HALMSTAD
UNIVERSITY

Tools to Compile Dataflow Programs for Manycores

Essayas Gebrewahid

Supervisors:
Zain Ul-Abdin
Veronica Gaspes
Bertil Svensson

DOCTORAL THESIS | Halmstad University Dissertations no. 33

Tools to Compile Dataflow Programs for Manycores

Halmstad University Dissertations no. 33
ISBN 978-91-87045-68-4 (printed)
ISBN 978-91-87045-69-1 (pdf)
Publisher: Halmstad University Press, 2017 | www.hh.se/hup
Printer: Media-Tryck, Lund

Tools to Compile Dataflow Programs for Manycores

Abstract

The arrival of manycore systems enforces new approaches for developing applications in order to exploit the available hardware resources. Developing applications for manycores requires programmers to partition the application into subtasks, consider the dependence between the subtasks, understand the underlying hardware and select an appropriate programming model. This is complex, time-consuming and prone to error.

In this thesis, we identify and implement abstraction layers in compilation tools to decrease the burden of the programmer, increase program portability and scalability, and increase retargetability of the compilation framework. We present compilation frameworks for two concurrent programming languages, `occam-pi` and `CAL Actor Language`, and demonstrate the applicability of the approach with application case-studies targeting these different manycore architectures: STHorm, Epiphany, Ambric, EIT, and ePUMA.

For `occam-pi`, we have extended the Tock compiler and added a backend for STHorm. We evaluate the approach using a fault tolerance model for a four stage 1D-DCT algorithm implemented by using `occam-pi`'s constructs for dynamic reconfiguration, and the FAST corner detection algorithm which demonstrates the suitability of `occam-pi` and the compilation framework for data-intensive applications. For `CAL`, we have developed a new compilation framework, namely Cal2Many. The Cal2Many framework has a front end, two intermediate representations and four backends: for a uniprocessor, Epiphany, Ambric, and a backend for SIMD based architectures. Also, we have identified and implemented of `CAL` actor fusion and fission methodologies for efficient mapping `CAL` applications. We have used QRD, FAST corner detection, 2D-IDCT, and MPEG applications to evaluate our compilation process and to analyze the limitations of the hardware.

Acknowledgments

First of all I would like to express my gratitude for my supervisors Zain-ul-Abdin, Veronica Gaspes, and Bertil Svensson. Thanks for the support, guidance and encouragement during the course of the work. I would also like to thank Tomas Nordström and my support committee members Mohammad Mousavi and Jorn W Janneck for their valuable comments and support. I thank Stefan Byttner for his support as a director of studies.

I want to give special thanks to Süleyman, Sebastian, Amin and Erik for the friendship and discussions. I would also like to thank my colleagues at HiPEC and STAMP project for the teamwork. I would also like to thank fellow PhD students at HRSS and colleagues at IDE for providing a friendly work environment.

Finally, I am thankful to my family and friends.

This work has been financially supported by grants from the Foundation for Strategic Research through the HiPEC project, from the European Union through the SMECY project, from the ELLIIT initiative through the STAMP project, as well as by Halmstad University. I am thankful for all.

List of Publications

The thesis summarizes the following papers.

A. Z. Ul-Abdin, E. Gebrewahid, and B. Svensson "Managing dynamic reconfiguration for fault-tolerance on a manycore architecture." In *Proceedings of the 19th International Reconfigurable Architectures Workshop (RAW'12) in conjunction with International Parallel and Distributed Processing Symposium (IPDPS'12)*, Shanghai, China, May 2012, pp. 312-319.

Contribution: I have developed and added the STHorm aka the P2012 platform backend to the `occam-pi` compilation framework (Tock). Based on the first author's idea, I have implemented the expression of fault-recovery mechanisms based on dynamic reconfiguration of the STHorm architecture. I have contributed text for compilation of `occam-pi` to STHorm, experimental case-study, and analysis of the result.

B. E. Gebrewahid, Z. Ul-Abdin, B. Svensson, V. Gaspes, B. Jego, B. Lavigueur, and M. Robart, "Programming real-time image processing for manycores in a high-level language." In *Proceedings of the 10th International Symposium on Advanced Parallel Processing Technologies (APPT)*, Stockholm, Sweden, August 2013, Springer Berlin Heidelberg, pp. 381-395.

Contribution: I have revised the frontend of the Tock framework and the STHorm backend to provide competitive support for data-intensive applications. For the experimental case study, I have implemented the FAST corner detection algorithm in `occam-pi`. I have led the writing of the paper.

C. E. Gebrewahid, M. Yang, G. Cedersjö, Z. Ul-Abdin, J. W. Janneck, V. Gaspes, and B. Svensson, "Realizing efficient execution of dataflow actors on manycores." In *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, Milan, Italy, August 2014, pp. 321-328.

Contribution: I have designed and added an intermediate representation and backend for GP-CPU, Epiphany and Ambric to a new `CAL` compilation

framework. named Cal2Many. The input to the compilation framework is a machine model for CAL actors, developed by J. W. Janneck and G. Cedersjö. I have led the writing of the paper.

D. S. Savas, E. Gebrewahid, Z. Ul-Abdin, T. Nordström and M. Yang, "An evaluation of code generation of dataflow languages on manycore arhictectures." In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Chongqing, China, August 2014, pp. 1-9.

Contribution: The first author and I have used 2D-IDCT as an experimental case study to evaluate the Epiphany code generation from CAL, using a hand-written implementation as baseline. Based on the evaluation, I have implemented three different optimizations for Epiphany code generation. The first author and I have led the writing of the paper.

E. E. Gebrewahid, M.A. Arslan, A. Karlsson, and Z.Ul-Abdin. "Support for data parallelism in the CAL actor language." In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, p. 2. ACM, 2016.

Contribution: I have revised the Cal2Many framework to enable support for SIMD operations and data types. I have added a SIMD backend to generate a Target Specific Language that is then used to program the EIT and ePUMA architectures. I have led the writing of the paper.

F. S. Savas, S. Raase, E. Gebrewahid, Z. Ul-Abdin, and T. Nordström. "Dataflow implementation of QR decomposition on a manycore." In *Proceedings of the Third ACM International Workshop on Many-core Embedded Systems* (pp. 26-30). ACM, 2016.

Contribution: The first three authors have implemented three QR decomposition algorithms (Givens Rotations, Householder and Gram-Schmidt) in both CAL and C languages. These were executed on the Epiphany architecture to evaluate the CAL2Many and the algorithms performance, scalability and development effort. I have contributed text for the Gram-Schmidt QRD implementation, for the CAL code generation, and the discussions on result section.

G. E. Gebrewahid, Z. Ul-Abdin, V. Gaspes, "Cal2Many: A framework to compile dataflow programs for manycores." *A manuscript for journal publication.*

Contribution: The paper focuses on the Cal2Many framework and the Epiphany manycore architecture. In this paper, I have performed an in-depth analysis of an MPEG4-SP implemented on the Epiphany, studied the effects of actor composition, and identified the limitations in the Epiphany architecture. I have led the writing of the paper.

H. E. Gebrewahid, Z. Ul-Abdin, "Actor Fission Transformations for Executing Dataflow Programs on Manycores." In *Proceedings of the 2017 Forum on specification & Design Languages*. 2017.

Contribution: The paper presents actor replication, actor decomposition, and `CAL` action decomposition transformations. I have implemented the actor fission techniques in the Cal2Many framework and analyzed their practicality and feasibility. I have led the writing of the paper.

Contents

List of Figures

Chapter 1
Introduction

1.1 Motivation

Nowadays, embedded manycores are used for various digital signal processing (DSP) applications such as radar signal processing, video/audio processing, and computer vision. Achieving high performance in a single processor operating at higher frequency requires high power and results in more heat generation, while multiple processors clocked at a lower frequency running in parallel require less power and generate less heat while sustaining similar performance. This makes parallelism a feasible way to achieve high performance and continue to take advantage of Moore's law [15]. However, to take full advantage of the advances in the hardware, programmers often have to redesign their programs—"the free lunch is over" [35].

In single-core platforms, application developers focus on writing correct code and rely on the compiler for efficient code generation. For manycore architectures, due to limited tool support, developers usually have to perform extra work to assist the compilation process. This extra work may include *decomposition*—partitioning of the overall application into small tasks, *mapping*—assigning execution engine for the task, *scheduling*—deciding when the task should run, and *low-level coding*—hand-tuning the generated code even at assembly level to exploit hardware-specific features. Also, porting an application to another platform requires restructuring the program and changing the hardware-specific codes. Embedded manycores are specialized for diversified application areas which result in a variety of proprietary programming tools. Also, the amount and number of different types of cores are increasing. Programming and porting code to different manycores without the help of mature compilation tools is time-consuming, prone to error and results in less portable applications.

One approach to deal with these difficulties is to use a high-level programming language that provides means to explicitly express the parallelism in the application and to delegate the complexity of exploiting a parallel architecture

to a compilation tool. This approach separates application developers from low-level hardware details by using high-level of abstractions provided in the programming languages.

The hardware could support instruction-, data- and task-level parallelism [14]. *Instruction level parallelism* can be exposed by pipelining, where multiple instructions are executed simultaneously in a single clock cycle. In *data level parallelism* the same instruction is performed on multiple data items simultaneously, i.e., Single Instruction Multiple Data (SIMD). In *task level parallelism* multiple tasks are executed in parallel using multiple processors, i.e., Multiple Instruction Multiple Data (MIMD). Instruction parallelism and fine-grained data parallelism can be exploited within a processor by an efficient schedule that considers the dependencies between the instructions and the utilization of data. On the other hand, coarse-grained data- and task-level parallelism are usually exploited by executing the instruction streams on a number of processors. In addition, embedded manycores usually have dedicated hardware support for a specific application area at the cost of decreased generality and flexibility. For example, manycores that target multimedia processing often support data parallelism through a SIMD capable instruction set.

This thesis mainly presents Cal2Many: a compilation framework to compile the CAL Actor Language (CAL) for manycores. Also, this thesis presents our contribution to Tock, [1] a framework to compile `occam-pi`. Both languages are based on the dataflow programming model. Actor-oriented dataflow programming has gained acceptance for streaming applications since it reflects the inherent parallelism in DSP applications; additional examples include Erlang [3], SALSA [40] and E-language [12]. The dataflow model of `CAL` and `occam-pi`, and their compilation frameworks provide three key features:

- **Portability** is the potential reusability of code. `CAL` and `occam-pi` provide high-level abstractions for explicit parallelism, which increases portability by avoiding target-dependent abstractions and low-level issues such as synchronization, data locality management, and race conditions.

- **Scalability** is the potential of an application to cope with the change in the number of processing units. `Occam-pi` processes and `CAL` actors are encapsulated modular entities that could be composed to run on a few processing units. `CAL` actors can be decomposed and replicated to run on a large number of cores. The Cal2Many framework enables scalability by utilizing composition, decomposition and replication transformation. `Occam-pi` enables scalability by supporting dynamic process creation, invocation, and reconfiguration using *FORK* and *FORKING* keywords.

- **Retargetability** is the potential of customizing the framework to target and generate code for different architectures. The Tock and Cal2Many frameworks perform target independent transformations in the frontend and intermediate representation and exploit the particular features of the

underlying parallel hardware in the backend. This eases customizations and enables retargetability.

1.2 Problem Statement

Recent advances in hardware with the emergence of manycores poses a challenge on programmers and compilation tools in order to exploit the available resources. The problem statement of this thesis is: *to identify and implement abstraction layers in compilation tools to decrease the burden of the programmer, increase programming productivity, scalability and program portability for manycores and to analyze their impact on performance and efficiency.*

1.3 Research Approach

Our approach is to use a high-level programming language that provides means to express the parallelism in the application and to develop a compilation framework for programming embedded manycore architectures to reduce the effort and the involvement of application developers without compromising the performance of the resulting implementation. The papers in this thesis present a compilation framework for `CAL Actor Language` and `occam-pi`. For `occam-pi`, we have extended the Tock [1] compiler, and for `CAL` we have developed a new compilation framework. Both compilation frameworks bridge the gap between the language and the architectures, and increase programming productivity and program portability. Both frameworks generate code in the proprietary language of the target architecture and can leverage the native development tools to generate efficient machine code.

1.4 Contribution

This thesis contributes to the possibility of programming manycores using high-level languages by implementing compilation layers that provide support for high-level transformations and for exploiting the underlying parallel hardware. We express the inherent parallelism of applications using `occam-pi` and `CAL` and map them onto manycore architectures by using the Tock and the Cal2Many frameworks, respectively. More specifically, the contributions are:

- Identification and implementation of abstraction layers of compilers for manycores **[Paper B, C, G]**.
  - We have added a backend for STHorm [6] to the Tock [1] compiler and programmed STHorm using `occam-pi` **[Paper B]**.
  - We have contributed to the development of a new compilation framework for the `CAL Actor Language`.

We have developed an intermediate representation and added backends for the Ambric [8] and Epiphany [28] manycore architectures **[Papers C, G]**.

- Identification and implementation of actor fusion and fission for efficient mapping **[Paper H]**.

  We identify and implement methodologies for actor fusion and fission with descriptions for conditions under which these methodologies give a valid result.

- Realization of data-parallelism for data-intensive applications **[Papers B, E]**.

  We have revisited the Tock frontend and the STHorm backend to provide support for channels that communicate an entire array of data in a single transfer and to access low-level STHorm APIs **[Paper B]**.

  We have extended the compilation process of `CAL` to support vector operations **[Paper E]**.

- Evaluation and case studies.

  We have used QRD, FAST corner detection, 2D-IDCT, and MPEG applications to evaluate our compilation process and to analysis the limitations of the hardware **[Papers A-H]**.

Chapter 2
Background: Manycore Architectures and Applications

2.1 Manycore Architectures

Nowadays, manycores are used in high performance computing systems to execute computationally demanding applications in various areas, e.g. e scientific computing, signal processing, imaging and audio/video processing. In this thesis, we have experimented on commercial embedded manycore architectures suitable for coarse-grained task parallel applications, and two academic vector processing architectures. In addition to the ones used in this thesis, examples of commercial manycores include Kalray [11] , Tilera [36] and XMOS [25]. These architectures encompass an array of processing elements which work independently but communicate with each other via Network-on-Chip and shared memory systems. Usually, manycores are structured in tiles that may contain cores, caches, local memory, network interfaces, or hardware accelerators. Compared to conventional processors, a core in embedded manycores operates at a relatively low frequency, limited power, and low memory bandwidth. From this group, this thesis uses three such manycore architectures: STHorm aka the Platform 2012 (P2012) [6], Ambric [8] and Epiphany [28].

STHorm [6] is a scalable, modular and power-efficient System-on-Chip based on multiple clusters with independent power and clock domains. Each cluster contains a cluster controller, one to sixteen ENcore processors, an advanced DMA engine, and a hardware synchronizer. The processing elements encompass a customizable 32-bit RISC, 7-stage pipeline, dual-issue core called STxP70-v4. The cores have private L1 program caches and share L1 tightly coupled data memory. They also share the DMA engine and the hardware synchronizer. STHorm targets data-intensive embedded applications and aims to provide more flexibility and higher power efficiency than general purpose GPUs. The STHorm SDK supports a low-level, `C`-based API, a Native Pro-

gramming Model and industry standard programming models such as OpenCL and OpenMP.

Ambric [8] is a massively parallel array of bricks with a total of 336 processors. Each brick comprises two Compute Units and two RAM Units. The Compute Unit consists of two Streaming RISC (SR) processors and two Streaming RISC processors with DSP extensions (SRD). The RAM Unit consists of four independent banks of RAM with 512 words. Ambric also has 32-bit hardware channel that is used to interconnect processor and memory, and for inter-processor communication. Ambric has low power (6-12 watt) and little memory (2KB per processor), which makes Ambric suitable for embedded systems. Ambric targets video and image processing applications. Ambric supports only Kahn process network, and it can be easily programmed using structural object programming model by languages called `aJava` and `aStruct`.

The Epiphany [28] architecture comes with the **Parallella** board [30]. The Parallella is a heterogeneous platform containing one 16-core Epiphany chip, a dual-core ARM CPU, and some FPGA fabric. The ARM processor is the host. It runs Linux and is responsible for managing and loading a program onto the Epiphany. FPGA implements the eLink, a communication link among the ARM CPU and the Epiphany.

Epiphany is a 2D array of nodes connected by a low-latency mesh Network-on-Chip. Each node consists of a floating-point RISC processor with ANSI-C support, 32 KB of globally accessible local memory, a network interface and a DMA engine. The DMA engine can generate a double-word transaction on every clock cycle and has its own dedicated 64-bit port to the local memory. Even though all the internal memory of each core is mapped to the global address space, the cost of accessing individual cores is not uniform, as it depends on the number of hops and contention in the mesh network. Epiphany's network is composed of three different networks which are used for writing on-chip, writing off-chip, and all reading requests, respectively. For on-chip transactions, writes are approximately 16 times faster than reads.

The two academic architectures are the EIT [44] from Lund University and the ePUMA [22] from Linköping University, custom architectures with SIMD support. **ePUMA** is a heterogeneous architecture that targets digital signal processing applications. It comprises a master processor, computing clusters (CCs), and a Network-on-Chip. The master is responsible for delivering tasks to the clusters and managing the usage of the off-chip memory and the Network-on-Chip. The Network-on-Chip includes a star network for data transfers from the computing clusters to main memory, a bi-directional ring network for communication between computing clusters and a mailbox system for notification and synchronization. The computing clusters perform the actual DSP computing. Each computing cluster has shared local vector memories (LVMs)–organized into multiple banks in order to provide conflict free parallel access, a cluster controller–a simple RISC core to manage commu-

nication and local memories, and matrix processing elements (MPEs)–single issue cores specialized for vector/matrix computing.

EIT [44] is a highly reconfigurable coarse grained architecture that targets signal processing applications related to large antenna systems, aka Massive Multiple Input Multiple Output (MIMO). The architecture comprises a master processor (PE1), five processing elements (PE2-6) and two memory elements (MEs) interconnected via high-bandwidth low latency links. PE2-4 and ME2 are used to perform computationally intensive vector operations. PE3 has four parallel processing lanes with complex-valued multiply-accumulate (CMAC) units to perform vector operations. PE2 and PE4 assist the vector computation by doing pre- and post-processing operations. PE5 and PE6 perform scalar operations such as division, square-root, and CORDIC (COordinate Rotation DIgital Computer). The memory is organized in 16 banks to enable parallel access. Banks are further grouped into pages to regulate the access to different lines in the banks.

2.2 Applications

This thesis focuses on digital signal processing (DSP) applications such as radar signal processing, video/audio processing, and computer vision. Specifically, we have used Moving Pictures Experts Group -4 Simple Profile (MPEG-4 SP) decoder, Discrete Cosine Transform (DCT), Inverse DCT (IDCT), QR Decomposition (QRD), and Features from Accelerated Segment Test (FAST) corner detection application to evaluate our compilation process and to analyze the limitations of the hardware.

2.2.1 Moving Pictures Experts Group -4 Simple Profile

A digital video is a sequence of still images (*frames*) displayed at a constant frame rate. The frame rate is measured in the number of frames displayed per second (FPS). A digital video signal is known to have high data redundancy among adjacent frames and within a frame. Video coding (or video compression) is a process of encoding video sequence to a reduced content format, such as bitstream, that is suitable for storage and transmission.

An image (*frame*) often comprises adjacent pixels with similar values. Thus, instead of storing the values of all the pixels, the image is split into blocks of similarly valued pixels, and the average color of the block and the deviation of each pixel is encoded by a process called spatial compressions. Also, the high-frequency areas of the picture can also be reduced since they cannot be recognized by the human eye. Video codecs usually use the DCT and the wavelet transforms for spatial compressions.

Adjacent frames also exhibit high correlation since they have a similar background with the exception of a few moving objects in the foreground. Thus, the idea is to encode only the difference between successive frames, instead of

storing whole frames. This process is called temporal compression. However, a slight motion of the object and/or camera can result in substantial differences. So, to remove the effect of motion from the difference video codecs have incorporated motion compensation. The motion compensation process, first, compares a reference frame and another frame to find the motion vectors of the picture, then removes the effect of the motion by shifting the reference picture in accordance with the motion vectors. Both spatial and temporal correlation have been effectively exploited in video codecs to achieve substantial compression.

For the case study, we have used a video codec from the MPEG Reconfigurable Video Coding (RVC) framework [29]. The RVC framework is an ISO/IEC standard that provides video codec specifications as a standard library written in the `CAL Actor Language`. It provides a modular and reusable framework to implement the MPEG standard. Choosing a `CAL` implementation of MPEG from the RVC framework gives us a reference application to compare our result for both correctness and efficiency. The `CAL` implementation of the MPEG-4 SP decoder has four blocks. The Parser block parses the input bitstream of the encoded video to a form that is suitable for the next stages of the decoder. The ACDC block reconstructs the spatial information by exploiting the correlation between adjacent pixels in blocks. The 2D-IDCT block performs the inverse discrete cosine transform. Finally, the Motion block does motion compensation for blocks with motion information to generate the video output.

2.2.2 Discrete Cosine Transform

DCT is a transform algorithm that compresses an image block into a few discrete cosine coefficients. It transforms an image block from the spatial domain to the DCT domain. The DC coefficients are uncorrelated, so they can be encoded independently. Since the DCT is invertible, the compressed image block can be decoded by using inverse DCT. We have experimented with both forward and inverse DCT algorithms. In Occam-pi we have computed the 1D-DCT as a matrix multiplication implemented in a four-stage streaming application. We have used the implementation to investigate the benefit of reconfiguration for fault tolerance. For CAL, we have used three different implementations of the 2D-IDCT algorithm. The first implementation has a total of 15 actors that communicate in a pipeline manner: two instances of 1D-IDCT (each implemented in five actors and the remaining five to interconnect the two instances, to transpose and re-transpose their outputs). The second implementation has 7 actors, where the 1D-IDCT instances are implemented as one actor. In the third implementation, the 2D-IDCT algorithm is implemented as one actor.

2.2.3 Features from Accelerated Segment Test Corner Detection

FAST is an algorithm that is used to detect corners in an image [33]. In image processing, corners are detected and used to derive a lot of information that is important for computer vision systems. The FAST corner detection algorithm is a high performance detector, suitable for real-time visual tracking applications that run on limited computational resources. FAST examines a pixel by comparing the intensity value of the pixel with the values of sixteen neighboring pixels within a Bresenham circle of radius three. Among these sixteen pixels, if the intensity of N pixels is either greater than or less than the intensity of the pixel by a threshold T, then that pixel is categorized as a corner.

2.2.4 QR Decomposition

QR decomposition is used by many detection algorithms in massive multiple-input multiple-output (MIMO) applications. QRD is a decomposition of a matrix into an upper triangular matrix R and an orthogonal matrix Q. The equation of a QRD for a square matrix A is simply $A = QR$. The matrix A does not necessarily need to be square. The equation for an $m \times n$ matrix, where $m \geqslant n$, is as follows:

$$A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1$$

We have implemented three QRD algorithms (Givens Rotations, Householder, and Gram-Schmidt) in both CAL and native C for the Epiphany architecture. The Givens Rotations (GR) algorithm applies a set of unitary rotation G matrices to the data matrix A. The Householder (HH) algorithm describes a reflection of a vector across a hyperplane containing the origin. The Gram-Schmidt (GS) algorithm produces the upper-triangular matrix R row-by-row and the orthogonal matrix Q as a set of column vectors q from the columns of the data matrix A in a sequence of steps.

2.3 Suitable Programming Models

The presented applications and many other DSP applications work on large streams of data and require high data throughput. However, the major bottleneck in modern embedded manycore architectures is the memory bandwidth: the rate of retrieving data is much lower than the rate of instruction execution. To overcome this problem the architectures have leaned towards a distributed memory organization. Likewise, application developers try to achieve high data

locality to gain performance. The semantics of the concurrent models of computation described in Section 3.1 are very suitable for this kind of architectures. The models have encapsulated actors or processes which communicate based on message passing. This removes race conditions for shared variables and the need to use explicit synchronization mechanisms, reduces the network contention and improves the overall performance.

Chapter 3
Background: Concurrent Models of Computation and Programming Languages

There are a number of concurrent models of computation and programming languages; this chapter presents those that are used in the appended papers.

3.1 Concurrent Models of Computations

A model of computation (MoC) is a high-level abstraction that defines the types and semantics of operations that are used in computations. Concurrent models of computation have a set of rules and definitions that govern the behavior of parallel applications, i.e., a model for parallelism/concurrency, communication, and synchronization. Efficient implementation of concurrent models as programming languages and software development tools can exploit the underlying parallel hardware and satisfy the high-performance demand of the applications. The absence of low-level details in a MoC benefits both application developers and hardware designers.

Dataflow Models of Computation

The dataflow model was introduced and implemented as a visual programming language by Sutherland [34]. In the model, an application is organized as a flow of data between the nodes of a directed graph. The nodes are computational units, usually called actors or processes. Edges describe the dependencies between nodes by connecting explicitly defined inputs to outputs. Nodes use the edges to send and receive tokens. The execution of the nodes is constrained only by the availability of the input tokens. The dataflow model has been used to develop various signal processing applications. Also, the model has influ-

enced many visual and textual programming languages such as `occam-pi` [42], `CAL` [13], `LabVIEW` [37] and `SISAL` [40].

Kahn Process Networks (KPNs)

KPNs [20] are examples of computational models that define specific semantics for the dataflow model. In KPN, processes communicate by sending tokens via unbounded, unidirectional FIFO channels. Writes to a channel are non-blocking, however, reads from an empty channel blocks until sufficient tokens are available. A process cannot check the availability of tokens. Thus, KPN cannot model processes that behave in accordance with the arrival time of input tokens. Since timing does not affect the output, KPNs are deterministic, i.e., for a given set of input tokens, the output is always the same. Also, the order of execution does not affect the output. KPN processes are monotonic, meaning they only need partial information of the input stream to produce partial information of the output stream.

Dataflow Process Networks (DPN)

Like KPN, DPN [24] express sets of processes that communicate over unbounded FIFOs. However, DPN extends KPN by allowing processes to test the availability of input tokens. Reads can return immediately, even if the channel is empty. DPN nodes are stateless actors. Each actor has a set of firing rules that govern the execution of the actor. The firing rules depend on the number and the value of input tokens. When one of the firing rules is satisfied, the corresponding action will be fired. Actions are computational blocks of DPN that map input tokens into output tokens when fired. If more than one firing rule are satisfied at the same time, then only one will be chosen. This makes the network non-deterministic. The CAL language is an implementation of the DPN computational model.

Synchronous Dataflow (SDF)

SDF [23] is a restricted DPN model with a single firing rule. In SDF, any firing consumes and produces a constant number of tokens. Having a fixed consumption/production rate makes SDF the easiest model to be analyzed. The restriction enables the possibility of making static compile time analysis on the SDF graph to determine the mapping and scheduling of SDF actors. This makes the SDF model suitable for various signal processing domains.

Cyclo-Static Dataflow (CSDF)

CSDF model [7] is a DPN model that has a cyclic firing of functions. CSDF allows firings to have different consumption/production rates, but each cycle

has a constant consumption/production rate. Thus, a CSDF graph can also be mapped and scheduled at compile-time. Usually, CSDF is implemented by extending the firing rules by a state variable that returns to the initial value after a sequence of firings or by using state machines that come back to the initial state.

The Actor Model

The actor model [18] was developed in 1973 as a model for programming languages in the domain of artificial intelligence. The model encapsulates computation in components called actors. Actors are autonomous, concurrent and isolated entities that execute asynchronously. Actors communicate asynchronously by sending a message to named actors. In the original model, actors can be created and reconfigured dynamically. In addition, the model has no restriction on the order of messages, i.e., messages sent to an actor can be received in any order. Thus, an actor does not require a queue to store messages. However, to support asynchronous communication, buffering of messages is necessary. The actor model has been used to model distributed systems, and recently the model has been adapted to model concurrent computation. Most standard languages have added actors as a library facility, for example, `C++` and `Java` as *threads* and `Scala` as *actors*. The model has inspired languages like `CAL` [13], `Erlang` [3] and `SALSA` [40].

Communicating Sequential Processes (CSP)

In CSP, processes *share nothing*; they communicate using synchronous message passing via unidirectional and named channels. CSP is a mathematical model with a set of algebraic operators that operate on two classes of primitives: *events* to represent communication or interactions and *processes* to describe fundamental behaviors of a process. CSP message passing follows a rendezvous communication—the sender blocks until the message is received. The CSP model has been implemented in a number of languages, such as, `occam` [2], `JCSP`—CSP for `Java` [43] and `XC`—the native language of the XMOS architecture [41].

Pi-calculus

The pi-calculus was introduced by Milner et al. [27] to express processes that change their structure dynamically. The pi-calculus has an expressive semantics to describe a process in concurrent systems. The central feature of the pi-calculus is the communication of named channels between processes, i.e., a process can create and send a channel to another process, and the receiver can use the channel to interact with another process. This enables the pi-calculus to express *mobility* in terms of dynamic network change and re-configuration.

The concepts of the pi-calculus have been used in various programming languages, such as `occam-pi` [42] and Pict [31].

3.2 CAL Actor Language

`CAL` [13] is an actor-oriented dataflow programming language that extends DPN actors with states. CAL actors may have parameters to create actor instances with different property, private variables to control the state of the actor, named input/output ports to communicate with other actors and actions to perform a particular task. A `CAL` actor does not have access to the state variable of other actors. Therefore, interaction among actors happens only via input and output ports. Actions are the computational units of a `CAL` actor. As in DPN, `CAL` actors take a step by firing actions that satisfy all the required conditions. Unlike DPN, these conditions depend not only on the value and number of input tokens but also on the actor's internal state. Also, `CAL` firing conditions may include *finite state machines (FSM)* to order the firing of actions and *action priorities* to select an action with the highest priority if there is more than one eligible action. Depending on the use of specific constructs, `CAL` can support various models of computation, such as Synchronous Dataflow (SDF) [23], Cyclo-Static Dataflow (CSDF) [7], Kahn Process Networks (KPN) [21], and Dataflow Process Networks (DPN) [24].

Each CAL action may have:

- A label: to name an action, e.g. 'A1' and 'A2' in Listing 3.2.
- Input patterns: to define the number of the tokens and to name the token for the rest of the action.
- Output patterns and expressions: to send the output tokens computed using the input and the state variables. Both input and output patterns could have the *repeat* keyword to define or send an array of tokens.
- A guard: a Boolean expression that must be true to enable the firing of the action.
- A body: to perform computations such as updating state variables and computing output tokens. An action firing starts with consumption of all input tokens, continues with execution of statements in the body, and ends by sending output tokens.

The interconnection between actors is expressed using Network Language (NL). NL has three sections: a *variable declaration section* to define variables that are used as attributes for actors and sub-networks, an *entity section* to declare actors or sub-networks, and a *structure section* to list channels that sketch the dataflow network. In NL, a programmer can add additional information, like FIFO size. Listing 3.1 shows the expression of the network of the two actors that is shown in Figure 3.1.

```
network SimpleNL () In ==> Out:
entities
    split = Split ();
    combine = Combine();
structure
    In --> split.A;
    split.P --> combine.P;
    split.N --> combine.N;
    combine.C --> out;
end
```

Listing 3.1: Simple Network.

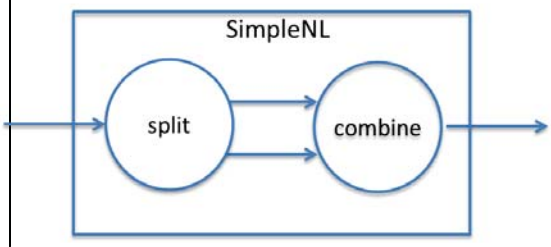

Figure 3.1: Simple Network.

```
actor Split ( ) A ==>  P, N:
  A1 : action A: [ v ] ==> P: [ v ]
     guard v >= 0
     end
  A2 : action A: [ v ] ==> N: [ v ]
     guard v < 0
     end
end
```

Listing 3.2: Split: A CAL actor.

```
actor Combine ( ) P,N ==>  C:
  int z := 0;
  action P:[x], N:[y] ==> C:[z]
  do
      z := x + y;
  end
end
```

Listing 3.3: Combine: A CAL actor.

The *Split* and *Combine* actors are presented in Listing 3.2 and Listing 3.3, respectively. The *Split* actor has three firing conditions–the availability a token in input port A and the two predicates on the value of the token. As stated in the predicate of action A1 if the value v of the token read from the input port is zero or greater than zero, then action A1 will be fired, and the token is sent to port P. Otherwise, v is negative, action A2 will be fired and the token is sent to port P. *Combine* actor reads input from port P and N add them and send the result to port C

For detailed discussion on the specification of the `CAL Actor Language` please read the `CAL` language report by J. Eker and J. Janneck [13].

3.3 Occam-pi

Occam-pi [42] is a programming language that extends occam by introducing mobility features of the pi-calculus [27]. Occam was developed based on CSP in order to program the Transputer [5] processors. Like in CSP, processes in occam-pi *share nothing*, they communicate via channels using message passing. However, in occam-pi, if the data is declared as MOBILE, the ownership can be moved to different processes. Occam-pi has also extended the channel definition of Occam by adding direction specifiers (?) for input and (!) for output channels. Compared with occam, occam-pi supports

- asynchronous communication via directed channels,
- dynamic parallelism, and
- dynamic process invocation

These features of occam-pi enable a network reconfiguration process that can be changed based on the application requirements and the available resources.

The primitive processes of occam-pi include assignment, input (?) and output (!). Occam-pi have constructs for sequential processes (SEQ), parallel process-es (PAR), iteration (WHILE)selection (IF/ELSE, CASE) and replication. The SEQ and PAR constructs can be replicated. A replicated SEQ is similar to a for-loop. A replicated PAR can be used to instantiate a number of processes instances in parallel.

Listing 3.4 shows the corresponding occam-pi code for the *Split* actor, *Combine* actor and *SimpleNL* network file presented in Section 3.2.

```
PROC Split (CHAN INT A?, P!,N!)
  INT v:
  SEQ
    in ? v
    IF
      read >= 0
        P ! v
      read < 0
        N ! v
:
PROC Combine (CHAN INT P?, N?, C!)
  INT v1,v2,v3:
  SEQ
    P ? v1
    N ? v2
    v3 := v1 + v2
    C ! v3
:
PROC SimpleNL (CHAN INT In? Out!)
  CHAN INT P,N
  PAR
    Split(In?,P!,N!)
    Combine(P?,N?,C!)
```

```
    :
```

Listing 3.4: `Occam-pi` code for Split, Combine and SimpleNL proecesses

3.4 Conclusion

The languages used in the thesis are `occam-pi` and `CAL`, which are practical implementations of CSP with pi-calculus and the actor-oriented dataflow model, respectively. The CSP model has static processes that communicate with each other via static synchronous channels. However, `occam-pi` has extended the CSP model using mobility feature of *pi-calculus* [27], which enables `occam-pi` to model dynamic reconfiguration and asynchronous communication. Like the dataflow model, `CAL` has pre-defined nodes (actors), and data flows from explicitly defined output ports to input ports. In contrast to the actor model, `CAL` constructs do not allow dynamic creation of actors and reconfiguration of channels, but depending on the implementation of the actors, it can abstract restricted actor models and dataflow models, DPNs and various communication and computation models.

Chapter 4
Compiling `Occam-pi` for Manycores

To enable programming manycores using `occam-pi` we have extended the Translator from `occam` to `C` from Kent (Tock) [1], Fig 4.1. Tock is a compiler for `occam` developed in the Haskell programming language at the University of Kent. It has three main phases: front end, transformations, and backend. Each of these phases transforms the output of the previous step into a form closer to the target language while preserving the semantics of the original program. The frontend performs lexing, parsing, type checking, and name resolving. The transformation phase comprises step-by-step passes that perform machine-independent passes, such as *simplifications*, e.g. turning the parallel assignment into a sequential assignment, and *restructurings*, e.g., grouping variable declarations. The backend performs target-specific transformations and code generation. In earlier work, Z. Ul-Abdin and B. Svensson have extended Tock to program the Ambric [38] and XPP [39] architectures using `occam-pi`. In **Paper A** and **B**, we perform additional extensions to Tock by adding a new backend for STHorm aka the Platform 2012 (P2012) [6].

The STHorm backend generates a `C` code for the host-side program and Native Programming Model (NPM) for the STHorm fabric. The host program deploys, runs and controls the application. The NPM sketches the complete structure of the application using three languages: extended `C`-code for the ENcore processors and cluster controller, Architecture Description Language (ADL) to define the structure of each component (process), and Interface Description Language (IDL) to specify the interfaces of the processes. The cluster controller is responsible for starting and stopping the execution of the ENcore processors and notifying the host system. The ENcore processors run the main implementation of an application.

The backend has two passes. The first pass collects all process calls, sketches the network of the processes, flattens the network and generates `C` code for the host-side, ADL code to specify the input/output of each process and IDL

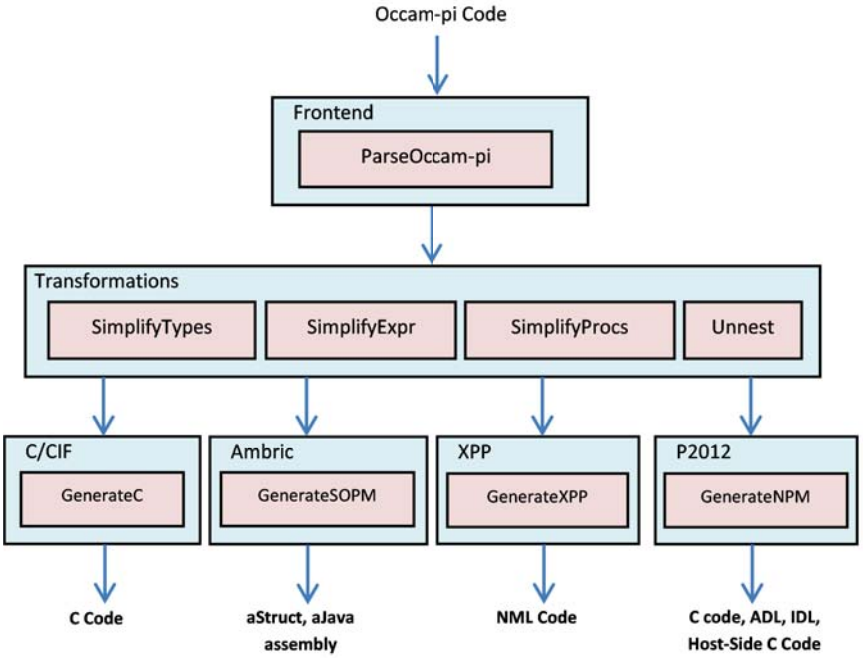

Figure 4.1: `Occam-pi` to manycores compiler block diagram

code to interface and bind each process. The second pass uses the definition and body of an `occam-pi` process to generate extended `C` code for ENcore processors and the cluster controller.

4.1 `Occam-pi` for Fault Tolerance

Paper A shows the use of `occam-pi` to program the fault tolerance aspects of a parallel application. `Occam-pi` provides high-level constructs that enable the programmer to facilitate the management of a dynamic task relocation from the faulty processing elements to the faultless ones. These high-level constructs include channel direction specifiers, mobile data and channel types, dynamic process invocation, and process placement attribute. Using these constructs, we have implemented a dynamic reconfiguration of the hardware resources in the STHorm platform.

As a case study, the One-Dimensional Discrete Cosine Transform (1D-DCT) algorithm is implemented in a four stage pipeline and executed on four ENCore processors. During execution, if the run-time system detects a fault in one of the processing elements, it will pass an error code to the application. Then the fault tolerance model will issue a new configuration that avoids the use of the faulty processing element. The fault tolerance model adds 23% overhead to communicate the error code. If a fault is detected, the reconfiguration process adds about 7% overhead to issue and deploy a new configuration. Since

the reliability of the system is increased, we believe the overhead is tolerable and reasonable to justify the usefulness of the approach.

4.2 Occam-pi for Data-Intensive Applications

Paper B demonstrates the suitability of occam-pi for data-intensive application domains such as image analysis and video decoding. STHorm has useful hardware features, like a multi-channel DMA engine, to accelerate the transfer of data in data-intensive applications. To generate code that utilizes these resources efficiently, we have revised the Tock front end and the STHorm backend. In the front end, we add support for channels that communicate an entire array of data in a single transfer, and in the STHorm backend, we translate these data transfers to low-level APIs that access the specific hardware features. As a proof of concept, we have implemented the FAST (Features from Accelerated Segment Test) corner detection algorithm in occam-pi. The algorithm utilizes data-level parallelism by duplicating critical sections and by using channels that transfer an entire array of data. In addition, we have used parameterized replicated PAR (the occam-pi construct for parallelism) to run the algorithm on a given number of processes.

Using the FAST implementation, we have shown the simplicity of programming in occam-pi and the competitiveness of our compilation scheme in terms of performance. We have compared the occam-pi FAST implementation with NPM and OpenCL implementations. The result shows that the execution time of the occam-pi version is almost the same as for the OpenCL implementation and much shorter than the NPM version. To compare the development efforts for the implementations, we have counted the number of source lines of code (SLOC); when both NPM and OpenCL implementations use around 450 SLOC, in occam-pi we used only 190 SLOC.

Chapter 5
Compiling CAL Actor Language for Manycores

To compile CAL for manycores we have developed the Cal2Many compilation framework **[Paper C-H]**. The compilation takes a CAL program and generates an implementation expressed in the native language of the target architecture. This is done in three steps. First, each CAL actor is translated to an actor machine (AM) intermediate representation, which is then translated to Action Execution Intermediate Representation (AEIR). Finally, the AEIR and the description of the network of actors are used by the different backends to generate target-specific source code.

An AM is a controller with a set of *states* made from all firing conditions of actions together with the finite state machines and the action priorities. AM *states* have knowledge about conditions and a set of AM instructions that can be performed on the *state*. AM instructions can be: a *test* to test one of the firing conditions, an *exec* for the execution of an action, or a *wait* to change information about the absence of tokens to unknown, so that a test on an input port can be performed after a while.

To execute an AM, its constructs have to be transformed to different programming language constructs, such as function calls to execute the AM instructions, *if* statements to test the conditions and flow control structures to traverse from the current AM *state* to the destination *state*. These constructs have different implementations in different programming languages and platforms. The AEIR is used to abstract these constructs and to bring AM closer to an imperative sequential action scheduler without having to select a target language. The translation of AM to AEIR deals with two main tasks. The first task is the translation of CAL constructs to imperative constructs. This includes CAL actions, variable declarations, functions, statements, and expressions. The second task is the translation of the AM into a sequential action scheduler. This is kept as a separate function that is made up of statements

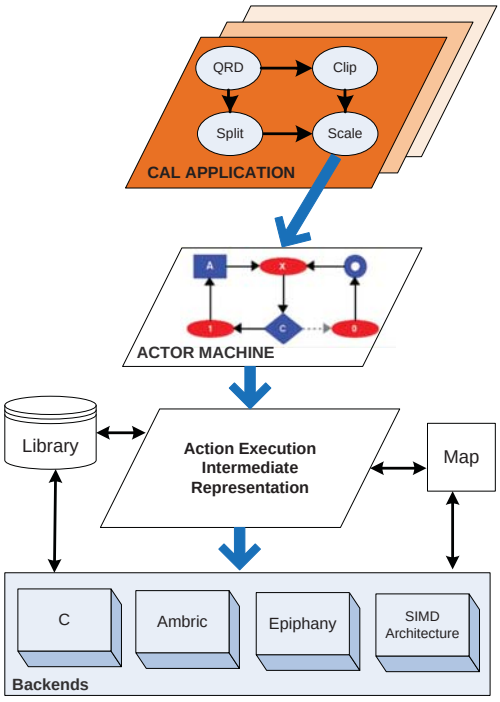

Figure 5.1: The Cal2Many compilation framework.

translated from the nodes of the AM and a scheme to traverse from AM states to destination states.

Using CAL and the two intermediate representations we have increased the portability and productivity of applications and retargetability of the Cal2Many framework. Also, we have enabled scalability using composition, decomposition and replication transformation. Currently, the Cal2Many has four backends: a uniprocessor backend that generates sequential C code for a single-core general purpose processor, an Epiphany backend that generates parallel C code for the Epiphany chip, an Ambric backend that generates aJava and aStruct for the Ambric Massively-parallel processor array, and a SIMD backend that translates AEIR to Target Specific Language (TSL) that is then used to generate code for ePUMA and EIT [17] [16].

We have used the Network Language (NL) [19] to sketch the network of the complete CAL program. NL defines instances of actors and creates channels that connect outputs and inputs. After generating code for each actor, we have used NL to generate a round-robin scheduler for the sequential C code, host code for Epiphany and a top-level design file for Ambric to bind the aJava objects that correspond to instances of CAL actors.

5.1 Compiling for KPN and DPN

CAL provides structures such as guard, priorities, finite state machines and token rates to support various models of computations such as SDF, CSDF, KPN, and DPN. **Paper C** presents the DPN and KPN interpretations and compilation of CAL actors. The paper shows the feasibility and portability of our approach by compiling DPN interpretation of 2D-IDCT algorithm for a general purpose processor and the Epiphany [28], and KPN interpretation of the same application for the Ambric massively parallel processor array [8]. The 2D-IDCT implementation has 15 actors communicating in a pipeline manner. We have used a fine-grained version in order to test the framework via a network of actors. For sequential C and the Epiphany we have developed a custom communication library, but for the Ambric we have used the available support for channels, communication APIs, and the KPN model. While implementing the communication library for the Epiphany architecture, we have exploited specific features of the architecture such as the speed difference between *read* and *write* transactions (writes are faster) and the use of DMA to speed up memory transfers. For the general purpose processor (sequential version) we used both inlined and non-inlined code generators. For the Epiphany, we have used only the non-inlined version because it has smaller code memory footprint. Similarly, for Ambric we have used the non-inlined version and adjusted the code generation in accordance with KPN since Ambric only supports KPN.

The results show that the inlined sequential code generation has improved the performance of the unoptimized non-inlined version by 33%. The unoptimized non-inlined parallel C code generation for the Epiphany has also improved the corresponding sequential version by 30%. However, the optimized inlined version still has better performance than the parallel implementations. This is because the performance of the parallel codes is significantly affected by the communication overhead, which is very common in fine-grained parallelism. The clock speeds of the parallel architectures are much slower than that of the general purpose CPU (close to five times slower for Epiphany and ten times slower for Ambric) which limits the expected speedup. Additionally, the parallel versions are slowed down by shared memory accesses. In particular, the last actor spends most of the clock cycles in dealing with off-chip memory accesses; this caused the output buffer of the previous actor to be full. This full buffer led to backward pressure that affected the whole implementation, making all the actors wait till there is room in the output buffer.

However, without any code optimization, and considering the low clock frequency and the extra communication overhead, both parallel implementations show a potential for performance portability.

In **Paper D**, **F**, and **G** we have done further experiments on the Epiphany architecture with the purpose of evaluating the code generation from CAL and understanding the limitations of the underlying architecture. In **Paper D**, we evaluate the communication library and the code generation for Epiphany us-

ing the 2D-IDCT algorithm. Also, we have compared the code generated from CAL with a handwritten implementation developed in C. While comparing, we have found many optimization opportunities, of which we have implemented three. The first and most important optimization was the removal of unnecessary external memory accesses. The other two optimizations are concerned with function inlining. In the non-inlined version, CAL actions are translated to two functions: *action_body* to implement the body of the action and *action_guard* to evaluate the guard of the action. Thus, the second optimization inlines *action_guard* and the third optimization inlines both *action_body* and *action_guard*.

Initially, the hand-written implementation had 4.3x better throughput performance than the implementation from the Cal2Many code generator. Optimizing the memory access, in the generated code and the communication library, increased the throughput by 63% which brings the performance as close as 1.6 times when compared with the performance of the handwritten implementation. Combining all optimizations, we were able to reduce the difference in execution time down to a factor of 1.3x.

In **Paper F** three QR decomposition algorithms (Givens Rotations (GR), Householder (HH), and Gram-Schmidt (GS)) have been implemented in CAL and hand-optimized C code to compare the performance of CAL to C code generation with the handwritten C code. The performance of the CAL (generated C) implementations gets as good as 2% slower. While using the external memory, the average result of generated code of the three algorithms is 4.3% slower than the hand-written code. When the internal memory is used the execution time of the generated code is 1.46x, 1.14x, and 1.09x for GR, HH, and GS, respectively, compared to the performance of the handwritten implementation.

Also, in both **Paper D** and **F** we have compared the number of source lines of code (SLOC) used in the CAL program and in the hand-written implementation to estimate the development effort. In **Paper D** 495 SLOC were used to write the 2D-IDCT application in CAL while more than four times as many (2229 SLOC) were needed for the C implementation. In **Paper F** the average number of source lines of code that are needed for the CAL implementations is 25% smaller. This clearly indicates the simplicity and expressiveness of the CAL language and supports the acceptability of the performance level that can be gained when using the compilation framework.

In **Paper G**, we have used MPEG-4 SP to analyze the Epiphany code generation and to investigate the limitation of the Epiphany manycore architecture [28]. In the paper, first, we have executed the whole MPEG-4 SP on the host processor. Next, we have executed 15 actors and a *Distributor* actor in the Epiphany chip and the remaining 4 actors on the host processor. The *Distributor* actor reads the output of the off-chip actors and distributes them to on-chip actors. The speed up of the initial result was only 2.5x.

To investigate why the application does not benefit from the available parallelism, we have formulated four hypotheses. The hypotheses are a) the impact of the computation load of the actors, b) the impact of the core to core communication overhead, c) the impact of the off-chip access, and d) the impact of the implementation and the dataflow graph of the application. Based on the findings of the hypotheses, we have modified the implementation to improve the overall performance. The first hypothesis has a minor effect since the actors spend a very low percentage of the execution time to compute data. Standing on the findings of the second hypothesis, we have improved the communication library (CommLib) [32] by removing `memcpy` and generate a new mapping of actors. Doing so have improved the performance by 40%, from 15 to 21 FPS. For the third hypothesis, we have introduced local arrays to store the output of the off-chip actors. This has reduced the busy waiting time of full output channels and improved the result to 27 FPS. The result of examining the final hypothesis shows that the Motion block is the bottleneck of the application, contributing to more than 70% of the overall computation.

5.2 Compiling for Vector Processors

Some embedded manycore architectures target application areas that exhibit both task and data level parallelism, e.g. MIMO and image precessing applications. Such applications can be modeled as streaming applications that encapsulate the computations in communicating concurrent actors to exploit the task parallelism, and use SIMD data types and operations to exploit the data level parallelism. In **Paper E**, in order to program the two academic vector processing architectures, the EIT [44] from Lund University and the ePUMA [22] from Linköping University, we have extended compilation process of `CAL` to support SIMD data types and operations. We have added SIMD support in `CAL` to enable efficient utilization of manycore architectures with specialized ISA to support vector and matrix operations. The frontend and the two IRs represents SIMD operations and data types in the AST as it is, which gives the backend the required information for competent code generation. In the backend, depending on the target architecture, the SIMD operations can easily be translated to a specialized hardware accelerator, optimized kernel, or even to an instruction that executes the operation in one cycle.

To program EIT and ePUMA, we have added a SIMD backend that translated AEIR to Target Specific Language (TSL), a language that encapsulates the SIMD-like nature of the architectures. The TSL is then used by instruction scheduling and memory allocation tool developed at Lund University [4]. In the tool, the scheduling and the memory allocation are done in a single constraint programming (CP) model, which produces a schedule with memory allocation for a code generator that turns this schedule into machine code.

To show the feasibility of our approach, we have generated code for QR decomposition (QRD) and Matrix Multiplication (MM) from a `CAL+ SIMD` im-

plementation and compared it with hand-written implementation. For EIT we used QRD to evaluate the performance of the generated instruction schedule, and for ePUMA, we generated assembly code for MM and compared the execution time. In EIT, compared to the manual schedule created by the architecture designer, our schedule performs around 20% worse. In ePUMA, for 1 to 32 concurrent MM operations, the generated code adds from 63.5% to 14.4% of overhead compared to the hand optimized assembly code. The generated code performs better for a higher workload. In both cases, the overhead of our approach is understandable since the architecture experts have used low-level tuning and specialized addressing modes. However, our method goes from a CAL code to a schedule with memory allocation and assembly code within seconds while the manual scheduling and assembly coding takes many man-hours and is a highly error-prone task.

5.3 Compiling with Actor Transformations

In Cal2Many, the AM is a high-level abstraction that retains the high-level information present in the CAL actor. Thus, AM can be used for high-level transformations such as composing and decomposing actors. AEIR is a low-level abstraction compared to AM. AEIR lacks information about action firing conditions such as predicate condition, action priority, and FSM. Thus, AEIR is more suitable for low-level optimizations, such as constant propagation, dead code elimination, and inlining functions. Since the low-level optimizations are usually done by the native compiler of the target architecture, in Cal2Many, we focus more on actor transformations.

Actor Composition

In actor-oriented programming usually the number of actors is greater then the available execution units. Thus, when mapping streaming applications onto manycore architectures more than one actor could be mapped to one processing element. In such a case the execution of the instructions of the actor are interleaved. One way of realizing this process is *composite actor machine*. Composite AM interleaves the firing of actions of the actors of a network. While scheduling the firing of actions the composite AM can keep track of the number of tokens that are written to and read from the internal connection and avoid the need to test all input conditions on the internal connections.

In **Paper G**, we have used a fine-grained version of 2D-IDCT in order to test the actor composition on the Epiphany chip. The model of the composite actor machine that we have used is designed by Cedersjö and Janneck [10]. The 2D-IDCT implementation has a total of 15 actors communicating in a pipeline manner, two 1D-IDCT instances each implemented in five actors and the other five actors to interconnect the two instances.. We have tested three scenarios to evaluate the actor machine composer. The first scenario translates

each actor to an actor machine and maps it on one core, i.e., 15 AMs on 15 cores. The second scenario also has 15 AMs, but the five AMs of each 1D-IDCT instance are mapped on one core, i.e., 15 AMs on 7 cores, where two cores run five actors using a round-robin scheduler. The third scenario composes the five actors in the 1D-IDCT to produce one composite actor machine resulting in a total of seven AMs, i.e., 7 AMs in 7 cores.

We have executed the three scenarios on the Epiphany architecture, and two scenarios (1^{st} and 3^{rd}) on a GP-CPU. The result shows that the composite AM has improved the overall performance by approximately 16% and 10% for the Epiphany and the GP-CPU, respectively. Composite AM enforce a group of actors to be mapped to the same core as one actor. For the Epiphany, we have repeated the excrement using a CommLib that has a reduced overhead. Using the latest CommLib, the results of the first and third scenarios become more or less the same. However, the second scenario is around 20% slower due to the context switching on the two cores that run five actors using round-robin scheduler. In both cases, the result shows that, even if composite actor machine reduce the parallelism of an application, it can improve the overall performance by removing the internal communication overhead.

Actor Replication

Actor replication is the process of creating multiple replicas of an actor in the dataflow graph of the application. It increases the utilization of the hardware by mapping the replicas on different hardware resources. It performs data-level parallelism by distributing the input tokens to actor instances that run the same code. It also reduces backpressure by replicating the slow downstream actors.

As presented in **Paper H**, Cal2Many perform actor replication of stateless actor by encircling the replicas by a *Distributor* and a *Collector* actors to interconnect them with the other actors in the dataflow graph of the application. For SDF and subset of cyclo-static (written by using finite state machine) CAL actors, the replication uses simple *Distributor* and *Collector* actors to distribute and collect a fixed number of tokens are in round robin. For cyclo-static, the number of the tokens is equal to the consumption/production rate of one cycle, but for the SDF the rate is equal to the I/O rate of any action since they all are the same. If a stateless actor is not identified as an SDF or a cyclo-static actor, the replication uses an advanced *Distributor*. The *Distributor* runs the action scheduler, identifies the eligible action, distributes the required tokens, and sends a *control token* to the replicas and the *Collector* to point out the eligible action. The *Distributor* also has a round robin scheduler to distribute the firing of action among the replicas: the first eligible action will be executed by the first replica, the next action by the second replica, and so on.

To analyze the gain in using replication, we have used 2D-IDCT implemented in 7 actors with two instances of 1D-IDCT each implemented as one

actor and remaining 5 actors. The experiment has replicated the 1D-IDCT actor. The result shows that the gain while having two replicas is negligible. This is because the two replicas are performing the computation and also the distribution and collection. Also, the utilization of the hardware is increased only by two cores. However, when the replicas are increased to four, where the two replicas are just computing data, the overall computation time have decreased by 40%.

Actor Decomposition

Actor-decomposition decomposes an actor with a number of actions into actors that have a smaller number of actions. The transformation is mainly used when an actor is too big to fit in the memory footprint of a core in an embedded manycore architecture. In such case, the big actor is decomposed into a *Splitter*, a *Collector* and zero or more *Worker* actors. This transformation can be applied to any CAL actor. The *Splitter* actor manages and controls the firings of the actions of the slice of the actors. The *Splitter* distributes the input tokens to the *Collector* and *Worker* actors. Also, the *Splitter* runs the action scheduler from the AM of the actor and sends a *controller token* to indicate which action the *Collector* and *Worker* actors should fire. The *Collector* has a simple action scheduler that reads the *controller token* and fires the eligible action. The *Worker* actors also have a simple action scheduler guided by the *controller token* and actions that perform an actual computation.

During the implementation of actor-decomposition, identifying the group of actions to be partitioned plays a significant role in the overall performance. In a CAL actor, actions usually access the same state variables and may share some I/O ports. Actions that do not use the same ports and state variables can be mapped in different cores and can run in parallel. We classify such actions as *disjoint* actions. However, finding disjoint actions is very rare, because developers usually put such actions on different actors from the beginning. So, for actor-decomposition, the practical solution is to find a group of actions that share few ports and state variables.

To analyze the impact of using actor decomposition, we have used the 2D-IDCT algorithm implemented in 7 actors and decompose the 1D-IDCT actor. Also, we have decomposed the *ParseHeaders* actor from the Parser block of the MPEG-4 SP. In the 2D-IDCT experiment, the result shows that decomposing the 1D-IDCT actor into two have a negligible gain. Increasing the decomposition to three slices of actors have resulted in 6% decrease in the overall computation time. The gain comes from the created pipelined parallelism. The third sliced actor (the *Worker*) reads input tokens from the *Distributor*, computes the data, and sends its output to the *Collector* actor.

The *ParseHeaders* has 67 actions to parse the bits and to distribute the video object plane (VOP) and the block type information. In addition, *ParseHeaders* uses a large table to store codewords for computing the variable length

decoding (VLD). Since the Parser block is the first step, its throughput has a significant impact on the overall performance of the decoder. Unfortunately, the VLD-table, the 67 actions, the action schedulers, and the channels of the *ParseHeaders* cannot fit in the memory of a single core of the Epiphany architecture. We have run the four actors in the Parser block on the host ARM A9 processor and the Epiphany chip. To map the actors in the Epiphany chip, we have decomposed the *ParseHeaders* actor into two actors. Compared with the mapping of the actors in the host core, the decomposition has decreased the execution time by 5.5%. The gain in performance is quite small considering the number of cores. However, decomposing and mapping actors within a chip is a better choice than the mapping of big actors on the host core and communicating them through the slow off-chip network to the rest of the actors in the Epiphany chip.

Chapter 6
Conclusions and Future Work

Conclusions

With the arrival of manycore architectures with substantial capabilities for parallelism, software development needs new methodologies, programming languages, development tools, and compilers. The thesis has presented compilation frameworks for two concurrent programming languages, `occam-pi`, and `CAL Actor Language`, and demonstrated the applicability of the approach with application case-studies.

To compile `occam-pi`, we have extended the Tock compiler and added a backend for STHorm. The STHorm backend starts with a transformed abstract syntax tree of `occam-pi` and generates `C` code for the host-side program and Native Programming Model code for the STHorm fabric. The approach is evaluated using two case studies. The first case study implemented and evaluated a fault tolerance model for a four stage 1D-DCT algorithm using `occam-pi` constructs for dynamic reconfiguration, like dynamic process invocation, process placement, and mobile channels. The second case study implemented the FAST corner detection algorithm in `occam-pi` using channels that transfer an entire array of data and replicated `PAR` in order to demonstrate the suitability of `occam-pi` and the compilation framework for data-intensive applications.

Using the two case studies, we have demonstrated the applicability and competence of `occam-pi`'s compilation framework for reconfigurable, communication intensive and data intensive applications.

For `CAL`, we have started to develop a new compilation framework that we call the Cal2Many. The current `CAL` compilation framework has a front end, two intermediate representations, and four backends: a uniprocessor backend that generates sequential C code for a single-core general purpose processor, an Epiphany backend that generates parallel C code for the Epiphany [28], an Ambric Backend that generates aJava and aStruct for Ambric Massively-parallel processor array [8], and a SIMD backend that translates AEIR to Target Specific Language (TSL) that is used to generate code for ePUMA and

EIT [17] [16]. We have shown the feasibility of our approach by compiling a CAL implementation of three QR decomposition algorithms, the 2D-IDCT, and MPEG-4 SP. We have compared the Epiphany code generation from CAL with a hand-written C code implementation of 2D-IDCT and three QR decomposition algorithms. We have performed a detailed evaluation and optimization on Epiphany's code generation and on the custom communication library which was developed for Epiphany. We have extended the compilation process to support SIMD data type and operations. We have proposed and experimented with actor replication and decomposition transformations. Also, we have tested actor composition via composite actor machine.

In conclusion, languages that implement concurrent computation models hide the low-level details of the hardware from the application developer while allowing the compiler to achieve efficiency. Occam-pi and CAL are such languages that have practical, simple and powerful semantics to model concurrent applications. We have compiled the two languages using two compilation frameworks and addressed productivity, portability, efficiency, fault tolerance and scalability aspects of parallel applications. Both occam-pi and CAL are more suitable for task-parallel applications. However, in **Paper B** and **E** we have revised the compilation of both languages to increase their competitiveness for data-intensive applications. Also, we have used high-level abstractions like actor machines to increase portability and low-level abstractions in the backends to increase efficiency. The results of **Paper B** $^{\text{occam-pi}}$, **D** $^{\text{CAL}}$, and **F**$^{\text{CAL}}$ show that the generated code can achieve results that come very close to the hand-written native language implementation. Also, by comparing the sizes of the source codes we can conclude that it is easier to develop and debug parallel applications in occam-pi and CAL rather than in the native languages provided by the manycore developers. The identification of different levels of abstraction has created more room for optimization, analysis, and transformation processes. As presented in **Paper H**, Cal2Many enables scalability by utilizing composition, decomposition and replication transformation.

As presented in Figure 4.1 and Figure 5.1, both Tock and Cal2Many support more than one manycore architecture. Our results for all of the target architectures show that the availability of different abstraction layers in the compilation tools eases retargeting and program portability without compromising too much of the performance of the resulting implementation.

Future Work

The thesis has presented the highlights of recent work, and there are some issues that must be addressed in the future. We plan to focus on the CAL compilation framework and, to strengthen our results, evaluate our work using more complex case studies, such as deep learning algorithms and signal processing applications related to large antenna systems, aka massive multiple input multiple output (MIMO). For deep learning, besides CAL, we would also

like to start using domain-specific language (DSL). The work involves selecting or developing a DSL and implementing a front-end that translates the DSL to either actor machine or action execution intermediate representation.

We have also planned to integrate automatic mapping and scheduling solutions that explore the dataflow graphs of relatively complex applications, and that consider constraints and architecture specific features of the underlying parallel architecture. The Cal2Many compilation framework can perform composition and decomposition transformations. However, we believe that the *decisions* to compose and decompose actors should be made by trace based dataflow design space exploration tools such as TURNUS [9, 26]. We are working on adapting TURNUS for Epiphany to link it with Cal2Many. TURNUS generates and uses an execution trace graph of the action firings to find efficient scheduling and mapping solutions. The CAL2Many framework can then be used to implement the solutions via composition and decomposition transformations. Also, AM can be extended to generate an execution trace graph.

References

[1] Tock: Translator from occam to C by Kent. `http://projects.cs.kent.ac.uk/projects/tock/trac/`. Accessed: 2015-01-30. (Cited on pages 2, 3, and 19.)

[2] Occam ® 2.1 reference manual. Technical report, SGS-Thomson Microelectronics Limited, 1995. (Cited on page 13.)

[3] Joe Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007. (Cited on pages 2 and 13.)

[4] Mehmet Ali Arslan, Flavius Gruian, Krzysztof Kuchcinski, and Andréas Karlsson. Code generation for a SIMD architecture with custom memory organisation. In *Proceedings of the IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 90–97, 2016. (Cited on page 27.)

[5] Iann M Barron et al. The transputer. *The microprocessor and its application*, pages 343–57, 1978. (Cited on page 16.)

[6] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012. (Cited on pages 3, 5, and 19.)

[7] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996. (Cited on pages 12 and 14.)

[8] Michael Butts, Anthony Mark Jones, and Paul Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *Proceedings of Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 55–64, Washington, DC, USA, 2007. (Cited on pages 4, 5, 6, 25, and 33.)

[9] Simone Casale-Brunet, Claudio Alberti, Marco Mattavelli, and Jorn W Janneck. Turnus: a unified dataflow design space exploration framework for heterogeneous parallel systems. In *Proceedings of the IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 47–54, 2013. (Cited on page 35.)

[10] Gustav Cedersjö and Jörn W Janneck. Software code generation for dynamic dataflow programs. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pages 31–39. ACM, 2014. (Cited on page 28.)

[11] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013. (Cited on page 5.)

[12] E-Lang. The E language, 2016. (Cited on page 2.)

[13] Johan Eker and Jörn W Janneck. CAL language report: Specification of the CAL actor language. Technical Memorandum UCB/ERL M03/48, University of California, Berkeley, CA, USA, 2003. (Cited on pages 12, 13, 14, and 15.)

[14] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. (Cited on page 2.)

[15] Samuel H Fuller and Lynette I Millett. Computing performance: Game over or next level? *IEEE Computer*, 44(1):31–38, Jan 2011. (Cited on page 1.)

[16] Essayas Gebrewahid, Mehmet Ali Arslan, Andreas Karlsson, et al. Support for data parallelism in the cal actor language. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing (WPMVP'16)*. ACM, 2016. (Cited on pages 24 and 34.)

[17] Essayas Gebrewahid, Mingkun Yang, Gustav Cedersjö, Zain Ul Abdin, Veronica Gaspes, Jörn W Janneck, and Bertil Svensson. Realizing efficient execution of dataflow actors on manycores. In *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC'14)*, pages 321–328, 2014. (Cited on pages 24 and 34.)

[18] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973. (Cited on page 13.)

[19] Jörn W Janneck. *NL–a network language.* ASTG, Processing Solutions Group, Xilinx Inc, 2006. (Cited on page 24.)

[20] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, 1974. (Cited on page 12.)

[21] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. *Information Processing 77*, 1977. (Cited on page 14.)

[22] Andreas Karlsson, Joar Sohl, and Dake Liu. ePUMA: A processor architecture for future DSP. In *Proceedings of the IEEE International Conference on Digital Signal Processing (DSP)*, pages 253–257. IEEE, 2015. (Cited on pages 6 and 27.)

[23] Edward A Lee and David G Messerschmitt. Synchronous data flow. *In Proceedings of the IEEE*, 75(9):1235–1245, 1987. (Cited on pages 12 and 14.)

[24] Edward A Lee and Thomas M Parks. Dataflow process networks. *In Proceedings of the IEEE*, 83(5):773–801, 1995. (Cited on pages 12 and 14.)

[25] David May. The XMOS architecture and XS1 chips. *IEEE Micro*, 32(6):28–37, 2012. (Cited on page 5.)

[26] Małgorzata Michalska, Jani Boutellier, and Marco Mattavelli. A methodology for profiling and partitioning stream programs on many-core architectures. *Procedia Computer Science*, 51:2962–2966, 2015. (Cited on page 35.)

[27] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and computation*, 100(1):41–77, 1992. (Cited on pages 13, 16, and 17.)

[28] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with Epiphany. In *Proceedings of the 48th Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726. IEEE, 2014. (Cited on pages 4, 5, 6, 25, 26, and 33.)

[29] ORCC. Open RVC-CAL compiler, 2017. (Cited on page 8.)

[30] Parallella. Parallella – supercomputing for everyone, 2017. (Cited on page 6.)

[31] Benjamin C Pierce and David N Turner. Pict: A programming language based on the pi-calculus. In *Proof, language, and interaction*, pages 455–494, 2000. (Cited on page 14.)

[32] Sebastian Raase. A dataflow communications library for Adapteva's Epiphany. 2015. (Cited on page 27.)

[33] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. *Journal of the IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):105–119, 2010. (Cited on page 9.)

[34] William Robert Sutherland. On-line graphical specification of computer procedures. Technical report, DTIC Document, 1966. (Cited on page 11.)

[35] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. (Cited on page 1.)

[36] TILE. The TILE-Gx and The TILEPro Processor Family. `http://www.tilera.com/products/processors`, 2017. (Cited on page 5.)

[37] Jeffrey Travis and Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (National Instruments Virtual Instrumentation Series)*. Prentice Hall PTR, 2006. (Cited on page 12.)

[38] Zain Ul-Abdin and Bertil Svensson. Using a CSP based programming model for reconfigurable processor arrays. In *Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig'08)*, 2008. (Cited on page 19.)

[39] Zain Ul-Abdin and Bertil Svensson. Occam-pi as a high-level language for coarse-grained reconfigurable architectures. In *Proceedings of the 18th International Reconfigurable Architectures Workshop (RAW'11) in conjunction with International Parallel and Distributed Processing Symposium (IPDPS'11)*, 2011. (Cited on page 19.)

[40] Carlos A Varela, Gul Agha, Wei-Jen Wang, Travis Desell, Kaoutar El Maghraoui, Jason LaPorte, and Abe Stephens. The SALSA programming language 1.1. 2 release tutorial. *Dept. of Computer Science, RPI, Tech. Rep*, pages 07–12, 2007. (Cited on pages 2, 12, and 13.)

[41] Douglas Watt. *Programming XC on XMOS devices*. XMOS Limited, 2009. (Cited on page 13.)

[42] Peter H Welch and Frederick RM Barnes. Communicating mobile processes. In *Communicating Sequential Processes, The First 25 Years of CPS*, pages 175–210. Springer, 2005. (Cited on pages 12, 14, and 16.)

[43] Peter H Welch, Neil CC Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and extending JCSP. *Communicating Process Architectures 2007*, 65:349–370, 2007. (Cited on page 13.)

[44] Chenxin Zhang. *Dynamically reconfigurable architectures for real-time baseband processing*. PhD thesis, Lund University, 2014. (Cited on pages 6, 7, and 27.)