An LLVM compiler for CAL

Master’s Thesis in Embedded and Intelligent Systems

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

September 2011
Preface

This thesis is the concluding part of the Masters Program in Embedded and Intelligent systems with specialization in embedded systems at Halmstad University, Halmstad, Sweden. Firstly we would like to thank our supervisor Veronica Gaspes for helping us with this thesis. She was very helpful and always available for the queries we had during the thesis work. We are also thankful to Jerker Bengtsson. He helped us a lot to improve the thesis work by giving his positive feedback.

This thesis work is dedicated to people all over the world suffering from natural or human made crisis.

Haseeb Ullah & Amir Mofakhar

Halmstad University, September 2011
An LLVM compiler for CAL
Abstract

Massively parallel architectures are gaining momentum thanks to the opportunities for both high performance and low-power consumption. After being a matter for experiments in academia, manycores are now in production and industries with products that rely on high performance, for example, in the field of telecom and radar, are in the process of adopting them. In order to encourage adoption, there is a need for compiler technologies that can make appropriate use of the opportunities of these architectures.

Programming languages with constructs that the compiler can easily identify as parallel are a reasonable starting point for these technologies. For example, in CAL (Caltrop Actor Language) [1], a program is organized as a network of actors and these actors can be executed in parallel. As a first approach, if there were enough processors for all actors in the network, each actor could be mapped on to one processor.

Writing a compiler is a comprehensive software engineering project and there are a number of tools, data structures and algorithms that have to be chosen and that facilitate the task. Some tools are lexer and parser generators, some data structures are abstract syntax trees and intermediate representations for code generation, some of the algorithms are for analyzing properties of the program and for translating between different data structures. LLVM (Low Level Virtual Machine) [7] is a compiler infrastructure that offers a well defined language and target independent intermediate representation for programs. LLVM also provides compile-time, link-time and run-time optimization.

This report discusses the front end of a compiler for CAL, including generation of code to LLVM. The work described in this report translates CAL programs into LLVM's intermediate representation and includes a runtime system that handles intercommunication links between actors and actor scheduling. This is an intermediate step in generating LLVM code for parallel architectures. With this compiler, application developers that need to evaluate different parallel platforms will be able to have their applications written in CAL and only the back-end, form LLVM to the platform, will need to be re-programmed.
An LLVM compiler for CAL
## Contents

**PREFACE** ........................................................................................................................................ III

**ABSTRACT** ..................................................................................................................................... V

**TABLE OF FIGURES** ........................................................................................................................... IX

**1. INTRODUCTION** ............................................................................................................................ 1

1.1. Compiler for multi-core platform .............................................................................................. 1

1.2. Caltrop actor language (CAL) .................................................................................................. 2

1.3. Low level virtual machine (LLVM) ........................................................................................... 2

1.4. Motivation .................................................................................................................................. 3

1.5. Goals and objectives ................................................................................................................. 3

**2. HISTORY AND BACKGROUND** .................................................................................................. 5

2.1. Caltrop actor language (CAL) .................................................................................................. 5

2.1.1. History of CAL ....................................................................................................................... 5

2.1.1.1. Ptolemy II ......................................................................................................................... 5

2.1.2. Guarded actions in CAL ....................................................................................................... 9

2.1.3. Actions with state in CAL ................................................................................................... 10

2.1.4. Schedule statement in CAL .................................................................................................. 10

2.1.5. Priority statement in CAL .................................................................................................... 11

2.1.6. Example in CAL ................................................................................................................... 11

2.1.7. Lexical tokens in CAL .......................................................................................................... 12

2.1.7.1. Keyword ........................................................................................................................... 12

2.1.7.2. Identifiers .......................................................................................................................... 12

2.1.7.3. Operators ........................................................................................................................ 12

2.1.7.4. Delimiters ........................................................................................................................ 12

2.1.7.5. Comments ....................................................................................................................... 12

2.1.8. CAL industrial application .................................................................................................. 12

2.1.8.1. Reconfigurable Video Coding (RVC) ........................................................................... 13

**3. TOOLS AND METHODOLOGY** .................................................................................................. 15

3.1. Lexical analysis ......................................................................................................................... 15

3.1.1. JFlex: lexical analyzer generator ....................................................................................... 16

3.2. Syntactic analysis ...................................................................................................................... 17

3.2.1. JACC: a parser generator .................................................................................................... 18

3.3. Semantic analysis ...................................................................................................................... 21

3.4. Intermediate code generation .................................................................................................. 22

**4. IMPLEMENTATION** ..................................................................................................................... 23

4.1. HHCal ......................................................................................................................................... 23

4.1.1. Lexer ................................................................................................................................... 23

4.1.2. Parser ................................................................................................................................... 25

4.1.3. Code generator ..................................................................................................................... 27

4.1.4. Limitations in HHCal .......................................................................................................... 28

4.1.4.1. Limitations in connection ............................................................................................. 28

4.1.4.2. Constant parameters ...................................................................................................... 28

4.1.4.3. Actions with state ........................................................................................................... 28

4.1.4.4. Limitations with type ...................................................................................................... 28

4.1.5. How to use HHCal .............................................................................................................. 29
4.2. HHHNETWORK
4.2.1. Network of actors
4.2.2. Runtime system
4.2.2.1. Scheduling of actors
4.2.2.2. Communication strategies
4.2.2.3. Buffer selection for implementation
4.2.3. How to use HHHNetwork
4.3. EVALUATION
4.3.1. Matgen (Matrix generator)
4.3.2. Feedback
4.3.3. Multiplexer
4.3.4. Multiply
4.3.5. Feedback2
4.3.6. Rowcol
4.3.7. Interconnection and working
5. CONCLUSION AND FUTURE WORK
5.1. CONCLUSION
5.2. FUTURE WORK
REFERENCES
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Simple actor in CAL</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Actor with guarded actions</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Actors with state</td>
<td>10</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Example of CAL actor</td>
<td>11</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Overview of HHCal</td>
<td>23</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Network of actors (simple)</td>
<td>30</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Network of actors</td>
<td>32</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Network of actors with buffers</td>
<td>33</td>
</tr>
<tr>
<td>Figure 9</td>
<td>MATGEN.CAL</td>
<td>35</td>
</tr>
<tr>
<td>Figure 10</td>
<td>FEEDBACK.CAL</td>
<td>36</td>
</tr>
<tr>
<td>Figure 11</td>
<td>MULTIPLEXER.CAL</td>
<td>36</td>
</tr>
<tr>
<td>Figure 12</td>
<td>MULTIPLY.CAL</td>
<td>37</td>
</tr>
<tr>
<td>Figure 13</td>
<td>FEEDBACK2.CAL</td>
<td>37</td>
</tr>
<tr>
<td>Figure 14</td>
<td>ROWCAL.CAL</td>
<td>38</td>
</tr>
</tbody>
</table>
1. Introduction

This report discusses the front end of a compiler for CAL (Caltrop Actor Language). This front-end compiler translates CAL program into LLVM (Low Level Virtual Machine) Intermediate Representation. This compiler is designed to work on a single core platform. The compiler has to translate individual CAL actors to LLVM also it has to implement a run time system that will map the actors onto a single core. The runtime system creates the communication links for communication between different modules of CAL language and schedules the actors in a way that every actor in the program can use the processing resource.

1.1. Compiler for multi-core platform

The increase in performance of computational systems is being a hot research topic since the emergence of processors. Different tasks were processed in parallel on a single core using concepts of threads. The parallel processing has gone one step further with the concept of multi-core system. In a multi-core system, different tasks are processed on different cores.

New programming languages or some modifications to the existing ones are required to write programs for multi-core platforms. There should be some extra support and functionalities in the programming language as compared to the ones that are used for single-core platforms. The programming language should have parallelism support for the programmer. It should allow users to write independent executable modules instead of sequential executable modules. If there will be some independent executable modules, then those modules can be mapped onto a multi-core system to save time and increase performance.

Compilers are the tools to translate the source code into machine language. The techniques used to write compiler for single-core systems are not enough to write compiler for multi-core systems. A compiler for multi-core system not only has to translate the code into machine language but also has to manage the communication between different cores to make sure that the dependent data is safely and securely transferred from source core to destination core. For systems with shared memory, the compiler also has to device ways to manage memory access.
1.2. Caltrop Actor Language (CAL)

CAL [1] was developed by Johan Eker and Jorn W. Janneck at the university of California, Berkley. CAL is a domain specific language aimed at providing a medium to define actors.

Actor is a combination of computational statements on input tokens, and after manipulation it produces output tokens. The execution of actor takes place in steps. The process is called firing. Each firing takes place on certain actor state. As a result of the firing process the input tokens are consumed. Finite output tokens are generated on each output port after firing process. The state of the actor is determined by the input sequence and conditional statements in the actor.

Several actors are joined together to form a graph-like network structure. The output ports of the actors are joined with the input ports of the same or different actor. This means that the output generated by one actor can be input for other actors. That is why it is easy to execute different actors on different cores because they are independent and can perform their task whenever the input will be available.

This programming language is well suited for writing parallel programs especially for multi-core systems. Each actor in CAL is an independent entity and each actor can be mapped to separate processing core. Compiler will have to make connections for communication between actors.

1.3. Low Level Virtual Machine (LLVM)

LLVM [7] is a compiler infrastructure designed for compile-time, link-time and runtime optimisations of programs written in any language. The LLVM project was started by Vikram Adne and Chris Lattner in 2000 at the University of Illinois at Urbana-Champaign. The main development idea of LLVM was to provide a research platform to investigate dynamic compilation techniques for dynamic and static languages.

Compiler infrastructure provides modular and reusable components for building compilers. These components mainly include libraries to compile a code of source programming language. These components ideally should be independent of language and platform. It reduces the time and cost of building a compiler as it provides built in libraries.

The LLVM includes a well defined and organised IR (Intermediate Representation) [7]. LLVM-IR is easy to use and target and language independent intermediate representation. It can be used for any programming language. LLVM has built in
libraries for optimisation, modular code generation, JIT compilation, profiling, debugging, link-time optimisation and code generation.

LLVM code representation is designed to be used in three different forms i.e. in-memory compiler IR, on-disk bit code representation and human readable assembly language. By virtue of these three forms, LLVM provides a powerful IR for efficient compiler transformations and analysis also a natural mean to debug and visualise the transformations. The LLVM-IR aims to be light-weight and low-level but still being extensible and expressive. It aims to be a “Universal-IR” by being at a low-enough level that almost all high level ideas can easily be mapped onto it.

The LLVM system architecture is designed to produce the highest performance executables through an aggressive system of continuous optimization. The program representation that LLVM uses is the key feature that differentiates it from other systems. This program representation must be low-level enough to allow significant amounts of optimization in the early phases of compilation, while being high-level enough to support aggressive link- and post-link time optimizations. The LLVM instruction set is a low level representation with a high level type information. It provides type information about every value in the program and exposes the memory allocation to the compiler [11].

The LLVM instruction set holds the system together. It is a low level representation with a high level type information. This provides the benefits of a low-level representation as well as providing high-level information to support aggressive inter-procedural optimizations at the link and post-link time. LLVM is designed for run-time and idle time optimisations on the machine [11].

1.4. Motivation

This thesis work is a sub part of a massive research project carried out at CERES (Centre for Research on Embedded Systems) Halmstad University in collaboration with Lund University and Linkoping University. Other groups are working on the back end compiler which will translate the LLVM code, generated by this thesis, to multi-core platform. I.e. they will map this compiler onto multi-cores. This compiler will be modified to map it onto multi-cores. Some parts of the compiler can be omitted while mapping but still this compiler will provide a working and tested starting point to them.

1.5. Goals and Objectives

The work presented in this thesis consists of.
Front-end for CAL resulting in LLVM assembler

This part is a front end of a compiler which translates CAL source code into LLVM assembler. This part of the thesis consists of laxer generator, parser generator and code generator. Laxer generator scans the source code for actor(s) written in Cal language. It looks for illegal characters and generates errors. The source file will be passed to the Parser if there is no error. The parser will look for the allowed statement syntax in the source file. If there is any syntax error the parser will generate an error message. Otherwise, parser will generate tokens and these tokens will be passed on to Code generator. Code generator does type checking and produces LLVM assembler instructions.

A runtime system for CAL in LLVM

The first part will generate LLVM assembler instructions for the actors. There is a need for the runtime system. This run time system generates threads and codes for intercommunication of actors in CAL language. The actors are linked in the form of a network. This network is described in a different file when writing code in CAL. The network will be translated by this run time system and the resources required to perform operations in this regard will be made available by this module.

A test case

The last part of the thesis consists of a CAL example program. This example is used to test the developed compiler. The test example is an 8x8 parallel matrix multiplier. Matrices are generated by the actors in the program. Each row of the first matrix is multiplied to a specific row of other matrix in separate actor. Each entry of the resultant matrix is generated by individual actor.
2. History and Background

2.1. Caltrop Actor Language (CAL)

CAL is an actor based language. Actors communicate with other actors by message passing. The messages passed are called tokens. The message received by the actor is called input token and the one sent by the actor is called output token. Each actor has some specific task to perform. The tasks are called actions. Every action has a conditional statement, which should be fulfilled for execution of action. These conditional statements are called firing rules and execution of action are called firing in CAL.

Consider the example of CAL actor written in figure 1.

```
actor consmul () input ==> output :
  action mul : [a] ==> output : 
  do
  output:=[a*k];
  end
end
```

Figure 1: Simple actor in CAL

The actor shown in figure 1 is a simple actor that has one input and one output. It multiplies the input with constant and writes the result to the output. The first line declares an actor named consmul and defines the input and output. There is one input ‘input’ and one output ‘output’ in this actor. The second line declares an action named ‘mul’. This action reads the input and multiplies it with the constant ‘k’ and generates the output.

2.1.1. History of CAL

CAL was created by Johan Eker and Jorn w. Janneck at the University of California [1], Berkley as a part of Ptolemy II project to implement directors defined in Ptolemy II [6].

2.1.1.1 Ptolemy II

The Ptolemy [5] project is carried out at the centre for hybrid and embedded software systems (CHESS) [9] in the department of electrical engineering and computer sciences at University of California, Berkley. The project is named after the second
century Greek astronomer, mathematician and geographer Claudis Ptolemaus [9]. The project is headed by Professor Edward A. Lee. This project aims at modelling, simulating and designing concurrent, embedded and real time systems. The main emphasis of this project is on assembling concurrent components. The appreciable outcome of this project is its model based design and simulation tool called Ptolemy II.

Ptolemy II is an open source frame providing a platform for experimentation with actor oriented design. Ptolemy II has been under development since 1996. It is the revised version of Ptolemy classic which was developed since 1990. Collection of java classes and packages provided for increased specific capabilities is the core of Ptolemy II. The kernel supports abstract syntax hierarchical entities with interconnections. Vergil, a graphical editor, is provided to edit this abstract syntax structure.

The main characteristic of the infrastructure is actor abstract semantics. This feature enables the interoperability of distinct models of computation with well defined semantics.

Actors are software components executing concurrently and communicating with each other via messages sent over fixed interconnected ports. Hierarchical interconnection of actors is called a model.

In Ptolemy II the semantics of a model are not described by the platform. They are defined in a software component called director. A model of computation is implemented in director. Each level of hierarchy can have its own directors and directors can be created in a hierarchy. The major emphasis of this project is on understanding the heterogeneous combinations of the models described by the directors. Different directors can be combined with state machines to make big models.

The Ptolemy II supports directors of following communicational model types:

- Process Networks (PN)
- Discrete Events (DE)
- Synchronous Data Flow (SDF)
- Synchronous/Reactive (SR)
- Communicating Sequential Process (CSP)
- Finite State Machine (FSM)
- Continuous Time model (CT)

**Process Networks (PN)**

Process network (PN) communicational model was developed by Mudit Goel. In this model processes communicate by sending messages through channels that can buffer the messages. The sender of the message needs not to wait for the receiver
to be ready to send a message. This communication style is also called asynchronous message passing.

In PN model the interconnections represent the token sequences and the nodes (actors) represent the entities that map the input tokens to output tokens. PN models are relatively easy to implement for parallelism distributed systems because they are loosely coupled. They can be easily implemented on both hardware and software. Hence, provide with many implementation options. One weakness of this model is that this cannot be used for control logic. But combined with other models this weakness can be solved.

**Discrete Events (DE)**

This communication model was developed by Lukito Muliadi. In this model the actors communicate via sequence of tokens placed in time along a real time line. Every token consists of data value and time stamp. Actors can be processes that respond to the tokens or functions that fire when new input sequence is made available to them. This model is best suited for digital hardware and simulation of telecommunication systems. It has been adopted in many simulation tools and hardware descriptive languages. VHDL and Verilog are the example languages that use this model.

DE models are excellent representation of concurrent hardware. Global time phenomenon is problematic task for this model. It over-models the system where maintaining a global time is a difficult task. Every event is placed on globally synchronised time line. On large VLSI chips, with higher clock frequencies, it is difficult to maintain global clock synchronisation.

DE simulators need to store the pending events in the global array according to time stamp. This is quite similar to priority queuing. This task is quite expensive in terms of computation because for every event whole queue has to be searched to find an appropriate place.

**Synchronous Data Flow (SDF)**

This model was developed by Steve Neuendorffter. This model works with the computation that operates on tokens. Data flow models, renowned in signal processing, are a special case of process networks. They make processes of process network as a sequence of atomic actor firing. SDF is a particular model with certain restrictions to decide deadlocks and bounds. The schedule of firing, parallel or sequential, can be computed statically making SDF very useful for embedded-real time systems.

**Synchronous/Reactive (SR)**

In synchronous/reactive (SR) computational model the interconnections represent data values that are strictly synchronised with global clock cycles. They are discrete
An LLVM compiler for CAL

signals but unlike discrete time signal they do not need value at every clock cycle. The nodes/actors represent the relation between input tokens and output tokens at every clock cycle. The languages that are built upon this model include Esterel, Lustre, and Argos.

This model is best suited for applications with concurrent and complex control logic. This model is the best choice for safety-critical real-time systems because of tight clock synchronisation. But some applications can be over-modelled because of this tight clock synchronisation. In some cases modularity has to be sacrificed to have tight time synchronisation.

**Communicating Sequential Process (CSP)**

CSP Ptolemy model was created by Neil Smyth. In this model actors represent concurrently executing processes. The processes communicate via instantaneous atomic events called rendezvous. Rendezvous is sometime called synchronous message passing. If two processes need to communicate than the first one will get ready to communicate and will wait for the second one to get ready. The data exchange is completed in single uninterruptable step. Both of the processes are involved in communication at the same time that is why it is called atomic process. Lotos and Occam are the two concurrent languages based on this model.

Rendezvous models are best suited for systems where resource sharing is an important element. Examples of these systems include client-server database models and multitasking or multiplexing of hardware. Cleanly modelling of non-determinate interactions is the key feature of this model.

**Finite State Machine (FSM)**

FSM computational model was written by Xiaojun Liu. This model is quite different from the other models in Ptolemy II. The nodes in this model represent the state instead of actors. The interconnections represent the transitions between the states. Firing or execution is a strict sequence of state transitions. This model has a big advantage of evaluating guard statements, which determines the state transitions.

FSM model is best suited for embedded systems specially control logic and safety-critical systems. FSM on the other hand also has some weaknesses. One of the weaknesses is that it is not expressive as other models. It is also poor to describe recursive functions. But these problems can be resolved by using FSM in combination with other models.

**Continuous Time model (CT)**

CT model was created by Jie Liu. In this model the actors represent the entities that interact to each other via continuous time signals. Actors typically in this model specify algebraic or differential relations between input and output tokens. The task of the director is to find set of continuous-time functions that satisfy all the relations.
The CT domain includes huge set of the differential equation solvers. This model is therefore very useful for modelling physical systems with linear or non-linear algebraic differential equation descriptions e.g. analogue circuits and mechanical systems. Simulink, Saber, VHDL-AMS and Spice circuit simulator use computational model quite similar to CT model.

Physical systems often have simple models that are valid over specific platform. But on other platform some other model is useful for those systems. A modal model is one that switches between models whenever there is a transition from one regime to other. The CT model combined with FSM model creates a modal model.

CAL was developed by researchers working with Ptolemy ii project to implement the above stated directive models. These models are purely actor based. So they developed an actor based language CAL. Some key features of CAL language are discussed below.

2.1.2. Guarded actions in CAL

The most common firing condition for actions is that there are sufficiently many tokens for them to consume, as specified in their input patterns. In some cases additional criteria need to be satisfied for an action to fire, for example, dependent on the values of the tokens, or the state of the actor, or both. These conditions can be specified using guards.

```
actor consmul (k,l) In == out:
  action [a] == out : guard a >= 100
    do
      out := [a*k];
    end

action [a] == out : guard a < 100
  do
    out := [a*l];
  end
end
```

Figure 2: Actor with guarded actions

The guard clause of an action contains a number of expressions that all need to be true in order for the action to be fireable. For the first action to be fireable the input value should be greater than 100 and in this case the input will be multiplied with the constant ‘k’. Otherwise this action cannot be fired. Conversely, for the second action to be fireable, the value of the token needs to be less than 100 and in this case the input will be multiplied by constant ‘l’.
2.1.3. Actions with state in CAL

Using state variables, action firings can leave information for oncoming firings of either the same or a different action of the same actor. A simple example of this is the Sum actor:

![Figure 3: Actors with state](image)

In the example shown in figure 3, the actor maintains a variable in which it accumulates the sum of all tokens it has consumed.

2.1.4. Schedule Statement in CAL

Consider the actor with a finite number of states, and each state is forwarded to another state when one of the actions is fired, the current state depends on the guard statements and previous state. Calling the actions in a specific and deterministic order is the target of the state machine in the CAL actor. Using the schedule keyword in the CAL language permits to determine the transitions between different states.

Each state transition consists of three parts: the original state, a list of action tags, and the following state.

\[ \text{str (A)} \rightarrow \text{B}; \]

str is the original state, A is an action tag and B is the following state.

Let us consider an actor with one input and two outputs, in which we want to send the input value alternatively to the first and the second output ports. This actor can be written without internal states or guard conditions like this [3]:

```cal
actor Split () Input ==> out1, out2:
A: action [a] ==> out1: [a] end
B: action [a] ==> out2: [a] end
Schedule fsm S1:
S1(A) → S2;
S2(B) → S1;
end
end [3]
```
2. History and Background

2.1.5. Priority statement in CAL

If two actions have the right to execute from the one state to move to another state, the CAL engine executes the higher priority action.

The priority statements in CAL language are written in this syntax format:

```
priority
A > B;
B > C;
End
```

Here A, B and C represents the actions.

2.1.6. Example in CAL

The actor shown in figure 4 generates a number sequence. It multiplies the input with two and writes to the output.

The actor named matgen has one input and two outputs. The input and one of the outputs is used for feedback. The input and both of the outputs are in integer type. Actor has two actions. The first action is always ready to fire as it is not dependent on the input. It writes one to both outputs. The second actor waits for the input and then multiplies input by two and writes to the output. If the input is greater than 98 it resets the output to one.

```
actor matgen() int in1 --> int out1,int out2:

    action --> out1,out2
        do
            out1:=1;
            out2:=1;
        endaction

    action in1 ==> out1,out2
        do
            if (in1>98) then
                out1:=1;
                out2:=1;
            else
                out1:= in1 * 2;
                out2:= in1 * 2;
            endif
        endaction

Figure 4: Example of CAL actor
```
2.1.7. Lexical tokens in CAL

Lexical tokens help the user to understand the functionality provided by any language. CAL has the following kinds of lexical tokens.

2.1.7.1. Keyword
Keywords are a special type of identifiers. They are already reserved in the programming language by default. These keywords can never be used as identifiers in the code. Some of these keywords are “action”, “actor”, “begin”, “else”, “if”, “while” and “true”. These keywords have some specific fixed tasks or meaning assigned to them.

2.1.7.2. Identifiers
Any sequence of Alphabetic characters of any case, digits, the underscore character and the dollar sign that is not a keyword is called identifiers. The sequence of characters that are not legal identifiers can be turned into legal identifiers by delimiting them with a backslash.

Users are discouraged from using identifiers containing dollar sign ($). Because identifiers containing dollar sign ($) are reserved identifiers. These are intended to be used by the programs that generate CAL program codes. Names generated by these programs do not conflict with the names chosen by the user of CAL language.

2.1.7.3. Operators
Operators usually represent mathematical, logical or algebraic operations. Operators are written as any string of characters ‘!’, ‘@’, ‘#’, ‘$’, ‘%’, ‘^’, ‘&’, ‘*’, ‘/’, ‘+’, ‘-’, '=', ‘<’, ‘>’, ‘?’ ‘~’, ‘|’ etc. In addition to these strings some identifiers also can be used as operators. The CAL language itself defines very few operators.

2.1.7.4. Delimiters
Delimiters are used to indicate the start or the end of this syntactical element in the CAL. Following elements are used as delimiters: ‘(‘, ‘)’, ‘{‘, ‘}’, ‘[‘, ‘]’.

2.1.7.5. Comments
Comments in CAL language are same as in Java, C and C++. Single line comments start with ‘//’ and multiple line comments start with ‘/*’ and end with ‘*/’.

2.1.8. CAL industrial application

CAL was developed at the university of Berkley as a part of the Ptolemy project [1]. It was used to implement the actor models defined in Ptolemy II project. These actor models are discussed in Ptolemy II section above in the report. CAL is getting attention of industry as a full fledged programming language. CAL is being extended by the users as per their requirements. One good example to support the fact that CAL is getting a place in the market as full fledged programming language is the new
video coding tool by MPEG. In this tool MPEG used CAL as core programming language.

2.1.8.1. Reconfigurable Video Coding (RVC)

Reconfigurable Video Coding (RVC) is a new video coding framework developed by the MPEG and standardized by ISO/IEC in 2009[8]. RVC is an open framework that is capable of specifying and reconfiguring video codec’s by connecting a number of functional units (FUs) called video coding tools.

The RVC framework is built on top of three languages:

- **RVC-CAL**: A constrained set of CAL (caltrop actor language), a dataflow programming language, is used for modelling functional units (FUs) i.e. video coding tools.
- **FNL**: FU Network Language is used to mention the network of CAL modules (actors). This language describes the connections between FUs implemented in CAL.
- **RVC-BSDL**: A constrained set of BSDL (Bit stream Syntax Description Language) is used for describing the syntax of the bit stream.

A Video Tool Library (VTL) is standardized to offer FUs that can be connected as a network of video codec’s. The reference software of VTL is written in RVC-CAL and functional unit network is described in FNL. The bit stream parser can be automatically synthesized from BSD (Bit stream Syntax Description) written as a RVC-BSDL schema.
3. Tools and Methodology

In the advanced compiler design techniques, compiler design is divided into parts: front-end and back-end. In the first step, i.e. front-end, compiler scans and translates the source code written in source programming language to the intermediate representation. In the back-end module, the compiler do some optimization and translates the intermediate representation into binary executable code. In some cases, the intermediate code is translated to some other computer language to generate executable .exe file.

This report deals with a front-end compiler for CAL language. This front end will translate the CAL program into LLVM-IR (LLVM-Intermediate Representation).

The front-end compiler step is further divided into following steps.

- Lexical analysis
- Syntactical analysis
- Symantec analysis
- Intermediate code generation

3.1. Lexical analysis

The lexical analyzer, also called scanner, of the compiler front-end reads the source code characters and generates the expressive words called lexemes. These lexemes are also called tokens. The token consists of two parts: a type and a value. For example, the token for the string “Halmstad” will be as follows:

Token: Type → string

Value → Halmstad

The lexical analyzer takes a stream of characters and generates the stream of tokens. This stream of tokens will be input for syntactic analyzer.

Lexical generators are used to generate a lexical analyzer or scanner for any computer language. The lexical tokens are specified using the formal language of regular expressions. By the use of a regular expression, it is possible to represent a regular language that is set of a finite sequence of symbols. For example to recognize the identifiers and integer literals, following regular expressions can be used.

Identifier = [a-zA-Z][a-zA-Z0-9]*

Integer literal = [0-9][0-9]*
These two regular expressions mean that the identifier will be the sequence starting with letters a-z or A-Z and followed by a combination of alphabets and numeric digits. The integer value will be a combination of numeric digits from 0 to 9.

In some cases on set of input can be matched to more than one regular expression. The situation of ambiguity and clash arises. To avoid the ambiguity and clash, the scanners use two rules. The rules are as follows.

**Longest match:**

The next token is the longest substring of the input that can match a definition of a regular expression.

**Rule priority**

The order of writing regular expression sets the priority of each regular expression. The regular expression written first or higher up the order has the high priority as compared to the following expressions. If two rules match the same sequence of input, then the tokens will be generated by the rule written first or higher up the order.

### 3.1.1. JFLEX: lexical analyzer generator

JFLEX is a lexical analyzer generator written in java. It allows a fast scanner generation and design of fast scanner generators. A JFLEX specification file (.flex) generates a java file which implements a scanner class.

JFLEX specification file has three portions for user code, options and declarations and lexical rules. Each portion is separated by %%% symbol.

**User code**

This portion consists of import statements and packages declarations. The code written in this portion is copied as it is to the beginning of scanner class.

**Options**

This portion includes the directives and the java code. The java code in this portion is copied to scanner class as is. The directives determine the signature and construct of the scanner class. Each directive starts with % sign. The directives related to the signature of scanner class are written below.

```
% public
% final
% class 'CAL_in_LLVM'
% extends 'CAL'
```
By using these directives the signature of the scanner class will be

```java
public final class CAL_in_LLVM extends CAL
```

**Declarations**

This portion contains the specification of lexical states and macros that will be useful in the last part of the flex file. Macros like integers, identifiers and comments can be defined here and then can be used in the lexical rules.

**Lexical rules**

This portion consists of regular expressions followed by corresponding actions. During the running, the scanner consumes its input and determines the matched regular expression for the longest portion of the input. If more than one rule matches the input, then the scanner chooses the one written first in the section. After that it executes the corresponding action. If the input does not match any regular expression, the scanner terminates and generates an error.

JFLEX generates only one file containing the scanner class in accordance to the specification file. This class contains the deterministic finite automaton tables, an input buffer, the lexical states of the specification, a constructor and the scanning method with user defined scanning rules and actions. The input buffer of the lexer is connected with an input stream over java.io.reader object. When it is called, it will consume input until either an error occurs or EoF is reached.

**3.2. Syntactic analysis**

The syntactic analysis phase is the process of analyzing a stream of tokens to determine their grammatical structure in accordance to a given formal grammar. The task of this step is executed by two key components: parser and abstract syntax tree (AST) builder.

Parser checks for the correct syntax and builds a parse tree. Abstract syntax tree (AST) builder builds a simplified structure called abstract syntax tree from the parse tree.

The leaves of the parser tree are the tokens generated by the scanner. The important thing in this tree is how these nodes are connected to each other and what structure they form.

The parser obtains a series of tokens from the scanner and verifies if the series can be generated by the grammar of the source language. There are two types of parsers, which use top-down (LL parsers) or a bottom-up (LR parsers) approach. In both cases, the string of tokens in the input is scanned from left to right one symbol
at a time. The syntax of programming language, like expressions and statements, can be expressed by a context-free grammar.

Context-free grammar consists of four components. $G = (N, T, P, S)$.

**N:** It is the finite set of non-terminal symbols. A non-terminal symbol is a syntactic variable that represents a phrase or clause.

**T:** It is the finite set of terminal symbols. It specifies the language by virtue of grammar.

**P:** It is the set of productions. The ways of combination of terminals and non-terminals are specified by the productions. Each production consists of two parts.

- Head or left side of production. It consists of non-terminals.
- Body or right side of production. It consists of zero or more terminals and non-terminals.

**S:** it is the start symbol that generates the grammar.

There are three operations to combine the terminal and non-terminal symbols i.e. alternation ($|$), concatenation and repetition or kleene closure ($*$).

### 3.2.1. JACC: a parser generator

JACC is a parser generator for java, which is closely modelled on Johnson’s classic YACC parser generator for C. JACC is the abbreviation of ‘just another compiler compiler for java.

Some of the features that make JACC different from other parser generator are:-

- Close syntactic compatibility with Johnson’s classic YACC parser generator.
- JACC generates bottom-up/shift reduce parsers for LALR grammars with disambiguating rules.
- It is a pure java implementation and runs on many java platforms.
- Primitive support for distributing grammar descriptions across multiple files to support modularity.

**Command line syntax**

JACC is used as a command line utility using ordinary text files for input and output. The input to JACC, a context-free grammar with semantic actions, should be put in a file with .jacc extension. E.g. let suppose HA.jacc is a file containing the context free grammar. The parser generator is executed using a simple command

```
Jacc HA.jacc
```
JACC will generate two output files by default, one HAParser.java with the implementation of java class HAParser and the second file HAtokens.java that contains Interface called HAtokens. HAtokens.java specifies integer values for every token in the input grammar. JACC will also generate warning messages for any grammatical conflict in the input.

The jacc command accepts many command line options that can be used to modify the basic behaviour. These options are discussed below.

-p: Do not attempt to write HAParser.java. This option is used to check that the input is well formatted and there are no conflicts in the grammar without generating parser.

-t: Do not attempt to write the HAtokens.java. This option is mostly used in combination with -p.

-v: Write a plain text description of the generated machine in the file HA.output. The output file provides a description of each state and concludes with brief statistics for input grammar and generated machine.

-h: Generate a description of generated machine in HTML. The basic structure of output is same as in –v option.

-f: Includes results of first set, follow set and nullable calculations for each non-terminal in the input grammar as part of the output produced by the options –v or –h.

-a: Uses the LALR strategy to resolve conflicts. This is the default behaviour for jacc.

-s: Uses the SLR strategy to resolve conflicts.

-o: Uses the LR strategy to resolve conflicts.

-r file: JACC reads a sequence of grammar symbols from the given file and generates a trace to show the sequence of shift and reduce steps that the generated parser would follow on that input.

-n: Includes state members in the traces that are produced by –r option.

Input file syntax

The basic structure of the jacc input file is as follows.

---- Directives section -----

%%%%

---- Rules section -----

%%%%

---- Additional code -----
The second %% symbol and additional code section can be omitted if not required. Comments can be included in any section of jacc file. Comments in the jacc are same as in C or java.

**Directives section**

This section is the opening section of a jacc file. This section contains a sequence of directives that can be used to customize certain aspects of generated java file, to specify the interface between lexical analysis and parsing, and to describe properties of the terminal and non-terminal symbols in the input grammar.

**Customizing the generated parser**

Following directives are used to customize the generated parser.

- `% package`: used to specify the package.
- `%{ %} (code block): The java code written in this block is copied at the start of the parser class.
- `% interface`: Used to change the name of the interface.
- `% extends`: Used to specify super class for parser.
- `% implements`: Used to specify the interfaces implemented by the parser.

**Customizing the lexer/parser interface**

Following directives are used to customize the lexer/parser interface.

- `% next`: Used to specify the code sequence to invoke the lexer.
- `% get`: Used to specify the code sequence to obtain integer code for current token.
- `% semantics`: Used to specify the type of semantics values.
- `% start`: Used to specify the start symbol for grammar.
- `% token`: Used to define terminal symbols.
- `% type`: Used to define non-terminal symbols.
- `% left, %right, % non assoc`: Used to define terminal symbols and their precedence.

**The rules section**

The rule section of the jacc file follows immediately after the first %% marker. It specifies the context free grammar for the language that generated parser is intended to recognise. It also associates each production with a fragment of code called semantic action. The parser will execute this semantic action each time the production is reduced. Semantic is able to access the semantic values corresponding to each of the symbols on the right hand side of the rule. They are typically used to construct a portion of parse tree or to perform other computations.
3. Tools and Methodology

The additional code section

The final section of the jacc file follows the second \%% marker and provides an area to write extra code. This code is as it is copied to body of the generated parser class. JACC does not attempt to check that this code is valid. Syntax errors in this portion of the code will be detected when the java file will be compiled.

3.3. Semantic analysis

The third step of the compiler front-end is the semantic analysis. In this step the compiler adds semantic information to the abstract syntax tree. The task in this step can be further divided into two sub tasks:

Name Analysis

Each object in programming language code has a unique ID or name associated with it. This name of the object is used to refer the object in the code wherever it is required to use to complete goal task. Associating use and declaration are called name binding if a declaration of this object is visible in the whole program then it is said to be global but if it is only visible in some parts of the code then it is said to be local. The scope of local objects is restricted to the area where they are defined.

Symbol tables are used to do name bindings. All the information about the object is stored in this symbol table. In the code generation step any object can be traced by using symbol tables. The defined functionality and the intended functionality of any object can be verified at every position where that object is used. For examples if an object was defined as variable but at some point the user mistakenly tried to use as a function, the compiler can generate an error by looking in the symbol table that the intended use of this object is not defined.

Type Analysis

As the name suggests, the step checks the type of expressions and statements. It computes the type of each expression and checks whether these types are legal in the programming language. For example, consider the code below.

\[
\text{Int } x, y; \\
\text{Boolean } z; \\
X = z + y; \\
\]

In the code above two integer variables and one Boolean variable is declared. In the third step user is trying to add integer and Boolean value. That is not legal in computer languages. So the type analysis part will generate an error that all the variables in this statement should be in the same type.
3.4. Intermediate code generation

The final goal of this compiler is to translate the user’s written high level language code into a low level machine code. So that it can be executed on the hardware. In the front-end compiler it is better to translate this high level language code to intermediate representation rather than low level machine code. If the compiler translates directly into low level machine code and the user has to change the hardware platform, then they will have to write a new compiler. By using intermediate representation, the user only will have to write back-end compiler. The front-end compiler will be the same.

The intermediate representation language should have some distinct features. These features are listed below.

- It should be easy to translate from a high-level language to the intermediate language.
- It should be easy to translate from the intermediate language to machine code.
- The intermediate format should be suitable for optimizations.

The first two of these properties can be somewhat hard to reconcile, so a language that is intended as target for translation from a high-level language should be as close as possible to these three rules.

The compiler discussed in this report used LLVM as the intermediate representation language. As the name of the thesis suggests, it was a requirement to use LLVM intermediate representation.
4. Implementation

This report discusses a front-end compiler for CAL language to LLVM translation. The compiler has two parts. One sub-compiler to translate the CAL actors into LLVM and the other, sub-compiler to compile the network defined in network description file.

4.1 HHCAL

The first sub section of the compiler is HHCAL. This section translates the CAL actors into LLVM-IR (LLVM intermediate representation). The figure 5 shows the overview of HHCAL.

![Figure 5: Overview of HHCAL](image)

HHCAL block in figure 5 implements lexer, parser and code generator. This section is the example of what typical compilers do.

4.1.1. Lexer

As described in chapter 3, laxer or scanner is the module that scans the input source file, in this case CAL actor code, and generates the stream of tokens. CAL, as most of the programming languages, has following types of lexical tokens: keywords, identifier, operators, delimiters, comments and numerical literals. Each type of lexical token is described using regular expressions in JFLEX. The regular expressions for identifier and numerical literals are shown below.

Identifier = [a-zA-Z][a-zA-Z0-9]*

Numerical literal = 0 | [1-9][0-9]*

The tokens implemented in the compiler discussed in this report are shown in the table below.
<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Token</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>Operator</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>Operator</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>Operator</td>
</tr>
<tr>
<td>4</td>
<td>/</td>
<td>Operator</td>
</tr>
<tr>
<td>5</td>
<td>(</td>
<td>Operator</td>
</tr>
<tr>
<td>6</td>
<td>)</td>
<td>Operator</td>
</tr>
<tr>
<td>7</td>
<td>;</td>
<td>Operator</td>
</tr>
<tr>
<td>8</td>
<td>:</td>
<td>Operator</td>
</tr>
<tr>
<td>9</td>
<td>=</td>
<td>Operator</td>
</tr>
<tr>
<td>10</td>
<td>&gt;</td>
<td>Operator</td>
</tr>
<tr>
<td>11</td>
<td>&lt;</td>
<td>Operator</td>
</tr>
<tr>
<td>12</td>
<td>[</td>
<td>Operator</td>
</tr>
<tr>
<td>13</td>
<td>]</td>
<td>Operator</td>
</tr>
<tr>
<td>14</td>
<td>,</td>
<td>Operator</td>
</tr>
<tr>
<td>15</td>
<td>{</td>
<td>Operator</td>
</tr>
<tr>
<td>16</td>
<td>}</td>
<td>Operator</td>
</tr>
<tr>
<td>17</td>
<td>//</td>
<td>Comments</td>
</tr>
<tr>
<td>18</td>
<td>Print</td>
<td>Keyword</td>
</tr>
<tr>
<td>19</td>
<td>Actor</td>
<td>Keyword</td>
</tr>
<tr>
<td>20</td>
<td>Endactor</td>
<td>Keyword</td>
</tr>
<tr>
<td>21</td>
<td>Action</td>
<td>Keyword</td>
</tr>
<tr>
<td>22</td>
<td>Endaction</td>
<td>Keyword</td>
</tr>
<tr>
<td>23</td>
<td>Int</td>
<td>Keyword</td>
</tr>
<tr>
<td>24</td>
<td>Var</td>
<td>Keyword</td>
</tr>
<tr>
<td>25</td>
<td>Bool</td>
<td>Keyword</td>
</tr>
<tr>
<td>26</td>
<td>True</td>
<td>Keyword</td>
</tr>
<tr>
<td>27</td>
<td>False</td>
<td>Keyword</td>
</tr>
<tr>
<td>28</td>
<td>And</td>
<td>Keyword</td>
</tr>
<tr>
<td>29</td>
<td>Or</td>
<td>Keyword</td>
</tr>
<tr>
<td>30</td>
<td>Not</td>
<td>Keyword</td>
</tr>
<tr>
<td>31</td>
<td>If</td>
<td>Keyword</td>
</tr>
<tr>
<td>32</td>
<td>Then</td>
<td>Keyword</td>
</tr>
<tr>
<td>33</td>
<td>Else</td>
<td>Keyword</td>
</tr>
<tr>
<td>34</td>
<td>Endif</td>
<td>Keyword</td>
</tr>
<tr>
<td>35</td>
<td>Do</td>
<td>Keyword</td>
</tr>
<tr>
<td>36</td>
<td>Guard</td>
<td>Keyword</td>
</tr>
<tr>
<td>37</td>
<td>Foreach</td>
<td>Keyword</td>
</tr>
<tr>
<td>38</td>
<td>Endforeach</td>
<td>Keyword</td>
</tr>
<tr>
<td>39</td>
<td>In</td>
<td>Keyword</td>
</tr>
</tbody>
</table>
4.1.2. Parser

Parser is the section of compiler where it scans the tokens and checks the syntax of the source code. Then it creates an abstract syntax tree from the tokens passed to it by lexer. This report used the JACC tool to implement the parser. The syntax of the language is defined using context-free grammar. This grammar describes the syntax of the source language CAL in this case. The list of productions is divided into five categories.

- **Actor**: it specifies the rule to declare actor, imports, variables and functions.
- **Expressions**: It specifies how to construct expressions according to construct of CAL.
- **Statements**: it specifies how to use the statements in CAL.
- **Actions**: it specifies how to declare and use the actions in CAL programs. It also includes the declaration of action’s input and output.
- **Action control**: it specifies the control of action flow. How and when action will be fired.

Below are some grammatical rules for some productions. These rules will help the user in understanding how to make a CAL program.

**Expressions**: The context free grammar implemented for expressions is:

```
exp       :  '-'  exp
          |  exp  '+'  exp
          |  exp  '-'  exp
          |  INTEGER_LITERAL
          |  ID
          |  TRUE
          |  FALSE
          |  exp  AND  exp
          |  exp  OR  exp
          |  NOT  exp  %prec  UNEG
          | ' ('  exp  ')'
          |  exp  '<'  exp
          |  exp  '>'  exp
          |  exp  '='  exp
          |  exp  '*'  exp
          |  exp  '/'  exp;
```

The grammar above declares the syntax for add, subtract, greater than, less than, assignment, multiplication, division, and, or and not expressions.
Statements: There is no print statement in CAL. But to provide a little bit of debugging technique this compiler implements the print statement. The grammar for the print statement is as follows:

PRINT '(' exp ')' ';'  

The example of how to use the print statement is shown below. The syntax of print is quite similar to system.out.print in java.

Example:    
            print (2);  
            print (a);  

To create a loop in CAL, ‘foreach’ keyword is used. The grammatical syntax for foreach is

| FOREACH id IN '{' explst '}' DO stmls ENDFOR  

The ‘explst’ above represents the expression list. These expressions will decide the iterations of the loop. The stmls represents the statement list. These statements will be executed in each iteration of the loop.

The example of foreach loop is given below.

Example:  
            foreach a in {1,2,3}  
              do  
                  ...  
              endfor  

The example above demonstrates the use of ‘foreach’ loop. In the example above the variable ‘a’ is a control variable. It controls the iterations of the loop. The loop will run for any values of ‘a’ between the range of 1-3 as declared above after ‘in’ keyword.

Actors: The actors are main building block in CAL. All variable and action declarations are made in actors. The grammatical syntax that has been implemented in the thesis is as follows

actor      : ACTOR id '(' ')' acportls '=' '=' '>' acportls  
             ':' varls actls ENDACTOR  { $$ = new Actor($2,$5,$9,$11,$12);}  

Example:  
            actor  
              actor1() int in1,bool in2 ==> int out1,bool out2:  
              var  
                  int var1;  
                  bool var2;  
                  int var3;  
                  ...  
              endactor
In the example above the actor has two inputs. One input is of integer type and the second is of Boolean type.

**Actions:** Actions are computational entities. Every actor consumes some input tokens and then generates output tokens. Action can be represented by context free grammar as follows.

\[
\text{act} : \text{ACTION portls} \ '=>' \ '=>' \ '>'+ \ \text{portls} \ \text{DO} \ \text{stmls} \ \text{ENDACTION} \\
| \text{ACTION portls} \ '=>' \ '=>' \ '>'+ \ \text{portls} \ \text{GUARD} \ \text{exp} \ \text{DO} \ \text{stmls} \ \text{ENDACTION} \\
\]

\[
\text{Actls} : \text{act} \\
| \text{actls act} ;
\]

Actls above represents the action list. Below is the example of action in CAL.

**Example:**

\[
\text{action} \ \text{in1,in2} \implies \text{out1, out2} \ \text{do} \\
\ldots \\
\text{endaction}
\]

\[
\text{action} \ \text{in1} \implies \text{out1} \ \text{guard} \ (\text{in1}>1) \ \text{do} \\
\ldots \\
\text{Endaction}
\]

The first action in this example consumes two inputs and generates two outputs. And the second action is guarded by guard statement. Second actor will be fired only if the input will be greater than 1.

**4.1.3. Code generator**

The last part of the compiler is the code generator. It translates the abstract syntax tree generated by the parser into LLVM-IR (Low Level Virtual Machine-Intermediate Representation). While translation it also type checks the statements and expressions. The code for code generator is attached in the appendix.

The code generator gets abstract syntax tree as input from parser. It then recognizes the statement or expression and then run the corresponding routine to generate LLVM intermediate representation. It maintains symbol table to keep track of the variables and actions in the program. It also has interfaces to store the signature of the actions. In the interface it stores the number of input and output parameters of the actions. It also keeps the address of the variables and actions to check if certain variable was declared or not whenever it is called in the program.
4.1.4. Limitations in HHCal

There are some limitations in HHCal. These limitations are there because of limited implementation of CAL language. Some features of CAL are implemented in HHCal and these limitations tell about the features that are not implemented. Some limitations in HHCal are described below.

4.1.4.1. Limitations in connection

There are certain limitations concerning connections between actors. The input and output of the same actor cannot be connected together. If the feedback loop is required then another actor should be used to map output of the actor to its input. Every output of an actor can be connected to only one input of any actor. One output cannot be connected to multiple inputs. If same output is required at several input ports then another actor should be used that can map one output to several outputs.

4.1.4.2. Constant parameters

CAL allows the programmer to pass constant parameters to the actors. But this feature is not supported by HHCal. If one wishes to pass constant value to actor for manipulation then it has to be defined as variable in the actor.

4.1.4.3. Actions with state

CAL provides a form of actor that can maintain their state via state variable. That state variable is considered as global variable in LLVM. To do so in HHCal, programmer has to use the feedback loop to store value of state variable. Every variable in actor is local but global to actions. In order to maintain previous value of state for the next iteration of an actor, it should be feed backed to the actor as an input.

4.1.4.4. Limitations with type

CAL language itself does not have hard and fast rules for types. The variables can be in any type and it is the duty of the programmer to look for typographic mistakes. In HHCal, type checking is considered as a job of the compiler. In HHCal variables can be in two types i.e. integer and Boolean.
4. Implementation

4.1.5. How to use HHCAL

This compiler can be used from the command prompt environment. Let’s suppose, the CAL file to be compiled is named example.cal. The command to compile this file is:

$> HHCal [actor name]

Example:

$> HHCal example.cal

This will generate a LLVM back end file. To generate bit code from LLVM file, llvm-as tool should be used:

$>llvm-as -f example.ll

4.2. HHNetwork

HHNetwork is the second part of the compiler. This module generates LLVM code from a network of actors. It compiles the actors also makes runtime system. The final generated code by this module is a runtime system and codes to describe the network and connections.

4.2.1. Network of actors

Each actor is a separate independent entity in CAL language. To make a big and useful program more than one actor can be joined in a form of network. In the network configuration the actors are joined in such a way that output of some actors is input of others. CAL is a better alternative for parallel processing. The network of actors is defined in xdf file. XDF is the abbreviation of extensible data format. It is an xml standard developed by NASA, to be used throughout scientific disciplines. XDF is used to store high dimensional data and information related to it in compact XML format.

Consider the network of actors shown in figure 6.
Figure 6: Network of actors (simple)

The figure 6 shows the network of two actors. The first actor a1 has one input and two outputs. Its implementation is in act1.cal file. The second actor a2 has two inputs and one output. Its implementation is in act2.cal file. This network is sort of feedback network. These two actors are manipulating outputs of each other. The inputs of actor a2 are connected to the outputs of actor a1 and the input of actor a1 is connected to output of actor a2.

The connection between these actors can be described in XDF file as shown below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="netexm">
    <Instance id="a1">
        <Class name="act1"/>
    </Instance>
    <Instance id="a2">
        <Class name="act2"/>
    </Instance>
    <Connection dst="a2" dst-port="in1" src="a1" src-port="out2"/>
    <Connection dst="a2" dst-port="in2" src="a1" src-port="out1"/>
    <Connection dst="a1" dst-port="fi" src="a2" src-port="fo"/>
</XDF>
```

The second line of the xdf code sets the name of the network, netexm in this case. After that two instances are made for both actors and linked with the respective CAL files. In the last three lines the connections are defined.

### 4.2.2. Runtime system

Run time system is the main core of HHNetwork. The task of the runtime system is to make connections between the actors according to connection definition in XDF file. It also has to generate table or queue to store the data transferred from one actor to other via output and input ports. It also has to run each actor in the network. The main task of the runtime system is to schedule the execution of actors. This report deals with the compiler for CAL language for single core system. Scheduling different actors on a single core is the most important task for the runtime system.

Below are some externals functions that any actor can call from the runtime system.
To write data on output ports

To write data on output ports ‘writenode’ function is used. The syntax of the function is:

```
@writenode.<data type>(value, actorid, port#)
```

The data type represents the type of data to be written. Value is the value to be written. actorid is the unique id assigned to each actor and port# represents the number of output port.

Integer and Boolean data can be written like as shown below using this function.

For i32 type: i32 @writenode.i32 (i32, i32, i32)
For i1 type: i32 @writenode.i1 (i1, i32, i32)

To read data from input ports

To read data from input ports ‘readnode’ function is used. The syntax of the function is:

```
@readnode.<datatype>(actorid, port#, 1/0 = deque/just read)
```

The data type represents the type of data to be written. actorid is the unique id assigned to each actor and port# represents the number of output port. The third argument is for just read or de-queuing from input buffer.

Integer and Boolean data can be written like as shown below using this function.

For i32 type: i32 @readnode.i32 (i32, i32,i1)
For i1 type: i32 @readnode.i1 (i32, i32,i1)

To check if the node has fired

Every actor whose input is connected to output of other actor has to check if the predecessor actor has generated data or not. For this the function ‘chknodefired’ is used. The syntax of this function is:

```
@chknodefired. <data type> (actor id, port#)
```

Here actor id represents the actor generating output and port number is the address of interesting output port. The data type represents the data type.

Nodes with data type’s i32 and i1 can be checked by the instructions written below.

For i32 type: i32 @chknodefired.i32 (i32, i32)
For i1 type: i32 @chknodefired.i1 (i32, i32)
4.2.2.1. Scheduling of actors

The main job of the runtime system is to schedule the execution of actors in the network because this compiler is aimed for single core platform. Let’s assume there are n actors in a network and one processing core. The crucial task is to allocate the computational resource properly to every actor in the network so that they can perform their specific job.

The compiler discussed in this report has the arbitrary scheduling strategy. Every actor gets turn to execute one by one irrespective of fact if the actions in the actor are ready to fire or not. Each actor will get its turn and will check how many actions are ready to fire. Only one of the ready actions will be fired and the actors will terminate and pass turn to the next actor in the queue. For firing of remaining ready actions, actor will wait for its next turn.

Consider the network of actors shown in figure 7. Let’s assume there are two ready actions in actor ‘A’, one in actor ‘B’ and no ready action in actor ‘C’. According to the scheduling policy actor A will get the turn first. It will fire its one ready action and pass the turn to actor B. To fire the second ready action actor A will have to wait for its next turn. Actor B will fire the ready action and pass the turn to actor C. As there is no ready action in actor C, it will terminate immediately and will pass the turn to actor A.

4.2.2.2. Communication strategies

One fundamental task of the runtime system is to make communication links for intercommunication between actors. The way this communication is implemented in this report is by use of shared buffers between the nodes. For every communication link there is one shared buffer between input and output ports. The figure below illustrates the concept.
4. Implementation

In the example illustrated in the figure 8, act1 will write to the buffer_1 whenever it will generate output. It will manipulate the input and then will write the output to shared buffer. The actor act2 instead of waiting for act1 to complete its execution will just read new value of input from shared buffer. Act2 will first check the buffer if it is empty or not. If it has some data act2 will read from it.

**Alternatives for buffer implementation**

There are three options for buffer implementation.

**Bounded single entry buffer**

In this option the size of the buffer is one entry. It means that an actor can write only one output to it. If the actor reading the buffer did not read before second write, the previous data will be lost. Also this option can stall the network. E.g. if the receiver is waiting for other actor to write but the writing actor is not ready to write, then whole network will be blocked.

**Bounded multiple entry buffer**

In this option the size of the buffer varies with the implementation. The size of the buffer should be two or more. It can avoid the blocking problem up to some extent. If the actor writing to the buffer is writing faster than the one reading it, then the blocking situation will arise. It also needs more space depending upon the size of the buffer.

**Unbounded buffer**

In this option buffer size is unlimited. The size of the buffer will be limited by the memory size. The blocking problem will not occur in this option. But the memory requirement to implement this strategy can be quite high. This strategy can have a
problem for embedded hardware as memory is the main issue in embedded systems.

4.2.2.3. Buffer selection for implementation

The compiler discussed in this report uses the unbounded buffers. By using this strategy blocking problem is resolved. As more memory space is required for implementation of this strategy. But this compiler is aimed for personal desktop computers or laptops. Memory space is not an issue.

4.2.3. How to use HHNetwork

This compiler can be used from the command prompt environment. Let suppose, the CAL network file to be compiled is named example.xdf. The command to compile this file is:

```
$> HHNetwork [network file]
```

Example:

```
$> HHNetwork example.xdf
```

Or

```
$> HHNetwork example.xdf –a
```

In case of network of actors, there is no need to compile all actors in the network separately. HHNetwork will automatically call HHCAL and will compile everything.

4.3. Evaluation

To evaluate the compiler HHCAL and HHNetwork an 8x8 matrix multiplier example is developed in CAL. This example generates the two matrices row by row and then multiplies them together to calculate the result. The example consists of the following actors.

- Matgen
- Feedback
- Multiplexer
- Multiply
- Feedback 2
4. Implementation

- Rowcol

4.3.1. Matgen (Matrix generator)

This actor generates the rows of the matrices to be multiplied. It uses the feedback mechanism to generate row values. First element is 1 always and then it generates the multiple of the previous values.

```java
actor matgen() int in1 ==> int out1, int out2:

  action in1 ==> out1, out2
  do
    if (in1>98) then
      out1:=1;
      out2:=1;
    else
      out1:= in1 * 2;
      out2:= in1 * 2;
    endif
  endaction

  action ==> out1, out2
  do
    out1:=1;
    out2:=1;
  endaction

endactor
```

In the actor in figure 9, out2 is an output used for feedback loop. Second action in the actor will fire only for the first time when there will be no input. It will write 1 to output. When there will be input from feedback mechanism, first action will be fired. This action will multiply the input by 2 and then write the result to both outputs.

4.3.2. Feedback

This actor is used to provide a feedback loop for matgen actor. Due to certain limitations discussed in chapter 4 section 4.1.4 matgen requires another actor for feedback loop.
As shown in figure 10, feedback actor has one input and one output. The only action in this actor writes the input to output whenever input is available.

### 4.3.3. Multiplexer

Rows generated by the matgen actor are to be used by eight different modules to calculate resultant matrix. As discussed in section 4.1.4 one output cannot be connected to multiple inputs. So, this actor comes into action. As shown in figure 11, it makes eight copies of one input. These eight outputs then can be used by different actors.

```
actor feedback () int in1 ==> int out1;
action in1 ==> out1
do
    out1 := in1;
endaction
endactor
```

**Figure 10: feedback.cal**

```
actor multiplexer () int in1 ==> int out1,int out2,int out3,int out4,int out5,int out6,int out7,int out8;
action in1 ==> out1,out2,out3,out4,out5,out6,out7,out8
do
    out1 := in1;
    out2 := in1;
    out3 := in1;
    out4 := in1;
    out5 := in1;
    out6 := in1;
    out7 := in1;
    out8 := in1;
endaction
endactor
```

**Figure 11: multiplexer.cal**

Multiplexer actor does the job remarkably similar to feedback actor. Its only action waits for the input and then writes input to all eight outputs.

### 4.3.4. Multiply

Multiply actor gets a row of each matrix from multiplexer actor. As shown in figure 12, it multiplies the two rows and generates the result. It multiplies the rows element by element and calculate cumulative sum. The result generated by this module is one element of the resultant matrix. In1 and in2 are row elements of the matrices to be multiplied.
This actor has two state variables sum and cnt. Values of these two variables are maintained by a feedback loop. In3 and in4 are used as feedback inputs to maintain the state.

4.3.5. Feedback2

This actor is used to provide a feedback loop for multiply actor. Due to certain limitations discussed in chapter 4 section 4.1.4 multiply actor requires another actor for feedback loop to maintain state.
As shown in figure 13, feedback actor has two inputs and two outputs. The only action in this actor writes the inputs to outputs whenever inputs are available.

4.3.6. Rowcol

Rowcol actor serves as collector. It collects results from different multiply actors and combine them to form rows of the resultant matrix. As shown in the figure 14, it has 8 inputs and 1 output. This actor gets input from eight different multiply actors and combines them to form a row. It also prints the values of row elements to console.

```cal
actor rowcol () int in1, in2, in3, in4, in5, in6, in7, in8 ==> int out1 :
1  int a; int b; int c;
2  int d; int e; int f;
3  int g; int h;
4  
5  action in1, in2, in3, in4, in5, in6, in7, in8 ==> out1
6  do
7     a := in1;
8     b := in2;
9     c := in3;
10    d := in4;
11    e := in5;
12    f := in6;
13    g := in7;
14    h := in8;
15    
16    out1 := a;
17    print(a);
18    out1 := b;
19    print(b);
20    out1 := c;
21    print(c);
22    out1 := d;
23    print(d);
24    out1 := e;
25    print(e);
26    out1 := f;
27    print(f);
28    out1 := g;
29    print(g);
30    out1 := h;
31    print(h);
32    
33   endaction
34 endactor
```

Figure 14: rowcal.cal

Rowcol actor waits till it has all eight inputs ready and prints and write them one by one to console and output respectively.
4.3.7. Interconnection and working

The first actor in the example is matgen actor. It generates the rows of the matrices. It is then assisted by feedback actor. Feedback actor copies one output of matgen actor to its input. Matgen actor is followed by multiplexer actor. Each row of the matrix is required eight times by multiply actor. So, this multiplexer actor makes eight copies of every row of both matrices. Then comes the multiply actor. It gets one row of both matrices element by element. It multiplies the entries and calculate cumulative sum of multiplication. It then passes the result to Rowcol actor. Rowcol actor gets the result from eight different actors and combines them together to form a row of resultant matrix.
5. Conclusion and Future Work

5.1. Conclusion

This report is about a front-end compiler for CAL (Caltrop Actor Language). It translates CAL programs into LLVM-IR (Low Level Virtual Machine-Intermediate Representation). This compiler is aimed to work on desktop computers or laptops i.e. intended for single-core machines. The compiler has two main parts: HHCAL and HHNetwork.

HHCAL translates CAL actors into LLVM intermediate representation. It includes a lexer, a parser and a code generator. The lexer scans the source file and generates a sequence of tokens. The parser implements the context free grammar of the source language, CAL in this case. It generates an abstract syntax tree from the tokens generated by the lexer. Finally, the code generator produces LLVM-IR from the abstract syntax tree created by the parser.

HHNetwork is the second part of the compiler. It is the main component as it translates the network of actors written in XDF network description file. It implements a run time system for CAL programs. The runtime system generates the communication links for the intercommunication between actors. It implements unbounded buffers for communication between actors. It also schedules the execution of actors. It implements the arbitrary scheduling technique for allocation of processing element to actors. I.e. every actor will execute one by one in a row and every action in actor runs to completion.

A CAL program, i.e. a network of actors, can be compiled using our compiler. The output of the compiler includes an LLVM-intermediate representation of actors along with code of scheduling and communication techniques. The generated intermediate representation can then be mapped to any platform using back-end compiler.

5.2. Future Work

This compiler produces LLVM-IR for CAL programs. It is aimed for personal desktop computers or laptops. It is made to run on a single core platform. It can deal with most of actor networks but still a lot can be done to improve the system. Some suggestions for future work are as follows.
• Add remaining keywords and lexical tokens to provide more functionality
• Implement more types and type checking
• Make it run for multi-core platforms
• Make it run for embedded hardware chips
References


