Compiling Concurrent Programs for Manycores

Essayas Gebrewahid
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 programming productivity and program portability for manycores and to analyze their impact on performance and efficiency. 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 and Ambric.

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. We also present a new CAL compilation framework which has a front end, two intermediate representations and three backends: for a uniprocessor, Epiphany, and Ambric. We show the feasibility of our approach by compiling a CAL implementation of the 2D-IDCT for the three backends. We also present an evaluation and optimization of code generation for Epiphany by comparing the code generated from CAL with a hand-written C code implementation of 2D-IDCT.
Acknowledgments

First of all I would like to express my gratitude for my main supervisor Bertil Svensson, and co-supervisors Veronica Gaspes and Zain-ul-Abdin. Thanks for the support, guidance and encouragement during the course of the work.

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.
List of Publications

The thesis summarizes the following papers


**Contribution:** I have developed and added the P2012 backend to the occam-pi compilation framework, under the guidance of the remaining authors. Based on the idea of the first author, I have implemented the expression of fault-recovery mechanisms based on dynamic reconfiguration of the Platform 2012 (P2012). In the paper, I have contributed text for compilation of occam-pi to P2012, experimental case-study and analysis of the result.


**Contribution:** I have revised the frontend of the Tock framework and the P2012 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.


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


**Contribution:** The first author and I have used 2D-IDCT as an experimental case study to evaluate the Epiphany code generation from CAL, using 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.
List of Figures

3.1 Occam-pi to manycores compiler block diagram ............... 12
3.2 CAL-AM compilation framework. ........................... 14
Chapter 1
Introduction

1.1 Motivation

Over the past two decades, we have witnessed the emergence of multicore and manycore systems. Today, personal computers, smartphones, and tablets are using multicore hardware platforms. Embedded systems in general are also shifting to parallel hardware platforms. This is because increasing performance in a single-processor by increasing the clock frequency requires high power and results in more heat generation. In the mid-2000s, power consumption and thermal dissipation reached a physical limit. On the other hand, multiple processors that run in parallel at lower frequencies generate less heat while sustaining the required performance. This makes parallelism a feasible way to achieve high performance and continue to take advantage of Moore’s law [11]. This has led to a shift in computing systems from single core to manycore. The downside of this development is that, in order to take full advantage of the advances in the hardware, programmers often have to redesign their programs —“the free lunch is over” [25].

In order to program these emerging manycores, we need programming models that capture the parallel nature of applications, that provide high-level abstractions to avoid concurrency issues such as deadlocks and race conditions, and that abstract the underlying manycore to enable efficient compilation and execution. However, writing correct and efficient parallel programs is a challenging task.

The first challenge in developing parallel programs involves decomposition of the application into tasks that run in parallel on the underlying parallel hardware. There are several ways to decompose an application, all with their own advantages and disadvantages. Depending on the organization of the tasks, decomposition paradigms can be classified as functional or domain decomposition [29]. These two types of decompositions differ in terms of their focus on the partitioning process. Functional decomposition focuses on the algorithm and procedures of the computation whereas domain decomposition focuses on
the input or output data of the computation. Functional decomposition breaks down the computation into tasks that each performs a portion of the whole computation. In domain decomposition, first the data (input and/or output) of the computation is partitioned into pieces, then the computation is divided up into parallel tasks that each uses or generates its piece of the data. The choice of the decomposition strategy depends on the nature of the application. Functional decomposition can be suitable for applications with a set of small independent procedures, and domain decomposition for applications with huge amounts of data.

While decomposition allows programmers to express parallelism, overdoing it can decrease the performance of the system due to increased overhead in communication and synchronization.

The second challenge is designing the synchronization and communication mechanisms for the tasks generated by the decomposition process. Although tasks run concurrently, they usually require data from other tasks. Depending on the mechanisms for data communication, programming models can be classified into *Message Passing* and *Shared Memory*. In the message passing model, tasks have encapsulated private address spaces, and the communication is explicit using send and receive primitives. This model involves programming efforts since programmers are required to identify tasks and their communication patterns. However, the absence of a shared address space results in higher scalability. In the shared memory model, all tasks share a global address space, and communication is implicit using memory. However, synchronization primitives, such as locks and barriers, must be defined explicitly. Since there is no private address space, a write to a memory is visible to read operations of all tasks. Due to this, the shared memory model requires sophisticated memory management models and therefore has limited scalability.

Communication and synchronization mechanisms are an essential part of parallel programs and add an extra overhead that decreases the overall performance. Programming models and parallel decompositions cannot ignore these effects.

There is yet another challenge in developing parallel applications, namely exploiting the available parallelism and the particular features of the underlying manycore architecture. The parallelism can be instruction, data and task level parallelism. *Instruction level parallelism* can be exposed by pipelining, where multiple instructions are used simultaneously and also by the number of issued and executed instructions in a single clock cycle. In *data level parallelism* the same instruction is performed on multiple data items in parallel, 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 inside the 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 parallelism are usually
exploited by executing the instruction stream on a number of processors. In addition, embedded manycores usually have hardware support for a specific application area at the cost of loss of generality and flexibility. For example, manycores that target multimedia processing often support data parallelism by processors with a SIMD based instruction set that provides for high performance for the target application.

1.2 Problem Statement

Recent advances in hardware with the emergence of manycores pose 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 and program portability for manycores and to analyze their impact on performance and efficiency. We contribute to the possibility of using high-level languages by implementing compilation layers that provide support for exploiting the underlying parallel hardware. We express the inherent parallelism of applications using occam-pi and CAL, and map them on manycore architectures.

1.3 Research Approach

The process of developing efficient parallel programs, i.e., task decomposition and understanding low-level architectural details to design efficient communication and synchronization mechanism, is time-consuming, prone to error and results in less portable applications. In addition, since the number and the types of cores are increasing dramatically, programmers will soon no longer be experts in the architectural details of various manycores. Due to this, programmers need the help of compilation tools to exploit the particular features of the underlying parallel hardware, to enable retargetability, and to increase productivity and performance.

Unlike programming single core system (the programmer focuses on writing correct code and depends on the compiler for efficient code generation) when programming manycores, due to restricted tool support, the programmer usually performs extra tasks to facilitate the compilation process. The extra tasks may include decomposition—partitioning of the overall task, mapping—deciding 'where' the task should run, scheduling—'when' the task should run and code generation—hand tuning the generated code even at assembly level.

An approach to deal with this difficulty is to use a high-level programming language that provides means to express the parallelism in the application and to push the complexity of exploiting a parallel architecture to a compiler and accompanying tools. The goal of the thesis is to develop a compilation framework for embedded manycore architectures to reduce the effort and the
involvement of application developers without compromising the performance of the resulting implementation. The four papers in this thesis present two compilation frameworks, one for occam-pi and one for CAL. For occam-pi, we have extended the Tock [2] 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. The frameworks generate code in the proprietary language of the target architecture and can then use the native development tools to generate machine code.

1.4 Contribution

The contributions of the thesis are:

- Identification and implementation of abstraction layers of compilers for manycores [Paper B, C].
  - We have contributed to the development of a new compilation framework for CAL.
    We have developed an intermediate representation and added backends for Ambric [9] and Epiphany [4] manycore architectures [Paper C].

- Optimization of code generation mainly by efficient utilization of memory and DMA engines [Paper B, D].
  We have revisited the Tock front end 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 evaluated Epiphany code generation and optimized the utilization of local memory [Paper D].

- Enabling fault tolerance by dynamic reconfiguration using high-level constructs [Paper A].
  We have used the occam-pi language constructs such as dynamic process invocation, process placement and mobile channels to express fault recovery via run-time reconfiguration.
Chapter 2
Background

2.1 Manycore Architectures

Nowadays, manycores are gaining acceptance as high performance computing (HPC) systems for the ever more complex and computationally demanding applications in various areas, like scientific computation, signal processing, large antenna systems, imaging and audio/video processing. The thesis focuses on embedded manycore architectures which are suitable for coarse-grained task parallel applications. Examples of such architecture include STHorm [7], Adapteva’s Epiphany [4], Ambric [9], Tilera [1] and XMOS [21]. These architectures encompass an array of processing elements that work independently, but communicate with each other via network-on-chip and/or shared memory. Usually, manycores are structured in tiles that may contain cores, caches, local memory, network interfaces and some hardware accelerators. Compared to conventional processors, a core in embedded manycores operates with low frequency, limited power, and low memory bandwidth. For case studies, this thesis uses three manycore architectures: STHorm aka the Platform 2012 (P2012) [7], Ambric [9] and Epiphany [4].

STHorm [7] 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 are built by a customizable 32-bit RISC, a 7-stage pipeline, and a dual-issue core called STxP70-v4. The cores have private L1 program caches and share L1 tightly coupled data memory. The cores also share an advanced DMA engine and a 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 low-level C-based API, a Native Programming Model and industry standard programming models such as OpenCL and OpenMP.
Ambric [9] is a massively parallel array of bricks with a total of 336 processors. Each brick comprises two pairs of Compute Unit and RAM Unit. The Compute Unit consists of two Streaming RISC (SR) processors and two Streaming RISC processors with DSP extensions (SRD). The Compute Unit also has 32-bit channel interconnect for inter-processor communication or inter-CU communication. The RAM Unit consists of four independent banks of RAM with 512 words. Ambric has low power (6-12w) and small memory (2KB per processor) footprints, 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 programed using structural object programming model by languages called aJava and aStruct.

Epiphany [4] 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, 32KB globally accessible local memory, network interface and 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 address space 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.

2.2 Concurrent Models of Computation

A Model of Computation (MoC) is a high-level abstraction that defines the type and semantics of operations used for computations. The absence of low-level details in a MoC helps both application developers and hardware designers. 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. There are a number of concurrent models of computation, in this chapter we present those that influence our work.

Dataflow

The dataflow model [24] was introduced as a visual programming language by Sutherland in 1966. In the dataflow model an application is organized as a flow of data between the nodes of a directed graph. The nodes are computational units; usually they are called actors or processes. Edges create the dependencies between nodes by connecting explicitly defined inputs to outputs. The streams of data that flow through the edges are called tokens. The connection among
the nodes can be one-to-one, one-to-many and many-to-many. The execution of the nodes is constrained only on the availability of the input tokens. Nodes do not have global storage, hence no side-effects. The dataflow model has been adapted in various signal processing applications and has also influenced many visual and textual programming languages such as CAL [10], LabVIEW [26] and SISAL [30]. There are several computational models that define specific semantics for dataflow models, such as Kahn Process Networks and Dataflow Process Networks.

Kahn Process Networks

Kahn Process Networks (KPN) are named after Gilles Kahn who introduced them in 1974 [16]. KPN is a dataflow model where processes communicate by sending tokens via unbounded, unidirectional FIFO channels. Writes to a channel are non-blocking, however, reads from an empty channel block until sufficient tokens are available (a process cannot check the availability of tokens). Thus, KPN cannot model processes that behave based on 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. In addition, the order of execution does not affect the output. KPN processes are monotonic—they only need partial information of the input stream in order to produce partial information of the output stream.

Dataflow Process Networks

Dataflow Process Networks (DPN) [20] are also given as sets of processes that communicate over unbounded FIFOs. However, DPNs extend KPNs by allowing processes to test the availability of input tokens. This means that reads can return immediately, even if the channel is empty. This makes the network non-deterministic. DPN nodes are stateless so called actors. Each actor has a set of firing rules that govern the execution, also called firing, 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 is satisfied at the same time, then only one will be chosen. Since DPN allows polling on channels, it can be considered as more efficient than KPN. DPN is also more expressive than KPN, since it can express non-deterministic and time-dependent actors.

The Actor Model

The actor model [12] 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 [10], Erlang [5] and SALSA [30].

Communicating Sequential Processes

Communicating Sequential Processes (CSP) [13] was introduced in 1978 by Hoare as a set of simple primitives that communicate and synchronize sequential processes that run in parallel. In CSP, processes share nothing, they communicate using synchronous message passing via unidirectional and named channels. Currently, 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 [3], JCSP—CSP for Java [33] and XC—the native language of the XMOS architecture [31].

Pi-calculus

The pi-calculus [22] was introduced by Milner et al. 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 [32] and Pict [23].

2.3 Occam-pi and CAL Actor Language

Occam-pi [32] is a programming language that extends occam by introducing mobility features of the pi-calculus [22]. Occam was developed based on CPS in order to program the Transputer [6] 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. Compared with occam, occam-pi supports

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

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

CAL is an actor-oriented dataflow programming language that extends DPN actors with states. A CAL actor can have input/output ports, parameters, states, and actions. An actor does not have access to the state 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. In addition, CAL firing conditions include dependencies, priorities, and finite state machines. Thus, depending on the use of specific constructs, CAL can support various models of computation, such as Synchronous Dataflow (SDF) [19], Cyclo-Static Dataflow (CSDF) [8], Kahn Process Networks (KPN) [17], and Dataflow Process Networks (DPN) [20].

Discussion

A major bottleneck in modern embedded manycore architectures is the memory bandwidth: the rate of retrieving data is much lower than the rate of executing an instruction. To overcome this problem the architectures have leaned towards a distributed memory organization, and application developers try to achieve high data locality. The semantics of the concurrent models of computation described in Section 2.2 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.

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 [22], 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 inputs to outputs. In contrast with 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 3

Summary of Papers

This section presents an overview of the compilation processes for occam-pi and CAL with a summary of the appended papers.

3.1 Compiling Occam-pi for Manycores

To enable programming manycores using occam-pi we have extended the Translator from occam to C from Kent (Tock) [2], Fig 3.1. 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 parallel assignment into 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 [27] and XPP [28] architectures using occam-pi. In paper A and B, we perform additional extensions to Tock by adding a new backend for STHorn aka the Platform 2012 (P2012) [7].

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 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.

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 STHorn 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.

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 occam-pi's 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.

3.2 Compiling CAL Actor Language for Manycores

We have used CAL to program two manycores and a general purpose CPU [Paper C]. The compilation framework, Fig 3.2, has two intermediate representations (IRs): Actor Machines (AM) [15] and Action Execution Intermediate Representation (AEIR). Each CAL actor is translated to an AM that is then translated to AEIR. Finally, the AEIR and the description of the network of actors are used by the three backends to generate target-specific code.

An AM describes how to schedule the testing of conditions and the execution of actions. It consists of states that have knowledge about conditions and a set of AM instructions that can be performed on the state. The 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 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. Thus, we have chosen to introduce an **Action Execution Intermediate Representation** (AEIR) that brings us closer to a 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 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. Currently we have three backends: a uniprocessor backend that generates sequential C code for a single general purpose processor, an Epiphany backend that generates parallel C code for Epiphany, and an Ambric backend that generates aJava and aStruct code for the Ambric massively parallel processor array. We also have two types of code generation, inlined and non-inlined. In the inlined code generator, the action calls (AM exec instructions) are inlined. In addition, it performs analysis on the scope of variables to remove unused initializations. The non-inlined version does not perform any optimizations. In both versions AM test instructions are translated to if statements.

We have used the Network Language (NL) [14] 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.

For sequential C and Epiphany we have developed a custom communication library, but for Ambric we have used the available support for channels, communication APIs and for 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
3.2. **COMPILING CAL ACTOR LANGUAGE FOR MANYCORES**

write transactions (writes are faster) and the use of DMA to speed up memory transfers.

Paper C shows the feasibility and portability of our approach by compiling a CAL implementation of the Two-Dimensional Inverse Discrete Cosine Transform (2D-IDCT) for a general purpose processor, for the Epiphany manycore architecture [4] and for the Ambric massively parallel processor array [9]. The 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 the general purpose processor (sequential version) we used both inlined and non-inlined code generators. For 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.

Performance has been measured by execution on real hardware. 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 Epiphany has also improved the corresponding sequential version by 30%. However, the performance of the optimized inlined version is still better 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 the sequential 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, we evaluate the communication library and the code generation for Epiphany. We again use the two-dimensional inverse discrete cosine transform (2D-IDCT) and compare 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 the 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: \texttt{action\_body} to implement the body of the action and \texttt{action\_guard} to evaluate the guard of the action. Thus, the second optimization inline \texttt{action\_guard} and the third optimization inlines both \texttt{action\_body} and \texttt{action\_guard}.
Initially, the hand-written implementation had 4.3x better throughput performance than the 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 hand-written implementation. Combining all optimizations we were able to reduce the difference in execution time down to a factor of only 1.3x.

To estimate the development effort, we have compared the number of source lines of code (SLOC) used in the CAL program and in the hand-written implementation. In total, 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. 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.
Chapter 4
Conclusions and Future Work

Conclusions

With the arrival of manycore architectures with substantial capabilities for parallelism, software development needs new methodologies, new 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 STHorn. The STHorn 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 STHorn 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. The current CAL compilation framework has a front end, two intermediate representations and three backends: a uniprocessor backend that generates sequential C code for a single general purpose processor, an Epiphany backend that generates parallel C code for Epiphany, and an Ambric backend that generates aJava and aStruct code for Ambric massively parallel processor array. We have shown the feasibility of our approach by compiling a CAL implementation of the 2D-IDCT for the three backends. We have compared our Epiphany code
CHAPTER 4. CONCLUSIONS AND FUTURE WORK

generation from CAL with a hand-written C code implementation of 2D-IDCT; and we have performed a detailed evaluation and optimization on Epiphany's code generation and on the custom communication library which was developed for Epiphany.

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 and fault tolerance aspects of parallel applications. We have used high-level abstractions like actor machines to increase portability and low-level abstractions in the backends to increase efficiency. The identification of different levels of abstraction has created more room for optimization, analysis, and transformation processes.

Future Work

The thesis has presented the highlights of recent work, and there are a number of issues that must be addressed in the future.

We plan to focus on CAL compilation framework and evaluate our work using more complex case-studies to strengthen our results, such as MPEG-4 simple profile decoder and signal processing applications related to large antenna systems, aka Massive Multiple Input Multiple Output (MIMO). 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.

To this end, we have targeted commercial architectures, Epiphany and Ambic, and our compilation tool generates native code and use native development tools to generate machine code. In near future, we would also like to evaluate our work using two SIMD-based manycore architectures: EIT [34] and ePUMA [18], and generate the machine code for these architectures.

The action execution IR in our CAL compilation framework can easily be used to generate imperative code, but to generate machine code it requires extra effort. To reduce this effort, we have planned to generate directed acyclic graphs (DAGs) from CAL Actor Language using LLVM infrastructure\(^1\). This will facilitate the low-level instruction selection and scheduling processes. To gain more benefit from the LLVM infrastructure, we will extend the LLVM instruction set with low-level primitives that model concurrency using the dataflow model and generate LLVM instruction sets instead of native code. This will enable us to reuse LLVM's compilation, optimization, analysis and

\(^1\)http://llvm.org/
code generation tools. In addition, the Static Single Assignment form of the LLVM intermediate representation will give us an opportunity to experiment with a number of dataflow analyses and optimizations. Furthermore, we expect to exploit vector operations and optimizations in LLVM when compiling lists and list operations of CAL for the two SIMD-based manycores.

Programming with high-level languages like CAL and using the LLVM instruction set which is language and hardware orthogonal, will enable us to increase the software development productivity and to support a wide range of manycore architectures. After this, the next step will be to exploit performance and efficiency of manycores without compromising productivity and portability. To do this, we have planned to extend our compilation tool via high-level domain-specific IRs positioned just above the LLVM IR and architecture description languages below the LLVM IR. This will create an additional layer for the optimization, analysis, and transformation processes.
References


the CAL actor language. Technical Memorandum UCB/ERL M03/48,
University of California, Berkeley, CA, USA, 2003. (Cited on pages 7
and 8.)

over or next level? IEEE Computer, 44(1):31–38, Jan 2011. (Cited on
page 1.)

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ac-
tor formalism for artificial intelligence. In Proceedings of the 3rd Interna-
tional Joint Conference on Artificial intelligence, pages 235–245. Morgan


Group, Xilinx Inc, 2006. (Cited on page 14.)

In 45th Annual Asilomar Conference on Signals, Systems and Computers

In Proceedings of the IFIP Congress 74, pages 471–475, 1974. (Cited on
page 7.)

[17] Gilles Kahn and David MacQueen. Coroutines and networks of parallel

[18] Andréas Karlsson, Joar Sohl, Jian Wang, and Dake Liu. ePUMA: A
unique memory access based parallel DSP processor for SDR and CR.
In Global Conference on Signal and Information Processing (GlobalSIP),


[21] David May. The XMOS architecture and XS1 chips. IEEE Micro,

[22] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile
processes, II. Information and computation, 100(1):41–77, 1992. (Cited
on pages 8 and 9.)


[27] Zain Ul-Abdin and Bertil Svensson. Using a csp based programming model for reconfigurable processor arrays. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig’08)*, 2008. (Cited on page 11.)


