaving and rate matching functions, where data are either scrambled or modified on a bit level basis.

Remarks about intra function characteristics We have studied the downlink computation graph from a logical point of view, given by the 3GPP specifications. We conclude that logical parallelism is dominates a on function level. On the intra function level, performance gain is related more to an acceleration of computations. However, a thorough analysis of opportunities for instruction parallel computations requires implementation studies of the complete task graph.

3.2.6 3G service use cases

We selected two service use cases (given by the 3GPP standard) that have different processing requirements on the baseband: one service configuration for voice transmission and the other for arbitrarily high bit rate data transmissions. These use cases are used to analyse the processing characteristics of the downlink functions.

Adaptive Multi Rate voice transmission Adaptive multi rate (AMR) is the technique for the coding and decoding of dynamic rate voice data included in the UMTS[2]. This technique allows dynamic alternations of the bit rate for voice transmissions during the service session. The output of the AMR encoder is arranged in three classes of bit streams (A,B and C), depending on how important specific bits are for quality. The A bits are the most important and the C bits are the least important. In this use case, each of the three bit stream classes is mapped on its own dedicated user transport channel. The output stream is mapped on a single physical channel.

High bit rate data transmission The 3GPP standard specifies a set of user equipment (UE) classes with different radio access capabilities[3]. These capability classes define the data rates and services that must be supported for a UE of a certain class. We used the requirements for the highest capability

[2]Technical Specification Group Radio Access Network; Services provided the physical layer, TS 25.302 (Release 5), www.3gpp.org

[3]Technical Specification Group Radio Access Network; UE Radio Access capabilities, TS 25.306 (Release 5), www.3gpp.org

class supporting bit rates up to 2048 kbps[4]. In this use case, the input is mapped on 16 user transport channels (maximum in this UE class) and the output is mapped on 3 (maximum) physical channels.

3.2.7 Mapping study of the use cases

In [Paper B] we discuss mapping of the service use-cases on different types of common parallel hardware. We studied how the services are mapped on transport channels and what the required parameter configurations are for the two use cases. Function by function, we reason about possibilities and complications for exploiting hardware supported SIMD and MIMD processing when mapping the downlink task graphs for each service.

SIMD mapping One possible mapping option we have studied is to logically group multiple transport channels (which are logically task parallel) to be computed using data parallel hardware. Thus we use one single instruction stream for processing n transport channels, where the user data are partitioned, mapped and computed on an n-words wide SIMD unit.

Efficient SIMD processing of multiple user channels requires that the channels are uniformly configured and that the transport blocks are of equal size. For the AMR use case, this is unfortunately not the case. The transport blocks mapped on their respective transport channels are not of equal length. To compute the bit streams on word length hardware, the algorithm control flow naturally becomes dependent on the length of the transport blocks (recall that many of the functions logically compute bit serially). In the AMR case, the algorithm control flow becomes asymmetric. Another complication is that different CRC polynomials and convolution coding rates are used for each of the three channels.

For the high bit rate data service, the transport blocks are of equal size. Furthermore, the configuration parameters for all transport channels can be configured equally. However, SIMD mapping on a transport channel basis still introduces complications for this use case as well. For example, algorithms based on look-up table techniques need to be serialised (for example, the CRC and the coding functions).

In conclusion, SIMD parallel computation on a task/data parallel user transport channel basis introduces many complications. However, it is an open question whether certain functions for certain services could be beneficially (in terms of performance gain) SIMD computed. Implementation studies on a lower level will be required to answer this question.

[4]The later addition of HSDPA and HSUPA to the 3G standard allows higher bit-rates.

MIMD mapping A MIMD processor enables asynchronous parallel computations of task, data and pipeline parallelism. The trade-off, compared to a SIMD parallel mapping, is a higher cost for parallel synchronisation and communication (moving the data between cores) at certain points in a program. The downlink task graph naturally constitutes a good match with MIMD hardware.

Considering parallel implementation of WCDMA baseband processing on a MIMD structure, there are many interesting issues related to the synchronisation of parallel computations and function configuration. First of all, how do we handle the synchronisation of functions processing different TTIs of data and how do we handle synchronised of distribution configuration parameters?

3.3 Summary and implications

This chapter provided a summary of our analysis of the WCDMA downlink processing in third generation wireless telecommunication systems [Paper B]. We have discussed different types of logical parallelism exposed in the WCDMA downlink symbol rate functions. The computations are primarily of a logical nature rather than of an arithmetical nature, further motivating expressing computations on variably sized bit streams of data. We have discussed possibilities and complications related to application mapping on certain types of common parallel hardware. Further, we find it motivated to investigate expressions of computations on SIMD parallel and bit level computations. For this application domain, it should be possible to express computations on variable bit fields of data as well as data parallelism on the word length of data. Real-time applications, such as the 3G baseband, require task graph mapping strategies with respect to non-functional properties such as computational timing constraints.

Chapter 4

STREAMING MODELS OF COMPUTATION

This chapter introduces streaming models of computation, and we describe the synchronous dataflow model of computation in particular. We motivate the focus on the synchronous dataflow model of computation with respect to the baseband study discussed in Chapter 3 and with respect to the type of manycore architectures we are investigating. Finally, we briefly describe the work presented in [Paper C], where we provide a small modelling framework for elaborating with domain specific SDF languages.

4.1 Introduction

A paradigm shift from centralised processor architectures to manycore architectures will naturally also require a paradigm shift in programming models, languages, compilers and development tools. Programs must be concurrent and malleable for highly parallel and communication exposed hardware interfaces of manycores. Sequential languages conventionally used in the embedded systems industry, such as C, and especially the supporting compiler technology, have been developed for sequential processor architectures exposing a global memory space. Especially considering non-coherent distributed memory manycores, C does not offer a suitable means for expressing concurrency and other knowledge that is important for such a machine target. Important parallel information present in the applications many times must be pruned and it is often not possible to automatically recover such knowledge.

The conventional way of describing concurrency in C programs is to use threads, which are sequential processes logically sharing memory. Relying on threads as a concurrent programming model for code generation to manycore processors is a bad approach for at least two reasons: 1) threads provide an illusion of a shared memory space, which becomes very complex and expensive to resolve when mapped to a distributed memory processor and 2) threads are highly non-deterministic in their nature [Lee, 2006]). A reliable and predictable implementation using threads is relying on programming style and well-defined thread overlay mechanisms, such as semaphores, locks, barriers etc. For automatised mapping to a manycore target, such a source constitutes a state-space that is highly complex to analyse (perhaps many times even impossible), in order to realistically predict its runtime behaviour.

4.1.1 Domain specific programming solutions

We argue that domain specific development methods and tools will be needed to achieve development efficiency for manycore technology. The structure of computations and processing requirements is often quite different for different applications.

As was discussed with one concrete example in Chapter 3, applications in the signal processing domain typically contain a high degree of parallelism and predictable data dependencies, which together form logical function flows in terms of directed graphs. The task of finding an optimal mapping of task graphs on a parallel processor has long been known to be an NP Hard problem [El-Rewini et al., 1995]. There are known solutions for finding near-optimal mappings in linear time [Kwok and Ahmad, 1999]. However, the graph of course has to be known and, in addition, it has to be suitably constructed for the specific optimisation objectives in question. We believe that means for constructing such task graphs, including computational constraints, must be lifted up all the way to the programmer.

Since we are mainly targeting real-time applications, another issue is timing analysis. It is important that the schedules of computations (tasks) are predictable to be able to optimise programs with respect to given real-time constraints. Moreover, timing analysis require well defined models of tasks and their properties when executed not only to ensure determinism, but also in order to minimise run-time overhead in terms of scheduling complexity, context-switching etc.

4.1.2 Stream processing

One of the more promising matches with the signal processing domain is various forms of stream processing. The term *stream* has become attributed to P J Landin for his work in the early 1960s, in which he used the notion of *streams* (in the context of lambda calculus) to model loop and I/O data flows [Landin, 1964]. The *stream* notion was later shown to be useful for research on computational theory in different kinds of computing systems, which are grouped using the general term *stream processing systems* (SPSs).

The common definition of an SPS is an implementation of a network of *processes* that communicates via *channels*. Such a system can be described by means of a graph. The *processes* logically compute in parallel, taking data *streams* as input and producing data *streams* as output. SPSs in general can be specified and analysed using different theories of *stream transformers* (STs) [Stephens, 1997]. An ST is defined as an abstract system that takes a set of n input streams and produces a set of m output streams. Mathematically, an ST can be described as a function

$$\Phi : [T \to A]^n \to [T \to A]^m$$

where A is some data set of interest, $T = \mathbb{N}$ represents discrete time and $n, m \geq 1$.

Dataflow constitutes one very interesting flavour of SPS, which has been a subject to research on computer architecture and modelling of concurrent software for more than 30 years. In dataflow, computations are either data driven, i.e. the processes are computing when input data is available, or demand driven, where processes actively request input data when they wish to fire. Much attention in the past years of research has been paid to different techniques for mapping and computing dataflow graphs on parallel hardware. A good overview of the advances in the research in this area can be found in [Najjar et al., 1999]. The vertices (processes) in a dataflow graph are called *actor*s[1] [Agha, 1986], and these perform the computations. The edges implement FIFO buffers representing the data paths (*channel*s) by which actors communicate. Communication via the channels is the only way an actor can influence the computation of another actor. The firing (computation) of an actor is locally controlled; an actor fires when the required input is available, and many actors may fire simultaneously. Thus, dataflow graphs are naturally concurrent. Actors can be of variable granularity, and hierarchical compositions of actors can be constructed. This also means that an actor can very well be another dataflow graph. This is possible by treating not only actors as compositional components but also the network [Eker et al., 2003].

4.2 Synchronous dataflow

One special case of dataflow is synchronous dataflow (SDF), developed by Lee and Messerschmitt in the middle of the 1980s [Lee and Messerschmitt, 1987]. In SDF, the number of *token*s produced and consumed by an actor when it fires is specified a priori for each input and output channel. A token is an abstraction for the type and size of the units of data communicated through the channels. If an actor can be specified with input and output rates, the actor is said to be synchronous. If all actors in a dataflow graph are synchronous, it is a synchronous dataflow graph.

There are many advantages of SDF compared to conventional dataflow. The a priori specified token consumption and production rates enable compile time scheduling of SDF graphs. This naturally means that more efficient runtime code can be generated since no or very little run-time execution supervision is required [Battacharyya, 1994]. Moreover, the channels in an SDF graph require some form of buffer implementation. Since an SDF graph is periodically executed by a static schedule, the required buffer size is predictable and the buffers are guaranteed never to overflow. Moreover, a consistent SDF schedule can be executed forever without deadlock. An important property is that, as long as the integrity of the dataflow is respected, any parallel or sequential implementation of the dataflow specification yields the same result.

[1]In older literature these are sometimes more abstractly referred to as nodes, blocks or computation units.

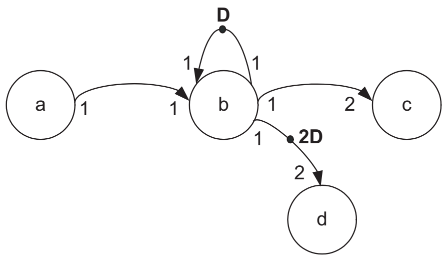


Figure 4.1: Simple SDF graph illustrating rate and delay specifications, as well as the use of feed-back loops.

4.2.1 Description of SDF

We will give a more detailed description of SDF graphs based on the definitions in [Lee and Messerschmitt, 1987]. Figure 4.1 shows a simple example of an SDF graph consisting of four actors and four channels. Each incoming and outgoing channel of an actor is associated with a non-negative integer that specifies the token consumption and production rates. We use r_{in} to denote consumption rate and r_{out} to denote production rate. A channel connecting an output of actor a to an input of actor b is denoted $c(a, b)$.

Actor An actor is an atomic unit of computation that consumes and produces data through a finite set, $P = P_{in} \cup P_{out}$, of ports. The set P_{in} is input ports and the set P_{out} is output ports and $P_{in} \cap P_{out} = \emptyset$. When an actor fires, it consumes r_{in} tokens from all $p \in P_{in}$ and, after firing, it has produced r_{out} tokens on all $p \in P_{out}$.

Channel A channel is a FIFO queue. When an actor consumes a token from an incoming channel, it is removed from the channel. Similarly, when a token is produced on a channel, it is discarded.

SDF graph An SDF graph (SDFG) is a tuple (A, C) with a finite set, A, of actors and a finite set, $C \subseteq P_{out} \times P_{in}$, of channels. Every port is connected to precisely one $c \in C$ and every $c \in C$ is connected to ports, such that the source of every $c \in C$ is an output port of some actor and the destination of every $c \in C$ is an input port of some actor.

Delay Let $c(a_i, a_j)$ be the channel connecting an output port of actor a_i to an input port of actor a_j. If $c(a_i, a_j)$ has k units of delay, it means that the nth token consumed by a_j will be the $(n - k)$th token produced by a_i on $c(a_i, a_j)$.

The SDF graph in Figure 4.1 has two channels specified with a sample delay: channel $c(b, b)$, which is a feed back channel (self loop) that has a delay of one unit (D), and channel $c(b, d)$, which has a delay of two units (2D). The specified delay corresponds to a token offset in the buffer implementing the channel in question.

The repetition vector, q, for the SDF graph in Figure 4.1 is $2a2bcd$ (or $[2211]^T$ in vector notation). The topology of an SDF graph can be described by means of a topology matrix Γ; each row is assigned to a uniquely numbered channel and each column is assigned to a uniquely numbered actor. If an actor i produces n tokens on channel j, the element on entry (j, i) in Γ is n (the production rate). If an actor k produces m tokens on channel l, the element on entry (l, k) in Γ is $-m$ (the consumption rate negated). An actor with a connection to itself (i.e. a self loop) has only one entry in Γ. Its value is the net difference between the actors' production and consumption rates on this channel. This net difference must obviously be zero; otherwise execution of the SDF will deadlock. For all other entries, the elements are zero.

The topology matrix for the SDF graph in Figure 4.1 can be constructed as follows. Let actors a, b, c and d be numbered $1, 2, 3, 4$ and let channels $c(a, b)$, $c(b, b)$, $c(b, c)$ and $c(b, d)$ be numbered $1, 2, 3, 4$, respectively. Then, the topology matrix for the SDF graph in Figure 4.1 is

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & -2 & 0 \\ 0 & 1 & 0 & -2 \end{bmatrix} \tag{4.1}$$

The repetition vector, q, is calculated by solving a set of linear *balance equations*. The balance equations can be written in matrix form as $\Gamma \times q = O$, where O is a vector of zeros. Not all SDF graphs are computable (if assuming an SDF graph to be periodic and non-terminating). For example, not appropriately specified production and consumption rates can lead to an inconsistent specification of an SDF graph. The consistency property (see below) of SDF guarantees that the graph can be indefinitely repeated without deadlocks.

Consistent repetition vector A repetition vector q is said to be non-trivial, if and only if $q(a) > 0$ for all $a \in A$. An SDFG is consistent if it has a non-trivial repetition vector q. For each SDFG that is consistent, there is a unique smallest non-trivial repetition vector that determines the minimum necessary firings of each actor for periodical execution of the SDFG.

Periodic and infinite admissible schedule A periodic admissible schedule ϕ is a non-empty ordered list of actors such that, if actors are fired in the sequence given by ϕ, the amount of data buffered by channels will remain non-negative and bounded. Each actor must appear at least once in ϕ. If ϕ is infinitely repeatable, ϕ is a periodic and infinite admissible schedule.

Proofs and necessary conditions on matrix Γ for finding a periodic admissible schedule can be found in [Lee and Messerschmitt, 1987].

4.3 StreamIt: a language implementing SDF

One example of a programming language implementing the SDF model of computation is StreamIt [Thies et al., 2002]. The StreamIt language has been designed with the objective of providing a portable programming model of streaming data applications, such as signal processing, for communication exposed architectures. Unlike many other languages intended for similar processor targets, StreamIt provides a single machine independent concurrent language that abstracts away both the processors and the communication [Duller et al., 2003][Das et al., 2004][PACT, 2005][Eichenberger et al., 2005].

In StreamIt, an actor roughly corresponds to a filter. Hierarchical StreamIt programs are created using pipelines, which are container constructions performing no computations themselves. Parallel streams are restrictively expressed using split and join constructions, offering a pre-defined and limited set of basic join and split policies. Furthermore, StreamIt filters are limited to having single stream input and output. Thus, StreamIt enforces a highly regular (restricted) channel connectivity of SDF programs.

4.3.1 Limitations in StreamIt

We studied issues when StreamIt is used as the implementation language for the WCDMA downlink processing. The single stream input and output of actors obviously limits expressibility. This can indeed become problematic for certain concurrent function flows with more complex (multiple) data dependencies. However, for implementation of the function flow in WCDMA downlink, it was shown in Section 3 that most data dependencies are of a single stream producer consumer type. Thus, neither the single stream limitation nor the restricted set of split and join policies becomes an issue from this point of view. However, we find the single stream more limiting, when considering periodical distribution of baseband configuration parameters. Mixing computation data with configuration data introduces undesired dependencies between token offsets in a stream and in the filter code. More precisely, the programmer must carefully consider early in the design which part (offsets) of the stream is control data and which part is computation data. If consumption or production rates of control or data at any time during development are changed, both control and computation code might have to be revised.

The concurrent stream constructions (split and join) and the hierarchical pipeline construction enable adjustment of the granularity of function parallelism when possible. However, there are no syntactical means for explicitly expressing fine-grained data parallelism for single instruction computations (i.e. SIMD) on vectors of data. Ritz et al. developed techniques for automatic vectorization based the SDF rate specifications [Ritz et al., 1993]. However, we believe that offering syntactical means for expressing computations on data parallel types will increase programmers' awareness of such parallelism and thereby enable different alternative implementations of data parallel filter (actor) code.

Further, as concluded in the WCDMA analysis, the dominating form of computations are bit level computations on serial bit streams of data. No syntactical means is provided to be able to express computations on variable sized bit fields.

4.4 A modelling framework for SDF languages

We implemented a modelling framework in Java for prototyping and elaborating with extensions and improvements on data types, operators and stream constructions related to the StreamIt language. The motivation for building such a framework is to provide a means to easily experiment with and demonstrate extensions to such languages. In particular, we were motivated to elaborate with possible improvements concerning the limitations we experienced in StreamIt. In [Paper C] we propose different machine independent data types and operators, as well as extended filter constructions for dealing with periodical filter reconfiguration.

4.4.1 The StreamBits language

In [Paper C] we further present a small prototype language, StreamBits, which we use to demonstrate types and operators for bit level and data parallel computations. Moreover, we propose the introduction of logical configuration streams to express distributed filter reconfiguration. We later learned that Solar-Lezama et al. in 2005 also presented a tool (unfortunately) named StreamBits, which addresses bit stream computations in the StreamIt language [Solar-Lezama et al., 2005]. However, their approach was to use a sketching technique (template based programming) to provide bit structural information to the compiler. Further, they did not address filter reconfiguration and fine-grained data parallelism.

Similar to StreamIt, a StreamBits program is composed using `Filter` and `Pipeline` components. However, StreamBits supports dual input and output streams – one for data and one for configuration data. The data stream constitutes the stream on which a filter performs its computation. The configuration stream is used to periodically distribute reconfiguration parameters to filters throughout the distributed network.

A StreamBits `Filter` has three basic execution modes – `init`, `work` and `configure`. Transitions between these modes are mapped automatically at compilation time, and the programmer only needs to define the functionality within each mode. The `init` mode is executed once, before the first firing of the filter, to initialise variables and parameters, and the `work` mode executes the fire code. The `configure` mode is a feature not supported in StreamIt. It is executed once before each execution of the `work` mode. The `configure` mode has been implemented to support more flexible programming of periodical filter reconfiguration of the baseband algorithms (recall the transport format parameters describe in Section 3). With a `configure` mode and a logically separate configuration stream, configuration programming can be defined and modified without any changes in the filters' fire (`work`) code.

StreamBits supports the common types and arithmetic operators in StreamIt. We have also introduced types to support representation and computation on bit field data and data parallel vectors. We describe these types in [Paper C] and also briefly demonstrate the usage of the types and some of the type specific operators with a small example.

4.5 Related work

There are several other, both recent and earlier, approaches that have been taken for domain-specific programming of embedded systems based on stream-oriented models. A large amount of various stream processing languages has been researched and developed during the past 30 years. See [Stephens, 1997] for a good survey. Well known synchronous languages are SIGNAL [Gautier et al., 1987], LUSTRE [Halbwachs et al., 1991] and SISAL [Gaudiot et al., 1997], which require inputs to be available before firing computations. ESTEREL is a concurrent real-time programming language but which was designed for asynchronous dataflow systems [Berry and Gonthier, 1992].

Brook is a programming language based on C but it has been extended with data-parallel constructs [Buck et al., 2004]. Computations that operate on local data are described using a *kernel* construct. Access to memory must be arranged using special *stream* constructs, where the stream elements can be primitive scalars, vectors or record constructs. The Brook language was designed to be adoptable to a wide range of manycore and so called stream processors such as Imagine, Trips and Raw[Kapasi et al., 2003][Burger et al., 2004][Taylor et al., 2004]. In the work reported, it has, however, mainly been used for experiments on programming of graphics acceleration hardware, such as the NVIDIA GeForce 6800.

Other languages very similar to Brook are Cg [Mark et al., 2003] and the StreamC/KernelC language. The Cg language is a language specialized for graphics hardware, which uses the concept of stream kernels but does not provide syntactical notion for well structured specification kernel networks. The Stream-C/Kernel-C was developed for the Imagine stream processor, which is a SIMD/Vector processor architecture [Kapasi et al., 2003]. The main difference compared to to Brook is that, in StreamC/KernelC, kernels are programmed using a specific sub-set of the language (KernelC) while the complete program – the structure of kernel compositions and data streams – is described using the StreamC sub-set of the language. A more recent language, related to Stream-C/Kernel-C, is the Sequoia language [Fatahalian et al., 2006]. In summary, these languages mentioned have been developed with focus a on capturing fine-grained data parallelism and locality of data in programs.

We find the StreamIt language one of the more interesting, fairly recent languages for manycore code generation. StreamIt is a highly specialized language which, unlike many other stream processing languages, is not designed on the basis of the language C. Benchmarks written in StreamIt, compiled to Raw, have shown promising results in processor utilization [Gordon et al., 2006].

The Spidle description language [Consel et al., 2003] is similar to StreamIt. This language was developed to investigate how to support a higher degree of expressability and engineering efficiency in the development of signal processing applications. However, in this language, multiple parallel streams are supported and the input-output interfaces kernels are defined using interface declarations, which enables a higher degree of reusability of code compared to StreamIt.

Our introduction of multiple modes of actors (work and configuration) in the StreamBits language is similar to cyclo-static dataflow (CSDF) [Bilsen et al., 1995][Parks et al., 1995]. In CSDF, multiple firing rules (modes) can be specified and the number of tokens produced and consumed can vary from one firing to another according to a cyclic pattern. However, CSDF is much more general, and scheduling can become much more complicated compared to SDF.

Chapter 5

MACHINE MODEL, INTERMEDIATE REPRESENTATION AND ABSTRACT INTERPRETATION

This chapter is based on the contents of [Paper D] and [Paper E]. We discuss in detail a set of models for the development of a manycore mapping and code generation tool. Further, we introduce a timed intermediate representation with the purpose of both being used as input to a code generator and for evaluating performance and non-functional behaviour using abstract interpretation. We describe the essence of the models, how the intermediate representation is constructed using these models and how they fit in a tool flow.

5.1 Manycore code mapping tool

In embedded real-time systems, optimising scheduling and resource allocation for parallel processors is a problem that typically demands paying attention to many system requirements. Exploiting maximum available parallelism in a program is not necessarily the optimal solution seen from a system perspective. Embedded real-time systems also infer non-functional requirements, whereof one of the more important categories is time. Parallel computing using multiple processors naturally introduces costs for communication, synchronisation and for buffering of data. For example, increasing application throughput through hardware pipelining typically results in an increased end-to-end computation latency. Considering the amounts and various forms of logical parallelism exposed in the type of applications we address, the problem of finding the best fitted parallel implementation of a task graph - with respect to the available processor resources and the application requirements - quickly becomes a nontrivial implementation task.

To handle a multi objective optimisation in complex embedded systems, we believe that programmers must have a greater influence on the parallel mapping strategy. We believe that automated code generation tools need to include user directed mapping strategies. One possibility to do this is to include means for specifying non-functional constraints in the application model, such as timing and resource allocation constraints.

Our approach is a design flow offering iterative tuning of the code mapping process. Figure 5.1 outlines the modular abstraction of our design flow. The input constitutes a model of the application (SDF) and a model of the manycore machine. The output is parallel code that is statically mapped for a specific manycore. We have focused on methods and techniques for analysis of nonfunctional properties, mainly timing. Such information can be fed back in the context of an auto-tuning loop or, be presented to the user for manual tuning using mapping directives, as illustrated in the figure. In [Paper D] and [Paper E] we present our models, intermediate representation (IR) and a methodology for abstract evaluation of the IR with the purpose of:

- abstracting hardware specific details of a certain class of manycores

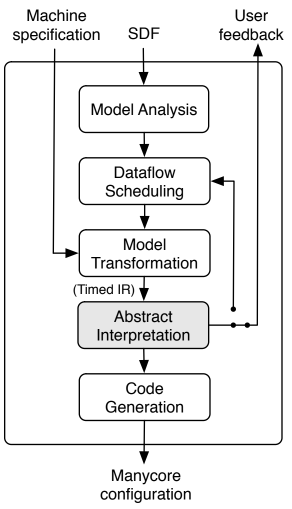


Figure 5.1: A modular illustration of the code mapping framework.

- performing early development analysis of non-functional properties for manual or automatised map tuning of synchronous data flow graphs.

5.1.1 Target processors

An increasing amount and variety of manycore processors are commercially available. However, a big commercial breakthrough of these processors in computing industry has still not appeared. This is to a large extent caused by the lack of efficient development tools, established programming models and processor independent languages. We believe that dataflow models of computation will play an important role in development tools for such processors. Our work mainly focuses on methods and techniques for dataflow code mapping to a certain class of array-structured MIMD (multiple instruction multiple data) type of manycores, as was discussed in Chapter 2. We make the following general categorisation (limitation) of the class of manycores we are targeting for:

Homogeneous and tightly coupled cores A homogeneous array of cores with nearest neighbour communication simplifies construction of tools for code mapping. The communication network is naturally one of the more important resources that highly influence the total cost (in execution time) for parallel execution of a task graph. Minimising communication latencies naturally enables exploitation of finer grained parallelism. Further, decentralizing communication between cores by means of a mesh structure increases predictability in communication timing, which is important in real-time applications.

Individual instruction sequencing Many DSP algorithms contains sequential computations. Naturally, some algorithms have more complex control flows and/or intra algorithm data dependencies [Paper B], which makes them either very difficult or very costly to process spatially (i.e. instruction-level streaming). We believe that manycores offering a balanced mix of spatial and temporal mapping possibilities are better suited for software implementation of arbitrary signal processing graphs.

Distributed memory A coherent shared memory architecture simplifies the programmer's task. However, such memory architectures tend not to scale well when the number of cores is growing [Barua et al., 1999]. Most data dependencies in DSP applications are of a producer consumer type and happen fairly regularly in time. Non-coherent distributed memory architectures (*bank exposed*) allow programmers and mapping tools to take advantage of locality and reduce network contention and overhead when accessing shared memory resources.

Software managed memory transactions Conventional hardware implemented caching introduces unpredictable timing and complicates real-time analysis [Muller et al., 1998]. Since the flow of functions and the data dependencies between functions in DSP applications often follow a fairly predictable pattern [Paper B], manycores with software managed transactions on shared memory banks can be used. This reduces the complexity of real-time analysis on memory transactions and scheduling of network communication.

5.2 Model set

The design and construction of a tool for manycore code mapping require a set of models for abstraction of the hardware resources and the way applications are computed on the target hardware. The models are needed to be able to develop strategic algorithms required for transformation of the high-level program input to a hardware configuration output. Besides serving as a resource abstraction for the hardware, the machine model further needs to provide performance metrics useful for estimation of dynamic computation costs.

5.2.1 Application model

Our choice of application model is the SDF. To be able to analyse the cost resulting from different choices of transformation during the mapping process, and to be able to analyse the execution cost when processing a certain SDF graph on manycore hardware, we need measures of execution times and requirements on communication and memory usage. We assume these data to have been collected (specified by the user and/or during the model analysis, see Figure 5.1). For each actor a we assign an individual tuple

$$< r_p, r_m, R_s, R_s >$$

where

- r_p is the worst case execution time of a, in number of operations
- r_m is the maximum requirement on memory allocation of a, in number of words
- $R_r = [r_{r_1}, r_{r_2}, \ldots, r_{r_n}]$ is a sequence where r_{r_n} is the consumption rate for each input channel $1 : n$ of actor a
- $R_s = [r_{s_1}, r_{s_2}, \ldots, r_{s_m}]$ is a sequence where r_{r_m} is the production rate for each output channel $1 : m$ of actor a.

We have chosen to rely on constant measures of execution times (the worst case) in order to keep our models simple. For actors where the computations are state or data dependent, the execution times will vary between best case and worst case. Naturally, this will affect the prediction accuracy of the execution trace more or less depending on the current state of the program. However, our goal is not to guarantee real-time feasibility. The main goal is to provide a dynamic cost model allowing us to direct map tuning of SDF graphs.

5.2.2 Machine model

The purpose of the machine model is two-fold:

1. to capture processor resources that are important for computing the costs of computation and communication when processing a distributed dataflow graph;
2. to provide a common machine abstraction to enable tool and application portability.

The machine model is a pair of tuples $(M, F(M))$ where M is a set of resource parameters abstracting common machine resources and $F(M)$ is a set of abstract performance functions abstracting common basic operations of a manycore machine. To specify a particular machine, it is required to 1) give values to all parameters of M and 2) define the performance functions

$$F(M) =< t_p, t_s, t_r, t_c, t_{gw}, t_{gr} >$$

where

- t_p is a function evaluating the time to execute a sequence of instructions
- t_s is a function evaluating the core occupancy when sending a data stream
- t_r is a function evaluating the core occupancy when receiving a data stream
- t_c is a function evaluating network propagation delay for a data stream
- t_{gw} is a function evaluating the time for writing a stream to global memory
- t_{gr} is a function evaluating the time for reading a stream from global memory.

The performance functions $F(M)$ are functions of the machine parameters M (see Section 2 in [Paper D]). These functions can be used to determine the execution costs resulting from different choices of graph transformation and resource allocation. Thus, the costs are functions of both resources and their locations, rather than being a static cost estimate, which for example is used in the StreamIt compiler infrastructure for clustering [Gordon et al., 2006]. In Section 5 of [Paper E] we show how these functions are defined for the Raw processor.

5.3 Timed configuration graphs

Portability of both tools and application code naturally requires a manycore intermediate representation. If the goal were just to abstract away processor specific details, such intermediate representation could very well be a fairly simple data structure. However, since we are also interested in analysing non-functional properties for a certain mapping of a graph, there is a need also to represent both time and the execution model. We have developed an IR which we call a *timed configuration graph*. The usage of this graph is two-fold:

1. the IR represents an SDF program after it has been transformed and mapped on the resources offered a specific manycore. We can use this IR as input to a code generator back-end, in order to generate the code for each core, the code for communication and the mapping of memory;
2. the IR can be used to evaluate different run-time properties using abstract interpretation. This evaluation can either be used for feed back in an auto-tuning optimisation loop or be presented to the programmer for step-wise tuning of the mapping constraints.

5.3.1 Construction of timed configuration graphs

We target a design flow using a two step strategy as was introduced by Sarkar [Sarkar, 1989][1]. In the two step strategy, the first step consists of task clustering and the second step consists of scheduling the clusters on the processor. These two steps are performed independently of each other. With scheduling we mean core allocation and scheduling of communication. Comparisons have shown that a two step strategy tends to produce more qualitative mappings than single step strategies [Kianzad and Bhattacharyya, 2006].

A timed configuration graph $G_M^A(V,E)$ describes a synchronous dataflow program A mapped on an abstract machine M (Section 4 in [Paper E]). V is the set of vertices and E is the set of edges. There are two types of vertices: core vertices, v_p, and memory vertices, v_b. A core vertex represents a set of SDF sub-graphs of A mapped on a core. Memory vertices represent buffers mapped in shared memory. The set of edges, E, represents the configuration for the on-chip network.

When constructing a timed configuration graph, $G_M^A(V,E)$, the SDF graph A is first clustered into sub-graphs. Each SDF sub-graph is then assigned to a core in M during the scheduling step. The edges of the SDF graph that end up inside a vertex of type v_p will be implemented using local memory, so they do not appear as top level (visible) edges in G_M^A. The edges of the SDF that reach between pairs of vertices not belonging to the same SDF sub-graphs can be mapped in two different ways:

1. as a network connection between the two cores; such a connection is represented by an edge;
2. as a buffer in global memory. In this case, a memory vertex is introduced.

When G_M^A has been completely constructed, each $v_p, v_b \in V$ and $e \in E$ has been assigned costs (in time) for computation, communication and memory read and writes, respectively. The costs are calculated using the parameters of M and the performance functions $F(M)$ (Section 3 [Paper E]). These costs comprise the static part of the costs, relative to the current time and the current state of the system, when computing the total cost for executing an application.

5.4 Abstract interpretation of timed configuration graphs

In [Paper D] we further present how we can make an abstract interpretation of the timed configuration graph and how an interpreter can be implemented by

[1]Automated clustering and cluster scheduling are not yet implemented in our tool. The transformation of SDF graphs to the IR is currently done by hand.

very simple means on top of a dataflow process network. We have implemented such an interpreter in the Ptolemy modelling environment [Brooks et al., 2008].

The process network (PN) domain[2] in Ptolemy is a super-set of the SDF domain. The main difference in PN, compared to SDF, is that PN actors fire asynchronously and channels have no a priori specified production and consumption rates. If an actor (process) tries to perform a read operation on an empty input channel, it will block until new data are available. The PN domain implemented in Ptolemy is a special case of Kahn process networks [Kahn, 1974]. Unlike in a Kahn process network, PN channels have bounded buffer capacity, which means that a process also temporarily blocks when attempting to perform a write to a buffer that is full [Parks, 1995]. This property makes it possible to easily model link occupancy on the network. A dataflow process network model mimics the behaviour of the types of manycores we are studying very well.

Timed configuration graphs are implemented by means of a heterogeneous and hierarchical dataflow model. The top level is a PN model. Core vertices are implemented using dedicated core actors, and memory vertices are implemented using dedicated memory actors. The SDF application model, provided as input to the tool, is transformed to a distributed SDF model embedded in the PN model. The inter cluster edges in the SDF graph are cut and replaced by top level PN channels representing the inter core communication. However, apart from this temporary cut, the SDF model is still intact. We will now briefly summarise our implementation of timed configuration graphs and how we make abstract interpretation [Paper D].

5.4.1 Interpretation using Process Network

There are different possible approaches for analysing the dynamic behaviour of a dataflow model. One is to let the model calculate resource and timing properties on itself during execution of the model. Another is to generate an abstract representation of the model. We have chosen the latter alternative. The main motivation for this decision is that we are striving for an implementation that does not require modification of the underlying modelling architecture in Ptolemy. Our intermediate representation and abstract interpreter are added on top of the Ptolemy infrastructure, where no modification is required. Although we have not made any experimental comparison, we believe the approach of interpretation on an abstract representation of the mapped SDF graph to be more beneficial in terms of modelling performance.

Program abstraction We abstract the firing for each SDF actor by means of a sequence, S, of *receive*, *compute* and *send* operations:

[2]In Ptolemy, a model of computation is called a domain.

$$S = t_{r_1}, t_{r_2} \ldots t_{r_n}, t_p, t_{s_1}, t_{s_2}, \ldots, t_{s_m}$$

The abstract operations have been bound to static costs computed using the machine model M and the performance functions $F(M)$, presented in Section 5.2.

Time There is no notion of global time in a process network. Each of the top level vertices of G_M^A (cores and memories) is an individual process. Each of the core processes has a local clock, t. The clock, t, is stepped by means of (not equal) time segments. The length of a time segment corresponds to the static cost bound to a certain operation in S and possibly a dynamic cost (blocking time) when issuing send or receive operations addressed to other cores or shared memories.

Cores and memory A core actor implements a state machine. For each actor that is firing, the current *state* is evaluated and then stored in the *history*. The *history* is a chronologically ordered list describing the *state* evolution from time $t = 0$.

Memory actors model competing memory transactions by the first come first served policy. A read or write latency is added to the current time of a read or write operation.

States For each core, we record during which segments of time computations and communication operations were issued. For each core actor, a *state* list maps to a state *type* $\in$ *StateSet*, a start time, t_{start}, and a stop time, t_{stop}. The *state* of a vertex is a tuple

$$state = < type, t_{start}, t_{stop} >$$

The *StateSet* defines the set of possible state types:

$$StateSet = \{receive,\ compute,\ send,\ receiveMem,\ sendMem\}$$

The value of t_{start} is the start of the time segment corresponding to the currently processed operation, and t_{stop} is the end of the time segment. For all *state*s, time t_{stop} corresponds to $t_{start} + \Delta$, where Δ includes the static cost bound to the particular operation. For *send*, *receive*, *receiveMem* and *sendMem* Δ also possibly includes a dynamic cost (blocking time) while issuing the operations.

Synchronisation of clocks The timed configuration graph is synchronised by means of discrete events. Send and receive are blocking operations. A read operation blocks until data are available on the edge, and a write operation blocks until the edge is free for writing. During a time segment, only one message can be sent over an edge. Synchronisation of time between communicating actors requires two way communication. Thus, each edge in the mapped SDF graph is represented by a pair of oppositely directed edges in the implementation of G_M^A.

Network propagation time Channels perform no computation. We use a delay actor to account for propagation times over edges. The delay actor adds the edge weight (corresponding to $t_c \in F(M)$), that has been assigned to each top level edge during the construction of G_M^A

Program interpretation The core interpreter makes state transitions depending on the current operation, the associated static cost of the operation and whether *send* and *receive* operations block or not (the dynamic cost). A state generating function takes timing parameters as input and returns next $state \in StateTypes$.

5.4.2 Modelling limitations of the IR

The use of a process network for implementation of timed configurations graphs has some limitations. In our current implementation, the network is modelled on a relatively high level, at which communication is represented using point-to-point channels. Thus, we are currently not able to model link level concurrency and buffer capacity of the network. This was partly a design decision, because we wanted to keep the network abstraction at a high level for reasons of modelling performance. However, this decision was also made because of the properties of process networks, which complicate modelling of shared resources. Processes in PN execute asynchronously, driven by the availability of data. Processes connected to shared resources therefore require active time sharing to keep the model running. These are problems that can be solved, but at the price of a lower level of implementation of the intermediate representation.

5.5 Evaluating modelling accuracy

In [Paper E] we make an evaluation of the tool with the purpose of:

- evaluating the accuracy of the tool's performance predictions with respect to actual performance;

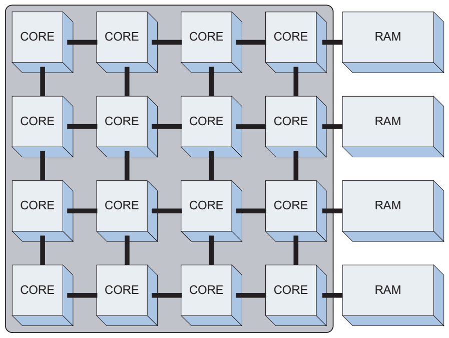


Figure 5.2: Overview of the Raw processor, including the Raw memory configuration used for our experiments. Four non-coherent shared memory banks are located on the east side ports of the chip.

- investigating whether the predictions can be used to rank different mappings of an application with respect to latency and throughput.

In Section 3 in [Paper E] we demonstrate how we configure the machine model in order to model the Raw processor. Raw is a tiled, moderately parallel MIMD architecture with 16 (4×4) programmable tiles (cores) [Taylor et al., 2002]. The cores are tightly connected via two different types of communication networks: two statically and two dynamically routed. For our experiments, we have use a Raw set-up with four off-chip, non-coherent memory banks, see Figure 5.2. In the rest of this chapter we briefly summarise our experiments and the results presented in [Paper E].

5.5.1 Experimental mapping cases

For evaluation of our tool's prediction accuracy, we experimented with different mappings of two applications: matrix multiplication and merge sort. We chose matrix multiplication since it requires fairly large amounts of data to be communicated over the network and since it is good for testing the modelling of communication between the cores and off-chip shared memory. The matrix mappings on Raw are illustrated in Figure 5.3.

In contrast to matrix multiplication, our implementation of the merge sort algorithm has very low requirements on computation and communication. The merge sort mappings on Raw are illustrated in Figure 5.4. The computation and communication load for each node in the tree increases with the depth of the application graph. The different mappings of matrix and merge sort are further described in Section 6 of [Paper E].

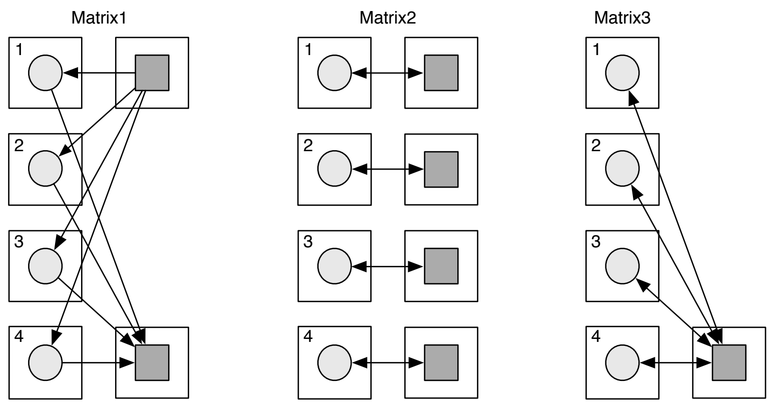


Figure 5.3: Three different mappings of the 32×32 elements matrix multiplication using four cores. Square symbols represent memory and circular symbols represent cores.

5.5.2 Execution strategy

Our implementations are self-timed executions [Lee and Ha, 1989]. In a self-timed execution, scheduling (execution order) and core allocation are determined at compile-time, while synchronisation is handled at run-time. Each core executes actors according to a predetermined firing schedule. A core executes the firing of an actor first when data are available.

Initially, a self-timed execution begins computing in a non-steady state and, later, after a number of iterations, converges to a steady state schedule. All predictions and corresponding measurements are made during steady state execution of the dataflow graphs. Bhattacharyya et al. studied and developed techniques for minimising synchronisation overhead in self-timed executions on multiprocessors [Bhattacharyya et al., 1995].

5.5.3 Comparing communication costs

Communication overhead is a major factor that typically limits performance gain in a parallel implementation. Therefore we have studied the accuracy of the predicted performance on send and receive operations. We use "Raw_{mm}" to denote predicted performance (using our tool) and "Raw" to denote the real performance measured on Raw.

Matrix send and receive times For each of the three test mappings of Matrix reported (see Table 1, Section 6 of [Paper E]), the predicted receive

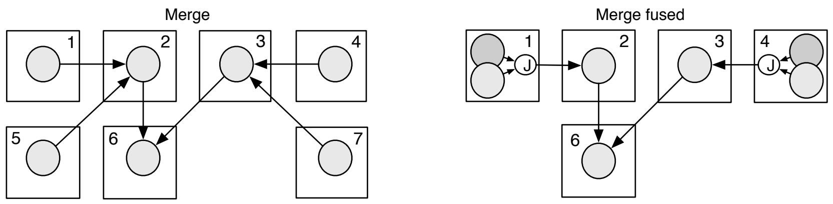


Figure 5.4: The graph to the left is a fully parallel mapping of the merge sort (denoted Merge) and, in the graph to the right, leaf nodes have been pair-wise clustered and mapped to the same core. The smaller node denoted J, in cores 1 and 4, symbolise a join operation performed on the output channels.

times are slightly pessimistic (which is preferable to optimistic). The differences between the predicted and the measured receive times vary between +2.3% and +12.6%.

In Matrix1 and Matrix3 (Figure 5.3), input and output are stored in shared memory banks. While the timed configuration graph do account for asymmetric memory access, it does not model network buffers and link concurrency. This is the main reason for the varying accuracy of the predicted core receive times in Matrix1 and Matrix3.

In Matrix2, each core has exclusive access to its own memory bank, i.e. there are no collisions on the network. Raw cores (which are of the MIPS type) are eight stage pipelined, meaning that execution of instructions experiences variable latency in practice. In contrast to our machine model, the true send and receive occupancies on Raw are therefore not always constant.

The predicted send times for each of the three mappings of Matrix were found to be accurate with the measured send time on Raw (Table 2, Section 6 in [Paper E]). There are no variances in the send times because no (or very few) collisions occur during concurrent send phases, which is further discussed in Section 6 in [Paper E].

Merge sort send and receive times In the corresponding experiment on merge sort, only core-to-core communication is utilised and the communication consists of very small messages (of a length of 1 to 4 words). This mapping was chosen to force unbalanced core communication and computation loads. The goal of this particular experiment was to obtain an indication of how accurately Raw_{mm} models short messaging and unbalanced communication loads. For Merge, the predicted times were found to be exact or very accurate.

For Merge fused, we could see that Raw_{mm} evaluated the receive time 75% higher, compared to the measurements on Raw for cores 2 and 3 (Table 3, Section 6 in [Paper E]). The reason is that the computation loads for the downstream cores 2 and 3, after the clustering, are lower than for the upstream stream cores (1 and 4). On Raw, each core and router entry has a buffer capacity of 5 words, allowing for several short messages to be stored on the network. In contrast, Raw_{mm} models communication pessimistically in the sense that we only allow one message at a time on a network link. Thus, communication between sender and receiver is more tightly synchronised in our model, which can potentially lead to falsely predicted blocking times, as in this particular case.

Similarly, the predicted send times are fairly close to the measured times (the difference is 9.1% or less). The send time comparisons are shown in Table 4, Section 6 in [Paper E].

5.5.4 Latency and throughput measurements

In [Paper E], we used the same mappings to compare predicted and measured end-to-end latency and throughput. The purpose was to evaluate whether the predictions, despite a certain deviation, could be used to correctly rank the different mappings with respect to shortest latency and highest throughput.

Latency based ranking of merge sort mappings Figure 5.5 shows the predicted latencies (in clock cycles) for Merge and Merge fused as a function of the current iteration, compared to the measured latencies. It can be seen in the figure at which iteration each of the mappings reaches its steady state of execution, i.e. when the latency curve levels out. For the Merge mapping, the measured latency is underestimated by a factor of 2. This is explained by the fact that the machine model is currently not able to accurately model the on-chip network's network buffer state. Thus, the difference in iteration count between the first upstream core and the last downstream core is larger in the Raw implementation than in the modelled execution of Raw. On Raw, the end-to-end latency keeps growing until a steady state is reached. A steady state will be reached faster in the modelled execution. To tighten the latency predictions for certain cases of unbalanced computation loads, this experiment shows that we need to account for the network buffer capacity in the machine model.

For Merge fused, we can see that the latency has instead been overestimated, but is closer to the measured latency. The reason is that both the workload and the communication load in Merge fused are better balanced than in Merge, which forces Merge fused to reach a steady state after fewer iterations.

Further, if we rank the predicted latencies of Merge and Merge fused, even if the predictions have varying accuracy, we still see that an optimisation decision based on the predictions would (for this case) correctly identify Merge fused as the better mapping.

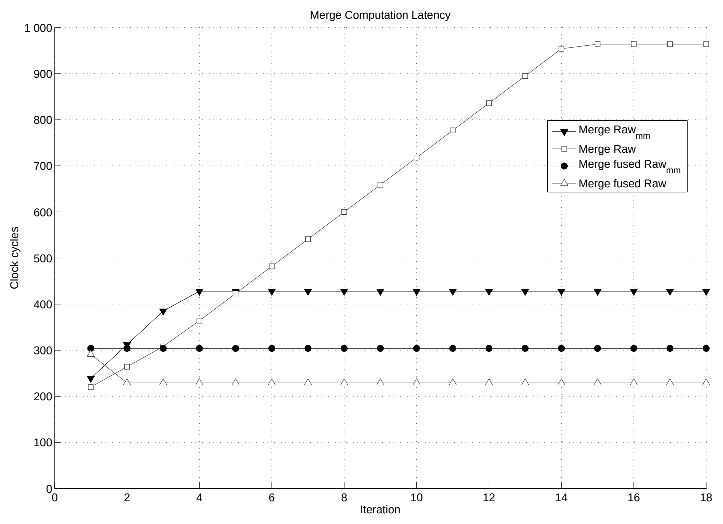


Figure 5.5: Comparison of predicted (Raw$_{\mathrm{mm}}$) and measured (Raw) end-to-end latency for Merge and Merge fused.

Latency based ranking of matrix multiplication mappings Figure 5.6 shows the predicted end-to-end latencies for Matrix1, Matrix2 and Matrix3, compared to the measured latencies in Raw. We can see that the three different mappings of the matrix multiplication converge to steady state at different numbers of iterations. Unlike in the merge sort experiment, the computation tasks distributed on the cores are naturally load balanced. The reason for the different implementations reaching steady state at different points in time is that the cores used in the different mappings are experience different communication delays due to network contention. Contention effects are a large contributing factor to an underestimate of the latencies for Matrix1 and Matrix3. This can be verified by observing that the plot for Matrix2 on Raw$_{\mathrm{mm}}$ and Raw (which is a contention free mapping of the matrix multiplication) is fairly accurate compared to the predictions for Matrix1 and Matrix3. However, if we rank the predicted steady-state latencies for all mappings, we see that an optimisation decision based on latency minimisation would also correctly suggest Matrix2 as the best alternative and that Matrix has shorter latency than Matrix3.

Throughput based ranking of merge sort mappings Table 5.1 shows the predicted and the measured throughput for Merge (within a 4.4% differ-

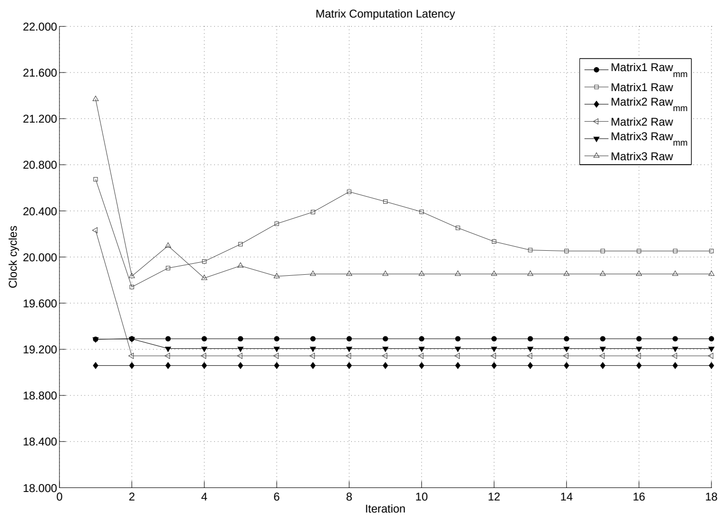


Figure 5.6: Comparison of the modelling accuracy of the computation latency of three different mappings of the parallel matrix multiplication.

ence) and Merge fused (within a 10% difference). The predictions are fairly close to the measurements on Raw for both Merge and Merge fused. We can also see that both the predicted and the measured throughput show that Merge has a higher throughput than Merge fused. When optimising for throughput, our predictions correctly rank Merge as the best.

Throughput based ranking of matrix multiplication mappings Table 5.2 shows the comparisons of throughput for Matrix1, Matrix2 and Matrix3. Note that, unlike in all the other experiments, our model has predicted a slightly optimistic throughput. However, if we rank both the predicted throughput and the measured throughput, we can also in this case see that the predictions will be ranked in the same order as for the measured. Thus, if we use the predictions for throughput optimisation, our tool correctly identifies the better solution for this example as well.

5.6 Discussion of the results

This chapter has presented an evaluation of the prediction accuracy of the tool. The communication times predicted between cores are slightly pessimistic, but we demonstrated that, for the small set of mappings explored in the experi-

Application	Raw_{mm}	Raw	diff
Merge	119	104	+4,4%
Merge fused	132	120	+10%

Table 5.1: Merge steady state periodicity (clock cycles).

Application	Raw_{mm}	Raw	diff
Matrix1	19249	19434	-0,9%
Matrix2	19059	19143	-0,4%
Matrix3	19248	19401	-0,8%

Table 5.2: Matrix steady state periodicity in (clock cycles).

ments, our tool can still correctly rank the different mappings with respect to highest throughput or shortest latency. However, the comparisons also reveal that the predictions of end-to-end latency for unbalanced computation loads can be quite inaccurate. This was demonstrated to be mainly dependent on the high abstraction level used in the modelling of on-chip communication; it does not capture buffer state or contention effects in the network. However, this is mainly a limitation set by the current implementation of the timed configuration graphs.

We have at this stage only experimented with a small set of different graph layouts (for merge sort and matrix multiplication), which are normally parts of larger application graphs. However, we believe that the slight inaccuracy (pessimism) of the predicted send of receive times is of lesser importance than capturing buffer capacity and network contention, in terms of providing more reliable predictions for ranking task graph mappings. To capture such effects, it is necessary to include a lower level modelling abstraction of on-chip networks. Further, experiments with a more extensive set of both larger and smaller task graphs are needed to be able to quantify the reliability of our tool.

5.7 Related work

One of the more extensively explored problems in parallel processing is the problem of mapping task graphs to multiprocessors. Heuristic solutions are required since this is known to be an NP complete problem [El-Rewini et al., 1995][Kwok and Ahmad, 1999]. In the late 1980s, Sarkar introduced the two step mapping method, where clustering is performed in a first step independently of the second step of cluster scheduling (resource allocation), which can both be applied at compile time [Sarkar, 1989]. A number of leading algorithms for both single step and two step clustering, with common objectives of transforming and mapping task graphs for multiprocessor systems, are reviewed in [Kianzad and Bhattacharyya, 2006]. A rich set of solutions is available, and thus we have not put any effort into this particular problem in our work.

The dynamic level scheduling algorithm proposed by Sih and Lee is single-step heuristic, taking inter-processor communication overhead into account during clustering. In a way similar to our work, Sih developed this scheduling algorithm to be used for producing feedback to the programmer for iterative refining of the task graph and the architecture [Sih, 1991]. However, it has been demonstrated through experimental comparisons by Kianzad and Bhattacharyya that two step methods tend to produce more qualitative schedules than single step methods [Kianzad and Bhattacharyya, 2006].

Unfortunately, expanding an SDF graph to an acyclic precedence graph – which is the assumed representation for many older scheduling and mapping algorithms – can lead to an explosion of nodes. This problem can partly be reduced by using different clustering techniques before expanding the SDF graph to an acyclic precedence graph (APG) [Pino and Lee, 1995]. We are interested in techniques for analysis and mapping of SDF graphs without conversion to an APG, i.e. using direct SDF representation.

The StreamIt language implements a restricted set of SDF. The StreamIt compiler implements a two step mapping (dataflow scheduling and clustering, followed by core allocation) using direct representation of SDF graphs [Gordon et al., 2006][Taylor et al., 2002]. However, the StreamIt compiler uses a rather static cost model to determine clustering and core allocation costs. Further, neither the language nor the compiler provides any means to express non-functional constraints or other application specific optimisation criteria to tune the parallel mapping and code generation. The dataflow scheduling, clustering and mapping algorithms implemented by the StreamIt compiler generate parallel mappings that make a fair trade-off between throughput and end-to-end latency. In order to improve a mapping, in terms of reducing latency or improving throughput, programs have to be restructured.

Throughput is one important non-functional requirement in the real-time applications we are addressing. Ghamarian et al. provide methods for throughput using state space analysis (simulating execution of the SDF) on direct representation of multi-rate SDF graphs [Ghamarian et al., 2006]. Further, Stuijk et al. developed a multiprocessor resource allocation strategy for throughput constrained SDF graphs [Stuijk et al., 2007]. We are addressing techniques that allow application specific combinations of timing constraints and for how these can be used to direct the mapping process.

Bambha et al. reviewed different intermediate representations for different objectives on optimisation and synthesis for self-timed execution of SDF programs to multiprocessor DSP systems [Bambha et al., 2002]. They assume homogenous representation of SDF graphs, which exposes a higher degree of task parallelism based on the rate signatures. Our work is similar, but we are mainly interested in intermediate representations on multi-rate SDF and in minimising transformation between different representations during the mapping process.

Several researchers have addressed the development of frameworks for auto tuned optimisation of signal processing kernels. The FFTW system focuses on adapting the implementation of the discrete Fourier transform for different target platforms using profile feed- back [Frigo and Johnson, 2005]. The SPIRAL project focuses on automated feed- back driven optimisation of a broader scope of signal processing kernels using machine learning techniques, starting from formal mathematic specifications using a symbolic language called SPL [Püschel et al., 2005]. Similar to FFTW, ATLAS is an auto tuning system for linear algebra kernels. Commonly for these systems is that they only deal with algorithm libraries provided by the library designer [Demmel et al., 2005].

Yotov et al. have further compared an empirical tuning engine with a model-driven tuning engine in the ATLAS system [Yotov et al., 2003]. The machine parameters used in their experiments are cache and register file sizes and the number of floating-point registers and floating-point multiplication units. They found that model-driven optimisers can generate code with performance comparable to code generated using empirical optimisers.

PetaBricks is a parallel language and an auto tuning compiler [Ansel et al., 2009]. In PetaBricks, it is the programmer that specifies different implementations of algorithms and how they can be combined. The decision on which algorithm to use is determined by the PetaBricks compiler and run-time system.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The computing industry is facing a grand challenge with the entrance of manycore technology. The work presented in this thesis has addressed a few of the many issues related to development of industrial embedded high-performance applications using manycore technology.

The first goal of the work was to explore performance trade-offs related to the choice of manycore processing paradigm. In particular, we compared area performance trade-offs related to spatial processing on fine-grained manycores versus temporal and spatial processing on coarse-grained manycores. It was estimated by calculations that a finer-grained manycore with no dedicated control logic has a clear area performance advantage over more coarse-grained manycores with program sequencing in each core, even if much of the data processing resources must be used for program control. On the basis of typical processing requirements of high-performance DSP applications, we outlined a two level reconfigurable computer architecture based on finer-grained manycore technology at the lowest level.

The second goal was to investigate suitable computation models and techniques for programming manycores and for the development of tools for computer assisted mapping of signal processing task graphs. We performed an analytical study of parts of the baseband processing required in WCDMA radio base stations. This study concluded that the WCDMA function flows indeed provide a good match for implementation using stream processing models of computation. We studied the WCDMA specifications with respect to different types of large-grain parallelism, the real-time aspects of such systems, and system reconfiguration and computation characteristics on the algorithm level. The synchronous dataflow language StreamIt was used to evaluate implementation issues of the WCDMA processing in a stream processing language. On the basis of this evaluation, we proposed extensions to the StreamIt language to be able to deal with instruction level data parallelism and express computations on bit-serial streams and periodical reconfiguration of the distributed filters (actors). We provided a modelling framework for practical elaboration with languages based on the synchronous dataflow model of computation.

Finally, the third goal was to develop techniques for analysing non-functional properties and predicting dynamic execution costs on manycore processors. The portability of tools and application software requires a set of models to abstract application software and target processors. We presented a machine model that allows tool builders and application developers to specify the resources and the computation performance for a certain class of array structured manycores. Compiling source programs to an abstract manycore target further requires a suitable intermediate representation. To be able to predict the dynamic execution behaviour of a certain task graph on a certain manycore target, there is a need also to represent both the execution model and time. We have proposed and developed one way to implement our intermediate representation using a discrete event model on top of a dataflow process network. Furthermore, we present techniques for predicting run-time performance by means of abstract interpretation. It was further demonstrated, even if only for a small set of task graphs, that the performance predictions generated were useful in ranking mappings by optimisation with respect to application end-to-end latency and throughput.

6.2 Future work

There are several problems that we have not yet addressed and solutions we proposed that can be further refined:

Prediction accuracy It is motivated to further investigate how to include a lower, more detailed, level of network abstraction in the intermediate representation. We demonstrated that it is of particular importance for prediction accuracy to include modelling of network buffers and network contention effects.

Automated mapping and tuning We have not dealt with automation of the task graph transformation and mapping process. An extensive number of solutions has been reported on task graph transformation and task graph mapping on multiprocessors that can be used as a basis. Further work should investigate how predicted feed back can be beneficially used in combination with user specified mapping constraints to tune the mapping process.

Heterogeneous application models The synchronous dataflow model of computation has many advantages that simplify model analysis, model transformation and the generation of compact code. However, one trade-off for these advantages is a limited expressibility. We see a need for investigating the usage of heterogenous models of computation for application development and how to efficiently implement them on manycore hardware.

Dynamic resource mapping In our work, we have only dealt with the case of static mapping of application graphs. Such a mapping can often only be used to describe smaller parts of a system or a system in a limited state space. In industrial applications of the kind that we address, systems must be able to handle fast and adaptive reconfiguration of task graphs. There is a need to further investigate combined static and dynamic task graph mapping and resource allocation methods.

Bibliography

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA.

Åhlander, A. (2007). *Efficient Parallel Architectures for Future Radar Signal Processing.* PhD thesis, Chalmers University of Technology.

Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. (2009). PetaBricks: A language and compiler for algorithmic choice. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Dublin, Ireland.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Bambha, N., Kianzad, V., Khandelia, M., and Battacharyya, S. (2002). Intermediate representations for design automation of multiprocessor DSP systems. In *Design Automation for Embedded Systems*, volume 7, pages 307–323. Kluwer Academic Publishers.

Barua, R., Lee, W., Amarasinghe, S., and Agarwal, A. (1999). Maps: a compiler-managed memory system for Raw machines. *ACM SIGARCH Computer Architecture News*, 27(2):4–15.

Battacharyya, S. S. (1994). *Compiling Dataflow Programs for Digital Signal Processing.* PhD thesis, University of California at Berkeley.

Baumgarte, V., May, F., Vorbach, M., and Weinhardt, M. (2001). PACT XPP - a self-reconfigurable data processing architecture. In *Proc of Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, USA. CSREA Press.

Bengtsson, J. and Lundin, B. (2003). Reconfigurable architectures for high-performance computing. Master's thesis, Technical report IDE0306, Halmstad University, Halmstad, Sweden.

Berry, G. and Gonthier, G. (1992). The ESTEREL synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(19).

Bhattacharyya, S. S., Sriram, S., and Lee, E. A. (1995). Minimizing synchronization overhead in statically scheduled multiprocessor systems. In *Proc. of the IEEE Int'l Conf. on Application Specific Array Processors*, page 298, Washington, DC, USA. IEEE Computer Society.

Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. (1995). Cyclo-static data flow. In *Proc. of the IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, volume 5, pages 3255–3258 vol.5.

Brooks, C., Lee, E. A., Liu, X., Neuendorffer, S., Zhao, Y., and Zheng, H. (2008). Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Dept., University of California, Berkeley.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics (SIGGRAPH 2004)*, 23(3):777–786.

Burger, D., Keckler, S. W., McKinley, K. S., Dahlin, M., John, L. K., Lin, C., Moore, C. R., Burril, J., McDonald, R. G., and Yoder, W. (2004). Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55.

Carmean, S. H. G. F. B. D. M. and Hall, J. C. (2001). Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 5(1).

Consel, C., Hamdi, H., Reveillere, L., Singaravelu, L., Yu, H., and Pu, C. (2003). Spidle: A DSL approach to specifying streaming applications. In *Proc. of the 2nd Int'l Conf. on Generative Programming and Component Engineering*, pages 1–17. Springer-Verlag New York, Inc.

Dally, W. J. (1990). Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. on Computers*, 39(6):775–785.

Das, A., Mattson, P., Kapasi, U., Owens, J., , and Rixner, S. (2004). Imagine programming system user's guide. cva.stanford.edu/projects/imagine.

Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R., and Yelick, K. (2005). Self-adapting linear algebra algorithms and software. *Proc. of the IEEE*, 93(2):293–312.

Duller, A., Panesar, G., and Towner, D. (2003). Parallel processing - the picoChip way! In *Proc. of Communicating Process Architectures*, pages 125–138, Enschede, Netherlands.

Eichenberger, A. E., O'Brien, K., O'Brien, K., Wu, P., Chen, T., Oden, P. H., Prener, D. A., Shepherd, J. C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., and Gschwind, M. (2005). Optimizing compiler for the CELL processor. In *Proc. of the 14th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA. IEEE Computer Society.

Eker, J., Janneck, J., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S. R., and Xiong, Y. (2003). Taming heterogeneity - the Ptolemy approach. *Proc. of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):127–144.

El-Rewini, H., Ali, H., and Lewis, T. (1995). Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37.

Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., Park, J. Y., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. (2006). Sequoia: Programming the memory hierarchy. In *Proc. of the 2006 ACM/IEEE Conf. on Supercomputing*.

Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Gaudiot, J.-L., Bohm, W., Najjar, W., DeBoni, T., Feo, J., and Miller, P. (1997). The Sisal model of functional programming and its implementation. In *Proc. of the 2nd AIZU Int'l Symp. on Parallel Algorithms / Architecture Synthesis*, pages 112–123.

Gautier, T., Guernic, P. L., and Besnard, L. (1987). Signal: A declarative language for synchronous programming of real-time systems. Technical Report N761, INRIA, Campus de Beaulieu, 35042 Rennes Cdex, France.

Ghamarian, A., Geilen, M., Stuijk, S., Basten, T., Theelen, B., Mousavi, M., Moonen, A., and Bekooij, M. (2006). Throughput analysis of synchronous data flow graphs. *Proc. of Intl Conf. on Application of Concurrency to System Design*, pages 25–36.

Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of Twelfth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous dataflow programming language LUSTRE. In *Proc. of IEEE*, volume 79, pages 1305–1320.

Holma, H. and Toskala, A. (2004). *WCDMA for UMTS: Radio Access for Third Generation Mobile Communications*. J. Wiley & Sons Ltd., 3'rd edition.

Johnsson, D., Åhlander, A., and Svensson, B. (2005). Analyszing the advantages of run-time reconfiguration in radar signal processing. In *Proc. of the 17th IASTED Int'l Conf. on Parallel and Distributed Computing Systems*, Phoenix, Az, USA.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *IFIP Congress 74*, pages 471–475, Stockholm, Sweden. North-Holland Publishing Company.

Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P., and Owens, J. D. (2003). Programmable stream processors. *IEEE Computer*, 36(8):54–62.

Kianzad, V. and Bhattacharyya, S. (2006). Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 17(7):667–680.

Kwok, Y.-K. and Ahmad, I. (1999). FASTEST: A practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 10(2):147–159.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320.

Lee, E. and Ha, S. (1989). Scheduling strategies for multiprocessor real-time DSP. In *Proc of the IEEE Global Telecommunications Conference, 1989, and Exhibition. Communications Technology for the 1990s and Beyond*, pages 1279–1283 vol.2.

Lee, E. A. (2006). The problem with threads. *IEEE Computer*, 39(5):33–42.

Lee, E. A. and Messerschmitt, D. G. (1987). Static Scheduling of Synchronous Data Flow Programs for Signal Processing. *IEEE Trans. on Computers*, 36(1):24–35.

Mangione-Smith, W. H., Hutchings, B., Andrews, D., DeHon, A., Ebeling, C., Hartenstein, R., Mencer, O., Morris, J., Palem, K., Prassana, V. K., and Spaanenburg, H. A. E. (1997). Seeking solutions in configurable computing. *IEEE Computer*, 30(12):38–43.

Mark, W. R., Steven, R., Glanville, R. S., Akeley, K., and Kilgard, M. J. (2003). Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 22:896–907.

Moritz, C. A., Yeung, D., and Agarwal, A. (2001). SimpleFit: A framework for analyzing design tradeoffs in Raw architectures. *IEEE Trans. on Parallel and Distributed Systems*, 12(7):730 – 742.

Muller, H., May, D., Irwin, J., and Page, D. (1998). Novel caches for predictable computing. Technical report, Department of Computer Science, Bristol, UK, UK.

Najjar, W. A., Lee, E. A., and Gao, G. R. (1999). Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929.

PACT (2005). Programming XPP-IIb Systems. www.pactcorp.com.

Parks, T. M. (1995). *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA.

Parks, T. M., Pino, J. L., and Lee, E. A. (1995). A comparison of synchronous and cycle-static dataflow. In *Proc. of the 29th Asilomar Conf. on Signals, Systems and Computers*, pages 204 – 210, Washington, DC, USA. IEEE Computer Society.

Pino, J. L. and Lee, E. A. (1995). Hierarchical static scheduling of dataflow graphs onto multiple processors. In *Proc. of IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, pages 2643–2646.

Proakis, J. G. and Manolakis, D. G. (1996). *Digital Signal Processing: Principles, Algorithms and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. (2005). SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, 93(2):232– 275.

Ritz, S., Pankert, M., Zivojinovic, V., and Meyr, H. (1993). Optimum vectorization of scalable synchronous dataflow graphs. In *Proc. of Int'l Conf. on Application-Specific Array Processors*, pages 285–296.

Sarkar, V. (1989). *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA.

Sih, G. C. (1991). *Multiprocessor Scheduling to Account for Interprocessor Communication*. PhD thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA.

Solar-Lezama, A., Rabbah, R., Bodík, R. R., and Ebcioğlu, K. (2005). Programming by sketching for bit-streaming programs. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation*, pages 281–294, New York, NY, USA. ACM Press.

Stephens, R. (1997). A survey of stream processing. *Acta Informatica*, 34(7):491–541.

Stuijk, S., Basten, T., Geilen, M., and Corporaal, H. (2007). Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *44th ACM/IEEE Design Automation Conference*, pages 777–782.

Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A. (2002). The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35.

Taylor, M. B., Lee, W., Miller, J., Wentzlaff, D., Bratt, I., Greenwald, B., Hoffmann, H., Johnson, P., Kim, J., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., , and Agarwal., A. (2004). Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. of Int'l. Symp. on Computer Architecture*, pages 2–13, Munchen, Germany.

Thies, W., Karczmarek, M., and Amarasinghe, S. (2002). StreamIt: A language for streaming applications. In *Proc. of the 2002 Int'l Conf. on Compiler Construction*, Grenoble, France.

Wentzlaff, D. and Agarwal, A. (2004). A quantitative comparison of reconfigurable, tiled, and conventional architectures on bit-level computation. In *12th IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 289–290, Napa, CA.

Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., and Wu, P. (2003). A comparison of empirical and model-driven optimization. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming language design and implementation*, pages 63–76, New York, NY, USA. ACM.

Zhang, Z., Heiser, F., Lerzer, J., and Leuschner, H. (2003). Advanced baseband technology in third-generation radio base stations. *Ericsson Review*, (1):32–41.

List of Figures

List of Tables

Paper A

Two-level Reconfigurable Architecture for High-performance signal processing

---

Johnsson, D., Bengtsson, J. and Svensson, B. (2004). Two-level reconfigurable architecture for high-performance signal processing. In *Proc. of Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, pages 177-183, Las Vegas, USA.

Two-level Reconfigurable Architecture for High-Performance Signal Processing

Dennis Johnsson, Jerker Bengtsson, and Bertil Svensson
Centre for Research on Embedded Systems, Halmstad University,Halmstad, Sweden
Dennis.Johnsson@ide.hh.se, Jerker.Bengtsson@ide.hh.se, and Bertil.Svensson@ide.hh.se

Abstract

High speed signal processing is often performed as a pipeline of functions on streams or blocks of data. In order to obtain both flexibility and performance, parallel, reconfigurable array structures are suitable for such processing. The array topology can be used both on the micro and macro-levels, i.e. both when mapping a function on a fine-grained array structure and when mapping a set of functions on different nodes in a coarse-grained array. We outline an architecture on the macro-level as well as explore the use of an existing, commercial, word level reconfigurable architecture on the micro-level. We implement an FFT algorithm in order to determine how much of the available resources are needed for controlling the computations. Having no program memory and instruction sequencing available, a large fraction, 70%, of the used resources is used for controlling the computations, but this is still more efficient than having statically dedicated resources for control. Data can stream through the array at maximum I/O rate, while computing FFTs. The paper also shows how pipelining of the FFT algorithm over a two-level reconfigurable array of arrays can be done in various ways, depending on the application demands.

Keywords: *Signal processing, reconfigurable array, dataflow*

1. Introduction

Many advanced signal processing applications put hard requirements on the embedded computer architecture. These requirements manifest themselves as high compute requirements, limited power consumption, small space, low cost, etc. This means that there has to be efficient signal processing architectures. The use of dedicated hardware is common in these applications, but more flexible solutions are sought for. Rather than going for the acceleration of one specific calculation one should identify domains of applications and design an architecture that supports this entire domain. This enables building domain specific architectures rather than application specific architectures. Further, by the use of reconfigurable hardware it is possible to accelerate larger parts of the algorithms than with a fixed computer architecture.

In this paper we outline an architecture targeted for multi-dimensional signal processing. It is characterized by its two abstraction levels: the macro-level and the micro-level. In order to make efficient processing possible in a dynamic way, reconfiguration of the hardware is allowed on both levels. On the macro-level, the architecture can take advantage of the available function and high-level pipeline parallelism in the applications. On the micro-level, the massive, fine-grained data parallelism allows efficient parallel and pipelined computations of algorithms such as FFTs, FIR filters, and matrix multiplications.

The micro-level is characterized by regular, highly parallel dataflow and repeated calculations. Thus, in structures for this level, it should be possible to allocate most of the hardware resources to the parallel data path and only minor parts to the control of the computations. Since this division of tasks is application dependent, it is interesting to study the use of architectures where the same hardware resources can be used both for data processing and for the control flow. Therefore, in this paper, we study the implementation of an FFT on a reconfigurable array of arithmetic logic units (ALUs) that can be used for either of these tasks. The goal is, of course, to use as much as possible of the available resources for the data path.

The rest of the paper is organized as follows: First we briefly describe the characteristics and demands of multi-dimensional signal processing and argue that a two-level architecture is an efficient way to achieve the required flexibility and computational power. We then discuss various micro-level structures and in particular the more coarse-grained, word-level reconfigurable array architectures. We describe an implementation of an FFT on one such, commercially available array — the XPP. We find that, in our implementation, a larger portion of the array is actually used for the flow control than for the actual butterfly operations in the FFT. Still, it can be argued that the array is actually more area-efficient than a more conventional solution. The work of the control part roughly corresponds to the task of the program code and control logic in a microprocessor or digital signal processor. Whether the ratio between the two parts could be changed by alterations to the architecture or by more efficient mapping is still an open question.

We conclude the paper by illustrating how signal processing applications can be mapped onto the architecture on the macro-level, including how an FFT computation can be mapped onto a pipeline of processor arrays in order to exactly match the requirements of the application.

2. Structures for high-performance embedded signal processing

In high-performance embedded signal processing it is crucial that the architecture exploits the available parallelism in an efficient way. Parallelism exists on different levels — from the macro-level function parallelism down to the micro-level, fine-grained data parallelism. The applications considered here use multi-dimensional signal processing on streaming data. A typical example is phased array radar signal processing. In these applications, functions such as transformations and linear operations are applied on the input data volume. The functions are performed in succession, implying that a natural approach for parallelization of the application is to arrange a pipeline of function blocks. Since the applications considered here are multi-dimensional it also means that, for each dimension, the same sub-function is applied in parallel to all elements in that dimension. Of course, this makes it possible to execute the same function on different dimensions in parallel. The type of functions could be matrix by vector multiplications, FIR filters, FFTs, etc. These functions themselves have inherent data parallelism. Figure 1 shows an example of a data volume in a radar application. The transformations are made on vectors along different dimensions of the volume and after each other. The size and shape of the volume can vary depending on the task performed by the radar. Moreover, radar tasks — and hence data volume shapes — may change during operation.

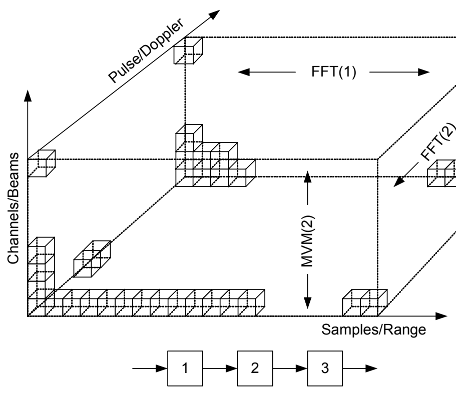


Figure 1. A multi-dimensional signal processing example. Functions are performed in different directions and in succession.

The applications also have real-time demands, both in terms of high throughput and in terms of latency. As a consequence of the changes in radar tasks these requirements also change. Different tasks have different requirements on throughput and latency.

An efficient architecture for embedded high speed signal processing is characterized by quantitative measures such as high compute density, both regarding space (operations/cm^3) and power (operations/W). Other, more fuzzy, metrics could be flexibility or how much general-purpose the architecture is. The architecture should offer more flexibility than those built with ASICs but it does not require the generality of a system consisting of one or more general purpose processors. The question is then what the required flexibility is for a certain domain of applications. The architecture needs to manage parallel data paths. The control of these data paths could be made in different ways. The most flexible — but least efficient — way is to have one control program for each data path, as in a microprocessor. Using one program for a group of data paths, as in a SIMD processor, or a more static mapping of the control, as found in configurable architectures, offers significant efficiency gains. These tradeoffs can be investigated by actual implementation studies on different platforms. Another vital aspect, not touched upon here, is how to develop applications for an architecture, i.e., what tools there are to support the application developer.

Demanding application domains, such as radar signal processing, need highly parallel architectures. Advanced radar systems may require computing 10^{12} arithmetic operations per second, with constraints on both space and power usage [1]. These architectures should be scaleable and support the computation of applications with highly different characteristics, e.g. different sizes and shapes of the data volume, and different demands on throughput and latency. In order to efficiently map the applications we envision mapping of functions onto a parallel architecture where, for each function, a node or group of nodes are responsible for executing the function. The nodes should then implement the calculations of the functions in an efficient way. The architecture should provide for static mapping of the functions and the dataflow, but also provide means to quickly, during runtime, reconfigure the mapping and communication pattern to adapt to a different scenario, e.g., changing the data set shape and modifying the signal processing functions.

Embedded high-performance signal processing applications often require complete systems consisting of multiple interconnected chips and even boards. It is therefore important that the architecture allows scaling. In order to ease the integration the same structure should be applied repeatedly. We believe a suitable architecture is a two-level configurable architecture. The first level, the macro-level, is the target for mapping of functions. The dataflow between nodes is configured on this higher level, and functions are assigned to one or a group of nodes depending on the requirements. System configuration on the macro-level should support run-time re-allocation of nodes to functions.

An efficient architecture could be built using a mesh of nodes. A mesh architecture is scalable to larger structures and is also favorable from an implementation point of view. Functions could be mapped to the mesh in different ways depending on the requirements of the application. The nodes in the mesh should then, on the micro-level, execute the functions as efficiently as possible, meaning that the low-level data parallelism within the functions should be exploited in the most efficient way. Figure 2 illustrates this two-level approach.

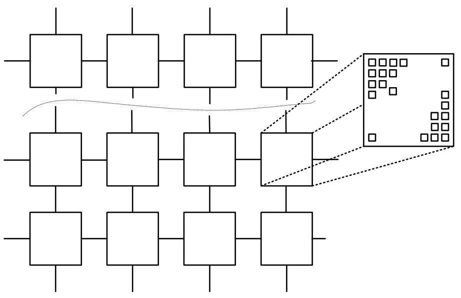


Figure 2 A two-level reconfigurable architecture.

3. Micro-level structures

The primary issue regarding the design of the micro-level architecture is to find a proper structure that offers a sufficient degree of flexibility to exploit the data parallelism found within the different functions. Microprocessors and DSPs offer full flexibility but do not have the performance obtainable with ASICs or configurable architectures [2]. ASICs offer the best performance regarding compute density and power consumption and are often used for high-end signal processing when DSPs fail to meet the requirements. However, ASICs are usually highly adapted for a specific problem and therefore not very flexible. Another technology that has evolved and, in recent years, matured to become a competitive alternative for application specific solutions is the field-programmable gate array (FPGA). FPGAs have been shown to be performance effective; however, fine grained FPGA circuits tend to require large portions of logic to implement common operations like multiplication on word length data. Major FPGA manufacturers such as Xilinx and Altera try to reduce this drawback by embedding dedicated multiplication units within the fine grained logic. Conclusively, our desire would be to find a programmable micro-level architecture that correlates well with algorithm requirements within the domain of signal processing, and which is flexible enough to exploit data parallelism within functions but also allows swift reconfigurations between functions within an application.

When choosing architecture for the micro-level there are a number of aspects that have to be addressed. What are the functional requirements on the nodes? Is it possible to take advantage of the limited range of algorithms that need to be implemented? Is it possible to do without a general microprocessor and, if so, how will the control of the algorithms be implemented? Coarse grained reconfigurable architectures offer the possibility of implementing control flow through run-time reconfiguration, or use static configurations of the control flow when possible. If a reconfigurable architecture is used there are questions of how efficient the resource usage is, how the reconfiguration is done and how fast it is.

Further, the I/O bandwidth is often an issue with all signal processing architectures. The micro-level should have a balanced architecture with I/O bandwidth matching the compute performance. The memory requirement also has to be considered. What is the proper balance between on-chip memory and computing resources?

Word-level reconfigurable architectures represent one class of candidates for implementing high-performance multidimensional signal processing. The functional, control, I/O and memory issues of these architectures need to be studied in relation to the application domain.

3.1. Coarse-grained reconfigurable architectures

Besides programmable solutions like FPGA, which are considered fine-grained architectures, there are many other configurable alternatives today. Many companies and academic research groups have developed coarser grained, programmable processor concepts. The processing elements in these span a quite wide spectrum from simple ALUs [3][4] to pipelined full scale processors [5][6][7]. Two instances of these different coarse-grained paradigms are the XPP processor from PACT XPP Technologies and the RAW processor developed at MIT.

The RAW processor is in principle a chip multiprocessor, constituting an array of 16 full scale MIPS-like RISC processors called tiles. Characteristic for RAW is the high I/O bandwidth distributed around the chip edges and the dynamic and static networks that tightly interconnect the processors, directly accessible through read and write operations in the register file of the processors. RAW has been implemented in a 0.15 µm process running at a worst case clock frequency up to 225 MHz. The area is reported to be 256mm^2 [7]. The RAW processor with its 16 tiles has a maximum peak performance of 3.6 GOPS. Large portions of a RAW tile constitute instruction fetch and decode units, a floating point unit and units for packet oriented interconnection routing.

The XPP, on the other hand, constitutes a 2D array of word oriented, single instruction-depth processing elements. The processing elements in the XPP are fixed-point ALU units without instruction or data memory. This means that instructions are statically mapped to processing elements and all control functions must be implemented using these. The instruction sequencing as done in a microprocessor must be performed by mapping instructions statically to the XPP array and/or using dynamic reconfiguration of the array. Unlike RAW, which have multiple, full-fledged packet routed interconnection networks, XPP ALUs are connected via statically switched networks.

The XPP core has been implemented in a 0.13 µm process running at 64 MHz and the area requirement was reported to be 32 mm^2. This roughly corresponds to a peak performance of 4.1 GOPS for the 64 ALU XPP array [8]. We are not aiming at comparing the two specific architectures. Rather, we see them as examples of two architecture classes: one chip multiprocessor with dedicated control and program sequencing circuitry (RAW), the other one an ALU array (XPP) with resources only for pure dataflow processing. In the latter case, some of the processing resources need to be used for control, and we are interested to see how large portion is needed.

It is probably easier to come close to the peak performance in the chip multiprocessor. On the other hand, the peak performance is much higher in the ALU array. Therefore we are interested in the processing efficiency of the XPP array and use RAW as a reference. To make the comparison we need to normalize the two architectures, which we do by estimating the area of the XPP array in the same process as RAW. When normalizing the area of the XPP to the 0.15 µm process used for the RAW implementation, one XPP array would be about 42mm^2, which means that an XPP array occupies about 17 % (42/256) of the RAW area. Thus, 6 XPP arrays could be fitted on the same area as one RAW array. When calculating peak performance we assume the same clock frequencies as reported for XPP and RAW, this corresponds to a peak processing performance of 24.5GOPS (6 arrays at 4 GOPS) for XPP compared to 3.6 GOPS for RAW, using the same area. Using about 15 percent of the peak performance for computations in an application would still mean that the ALU array is competitive compared to the more traditional chip multiprocessor architecture. (However, it should be noted that XPP64-A is a 24-bit architecture whilst RAW is a 32-bit architecture). Theoretically this means that up to 85% of the ALU resources, can be used to implement control functions for more complex algorithms, still the ALU array could be more effective than a chip multiprocessor, when comparing performance/area.

An obvious tradeoff is that, by making the processing elements simpler there is also a requirement that more ALU resources are used to implement the program control flow. Since the XPP processing elements have no instruction memory, the array needs to be reconfigured if algorithms are too large to fit on the array. This means that the designer first tries to statically map as large portion of the application, data and control flow, as possible. If the complete algorithm does not fit on the array it must be temporally divided and the array is reconfigured during runtime when executing the algorithm. Even if ALU arrays offer a lot of parallelism, there is a great challenge to efficiently make use of it.

4. Implementation study

We have chosen to study the XPP architecture as a candidate for micro-level reconfigurability in our outlined architecture as shown in Figure 2. In the implementation study we address the control of the computations implemented on this coarse-grained reconfigurable architecture. On this type of reconfigurable array architecture both computations and control functions have to be implemented using reconfigurable elements and communication paths. Therefore, it is interesting to see the relation between how much of the resources are used for computations and how much are used for control.

4.1. The XPP Architecture

The “extreme processing platform”, XPP, architecture from PACT XPP Technologies, represents a word-level reconfigurable architecture. A recent implementation of this architecture, the XPP64-A, operates on 24-bit data.

The architecture consists of a number of parallel processing array elements labeled PAEs. There are in turn two types of PAEs, arithmetic and memory, labeled ALU-PAE and RAM-PAE respectively. The PAEs are arranged in an array with both ALU-PAEs and RAM-PAEs, as illustrated in Figure 3. The XPP64-A has an 8x10 array, where the first and the last columns are RAM-PAEs. Four I/O element objects are connected to each of the four corners of the array. The I/O can be configured to read simple streaming input as well as to access an external memory.

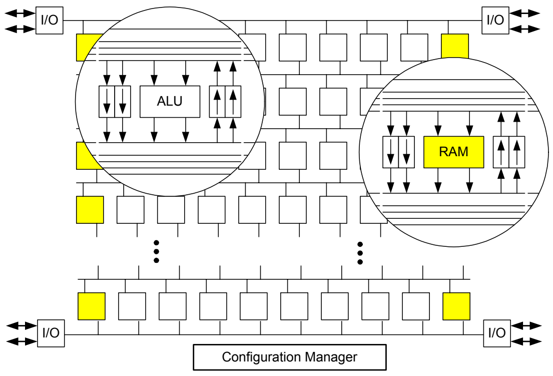


Figure 3 The XPP architecture. The highlighted areas show an ALU-PAE and a RAM-PAE including the vertical routing resources.

The array elements can be configured to execute their operations when triggered by an event signal indicating that new data is available at the input ports. A new output can be produced every clock cycle and the result constitutes a data output and an event signal indicating that data is ready on the output port.

The ALU-PAE comprises a data path, with two inputs and two outputs, and two vertical routing resources. The vertical routing resources can also perform some arithmetic and control operations. One of the two is used for forward routing and the other for backward routing. The forward routing resource, labeled FREG, is, besides for routing, also used for control operations such as merging or swapping two data streams. The backward routing resource, BREG, can be used both for routing and for some simple arithmetic operation between the two inputs. There are also additional routing resources for event signals which can be used to control PAE execution. The RAM-PAE is exactly the same as the ALU-PAE except that the data path is exchanged by a static RAM. The RAM-PAE can be configured to act either as a dual-ported memory or as a FIFO.

The design flow for the XPP technology constitutes using either a vectorizing C-compiler or direct programming in the native mapping language (NML), which is the XPP-ISA assembly language.

4.2. Implementation of the FFT algorithm

FFT is one important class of the algorithms often used in signal processing applications. Some of the most common FFT algorithms are the well known Radix FFT algorithms where the algorithm complexity in computing the Fourier transformation can be reduced from N^2 to $N \log N$ [8]. By inspecting the characteristic Radix-2 pattern, which is illustrated in Figure 4, it is easily seen that the algorithm can be divided into $\log_R N$ consecutive stages, where N is the number of samples to be processed and R is the radix of the FFT. The data in each stage is being processed and rearranged according to the characteristic radix pattern, often referred to as bit reversed order, while it is propagated through the algorithm. The basic computation performed at each stage is called a butterfly. As can be seen in Figure 4, the complex sample b is multiplied with a complex phase shift constant W_N. The output pair (A,B) is then formed by adding and subtracting the complex sample a with bW_N. In total, N/R butterflies have to be computed in each FFT stage. The FFT was chosen for the implementation study since the radix pattern requires fairly complex control functionality.

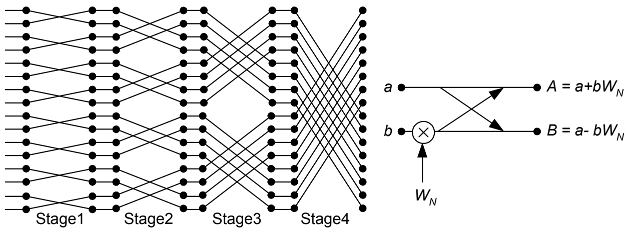


Figure 4 A 16-point Radix-2 Decimation In Time FFT communication pattern is shown to the left and a butterfly computation to the right.

We have used available development tools for the XPP reconfigurable architecture to implement and simulate a pipelined Radix-2 FFT. Figure 5 shows the functional schematic of an FFT module that can be configured to compute one or several butterfly stages. In this case, we use a double buffering technique between two consecutive stages in the FFT so that an entire sample sequence can be read and written in parallel without conflicts in the address space.

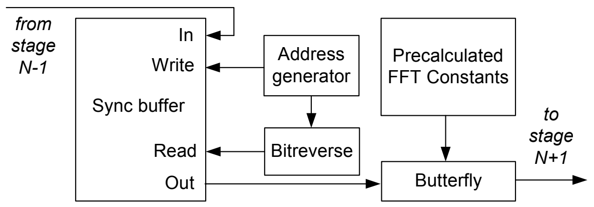


Figure 5 Double buffered FFT stage.

Radix-2 address calculation is performed when reading the data from the buffer and streaming it to the butterfly. In order to prevent the input and output streams from accessing the same address range concurrently, the write and read address streams have to be synchronized before the address counting is restarted after each butterfly stage. The phase shift constants that are multiplied with the input in the butterfly computation are precalculated and stored internally using RAM-PAEs. After the data has been streamed and processed through the butterfly it is being passed on to the consecutive stage. The RAM-PAEs can be configured to form one or several larger RAM banks dependent on the required size. We are using two separate I/O channels to stream data into the array, since we are using separate words for the real and the imaginary parts of the complex valued input. Therefore two double buffers are used to store the real and imaginary data and two memory banks to store the real and imaginary parts of the FFT constants. Figure 6 shows a block schematic of the implemented double buffer.

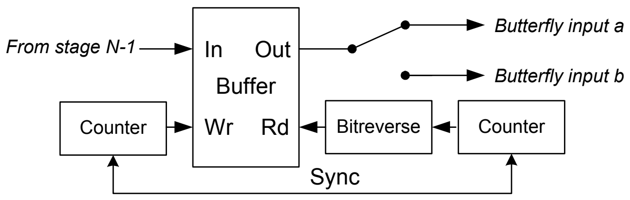


Figure 6 Synchronized I/O buffers between two consecutive FFT stages.

The buffer address space is divided into a lower and an upper half and we use two separate counters to generate addresses for reading and writing to the buffer. The counter to the left in Figure 6 generates write addresses in consecutive order for the input stream. The counter to the right produces the read address which then is bit reversed to generate the characteristic Radix-2 address pattern. A counter can be implemented by combining several FREG and BREG instructions in a single PAE. The Radix-2 bit reversal is implemented using a combination of PAEs performing shifts and additions. When both counters have reached the end of each sequence, a synchronization signal is generated and fed back to the counters to make them reset and start over.

4.3. Radix-2 Butterfly

Figure 7 shows the data operation that is performed in the butterfly. Two samples are needed to compute the output from each of the $N/2$ butterflies in each stage. As could be seen in Figure 6, the sequential stream of samples are alternatingly forwarded to the a and b inputs in the butterfly through a PAE demux instruction. Since the XPP is a fixed point arithmetic architecture, the results from the arithmetic operations have to be scaled to assure the output is kept within the used integer range. After the data has propagated through the butterfly, the resulting A and B samples are merged into a serial stream and then fed through the I/O channels to the next stage. This is implemented using PAE merge instructions.

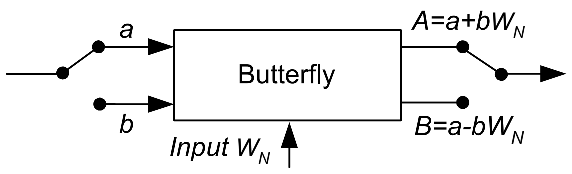


Figure 7 Data stream through the Radix-2

The implemented FFT configuration is capable of processing a continuous stream of data with the speed of one complex-valued word length sample per clock cycle. All programming has been done using the NML language. All PAE configurations have been placed manually. This was made in order to achieve a functionally correct and balanced dataflow through the array.

Table 1 shows how large portion of the available reconfigurable resources that have been used for the FFT implementation. The available number of configurable objects of each kind on the array is marked within the parentheses next to the name of each object. This is shown to the left in the table. For each class, the number of objects that have been configured for the FFT implementation is listed in the middle. The rightmost column shows the ratio, in percent, between used objects of each kind through the available objects of that kind.

Table 1. Array resource usage for the Radix-2 FFT.

Object	Used	Total share
ALUs (64)	12	19%
FREGs (80)	31	39%
BREGs (80)	44	55%
IOs (8)	4	50%

As can be concluded from the figures in the table, we use less than half of the available array resources, except for the BREGs. However, a portion of these BREGs are used for vertical routing of data on the array. We have not optimized the usage of routing resources which would likely be possible to do with more careful and strategic placement of the configured PAEs on the array.

A considerable part of all the arithmetic operations performed on the array are used for controlling the dataflow on the array. In Table 2 we have listed how many of the configured ALU operations in total that are used for controlling the dataflow and how many are used in the butterfly, where the signal output is calculated. About 74 percent of the configured ALU resources are used for dataflow control and 26 percent are used for computing the butterfly outputs. A great portion of the resources used for the dataflow control is a consequence of using separate address counters and control functions for each of the I/O streams. The distribution of the figures may, at first look, be surprising. However, one should bear in mind that, in ordinary program code, many algorithms normally require portions of the code to do address calculations, incrementing and testing loop bound counters etc.

Table 2. Distribution of the total Arithmetic operations.

Functionality	ALU ops	Total share
Dataflow control ops	34	74%
Butterfly ops	12	26%

Out of the 12 ALU instructions used for the butterfly configuration, two are used for fixed point scaling. The 34 ALU instructions implementing the dataflow control functionality are used for address counters, bit reverse and data path switching operations across the array. What proved to be difficult was the implementation of the dataflow control structures while maintaining a balanced dataflow within the entire algorithm. Even though the XPP has automatic dataflow synchronization, it required time consuming, low-level engineering to balance the dataflow through all parts of the FFT and keep up a high throughput rate. Algorithms like the FFT that contain address calculations, loop control and memory synchronization tend to require much of the resources to be spent on implementing the dataflow control in the algorithms. It might be possible to optimize the FFT implementation so that fewer configurable objects would be needed. However, solutions using long pipeline data paths tend to be complex to implement and balance on the XPP array. Therefore, in order to reduce complexity with synchronization and dataflow balance, we used separate flow control for each of the two I/O streams which resulted in the presented area tradeoff.

Still, referring back to the coarse area comparison between an array of ALUs and a chip multi processor, one could argue that, for the computation studied, the ALU array makes more efficient use of the silicon area. Even if the majority of the resources were used for control purposes, the performance achieved in our implementation is well beyond the peak performance of the more coarse-grained multi processor width dedicated control resources.

4.4. Macro-level mapping

The implemented FFT module could be used as a building block for FFTs of variable size. The performance could be adapted for different requirements, by pipelining FFT modules using several nodes, which is illustrated in Figure 8. The highest configuration level — the macro level — constitutes a higher abstraction level for efficient mapping of pipeline parallelism within an algorithm, like the implemented FFT, or between different kinds of functions in a signal processing chain, like in the earlier described radar signal processing application. Figure 9 illustrates the general situation. Different functions in the graph should be allocated to either a single node or a group of nodes depending on resource and performance requirements. The mapping onto the macro level should try to exploit pipeline parallelism as well as parallelism stemming from multiple inputs.

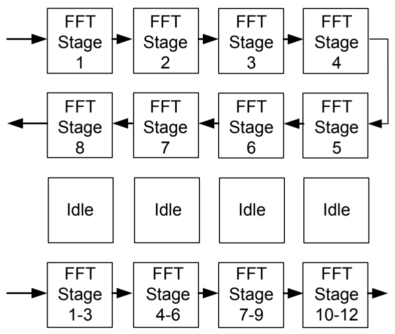


Figure 8 Pipelined FFT computations mapped onto the macro-level.

To support dynamic mapping, the macro level architecture needs to support reconfigurable interconnection between the nodes. Further studies of how to best implement this, as well as how to control the configuration flow at the macro level and the control of dataflow between nodes, will be needed. These studies must be made using more and realistic applications to capture the characteristics of the application domain.

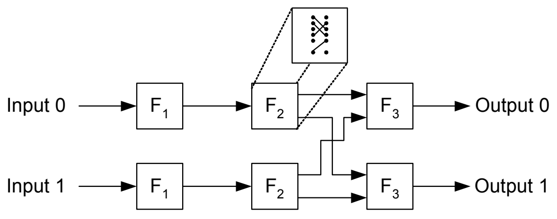


Figure 9 Functions that should map onto the macro-level structure.

5. Discussion and Conclusions

We have outlined a two-level configurable architecture, intended for embedded high-performance signal processing. The coarse-grained macro-level consists of nodes interconnected in a mesh. For the micro-level we believe reconfigurable hardware could be used for efficient implementation of the signal processing functions.

We described an implementation mapping an FFT algorithm onto a coarse-grained reconfigurable architecture. With this implementation study we quantified the resource usage in the FFT algorithm. The main focus of the study was to find out how the architectural resources were allocated to different parts of the algorithm, in particular the amount of resources needed to feed the main computational core with data. This would roughly correspond to the program code and control logic in a microprocessor.

The implementation study shows that a considerable part of the resources are used for implementing the control structures in the FFT implementation. By implementing these structures in parallel the data can be kept streaming through the nodes continuously at maximum I/O speed. The needed allocation of resources for control is not surprising since, in this architecture, there are no dedicated functional units for control purpose. Many algorithms in signal processing require less control than the FFT. It is possible to map these with good efficiency. Other algorithms will need reconfiguration of the array during run-time. This implies efficient reuse of hardware, but may have negative impact on performance.

The FFT implementation has also been used to show different ways of mapping a pipelined algorithm onto several nodes of configurable arrays.

Further studies must be made to analyze the requirements on the interface to the micro-level nodes. On the macro-level the requirements on the communication and configuration must be analyzed for different, realistic applications.

6. Acknowledgments

This research has been financed by a grant from the Knowledge Foundation. We are also grateful to PACT XPP Technologies for making their development tools available to us.

7. References

[1] W. Liu and V. Prasanna, "Utilizing the Power of High-Performance Computing", *IEEE Sig. Proc. Mag.*, Sept. 1998, pp. 85-100.

[2] A. DeHon, The Density Advantage of Configurable Computing, *IEEE Computer*, Vol. 33, No. 4, April 2000, pp 41-49.

[3] Elixent, "Changing the electronic landscape", http://www.elixent.com.

[4] V. Baumgarte, F. May, M. Vorbach, and M. Weinhardt, "PACT XPP – A Self-Reconfigurable Data Processing Architecture," *Int'l. Conf. on Engineering of Reconfigurable Systems and Algorithms ERSA 2001*, Proc., CSREA Press, 2001.

[5] R. Baines and D. Pulley, "A Total Cost Approach to Evaluating Different Reconfigurable Architectures for Baseband Processing in Wireless Receivers", *IEEE Communications Magazine*, vol. 41,no. 1, Jan. 2003, pp. 105-113.

[6] Paul Masters, "A look into QuickSilver's ACM architecture", *EETimes*, http://www.eetimes.com/story/OEG20020911S0070

[7] M. B. Taylor, et. al., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs", *IEEE Micro*, Mar/Apr 2002, pp. 25-35.

[8] J. G. Proakis, D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd ed., Prentice-Hall International, UK (1996)

Paper B

Baseband Processing in 3G UMTS Radio Base Stations

Bengtsson, J. (2006). Baseband Processing in 3G UMTS Radio Base Stations. Technical report IDE0629, School of Information Science, Computer and Electrical Engineering, Halmstad University, Halmstad, Sweden, Feb. 2006.

Baseband Processing in 3G UMTS Radio Base Stations

Technical Report IDE0629

Jerker Bengtsson

Centre for Research on Embedded Systems
School of Information Science, Computer and Electrical Engineering
Halmstad University
Halmstad, Sweden

Abstract

This report presents a study of functionality, service dataflows, computation characteristics and processing parameters for baseband processing in radio base stations. The study has been performed with the objective to develop a programming model that is natural and efficient to use for baseband programming and which can be efficiently compiled to parallel computing structures. In order to achieve this objective it is necessary to analyse and understand the logical architecture of the application in order to be able to define processing characteristics and thereby requirements on languages as well as on physical system architectures. Moreover, to be able to test and verify programming and mapping of functions it is necessary to have realistic but still manageable test cases.

The study is focused on the third generation partnership project (3GPP) standard specifications for 3G radio base stations. The specifications cover the complete 3G network-architecture and are quite extensive and complex. To make experiments manageable, it is necessary to abstract system functionality that is not directly relevant for the RBS baseband processing. Moreover, the standard specifications only describe the required processing functionality on an abstract logical level. In this report, the functionality of the baseband functions is explained and also described using illustrations of dataflows and abstract mapping of two 3G service cases.

The results of the study constitute a comprehensive description of the processing flow and the mapping of user data channels in 3G radio base stations – spanning data and control input from layer 2 to physical channel output from layer 1. Data dependencies between functions are illustrated with figures and it is concluded that these dependencies are of producer/consumer type. It is discussed how different functions can be mapped in MIMD and SIMD fashion with regard to the data dependencies, the data stream lengths and the control operations required to handle bit stream processing on word-length processor architectures.

Table of contents

1. Introduction

This report concludes a study of functionality, dataflows as well as computation characteristics and processing parameters for baseband processing in Universal Mobile Telecommunication Standard (UMTS) radio base stations (RBS). The study is part of a research project where the objective is to investigate programming models and efficient mapping techniques for parallel and reconfigurable processing platforms. Efficient parallel mapping requires a programming model that is natural to use for application programming and which can be efficiently compiled to parallel structures. Thus, it should be possible to express parallelism and application-characteristic dataflows, which can be analysed and exploited at compile-time. Moreover, such a programming model should also be portable to different architectures. In order to achieve this objective it is required to analyse and understand the application, to be able to define requirements on a language and to be able to conduct realistic implementation experiments.

There are two main purposes with this study. First, the standard specifications for third generation (3G) networks are quite extensive and complex and quite difficult to grasp. To make experiments manageable, it is necessary to abstract system details that are not directly relevant for the RBS data processing. The second purpose is to characterise the baseband functions, the computations and the dataflows to understand what kind of operations and logical functionality that must be expressed in a programming language.

There are several 3G-enabling radio technologies, such as EDGE, CDMA 2000 (also called CDMA 2K) and WCDMA. WCDMA is an abbreviation for wideband code division multiple access and it is the radio technology chosen by the 3GPP organisation for UMTS networks [1]. Unlike earlier generations, 3G systems have been designed to support high bandwidth multimedia services and to support multiple simultaneous services, multiplexed on a single user channel [2].

In an RBS, the term downlink denotes the transmitting part of a communication link and the receiver link is called uplink. FDD is an abbreviation for Frequency Division Duplex, in which separate frequency bands are used for uplink and downlink carriers. There is also a standard for Time Division Duplex (TDD) in UMTS, but the FDD standard is the technology used in existing telecommunication systems. This study is focused on the WCDMA FDD standard and the downlink processing in an RBS. The technical specification in the study document has been compiled from release 5 of the 3GPP standard specifications [1].

This report is organized as follows. Section 2 describes the logical network architecture and how the RBS relates to the network. The RBS data input format, configuration parameters and the logical function flow for the baseband processing are presented in Section 3. In section 4, the WCDMA code spreading technique and processing rates are discussed. Section 5 discuses two service examples used to describe data- and function-flows. The technical specifications of the studied baseband functions are presented in Section 6 which is the main section of the report. We explain the purpose the functions and illustrate dataflows and mapping characteristics using the service examples from Section 5. Finally, the study is summarised in Section 7.

2. The UTRAN architecture

This section presents an overview of the modular UMTS network architecture. The logic modules and interfaces in the network architecture are briefly described to explain the role of an RBS in an UMTS network. The WCDMA baseband technology constitutes the core of the UMTS terrestrial radio access network (UTRAN) architecture [2]. The UTRAN architecture is a standardised, logical architecture comprising one or several radio network subsystems (RNS), see Figure 1. An RNS is in turn a sub-network comprising a radio network controller (RNC) and one or several Node Bs. Node B is the terminology used for an RBS by the 3GPP.

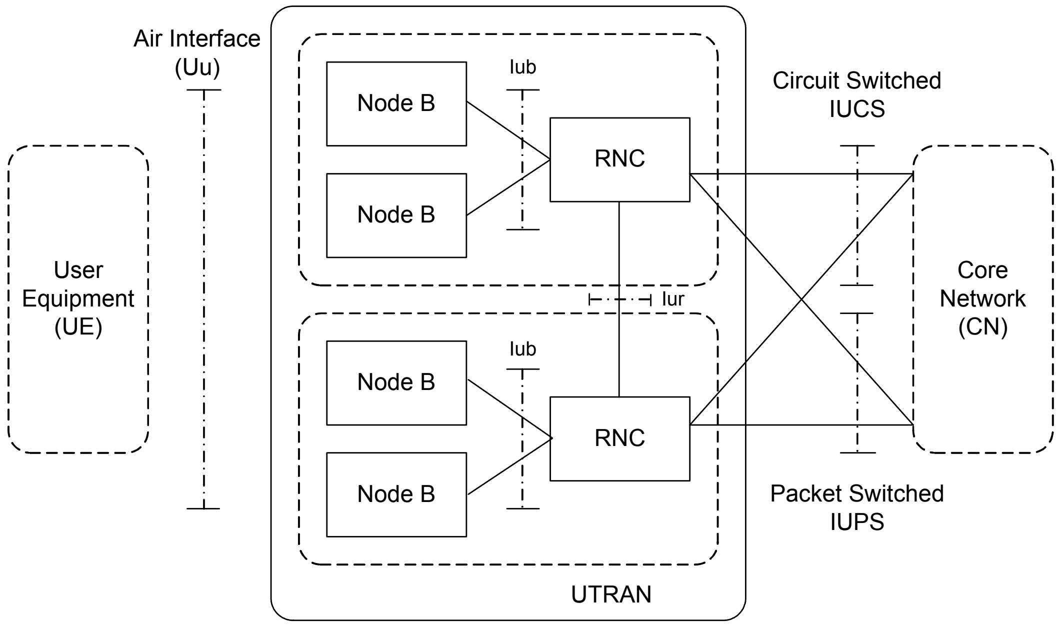


Figure 1 The UTRAN architecture

The RNC controls the Node B radio access resources within an RNS, i.e. the lowest layer of the network layers – the physical layer (L1) – and the radio. The RNC is responsible for setting up physical radio links via the radio access resources – so called radio access bearers (RAB) – on user service requests. The RNC also manages congestion control and load balancing of the allocated channels and it constitutes the termination point between the RNS and the core network. This is illustrated with the CN interface in Figure 1.

2.1 Node B

The functionality of Node B can in general terms be described as a mapping procedure, between logical channels from higher layers (L2 and above) and the physical channels (L1). In the downlink, data frames from higher layer transport channels are encoded, grouped and modulated before being transmitted through the antenna. In the uplink, physical channels are demodulated, decoded and mapped onto higher layer data frames.

The transport channels comprise channels of both dedicated and common type. The dedicated channels (DCH) are allocated for single users while the common channels are shared for several users. Several transport channels can be allocated for a single user and these are multiplexed into one coded composite transport channel (CCTRCH). Multiple transport channels can be allocated for one service and/or for multiple services running in parallel. For example, a user could be downloading e-mails in the background while speaking in the phone at the same time. In release 5 of the 3GPP standard, which is the standard release used for this report, it is only possible to allocate one CCTRCH for a single mobile user equipment (UE).

2.2 The UTRAN interfaces

There are two RNC interfaces to the core network (CN) — the IUPS interface for packet switched communication and the IUCS interface for circuit switched communication. The UTRAN CN interface is designed to be logically compatible with the GSM network infrastructure.

The IUR interface is an interface for communication between RNCs of different RNSs. It is used for network relaying, or hand-over, of radio links when a mobile UE has relocated geographically in the network. The RNC that initiates a radio access link is denoted the serving RNC (SRNC). The SRNC is the owner of the link. When a mobile UE relocates to a different RNS, the RNC in the RNS to which the mobile has relocated to becomes a drift RNC (DRNC). The DRNC relays the radio link to the SRNC and no L2 processing is performed at the DRNC.

The IUB interface is the L2 to L1 interface between a Node B and the RNC. This interface is the termination point for the RBS managed resources. The Uu interface is the WCDMA air interface between Node B and mobile UEs. The physical layer processing in the RBS is encapsulated between the Iub and the Uu interfaces.

3. Transport channel multiplexing in Node B

This section presents the L1 downlink logical function flow in the RBS and the frame format of input data from L2. Service payload is mapped on dedicated transport channels (DCH) and a transport block is the smallest data unit for input payload. Several transport blocks can be mapped on one transport channel and new blocks of data arrive by a deterministic time interval, Figure 2. This time interval is denoted as the transmission time interval (TTI).

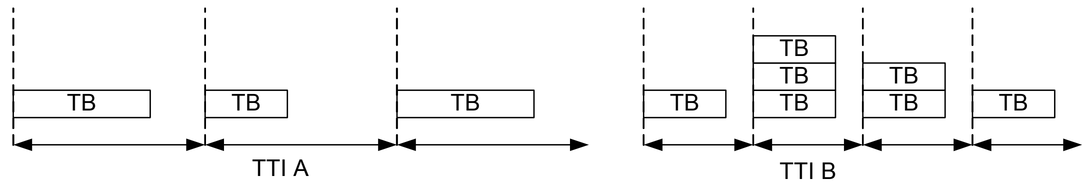
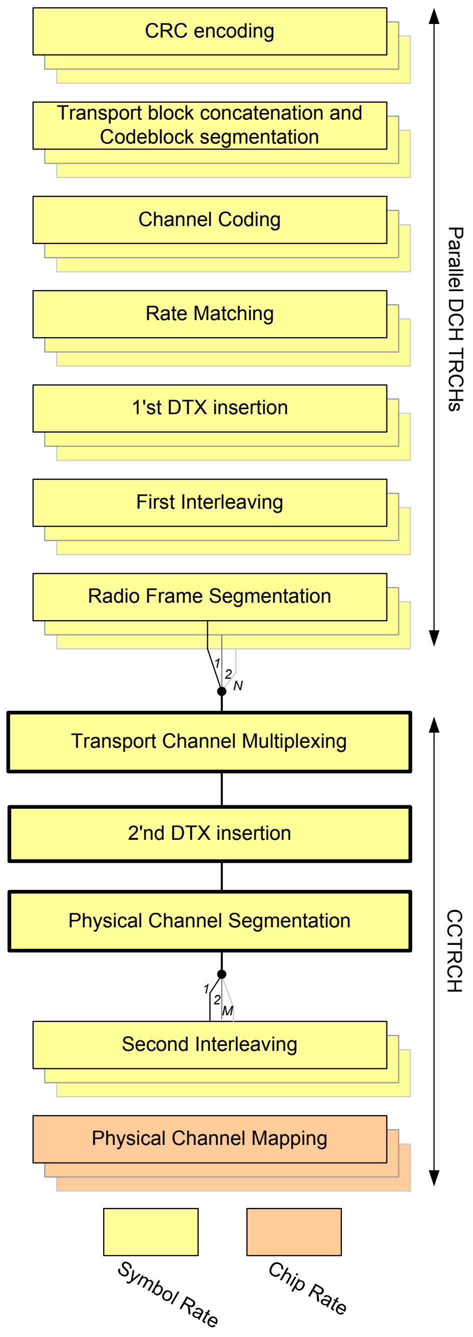


Figure 2 Examples of Transport block combinations

Transport blocks that are grouped within a TTI and belong to the same service constitute a transport block set [5]. A new transport block set is processed every TTI and the TTI is always a multiple of the 10 ms radio frame (10, 20, 40 or 80 ms). Transport blocks in different TTIs can have variable bit lengths (illustrated to the left in Figure 2), but within a transport block set, all blocks must be of equal length (illustrated to the right in Figure 2).



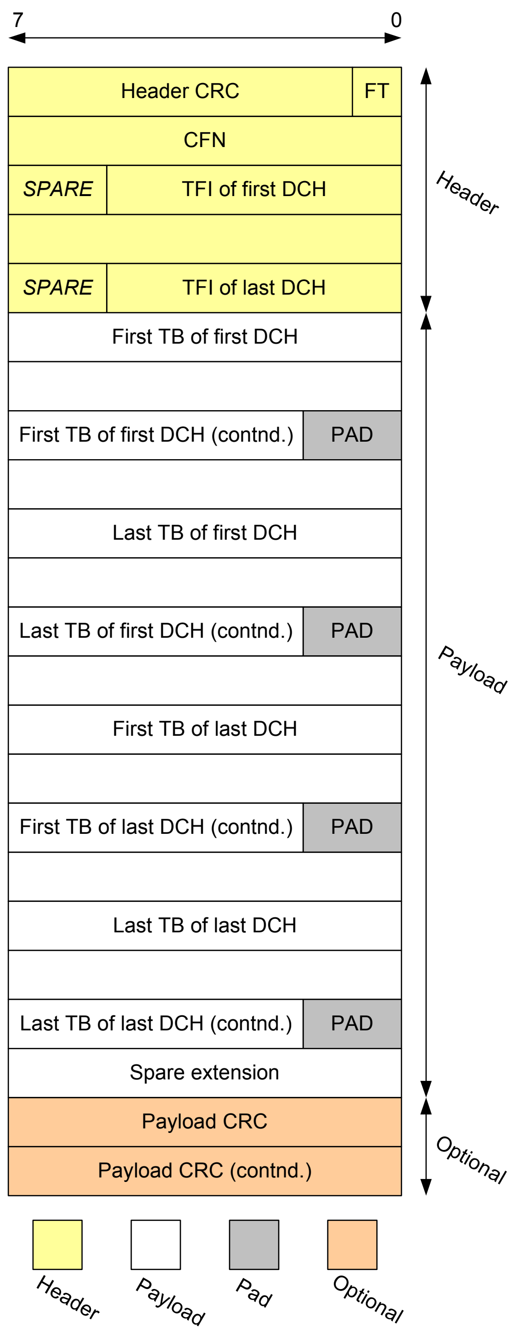


Figure 3 DCH frame format. The horizontal axis is graded in bytes and the vertical axis by byte octets

Figure 4 Downlink processing functions

There are two L2 frame format types for the DCHs – data frames and control frames [3]. All frames have a header field and a field for payload. The structure of a data frame can be seen in Figure 3.The baseband processing in Node B constitutes a pipelined sequence of functions, as shown in Figure 4. The figure shows *N* parallel DCH transport channels, which are multiplexed to form a CCTRCH, and later, depending on the bandwidth requirements, de-multiplexed on *M* physical channels. The processing mode of the baseband functions is configured using a set of service parameters. These processing parameters must also be set at the receiver side. Different methods can be used to configure the processing mode parameters in the receiver – TFCI based detection, blind transport format detection or guided detection [4]. The mapping examples, described later in Section 5, assume TFCI based detection.

The transport blocks transmitted within a transport channel in the same radio frame interval are combined with a transport format indicator (TFI). A transport format combination indicator (TFCI) combines the TFIs for the services that are multiplexed on a single CCTRCH. The TFCI is encoded and transmitted over the physical channel together with the DCHs which are multiplexed to form a CCTRCH. If several DCHs are used, only one TFCI is sent with one of the DCHs. At the receiver side, the TFCI can be decoded and the function parameters are then configured so that the data frames can be processed correctly.

The TFI has two parts – a dynamic part and a semi-static part. The dynamic parameters can be altered each TTI, while the semi-static are configured once when a service is set up. The TFI parameters are listed in Table 1 below.

Table 1 Transport Format Attribute options

Dynamic Part	Transport Block Size	0 to 5000 bit
	Transport Block Set Size	0 to 200000 bit
Semi-Static Part	Transmission Time Interval	10, 20, 40, 80 ms
	Channel Coding Type	No coding, Convolution coding, Turbo coding
	Code Rates (Convolution)	1/2 or 1/3
	CRC size	0, 8, 12, 16, 24

4. Code spreading and processing rates

The L1 processing is performed with different processing rates at different stages. These rates are categorised as symbol rate processing and chip rate processing. Symbol rate corresponds to the rate of the information bits. That is, each bit processed in the physical layer corresponds to one information symbol (e.g. 1 or 0) in the payload. The WCDMA technology uses code spreading to enable transmission of multiple channels on the same frequency band. Symbols are spread using orthogonal channel codes. A channel code is a finite sequence of 1's and 0's, which are denoted chips, and these sequences are selected so that they are orthogonal to each other. Each symbol is spread to a sequence of chips and the processing rate is higher after this spreading, since each logical symbol now is represented by a longer chip-sequence. This rate is referred to as chip rate.

In the downlink, all baseband functions from the CRC attachment through the physical channel mapping are processed at symbol rate. The spreading operation is performed after physical channel mapping and the output after spreading are chip rate. In the uplink, parts of the RAKE[1] receiver and the preceding functions are processed at chip rate. Functions proceeding to the RAKE receiver are processed at symbol rate.

The chip rate on a physical channel in UMTS is 3.84 Mchips and this rate is constant. A radio frame is transmitted during a 10 ms interval and this corresponds to a length of 38400 chips. Each radio frame is divided into 15 slots and each slot corresponds to 2560 chips. The length of the spread factor (SF) determines the rate with which symbols are mapped on the physical channels. The spread factor is always a multiple of 2 and in the downlink the SF can vary between 4 and 512. Dynamic bit rates can be allocated through configuration of the spread factor length and by using multiple channel codes.

5. Service mapping

This section presents two 3G service examples which are used to reason about mapping of data streams and about processing characteristics of the baseband functions in Node B. The first example is derived on a standard service for voice transmission. The second example is a more general case for high-bit rate services. Only mapping and processing on DCH transport channels are considered and it is assumed that the required service RABs have been setup by the RNC.

5.1 AMR voice

Adaptive Multi Rate (AMR) is a technique used for coding and decoding of dynamic rate speech services in UMTS [5]. The bit rate can be altered by the radio network during the service session, hence the name Adaptive Multi Rate[2]. There are two standard AMR codecs included in the 3GPP specification – narrowband AMR [6] [7] and wideband AMR [8]. The wideband AMR is specified in release 5 of the 3GPP technical specifications. The main difference is that the wideband AMR coder offers higher voice quality by means of an increased sample rate (16 kHz for wideband compared to 8 kHz when using narrowband).

Nine data rates between 6.60 kbps and 23.85 kbps are available in the wideband AMR standard. The narrowband AMR codec offers 8 data rates between 4.75 kbps and 12.2 kbps, where some of these codec rates are compatible with the AMR codec used for the

[1] The RAKE receiver is used in the uplink to decode the channel codes
[2] Current networks have been deployed with AMR that uses fixed bit-rates

GSM system. The radio access network can adaptively control the bit rate and for the narrowband AMR service the bit rate can be changed for each TTI.

When the AMR encoder encodes data, the bits are arranged in different classes according to how important they are for speech quality. The encoded bits are categorized into three classes - A, B and C - where A are the most important bits and C the least important. Stronger coding and CRC attachment can be applied on the class A bits separately, while the less important bits can be transmitted using less coding strength and without CRC attachment.

5.2 High bit-rate data transmission

The standard specifies a set of UE classes with different radio access capabilities. These capability classes define what data rates and services that must be supported for a UE of a specific class. We have used parameters required for UEs of the highest capability class, which can handle bit rates up to 2048 kbps [9]. The maximum number of simultaneous transport channels for the 2048 kbps class is 16. It is assumed that all transport channels can be of DCH type. One CCTRCH using up to 3 physical channel codes can be received by a UE of this class. The maximum number of transport blocks within a TTI is 96 and the maximum number of bits that can be received within a 10 ms radio frame interval is 57600.

6. Downlink processing functions

In this section we explain and discuss the baseband functions (shown in Figure 4) with focus on the downlink data channel processing in the RBS. We illustrate processing and mapping of services to these functions using the two cases of service transmissions that were presented in Section 5. The mapping is based on the logical specifications given by the 3GPP standard documents and test implementations that have been made for a sub set of these functions [1]. We use figure abstractions in connection to each of the described functions to abstractly reason about computations, potential parallelism and dataflow mapping. The mapping has been kept on an abstract level for each function separately and the specific services are discussed under the paragraphs **AMR service** and **High bit-rate data service** in connection to each function sub-section.

In first hand, we are interested in identifying data dependencies, potential parallelism and typical computations in order to define different function implementations which desirably should be possible express in a programming language.

6.1 CRC attachment

Cyclic Redundancy Check (CRC) is a function used to detect bit errors after transmission. At the sender side, a checksum is calculated and appended to the data to be transmitted, Figure 5. In the figure, A_i represents the length of the transport block before CRC attachment and L_i the length of the checksum. At reception, the checksum is recalculated and if no errors have been introduced, this checksum will match the appended checksum. The CRC checksum is calculated using polynomial division [10]. The dividend corresponds to the bits in a transport block and the divisor is a specified generator poly-nomial, known both by the sender and the receiver. This polynomial division results in a rest term - the checksum. The generator polynomials used for CRC in UMTS are of length 24, 26, 12, 8 or 0 bits. The generator polynomial to be used for a specific transport channel is signalled to the physical layer from higher layers in the TFIs.

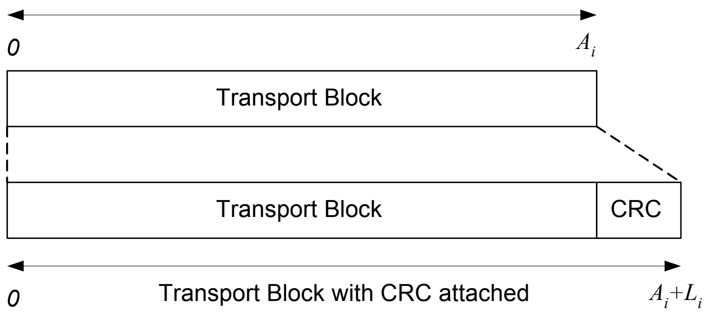


Figure 5 A CRC checksum is appended to each transport block

The polynomial arithmetic is performed in GF(2), where GF is an abbreviation for Galois Field and 2 represents a number field of width 2. In practice this means that addition and subtraction are performed modulo 2. (i.e. no carry is propagated) [11]. Thus, addition and subtraction can be implemented using simple XOR arithmetic. The following generator polynomials are specified for usage in the 3GPP UMTS standard

$G_{CRC24}(D) = D^{24} + D^{23} + D^6 + D^5 + D + 1$
$G_{CRC16}(D) = D^{16} + D^{12} + D^5 + D + 1$
$G_{CRC12}(D) = D^{12} + D^{11} + D^3 + D^2 + D + 1$
$G_{CRC8}(D) = D^8 + D^7 + D^4 + D^3 + D + 1$

In GF(2) each D^i term represents a '1' at position *i* in the binary number format and the others are '0'. For example, the polynomial G_{CRC8} corresponds to the binary number 110011011.

Transport blocks are padded with a number of zeros, as many as the length of the generator polynomial. That is, if G_{CRC16} is used, 16 zeros shall be appended to the transport block. After zero padding, the entire transport block (the dividend), is divided by the generator polynomial (the divisor). If there are no transport blocks on the input, no checksum is calculated. If the input transport block size is of zero length, a checksum is still calculated (all checksum bits will be zero).

AMR service

The AMR bit classes are mapped using three RAB sub-flows; one transport block for each bit class, mapped on three separate DCH transport channels. CRC attachment is only applied for the class-A sub-flow using the CRC_{12} polynomial. Thus, one transport block per user must be processed each TTI, see Figure 6. A possibility for exploitation of SIMD parallelism is to combine AMR class A channel streams from several users. However, efficient SIMD parallelization of the CRC function for AMR services will require that the channels are computed using the same CRC polynomial. This is because that many efficient CRC algorithms require that the input must be sliced according to the polynomial length [10]. Also, it will require that all users are using the same AMR data rate encoding modes to ensure that all bit streams are equal in length. Input slicing and stream alignment require control operations. These control operations must be the same for all streams to be able to SIMD vectorize efficiently.

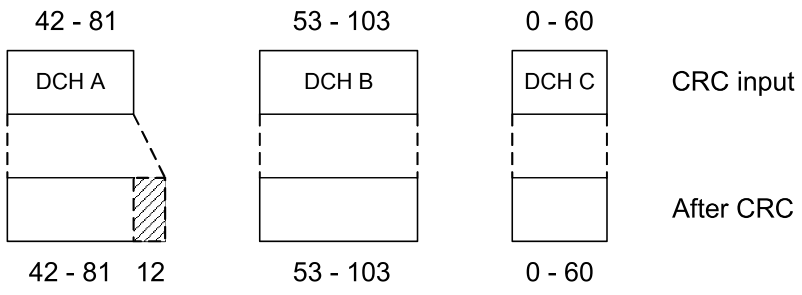


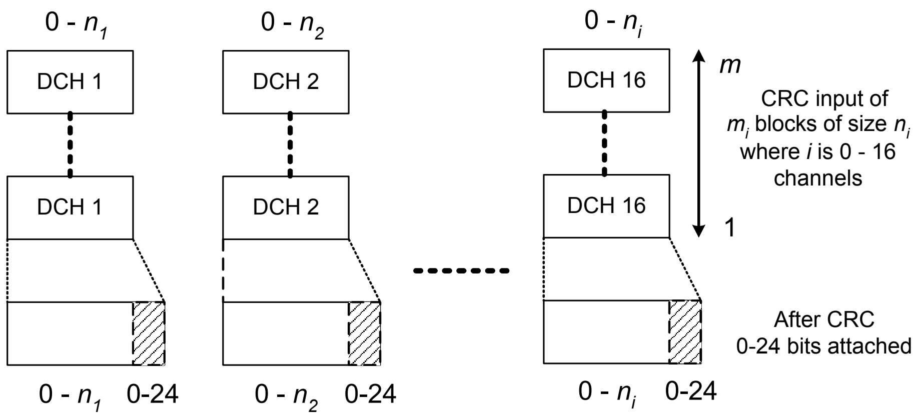
Figure 6 Transport block lengths can take 8 different discrete values within the given bounds. Each value corresponds to a certain AMR bit-rate.

High bit-rate data service

Data can be mapped using multiple transport blocks distributed on several transport channels. All transport blocks within a transport channel are of equal length (*n* in Figure 7) and the CRC polynomial is the same for all blocks. Since there are no data dependencies between transport blocks, it is possible to SIMD vectorize several blocks within a channel.

Like in the AMR case, efficient SIMD vectorization of data from several transport channels will require that the transport block lengths and CRC polynomial are the same for these channels. Otherwise, more coarse-grained parallelism can be exploited by mapping the channel streams in a MIMD parallel fashion.

For the 2048 kbps class, at most 96 transport blocks can be mapped to 16 DCH streams, as shown in the figure. The maximum sum of bits that can be received within a TTI (all DCH:s included) is 57600 ($m_i \times n_i \times n_{channels}$).



0 - n_1
0 - n_2
0 - n_i
DCH 1
DCH 2
DCH 16
m
CRC input of m_i blocks of size n_i where i is 0 - 16 channels
1
After CRC 0-24 bits attached
0 - n_1 0-24
0 - n_2 0-24
0 - n_i 0-24

Figure 7 Transport blocks are mapped on *0-16* channel streams. A CRC checksum of length *0 - 24* bits is attached to each block

6.2 Transport block concatenation and code block segmentation

Transport blocks mapped within a transport channel during the same TTI are concatenated to larger code blocks. If the resulting code block is larger than a maximum code block size, it will be segmented into several code blocks. The purpose with code block concatenation and segmentation is to build code blocks with a size that yields better coding performance of the channel [2]. Small blocks are concatenated to form blocks that yield lower coding overhead, and code blocks that are too large are segmented into smaller blocks to reduce coding complexities. The maximum code block size is dependent of the type of forward error coding (FEC) function that will be used in the following channel coding operation.

Transport block Concatenation

A transport block set is a finite bit sequence $b_{im1}, b_{im2}, b_{im3}, \ldots, b_{imBi}$, where i is the transport channel, m is the transport block number and B_i is the number of bits in each transport block, including the previously appended CRC code, see Figure 8. The number of transport blocks on channel i is denoted M_i and the bit sequence after block concatenation is $x_{i1}, x_{i2}, x_{i3}, \ldots, x_{iXi}$, where $X_i = M_iB_i$.

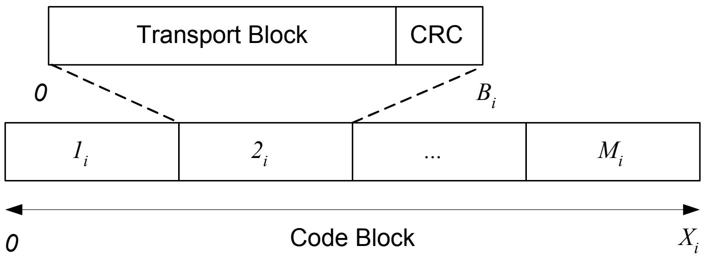


Figure 8 Transport blocks are concatenated to code blocks

Code block segmentation

The maximum size of a code block for channel coding is denoted Z. For turbo coded channels Z = 5114 and for convolution coded channels Z = 504. Segmentation is performed only if X_i exceeds Z. If turbo coding is to be applied and the size of the block is less than 40 bits (X_i is < 40), the initial part of the code block must be padded with zeros so that the code block length equals 40 bits. All code blocks that are segmented are segmented to equal size. The number of code blocks after segmentation is denoted C_i, where $C_i = \lceil X_i / Z \rceil$. The input X_i must be padded with zeros, in the beginning of the sequence, if X_i is not a multiple of Z.

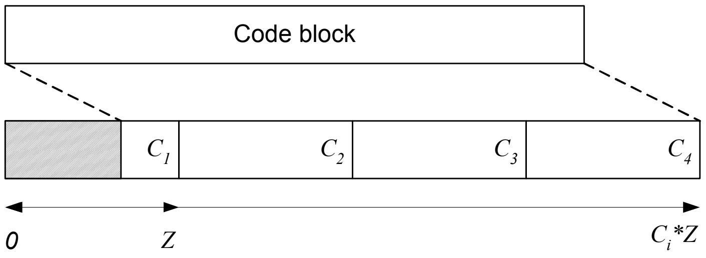


Figure 9 First code block padded with zeros if the input length is not a multiple of Z

AMR service

Block concatenation is not required for AMR transport channels; only one transport block is mapped on each channel, see Figure 10. Convolution coding is applied for all channels and, since each transport block is less than 504 bits, no segmentation is required either. Thus, no concatenation or segmentation processing is required for the AMR service.

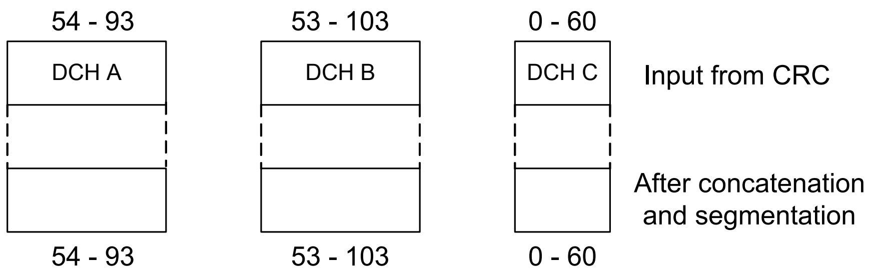


Figure 10 No concatenation or segmentation performed on AMR data

High bit-rate data service

The maximum number of transport blocks that can be transmitted within a TTI for the highest UE capability class is 96. These blocks can be mapped using at most 16 allocated transport channels, see Figure 11 Concatenation and segmentation is performed for up to 16 channels (transport block streams). Like in the previous CRC function, SIMD vectorization can be performed by arrangement of transport blocks within each channel in parallel. Multiple transport channels can be mapped and processed MIMD parallel, assuming that the channels contain transport blocks with different lengths. Thus, less consideration have to be taken of control operations required for alignment of different transport block lengths between channels.

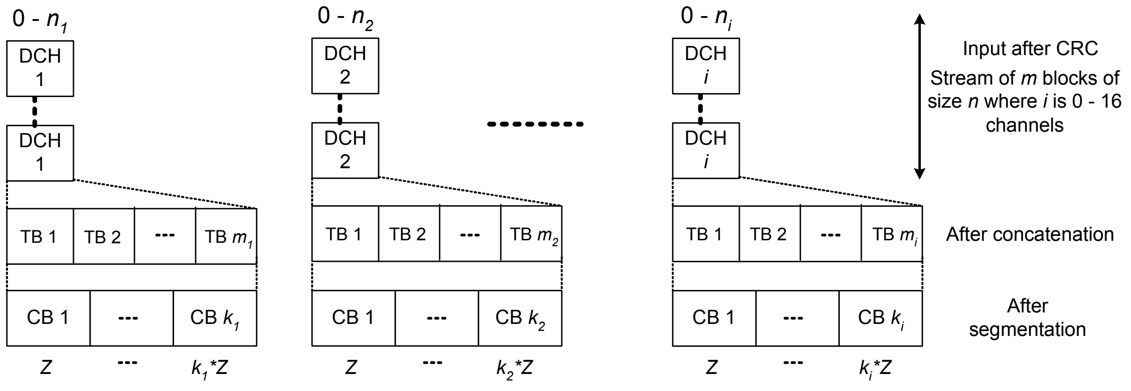


Figure 11 Concatenation and segmentation is performed for up to *16* channels (transport block streams)

6.3 Channel coding

The CRC operation can be used to detect bit errors but it is not possible to identify erroneous bit positions and thus be able to correct them. Forward error correction (FEC) is a coding technique that can be applied in order to, at some extent, restore distorted bits.

Redundancy bits are added to the transmitted bit sequence so that it is possible, to a certain degree, to correct bit errors when decoding the message in the receiver [12].

There are two types of coding techniques that can be used for FEC in UMTS; turbo coding and convolution coding. The coding rate is the main parameter that can be alternated when using these coding techniques. The coding rate corresponds to the bit redundancy ratio after the encoding procedure; i.e. the number of encoded bits inserted per original bit. Thus, with 1/2 rate coding each input bit is associated with two redundant bits and with 1/3 rate coding each bit is associated with three redundant bits.

6.3.1 Convolution coding

Two different coding rates can be applied for convolution coding in UMTS; 1/2 and 1/3 ratio. The coding procedure is logically represented by an 8-stage shift register and a sequence of XOR operations. The 1/2 rate encoder is shown in Figure 12, and the 1/3 rate encoder is shown in Figure 13. When applying 1/2 rate coding, two redundant bits are produced for each bit fed into the register. When coding with 1/3 rate, three redundant bits are produced instead of two. The bits in the shift register must be set to zeros in all positions before the encoding of a new sequence is started.

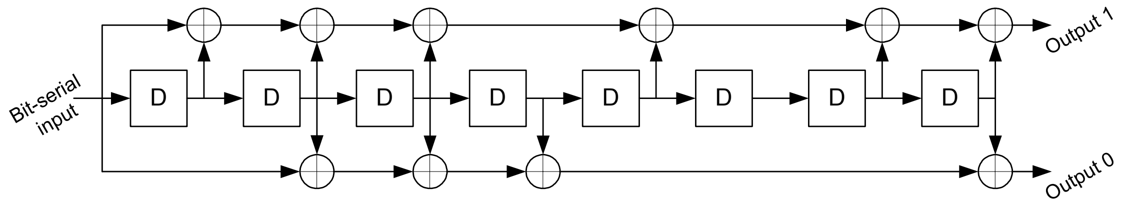


Figure 12 Convolution coding with 1/2 coding rate

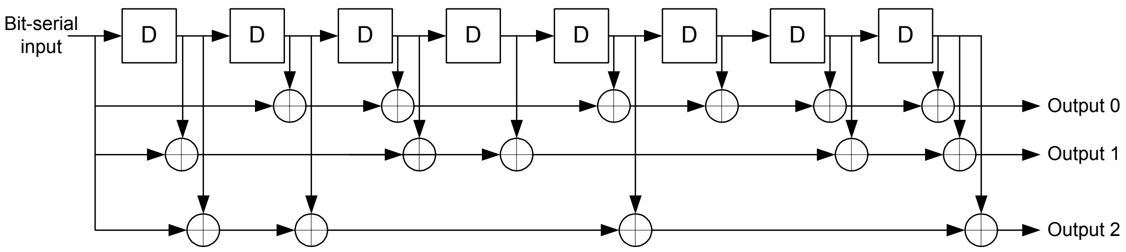


Figure 13 Convolution coding with 1/3 coding rate

The output sequence from the 1/2 rate encoder is: output 0, output 1, output 0, output 1, …

The output sequence from the 1/3 rate encoder is: output 0, output 1, output 2, output 0, output 1, output 2, …

The shift register is one byte wide and each output bit is dependent on the input bit and specific bit positions in the register. The encoding procedure ends when the last bit has been passed through the register. When using 1/3 rate, the length of the encoded sequence will be $3 \times X_i + 24$ bits, and for 1/2 it will be $2 \times X_i + 16$ bits, where X_i is the length of the bit sequence before encoding. The 24 bits correspond to 3 bits times the length of the shift register for the 1/3 rate coding, and the 16 bits correspond to 2 times the length of the shift register for the 1/2 rate coding.

6.3.2 Turbo coding

Turbo coding is performed using two parallel, concatenated convolution coders (PCCC), see Figure 14. The input sequence is interleaved before the second encoder encodes it. The purpose with the interleaving procedure is to spread the coding dependencies over longer bit sequences rather than using only adjacent bits, as is the case with a single convolution coder.

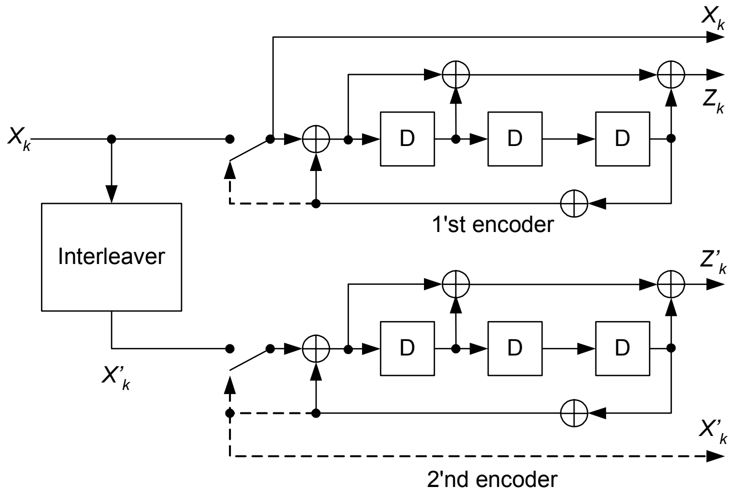


Figure 14 Turbo encoder for UMTS

The output sequence from the encoder is the sequence described in Eq 1.

Eq 1 $X_1, Z'_1, Z_1, X_2, Z'_2, Z_2, X_3, Z'_3, Z_3, ..., X_n, Z'_n, Z_n$

6.3.4 Trellis termination

The turbo-coded output is ended with a termination sequence. The termination sequence constitutes the sequence that is produced when shifting out the bits present in the shift register, after closing the feedback loop, as can be seen in Figure 14. This procedure is called Trellis termination and the termination sequence, described by Eq 2, is appended to the encoded bit sequence, described by Eq 1.

Eq 2 $X_1, Z_1, X_2, Z_2, X_3, Z_3, X'_1, Z'_1, X'_2, Z'_2, X'_3, Z'_3$

6.3.5 The Turbo Code interleaver

The input bit sequence is arranged according to a matrix pattern before the interleaving procedure. The bit matrix is interleaved through an intra-row bit permutation and then by an inter-row permutation. The length of the input sequence determines the size of the matrix. A logical algorithm to perform the interleaving procedure and the arrangement of the matrix size is described in [4].

AMR service

The three transport channels used for the AMR service require convolution encoding, where the class A bit stream is encoded using 1/3 rate and class B and C bit streams are encoded using 1/2 encoder rate. The relation between the input data streams and the output after convolution coding is shown in Figure 15. SIMD vectorization is complicated by the fact that the lengths of the input streams are different for each channel and channels are encoded using different parameters. A possibility is to, like in the CRC function, SIMD vectorize according to groups of class A, class B or class C streams from different composite user channels. Thus, it is possible to take advantage of parameter and stream length equality. Another possibility is to map and process the transport channels simultaneously in MIMD parallel fashion.

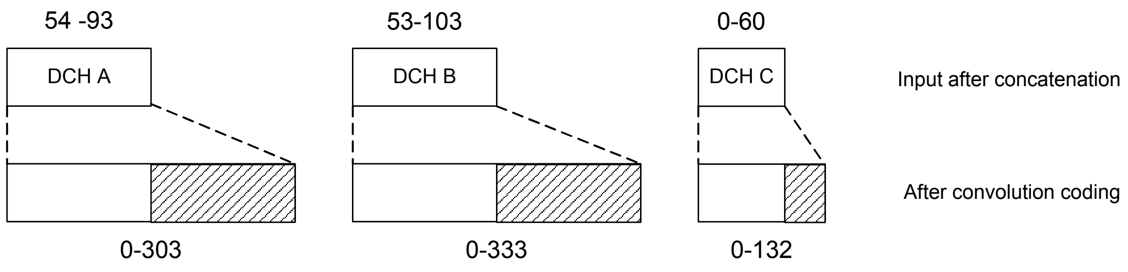


Figure 15 Convolution coding for AMR channels

High bit-rate data service

In the high bit-rate service example, at most 16 parallel input streams, each carrying m_i code blocks of size n_i must be processed. The input and output relations for convolution and turbo coded streams are shown in Figure 16. Like in the previous functions, SIMD vectorization can be performed by grouping code blocks within a single transport channel since there are no data dependencies between blocks. Like in the AMR example, assuming that code block lengths and processing parameters can be unequal for the transport channels complicates SIMD vectorization through grouping of code blocks from several transport channels. A possibility is to process multiple in channels in MIMD parallel fashion.

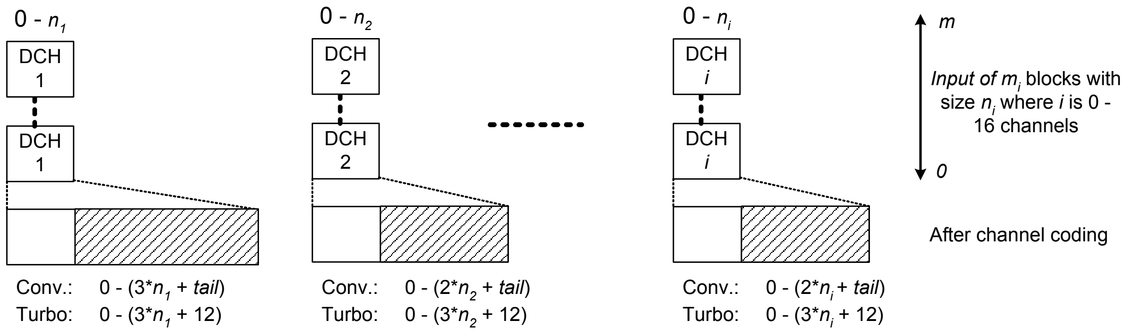

0 - n_1
0 - n_2
0 - n_i
DCH 1
DCH 1
DCH 2
DCH 2
DCH i
DCH i
m
Input of m_i blocks with size n_i where i is 0 - 16 channels
0
After channel coding
Conv.: 0 - (3*n_1 + *tail*)
Turbo: 0 - (3*n_1 + 12)
Conv.: 0 - (2*n_2 + *tail*)
Turbo: 0 - (3*n_2 + 12)
Conv.: 0 - (2*n_i + *tail*)
Turbo: 0 - (3*n_i + 12)

Figure 16 Output after convolution or turbo coding. The *tail* variable takes the value *16* or *24* depending on the coding rate

6.4 Rate matching

Rate matching is performed to match the input data length with the available radio frame bits. The rate match pattern has to be calculated with regard to the input rates of all the channels, which are to be multiplexed to form a user CCTRCH. Bits are either repeated or punctured according to this rate-match pattern.

Rate matching is performed differently depending on whether a channel is transmitted using compressed or normal mode, and whether fixed or flexible positions are used (in this study only rate-matching for fixed positions are considered). Compressed mode is a technique that can be applied to perform channel measurements in UEs. Channel compression is used to create transmission gaps (transmission free slots), which can be used by the UE to perform measurements [13]. There are different compression techniques for the downlink transmissions; spread-factor reduction, bit puncturing and higher-layer scheduling. With spread-factor reduction, the spread-factor is reduced so that the number of chips per symbol is decreased. In compressed mode by puncturing, bits are either punctured or repeated. Compression can also be handled through higher-layer scheduling; only a sub-set of the possible transport format combinations can be transmitted during the compressed interval [4]. That is, in order to make room for gaps the input data rate is decreased through scheduling. In all compressed modes, a rate-matching attribute that is used to calculate the number of bits that shall be repeated or punctured, is assigned by higher layers.

6.4.1 Bit separation and bit collection

In Turbo coded channels, only the parity bits and some bits from the trellis termination sequence are rate matched, as shown in Figure 17. This means that no puncturing or repetition is performed on the systematic bits plus some of the trellis bits (see [4] and **Eq 6** in the previous channel coding section). These bits are passed through without any rate matching. For convolution-encoded channels all bits are rate matched, as seen in Figure 18.

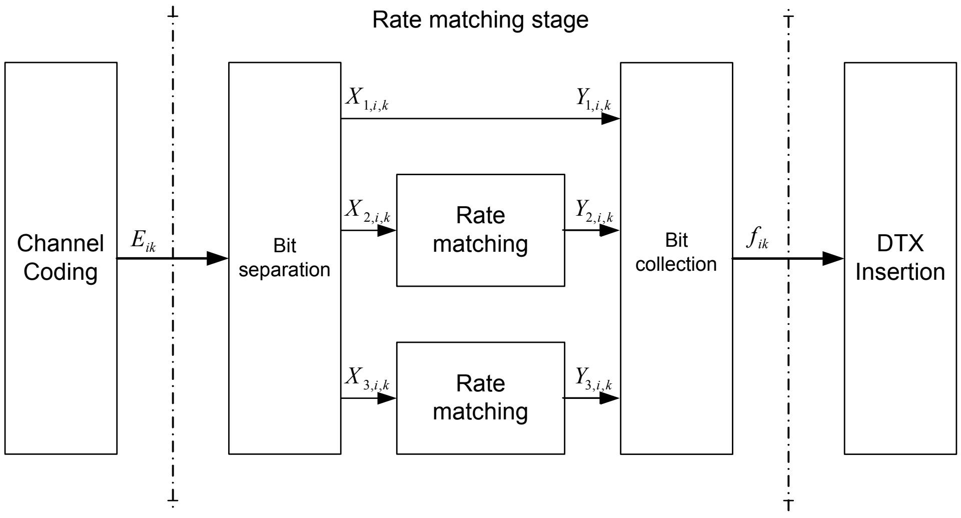


Figure 17 Rate matching for turbo coded channels

The variable E_i describes the number of input bits to the rate-matching function after channel coding. The relations between the number of input and output bits for turbo-coded channels are

$$X_{1,i,k} = C_{i,3(k-1)+1} \qquad k = 1,2,3,\ldots,X_i \qquad X_i = E_i / 3$$
$$X_{2,i,k} = C_{i,3(k-1)+2} \qquad k = 1,2,3,\ldots,X_i \qquad X_i = E_i / 3$$
$$X_{3,i,k} = C_{i,3(k-1)+3} \qquad k = 1,2,3,\ldots,X_i \qquad X_i = E_i / 3$$

For convolution coded channels and turbo coded channels using repetition the relation is

$$X_{1,i,k} = C_{i,k} \qquad k = 1,2,3,\ldots,X_i \qquad X_i = E_i$$

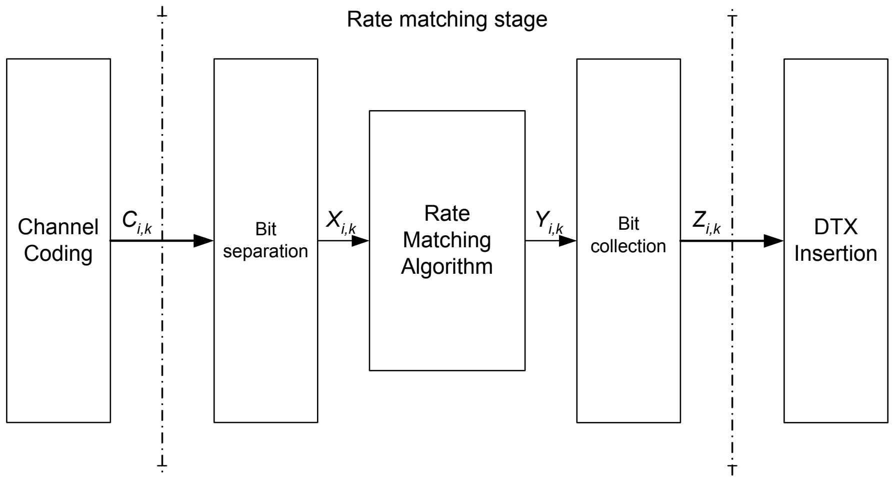


Figure 18 Rate matching for convolution coded channels

The bits are collected into a single output sequence after the rate matching. The bit positions that are punctured during the rate matching are marked with a third value, δ, where $\delta \notin \{0,1\}$. The δ bits can be removed directly after bit collection. For turbo encoded channels with puncturing the output sequence is

$$Z_{1,3(k-1)+1} = Y_{1,i,k} \qquad k = 1,2,3,\ldots,Y_i$$
$$Z_{1,3(k-1)+2} = Y_{2,i,k} \qquad k = 1,2,3,\ldots,Y_i$$
$$Z_{1,3(k-1)+3} = Y_{3,i,k} \qquad k = 1,2,3,\ldots,Y_i$$

For convolution encoded channels and turbo encoded channels using repetition the relation is

$$Z_{1,k} = Y_{1,i,k} \qquad k = 1,2,3,\ldots,Y_i$$

6.4.2 Calculation of rate matching parameters for fixed positions

The rate matching parameters are used to compute puncture and repetition patterns. The pattern is computed as described by an algorithm, presented in sub-section 4.2.7.5 in [4], where also the equations required for calculation of intermediate variables (**Eq 3** to **Eq 6** shown below) can be found. The number of available data bits within a CCTRCH is denoted $N_{data,j}$, where *j* is the transport format combination used. This variable is dependant on the number of physical channels and the spread factor used for the specific user CCTRCH. The number of physical channels is denoted by *P*. The total number of bits available for a user CCTRCH can be calculated as $N_{data,*} = P \times 15 \times (N_{data1} + N_{data2})$, where N_{data1} and N_{data1} depend on the slot format used. These slot formats are specified in Table 11 in [14].

Eq 3 $N_{i,*} = \frac{1}{F_i} \times \left(\max_{l \in TFS(i)} N_{i,l}^{TTI} \right)$

Eq 4 $Z_{i,j} = \left\lfloor \frac{\left(\left(\sum_{m=1}^{i} RM_m \times N_{m,j} \right) \times N_{data,j} \right)}{\sum_{m=1}^{i} RM_m \times N_{m,j}} \right\rfloor \quad \forall i = 1 \cdots I$

Eq 5 $\Delta N_{i,j} = Z_{i,j} - Z_{i-1,j} - N_{i,j} \quad \forall i = 1 \cdots I$

Eq 6 $\Delta N_{i,\max} = F_i \times \Delta N_{i,*}$

6.4.3 Parameters for normal mode and compressed mode by spread factor reduction

The number of bits to be punctured or repeated per TTI is denoted $\Delta N_{i,l}^{TTI}$, where *i* is the transport channel and *l* the transport format. A negative value of $\Delta N_{i,l}^{TTI}$ corresponds to the number of bits to be punctured and a positive number corresponds to the number of bits to be repeated. The output data rate is equal to the input data rate if $\Delta N_{i,\max} = 0$ for TRCH *i*. Thus, the rate-matching algorithm, as specified in [4], shall not be executed. This means that the number of bits for all transport channels to be repeated or punctured is 0, as described by **Eq 6**.

Eq 6 $\Delta N_{i,l}^{TTI} = 0 \quad \forall l \in TFS(i)$.

The rate-matching algorithm shall be executed if $\Delta N_{i,\max} \neq 0$, after calculation of the variables e_{ini}, e_{plus} and e_{minus}.

Parameters for compressed mode by puncturing

The variables $N_{i,*}$ and $\Delta N_{i,*}$ are calculated in the same way as in the compressed mode by spread factor reduction, i.e. using **Eq 3** to **Eq 5**.

Eq 7 $\Delta N_{i,\max}^{m} = F_i \times \Delta N_{i,*}$

The number of bits to be removed on TRCH *i* (to create the gap for compressed mode and to compensate for reduction of bits in the slot format compared to normal slot format), is denoted $Np^{n}_{i,\max}$. The value of $Np^{n}_{i,\max}$ is calculated for each radio frame *n* of the TTI and $Np^{n}_{i,\max} = (Z_i - Z_{i-1})$ *for i=1 to I*

Eq 8 $Z_{i,j} = \left\lfloor \dfrac{\left(\left(\sum_{m=1}^{i} RM_m \times N_{m,j}\right) \times \left(N_{TGL}[N] + \left(N_{data,*} - N'_{data,*}\right)\right)\right)}{\sum_{m=1}^{i} RM_m \times N_{m,j}} \right\rfloor$

The variable N_{TGL} is the number of bits in each radio frame corresponding to the gap. The number of empty slots is denoted *TGL* and the first empty slot in the gap is denoted N_{first}.

$$N_{TGL} = \begin{cases} \dfrac{TGL}{15} \times N'_{data,*} & \text{if } N_{first} + TGL \le 15 \\ \dfrac{15 - N_{first}}{15} \times N'_{data,*} & \text{in first radio frame of the gap if } N_{first} + TGL > 15 \\ \dfrac{TGL - (15 - N_{first})}{15} \times N'_{data,*} & \text{in following radio frame if } N_{first} + TGL > 15 \end{cases}$$

The total number of bits in the gaps when applying compressed mode is calculated as

$$Np^{TTI,m}_{i,\max} = \sum_{n=m\times F_i}^{n=(m+1)\times F_{i-1}} Np^{n}_{i,\max}$$

AMR service

Input and output data flows for rate matching of the AMR service, using the highest AMR data rate, are illustrated in Figure 19. The rate matching function is one of the more problematic functions in the baseband because the data streams must be manipulated on single bit level (puncturing and repetition of bits) according to the channel specific rate matching pattern. There is no obvious way to vectorize the matching computation since the bit stream lengths and rate match patterns are different for each transport channel. More obvious is that each transport channel stream can be calculated in MIMD fashion.

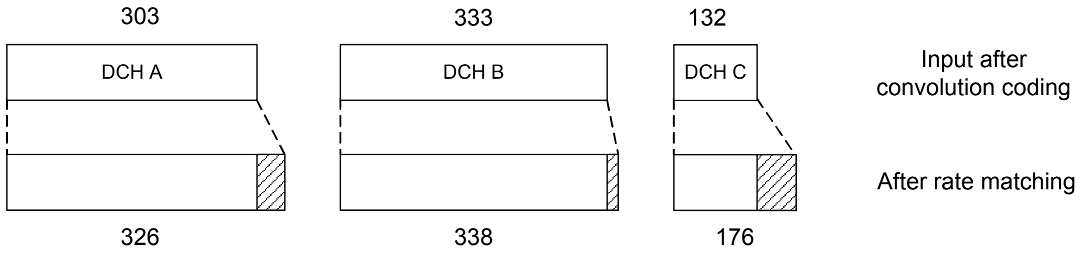


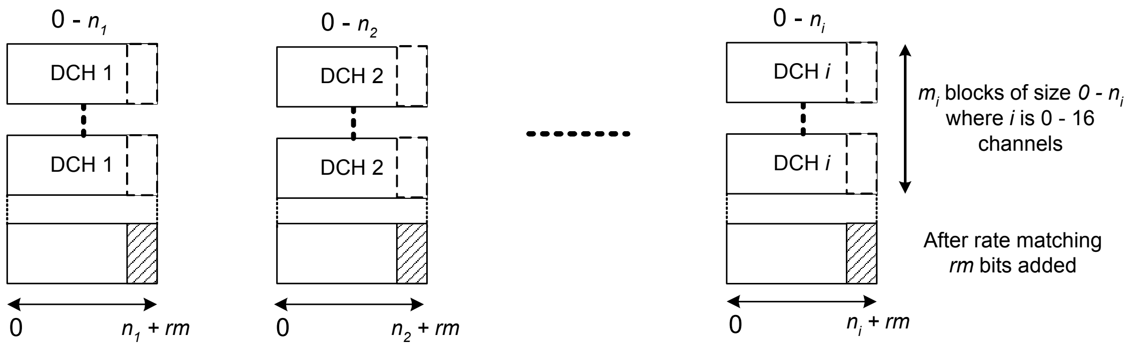
Figure 19 AMR output after rate matching

The following fixed parameters can be used for implementation of the rate matching function for AMR

- Static rate matching parameter for 180 DCH A, 170 for DCH B and 215 for DCH C
- Compression parameters used for punctured normal/punctured channels
    - Normal mode, bits per CCTRCH is $N_{data,*} = 1 \times 15 \times (6 + 22)$
    - Compressed mode, bits per CCTRCH is $N_{data,*} = 1 \times 15 \times (12 + 40)$
- Compression parameters for normal/spread factor reduction channels
    - Normal mode, bits per CCTRCH is $N_{data,*} = 1 \times 15 \times (6 + 22)$
    - Compressed mode, bits per CCTRCH is $N_{data,*} = 1 \times 15 \times (12 + 44)$

High bit-rate data service

Like with the AMR data streams, there is no obvious way to vectorize computation of the data streams but the rate matching function be mapped in a MIMD parallel fashion. A bit-rate of 2048 kbps requires a channel spread factor of length 4. Compressed mode by spread factor reduction is not supported for spread factors of length 4, which means that only normal mode, higher layer scheduling and compressed mode by puncturing can be applied. The input constitutes 0 to 16 parallel channel streams, each with M_s code blocks of length $N_{\max}$, where $M_s \times N_{\max} \le N_{data,*}$. ($N_{data,*}$ can maximally be 248 + 992 when using SF of length 4).



0 - n_1
DCH 1
DCH 1
0 - n_2
DCH 2
DCH 2
0 - n_i
DCH i
DCH i
m_i blocks of size $0 - n_i$ where i is 0 - 16 channels
After rate matching rm bits added
0
$n_1 + rm$
0
$n_2 + rm$
0
$n_i + rm$

Figure 20 Rate matching on data streams

6.5 First insertion of discontinuous transmission (DTX) indication bits

Unused bits in the radio frames are filled with DTX bits. The DTX bits are never transmitted over the air, they are used just to mark when transmission should be turned off. This first DTX insertion is performed only if fixed channel positions in the radio frames are used, see Figure 21. With fixed positions it means that a fixed number of bits are reserved for each transport channel in the radio frames. If bits were punctured in the rate-matching, the space required for the punctured bits should be reserved for later insertion of p bits.

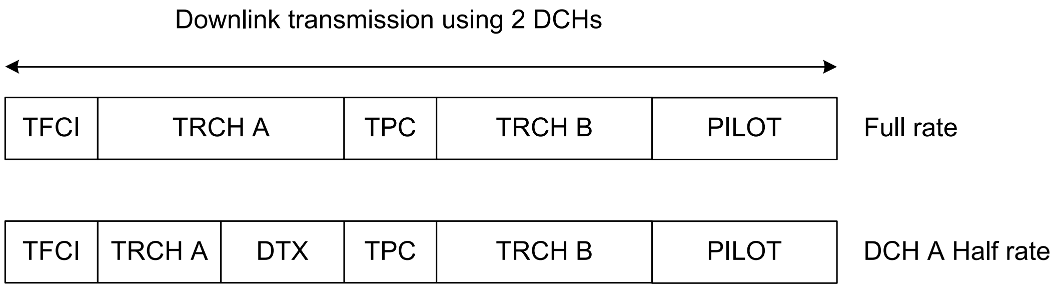


Figure 21 Fixed channel slots with variable rate and insertion of DTX bits

For radio frames in which flexible positions are used, the DTX insertion is performed after the multiplexing of the transport channels. The decision of using fixed or flexible positions is controlled from higher layers in the UTRAN. The bit output after the DTX insertion is three-valued {0, 1, δ}.

The bits after rate matching are denoted $g_{i1}, g_{i2}, g_{i3}, \dots, g_{iG_i}$, where G_i is the number of bits in one TTI of Trch *i*. The number of bits available for a radio frame in Trch *i* is denoted H_i and the number of radio frames per TTI for Trch *i* is F_i. The total number of bits after DTX insertion is denoted D_i. If no compression or compressed mode by spread factor reduction is performed, then $D_i = F_i \times H_i$. If compressed mode by puncturing is used, space for the bits that were punctured in the rate matching function should be reserved. This space will be used for insertion of *p* bits in the proceeding interleave function. For compressed mode by puncturing, DTX bits should be inserted until $D_i = F_i \times H_i - Np_{i,\max}^{TTI,m}$ and $H_i = N_{i,*} + \Delta N_{i,*}$. The variables $N_{i,*}$, $\Delta N_{i,*}$ and $Np_{i,\max}^{TTI,m}$ were defined for the rate matching function. The bits after DTX insertion are denoted $h_{i1}, h_{i2}, h_{i3}, \dots, h_{iD_i}$. The input and output relation after DTX insertion is:

$$h_{ik} = g_{ik} \text{ for } k = 1, 2, 3, \dots, G_i$$
$$h_{ik} = \delta \text{ for } k = G_i + 1, G_i + 2, G_i + 3, \dots, D_i$$

AMR service

In this specific case for the AMR service it is assumed that processing is performed in normal mode. Bit repetition is performed in the previous rate matching function to fill up the radio frames. Therefore no DTX insertion is required in this case. The output after DTX insertion are represented by two bits, thus, the output data size is doubled. Each bit after DTX insertion is three-valued and therefore each symbol is represented by two bits.

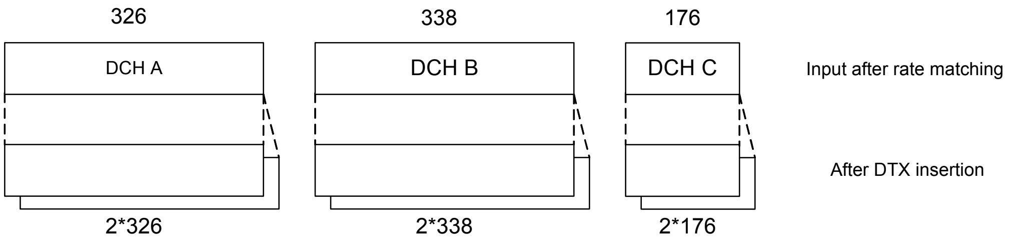


Figure 22 Output after DTX insertion. Each symbol is three-valued {0, 1, δ} and represented using two bits

High bit-rate data service

The inputs for DTX insertion comprise 0 to 16 rate matched channel streams, each of length 0 to $ni_{\max}$, as can be seen in Figure 23. The channel number is denoted by i and $ni_{\max}$ is the maximal length that can be processed within a transport channel during a TTI. The DTX insertion function should be performed if compressed mode by puncturing is used. DTX bits are inserted on the last positions in the data bit sequence. Thus, DTX bit insertion in the data streams can to some extent be SIMD vectorized. Some operations are likely required to handle different start indexes, because of different channel data lengths and space reserved for *p* bits. Like earlier stages, the channels can also be processed in MIMD fashion since there are no dependencies between the channels.

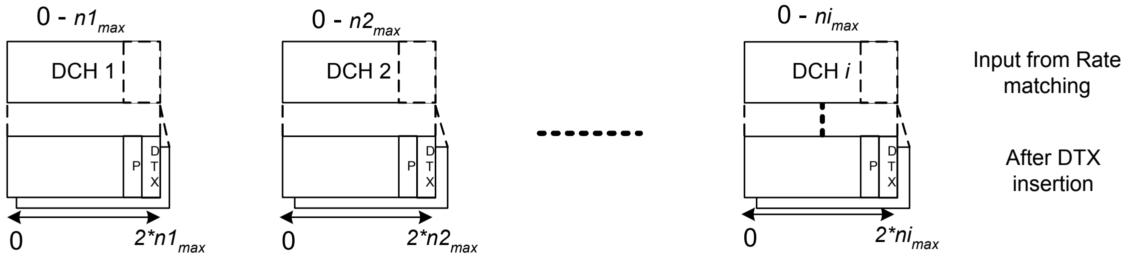


Figure 23 Output after DTX insertion. Each three valued symbol {0, 1, δ} is represented using two bits

6.6 First Interleaving

To reduce the effect of bit error bursts the bits are interleaved before transmission. After de-interleaving in the receiver, possible error bursts will be distributed throughout the frame. If compressed mode is applied, a fourth symbol value, *p,* is inserted at this stage to mark any bits used for creation of transmission gaps (if bits were punctured in the rate matching function). The *p*-bits are always inserted at the first bit positions in the radio frames. The bits after the interleaving procedure can take any of the four values $\{0, 1, \delta, p\}$.

6.6.1 Insertion of transmission gap bits

This *p*-bit insertion should only be applied to radio frames that are compressed by puncturing. No *p*-bit insertion is performed for other frames. The punctured input sequence to the block interleaver is $X_i = Z_i + Np$ where Z_i is the input data sequence after the previous DTX function and Np is the number of bits to be punctured.

6.6.2 Column-wise block interleaving

The block interleaving is performed only when TTI:s span over 20, 40 or 80 ms. The input bits to the block interleaver, X_i, are arranged in a matrix of size $C_1 \times R_1$, where C_1 is the number of bit columns. The number of columns is decided by the TTI as described in

Table 2 below. The variable R_1 is the number of bit rows in the matrix. At this stage, the input bit length is always a multiple of the physical radio frame size, and R_1 can be calculated as $R_1 = \frac{X_i}{C_i}$. The input bits are arranged row wise in sequential order as shown in Figure 24.

$$\begin{bmatrix} X_{i,1} & X_{i,2} & X_{i,3} & \cdots & X_{i,C1} \\ X_{i,(C1+1)} & X_{i,(C1+2)} & X_{i,(C1+2)} & & X_{i,(2\times C1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ X_{i,((R1-1)\times C1+1)} & X_{i,((R1-1)\times C1+2)} & X_{i,((R1-1)\times C1+3)} & X_{i,((R1-1)\times C1+4)} & X_{i,(R1\times C1)} \end{bmatrix}$$

Figure 24 Bit input arrangement before interleaving

The bit matrix shall be interleaved column-wise. The permutation pattern that shall be used for interleaving is described in

Table 2. The number sequence in the table corresponds to the original positions in the input matrix from where the bits should be read. As can be seen in the table, no interleaving is performed during TTIs of 10 ms. When the columns have been interleaved the output bits should be read column-wise in sequential order, starting with the leftmost column.

Table 2 Number of columns and permutation pattern for interleaving

TTI	Number of columns (C1)	Column-wise permutation pattern $\langle P1_{C1}(0), P1_{C1}(1), \ldots, P1_{C1}(C1-1)\rangle$
10 ms	1	$\langle 0\rangle$
20 ms	2	$\langle 0,1\rangle$
40 ms	4	$\langle 0,2,1,3\rangle$
80 ms	8	$\langle 0,4,2,6,1,5,3,7\rangle$

AMR service

The interleaving function is performed for all three channel streams that are used to map the AMR data. The TTI is 20 ms and therefore, according to Table 2, the data must be arranged in a two-column bit matrix. The output after interleaving will be four-valued symbols $\in \{0, 1, \delta, p\}$, which require two bits to represent each symbol, as illustrated in Figure 25. The column interleaving pattern is the same for three used transport channels and it is possible to SIMD vectorize parts of the interleaving function. Alternatively, the streams can be processed in MIMD fashion like the earlier functions.

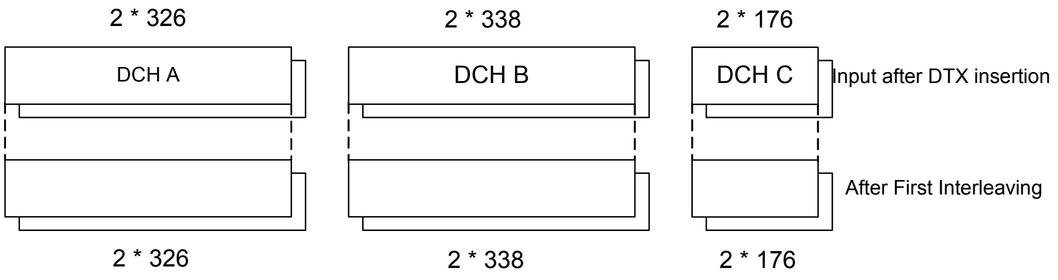


Figure 25 After first interleaving. Each four-valued symbol {0, 1, δ, *p*} is encoded using two bits

High bit-rate data servcie

The interleaving function is performed on 0 to 16 channel streams, each of length 0 to ni. The output stream constitute four-valued symbols ∈ {0, 1, δ, *p*}, which require two bits to represent each symbol as can be seen in Figure 26. Like for the AMR service, it is possible to SIMD vectorize parts of the processing since the interleaving pattern is the same for all transport channels. The transport channels can also be processed in MIMD fashion.

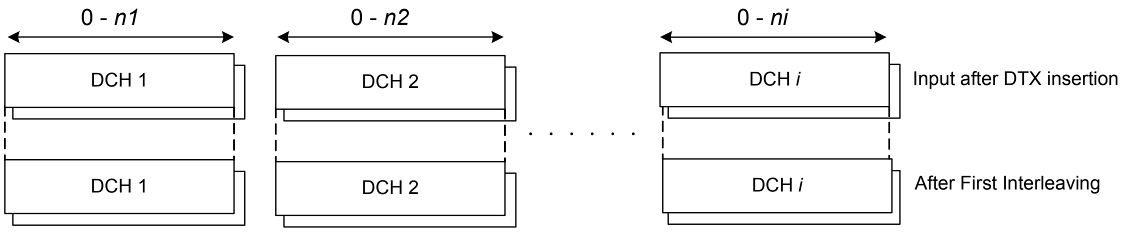


Figure 26 Output after first interleaving. Each four-valued symbol {0, 1, δ, *p*} is represented by two bits

6.7 Radio frame segmentation

The bit streams must be sliced in segments when TTIs longer than 10 ms are used. F_i denotes the required number of consecutive radio frames. Data for one radio frame is sliced in segments each 10 ms interval in order to fill one radio frame of data after transport channel multiplexing.

The input bit sequence is denoted $x_{i1}, x_{i2}, x_{i3}, \ldots, x_{iX}$, where i is the transport channel and X_i is the number of bits. This input bit sequence is divided into F_i output bit sequences denoted $y_{i,ni,1}, y_{i,ni,2}, y_{i,ni,3}, \ldots, y_{i,\, i}$, where n_i is the radio frame number during the current TTI and $Y_i = (X_i / F_i)$ is the number of bits per segment.

The relation between input and output bits is:

$$y_{i,ni,k} = x_{i,((n_i-1)\times Y_i)+k} \text{ , } n_i = 1 \ldots F_i,\ k = 1 \ldots Y_i$$

AMR service

The AMR input for the frame segmentation comprise 3 channel streams and the TTI is 20 ms. Thus, the streams will be sliced in two segments which are distributed over two radio frames, as shown in Figure 27. The first segment comprises the bits 1 to 163 in DCH A, bits 1 to 169 in DCH B and bits 1 to 88 in DCH C. The second segment comprises the bits 164 to 326 in DCH A, is bits 170 to 338 in DCH B and is bits 89 to 176 in DCH C. The transport channels with a composite user channel (DCH A, DCH B and DCH C) comprise bit streams of different lengths. However, segmentation can be more efficiently SIMD vectorized by grouping channels from several user channels in groups of class A bits channels, class B bits channels and class C bits channels, i.e. DCH A channels from several users are combined and SIMD vectorized. Alternatively, the channels can be processed in MIMD fashion since there are no data dependencies between transport channels in the frame segmentation function.

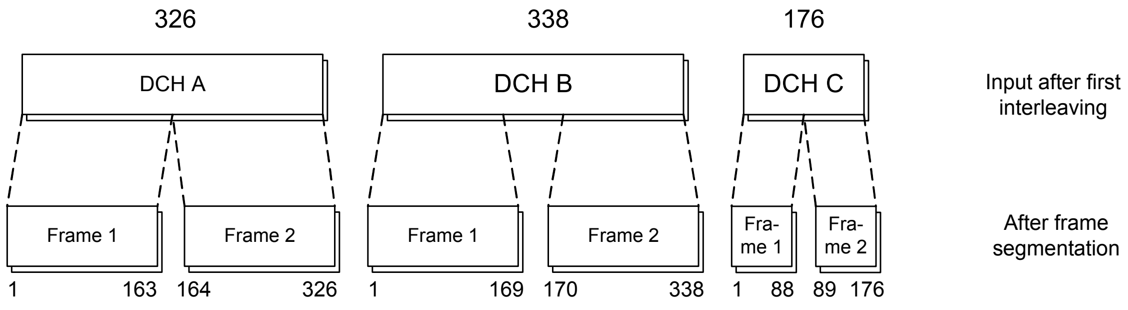


Figure 27 After frame segmentation {0, 1, δ, p}

High bit-rate data service

The input which is mapped on 0 to 16 channel streams for the high bit-rate data service will be sliced in segments dependently on the TTI and the number of physical radio frames used. At maximum, three physical channels can be used for a single user. Each channel data stream will be sliced in *m* frame segments, where *m* is $\frac{TTI}{10}$ (10 ms multiple of the TTI). The frames are sliced in blocks of equal lengths, which makes it possible to SIMD vectorize the slicing operation. Like earlier functions, it is also possible to perform the slicing operation in MIMD fashion since there are no data dependencies between the channels.

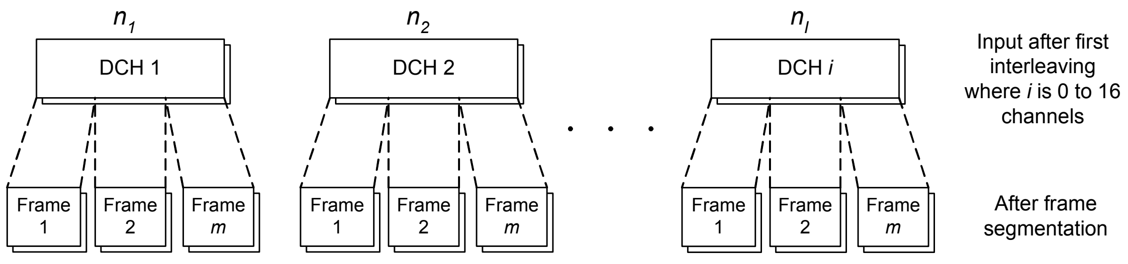


Figure 28 After frame segmentation {0, 1, δ, *p*}

6.8 Transport channel (Trch) multiplexing

Radio frame segments are delivered to the multiplexer function with a 10 ms interval, corresponding to the radio frame transmission frame rate. These frame segments from different transport channels are multiplexed to form a single user CCTRCH stream (only one CCTRCH is possible for one UE). The CCTRCH will later be mapped on one or several physical channels. The transport channels that are to be multiplexed to the same CCTRCH must use the same spreading factor. The input bits are denoted $f_{i1}, f_{i2}, f_{i3}, \ldots f_{iVi}$ where V_i is the number of bits for transport channel i. The number of transport channels is I. The bit output after channel multiplexing is $s_1, s_2, s_3, \ldots s_S$ where S is the number of bits corresponding to the number of physical radio frames used, $S = \sum_i V_i$

AMR service

The multiplexing function constitutes a synchronization stage in the function pipeline. The input constitutes one segment from each of the three used transport channels for the AMR service. These are multiplexed into a single output stream, as shown in Figure 29. The TTI for AMR is 20 ms which means that multiplexing is performed twice during a TTI to fill two consecutive radio frames. If the previous functions were processed in MIMD fashion, the multiplex function will comprise a merge of multiple channel input streams. If the previsous functions were processed by SIMD vectorizarion, the multiplex function will comprise rearrangement and merging of vectorized data, dependently on how the streams were SIMD vectorized.

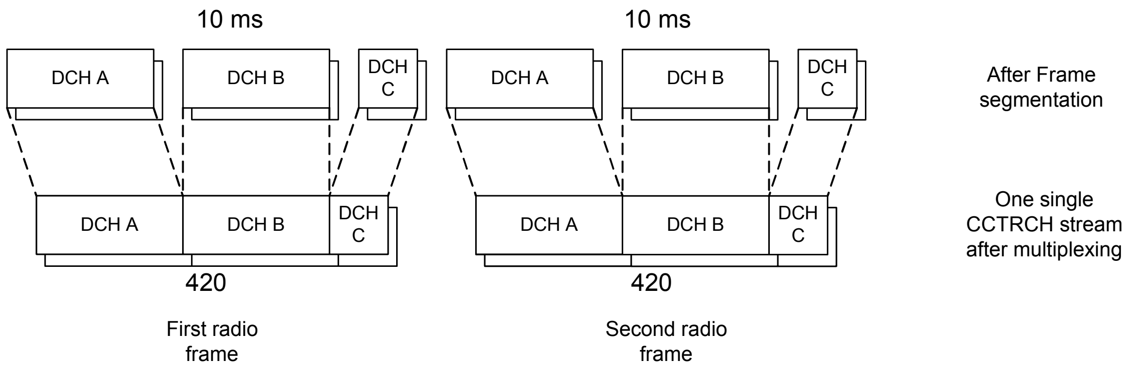


Figure 29 Before and after transport channel multiplexing in AMR processing

High bit-rate data service

The input constitutes I segments, one from each of I channel streams, and which are multiplexed into a single output stream, as shown in Figure 30. Like with the AMR processing, the multiplexing constitutes a synchronization stage in the function flow. If the previous functions were processed by SIMD vectorizarion, the multiplex function will comprise rearrangement and merging of vectorized data, dependently on how the streams were SIMD vectorized. If the previous functions were processed in MIMD fashion, the multiplex function will comprise a merge of multiple channel input streams.

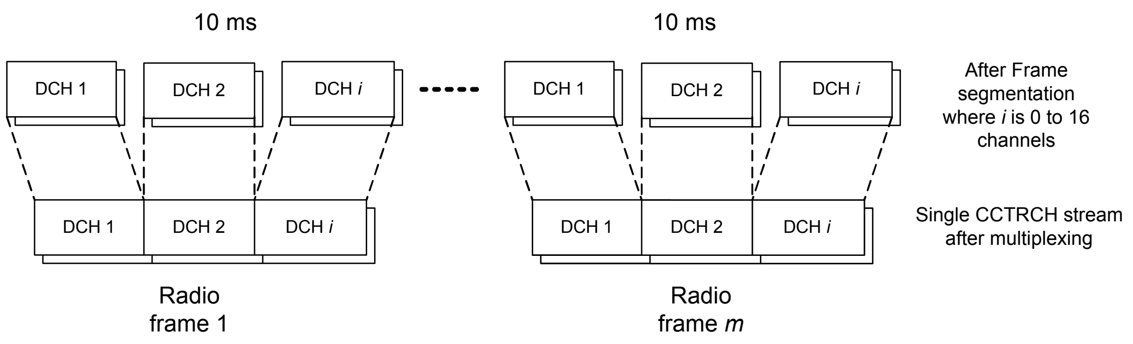

10 ms
10 ms
DCH 1
DCH 2
DCH *i*
After Frame segmentation where *i* is 0 to 16 channels
Single CCTRCH stream after multiplexing
Radio frame 1
Radio frame *m*

Figure 30 Before and after transport channel multiplexing in high bit-rate processing

6.9 Second insertion of discontinuous transmission (DTX) indication bits

The second DTX insertion is not performed on transport channels using fixed positions or compression by puncturing. The DTX bit insertion at this stage is applied for transport channels using flexible positions and the DTX bits will be inserted at the end of the multiplexed frames. The output bits are denoted $w_1, w_2, w_3, ..., w_{PR}$, where P is the number of physical channels used and R is the amount of available bits in the radio frame format used. The input bits are denoted $S_1, S_2, S_3, ..., S_S$, where S is the number of bits after transport channel multiplexing. Thus the number of DTX bits to insert is $(P \times R) - S$. The R variable is calculated differently depending on if frames are non-compressed or compressed by spreading factor reduction or higher layer scheduling.

For non compressed frames, $R = \frac{N_{Data,*}}{P} = 15 \times (N_{Data1} + N_{Data2})$. The variables N_{Data1} and N_{Data2} constitute the number of data bits available for payload in each slot. The possible values are listed in Table 11 in [14].

For compressed frames, $N'_{data,*} = P \times 15 \times (N'_{Data1} + N'_{Data2})$, where N'_{Data1} and N'_{Data2} are specific slot formats used for spreading factor reduction and higher layer scheduling respectively. The possible slot formats can be found in table 11 in [14]. Thus, $N'_{Data,*}$ is the number of data bits plus the bit space required for the transmission gap. The number of available data bits for transport channel multiplexing is $R = \frac{N^{cm}_{data,*}}{P}$. For compression by spread factor reduction, $N^{cm}_{data,*} = \frac{N'_{data,*}}{2}$, and for compression by higher layer scheduling, $N^{cm}_{data,*} = N'_{data,*} - N_{TGL}$. The variable N_{TGL} represents the number of consecutive idle slots used to create the transmission gap. TGL and N_{first} are defined in section 4.4 in [4].

$$N_{TGL} = \begin{cases} \frac{TGL}{15} \times N'_{data,*}\ , if\ N_{first} + TGL \le 15 \\ \frac{15 - N_{first}}{15} \times N'_{data,*}\ , in\ first\ frame\ if\ N_{first} + TGL > 15 \\ \frac{TGL - (15 - N_{first})}{15} \times N'_{data,*}\ , in\ \sec ond\ frame\ if\ N_{first} + TGL > 15 \end{cases}$$

The relation between the input bits and output bits is:

$$w_k = s_k \quad k = 1,2,3,\ldots,S$$
$$w_k = \delta \quad k = S+1, S+2, S+3, \ldots, P \times R$$

DTX indication bits are denoted by $\delta \notin \{0,1,p\}$ and non DTX bits $S_k \in \{0,1,p\}$.

AMR service

The AMR input constitutes a single stream of data segmented into two radio frames, see Figure 31. In the AMR service it was assumed that fixed frame positions are used and therefore no DTX insertion is required at this stage.

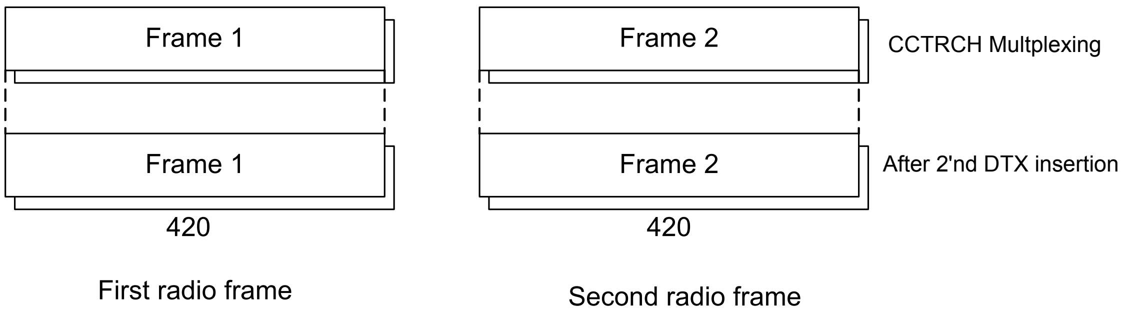


Figure 31 After second DTX insertion

High bit-rate data service

The input constitutes a single stream of data segmented into *n* frames, where *n* is a 10 ms multiple of the TTI, see Figure 32. Fixed frame positions are assumed to be used, so no DTX insertion is required at this stage.

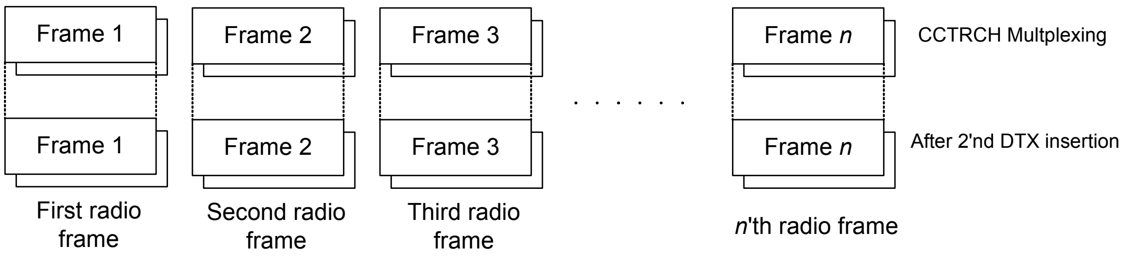


Figure 32 Output after second DTX insertion.

6.10 Physical channel segmentation

When several physical channels are used, the input must be segmented in blocks, as many as the number of physical channels used for the user CCTRCH. If compressed mode by puncturing is used, the *p*-bits inserted earlier, during the interleaving procedure, are removed before they are mapped on physical channels. For all other modes, all bits are mapped onto the physical channels. 1-8 simultaneous DPCH codes can be received simultaneously [4]. The input bits are denoted $x_1, x_2, x_3, \ldots, x_M$, where M is the total number of bits before physical channel segmentation. The bits after channel segmentation are denoted U and the number of physical channels P. For all modes, except compressed mode by puncturing, the relation is $U = \frac{X}{P}$. For compressed mode by puncturing, the relation is $U = \left(X - N_{TGL} - \left(N_{data,*} - N'_{data,*}\right)\right)/P$. The variables are explained above in subsection 6.9.

AMR service

Only one physical channel is required for the AMR service and therefore no physical channel segmentation is required. Thus, the channel stream is unmodified at this stage, as shown in Figure 33.

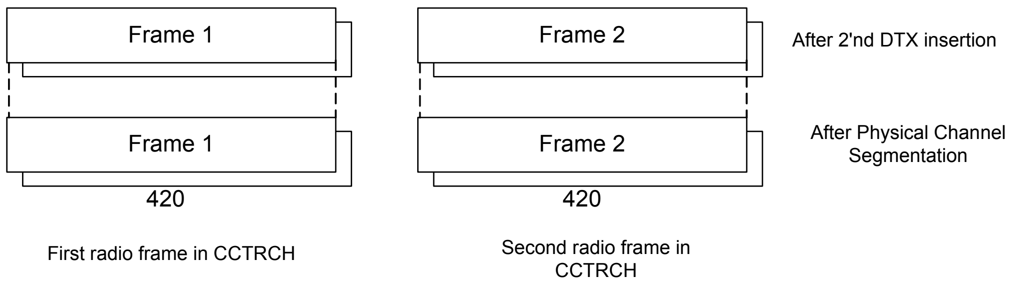

Frame 1
Frame 2
After 2'nd DTX insertion
Frame 1
Frame 2
After Physical Channel Segmentation
420
420
First radio frame in CCTRCH
Second radio frame in CCTRCH

Figure 33 After physical channel segmentation

High bit-rate data service

The maximum number of physical channels that can be used for the studied UE class is limited to three. See Figure 34 for an illustration of the physical channel segmentation.

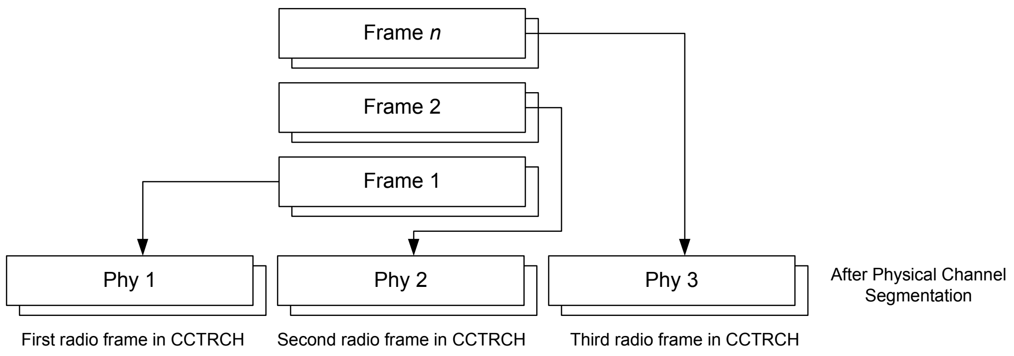

Frame *n*
Frame 2
Frame 1
Phy 1
Phy 2
Phy 3
After Physical Channel Segmentation
First radio frame in CCTRCH
Second radio frame in CCTRCH
Third radio frame in CCTRCH

Figure 34 Frames are segmented on separate on maximum 3 physical channels

The segments can either be distributed in MIMD fashion, into separate data streams for each channel, or kept in a single stream in order to process the data using SIMD vectorization in the following functions.

6.11 Second Interleaving

Before radio frames are mapped onto physical channels, a second interleaving procedure is performed. This interleaving is performed for each physical channel, if multiple physical channels are used. The bits in the radio frames are arranged in a bit matrix. The number of columns in the matrix, denoted C2, is always constant. Thus, the number of rows in the matrix is determined by the input length. If the number of input bits is not equal to the minimum possible matrix size, bits with zeros or ones are padded to the bit positions which correspond to the end of the frame. These pad bits will be removed after the interleaving. The interleaving is then performed by swapping whole columns in the matrix according to a constant interleaving pattern. After the interleaving, the bits are read in a transposed matrix order (i.e. column-wise instead of row-wise).

Interleaving Algorithm

I. The number of columns are numbered 0, 1, 2, …, *C2*-1, where *C2* always is 30 and the order is from left to right.

II. The number of rows in the bit matrix, denoted *R2*, is determined by finding the minimum *R2* such that $U \le C2 \times R2$, where *U* is the number of input bits on the physical channel *p*. The rows are numbered 0, 1, 2, …, *R2*-1, from the top to the bottom.

III. The input bit sequence $u_{p,1}$, $u_{p,2}$, $u_{p,3}$, …, $u_{p,U}$ is written in sequential order, row-wise into the bit matrix starting at column 0, row 0:

$$\begin{bmatrix} y_{p,1} & y_{p,2} & y_{p,3} & \cdots y_{p,C2} \\ y_{p,(C2+1)} & y_{p,(C2+2)} & y_{p,(C2+3)} & \cdots y_{p,(2\times C2)} \\ \vdots & \vdots & \vdots & \cdots \vdots \\ y_{p,((R2-1)\times C2+1)} & y_{p,((R2-1)\times C2+2)} & y_{p,((R2-1)\times C2+3)} & \cdots y_{p,(R2\times C2)} \end{bmatrix}$$

IV. The bit matrix shall be permuted according to the pattern $\langle P2(j)\rangle_{j\in\{0,1,\ldots,C2-1\}}$ where *j* is the original column position. The permutation pattern is shown in Table 3, where $P2(j)$ is the value at the original position *j*, and the column position after interleaving is determined by the corresponding index in the sequence in Table 3 (on row 2).

Table 3 Inter-column permutation pattern for second interleaving

Number of columns (C2)	Permutation pattern $\langle P2(0), P2(1), ..., P2(C2-1) \rangle$
30	$\langle$0,20,10,5,15,25,3,13,23,8,18,28,1,11,21, 6,16,26,4,14,24,19,9,29,12,2,7,22,27,17$\rangle$

V. The output after block interleaving should be read column-wise from the bit matrix. After the interleaving, bits in the $C2 \times R2$ bit matrix are denoted $y'_{p,k}$.

$$\begin{bmatrix} y_{p,1} & y_{p,2} & y_{p,3} & \cdots y_{p,C2} \\ y_{p,(C2+1)} & y_{p,(C2+2)} & y_{p,(C2+3)} & \cdots y_{p,(2\times C2)} \\ \vdots & \vdots & \vdots & \cdots \vdots \\ y_{p,((R2-1)\times C2+1)} & y_{p,((R2-1)\times C2+2)} & y_{p,((R2-1)\times C2+3)} & \cdots y_{p,(R2\times C2)} \end{bmatrix}$$

The pad bits that were added in order to compensate the size if $U < C_2 \times R_2$ should be deleted before the interleaved bits are mapped on the physical channels.

AMR service

The input constitutes one single channel stream of new radio frame data each 10 ms, two radio frames per TTI as shown in Figure 35. It is possible to perform parts of the interleaving procedure in SIMD vectorized fashion, depending on how the input data is arranged and how the output data must be arranged for the following function.

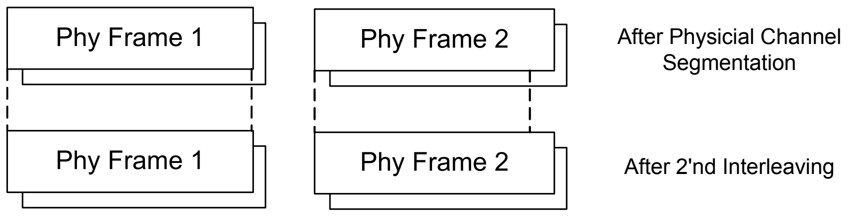


Figure 35 After second interleaving

High bit-rate data service

The input comprises one single channel stream of maximum three radio frames, or three separate streams (one frame mapped on each stream), delivered for each 10 ms interval dependent on how data was arranged after the segmentation function, see Figure 36.

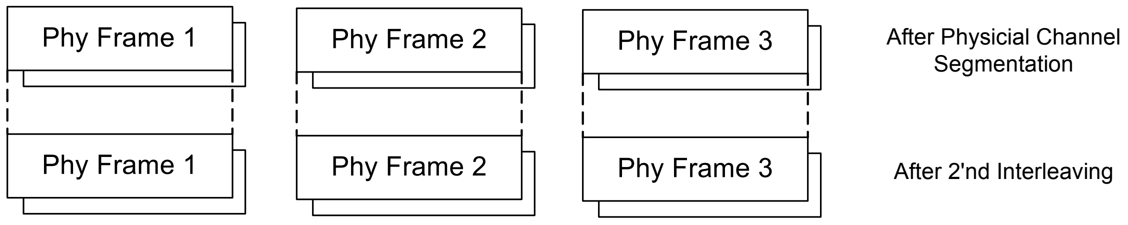


Figure 36 After second interleaving

Like with the AMR service, parts of the interleaving function can be performed in SIMD parallel fashion depending on the arrangement on the input data and how the data is required to be arranged for the following function.

6.12 Physical channel mapping

After the second interleaving, the bits are ready to be mapped on the physical channels [14]. The input bits for physical channel mapping are denoted $v_{p,1}, v_{p,2}, \ldots, v_{p,U}$, where p is the number of physical channels used and U is the number of bits to be transmitted in one radio frame (for each physical channel). The bits are mapped so that they are transmitted in ascending order via the air interface. Not all bits are necessarily sent over the air. Bits that have values $v_{p,k} \notin \{0,1\}$ correspond to DTX indicators or p-bits. The DTX indicators are mapped on the frames but the transmission is turned off. When compressed mode is applied, certain slots will be empty, i.e. no bits are mapped to those slots (the p-bits inserted in the interleaving function). If $N_{first} + TGL < 15$, all empty slots are mapped within a single frame. When $N_{first} + TGL > 15$, the empty slots are mapped over two consecutive frames. If the TGL span over two frames, the empty slots in the first frame should be $N_{first}, N_{first+1}, N_{first+2,\ldots}, 14$. In the second frame the empty slots shall be $0,1,2,\ldots,N_{last}$. When compressed mode by spread factor reduction is used, bits are mapped only using 7.5 slots.

AMR service

One composite input stream; one radio frame per 10 ms is mapped on 15 physical slots, see Figure 37.

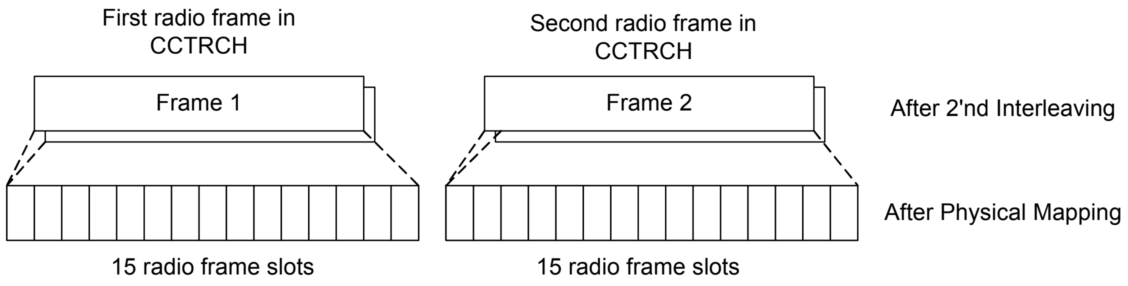
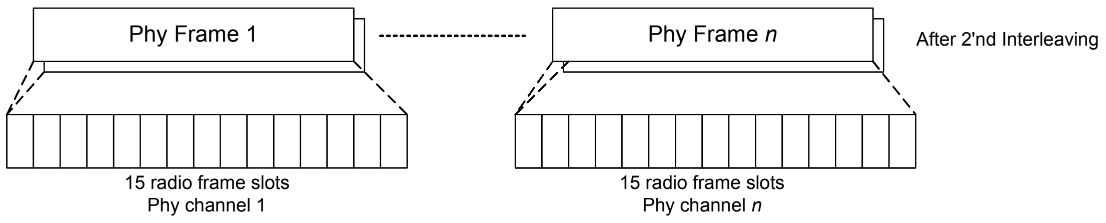


Figure 37 After AMR physical channel mapping

High bit-rate data service

The input comprise of up to 3 channel streams depending on the bit-rate requirements; one frame for each 10 ms interval is mapped using 15 slots, for each physical channel, see Figur 38.



Figur 38 After data physical channel mapping

7. Summary

This report concluded a study performed of required baseband processing in 3G UMTS radio base stations. The study is focused on the processing required for certain transmission modes in the RBS downlink using downlink dedicated transport channels (DCH:s). Two examples of user services – an AMR voice call service and a generalised case of high bit-rate data transmissions – have been used to abstractly describe and discuss processing characteristics and different ways of mapping the dataflows and dependencies through the baseband function flow. The service examples can be used as test cases for evaluation and mapping of software implemented baseband functions.

The functions are logically arranged in a pipelined sequence, operating on periodic input and output bit streams of user data. The 3G network standard is designed to support simultaneous service transmission by allocation of multiple parallel transport channels for a single mobile user. It was illustrated through abstract mapping of the two presented service examples that the function interdependencies are of producer/consumer type. Moreover, it was discussed how different kinds of parallelism can be exploited in the functions, with respect to the data stream-lengths, the number of transport blocks and allocated channels, as well as requirements for control operations to map bit stream data processing on word-length architectures. Thus it was concluded that several functions has the potential to be mapped in parallel; both in the time and the spatial domain, through exploitation of SIMD or MIMD parallelism.

Transport channels are multiplexed in the function pipeline to form a single composite transport channel. Depending on bandwidth requirements, it was illustrated that the composite channels are segmented and mapped on multiple physical channels. These multiplex, de-multiplex functions require synchronisation of data streams at several processing stages.

The operations performed by the baseband functions (in the down link), are of logical rather than arithmetical nature. Seen from an abstract functional level, the required data operations are dominated by shuffling, padding and logical manipulation on bit-field data. To be able to efficiently implement and compile the baseband functions to an architecture abstraction, based on parallel and reconfigurable processor architectures, it must be possible express the application characteristic properties in the supported programming model. Desirably, it should be possible to express bit field data types and operations as well as data and instruction parallelism on several granularity levels; function- and instruction-level parallelism.

Appendix A: Glossary

3GPP	3G Partnership Project
AMR	Adaptive Multi Rate
ASIC	Application Specific Integrated Circuit
CCH	Control Channel
CCTRCH	Coded Composite Transport CHannel
CDMA	Code Division Multiple Access
CN	Core Network
DCH	Dedicated Channel
DSP	Digital Signal Processor
DTX	Discontinuous Transmission bits
EDGE	Enhanced Data rate for GSM Evolution
FACH	Forward Access Channel
FDD	Frequency Division Duplex
FEC	Forward Error Correction
FPGA	Field Programmable Gate Array
MAC	Medium Access Control
MIMD	Multiple Instruction Multiple Data
PCH	Physical Channel
RAB	Radio Access Bearer
RACH	Random Access Channel
RNC	Radio Network Controller
RNS	Radio Network Sub system
RRC	Radio Resource Control
SF	Spreading Factor
SIMD	Single Instruction Multiple Data
TDD	Time Division Duplex
TFCI	Transport Format Combination Indicator
TFI	Transport Format Indicator
TFS	Transport Format Set
TGL	Transmission Gap Length
TTI	Transmission Time Interval
UE	User Equipment
UMTS	Universal Mobile Telecommunication Standard
UTRAN	UMTS Terrestrial Radio Access Network
WCDMA	Wideband Code Division Multiple Access

References

[1] *www.3gpp.org*
[2] H. Holma and A. Toskala, *WCDMA for UMTS*, 3rd edition, 2004, Wiley & sons
[3] 3GPP, ”UTRAN Iur/Iub interface user plane protocol for DCH data streams”, TS 25.427, *www.3gpp.org*, Release 5
[4] 3GPP, ”Multiplexing and channel coding (FDD)”, TS 25.212, *www.3gpp.org*, Release 5
[5] 3GPP, ”Services provided by the physical layer”, TS 25.302, *www.3gpp.org*, Release 5
[6] 3GPP, ”Adaptive Multi-Rate (AMR) speech codec frame structure”, TS 26.101, *www.3gpp.org*, Release 5
[7] 3GPP, ”Adaptive Multi-Rate (AMR) speech codec; Interface Iu, Uu and Nb”, TS 26.102, *www.3gpp.org*,, Release 5
[8] 3GPP, ”AMR Wideband Speech Codec; Frame structure ”, TS 26.201, *www.3gpp.org*, Release 5
[9] 3GPP, ”UE Radio Access capabilities”, TS 25.306, *www.3gpp.org*, Release 5
[10] T. V. Ramabadran and S. S. Gaitonde, ”A Tutorial on {CRC} Computations”, *IEEE-MICRO*, vol 8, no.4, aug, 1998, pp. 62-75
[11] Knuth, D., *Art of Computer Programming, Volume 2: Semi numerical Algorithms,* 3rd Edition , Addison Wesley, November 4, 1997
[12] E. Guizzo, “Closing in on the perfect code”, *IEEE Spectrum*, March 2004, pp. 28-34
[13] M. Gustafsson, K. Jamal, E. Dahlman, ”Compressed Mode Techniques for Inter-frequency Measurements in a Wide-band DS-CDMA System”, *The 8th IEEE Int. Symp. on Personal, Indoor and Mobile Radio Communications*, 1997, vol.1, pp. 231-235
[14] 3GPP, ”Physical channels and mapping of transport channels onto physical channels (FDD)”, TS 25.211, *www.3gpp.org*, Release 5

Paper C

A Configurable Framework for Stream Programming Exploration in Baseband Applications

---

Bengtsson, J. and Svensson, B. (2006). A Configurable Framework for Stream Programming Exploration in Baseband Applications. In *Proc. of 11th Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments in conjunction with Int'l Parallel and Distributed Processing Symp. (IPDPS 2006)*, Rhodes, Greece.

A Configurable Framework for Stream Programming Exploration in Baseband Applications

Jerker Bengtsson and Bertil Svensson

Centre for Research on Embedded Systems
Halmstad University
PO Box 823, SE-301 18 Halmstad, Sweden
{Jerker.Bengtsson, Bertil.Svensson}@ide.hh.se

Abstract

This paper presents a configurable framework to be used for rapid prototyping of stream based languages. The framework is based on a set of design patterns defining the elementary structure of a domain specific language for high-performance signal processing. A stream language prototype for baseband processing has been implemented using the framework. We introduce language constructs to efficiently handle dynamic reconfiguration of distributed processing parameters. It is also demonstrated how new language specific primitive data types and operators can be used to efficiently and machine independently express computations on bit-fields and data-parallel vectors. These types and operators yield code that is readable, compact and amenable to a stricter type checking than is common practice. They make it possible for a programmer to explicitly express parallelism to be exploited by a compiler. In short, they provide a programming style that is less error prone and has the potential to lead to more efficient implementations.

1. Introduction

Advanced embedded high-performance applications put very high requirements on computer systems design. Some examples are modern radar systems and baseband processing in radio base stations (RBS). Although the specific requirements are somewhat different, the computational characteristics are quite similar. Traditionally this kind of applications have required development of ASICs and special purpose hardware to cope with the requirements. Parallel architectures for high-performance applications has been a topic of research during many years. In recent years, results of this research and the advances in silicon process technology have opened up for a commodity market of highly parallel and reconfigurable architectures spanning from tens to several hundreds of processors on a single die [1, 2, 3].

Compiler technology and language development, on the other hand, have not kept pace with the advances in processor architecture. New approaches are required in order to exploit the vast amount of exposed parallelism and communication structures. On the one hand, languages must offer constructs and operations that allow a programmer to express parallelism and computations that are characteristic for a certain application domain. On the other hand, to enable efficient compilation, languages must be structured for a machine abstraction that correlates well with the target architectures. These arguments speak in favor of a domain specific approach rather than a general purpose programming approach.

The goal of our research is to investigate and develop a stringent programming and compilation framework for domain specific high-performance applications, targeting parallel and reconfigurable processors. In order to investigate what primitive language constructs, data types and operators are needed in an efficient programming language, our approach is to perform implementation experiments using realistic applications and experimental tools which can be used to quickly implement executable language prototypes. The application used for implementation studies in this work is baseband processing performed in 3G WCDMA radio base stations. An experimental framework has been implemented in Java to be able to perform quick prototype development and emulation of domain specific programming languages.

This paper is organized as follows. A background and motivation for the work is given in Section 2. The configurable framework that has been developed for implementation experiments is presented in Section 3. In Section 4, the language *StreamBits* which has been implemented for baseband processing is presented. Section 5 shows experiments that were conducted to demonstrate the applicability of the language. Finally, the paper is summarized with conclusions and future work in Section 6.

2. Background

The baseband provides the modem functionality in a wireless communication system and constitutes the core in the 3G WCDMA technology. A radio base station provides a set of full duplex physical data and control channels, which are used to map higher layer data packets to physical radio frames. [4]. The baseband resources of an RBS are managed by a higher layer Radio Network Controller (RNC), which is responsible for traffic scheduling on the physical user channels provided by the baseband. The computations performed in the RBS mainly constitute data flows of bit-intensive protocol and signal processing, where the processing is controlled by service parameters given by the RNC. A baseband processing board is a complex unit, for which many design parameters have to be considered. Besides meeting the hard requirements in performance, it must provide scalability and be sustainable for evolutions in standards. At the same time, customers want low-cost RBS product solutions [5]. The life cycle of an RBS is measured in several decades, not years. To decrease initial product development costs it is an advantage if COTS components can be used to as large an extent as possible. Even if in-house ASIC solutions are hard to compete with in terms of performance and energy efficiency, they require large volumes in order to be a cost efficient solution. Also, considering the life cycle of an RBS, it is desirable not to encapsulate more functionality than necessary into ASICs at early product generations. New standard network functionalities are constantly released and it must be possible to incorporate these in existing platforms with minimal changes in hardware.

To meet these kinds of requirements in system design, the trend is that more of the baseband functionalities are implemented using programmable solutions. One approach to support hardware flexibility is to abstract the baseband implementation through definition of an application programming interface (API). Thus, the physical implementation of the baseband processing components can be disregarded by the programmer, and the functionality of the components can be implemented in either software or hardware.

2.1 Parallel and reconfigurable processors

Of specific interest for the addressed application domain are the array structured parallel and reconfigurable processors. Most of these architectures have been developed for the purpose of compute and data intensive applications such as baseband processing. In this paper the term processor is used for the entire parallel processor on a chip, whereas we refer to the constituent processing elements as PEs. This specific category of processors offers parallelism on different granularity levels, which provides a highly formable program mapping space. The PEs of the array are in general tightly coupled, using low-latency communication networks controlled by the instruction set. The exposed details of the low-latency interconnect structures, combined with the high degree of parallelism, makes it possible to enhance performance by arranging parallel computations as streams.

Parallel and reconfigurable architectures can be grouped into three categories after granularity and processing principle. The more coarse-grained are usually designed after the MIMD principle [6, 2]. The second category can be characterized as SIMD/vector machines, constituting clusters of vector or SIMD units, orchestrated by one or several single instruction stream controllers [7, 8]. Finally the third category are what can be called semi-static configurable arrays [3]. These are more fine-grained architectures; they resemble FPGAs, but the PEs are of word-level ALU type instead of bit-level type. What is common for these architectural categories is that they expose a lower than usual level of the hardware for configuration by the compiler.

A general principle for these architectures is that they have no hardware-implemented cache logic and that most are designed with distributed private memory. Thus, complex cache and coherence mechanisms can be removed in favor of more computation-oriented logic. Instead, the data access arrangement needs to be expressed and configured by the programmer and the compiler.

2.2 Stream programming and compilation

The flexibility and parallelism offered by parallel and reconfigurable architectures have increased the complexity for both the programmers and the compiler tools. Most current architectures are accompanied with a specific approach for programming and compiling, and many of these approaches are based on the language C. This is done through either some machine specific extensions to the C syntax [8, 9] or as a combination with another machine specific language [2, 10]. This is not an optimal approach – neither from the perspective of application programming, nor for compilation efficiency. First of all, the C language and compilers have originally been developed and optimized for general purpose programming, to be compiled for a von Neumann based machine abstraction with a single instruction stream and global-memory abstraction. Thus, there is no support in the language to explicitly express application parallelism. Data-parallel operations usually have to be extracted from serial loop iterations [11]. The C language also imposes very liberal programming of memory usage, allowing global pointers, recursive function calls and dynamic memory allocation. Since the targeted parallel architectures have distributed memory and no automatic caching and coherence mechanisms, it is very hard – or perhaps even impossible – to produce performance efficient code based on this kind of programming.

One purpose with the baseband API is to be able to choose hardware components from several suppliers. Thus, the program code must be portable. Without a general programming language, which can be efficiently compiled to other architectures, a large amount of reimplementations would be required. For example, when a processor is programmed using an architecture specific combination of C (for function implementation) and a subset of VHDL (for the interconnect structures), porting programs to another architecture would most likely require changing programming paradigm as well as putting large efforts in code rewriting.

From the application point of view, one of the more interesting approaches taken is the stream programming paradigm. A stream program has the structure of a dataflow graph, constituting synchronous flows of data streaming through a pipelined set of functions [12, 9]. This is a natural way of expressing signal processing applications, which usually constitute pipelined execution of compute intensive filter kernels. Stream programming allows a programmer to explicitly express function parallelism and data locality in the program. The function dependencies in a stream program are limited to input and output streams between the functions in the flow graph; global data is not allowed to be expressed in a stream program. This deterministic flow description exposes information of function parallelism and data locality to a compiler.

One of the more interesting stream programming languages is StreamIt [12]. It is developed to be a portable programming language for array structured processors and, unlike other stream languages, both functions and program flow graphs are expressed using one language. The syntax is Java-like and it does not allow such things as global data allocation and function calls, as most C based languages do. The flow graphs are described using pipeline constructs, and functions are implemented as filters. An application can be expressed as an abstract component graph, using pipeline constructs, and the concrete API components can be defined by filter constructs. The filter construct provides a natural interface for autonomous stream functions in an API, which could as well be linked to a hardware defined component in the compiler process.

3. Experimental framework for stream programming

In this section we present a programming framework that has been developed for experiments with stream programming. Specifically, we are interested in experimenting with new language types and structures currently not supported in StreamIt. There are two important aspects that need to be addressed. First, the language must provide program structures that are natural to use for definition of abstract components. Second, it must offer primitive types and operators that allow a programmer to efficiently express application characteristic computations. With the experimental framework, it is possible to quickly set up programming experiments with StreamIt language extensions without the need for laboursome compiler modifications.

Our implementation studies of the WCDMA baseband standard show that the baseband functions require a high degree of low-level data manipulation on bit-fields, and also that many computations can be executed in data-parallel fashion [13]. When traditional high-level primitives, such as integers and bytes are used, low-level bit operations cannot be naturally expressed. Implementation of bit operations normally have to take machine-specific details, such as register word-length, into consideration. Furthermore, this kind of low-level programming is quite error prone and it would be desirable to perform compile-time type checks on such primitive operations.

With our framework it is possible to investigate how bit-level and data-parallel operations can be expressed more efficiently without considering machine-specific details when implementing algorithms. It is also possible to define type check rules. The framework is implemented in Java and it is based on a set of design patterns, which define the elementary structure of a stream language. This elementary structure is, to a large extent based, on the StreamIt language structure, but it is not identical. New data primitives and stream constructs can rapidly be implemented by making extensions to the framework. In the next subsection we discuss the predefined basic language structures of the framework, highlighting new features not offered by the StreamIt language. Then we discuss the implementation of these structures in the framework. Finally, we discuss the implementation of primitive types and operators.

3.1 Basic language constructs

A stream program is constructed using `Filter` and `Pipeline` components, which form a network of functions and data streams, see Figure 1. The `Filter` is the basic construct in which instructions are grouped to perform a computation. The `Pipeline` construct is used to organize stream components into a composite network. A component is added to a pipeline by the `add(component)` command.

The `Filter` and `Pipeline` components are connected with I/O tapes which constitute the data streams flowing through the network. A tape is implemented by a FIFO buffer of homogeneous data types. In StreamIt, `Filter` and `Pipeline` components can only be attached to a single stream. In our framework, we support implementation of dual tapes – one for data streams and one for streams of configuration parameters. The data tape constitutes the stream on which a filter performs its computation. The configuration tape is used to stream reconfiguration parameters to filters throughout the distributed network.

A `Filter` has three execution modes – `init`, `work` and `configure`. Transitions between these modes are mapped automatically at compilation time and the programmer only needs to define the functionality within each mode. The `init` mode is executed once, before the first firing of the filter, to initialize variables and parameters. The `work` mode implements the computations performed when a filter is working in steady state. The `configure` mode is a new language feature not supported in StreamIt. It is executed once before each execution of the `work` mode. The `configure` mode has been implemented to support more flexible programming of parameter configuration of the baseband algorithms (recall the configuration parameters signalled from the RNC), which must be performed periodically during program execution. With a `configure` mode and a separate configuration tape, configuration programming can be defined and modified without any changes in the `work` mode.

As in StreamIt, the I/O streams are accessed by `peek()`, `pop()` and `push()` operators. The `pop()` operator reads and removes an item from the stream, while `peek()` reads the item but does not remove it from the stream. In our framework, `popD()`, `peekD()` and `pushD()` are used for data streams and `popC()`, `peekC()` and `pushC()` for configuration streams.

The framework is designed for stream programs with static stream rates only. That is, input and output stream rates must be defined constant by the programmer when implementing a filter. A reason for this restriction is to make it possible to, at compile time, check and assert stream rate compatibility between filters in the network.

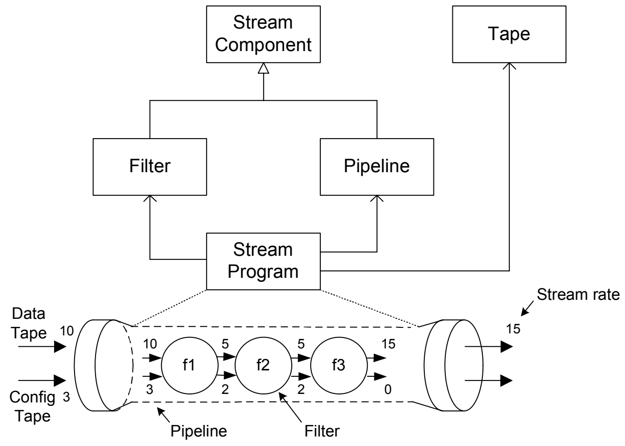


Figure 1. The framework structure

3.2 Implementation of the basic language constructs

In this subsection we discuss how the basic language constructs are implemented in the Java-based framework. This includes `Tape`, `Filter` and `Pipeline` components, and it is shown how they are put together to form a stream network. The framework is structured using a set of design patterns in combination with type generics in Java 5.0 SDK [14].

The `StreamComponent` is a type-generic interface that defines the contractual functionality that a component must implement to be executable in a stream program. A `StreamComponent` must be attached to both data and configuration I/O tapes, where each tape can be a stream of different data type. This is handled elegantly by usage of generics in Java. The interface is parameterized using four generic types, for data and configuration streams respectively. The generic data types are instantiated by the class that implements the interface.

There are currently two basic components in the framework which implement the `StreamComponent` interface – `Filter` and `Pipeline`. The `Filter` is a generic component that defines the basic structure of a filter construct in a stream language. The abstract parts of the `Filter` component constitute methods for `work`, `init` and `configure`, which must be implemented by a programmer to define the execution in these modes. The `configure` and `work` modes are called automatically in a deterministic order.

The `Pipeline` defines how `StreamComponents` are ordered to form a stream subnetwork. Since a `Pipeline` is a `StreamComponent` itself, it is possible to construct hierarchical pipelines. When a `StreamComponent` is attached to a pipeline with the `add(component)` operation, the stream rate compatibility with the preceding and the following components is checked. Both the `Pipeline` and the `Filter` templates are defined with type-generics for input and output streams. These types are defined by the programmer when instantiating a `Filter` or `Pipeline` using the `Filter` and `Pipeline` component templates.

Tapes are defined by a generic `Tape` component. The buffer data type is defined when a programmer makes an `add(component)` operation. The buffer size is determined at compilation time using the I/O stream rate directives that must be specified by the stream programmer when instantiating a component using `Pipeline` and `Filter` component templates.

The `StreamProgram` component is the top-level pipeline in a stream program. The programmer adds components to the program by using the `add(component)` directive in the method `streamProgram`, which is the main function called automatically at program execution. Besides adding components to the main pipeline, the programmer must also define the I/O stream types for both data and configuration streams.

3.3 Stream data types and operators

We now discuss the implementation of components for type definition and type operators. One of the main goals with the framework is to be able to elaborate with primitive stream data types for baseband applications. The framework allows strong type checking definitions on operations with primitive types. Since the framework is implemented using Java, some of the type checking must be done during run-time. However, in a real compiler implementation these type checks would be performed at compile time.

The `StreamType` is a generic interface for implementation of stream data primitives. This interface defines a common subset of abstract arithmetic, logic, relational and typecast operators. The `StreamType` interface must be implemented when defining a new stream data type, which in turn requires the common operations to be defined by the implementing type. Operators defined by `StreamType` (that are not valueless) take a generic type as input and return a value of generic type.

A major strength is that type checking rules can be defined for each data type that implements the `StreamType` pattern. All primitive stream types are implemented as abstract data types in Java.

4. Implementation of StreamBits

The framework has been used to implement *StreamBits*, which is a prototype language for baseband API development. The baseband input consists of bit-serial data streams that must be processed within hard real-time intervals. Currently, we have focused on the primitive type system and operators suitable for efficient implementation of bit-field manipulations and data-parallel operations. In this section we present these new types and operators. To demonstrate the advantages with our approach, we compare these types and operators with C-based expressions.

Table 1. StreamBits types compared with C

StreamBits type	C type
$bitvecST(n)$ is a bit field type for each value of n	*int*, *byte*
$vecST(e_{0w}, e_{1w}, e_{2w}, e_{3w})$ is a type for parallel vectors with elements e_n of width w	*int*[4], *byte*[4] scalar array of *int* or *byte*

When a traditional C-programming approach is used, computations on bit-streams require a large amount of assembly-like machine-dependent expressions based on bit masks and shifts. This normally results in source code that is very hard to read and understand. Also, this kind of operations requires careful implementation by the programmer, since C-like languages lack type notions for bit field computations and therefore can not provide type safety. Furthermore, it results in machine dependent code, since the programmer must pack bits and calculate masks and shifts that are bound to fixed-length machine registers.

StreamBits has been implemented with types for bit-field and data-parallel operations. The definitions of these types are presented in Table 1. The *StreamBits* primitives are listed in the first column of the table and the corresponding type expressions in C primitives in the second.

Types for bit-fields. `bitvecST` is the type for declaration of bit-fields of length n. Thus, it is possible to define a set of n distinct bit-field types $t(n)$. In comparison, since in C, there is no primitive type notion for bit-fields, such data quantities must be expressed using integer or byte types. Therefore, type-correct operations on integers are also type-correct for bit-fields of arbitrary length. This type-mismatched bit mapping is quite error prone and not desirable.

Data-parallel type. `vecST` is a type that allow fine-grained data-parallel operations to be expressed explicitly within a `Filter`, see row 2 in Table 1. The `vecST` type is defined as a vector of four elements, each of 32-bit width. Note that the definition of the vector type is parameterized by the number of elements e_n and the bit-field width w. Since there is no parallel notation in C, vector data are usually expressed using array constructs which are accessed scalar-wise.

Bit-field operations. A sub-field in `bitvecST` types is accessed using `bitslice` operators. Bit-fields

Table 2. StreamBits compared with C

bitvecST oper.	Corresponding C expr.
$bitslice(m:n)$	$(t \mathbin{\&} w_{m:0})$
$bitsliceL(m:n)$	$(t \mathbin{\&} w_{m:0}) \ll (w-m)$
$bitsliceR(m:n)$	$(t \mathbin{\&} w_{m:0}) \gg n$
$bitslicePack(m:n)$	N/A

$lmerge(k:l, m:n)$	if $l <= (m-n)$:
	$((t \mathbin{\&} w_{k:l}) \ll C_1) \mid ((s \mathbin{\&} w_{m:n}) \gg n)$
	if $l > (m-n)$:
	$((t \mathbin{\&} w_{k:l}) \gg C_2) \mid ((s \mathbin{\&} w_{m:n}) \gg n)$

Table 3. Operator comparison vecST

StreamBits oper.	Corresponding C expression
$vecslice(m:n)$	$for\ i = 0\ to\ 4\{t[e_i] \mathbin{\&} w_{m:n}\}$
$vecsliceL(m:n)$	$for\ i = 0\ to\ 4\{t[e_i] \mathbin{\&} w_{m:n} \ll (w-m)\}$
$vecsliceR(m:n)$	$for\ i = 0\ to\ 4\{t[e_i] \mathbin{\&} w_{m:n} \gg n\}$
$lmerge(k:l, m:n)$	$for\ i = 0\ to\ 4\{$ if $l <= (m-n)$: $(t[e_i] \mathbin{\&} w_{k:l}) \ll C_1 \mid (s[e_i] \mathbin{\&} w_{m:0}) \gg n$ if $l > (m-n)$: $(t[e_i] \mathbin{\&} w_{k:l}) \gg C_2 \mid (s[e_i] \mathbin{\&} w_{m:0}) \gg n$

of `bitvecST` type can also be merged using the `lmerge` operator. In Table 2, we list these operators and, for comparison, the corresponding C expressions. Bit-field upper and lower boundaries are annotated with k, m and l, n respectively. w is used for machine word-length, and bit masks of machine register length are annotated with $w_{m:0}$, where m represents the upper boundary of the bit mask.

The `bitslice` operator is currently defined for two cases – unaligned and aligned bit-slicing. Unaligned bit-slicing is performed with the `bitslice("m:n")` operator, row 1 in the table. The operator `bit-slice("m:n")` produces a `bitvecST` with the same length as the operand, where bits m through n are copied from the corresponding bit-field in the operand, and the rest are set to 0.

Aligned bit-slicing is performed by the operators `bitsliceL("m:n")`, `bitsliceR("m:n")` and `bitslicePack("m:n")` listed on rows 2 through 4. The `bitsliceL("m:n")` operator produces a bit-field copy, like `bitslice("m:n")`, but the copied bit-field is aligned to the `bitvecST(`n`)` upper bound n. Similarly, the `bitsliceR("m:n")` operator produces a bit-field copy aligned to the lower bound. The corresponding expressions in C, on rows 2 and 3, require logical *AND* and a *SHIFT* operations. In comparison, the `bitsliceL` and `bitsliceR` operators allow this operation to be more compactly expressed than the required C expression. Also, in the C expression for left-aligned masking on row 2, it is assumed that the upper bound is equal to the word length. But, since the `bitvecST(`n`)` type is defined for bit-fields of length n, this is generally not the case. The alignment of `bitsliceL` and `bitsliceR` values in *StreamBits* can be handled automatically at compile time.

The `bitslicePack` produces a left-aligned bit-field copy, just like the `bitsliceL` operator, but the result is packed into a `bitvecST(`s`)` where the field length s is equal to the length of the copied bit-field $(m-n)$.

The `lmerge` operator is used to merge two `bitvecST` bit-fields. The result is a `bitvecST(`s`)` where the length s is the sum of the two merged bit-field lengths $(k-l+m-n)$. The first operand is aligned to the left of the second operand. A right-aligned merge can be achieved by simply switching the order of the operands. The corresponding C expression requires at most 5 operations. The first operand should be aligned to the left of the second, which requires either a left or a right shift, depending on if the masked bit-fields are overlapping or not ($l <= (m-n)$ or $l > (m-n)$). This alignment is a shift constant specified by the programmer (C_1 and C_2). Thus, the if cases are used only to mark two separate alignment cases.

Data-parallel operations. Besides the basic arithmetic and logical operations, the `vecST` type also supports bit-field operations, such as `lmerge`, `vecslice`, `vecsliceL` and `vecsliceR`, see Table 3. The `lmerge` operation is defined precisely as `lmerge` for `bitvecST` types. The merge is performed *in parallel* for each of the vector elements and the result is of type `vecST`. In *StreamBits*, the maximum length of an element merge is w.

The `vecslice(m:n)`, `vecsliceL(m:n)` and `vec-sliceR(m:n)` operators are vector-parallel versions of `bitslice` operators. Since there is no corresponding parallel notation in C, parallelism must be transformed into sequential expressions, typically using loop constructs and scalar data arrays. This is illustrated in the right part of Table 3. Few will argue that it is a natural way of expressing application parallelism – to use sequential scalar constructs, which are then to be parallelized by a compiler that is only aware of the sequential constructs given by the programmer.

Some computations require scalar processing of `vecST` elements. Scalar elements in `vecST` are accessed by `getElement(`e_i`)` and `setElement(`e_i`,` val`)` operators, where e_i is the element index of the vector and val is the value.

5. Experiments with 3G UMTS baseband functions

In order to demonstrate and evaluate the applicability of the *StreamBits* language, we have conducted experiments with baseband functions. In this paper two different implementation examples of one baseband function are presented – cyclic redundancy check (CRC) processing for a voice call service and for high

```
for(intST cnt;cnt.lt(len);cnt.Assign(cnt.incr())){
1.  if(cnt.mod(new intST(4)).eq(new intST(0))){
2.     out.Assign(temp);
3.     temp.Assign(peekD());
4.     r_tmp.Assign(temp);
5.     pushD(popD());
6.  }
7.  t.Assign(r.bitsliceR("31:24"));
8.  r.Assign(r.lmerge("23:0", r_tmp.bitsliceR("31:24")));
9.  r.Assign(r.XOR(table[crc_poly.getVal()][t.getVal()]));
10. r_tmp = r_tmp.lshift(8);
}
```

Figure 2. CRC implemented in StreamBits

bit-rate data services [13].

CRC processing for voice services. Voice calls are coded using an adaptive multi rate codec (AMR). The baseband data input constitutes three AMR encoded bit-streams mapped onto separate transport channels, A, B and C. Each stream constitutes coded speech data of different importance to the quality of a voice channel; the A bits are the most important and the C bits the least. The channels are processed with different baseband parameters; only the A channel bits are transmitted with a CRC computed checksum. The CRC implementation, shown in figure 2, is a table-driven algorithm which encodes a single transport channel (Note that this is a selected part and not the complete CRC baseband function). The variables, except for the loop counter, are all of `bitvecST` type. In each loop iteration, this algorithm encodes eight-bit long fields of the input stream. The input bits are packed into a stream of `bitvecST` data type. Each `bitvecST` is 32 bits long, thus four iterations are required to process each `bitvecST` that is read from the input stream. Defined by the code within the `if` clause, every fourth iteration a new `bitvecST` value is read into a temporary input register (`r_tmp`), while the previous one is pushed to the output stream.

On line 7, the next 8-bit field to be encoded is read from the encoder register `r` using the `bitsliceR` operator. On line 8, the MSB from the temporary input register `r_tmp` is shifted into the LSB of the encoder register `r` using `bitslice` and `lmerge`.

Finally, the lookup value is read from the table to be XOR:ed (the division), with the encoder register, and the next 8-bit input in the temporary register is aligned to the MSB position for the next iteration.

CRC processing for high bit-rate data. Data transmissions can be mapped using multiple transport blocks of equal size, mapped on multiple transport channels. Since there are no data dependencies between the transport blocks, multiple blocks can be processed in parallel using SIMD control.

The *StreamBits* code in Figure 3 represents a SIMD parallel implementation of the same encoding algorithm previously demonstrated for the AMR service.

```
for(intST cnt;cnt.lt(len);cnt.Assign(cnt.incr())){
1:  if(cnt.mod(new intST(4)).eq(new intST(0))){
2:    out.Assign(temp);
3:    temp.Assign(peekD());
4:    r_tmp.Assign(temp);
5:    pushD(popD());
6:  }
7:  t.Assign(r.vecslice("31:24"));
8:  r.Assign(r.lmerge("23:0", r_tmp, "31:24"));
9:  v0.Assign(table[poly.val()][t.getElement(0).val()]);
10: v1.Assign(table[poly.val()][t.getElement(1).val()]);
11: v2.Assign(table[poly.val()][t.getElement(2).val()]);
12: v3.Assign(table[poly.val()][t.getElement(3).val()]);
13: r.Assign(r.XOR(new vecST(v0, v1, v2, v3)));
14: r_tmp.Assign(r_tmp.lshift(8));
}
```

Figure 3. CRC processing for DATA

The `cnt` and `v0-v3` variables are scalar variables of `intST` type, and the other are of `vecST` type. Like in the AMR example, the parallel encoder encodes 8 bits of the input stream per iteration.

Each data item in the I/O streams constitutes a 32-bit field of four simultaneous transport block input streams; one transport block stream per vector element. The parallel `vecslice` operator on line 7 copies the next 8 bits of each vector element in the encoder register `r`, and stores them aligned with the lsb positions in the `t` vector. Each element in `t` constitutes the next lookup index for each bit-stream being processed. On line 8, new input bits are shifted into the encoding register by copying the MSB of the input vector register `r_tmp`, which are merged with the remaining bits in the encoder register. Like for the `bitslice` operators, no machine dependent masking and shifting needs to be expressed in the code.

The table look-ups performed on lines 9 through 12 have to be expressed with scalar operations; one look-up for each element. This is because the input bit-fields in register `t` constitute arbitrary values from the four transport blocks and therefore it is not possible to vectorize the look-up operation. However, this does not mean that this portion of the code cannot be executed in parallel. Finally, the scalar values with the look-up values are vectorized and XOR:ed with the encoder register on line 13, and the next input bits to be shifted into the encoder register are shifted to the msb positions in the temporary input register, `r_tmp`.

6 Conclusions and future work

We have described a configurable framework to be used for experiments with stream programming development targeting embedded high-performance applications, such as baseband processing in radio base stations. A domain specific language prototype for baseband processing, called *StreamBits*, was implemented to demonstrate the use of the framework and to perform implementation experiments. We introduced stream constructs to be able to efficiently program dynamic reconfiguration of distributed processing parameters. It was shown that the language has the potential to lead to more compact and efficient codes for bit-field and data-parallel computations, compared to when typical von Neumann based languages, such as C are used. The primitive types in the language impose a programmer to explicitly express functions with inherent, fine-grained data parallelism. Moreover, it was demonstrated how the primitive data-parallel vector and bit-vector data types and operations can be used without exposing machine specific details such as register word lengths. Another advantage with the introduced primitive data types is the opportunity to perform strong type checking on low-level bit operations.

Future work will be focused on the definition of a tailored stream processing language for baseband API development. This will be based on extended prototype experiments using the configurable framework. The development will include the implementation of a compiler framework for a parallel machine abstraction, which can be applied for efficient mapping on parallel and reconfigurable, array structured processors. To support efficiency, the language should offer parallel expressions. To support portability, the syntax should not allow machine-specific details in the code.

Acknowledgment

The authors would like to thank Dr. Anders Wass at Ericsson AB and Dr. Veronica Gaspes at Halmstad University for valuable advice and suggestions. This work has been funded by research grants from Ericsson AB and the Knowledge Foundation.

References

[1] Freescale Semiconductor. MRC6011: Reconfigurable Compute Fabric (RCF) Device. www.freescale.com, Oct. 2004.

[2] G. Panesar A. Duller and D. Towner. Parallel Processing - the picoChip way! In *Proc. of Communicating Process Architectures*, pages 125–138, 2003.

[3] PACT XPP Technologies. XPP-IIb Core Overview. www.pactcorp.com, Sept. 2005.

[4] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access for Third Generation Mobile Communications*. Addison-Wesley, third edition, 2004.

[5] J. Lerzer Z. Zhang, F. Heiser and H. Leuschner. Advanced baseband technology in third-generation radio base stations. *Ericsson Review*, (01):32–41, 2003.

[6] M. B. Taylor et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proc. of Int. Symposium on Computer Architecture*, pages 2–13, 2004.

[7] J. H. Ahn et al. Evaluating the Imagine Stream Architecture. In *Proc. of Int. Symposium on Computer Architecture*, pages 14–25, 2004.

[8] A. E. Eichenberger et al. Optimizing Compiler for a CELL Processor. In *Proc. of Int. Conf. on Parallel Architectures and Compiation Techniques*, pages 161–172, 2005.

[9] A. Das et al. Imagine Programming System User's Guide. www.cva.stanford.edu/imagine, April 2004.

[10] PACT XPP Technologies. Programming XPP-IIb Systems. www.pactcorp.com, Sept. 2005.

[11] S. V. Rajopadhye. Dependence Analysis and Parallelizing Transformations. In *The Compiler Design Handbook*, pages 329–372. CRC Press, 2002.

[12] M. I. Gordon et al. A Stream Compiler for Communication-Exposed Architectures. In *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[13] J. Bengtsson. Baseband Processing in 3G UMTS Radio Base Stations. Technical Report IDE0629, Halmstad University, 2006.

[14] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, 1995.

Paper D

A Domain-specific Approach for Software Development on Manycore Platforms

---

Bengtsson, J. and Svensson, B. (2008). A domain-specific approach for software development on manycore platforms. In *ACM Computer Architecture News, Special Issue: MCC08 - Multicore Computing 2008*, 39(6):2-10.

A Domain-specific Approach for Software Development on Manycore Platforms

Jerker Bengtsson and Bertil Svensson
Centre for Research on Embedded Systems
Halmstad University
PO Box 823, SE-301 18 Halmstad, Sweden
Jerker.Bengtsson@hh.se

Abstract

The programming complexity of increasingly parallel processors calls for new tools that assist programmers in utilising the parallel hardware resources. In this paper we present a set of models that we have developed as part of a tool for mapping dataflow graphs onto manycores. One of the models captures the essentials of manycores identified as suitable for signal processing, and which we use as target for our algorithms. As an intermediate representation we introduce timed configuration graphs, which describe the mapping of a model of an application onto a machine model. Moreover, we show how a timed configuration graph by very simple means can be evaluated using an abstract interpretation to obtain performance feedback. This information can be used by our tool and by the programmer in order to discover improved mappings.

1. Introduction

To be able to handle the rapidly increasing programming complexity of manycore processors, we argue that *domain specific development tools are needed*. The signal processing required in radio base stations (RBS), see figure 1, is naturally highly parallel and described by computations on streams of data [9]. Each module in the figure encapsulates a set of functions, further exposing more pipeline-, data- and task level parallelism as a function of the number of connected users. Many radio channels have to be processed concurrently, each including fast and adaptive coding and decoding of digital signals. Hard real-time constraints imply that parallel hardware, including processors and accelerators is a prerequisite for coping with these tasks in a satisfactory manner.

One candidate technology for building baseband platforms is manycores. However, there are many issues that have to be solved regarding development of complex signal processing software for manycore hardware. One such is the need for tools that reduce the programming complexity and abstract the hardware details of a particular manycore processor. We believe that if industry is to adopt manycore technology *the application software, the tools and the programming models need to be portable*.

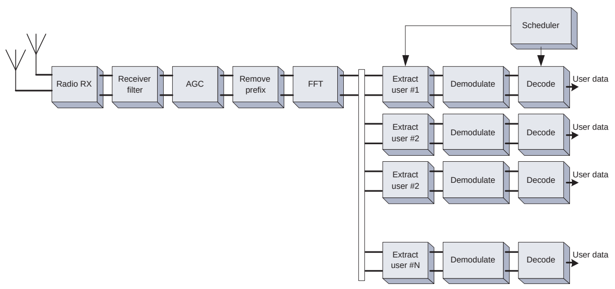


Figure 1. A simplified modular view of the principal functions of the baseband receiver in long term evolution (LTE) RBS.

Research has produced efficient compiler heuristics for programming languages based on streaming models of computation (MoC), achieving good speedup and high throughput for parallel benchmarks [3]. However, even though a compiler can generate optimized code the programmer is left with very little control of how the source program is transformed and mapped on the cores. This means that if the resulting code output does not comply with the system timing requirements, the only choice is to try to restructure the source program. We argue that *experienced application programmers must be able to direct and specialize the parallel mapping strategy by giving directive tool input*.

For complex real-time systems, such as baseband processing platforms, we see a need for tunable code parallelization- and mapping tools, allowing programmers to take the system's real-time properties into account during the optimization process. Therefore, complementary to fully automatized parallel compilers, we are proposing an iterative code parallelization- and mapping tool flow that allows the programmer to tune mapping by:

- analyzing the result of a parallel code map using performance feedback
- giving timing constraints, clustering and core allocation directives as input to the tool

In our work we address the design and construction of one such tool. We focus on suitable well defined dataflow models of computation for modeling applications and manycore targets, as well as the base for our intermediate representation for manycore code-generation. One such model, synchronous dataflow (SDF), is very suitable for describing signal processing flows. It is also a good source for code-generation, given that it has a natural form of parallelism that is a good match to manycores. The goal of our work is a tool chain that allows the software developer to specify a manycore architecture (using our *machine model*), to describe the application (using SDF) and to obtain a generated mapping that can be evaluated (using our *timed configuration graph*). Such a tool allows the programmer to explore the run time behaviour of the system and to find successively better mappings. We believe that this iterative, machine assisted, workflow, is good in order to keep the application portable while being able to make trade-offs concerning throughput, latency and compliance with real-time constraint on different platforms.

In this paper we present our set of models and show how we can analyze the mapping of an application onto a manycore. More specifically, the contributions of this paper are as follows:

- A parallel machine model usable for modelling array-structured, tightly coupled manycore processors. The model is presented in Section 2, and in Section 3 we demonstrate modeling of one target processor.
- A graph-based intermediate representation (IR), used to describe a mapping of an application on a particular manycore in the form of a (*a timed configuration graph*). The use of this IR is twofold. We can perform an abstract interpretation that gives us feedback about the dynamic behaviour of the system. Also, we can use it to generate target code. We present the IR in Section 4.
- We show in Section 5 how parallel performance can be evaluated through abstract interpretation of the timed configuration graph. As a proof of concept we have implemented our interpreter in the Ptolemy II software framework using dataflow process networks.

We conclude our paper with a discussion of our achievements and future work.

2 Model Set

In this section we present the model set for constructing *timed configuration graph*s. First we discuss the application model, which describes the application processing requirements, and then the machine model, which is used to describe computational resources and performance of manycore targets.

2.1 Application Model

We model an application using SDF, which is a special case of a computation graph [5]. An SDF graph constitutes a network of actors - atomic or composite of variable granularity - which asynchronously compute on data distributed via synchronous uni-directional channels. By definition, actors in an SDF graph fire (compute) in parallel when there are enough tokens available on the input channels. An SDF graph is computable if there exists at least one static repetition schedule. A repetition schedule specifies in which order and how many times each actor is fired. If a repetition schedule exists, buffer boundedness and deadlock free execution is guaranteed. A more detailed description of the properties of SDF and how repetition schedules are calculated can be found in [6].

The Ptolemy II modelling software provides an excellent framework for implementing SDF evaluation- and code generator tools [1]. We can very well consider an application model as an executable specification. For our work, it is not the correctness of the implementation that is in focus. We are interested in analyzing the dynamic, non-functional behaviour of the system. For this we rely on measures like worst case execution time, size of dataflows, memory requirements etc. We assume that these data have been collected for each of the actors in the SDF graph and are given as a tuple

$$< r_p, r_m, R_s, R_r >$$

where

- r_p is the worst case computation time, in number of operations.
- r_m is the requirement on local data allocation, in words.
- $R_s = [r_{s_1}, r_{s_2}, ..., r_{s_n}]$ is a sequence where r_{s_i} is the number of words produced on channel i each firing.
- $R_r = [r_{r_1}, r_{r_2}, ..., r_{r_m}]$ is a sequence r_{r_j} is the number of words consumed on channel j each firing.

2.2 Machine Model

One of the early, reasonably realistic, models for distributed memory multiprocessors, is the LogP model [2]. Work has been done to to refine this model, for example taking into account hardware support for long messaging, and to capture memory hierarchies. A more recent parallel machine model for multicores, which considers different core granularities and requirements on on-chip and off-chip communication is Simplefit [7]. However, this model was derived with the purpose of exploring optimal grain size and balance between memory, processing, communication and global I/O, given a VLSI budget and a set of computation problems. Since it is not intended for modeling dynamic behaviour of a program, it does not include a fine-granular model of the communication. Taylor et al. propose a taxonomy (AsTrO) for comparison of scalar operand networks [11]. They also provide a tuple based model for comparing and evaluating performance sensitivity of on-chip network properties.

We propose a manycore machine model based on Simplefit and the AsTrO taxonomy, which allows a fairly fine-grained modeling of parallel computation performance including the overhead of operations associated with communication. The machine model comprises a set of parameters describing the computational resources and a set of abstract performance functions, which describe the computational performance of computations, communication and memory transactions. We will later show in Section 5 how we can can model dynamic, non-functional behavior of a dataflow graph mapped on a manycore target, by incorporating the machine model in a dataflow process network.

2.2.1 Machine Specification

We assume that cores are connected in a mesh structured network. Further that each core has individual instruction decoding capability and software managed memory load and store functionality, to replace the contents of core local memory. We describe the resources of such a manycore architecture using two tuples, M and F. M consists of a set of parameters describing the processors resources:

$$M =< (x, y), p, b_g, g_w, g_r, o, s_o, s_l, c, h_l, r_l, r_o >$$

where

- (x, y) is the number of rows and columns of cores.
- p is the processing power (instruction throughput) of each core, in *operations per clock cycle*.
- b_g is global memory bandwidth, in *words per clock cycle*
- g_w is the penalty for global memory write, in *words per clock cycle*
- g_r is the penalty for global memory read, in *words per clock cycle*
- o is software overhead for initiation of a network transfer, in *clock cycles*
- s_o is core send occupancy, in *clock cycles*, when sending a message.
- s_l is the latency for a sent message to reach the network, in *clock cycles*
- c is the bandwidth of each interconnection link, in *words per clock cycle*.
- h_l is network hop latency, in *clock cycles*.
- r_l is the latency from network to receiving core, in *clock cycles*.
- r_o is core receive occupancy, in *clock cycles*, when receiving a message

F is a set of abstract functions describing the performance of computations, global memory transactions and local communication:

$$F(M) =< t_p, t_s, t_r, t_c, t_{gw}, t_{gr} >$$

where

- t_p is a function evaluating the time to compute a list of instructions
- t_s is a function evaluating the core occupancy when sending a data stream
- t_r is a function evaluating the core occupancy when receiving a data stream
- t_c is a function evaluating network propagation delay for a data stream
- t_{gw} is a function evaluating the time for writing a stream to global memory
- t_{gr} is a function evaluating the time for reading a stream from global memory

A specifc manycore processor is modeled by giving values to the parameters of M and by defining the functions $F(M)$.

3 Modeling the RAW Processor

In this section we demonstrate how we configure our machine model in order to model the RAW processor [10]. RAW is a tiled, moderately parallel MIMD architecture with 16 programmable tiles, which are tightly connected via two different types of communication networks: two statically- and two dynamically routed. Each tile has a MIPS-type pipeline and is equipped with 32 KB of data and 96 KB instruction caches.

3.1 Parameter Settings

We are assuming a RAW setup with non-coherent off-chip global memory (four concurrently accessible DRAM banks), and that software managed cache mode is used. Furthermore, we concentrate on modeling usage of the dynamic networks, which are dimension-ordered, wormhole-routed, message-passing type of networks. The parameters of M for RAW with this configuration are as follows:

$$\begin{aligned} M =< \quad (x,y) =& \quad (4,4),\\ p =& \quad 1,\\ b_g =& \quad 1,\\ g_w =& \quad 1,\\ g_r =& \quad 6,\\ o =& \quad 2,\\ s_o =& \quad 1,\\ s_l =& \quad 1,\\ c =& \quad 1,\\ h_l =& \quad 1,\\ r_l =& \quad 1,\\ r_o =& \quad 1 > \end{aligned}$$

In our model, we assume a core instruction throughput of p operations per clock cycle. Each RAW tile has an eight-stage, single-issue, in-order RISC pipeline. Thus, we set $p = 1$. An uncertainty here is that in our current analyses, we cannot account for pipeline stalls due to dependencies between instructions having non-equal instruction latencies. We need to make further practical experiments, but we believe that this in general will be averaged out equally on cores and thereby not have too large effects on the estimated parallel performance.

There are four shared off-chip DRAMs connected to the four east-side I/O ports on the chip. The DRAMs can be accessed in parallel, each having a bandwidth of $b_g = 1$ words per clock cycle per DRAM. The penalty for a DRAM write is $g_w = 1$ cycle and correspondingly for read operation $g_r = 6$ cycles.

Since the communication patterns for dataflow graphs are known at compile time, message headers can be precomputed when generating the communication code. The overhead includes sending the header and possibly an address (when addressing off-chip memory). We therefore set $o = 2$ for header and address overhead when initiating a message.

The networks on RAW are mapped to the core's register files, meaning that after a header has been sent, the network can be treated as destination or source operand of an instruction. Ideally, this means that the receive and send occupancy is zero. In practice, when multiple input and output dataflow channels are merged and physically mapped on a single network link, data needs to be buffered locally. Therefore we model send and receive occupancy – for each word to be sent or received – by $s_o = 1$ and $r_o = 1$ respectively. The network hop-latency is $h_l = 1$ cycles per hop and the link bandwidth is $c = 1$. Furthermore, the send and receive latency is one clock cycle when injecting and extracting data to and from the network: $s_l = 1$ and $r_l = 1$ respectively.

3.2 Performance Functions

We have derived the performance functions by studying the hardware specification and by making small comparable experiments on RAW. We will now show how the performance functions for RAW are defined.

Compute The time required to process the fire code of an actor on a core is expressed as

$$t_p(r_p, p) = \left\lceil \frac{r_p}{p} \right\rceil$$

which is a function of the requested number of operations r_p and core processing power p. To r_p we count all instructions except those related to network send- and receive operations.

Send The time required for a core to issue a network send operation is expressed as

$$t_s(R_s, o, s_o) = \left\lceil \frac{R_s}{\textit{framesize}} \right\rceil \times o + R_s \times s_o$$

Send is a function of the requested amount of words to be sent, R_s, the software overhead $o \in M$ when initiating a network transfer, and a possible send occupancy $s_o \in M$. The $framesize$ is a RAW specific parameter. The dynamic networks allow message frames of length within the interval $[0, 31]$ words. For global memory read and write operations, we use RAWs cache line protocol with $framesize = 8$ words per message. Thus, the first term of t_s captures the software overhead for the number of messages required to send the complete stream of data. For connected actors that are mapped on the same core, we can choose to map channels in local memory. In that case we set t_s to o zero time.

Receive The time required for a core to issue a network receive operation is expressed as

$$t_r(R_r, o, r_o) = \left\lceil \frac{R_r}{framesize} \right\rceil \times o + R_r \times r_o$$

The receive overhead is calculated in a similar way as the send overhead, except that parameters of the receiving core replace the parameters of the sending core.

Network Propagation Time Modeling shared resources accurately with respect to contention effects is very difficult. Currently, we assume that SDF graphs are mapped so that the communication will suffer from no or a minimum of contention. In the network propagation time, we consider a possible network injection- and extraction latency at the source and destination as well as the link propagation time. The propagation time is expressed as

$$t_c(R_s, d, s_l, h_l, r_l) = s_l + d \times h_l + n_{turns} + r_l$$

Network injection- and extraction latency is captured by s_l and r_l respectively. Further, the propagation time is dependent on the network hop latency h_l and the number of network hops d, which are determined from the source and destination coordinates as $|x_s - x_d| + |y_s - y_d|$. Routing turns add an extra cost of one clock cycle. This is captured by the value of n_{turns} which, similar to d, is calculated using the source and destination coordinates.

Streamed Global Memory Read Reading from global memory on the RAW machine requires first one send operation (the core overhead which is captured by t_s), in order to configure the DRAM controller and set the address of memory to be read. The second step is to issue a receive operation to receive the memory contents on that address. The propagation time when streaming data from global memory to the receiving core is expressed as

$$t_{gr} = r_l + d \times h_l + n_{turns}$$

Note that memory read penalty is not included in this expression. This is accounted for in the memory model included in the IR. This is further discussed in Section 4

Streamed Global Memory Write Similarly to the memory read operation, writing to global memory require two send operations: one for configuring the DRAM controller (set write mode and address) and one for sending the data to be stored. The time required for streaming data from the sending core to global memory is evaluated by

$$t_{gw} = s_l + d \times h_l + n_{turns}$$

Like in stream memory read, the memory write penalty is accounted for in the memory model.

4 Timed Configuration Graphs

In this section we describe our manycore intermediate representation (IR). We call the IR a *timed configuration graph* because the usage of the IR is twofold:

- Firstly, the IR is a graph representing an SDF application graph, after it has been clustered and partitioned for a specific manycore target. We can use the IR as input to a code generator, in order to configure each core as well as the interconnection network and plan global memory usage of a specific manycore target.
- Secondly, by introducing the notion of time in the graph, we can use the same IR as input to an abstract interpreter, in order to evaluate the dynamic behaviour of the application when executed on a specific manycore target. The output of the evaluator can be used either directly by the programmer or to extract information feedback to the tool for suggesting a better mapping.

4.1 Relations Between Models and Configuration Graphs

A *configuration graph* $G_M^A(V, E)$ describes an application A mapped on the abstract machine M. The set of vertices $V = P \cup B$ consists of cores $p \in P$ and global memory buffers $b \in B$. Edges $e \in E$ represent dataflow channels mapped onto the interconnection network. To obtain a G_M^A, the SDF for A is partitioned into subgraphs and each subgraph is assigned to a core in M. The edges of the SDF that end up in one subgraph are implemented using local memory in the core, so they do not appear as edges in G_M^A. The edges of the SDF that reach between subgraphs can be dealt with in two different ways:

1. A network connection between the two cores is used and this appears as an edge in G_M^A
2. Global memory is used as a buffer. In this case, a vertex b (and associated input- and output edges) is introduced between the two cores in G_M^A.

When G_M^A has been constructed, each $v \in V$ and $e \in E$ has been assigned computation times and communication delays, calculated using the parameters of M and the performance functions $F(M)$ introduced in Section 2.2. These annotations reflect the performance when computing the application A on the machine M. We will now discuss how we use A and M to configure the vertices, edges and then computational delays of G_M^A.

4.1.1 Vertices.

We distinguish between two types of vertices in G_M^A: *memory* vertices and *core* vertices. Introducing *memory* vertices allows us to represent global memory. A *memory* vertex can be specified by the programmer, for example to store initial data. More typically, *memory* vertices are automatically generated when mapping channel buffers in global memory.

For *core* vertices, we abstract the firing of an actor by means of a sequence S of abstract *receive*, *compute* and *send* operations:

$$S = t_{r_1}, t_{r_2} \dots t_{r_n}, t_p, t_{s_1}, t_{s_2}, \dots, t_{s_m}$$

The *receive* operation has a delay corresponding to the timing expression t_r, representing the time for an actor to receive data through a channel. The delay of a *compute* operation corresponds to the timing expression t_p, representing the time required to execute the computations of an actor when it fires. Finally, the *send* operation has a delay corresponding to the timing expression t_s, representing the time for an actor to send data through a channel.

For a *memory* type of vertex, we assign delays specified by g_r and g_w in the machine model to account for memory read- and write latencies respectively.

When building G_M^A, multiple channels sharing the same source and destination can be merged and represented by a single edge, treating them as a single block or stream of data. Thus, there is always only one edge $e_{i,j}$ connecting the pair (v_i, v_j). We add one *receive* operation and one *send* operation to the sequence S for each input and output edge respectively.

4.1.2 Edges.

Edges represent dataflow channels mapped onto the interconnection network. The weight w of an edge $e_{i,j}$ corresponds to the communication delay between vertex v_i and vertex v_j. The weight w depends on whether we map the channel as a point-to-point data stream over the network, or in shared memory using a *memory* vertex.

In the first case we assign the edge a weight corresponding to t_c. When a channel buffer is placed in global memory, we substitute the edge in A by a pair of input- and output edges connected to a *memory* actor. We illustrate this by Figure 2. We assign a delay of t_{gr} and t_{gw} to the input and output edges of the *memory* vertex.

Figure 3 shows an example of a simple A transformed to one possible G_M^A. A repetition schedule for A in this example is $3(2ABCD)E$. The repetition schedule should be interpreted as: actor A fires 6 times, actors B, C and D fire 3 times, and actor E 1 time. The firing of A is repeated indefinitly by this schedule. We use dashed lines for actors of A mapped and translated to S inside each core vertex of G_M^A. The feedback channel from C to B is mapped in local memory. The edge from A to D is mapped via a global buffer and the others are mapped as point-to-point data streams. The integer values represent the send and receive rates of the channels (r_s and r_r), before and after A has been clustered and transformed to G_M^A, respectively. Note that these values in G_M^A are the values in A multiplied by the number of the repetition schedule.

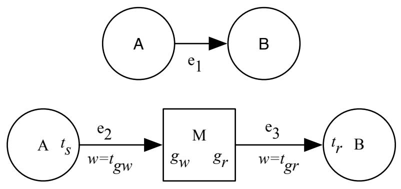


Figure 2. The lower graph (G_M^A) in the figure illustrates how the unmapped channel e_1, connecting actor A and actor B, in the upper graph (A), has been transformed and replaced by a global memory actor and edges e_2 and e_3.

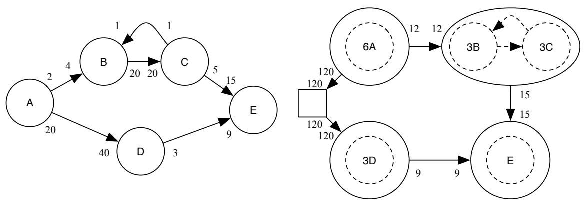


Figure 3. The graph to the right is one possible G_M^A for the graph A to the left.

5 Interpretation of Timed Configuration Graphs

In this section we show how we can make an abstract interpretation of the IR and how an interpreter can be implemented by very simple means on top of a dataflow process network. We have implemented such an interpreter using the dataflow process networks (PN) domain in Ptolemy. The PN domain in Ptolemy is a super set of the SDF domain. The main difference in PN, compared to SDF, is that PN processes fire asynchronously. If a process tries to read from an empty channel, it will block until there is new data available. The PN domain implemented in Ptolemy is a special case of Kahn process networks [4]. Unlike in a Kahn process network, PN channels have bounded buffer capacity, which implies that a process also temporarily blocks when attempting to write to a buffer that is full [8]. This property makes it possible to easily model link occupancy on the network. Conclusively, a dataflow process network model perfectly mimics the behavior of the types of parallel hardware we are studying. Thus, a PN model is a highly suitable base for an intermediate abstraction for the processor we are targetting.

5.1 Parallel Interpretation using Process Networks

Each of the core and memory vertices of G_M^A is assigned to its own process. Each of the core and memory processes has a local clock, t, which iteratively maps the absolute start and stop time, as well as periods of blocking, to each operation in the sequence S.

A core process evaluates a vertex by means of a state machine. In each clock step, the current *state* is evaluated and then stored in the *history*. The *history* is a chronologically ordered list describing the *state* evolution from time $t = 0$.

5.2 Local Clocks

The clock t is process local and stepped by means of (not equal) time segments. The length of a time segment corresponds to the delay bound to a certain operation or the blocking time of a send or receive operation. The execution of send and receive operations in S is dependent on when data is available for reading or when a channel is free for writing, respectively.

5.3 States

For each vertex, we record during what segments of time computations and communication operations were issued, as well as periods where a core has been stalled due to send- and receive blocking. For each process, a *history* list maps to a state $type \in Stateset$, a start time t_{start} and a stop time t_{stop}. The $state$ of a vertex is a tuple

$$state =< type, t_{start}, t_{stop} >$$

The *StateSet* defines the set of possible state types:

$$StateSet = \{receive, compute, send,\\ blockedreceive, blockedsend\}$$

5.4 Clock Synchronisation

Send and receive are blocking operations. A read operation blocks until data is available on the edge and a write operation blocks until the edge is free for writing. During a time segment only one message can be sent over an edge. Clock synchronisation between communicating processes is managed by means of *event*s. Send and receive operations generate an *event* carrying a time stamp. An edge in G_M^A is implemented using channels having buffer size 1 (forcing write attempts on an occupied link to block), and a simple delay actor. It should be noted that each edge in A needs to be represented by a pair of opposite directed edges in G_M^A to manage synchronization.

```
**receive**($t_{receive}$)
  $t_{available}$ = get next send event from source vertex
  **if**($t_{receive} >= t_{available}$)
    $t_{read} = t_{receive+1}$
    $t_{blocked} = 0$
  **else**
    $t_{read} = t_{available+1}$
    $t_{blocked} = t_{available} - t_{receive}$
  **end if**
  put read event with time $t_{read}$ to source vertex
  **return** $t_{blocked}$
**end**
```

Figure 4. Pseudo-code of the receive function. The get and put operations block if the event queue of the edge is empty or full, respectively.

5.4.1 Synchronised Receive

Figure 4 lists pseudo code of the blocking *receive* function. The value of the input $t_{receive}$ is the present time at which a receiving process issues a *receive* operation. The return value, $t_{blocked}$, is the potential blocking time. The time stamp $t_{available}$, is the time at which the message is available at the receiving core. If $t_{receive}$ is later or equal to $t_{available}$, the core immediately processes the receive operation and sets $t_{blocked}$ to 0. The *receive* function acknowledges by sending a read event to the sender, with the time stamp t_{read+1}. Note that a channel is free for writing as soon as the reciver has begun receiving the previous message. Also note that blocking time, due to unbalanced production and consumption rates, has been accounted for when analysing the timing expression for *send* and *receive* operations, T_s and T_r, as was discussed in Section 2.2. If $t_{receive}$ is earlier than $t_{available}$, the receiving core will block a number of clock cycles corresponding to $t_{blocked} = t_{available} - t_{receive}$.

5.4.2 Synchronised Send

Figure 5 lists pseudo code for the blocking *send* function. The value of t_{send} is the time at which the *send* operation was issued. The time stamp of the read event $t_{available}$ corresponds to the time at which the receiving vertex reads the previous message and thereby also when the edge is available for sending next message. If $t_{send} < t_{available}$, a *send* operation will block for $t_{blocked} = t_{available} - t_{send}$ clock cycles. Otherwise $t_{blocked}$ is set to 0. Note that all edges carrying receive events in the *configuration graph* must be initialised with a read event, otherwise interpretation will deadlock.

```
send($t_{send}$)
  $t_{available}$ = get read event from sink vertex
  if($t_{send} < t_{available}$)
    $t_{blocked} = t_{available} - t_{send}$
  else
    $t_{blocked} = 0$
  end if
  put send event $t_{send} + \Delta_e + t_{blocked}$ to sink vertex
  return $t_{blocked}$
end
```

Figure 5. Pseudo-code of the send function. The value of Δ_e corresponds to the delay of the edge.

5.5 Vertex Interpretation

Figure 6 lists the pseudo code for interpretation of a vertex in G_M^A. It should be noted that, for space reasons, we have omitted to include the state code for global read and write operations. The function $interpretVertex()$ is finitely iterated by each process and the number of iterations, $iterations$, is equally set for all vertices when processes are initated. Each process has a local clock t and an operation counter *op_cnt*, both initially set to 0. The operations sequence S is a process local data structure, obtained from the vertex to be interpreted. Furthermore, each process has a list $history$ which initially is empty. Also, each process has a variable $curr\_oper$ which holds the currently processed operation in S.

The vertex interpreter makes state transitions depending on the current operation $curr\_oper$, the associated delay and whether *send* and *receive* operations block or not. As discussed in Section 5.4.1, the *send* and *receive* functions are the only blocking functions that can halt the interpretation in order to synchronise the clocks of the processes. The value of $t_{blocked}$ is set to the return value of *send* and *receive* when interpreting send and receive operations, respectively. The value of $t_{blocked}$ corresponds to the length of time a *send* or *receive* operation was blocked. If $t_{blocked}$ has a value > 0, a state of type *blocked_send* or *blocked_receive* is computed and added to the *history*.

```
interpretVertex()
  if(list $S$ has elements)
    while($iterations > 0$)
      get element op_cnt in $S$ and put in $curr\_oper$
      increment op_cnt

      if($curr\_op$ is a Receive operation)
        set $t_{blocked}$ = value of receive($t$)
        if($t_{blocked} > 0$)
          add state ReceiveBlocked($t, t_{blocked}$) to $hist.$
          set $t = t + t_{blocked}$
        end if
        add state Receiving($t, \Delta$ of $curr\_oper$)
      end if

      else if($curr\_op$ is a Compute operation)
        add state Computing($t, \Delta$ of $curr\_oper$)
      end if

      else if($curr\_op$ is a Send operation)
        set $t_{blocked}$ = value of send($t$)
        if($t_{blocked} > 0$)
          add state SendBlocked($t, t_{blocked}$) to $hist.$
          set $t = t + t_{blocked}$
        end if
        add state Sending($t, \Delta$ of $curr\_oper$)
      end if

      if(op_cnt reached last index of $S$)
        set op_cnt $= 0$
        decrement $iterations$
        add state End($t$) to $history$
      end if
      set $t = t + \Delta$ of $curr\_oper + 1$
    end while
  end if
end
```

Figure 6. Pseudo-code of the interpretVertex function.

5.6 Model Calibration

We have implemented the abstract interpreter in the Ptolemy software modeling framework [1]. Currently, we have verified the correctness of the interpreter using a set of simple parallel computation problems from the literature. Regarding the accuracy of the model set, we have so far only compared the performance functions separately against corresponding operations on RAW. However, to evaluate and possibly tune the model for higher accuracy we need to do further experimental tests with different relevant signal processing benchmarks, especially including some more complex communication- and memory access patterns.

6 Discussion

We believe that tools supporting iterative mapping and tuning of parallel programs on manycore processors will play a crucial role in order to maximise application performance for different optimization criteria, as well as to reduce the parallel programming complexity. We also believe that using well defined parallel models of computation, matching the application, is of high importance in this matter.

In this paper we have presented our achievements towards the building of an iterative manycore code generation tool. We have proposed a machine model, which abstracts the hardware details of a specific manycore and provides a fine-grained instrument for evaluation of parallel performance. Furthermore, we have introduced and described an intermediate representation called *timed configuration graph*. Such a graph is annotated with computational delays that reflect the performance when the graph is executed on the manycore target. We have demonstrated how we compute these delays using the performance functions included in the machine model and the computational requirements captured in the application model. Moreover, we have in detail demonstrated how performance of a *timed configuration graph* can be evaluated using abstract interpretation.

As part of future work, we need to perform further benchmarking experiments in order to better determine the accuracy of our machine model compared to chosen target processors. Also, we have so far built *timed configuration graph*s by hand. We are especially interested in exploring tuning methods, using feedback information from the evaluator to set constraints in order to direct and improve the mapping of application graphs. Currently we are working on automatising the generation of the *timed configuration graph*s in our tool-chain, implemented in the Ptolemy II software modelling framework.

Acknowledgment

The authors would like to thank Henrik Sahlin and Peter Brauer at the Baseband Research group at Ericsson AB, Dr. Veronica Gaspes at Halmstad University, and Prof. Edward A. Lee and the Ptolemy group at UC Berkeley for valuable input and suggestions. This work has been funded by research grants from the Knowledge Foundation under the CERES contract.

References

[1] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Dept., University of California, Berkeley, Apr 2008.

[2] D. Culler, R. Karp, and D. Patterson. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. of ACM SIGPLAN Symposium on Principles and Practices of Parallel programming*, May 1993.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in stream programs. In *Proc. of Twelfth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 5-10 1974. North-Holland Publishing Company.

[5] R. M. Karp and R. E. Miller. Properties of a Model for Parallel Computations:Determinancy, Termination, Queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[6] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Signal Processing. *IEEE Trans. on Computers*, January 1987.

[7] C. A. Moritz, D. Yeung, and A. Agarwal. SimpleFit: A Framework for Analyzing Design Tradeoffs in Raw Architectures. *IEEE Trans. on Parallel and Distributed Systems*, 12(6), June 2001.

[8] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Dept., University of California, Berkeley, Berkeley, CA, USA, 1995.

[9] H. Sahlin. Introduction and overview of LTE Baseband Algorithms. Powerpoint presentation, Baseband research group, Ericsson AB, February 2007.

[10] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[11] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar Operand Networks. *IEEE Trans. on Parallel and Distributed Systems*, 16(2):145–162, 2005.

Paper E

Manycore Performance Analysis using Timed Configuration Graphs

---

Bengtsson, J. and Svensson, B. (2009). Manycore performance analysis using timed configuration graphs. To appear in *Proc of. Int'l Symp. on Systems, Architectures, Modeling and Simulation (SAMOS IX 2009)*, Samos, Greece.

Manycore Performance Analysis using Timed Configuration Graphs

Jerker Bengtsson
Centre for Research on Embedded Systems
Halmstad University
Halmstad, Sweden
Email: jerker.bengtsson@hh.se

Bertil Svensson
Centre for Research on Embedded Systems
Halmstad University
Halmstad, Sweden
Email: bertil.svensson@hh.se

***Abstract*—The programming complexity of increasingly parallel processors calls for new tools to assist programmers in utilising the parallel hardware resources. In this paper we present a set of models that we have developed to form part of a tool which is intended for iteratively tuning the mapping of dataflow graphs onto manycores. One of the models is used for capturing the essentials of manycores that are identified as suitable for signal processing and which we use as target architectures. Another model is the intermediate representation in the form of a timed configuration graph, describing the mapping of a dataflow graph onto a machine model. Moreover, this IR can be used for performance evaluation using abstract interpretation. We demonstrate how the models can be configured and applied in order to map applications on the Raw processor. Furthermore, we report promising results on the accuracy of performance predictions produced by our tool. It is also demonstrated that the tool can be used to rank different mappings with respect to optimisation on throughput and end-to-end latency.**

I. Introduction

For efficient handling of the programming complexity of manycore processors, *domain specific development tools are needed*. One concrete example is the signal processing required in radio base stations (RBS), which is naturally highly parallel and described by computations on streams of data [1]. Many user channels have to be processed concurrently, each including fast and adaptive coding and decoding of digital signals. Hard real-time constraints imply that parallel hardware, including processors and accelerators is a prerequisite for coping with these tasks in a satisfactory manner.

One candidate technology for building flexible high-performance processing platforms is manycores. However, there are many issues regarding development of complex signal processing software for manycore hardware. One such is the need for tools that reduce the programming complexity and abstract hardware details of a particular manycore processor. We believe that, if industry is to adopt commercial-off-the-shelf (COTS) manycore technology, the application software, the tools and the programming models need to a high degree be portable.

Research has produced specialised compiler techniques for programming languages based on streaming models of computation, achieving good speedup and high throughput for parallel benchmarks [2]. However, even though a compiler can generate optimised code, the programmer is typically left with very little control of how the source program is transformed and mapped on the cores. This means that, if the code output does not meet the non- functional requirements of the system, the only choice is to try to restructure the source program. We argue that in order to increase performance gain experienced application programmers must be able to control the parallel mapping strategy.

We are developing an iterative code mapping tool that allows the programmer to tune a mapping by:

- analysing the result of a parallel mapping using interpreted performance feedback
- giving timing, clustering and core allocation constraints as input to the tool

Figure 1 outlines the modular architecture of our tool. The tool is designed for using well defined dataflow models of computation. One special case of dataflow, synchronous dataflow (SDF), is very suitable for describing signal processing flows [3]. It is also a good source for code generation to parallel hardware, because it has a natural form of parallelism that is a good match to manycores. The programmer provides a manycore machine specification (using our machine model) and the program (using SDF) as input to the tool. During the model analysis stage, the tool will analyse the processing requirements of the SDF model. As the second stage, we compute a static dataflow schedule for the SDF graph (given that the SDF model is consistent). The scheduled graph is then passed through a model transformation. During the model transformation, the tool generates a timed intermediate representation, which represents an abstract mapping of the application on a specific target processor. We call our intermediate representation a *timed configuration graph*.

In this paper we present our achievements on the models and the timed intermediate representation used by the tool to compute performance feedback to the user. The interpreted performance feedback enables a programmer to, early in the development process, explore the run time performance of the software and to find successively better mappings. We believe that this iterative, machine assisted workflow, is advantageous in order to keep the application portable while being able to make trade-offs concerning throughput, latency and compliance with real-time constraint on different platforms. More

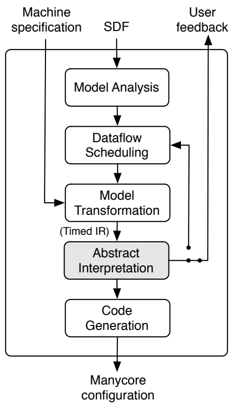


Fig. 1. Outline of the manycore code mapping tool.

specifically, the contributions of this paper are as follows:

- A parallel machine model usable for modelling array-structured, tightly coupled manycore processors. The model is presented in Section III, and in Section V we demonstrate the configuration of it for modelling the Raw processor[4].
- An intermediate representation (IR), used to describe a mapping of an application on a particular manycore in the form of a *timed configuration graph*. The use of this IR is twofold: We can perform an abstract interpretation that gives us feedback of performance during execution of the system. Also, we can use it to generate target code. We present the IR in Section IV.
- We make an evaluation of the accuracy and the usefulness of our tool in Section VI. It is shown that our tool is able to correctly rank different mappings of a graph by highest throughput or shortest end-to-end latency.

We conclude the paper with a discussion of the results of the evaluation and we point to improvements in order to increase the accuracy of some of the predictions.

II. Related Work

The problem of mapping task graphs in the form of acyclic precedence graphs (APG) to a parallel processor has been a widely addressed problem. Heuristic solutions are required since this is for a long time known to be an NP complete problem [5]. Sarkar introduced the two step mapping method, where clustering is performed in a first step independently from the second step of scheduling and processor allocation, which can be applied at compile time [6]. A number of leading algorithms, for both single step and two step clustering and merging, with objectives of transforming and mapping task graphs for multiprocessor systems are reviewed in [7].

The dynamic level scheduling algorithm proposed by Sih and Lee is an heuristic taking inter-processor communication overhead into account during clustering. Similar to our work, this scheduling algorithm can be used to produce feedback to the programmer for iterative refining of the task graph and the architecture [8]. However, it has been demonstrated by Kianzad and Bhattacharyya that two step methods tend to produce more qualitative schedules than single step methods [7]. Unfortunately, expanding an SDF graph to an acyclic precedence graph – which are the assumed representation for many scheduling and mapping algorithms – can lead to an explosion of nodes. This problem can partly be reduced using clustering techniques before the SDF graph is transformed to an APG [9]. However, we are interested in techniques for analysis and mapping of SDF graphs without conversion to an APG

The StreamIt language implements a restricted set of SDF. The StreamIt compiler implements a two phase mapping (dataflow scheduling and clustering, followed by core allocation) using direct representation of SDF graphs [2][4]. However, the StreamIt compiler uses a static and location independent cost model for clustering and core allocation. Further, neither the language nor the compiler provides any means to express non-functional constraints or other application specific optimisation criteria to tune the parallel mapping and code generation. Programs have to be restructured in attempts to improve a mapping.

Throughput is one important non-functional requirement in the real-time applications we are addressing. Ghamarian et al. provide methods for throughput using state space analysis on direct representation of multi-rate SDF graphs [10]. Further, Stuijk et al. have developed a multiprocessor resource allocation strategy for throughput constrained SDF graphs [11]. We are addressing techniques that allow combinations of timing constraints and show how to use them to direct the mapping process.

Bambha and Battacharyya provide a good review of different intermediate representations for different objectives on optimisation and synthesis for self-timed execution of SDF programs to multiprocessor DSP systems [12]. They assume homogenous representation of SDF graphs, which exposes a higher degree of task parallelism based on the rate signatures. Our work is similar, but we are mainly interested intermediate representations on multi-rate SDF and in minimising transformation between different representations during the mapping process.

III. Model Set

In this section we present the model set for constructing *timed configuration graph*s. First we discuss the application model, which describes the application processing requirements, and then the machine model, which is used to describe computational resources and performance of manycore targets.

A. Application Model

We model an application using multi-rate SDF. An SDF graph constitutes a network of actors – atomic or composite of variable granularity – which compute on data distributed via synchronous unidirectional channels. Each channel input and output of an actor has an a priori specified token consumption and production rate. By definition, memory and computations in an SDF graph are distributed, and actors fire (compute) in parallel when there are enough tokens available on the input channels. An SDF graph is computable if there exists at least one periodical repetition schedule. A periodical repetition schedule specifies in which order and how many times each actor fires. If a repetition schedule exists, buffer boundedness and deadlock free execution is guaranteed. One significant advantage with SDF is that the execution order can be determined at compile-time. This enables generation of compact code and elimination of run-time scheduling overhead [13]. The properties of SDF and the formal theory for scheduling of SFD graphs are in detail described in [3].

The Ptolemy modelling framework provides an excellent basis for implementing SDF analysis and code generation tools [14]. Besides serving as input to a code generator, the application model is an executable specification. However, for our work it is not the correctness or the functional properties of the application that is in focus. We are interested in techniques for analysing the non-functional properties of the system. For this we rely on measures like worst case execution time, communication and memory requirements. We assume that these data have been analysed and that each actor is associated with a tuple

$$< r_p, r_m, R_s, R_r >$$

where

- r_p is the worst case execution time, in number of operations.
- r_m is the requirement on memory allocation, in words.
- $R_s = [r_{s_1}, r_{s_2}, ..., r_{s_n}]$ is a sequence where r_{s_i} is the number of words produced on channel i each firing.
- $R_r = [r_{r_1}, r_{r_2}, ..., r_{r_m}]$ is a sequence where r_{r_j} is the number of words consumed on channel j each firing.

B. Machine Model

Scheduling and core allocation algorithms need to take inter processor (core) communication into account to provide realistic cost measures. These costs in general comprise a static cost for sending and receiving and a dynamic cost determined by the resource location and/or the amount of data to be communicated. However, for reasonably near clock-cycle accurate modelling of dynamic network behaviour it is necessary to use a fine grained cost model for communication. We discuss this further in conjunction with our experimental results in Section VI.

One well-studied and reasonably realistic model for distributed memory multiprocessors is LogP [15]. During the past, much work has been done to refine this model, for example taking into account hardware support for long messaging [16], and capturing network contention [17]. A more recent parallel machine model targeting fine-grained and large scale multicores is developed as a part of the SimpleFit framework [18]. SimpleFit considers variable core granularities and requirements on on-chip and off-chip communication. However, it was derived with the purpose of exploring optimal grain size and balance between memory, processing, communication and global I/O, given a VLSI budget and a set of computation problems. Since it is not intended for modelling the dynamic behaviour of a program, it does not include a fine-granular model of the communication. Taylor et al. propose a taxonomy (AsTrO) for comparison of scalar operand networks [19]. This taxonomy includes a five parameter tuple for comparing and evaluating performance sensitivity of on-chip scalar operand networks.

We propose a manycore machine model based on SimpleFit and the AsTrO five parameter tuple. This model allows a fairly fine-grained modelling of performance, including the overhead of operations associated with communication and off-chip resources. The machine model comprises a set of parameters describing the computational resources and a set of abstract performance functions, which describe the performance of computations, communication and memory transactions.

We assume that cores are tightly coupled via a mesh network. Further that each core has individual instruction sequencing capability and that transactions between core private and shared memory is software managed. The resources of such an abstract manycore architecture are described using two tuples, M and F. M consists of a set of parameters describing the resources:

$$M =< (x, y), p, b_g, g_w, g_r, o, s_o, s_l, c, h_l, r_l, r_o >$$

where

- (x, y) is the number of rows and columns of cores.
- p is the processing power (instruction throughput) of each core, in *operations per clock cycle*.
- b_g is global memory bandwidth, in *words per clock cycle*
- g_w is the penalty for global memory write, in *words per clock cycle*
- g_r is the penalty for global memory read, in *words per clock cycle*
- o is software overhead for initiation of a network transfer, in *clock cycles*
- s_o is core send occupancy, in *clock cycles*, when sending a message.
- s_l is the latency for a sent message to reach the network, in *clock cycles*
- c is the bandwidth of each interconnection link, in *words per clock cycle*.
- h_l is network hop latency, in *clock cycles*.
- r_l is the latency from network to receiving core, in *clock cycles*.
- r_o is core receive occupancy, in *clock cycles*, when receiving a message

F is a set of abstract common functions describing the performance of computations, global memory transactions and local communication as functions of M:

$$F(M) =< t_p, t_s, t_r, t_c, t_{gw}, t_{gr} >$$

where

- t_p is a function evaluating the time to compute a sequence of instructions
- t_s is a function evaluating the core occupancy when sending a data stream
- t_r is a function evaluating the core occupancy when receiving a data stream
- t_c is a function evaluating network propagation delay for a data stream
- t_{gw} is a function evaluating the time for writing a stream to global memory
- t_{gr} is a function evaluating the time for reading a stream from global memory

A specific manycore processor is modelled by giving values to the parameters of M and by defining the functions $F(M)$.

IV. Manycore Intermediate Representation

In this section we describe the manycore intermediate representation (IR). We call the IR a *timed configuration graph* because the usage of the IR is twofold:

- Firstly, the IR is a graph representing an SDF program that is transformed and partitioned for a specific manycore target. We can use the IR as input to a code generator, in order to configure each core as well as the interconnection network and plan global memory usage of a specific manycore target.
- Secondly, by introducing the notion of time in the graph, we can use the same IR as input to an abstract interpreter, in order to predict performance and evaluate the dynamic behaviour of the application when executed on a specific manycore target. The output of the evaluator can be used either directly by the programmer or by an auto-tuner for suggesting a better mapping.

A. Relations Between Models and Configuration Graphs

A timed configuration graph $G^A_M(V, E)$ describes a single connected SDF graph A, transformed and mapped on the abstract machine described by the pair of tuples (M, F). The set of vertices is a union $V = P \cup B | P \cap B = \emptyset$, where P is the set of cores and B is the set of off-chip shared memories. We use v_p to denote a vertex of core type and v_b to denote a vertex of memory type. Edges $e \in E$ are dataflow channels mapped onto the interconnection network of (M, F). To obtain a G^A_M, the vertices of A are clustered with respect to the integrity of the dataflow. Each cluster is assigned to a core in M. The edges of the SDF that end up in one cluster are implemented using local memory in the core, so they do not appear as edges in G^A_M. The edges of the SDF that reach between clusters can be implemented in two different ways:

1) as network connection between the two cores. Such connection is represented by an edge (v_{p_i}, v_{p_j}) in G^A_M
2) as a buffer in global memory. In this case, a vertex v_{b_k} is introduced. Further the edge (v_{p_i}, v_{p_j}) is replaced by a pair of edges (v_{p_i}, v_{b_k}) and (v_{b_k}, v_{p_j}) between the two cores in G^A_M.

When G^A_M has been constructed, each $v_p, v_b \in V$ has been assigned costs for computation and communication, calculated using the machine description (M, F) described in Section III-B. These costs reflect the relative costs for each specific operation when computing A on (M, F). We will now discuss how we use A and M to construct and assign costs to the vertices, the edges and the computation costs of G^A_M.

1) Vertices.: Memory vertices, B, allow us to represent a set of buffers mapped in shared memory. A memory vertex can be specified by the programmer, for example to store initial data. Memory vertices can also be automatically generated.

For core vertices, P, we abstract the firing of an actor by means of a sequence S of abstract *receive*, *compute* and *send* operations:

$$S = t_{r_1}, t_{r_2} \dots t_{r_n}, t_p, t_{s_1}, t_{s_2}, \dots, t_{s_m}$$

The cost for a *receive* operation depends on whether the source is another core or a shared memory. Let the source vertex of channel e be $source(e)$. Then for each incoming edge of a vertex p we add a receive operation with a cost bound to:

- $t_r \in F(M)$, if $source(e)$ is of type v_p
- $t_{gr} \in F(M)$, if $source(e)$ is of type v_b

The cost for a *compute* operation is calculated using the performance function t_p, which represents the time required to execute the computations of an actor when it fires.

Finally, for each outgoing edge of a vertex p we add a *send* operation. Let the sink vertex of channel e be $sink(e)$. The *send* operation has a cost bound to:

- $t_s \in F(M)$, if $sink(e)$ is of type v_p
- $t_{gw} \in F(M)$, if $sink(e)$ is of type v_b

Read and write requests on memory vertices are served by the first come first served policy. For a vertex v_b we assign read and write costs calculated using $g_r \in M$ and $g_w \in M$, to account for memory read- and write latencies when serving an incoming request.

When constructing G^A_M, multiple channels sharing the same source and destination can be orderly merged and represented by a single edge, treating them as a single stream of data.

2) Edges.: The weight w of an edge $e(v_i, v_j)$ corresponds to the link propagation. The value of the weight w corresponds to the value of the function $t_c \in F(M)$. Further, edges in SDF can be specified with a sample *delay*. Given an edge $e(v_i, v_j)$, a *unit delay* is defined to mean that the nth sample consumed by v_j corresponds to the $(n-1)$th sample produced by vertex v_i [3]. An edge delay is simply represented by a buffer offset value, needing no further treatment when constructing G^A_M.

Figure 2 shows an example of a simple SDF graph, A, after it has been transformed to one possible G^A_M. One static firing schedule for A in this example is 3(2abcd)e. The schedule should be interpreted as: actor a fires 6 times, actors b, c and d fire 3 times, and actor e 1 time. The firing of A is repeated indefinitely by this schedule. Thus, no runtime scheduling supervision is required. The feedback channel from actor c to actor b is buffered in core local memory. The edge from actor a to actor d is a buffer in shared (off-chip) memory and the others are mapped as point-to-point connections on the network. The integer values represent the send and receive

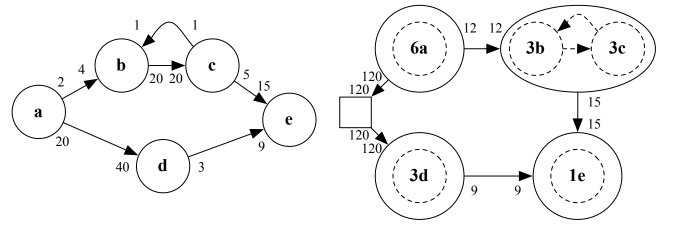


Fig. 2. The graph to the right is one possible graph G_M^A for the application graph A to the left.

rates of the channels (r_s and r_r), before and after A has been clustered and transformed to G_M^A, respectively. Note that these values in G_M^A are the values in A multiplied by the number of times an actor fires, as given by the firing schedule.

B. Interpretation of Timed Configuration Graphs

In order to implement and interpret timed configurations graphs, we need a computational model and a notion of time [20]. We have used dataflow process network (PN) to implement interpretable timed configuration graphs [21]. A process network very well mimics the behaviour of the types of parallel hardware we are studying. The PN domain in Ptolemy is a super set of the SDF domain. The main difference in PN, compared to SDF, is that actors are processes which fire asynchronously. If a process tries to read from an empty channel, it will block until there is new data available. The PN domain implemented in Ptolemy is a special case of Kahn process networks [22]. But, unlike in a Kahn process network, PN channels have bounded buffer capacity, which implies that a process also temporarily blocks when attempting to write to a buffer that is full. This property enables easy modelling of link occupancy on the network.

Each of the core and memory vertices of G_M^A is assigned to its own process. Each of the processes has a local clock, t, which iteratively maps the absolute start and stop time, as well as periods of blocking, to each operation in the sequence S.

Send and receive are blocking operations. A read operation blocks until data is available on the edge and a write operation blocks until the edge is free for writing. Currently, our machine model does not allow modelling of link concurrency. All cores experience the network as a collision free resource. To minimise the risk of providing optimistic performance predictions, we have taken a rather pessimistic approach; only one message is allowed to be sent over an edge during a segment of time, independently of the length of the messages and the network's buffer capacity.

There is no notion of global time in PN. We manage clock synchronisation between the communicating processes by means of communicating *discrete event*s. Send and receive operations generate a discrete event bound to current time. It should be noted that each edge in A needs to be represented by a pair of oppositely directed edges in G_M^A to manage synchronisation. Further, edges in Ptolemy have no ability to perform computations. For each edge, we generate a delay actor, which adds a delay corresponding to the link propagation time ($w \in e \in E$).

V. Modelling the Raw Processor

In this section we demonstrate how we configure the machine model in order to model the Raw processor for performance evaluation [4]. Raw is a tiled, moderately parallel MIMD architecture with 16 (4×4) programmable tiles. Each tile has a MIPS core and is equipped with 32 KB of data and 96 KB instruction caches. The tiles are tightly interconnected via two different types of communication networks: two statically and two dynamically routed.

A. Parameter Settings

We assume a Raw set-up with four off-chip, non-coherent shared memories, and that software managed cache mode is used. Furthermore, we concentrate on modelling the usage of one of the dynamic networks (which are dimension-ordered, wormhole-routed, message-passing types of networks). The parameters of M for Raw with this configuration are set as follows:

$$M =< (4,4), 1, 1, 1, 6, 2, 5, 1, 1, 1, 1, 3 >$$

In our model, we assume a core instruction throughput of p operations per clock cycle. We set $p = 1$. The four shared off-chip DRAMs are connected to four separate I/O ports located on the east-side of the chip. Thus, the DRAMs can be accessed in parallel, each having a bandwidth of $b_g = 1$ words per clock cycle. The latency penalty for a DRAM write is $g_w = 1$ cycle and for a read operation $g_r = 6$ cycles.

The overhead for initiating communication includes sending a header and possibly an address (when addressing any of the off-chip memories). We set the overhead $o = 2$. The four on-chip networks on Raw are mapped to the core's register files, meaning that after a header has been sent, the network can be treated as destination or source operand of an instruction. Ideally, this means that the receive and send occupancy is zero. In practice, when multiple input and output dataflow channels are merged and physically mapped on a single network link, data needs to be buffered locally. We have measured and estimated an average send and receive occupancy to be $s_o = 5$ and $r_o = 3$ respectively. Note that we then also include the overhead for reading and writing via buffers in local memory. The network hop-latency on Raw is $h_l = 1$ cycles per router hop and the link bandwidth is $c = 1$. Furthermore, the send and receive latency is one clock cycle when injecting and extracting data to and from the network: $s_l = 1$ and $r_l = 1$.

B. Performance Functions

The performance functions have been formulated by studying the specification of the Raw processor [23].

a) Compute: The time required to process the fire code of an actor on a core is defined as

$$t_p(r_p, p) = \left\lceil \frac{r_p}{p} \right\rceil$$

which is a function of the requested number of operations r_p and core processing power p. To r_p we count all instructions except those related to network send and receive operations.

b) Send: The time required for a core to issue a network send operation is defined as

$$t_s(R_s, o, s_o) = \left\lceil \frac{R_s}{\mathit{framesize}} \right\rceil \times o + R_s \times s_o$$

Send is a function of the requested amount of words to be sent, R_s, the software overhead $o \in M$ when initiating a network transfer, and a possible send occupancy $s_o \in M$. The framesize is a Raw specific parameter. The dynamic networks allow message frames of length within the interval $[0, 31]$ words. For global memory read and write operations, we use the Raw cache line protocol with $framesize = 8$ words per message. Thus, the first term of t_s captures the software overhead for the number of messages required to send the complete stream of data. For connected actors that are mapped on the same core, we can choose to map channels in local memory (if the local memory capacity is enough). In that case we set t_s to zero.

c) Receive: The time required for a core to issue a network receive operation is defined as

$$t_r(R_r, o, r_o) = \left\lceil \frac{R_r}{\mathit{framesize}} \right\rceil \times o + R_r \times r_o$$

d) Network Propagation: Providing means for modeling communication accurately for an abstract parallel target is difficult: high accuracy requires the use of a low machine abstraction level. We chose the approach of modeling communication as collision free.

In the network propagation time, we consider a possible network injection and extraction latency at the source and destination in addition to the link propagation time. The network propagation time is defined as

$$t_c(R_s, x_s, y_s, x_d, y_d, s_l, h_l, r_l) =$$
$$s_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d) + r_l$$

Network injection and extraction latency is captured by s_l and r_l respectively. Further, the propagation time depends on the network hop latency h_l and the number of network hops $d(x_s, y_s, x_d, y_d)$, which is a distance function of the source and destination coordinates. Routing turns add an extra cost of one clock cycle. This is captured by the value of $n_{turns}(x_s, y_s, x_d, y_d)$ which, similar to d, is a function of the source and destination coordinates.

e) Streamed Global Memory Read: Reading from global memory on the Raw machine requires first one send operation (the core overhead which is captured by t_s), in order to configure the memory controller and set the address of memory to be read. The second step is to issue a receive operation to receive the memory contents on that address. The propagation time when streaming data from global memory to the receiving core is defined as

$$t_{gr}(r_l, x_s, y_s, x_d, y_d, h_l) =$$
$$r_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d)$$

Note that memory read latency penalty is not included in this expression. This is accounted for in the memory model included in the IR (G_M^A).

f) Streamed Global Memory Write: Like the memory read operation, writing to global memory requires two send operations: one for configuring the memory controller (set write mode and address) and one for sending the data to be stored. The time required for streaming data from the sending core to global memory is evaluated by

$$t_{gw}(s_l, x_s, y_s, x_d, y_d, h_l) =$$
$$s_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d)$$

Like in stream memory read, the memory write penalty is accounted for in the memory model.

VI. Experimental Evaluation

In this section we present an evaluation of our tool with two purposes:

- to evaluate the accuracy of the tool's performance predictions with respect to actual performance.
- to investigate whether the predictions can be used to rank different mappings of an application with respect latency and throughput.

We have selected two applications with different relations between communication and computation demands to evaluate the accuracy and sources of possible inaccuracy. For the Raw implementations, we have used BEETLE, which is a cycle-accurate Raw simulator.

A. *Matrix Multiplication*

Our first case study is matrix multiplication, which requires fairly large amounts of data to be communicated over the network. Furthermore, it provides an excellent case for testing the tool on large amounts of communication between the cores and global memory. The input matrices are partitioned into overlapping sub-matrices and the computations are distributed equally on four cores. Thus there is no exchange of data between the cores. Both the input matrices and the result are stored in off-chip memory. Figure 3 shows three different mappings of a 32×32 matrix multiplication used in the experiments. Note that we kept the algorithm the same in all three cases.

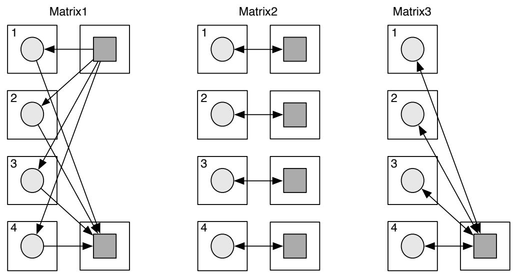


Fig. 3. Three different mappings of the 32×32 elements matrix multiplication using four cores.

In Matrix1, all cores read their assigned input data from the upper memory bank and store the result in the lower memory bank. In Matrix2, we assume that the input data has been arranged and distributed over four separate banks. Thus, in this case, each core has collision-free access to the network and off-chip memory. Finally, in Matrix3, input and output data are all stored in the same memory bank.

We expect the performance prediction for Matrix2 to be more accurate than the predictions for Matrix1 and Matrix3 since our model assumes a collision-free network. Furthermore, by comparing the predictions for Matrix1 with Matrix2 and Matrix3, we expect to get an indication of how sensitive the prediction accuracy is to contention effects. The main difference between Matrix1 and Matrix3 is that, in Matrix3, all communication to the off-chip memory controller is using the same network links. In short, we expect there to be fewer collisions in Matrix1 compared with Matrix3, but the performance should still be relatively close to the performance of Matrix3. This further provides an interesting test case to evaluate whether the tools predictions can be used to determine which mapping performs better.

B. Parallel Merge Sort

Our second case study is merge sort. Compared to matrix multiplication, the merge sort algorithm has very low requirements on computation and communication. Figure 4 shows two different mappings of the merge sort algorithm using 7 and 5 cores, respectively. The computation and communication load, for each vertex in the tree, increases with the level as the tree narrows down. Each vertex in the tree consumes a sorted sub-list from preceding nodes via two channels and produces a merged sorted output. The input data is distributed over the leaf vertices, and the result, a sorted list, is stored locally in the root vertex. In the first of the mappings (called Merge) each of the vertices is mapped to one core. This mapping is illustrated to the left in Figure 4. In the second mapping (called Merge fused, shown to the right in Figure 4), the four leaf vertices have been pair-wise clustered in order to obtain an improved load and communication balance compared to Merge.

C. Accuracy of Predicted Core Communication Costs

In the first experiment, we have studied the accuracy of the predicted performance on send and receive operations.

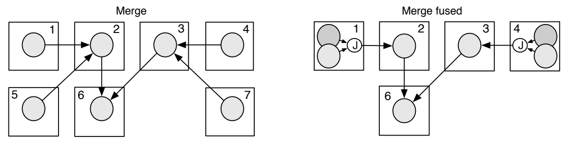


Fig. 4. The graph to the left is a fully parallel mapping of the merge sort (denoted Merge) and in the graph to the right, leaf nodes have been pair-wise clustered and mapped to the same core. The smaller node denoted J, in core 1 and 4, symbolise a join operation performed on the output channels.

For the applications used in the experiments, the programs generated for each core consist of a receive phase, followed by a compute, and then a send phase. We use Raw_{mm} to denote predicted performance (using our tool) and Raw to denote the performance measured on Raw. All predictions and corresponding measurements are made during steady state execution of the dataflow graphs.

Table I shows the predicted receive times, for each used core, compared to the measured receive times for Matrix1, Matrix2 and Matrix3 respectively. The receive time includes possible read blocked time. For each of the three test cases it can be seen that the predicted receive times are slightly pessimistic (which is preferred compared to optimistic). The differences between the predicted and the measured receive times vary between $+2,3\%$ and $+12,6\%$.

In Matrix1, cores 1 and 2 have shorter distance than 3 and 4 to the memory holding the input, which leads to lower read blocking time in the Raw measurements. Because the timed configuration graph views the network as a collision free resource, the receive performance is evaluated more fairly for all cores in Raw_{mm}. Similarly, in Matrix3, cores 3 and 4 have shorter distance to off-chip memory than cores 1 and 2. However, in Matrix3 the unfairness in distance to memory has less importance. Since both read and write request have to compete for the same physical links on the network, the read and write blocking becomes more fairly distributed on the cores.

In the Matrix2 mapping there are no collisions. The main reason for the pessimistic predictions (9,5%) is that we have used averaged measures to configure the send and receive occupancy for Raw. We can probably to some extent tune these parameters to get slightly better accuracy. However, to get a fully accurate prediction we would need to model execution at instruction-level, which would be very costly in terms of modelling performance.

Table II shows the predicted send times compared to the measured send times for Matrix1, Matrix2 and Matrix3. As can be seen, for all three mappings the predicted send time using Raw_{mm} is accurate compared to the measured send time on Raw. The unfairness in distance from the off-chip input memory forces a relative skew between cores during execution (as later explained in section VI-D). Moreover, the send phase comprises much fewer messages to be sent, compared to the receive phase: there are simply no (or very few) collisions during send.

TABLE I
MATRIX STEADY STATE RECEIVE TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Matrix1	1	1790	1589	+12,6%
	2	1790	1589	+12,6%
	3	1790	1750	+2,3%
	4	1790	1750	+2,3%
Matrix2	1	1600	1461	+9,5%
	2	1600	1461	+9,5%
	3	1600	1461	+9,5%
	4	1600	1461	+9,5%
Matrix3	1	1828	1701	+7,5%
	2	1814	1626	+11,6%
	3	1800	1716	+4,9%
	4	1786	1716	+4,1%

TABLE II
MATRIX STEADY STATE SEND TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Matrix1	1	408	408	0%
	2	408	408	0%
	3	408	408	0%
	4	408	408	0%
Matrix2	1	408	408	0%
	2	408	408	0%
	3	408	408	0%
	4	408	408	0%
Matrix3	1	408	408	0%
	2	408	408	0%
	3	408	408	0%
	4	408	408	0%

We will now discuss our corresponding experiment on send and receive times for the merge sort application. In this experiment only core-to-core communication is utilised and the communication consists of very small messages (1 to 4 words). Furthermore, we have deliberately designed one of the mappings (Merge) to force unbalanced core communication and computation loads. This experiment is expected to give an indication on how accurately Raw_{mm} models short messaging and unbalanced communication. The predicted send times compared to the measured ones can be seen in Table III. For Merge, the predicted times are exact or very accurate. Cores 1,2,4, and 7 compute the leaf vertices, which also generate the input in the parallel merge tree. Thus, no receive operations are issued by these cores. However, for Merge fused, we see that Raw_{mm} has evaluated the receive time 75% higher, compared to the measurements on Raw for cores 2 and 3. The reason is that the computation times for cores 2 and 3, after the clustering, are now shorter than for the preceding leaf vertices. Since Raw_{mm} models communication pessimistically – in the sense that we only allow one message at a time on a network link – communication is tighter synchronised in our model. This can introduce blocking times in communication between cores with unbalanced workloads, which are not experienced on Raw.

In Table IV we compare send times for the two different mappings of the merge sort algorithm. As can be seen in the table, the predicted send times are fairly close to the measured times (the difference is 9,1% or less).

TABLE III
MERGE STEADY STATE RECEIVE TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Merge	1	0	0	+0%
	2	16	16	+0%
	3	16	16	+0%
	4	0	0	+0%
	5	0	0	+0%
	6	29	28	+3,6%
	7	0	0	+0%
Merge fused	1	0	1461	+9,5%
	2	42	24	+75%
	3	42	24	+75%
	4	0	0	+0%
	5	0	0	+0%
	6	29	28	+3,6%
	7	0	0	+0%

TABLE IV
MERGE STEADY STATE SEND TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Merge	1	85	79	+7,6%
	2	22	22	+0%
	3	22	22	+0%
	4	85	79	+7,6%
	5	85	79	+7,6%
	6	0	0	+0%
	7	85	79	+7,6%
Merge fused	1	24	22	+9,1%
	2	22	22	+0%
	3	22	22	+0%
	4	24	22	+9,1%
	5	0	0	+0%
	6	0	0	+0%
	7	0	0	+0%

D. Latency and Throughput Measurements

In the second part of the experiments, we have used the same mappings to compare predicted and measured end-to-end-latency and throughput. We also evaluate whether the predictions, despite potential inaccuracy, can be used to rank the mappings correctly with respect to shortest latency and highest throughput.

The mappings are self-timed, meaning that synchronisation is handled at run-time [24]. Initially, a self-timed graph executes a non-steady state and later, after a number of iterations, converges to a steady-state schedule.

Figure 5 illustrates the dynamic behaviour for the self-timed mapping of the Merge application. Cores computing the upstream actors in the dataflow graph with lower workload finish faster and can proceed with the next iteration of the schedule, as long as the network buffer is large enough to store the produced data. As shown in the figure, core 1 has started its fourth iteration when core 6 begins computing its first iteration. When network buffers are full, a steady state execution is naturally forced.

Figure 6 shows the predicted latencies (in clock cycles), for Merge and Merge fused, compared to the measured latencies

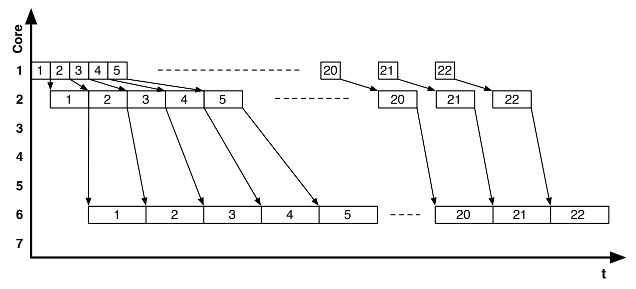


Fig. 5. Skewing experienced in the unbalanced Merge algorithm. The numbers represent the firing count of each actor, and the distance in time between the firings is dependent on the network buffer capacity.

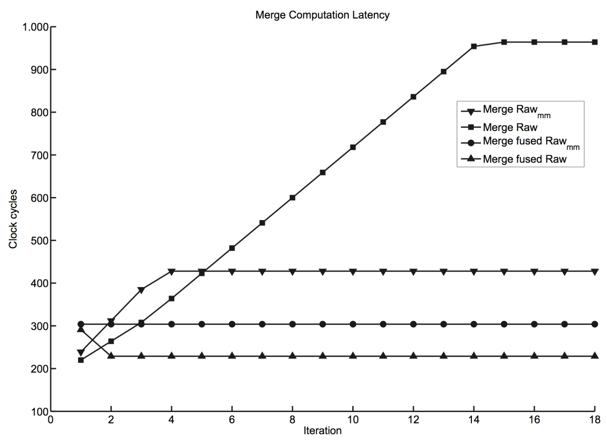


Fig. 6. Comparison of predicted (Raw$_{\text{mm}}$) and measured (Raw) end-to-end latency for Merge and Merge fused.

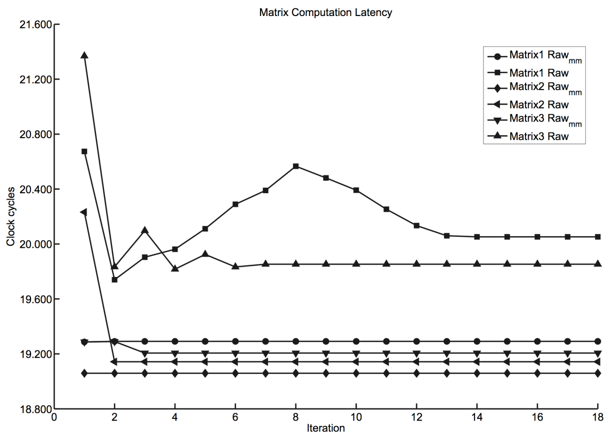


Fig. 7. Comparison of the modelling accuracy of the computation latency of three different mappings of the parallel matrix multiplication.

as a function of the current iteration. The figure shows at which iteration each of the mappings reaches a steady state of execution, i.e. when the latency curve levels out. We see that, for Merge, the measured latency is underestimated by a factor of 2. This is explained by the fact that the machine model is currently not able to model buffer capacity of the on-chip network. Thus, the difference in iteration count between the first upstream actor and the last actor in the graph is larger on Raw than in the modelled execution of Raw. To tighten the latency predictions for graphs with unbalanced communication, we need to account for network buffer capacity in the machine model.

For Merge fused, we see that the latency has rather been overestimated, but is closer to the measured latency. The reason is that both the workload and the communication in Merge fused is better balanced than in Merge (after clustering core 1 with 5 and core 3 with 7), which forces Merge fused to reach a steady state after fewer iterations.

If we rank the predicted latencies of Merge and Merge fused, even if the predictions have varying accuracy, we still see that an optimisation decision based on the predictions would (for this case) correctly identify Merge fused as the better mapping.

Figure 7 shows the predicted end-to-end latencies for Matrix1, Matrix2 and Matrix3, compared to the measured latencies on Raw. We see that the different mappings of the matrix multiplication converge to steady state at different numbers of iterations. Unlike in the merge sort experiment, the computation tasks distributed on the cores is naturally load balanced. The reason that the different implementations of the matrix multiplication reach steady state at different points in time is that the cores used in the different mappings are affected by different communication delays due to network contention. Contention effects is a large contributing factor causing an underestimate of the latencies for Matrix1 and Matrix3. This can be verified by observing that the plot for Matrix2 on Raw$_{\text{mm}}$ and Raw (which is a contention free mapping), is fairly accurate compared to the predictions for Matrix1 and Matrix3. However, if we rank the predicted steady state latencies for all mappings, we see that an optimisation decision based on latency minimisation would in this case correctly suggest Matrix3 better than Matrix 1 and Matrix2 as the best alternative of the three.

Table V shows the predicted and the measured throughputs for Merge (with 4,4% difference) and Merge fused (with 10% difference). The predictions are fairly close to the measurements on Raw for both Merge and Merge fused. We also see that both the predicted and the measured throughputs show that Merge has a higher throughput than Merge fused. When optimising for throughput, our predictions correctly rank Merge better than Merge fused.

Finally, Table VI shows the corresponding comparisons for Matrix1, Matrix2 and Matrix3. Note that, unlike in all the other experiments, our model has predicted slightly optimistic throughputs. However, if we rank both the predicted throughputs and the measured throughputs, we see that the predictions will be ranked in the same order as for the measured ones. Thus, if using the predictions for throughput optimisation, our tool finds the best cases for this example as well.

VII. Conclusion

In this paper we have presented our achievements on building an iterative manycore code mapping tool. In order

TABLE V
MERGE STEADY STATE PERIODICITY (CLOCK CYCLES)

Application	Raw_{mm}	Raw	diff
Merge	119	104	+4,4%
Merge fused	132	120	+10%

TABLE VI
MATRIX STEADY STATE PERIODICITY IN (CLOCK CYCLES)

Application	Raw_{mm}	Raw	diff
Matrix1	19249	19434	-0,9%
Matrix2	19059	19143	-0,4%
Matrix3	19248	19401	-0,8%

to provide estimates of performance, we have developed a machine model which abstracts a certain category of manycore architectures. We model the applications using synchronous dataflow, and the performance estimates are computed using an executable intermediate representation called *timed configuration graph*.

We have presented an evaluation in terms of the prediction accuracy of our tool and whether the predictions can be used to identify a better mapping. It is shown that communication times between cores are predicted slightly pessimistic, still fairly close to measured performance, with respect to the high level of modelling. Our comparisons indicate that, for the small set of mappings so far explored in the experiments, the tool can correctly rank different mappings with respect to highest throughput or shortest latency. However, the comparisons also reveal that the predictions of end-to-end latency for graphs with unbalanced communication can be quite inaccurate. This was demonstrated to mainly depend on the high abstraction level of on-chip communication implemented by the IR, which currently does not capture the buffer capacity or link concurrency of the network.

To increase the accuracy and the reliability of end-to-end latency measurements on dataflow graphs, we plan to investigate inclusion of network buffer capacity and modelling link concurrency in the intermediate representation. We are especially interested in exploring automatised tuning methods, using feedback information from the abstract interpreter, in order to direct and improve the mapping of application graphs.

ACKNOWLEDGEMENT

The authors would like to thank Dr. Veronica Gaspes at Halmstad University and the anonymous reviewers whose comments helped improving this paper. This work has been funded by research grants from the Knowledge Foundation under the CERES contract.

REFERENCES

[1] E. Dahlman, S. Parkvall, J. Skold, and P. Beming, *3G Evolution: HSPA and LTE for Mobile Broadband*, 2nd ed. Academic Press, 2008.

[2] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proc. of Twelfth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 152–162.

[3] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Signal Processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24–35, January 1987.

[4] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[5] H. El-Rewini, H. Ali, and T. Lewis, "Task Scheduling in Multiprocessing Systems," *IEEE Computer*, vol. 28, no. 12, pp. 27–37, Dec 1995.

[6] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[7] V. Kianzad and S. Bhattacharyya, "Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 667–680, July 2006.

[8] G. C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," Ph.D. dissertation, EECS Department, University of California, Berkeley, CA 94720, USA, April 1991.

[9] J. L. Pino and E. A. Lee, "Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors," in *Proc. of IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, 1995, pp. 2643–2646.

[10] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij, "Throughput Analysis of Synchronous Data Flow Graphs," *Proc. of Int'l Conf. on Application of Concurrency to System Design*, pp. 25–36, 2006.

[11] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal, "Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs," in *Proc. of the 44th annual conf. on Design automation*. New York, NY, USA: ACM, 2007, pp. 777–782.

[12] N. Bambha, "Intermediate Representations for Design Automation of Multiprocessor DSP Systems," in *Design Automation for Embedded Systems*. Kluwer Academic Publishers, 2002, pp. 307–323.

[13] S. S. Battacharyya, "Optimization Trade-offs in the Synthesis of Software for Embedded DSP," in *Workshop on Compiler and Architecture Support for Embedded Systems*, Washington, D.C, 1999.

[14] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)," EECS Dept., University of California, Berkeley, Tech. Rep. UCB/EECS-2008-28, Apr 2008.

[15] D. Culler, R. Karp, and D. Patterson, "LogP: Towards a Realistic Model of Parallel Computation," in *in Proc. of ACM SIGPLAN Symp. on Principles and Practices of Parallel programming*, May 1993, pp. 1–12.

[16] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. J. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation." in *Proc. of the seventh annual ACM Symp. on Parallel Algorithms and Architectures*, 1995, pp. 95–105.

[17] C. A. Moritz and M. I. Frank, "LoGPC: Modeling Network Contention in Message-Passing Programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, no. 4, pp. 404–415, 2001.

[18] C. A. Moritz, D. Yeung, and A. Agarwal, "SimpleFit: A Framework for Analyzing Design Tradeoffs in Raw Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 730–742, June 2001.

[19] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar Operand Networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 145–162, 2005.

[20] J. Bengtsson, "A Set of Models for Manycore Performance Evaluation Through Abstract Interpretation of Timed Configuration Graphs," School of IDE, Tech. Rep. IDE0856, 2008.

[21] T. M. Parks, "Bounded Scheduling of Process Networks," Ph.D. dissertation, EECS Department, University of California, Berkeley, CA 94720, USA, 1995.

[22] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of IFIP Congress 74*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North-Holland Publishing Company, August 5-10 1974, pp. 471–475.

[23] M. B. Taylor, "The Raw Processor Specification," CSAIL, MIT, Cambridge, MA, Tech. Rep., 2003.

[24] E. Lee and S. Ha, "Scheduling Strategies for Multiprocessor Real-time DSP," in *Proc. of IEEE Glob'l. Telecomm. Conf., 1989, and Exhibition. Communications Technology for the 1990s and Beyond.*, Nov 1989, pp. 1279–1283 vol.2.