



Embedded and Intelligent System

---

# MASTER THESIS

CAL code generator for Epiphany architecture

Mingkun Yang

August 2013





# **CAL code generator for Epiphany architecture**

Master's thesis in Embedded and Intelligent System

2013

Author: Mingkun Yang

Supervisor: Zain Ul-Abdin

Examiner: Tony Larsson

---

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

© Copyright Mingkun Yang, 2013. All rights reserved.  
Master thesis report 1401  
School of Information Science, Computer and Electrical Engineering  
Halmstad University

## **Abstract**

Various approaches have been proposed to address high performance parallel computing, among which, data flow paradigm is one particular promising candidate. This thesis work builds a code generator for CAL, one of languages supporting data flow paradigm, targeting many-core architecture, namely Epiphany, which is one parallel architecture, and promises to scale the number of processors horizontally. This project begins with the analysis on some existing code generators of CAL targeting multicore architecture, adapts a difference approach in code generation, with the rational that generated code should be quite readable as well in mind, and uses Object Oriented Programming style to remove the code duplication. The default build method bundled with Epiphany SDK is replaced by one customized solution to achieve clean user interface and rapid feedback so that developers react quickly when exceptions happen. The code generator uses actor machine, which is a machine model for data flow actors developed by Lund University, for scheduling actions within one actor. CAL supports actor communication by passing message, and the generated code uses message passing interface like function calls, and they have to be built on top of the shared memory model architecture in order to have the final functional application on Epiphany. Message passing interface is the main component of the runtime system for Epiphany, and three implementations of them have been proposed. The performance of different implementations of message passing interface is assessed using two dimension inverse discrete cosine transform (2D-IDCT).



## **Acknowledgements**

First of all I want to thank my supervisor Zain Ul-Abdin for his suggestions, advices and help in the whole process of doing this project.

Secondly, I want to thank all my friends, who have provide inspiration, generous help and elaborate explanation on this project whenever I am in doubt, especially Essayas, whose work makes this project possible, and Adam, who points out the culprit causing some mysterious behavior.

Thirdly, I want to thank Veronica for introducing me to Computer Science, laying solid foundation for future studying; I feel so lucky to have such one good teacher on entering one new field. In addition, I appreciate the intensive development experience I gained from Nicholas, and feels it's one of my milestones in CS studying. Thank Bertil for his wonderful lectures, wisdom and guide on my career management.

Fourthly, I want to thank Adapteva Inc for producing this wonderful platform so that we could conduct this project, and evaluate our ideas.

Lastly, I want to thank my families who have been encouraging me during the whole process. Only because of their support, could I concentrate on this project until the end.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement . . . . .	1
1.2	Objectives and Requirements . . . . .	2
1.3	Approach . . . . .	2
1.4	Limitation . . . . .	2
1.5	Outline . . . . .	3
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	CAL Actor Language . . . . .	5
2.1.1	Actor Constructs . . . . .	5
2.1.2	Network . . . . .	6
2.2	Existing Implementation . . . . .	6
2.2.1	D2c . . . . .	7
2.2.2	Orcc back end . . . . .	7
2.2.3	Summary . . . . .	7
2.3	Actor Machine . . . . .	7
2.4	Epiphany Architecture . . . . .	8
2.5	Epiphany Development Environment . . . . .	9
<b>3</b>	<b>Method</b>	<b>11</b>
3.1	Overview analysis . . . . .	11
3.2	Source-to-source compiler . . . . .	12
3.3	Build process . . . . .	13
3.3.1	Rethink the build process . . . . .	13
3.3.2	Another build process . . . . .	14
3.4	Communication API . . . . .	15
3.4.1	Incremental Refinement . . . . .	16
3.4.2	Concurrent programming . . . . .	19
3.4.3	Synchronous and Asynchronous Functions . . . . .	19
3.5	Synchronization . . . . .	25
3.6	Board and Host Interface . . . . .	26

3.7	Uniform actor interface . . . . .	27
<b>4</b>	<b>Result</b>	<b>31</b>
4.1	IDCT . . . . .	31
4.2	Mapping actors to cores . . . . .	31
4.3	Configuration . . . . .	33
4.4	Summary . . . . .	34
<b>5</b>	<b>Conclusion and Future Work</b>	<b>37</b>
5.1	Conclusion . . . . .	37
5.2	Future Work . . . . .	37
5.2.1	Multiple instances in one core . . . . .	37
5.2.2	Compose instances to form one heavy instance . . . . .	39
5.2.3	Unit testing . . . . .	39
5.2.4	Merging multiple connections into one . . . . .	39
5.2.5	Out of Order Broadcasting . . . . .	40
<b>A</b>	<b>Tweaks to Orcc and d2c</b>	<b>41</b>
A.1	Orcc . . . . .	41
A.2	d2c . . . . .	42

# List of Figures

3.1	CAL compiling process. . . . .	11
3.2	Dependency graph of project organization . . . . .	14
3.3	Running make on host, and result is shown on STDOUT of host. . . . .	15
3.4	FIFO structure is placed in destination core. . . . .	18
3.5	FIFO structure is placed in both source and destination core. . . . .	18
3.6	Two FIFO structure is placed in both source and destination core. . . . .	18
3.7	Lock free synchronization on source code. . . . .	20
3.8	Lock free synchronization on destination code. . . . .	20
3.9	The consecutive state of one FIFO, when there's only one empty slot. . . . .	21
3.10	do_flush is called when the FIFO is full. At the moment, the DMA is either finished or hasn't started at all. The caller is blocked until the DMA is finished successfully. . . . .	21
3.11	The state of the FIFOs, where the asynchronous function would be called. . . . .	24
3.12	do_distribute is called when all the FIFO are full. This function will pick up from where try_flush is left and continue with the distribution process. . . . .	24
3.13	Synchronization of four cores and host, which could be extended to many cores horizontally. . . . .	25
4.1	IDCT 2d actor diagram . . . . .	32
4.2	Mapping IDCT to cores on the chip . . . . .	32
4.3	Three plain implementations with different buffer size. . . . .	35
4.4	Applying 'polling-on-local' optimization three implementations with different buffer size. . . . .	36
A.1	Orcc Actor translation for two instances of one actor. . . . .	42
A.2	Orcc actions scheduler . . . . .	43
A.3	Access level error in build process. . . . .	44
A.4	D2c actions scheduler in CAL . . . . .	44
A.5	D2c actions scheduler in C . . . . .	45
A.6	D2c Actor translation . . . . .	46



# List of Tables

4.1	Clock cycles for the whole application using sequence mapping. . . . .	33
4.2	Clock cycles for the whole application using snake mapping. . . . .	34



# Chapter 1

## Introduction

This chapter will start with introducing the problem this project tries to address, and the approach undertaken. After that, objectives of this project and its limitations are presented.

### 1.1 Problem Statement

Generally, there are two groups of approaches to speed up perceived execution. The first group focuses on building more complex and powerful processor by increasing clock frequency, enlarging cache size or adding instruction level parallelism, which is the one that has been used throughout the past few decades, whose consequence is reassured by Moore's law. The other is to spend more effort on how to increase the number of cores in one chip. Because of some practical reasons, e.g. the current hardware has gradually reached the physical limitation that clock frequency could not be increased without burning the chip, the second approach begins to draw more and more attention.

Unfortunately, the traditional programming languages used in sequential model could not be applied to the multicore system. As a result, some new languages based on parallel and concurrent programming paradigms have to be considered, and data flow modeling seems one good candidate, for it's proposed as one way to address parallel computing.

Equipped with data flow programming concepts, developers could express programs using directed graph with edges representing channels data flows through. In this way, the implicit parallelism becomes explicit, and could take advantage of multiple cores. Even the performance of parallel architecture is not the primary factor, the extra readability is beneficial for maintainability reason. NoFlo[1] is one example of using data flow paradigm mainly because of modularity and readability.

## 1.2 Objectives and Requirements

This project tries to demonstrate how one data flow language could be implemented on parallel architecture, by building one compiler for this language. The back end of this compiler is specific to the parallel architecture used in this project, but the concept applicable for others.

This section contains the deliverables (requirements) of this project.

- The solution must support the tool chain, including code generation and building resultant C project, targeting Epiphany architecture.
- The project should do evaluation on Epiphany architecture using existing data flow programs.

## 1.3 Approach

We will use CAL as our data flow language, and Epiphany as our targeting parallel architecture. Some existing implementations for CAL targeting multicore architecture are studied so that it becomes clear what we could do to improve them. Firstly, inspired by Object Oriented Programming paradigm, we would like to generate code in class-vs-instance manner, so that multiple instances of the same actor could share the same code, which gets rid of code duplication in generated code. To ensure the separation of generated code from library code, we draw one clear boundary between hand-crafted code and auto generated code, which results into modular organization of the code base. Secondly, the architecture we are targeting doesn't require operating system, so, hopefully, it could provide us some good performance result because of the absence of OS overhead.

## 1.4 Limitation

There are three categories of limitation in this project:

- CAL language  
Only part of complete CAL language is covered. Since we reuse the existing sequential C code generator, this project covers the same subset of CAL as the sequential C code generator does.
- Epiphany evaluation board  
There are sixteen cores in this board, which might be insufficient to tackle large programs.



- No sophisticated optimization

This project is the first attempt to provide the back end for CAL targeting Epiphany architecture, so we try to have one mere functional system. Therefore, the code generator just does plain translation without any sophisticated code transformation.

## 1.5 Outline

In Background chapter, related work is discussed and how this project is different from them. In addition, some innovative technology are briefed.

In the Method chapter, the main content of this project is discussed in detail, such as the build process, communication API, etc.

In the Result chapter, two-dimensional IDCT is used to assess the performance of the generated code, and difference versions of communication API.

In the Conclusion and Future work chapter, conclusion of this project is drawn. Since this project is just one preliminary task of addressing Epiphany architecture, future work would be done, some of which are discussed.



# Chapter 2

## Background

### 2.1 CAL Actor Language

CAL[2] is one domain specific language that uses data flow programming paradigm with actors as building blocks, and it is the language this project will build code generator for. We will talk about two categories of files constituting complete CAL programs, actor files (.cal) and network files (.nl).

#### 2.1.1 Actor Constructs

Listing 2.1: One Actor Example

```
actor Example () In ==> Out :  
  a0: action In: [a] ==> Out: [a] end  
  a1: action In: [a] ==> Out: [2*a] end  
  schedule fsm s0:  
    s0 (a0) --> s1;  
    s1 (a1) --> s0;  
  end  
end
```

On the highest level, actors could be viewed as stateful operators that transform input streams of data tokens into output streams. As shown in List 2.1, this actor accepts one stream of tokens, perform some operation on them and emit them. Two major components of actors are actions and action scheduler. Actions play the role of function or subroutine in most imperative programming languages, and enable developers to write code in a modular manner. Actions could consume and (or) produce tokens, which is how actors communicate with outside world. Actions scheduler decides which action to fire given the state of the actor and input ports, and its syntax utilizes the declarative

style to express the action priority.

As shown in List 2.1, this actor has two actions, one passing data through without modifying anything, and the other doubling the data value. The scheduler fires two action in turns, using the internal state to form one state machine so that the decision could be made correctly. We have only touched the surface of CAL language, and audience is encouraged to read CAL Language Report[2] for further information of this language, such as guard, lambda expression, etc.

### 2.1.2 Network

Listing 2.2: Simple network

```
network Top () In ==> Out:
    x = Example();
    y = Example();
    ...
    x.out --> y.in;
    ...
```

Network files are used to instantiate actors and connect them to form one application. Similar to actors, network has input and output ports, so it's possible to treat networks as actors conceptually. Such uniformed interface implies the internal recursive structure so that the ubiquity of nested networks in some CAL applications is not surprising. As shown in List 2.2, one actor is instantiated twice, and their ports are connected.

Pay attention on how instances are created, and audience might notice some slight similarity to object instantiation in OOP. Actor model and OOP are completely different paradigms, and such similarity is only limited on syntax. However, we still could take advantage of this so that duplicate code could be avoided. Detailed discussion will be covered in method chapter.

## 2.2 Existing Implementation

Overall, there are two independent implementation framework of CAL language, namely OpenDF[4] [5], and Open RVC-CAL Compiler (Orcc)[3]. Both of them provide front end to parse CAL programs to their own Intermediate Representation (IR), and one simulator to consume IR. In other words, we could easily simulate our program without any special hardware support. Using the front ends provided by either of them, a few of back ends are developed, and we will focus on the two C back end which have multiple core support.

### 2.2.1 D2c

Built on top of the OpenDF framework, the d2c deliverable is part of ACTORS project[7]. By analyzing the network files, it firstly identifies all the actor instances. Then it generates one C file for each instance with all the code needed embedded, including the complete code for the actor and one action scheduler. As for multiple core support, each actor instance is put into one POSIX thread, and load balancing is achieved by Linux symmetry multiprocessing (SMP). One prototype of runtime system, which is responsible for token transferring, is developed as well.

### 2.2.2 Orcc back end

Using the front end from Orcc, this work[6] builds runtime system to support multiprocessing. Similar to d2c, they generate one C file for each instance with all the actor code embedded, are also using thread for parallel execution. However, they also take into account of the physical limitation of target system, and create as many threads as existing cores. In this way, unnecessary overhead could be avoided. In addition, one abstraction level on top of OS is constructed so that it's possible to run the generated code on Windows, as well.

### 2.2.3 Summary

Both implementations generate C code successfully. However, neither of them paid enough attention on readability and design of the generated code. The naive way to do the translation from CAL to C is to have one C file for each instance, that contains state variables, and all actions of one actor. It's a huge waste to have duplicate actions loaded to memory, if there are multiple instances of one actor. Inspired by how class provides blueprint for objects in some OOP languages, we plan to let all instances of one actor share the same actions defined in this actor.

In addition, as the support for multiple core, all the existing solutions are based on the scheduling from OS, which might not be desirable in some embedded systems, where memory and power is much limited compared to PC. Therefore, this project will target bare-metal systems, where code is running on top of hardware without the OS abstraction level.

## 2.3 Actor Machine

As we discussed before, CAL is one declarative languages, which provides high level description of tasks, instead of instructions closely related to underlying architecture. Similar to other high level languages, for instance, Ruby (without the OO part), CAL

frees developers from the complexity of hardware and ensure they could focus on the problem itself. However, one huge difficulty rises from the inherent concurrency of CAL; deciding which actions to fire for one actor efficiently is not one trivial question.

Actor Machine[8] proposed by Lund University is one attempt to address this question. It unifies the interface so that there are only three instructions for actor machine, namely, ITest, ICall, IWait, which are mapped to guard, action call and wait, respectively, in CAL language. In most cases, the actor machine will have only one instruction to execute, namely single-instruction actor machine[8]. Therefore, there will be no dynamic dispatching, execution runs in sequential manner, and the code generator performs straightforward translation to sequential code.

Given such uniform interface, it's possible to compose multiple actors into one, which has a couple of advantages. Firstly, actors are usually quite light compared with system thread. From the perspective of development, it's convenient to have lightweight actors so that developers could just create one new process (or actor) for any particular tasks during runtime (eg. Erlang). In CAL, we couldn't create new actor instances during runtime, but creating actor instances in static time is easy as well. However, if actor is mapped to system thread in one-to-one relationship manner, there will be huge performance penalty hit because of the context switching overhead. After composing, actors might become heavy enough to be granted with one thread. Secondly, it's possible that the number of execution unit hardware (core) or software (thread) might be limited to be a handful. Even the performance hit is tolerable, it's just impossible to hold so many actors directly.<sup>1</sup> Thirdly, some optimization could only be performed by combining multiple actors, for single actor doesn't have enough information for optimization.

## 2.4 Epiphany Architecture

During the past few decades, the performance of CPU is growing according to Moore's law more or less. Unfortunately, recently, we couldn't keep this trend of scaling vertically any more, because it's just too difficult or expensive to put so many transistors together. Gradually, another direction, scaling horizontally, is attracting more and more people. This example[9] proves that it's more economical to build high performance hardware using the latter method, that is assembling small computing units to achieve higher efficiency.

Taking this idea further, Adapteva Inc[10] has designed one architecture, Epiphany architecture[12], which makes it possible to put thousands of cores on one chip, without going through the trouble of connecting all the single cores together. By getting rid of some sophisticated mechanics, that supports branch prediction, speculative execution, or out of order execution, in most modern chips, this architecture features simple core,

---

<sup>1</sup>In fact, the evaluation board this project uses has only sixteen cores.

power friendliness, and reasonable price. All cores share one global address space, and any core could access any data using absolute address, which is how core-to-core communication is achieved. In addition, each core has fix amount of local address space with much lower latency, which is aliased of the corresponding absolute address. The evaluation board used in this project is provided by Adapteva company.

## 2.5 Epiphany Development Environment

The recommended development work flow is based on one augmented version of Eclipse, provided by Adapteva Inc. Following the Epiphany Quick-Start Guide, everyone could reproduce the project easily. Epiphany SDK Reference[11] provides detailed explanation on what happens behind the scene. The approach used by them is very similar to OpenOCD[16], where one server is set up, then loader (as the client) will request server to load the program to the hardware.

Actually, the two essential commands issued by Eclipse are<sup>2</sup>:

```
# e-server -no-test-memory -xml $EPIPHANY_HOME/bsps/emek3/emek3.xml
# e-loader -run-target EXECUTABLE_FILE
```

The file organization used by Eclipse is to have one C project for each core, and one “driver” project, containing only Makefiles to drive the build process. Since it’s not allowed to have more than one main function inside one project, and each core must have one main function as its entry point, it’s straightforward to have one project for each core. The Makefiles bundled with “driver” project is able to build each project for every core and combine all the binary files into one SREC file, loadable to the board, using the command specified above.

---

<sup>2</sup>By default, e-server will perform memory testing after starting, which is quite distracting and time consuming. Since hardware failure is not that common, it’s recommended to disable it.





# Chapter 3

## Method

In this chapter, various methods and design decisions involved in this project are explained in details. Much effort is spent on explaining the exact steps taken in this project, but it's still possible that something is overlooked. Audience is encourage to look into the online repository<sup>1</sup> on doubts.

### 3.1 Overview analysis

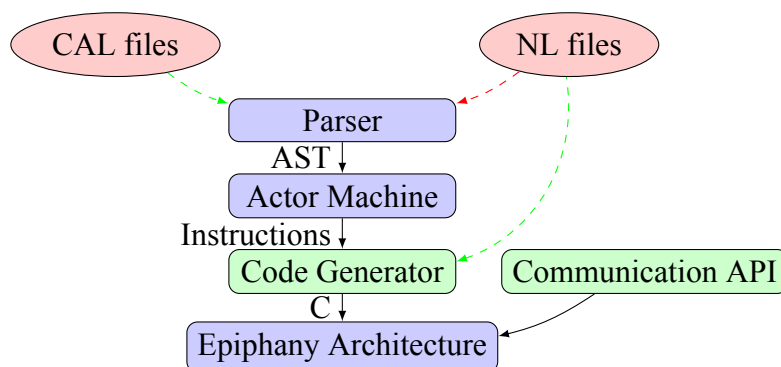


Figure 3.1: CAL compiling process.

The high level structure for the build tool chain, illustrated in Figure 3.1, captures the essence of one large project, from which this project is derived. The front-end (including the parser and some common utilities in most compilers, such as lexer, typer checker, that are not shown in the Figure, for it's not very of importance to our discussion) and the actor machine are contributed by Lund University; this project, on the other hand, focuses on back-end, namely the code generator. The CAL and NL files are the input

<sup>1</sup><https://bitbucket.org/albertnetyrk/epiphany/>

to this compiler. Abstract Syntax Tree (AST) generated by the parser is fed to the actor machine, which would provide uniform machine instructions as we discuss in Chapter Background. From here, the code generator takes the control, and generating C code according to the instructions of AM. Ideally, both CAL and NL files would take this route, but NL has not been integrated into AM yet, as indicated by the red route in figure. Therefore, NL files take another route, and the constructed AST is processed by the code generator directly. Communication API is one utility library, that is independent of input CAL or NL files, and it facilitates core-to-core communication in message passing manner. More about communication API will be covered latter, and now we would firstly reiterate this compiler, code generator, specifically.

## 3.2 Source-to-source compiler

Even though “compiler” is the name we used in above explanation, a better and precise name for this translator[19] would be source-to-source compiler or transpiler[20], since the generated code is high level language or source code which would be fed to one real compiler afterwards. It’s this particular property, source code (human readable code) is generated, that distinguishes it from ordinary compilers, such as gcc and clang. Because of this, extra care needs to be taken to ensure that the source code is readable. The reason we are emphasizing readability of generated code is twofold. Firstly, only rudimentary checks are perform in the tool chain, so it’s very likely some mistakes are found in run-time. Readable code could really makes the debugging process less painful. Secondly, it’s mere subjective preference to have beauty throughout the system, including the part, which might not be intended to be exposed to outside forever. The first argument will become rather weak, if we can be sure that the static checking could catch all the errors before running. However, this is unlikely to be future in the near future, considering even strongly typed languages, like Java, Haskell, couldn’t find the errors all the time. One successful example about readability of generated code is provided to prove having good readability brings a lot benefit. <sup>2</sup> CoffeeScript[22], one transpiler to JavaScript, is becoming more and more popular within web development community, and one of the reasons for this is that the generated code is quite readable and developers have no difficulty in associating them with real source code (CoffeeScript) whiling debugging. We hope we could achieve the same in this project, so this project will start by crafting C code by hand, which provides some guide how the generated C code should look like.

Since we have seen that there will be huge code duplication if instance-based approach is taken, as shown in both implementations discussed in background chapter, we will model actors in OOP way, because actor is to instances as class is to objects. The modeling of actors wraps one actor into one structure, expose only a few public methods

---

<sup>2</sup>Readability is just one reason among many, but I think it’s a important one.

to communicate with outside world, and encapsulation is achieved relying on the fact that static functions is only visible in one translation unit. In this case, this structure, illustrated in List 3.1, acts like one class in some OOP languages, for instance, Java.

Listing 3.1: Actor strucure, resembles class in OOP

```
// actor_double.h
...
typedef struct actor_double_struct {
    // ``fields``
    port_in *in;
    port_out *out;
    // methods interacting with outside world
    void (*run)(struct actor_double_struct *self);
    void (*end)(struct actor_double_struct *self);
    bool (*not_finished)(struct actor_double_struct *self);
} actor_double;
...
```

## 3.3 Build process

### 3.3.1 Rethink the build process

The build process discussed in previous chapter is one simple and quick way to get started, but soon becomes unwieldy once we start working on some more complex projects. In the process, each core is modeled as one independent project in Eclipse, so each one has its own console output, and it's the place to dump information to while building. However, there's only one real console output in Eclipse at one time. Therefore, the active or current console is always jumping from one console to another, confusing the user considerably. In addition, one building error in one project will not halt the whole build process, which is probably not desired, for the final delivery depends on any single project, while to Eclipse they are just independent projects, having no dependencies.

All these kinds of awkwardness originates from the naive modeling of executable on each core. Each core is supposed to run its own, regardless of its neighbors, which provides great abstraction during runtime. However, having the whole program working properly requires all the code running on each core is built successfully, which implies that developers should treat all the code as a whole. Apparently, Eclipse is too strict on project organization (namely only one main function inside one project), so we have to roll out new approach to meet the requirement. After all, it's just one generic IDE, not customized to any particular case.

### 3.3.2 Another build process

Considering all the drawbacks of the default build process, we would propose another build process, which would address all the existing problems. The new build solution would take all cores, namely the board, as one whole, and stops if any of them fails to build. Parallel build is possible, for there's no dependency on each other, but it's better to start from simple build process and go for powerful techniques incrementally. Managing Projects Using GNU Make[17] are quite helpful for implementing the new build system.

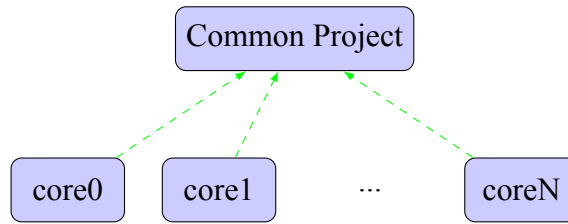


Figure 3.2: Dependency graph of project organization

The original organization provides one common project, that holds common code shared by all the cores, as shown in Figure 3.2. Behind the scene, this common project is compiled to one archive file, and statically linked to all the core related projects. At the first glance, it's not very necessary to use this approach, for each core's supposed to run one single actor, and there's not much to share. However, careful analysis reveals that it's the "actor instance" that is actually running on each core, which are initialized according to the network file. After realizing this, it's comprehensible why both D2c and Orcc are generating code based on actor instances instead of actors. Therefore, we will use the common project technique to avoid generating duplicated code so that it resembles how developers would craft code by hand.

Inspired by the common project technique, we will dump all the actors in one place, then expose it to all the cores using static linking. Due to the particular feature of static linking, the final executable only contains the code for one actor, which is initialized in main function. Using this method, we avoid source code duplication but each core still has one complete copy of the executable.

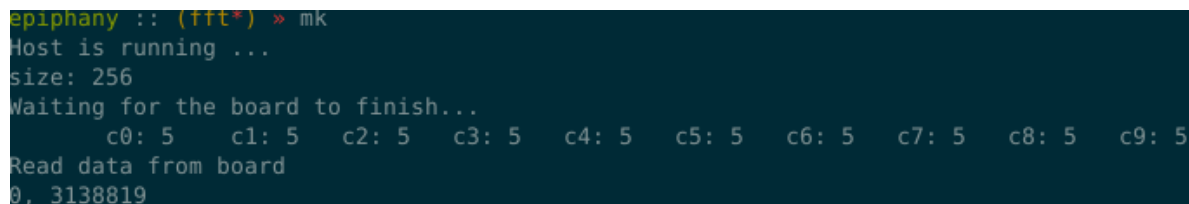
It's convenient to have some phony targets in Makefile to facilitate the process of build-load-run cycle. The program loaded to the board is running immediately, according to the passed argument, namely "-run-target". However, there's no any kind of interface for developers to "see" the status of the program. Therefore, it's idiomatic to define some interface for the board and the host to communicate. In this case, mailbox is used to share data between the board (all the cores) and the host. Once data is retrieved by the host, STDIO is used for data displaying, and developers would be aware of what's going on the board (or inside all the cores).

Because of the close coupling between the board and host program, it's better for the host program to be managed by the same Makefile as well. In addition, the common interface between the board and the host reflects code sharing, which would have to be duplicated if the host program lives in another project. Therefore, this host program reside in one directory inside this project, and is invoked immediately after loading of board program, as show in List 3.2.

Listing 3.2: Invoking host immediately after loading program onto the board.

```
...
load-run : main.srec
    e-loader -run-target
    $(MAKE) $(MFLAGS) -f ../Makefile ROOT_DIR=.. host-run
...
```

The default behavior of Makefile is to run host program after loading program into the board. Thus, it's possible to treat epiphany program as an ordinary PC programs, with STDOUT as its output, as shown in Figure 3.3. It's just like the program is running on the host, while, in fact, a couple of stages involving synchronization between host and the board are undertaken, which is hidden from end user and will be discussed in detail in synchronization section.



```
epiphany :: (fft*) » mk
Host is running ...
size: 256
Waiting for the board to finish...
c0: 5 c1: 5 c2: 5 c3: 5 c4: 5 c5: 5 c6: 5 c7: 5 c8: 5 c9: 5
Read data from board
0, 3138819
```

Figure 3.3: Running make on host, and result is shown on STDOUT of host.

## 3.4 Communication API

As is mentioned before, CAL has built-in support for consuming and producing tokens. Combined with the fact that connections are defined in network structure, generated tokens are passed to the right destination and consumed accordingly, which, in the end, achieves data transferring. In Epiphany architecture, there is no support for direct message passing, which is corresponding to how actors consumes and produces tokens in CAL, so we have to build communication API ourselves for Epiphany architecture. The communication API provides the interface to our C programs how to manipulate both input ports and output ports.

The proposed communicating APIs consist of:

- `epiphany_write` : Write value to this output port.
- `epiphany_read` : Read value from this input port.
- `has_input` : Check if one input port has certain number of tokens (token availability).
- `connect` : Connect two ports logically, so that data could be transferred seamlessly.
- `end_port` : Flush tokens in the buffer if any, and indicate the connected input ports that there's no further data anymore, inspired by end of packet in communication protocol.

### 3.4.1 Incremental Refinement

CAL Language Specification[2] doesn't enforce any communication model, but most implementations for CAL choose to queue tokens in FIFO for its simplicity and ubiquity. In our case, FIFO communication model serves our purpose well, so we will implement one data structure facilitating FIFO access on Epiphany architecture for actor-to-actor communication. Since Epiphany architecture doesn't support message passing natively, we will use shared memory to provide similar interface. In other words, actors are still communicating with each other by passing messages, but these are translated to corresponding direct memory access behind the scene.

Two special properties of Epiphany architecture should be mentioned explicitly, which provide some guidelines for what should be improved for each new version, before diving into any low level implementation. Firstly, according to the Epiphany architecture reference [12], write operation is a couple magnitude faster than read operation, which intuitively makes sense, for the program doesn't depend on write to be finished before continuing. Therefore, "Push-Driven" approach is taken in all implementations; in other words, the sender will push the data to the receiver side. Secondly, even though all the internal memory of each core is mapped to global address, the cost (latency) of accessing them is not uniform. Therefore, it's preferable to have some cache-like memory hierarchy for efficient execution. All implementations try to provide one good abstraction that the software cache is big enough so that there's no cache miss for the processor. Now, we are ready to discuss the implementations for the aforementioned communication API.

Version "destination-fifo" places the buffer in the destination core. In this version, the overhead of communication is borne by sender completely, for writing to memory belonging to other cores is expensive, compared to writing to local memory space. This is the simplest implementations possible, and the only synchronization is to ensure that

we don't overwrite data that has not been processed, and don't read data before data is written.

Version “both-fifo” places one buffer on both sender and receiver sides, as its name suggests. In this version, each core is always working on its local memory so that both write and read operations are relatively cheap. DMA is used for data transferring after both sides are ready, which requires sender's buffer is full and receiver's buffer is empty, specifically. Even though we are using DMA for data transferring, the CPU is inside the busy waiting for the DMA to finish. This version is mainly one transition from previous version to the next one.

Apparently, there's nothing two sides could do during the DMA process. Version “double-buffer” tries to solve this “idle” problem by introduction two buffers in each side. In this case, the core could work on the other buffer after one's finished processing. The rationale is enforcing two actors are producing and consuming tokens at the exact same rate is unrealistic, and the extra FIFO is used to mitigate the problem to some extent. Of course, if one side is significantly slower or faster than the other, API calls would be blocked eventually.

Figure 3.4, Figure 3.5 and Figure 3.6 provide graphical explanation on how FIFO is placed in these three implementations, respectively. Consistently, the source core is placed of the left of destination core, and the dashed indicates the connection between two actors. Red block means the state of the slot is unknown or it's not very important to what's discussed, white block refers to one empty slot and blue block indicates one occupied slot and the token inside has not been processed. The length of the FIFO is controlled by C macro definition, and 10 is used by default. If it's not specified, this is the convention that will be used for all the diagrams related to FIFO. Following, we will discuss some critical technique used in the implementation.

We have implemented the broadcast capability in all the three implementations of the communication API, so that it's possible have multiple input ports connected with one output port, because CAL allows broadcasting. However, the implementation is really simple, merely iterating over all the connected input ports, and transfer the data through the corresponding connection.

In all implementations, the synchronization between sender and receiver is achieved by polling the observed buffer directly. Because of this remote access a lot of traffic is generated during busy waiting. One possible optimization to reduce the traffic is to create local mirrors for the variables we are polling. In other words, the polling core should use the local mirrored version instead of the original variables in the remote core. Obviously, the observed core needs to update the mirrors in addition to the original variables, which might introduce some overhead.

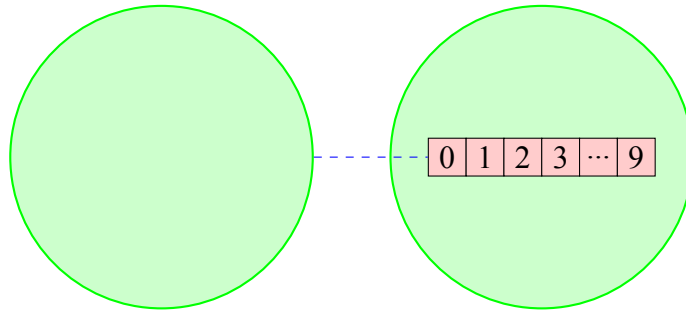


Figure 3.4: FIFO structure is placed in destination core.

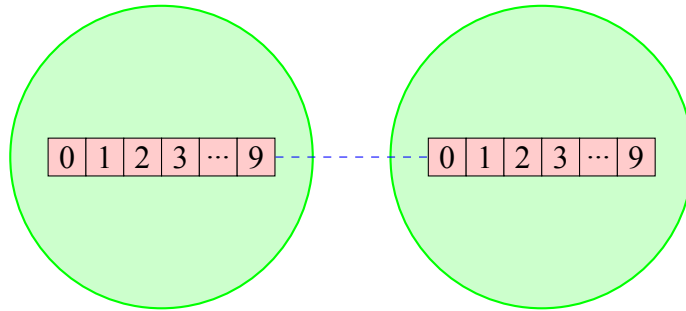


Figure 3.5: FIFO structure is placed in both source and destination core.

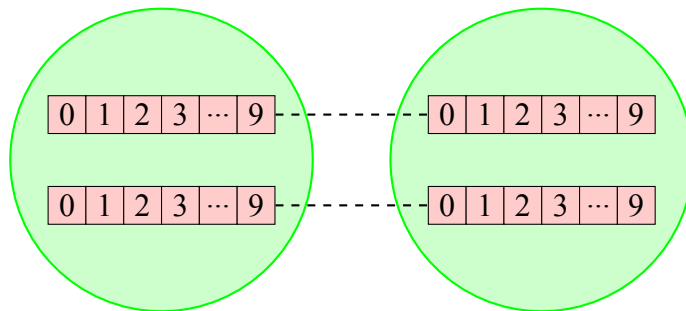


Figure 3.6: Two FIFO structure is placed in both source and destination core.



### 3.4.2 Concurrent programming

It's not surprising that we will be dealing with concurrent programming while implementing the communication API; after all each core's running independent of others and accessing (or mutating) shared resources. Concurrent programming is notoriously tricky to work with, and mostly, developers will resort to locks to enforce sequential execution for some part of the program, where there are concurrent accesses to shared resource, which should be covered at the beginning of almost all concurrent programming courses. Unfortunately, using locks often introduce performance penalty to the application. Therefore, lock-free algorithms, using atomic operations instead, become quite popular for efficient concurrent read or write.

In this project, the problem is to handle single producer and single consumer, which is much simpler than general multiple producers and multiple consumers problems. Because of this, it's possible to use single-writer policy, that is having only one core to write to one variable during one certain period, to simplify concurrent patterns so that we don't need to use lock or atomic operations.

The following is one example on how we could avoid the use of explicit locks or atomic operations. The scenario is involved with two buffers residing in source and destination sides (other than the first version of the communication API), and the shared mutable resource is the size of tokens in one buffer.

Figure 3.7 illustrates what's happening in the source core, and Figure 3.8 does the same for destination core. Here, we use the shared resource as its own lock to enforce regulated access: different values indicate difference accessibility. Source code will only mutate the shared resource (marking the buffer as full) when the buffer is empty. Similarly, destination core only mutate the shared resource (marking the buffer as empty) when the buffer is full. Apparently, only one core could mutate the shared resource at any time instance. The same technique is used for all the shared mutable resources in this project. Of course, this is possible only because of the memory order model used in Epiphany architecture[12].

### 3.4.3 Synchronous and Asynchronous Functions

Synchronous (also known as blocking) function call puts strict execution order on which part of the program should be running currently, and what should be followed subsequently. Asynchronous (also known as non-blocking) function call frees the caller from waiting for the result of one (possible) time-consuming operation so that the caller could do something else. Both of them have valid usage in reality, and they come in pairs in this implementation of this communication library. In other words, there's one corresponding asynchronous function for one synchronous function, and vice versa. There are two pairs of functions in the communication APIs that are worth mentioning, and we will discuss them in detail.

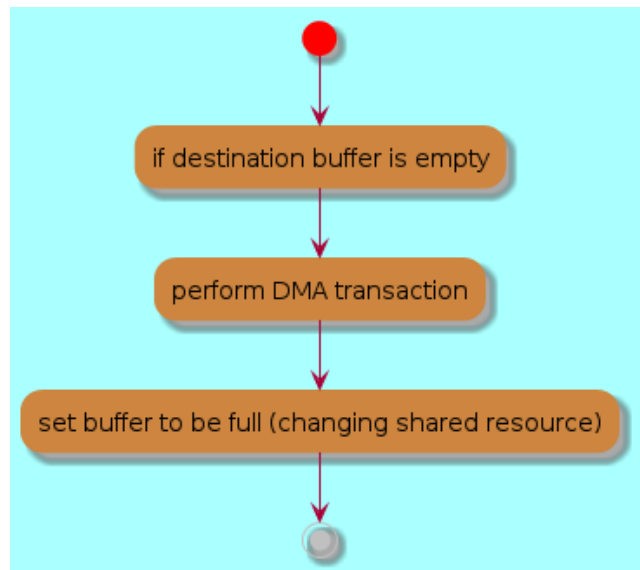


Figure 3.7: Lock free synchronization on source code.

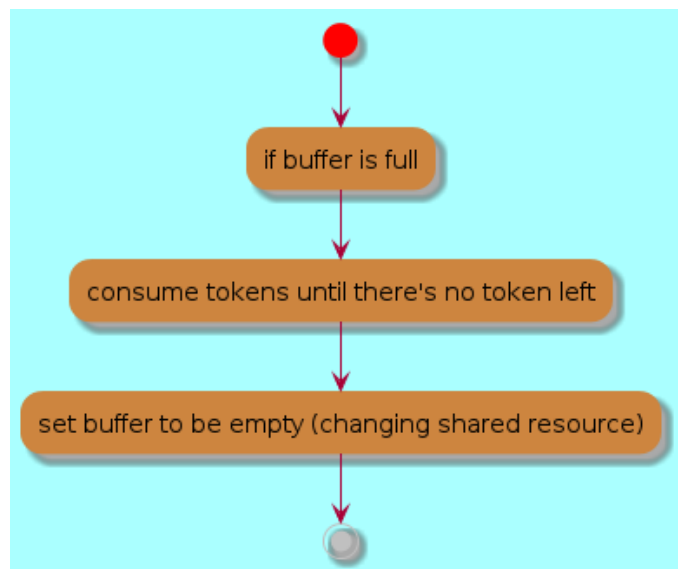


Figure 3.8: Lock free synchronization on destination code.

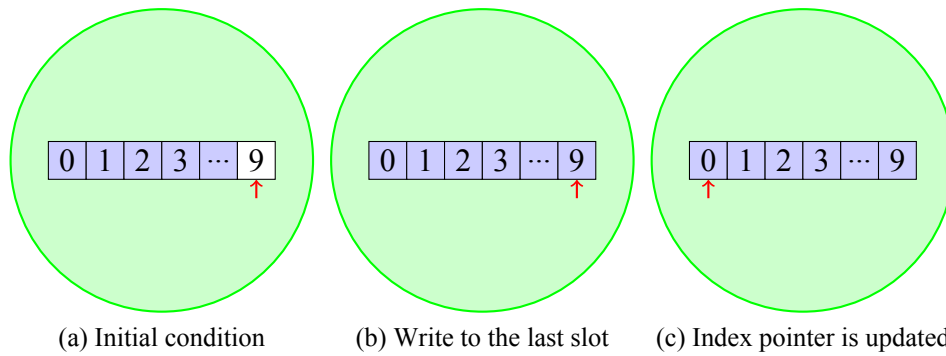


Figure 3.9: The consecutive state of one FIFO, when there's only one empty slot.

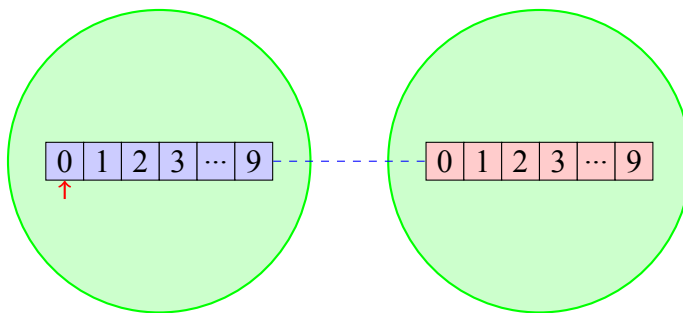


Figure 3.10: `do_flush` is called when the FIFO is full. At the moment, the DMA is either finished or hasn't started at all. The caller is blocked until the DMA is finished successfully.

As shown in List 3.3, there's great symmetry between these synchronous and asynchronous versions; basically, condition check in asynchronous is replaced by loop in synchronous version. It should be pointed out explicitly that busy-waiting in the synchronous version only makes sense in the current execution scenario, where only one actor instance is running on one core.

Listing 3.3: Flush one buffer so that the data arrives the destination core.

```
static void try_flush(fifo *b)
{
    ...
    // if source_fifo is full and dest_fifo is empty
    if (b->total > 0 && b->twin->total == 0) {
        // DMA channel is idle
        if (try_dma(b->dma)) {
            dma_copy(b->dma->id, b->twin->array, b->array,
                    sizeof(b->array), E_ALIGN_BYTE);
        }
    }
    ...
}

static void do_flush(fifo *b, uint size)
{
    ...
    // wait until source fifo is not empty
    while(!(b->total > 0)) ;
    // wait until source fifo is empty
    while(!(b->twin->total == 0)) ;
    // wait until dma is idle
    while(! try_dma(b->dma)) ;
    dma_copy(b->dma->id, b->twin->array, b->array,
            size, E_ALIGN_BYTE);
    ...
}
```

Figure 3.9, where only the sender core is drawn, and the red arrows indicates the slot will be occupied in the next write operation, presents how the state of FIFO is changed and the role of asynchronous function in this process. The scenario begins with the FIFO having only one free slot, as shown in Figure 3.9a. When one new token is generated and put into this FIFO by calling communication API, the state of FIFO will change to that shown in Figure 3.9b, and the asynchronous function is called before updating the index pointer. Since this is one asynchronous function, the caller exits without blocking and the resulting state is shown in Figure 3.9c. This function call would start DMA

transaction if all the following requirements are met: source FIFO is full, destination FIFO is empty, and there's one idle DMA channel. Otherwise it exits immediately.

Synchronous version is used to ensure that we don't overwrite the data that hasn't been transferred to the destination core yet. This function is called when the FIFO is full and the core tries to send one token, as shown in Figure 3.10. Since it's possible that DMA has not yet started because any of the three requirements are not met or alternatively, DMA has been started, but hasn't finished yet, this synchronous version has to take care of both cases so that it doesn't perform the same task more than once.

Listing 3.4: Flush one buffer so that the data arrives the destination core.

```
static void try_distribute(port_out *p, uchar current)
{
    if (p->buffers[current]->dma->status != DMA_IDLE) {
        try_flush(p->buffers[current]);
    } else {
        // iterate over the rest of input ports that are connected to
        // the current output port
        ...
        try_flush(p->buffers[current]);
        ...
    }
}

static void do_distribute(port_out *p, uchar current, uint size)
{
    do_flush(p->buffers[current], size);
    ...
    for (current_dest++; current_dest < p->dest_index;
        ++current_dest) {
        ...
        do_flush(p->buffers[current], size);
    }
    ...
}
```

The second pair is about distribution, which acts like broadcasting so that all the input ports connecting with one output port could receive the same data stream. Similar to previous pair, two versions are shown in List 3.4, and they are even more similar than the previous pair, due to the fact that condition checks and loops are encapsulated in previous pair.

Distribution is supported in any version of the three, but this pair is only used in version 3. In version “both-fifo”, the interval between calling ‘try\_flush’ and ‘do\_flush’ is only one token away, as shown in Figure 3.9b and Figure 3.10, so we don't have the

chance to call ‘try\_flush’ more than once and expect it to distribute the token automatically. However, if there are multiple FIFOs in each core, it’s possible to call ‘try\_flush’ every time we write one token to other FIFO, which is exactly the case in version 3. In fact, this pair is one level up on the abstraction ladder by encapsulating the logic of iterating along all the ports connected to the current, and the need for them emerges only when the number of underlying FIFOs in each core changes from one to two.<sup>3</sup> Therefore, neither ‘try\_flush’ nor ‘do\_flush’ is used explicitly in version “destination-fifo”, and developers could think on one higher level by building the communication APIs on top of the newer pair of functions.

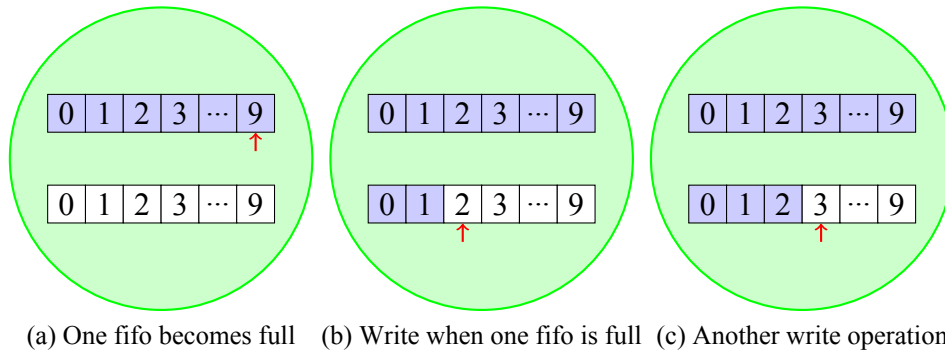


Figure 3.11: The state of the FIFOs, where the asynchronous function would be called.

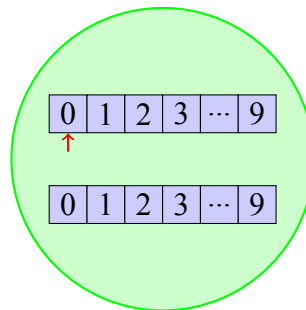


Figure 3.12: do\_distribute is called when all the FIFO are full. This function will pick up from where try\_flush is left and continue with the distribution process.

Figure 3.11 shows a few instances, where try\_distribute would be called, such as when one FIFO becomes full (the same as try\_flush), and each write operation when the other FIFO is full. Similar to version “both-fifo”, the synchronous function is called when there’s no empty slot, as shown in Figure 3.12.

<sup>3</sup>The difference between one and two is more than quantitative.

## 3.5 Synchronization

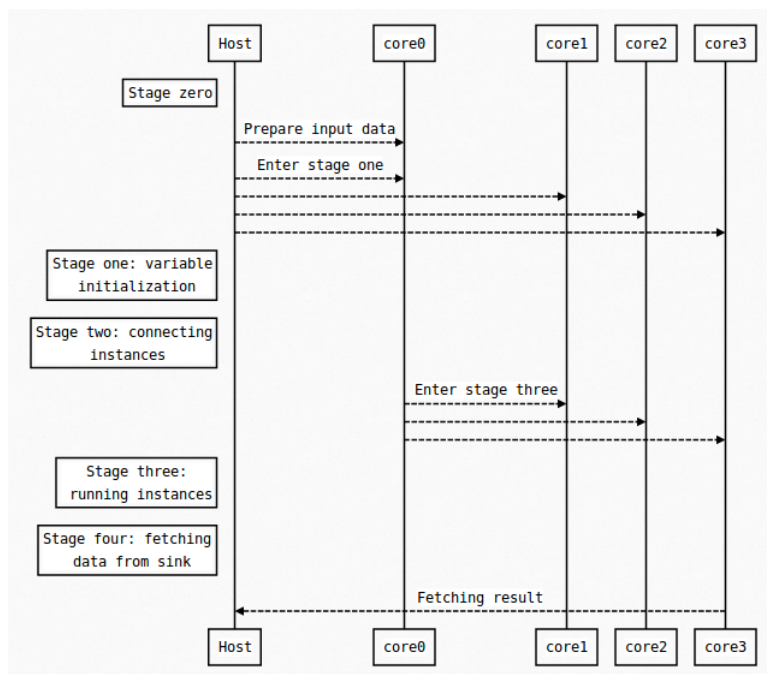


Figure 3.13: Synchronization of four cores and host, which could be extended to many cores horizontally.

All the cores on the board are running independent of each other. However, there are some special time instances where synchronization is required, such as host signals the board to start running after finishing preparing input data, informing all cores to run after all connections are established, etc. One example of the overall sequence of synchronization is illustrated in Figure 3.13.

In stage zero, input data is transferred from host to the board before starting the whole transaction. Once this process is finished, host will inform all the cores used in this transaction to enter stage one.

In stage one, all instances of actors in every core are initialized. Since we avoid using dynamic memory management, all variables are created on the stack. When it's finished on one core, it will enter stage two by its own.

All cores are synchronized before entering stage two to ensure the storage for all instances are allocated and properly initialized, which is inspired by `barrier[13]` concept in concurrent programming.<sup>4</sup> During stage two, one special core is chosen to be responsible to connect all the instances according to the network definition. In this case, `core0`

<sup>4</sup>It has built-in support in the latest Epiphany SDK.[14]

(32,32) is used for convenience reason. After connecting all the instances, this core will inform all cores used in this transaction to enter stage three, including itself.

In stage three, all instances will be running according to actor model, consume tokens, produce tokens, etc. This is the stage where actual work is done, and actor instances are running at its own pace.

Cores, reading data directly from network will enter stage four firstly, and call `end_port` to inform all the connected input ports of the end of the data stream.

The final core(s) have to write the data to particular storage, which will be fetched by host. Detailed information on how data is fetched by host will be covered in the following section.

## 3.6 Board and Host Interface

Observed from one high level, all actors have basically the same structure, a few internal variable local to this actor, which represents the state of one actor instance and a couple of actions, which will consume tokens from input ports and produce tokens to output ports. (Such uniformed structure is actually one indication how careful this language is designed.) However, after connecting all the actors according to the network definition, such consistence is lost due to the fact that the actors residing on the edge of the network must adapt themselves so that they could talk with some entity that is not an actor.

In order to regain the simplicity, we decide to break the whole operation into two phases. Firstly, translate one CAL file to C file, which has one-to-one correspondence. Secondly, identify the actors that are residing on the edge of networks, using information from NL files, and deal with them accordingly. This design decision is inspired by how C programs are built; compilation and linking are separate processes, even though it's possible to run both with one command.

Since we have decided to try to keep the generated C files intact, the most straightforward way to achieve it would be creating dummy actor instances that could talk to outside world and to ordinary actors as well. This approach is the least obtrusive one, for it doesn't modify generated code for actor instances but instead abstracts away the difference between outside world and real actors, so that each real actors could just proceed as if it's communicating with real actors.

However, due to the current strong requirement, running one instance on one dedicated core, the above solution would indicate that at least two cores are used for data transferring between host and the board, which could be one huge waste. Here, we propose one workaround for this problem. Considering one of the tasks the linker performs is to rename function name to memory address, we could do similar renaming so that this actor doesn't call the standard communication APIs anymore. Instead, it would use the new functions we provide. Perl is used for its universal availability and ease of use for simple tasks.



The Perl script would substitute the relevant communication APIs to:

- `networ_write` : Write value to the shared buffer between host and the board.
- `networ_read` : Read value from the shared buffer prepared by host.

The intrusive nature of this approach is hidden by the small Perl scripts, but it's error prone and possibly hard to maintain. In addition, augmented actors has access to external memory, which could introduce some noise in measuring latency of various versions of APIs. In other words, the statistics result of these augmented actors contains some noise introduced by slow memory access. The true culprit for this undesired scenario stems from the fact that one instance is monopolizing one core. One better solution would be discussed in future work, that solves this problem at its root.

## 3.7 Uniform actor interface

Listing 3.5: Main function for double.

```
// common/common.c
...
void core1_main(actor_double *a) {
    ...
    Mailbox.core.go[core_num()] = 0;

    stage(1);
    all.instance_double = a;
    actor_double_init(a);

    Mailbox.core.go[core_num()] = 2;
    stage_all(2);

    stage(3);
    // specific code for this actor
    ...
}
```

Listing 3.6: Main function for add.

```
// common/common.c
...
void core2_main(actor_add *a) {
    ...
    Mailbox.core.go[core_num()] = 0;

    stage(1);
    all.instance_add = a;
    actor_add_init(a);

    Mailbox.core.go[core_num()] = 2;
    stage_all(2);

    stage(3);
    // specific code for this actor
    ...
}
```

Since all actor instances are managed using the same synchronization mechanics, we need have some uniform access to instances. Otherwise, we will end up with sixteen functions with almost identical structure. List 3.5 and List 3.6 show the effective main function running on two cores. It could be observed that structure is unsurprisingly

similar, except the variable name and initializer for that particular actor.<sup>5</sup>

It's better to refactor this to avoid code duplication. This is the typical scenario for Strategy Pattern[15], running different code (behavior) on different core, but clients aren't aware of the implementation difference. Unfortunately, due to the fact that C does not support OOP (inheritance specifically) natively, some hacking is required to use this pattern. The entry point is to exploit the fact that any type of pointer could be up cast to void pointer automatically, which works like "Object" up casting in most OOP languages.

Each instance has interface to communicate with outside world, which is like public methods in OOP languages. For sure, the type information is different for different actors, and it has to be set properly according to the real definition of the actor. List 3.7 shows the defined public methods for one particular actor. The goal is to encapsulate the internal implementation details, and only expose these public methods to be called.

Listing 3.7: Actor Interface

```
// include/actor_add.h
...
typedef struct actor_add_struct {
    ...
    void (*run)(struct actor_add_struct *self);
    void (*end)(struct actor_add_struct *self);
    bool (*not_finished)(struct actor_add_struct *self);
} actor_add;
...
```

The following wrapper List 3.8 is used to abstract the difference among all actors to create one consistent interface, which is the interface all the subtypes would implement in one OOP language. Even though C has support for function pointer, the syntax is quite cryptic. Here, we are using some syntax sugar to make it less miserable.

Listing 3.8: Uniform Actor Interface

```
// include/common.h
...
typedef void run_t (void *);
typedef bool not_finished_t (void *);
typedef void end_t (void *);
typedef struct api_t_struct {
    run_t *run;
    end_t *end;
```

<sup>5</sup>The above code has been replaced by the solution introduced below, and could only be seen in the history of the online repository(<https://bitbucket.org/albertnetymk/epiphany>).

```

    not_finished_t *not_finished;
} api_t;
...

```

---

Implementing Strategy Pattern in C is accomplished in two steps. Firstly, the actor itself would expose the pointers to these functions, as shown in List 3.9. In these assignments, auto up-casting to void pointer is performed by compilers. After this, all these functions become methods bound to one instance (only pointers are copied). The resulting entity is quite similar to objects in OOP.

Listing 3.9: Actor Interfaces are Exposed

```

// inside one actor constructor
...
a->run = &run;
a->not_finished = &not_finished;
a->end = &end;
...

```

---

Secondly, the pointers are cast manually (down-casting), shown in List 3.10, to conform the same interface as. Since C is one static type language, down-casting have to be made explicitly to satisfy the compiler. This 'init' function would return the pointer to one actor instance with the consistent APIs, discussed in List 3.8 while being called. This function is passed as one argument to the effective main function, as shown in List 3.12.

Listing 3.10: Actor Interfaces are Initialized

```

// board/core2/main.c
...
static inline api_t *init(void *a)
{
    ...
    api.run = (run_t *)all.instance_double1->run;
    api.end = (end_t *)all.instance_double1->end;
    api.not_finished = (not_finished_t
        *)all.instance_double1->not_finished;
    return &api;
}
...

```

---

Listing 3.11: Type of init function

```
...
typedef api_t *init_t (void *);
...
```

---

Listing 3.12: Pass init as one argument to the effective main function.

```
// board/core2/main.c
...
core_main(address_from_coreid(mycoreid, &instance_double1), &init);
...
```

---

‘core\_main’ in list 3.13 is the effective main function, and its second argument is the pointer pointing to the ‘init’ function. One sugar type defined in list 3.11 is used to simplify the main function signature. Since all the ‘dirty’ work has been done by the actor who provides all APIs, it’s quite simple to consume them, as shown in list 3.13.

Listing 3.13: How Actor Interface is Used

```
// common/common.c
...
void core_main(void *a, init_t *init) {
    ...
    while(api->not_finished(a)) {
        api->run(a);
    }
    api->end(a);
    ...
}
...
```

---

It would be quite simple and easy to implement Strategy Pattern if inheritance is supported. However, that’s just one good wish, and we are stuck with the type casting dance in order to mimic inheritance in C.

# Chapter 4

## Result

In this chapter, result of this project is presented by using two-dimensional inverse discrete cosine transform (2D-IDCT) as one case study. The distinguishing feature from sequential version is the communication part, how data is transferred from one actor to another. Therefore, we would focus on how different implementations of communication API perform in the evaluation process.

### 4.1 IDCT

2D-IDCT is one component of MPEG standard video decoders, and consists of 15 actors in this implementation, which consists of two one-dimensional IDCT, surrounded by dotted line, shown in Figure 4.1.

### 4.2 Mapping actors to cores

Theoretically, there's no constrain on which actor could be mapped to which core, so one very simple mapping, 'sequence mapping', could be assign actors in the sequential order, *i.e.*, the actor sequence 0 to 15 is mapped to core id 0 to 15, thereby ignoring the fact that, *e.g.*, core 3 is not the nearest neighbor to core 4, illustrated in Figure 4.2a. One obvious improvement for this application is to take into account the physical layout of the cores on the chip so that consecutive actors are mapped into neighboring cores, illustrated in Figure 4.2b. We would call this one 'snake mapping', for it resembles the shape of a snake.

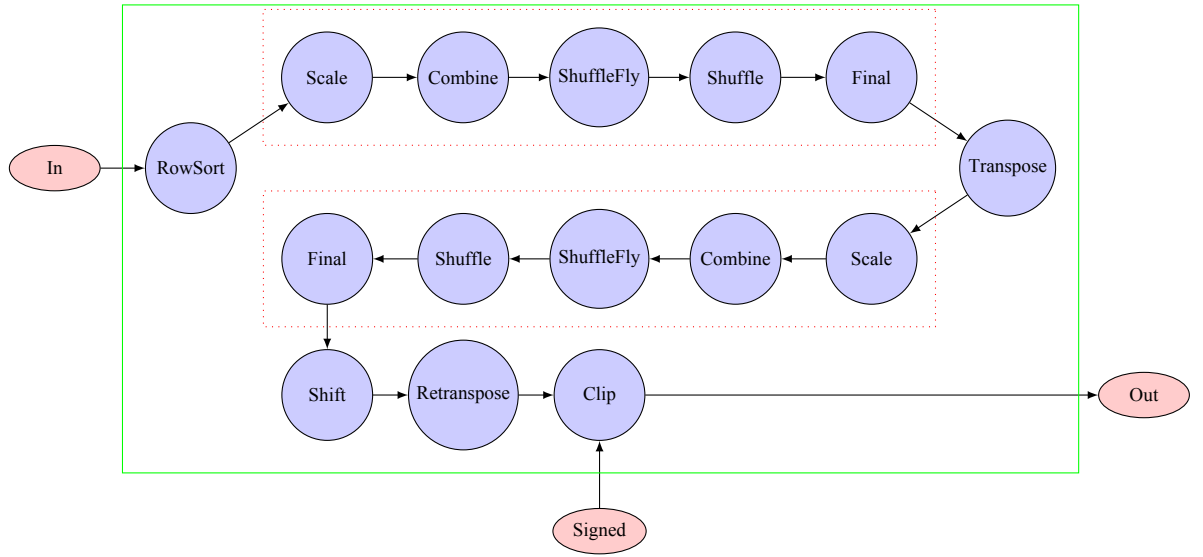


Figure 4.1: IDCT 2d actor diagram

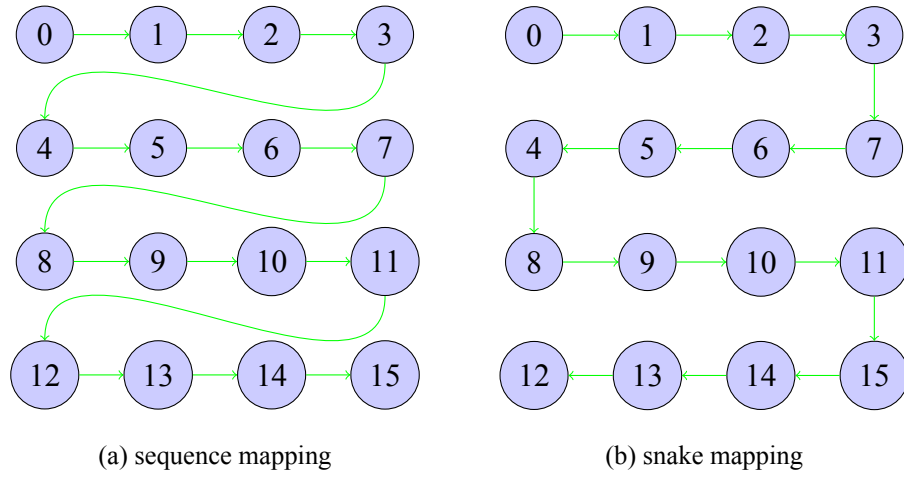


Figure 4.2: Mapping IDCT to cores on the chip

## 4.3 Configuration

We will be using 64,000 input data for all transaction process<sup>1</sup>, and the same transaction is run for 10 times to get the average cycle statistics. We will be using six implementations (versions) containing: three plain implementations, and the “polling on local variable” optimization, discussed in method chapter, applied to each of them. For each of the implementation, we would have three different buffer size. The memory footprint of the three buffering methods is kept equal, thus the buffer size is 16, 8, 4 words for ‘v1’, ‘v2’, ‘v3’, respectively, in 16 column. The same calculation applies for other columns as well.

The execution of the whole application (starting from the token consumed by the first actor until the last token emitted by the last actor) for all the configuration is illustrated in Table 4.1 and Table 4.2. As it turns out, different mappings don’t affect the result too much, so we would only present and discuss the result for ‘snake’ layout in the following.<sup>2</sup> Looking the total execution time, we couldn’t really understand how the communication library works with different configurations. Therefore, In order to have better understanding how this communication library works, we will also profile each actor (core) to see how much communication overhead each actor bears.

	16	100	200
v1	245,233,581	252,670,393	260,405,356
v2	243,233,938	255,006,720	268,633,440
v3	254,580,851	257,608,918	267,914,764
v1o	245,056,209	252,191,323	260,613,330
v2o	241,572,488	251,880,625	266,303,065
v3o	252,285,767	256,014,000	266,586,651

Table 4.1: Clock cycles for the whole application using sequence mapping.

The total execution cycle consists of two parts, actual computation and communication API time, which could be further divided into two parts, reading and writing. Considering that both functions are asynchronous functions by default, but fall back to synchronous if necessary, each of them could be subdivided into actual IO time and polling time. In summary, in addition to total execution cycle information, four other cycle is interesting to us:

- read : Cycles spend on calling ‘read’ API.

<sup>1</sup>Due to the particular ratio between two input ports, 64,000 input tokens go to one input port and 1000 tokens go to the other, actually.

<sup>2</sup>Complete collection of raw data and their figures is available at <https://github.com/albertnetymk/idct2d>

	16	100	200
v1	243,886,058	251,534,258	260,526,600
v2	243,957,889	253,076,020	267,458,261
v3	254,432,450	257,356,452	267,935,371
v1o	244,089,624	251,348,275	259,771,477
v2o	240,158,168	251,149,725	266,992,006
v3o	251,474,199	254,953,684	264,720,248

Table 4.2: Clock cycles for the whole application using snake mapping.

- `write` : Cycles spend on calling ‘write’ API.
- `p-read` : Cycles spend on polling inside ‘p-read’ API.
- `p-write` : Cycles spend on polling inside ‘p-write’ API.

Because actors will be running only if there are certain number of tokens on specified input ports, actors would be checking the token availability all the time if there’s no tokens, which is effectively ‘p-read’ tried to capture. Since, actor machine has generated corresponding code for this kind of polling, we can’t see ‘p-read’ in all the figures, and the computation time (including testing token availability) is not fixed because of this.

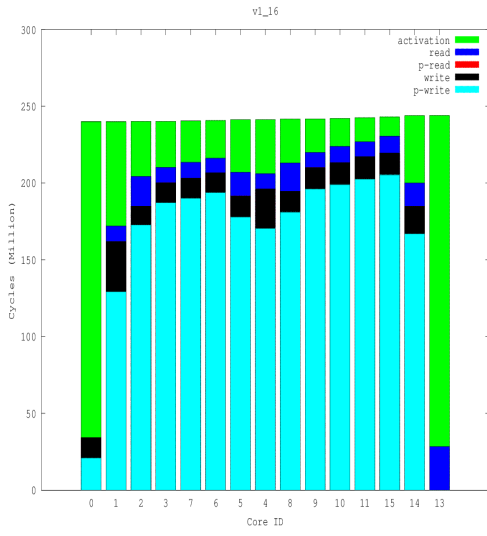
Another thing to note for all the cycles information is that the statistics for the first core (left most) and last core (right most) is different from the rest, due to the fact that we are using augmented communication API to handle network edge data transferring. They are shown in all the figures for the integrity and completeness of data, but will not be discussed in the following.

In both Figure 4.3 and Figure 4.4, two kinds of trend are easy to capture. Firstly, as the buffer size grows, looking at each row, especially the first and the third row, the first few actors spend less time on communication, because the buffering starts to take effect. Secondly, except the last few actors, which are dominated by the slow access to the shared memory, the communication time drops as we go from ‘v1’ to ‘v3’, looking at each column, especially with small buffer size (the first column).

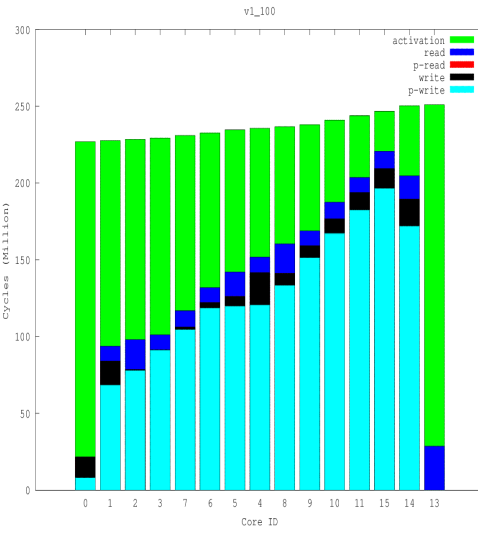
## 4.4 Summary

According to all figures presented above, performance (measured in clock cycles) could be improved by increasing buffer size, which is like increasing data cache size, and by using DMA for core-to-core data transferring, which is like using another thread to increase parallelism.

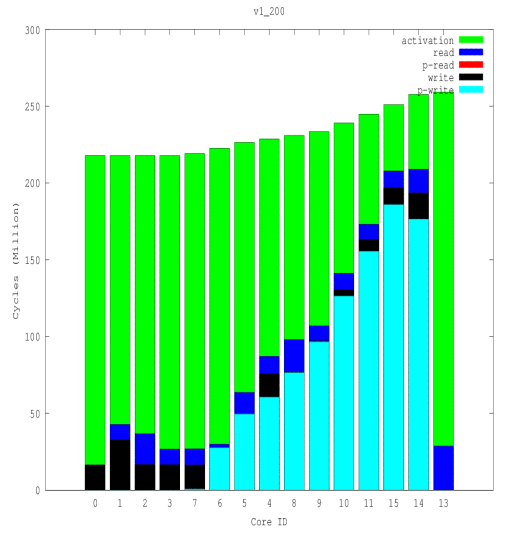




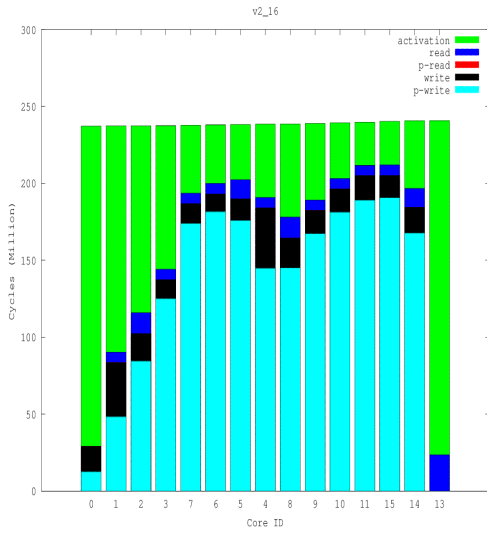
(a) v1 with buffer size 16



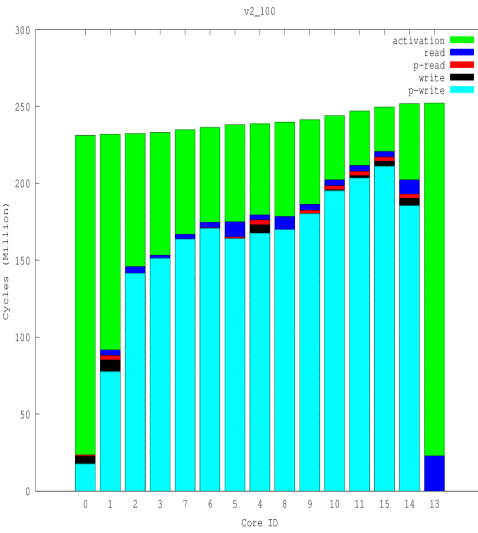
(b) v1 with buffer size 100



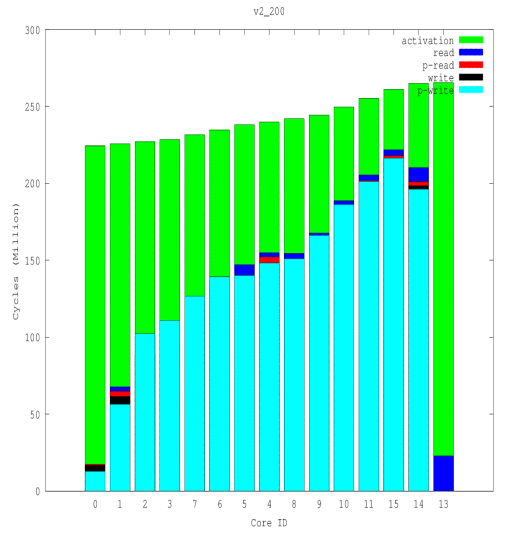
(c) v1 with buffer size 200



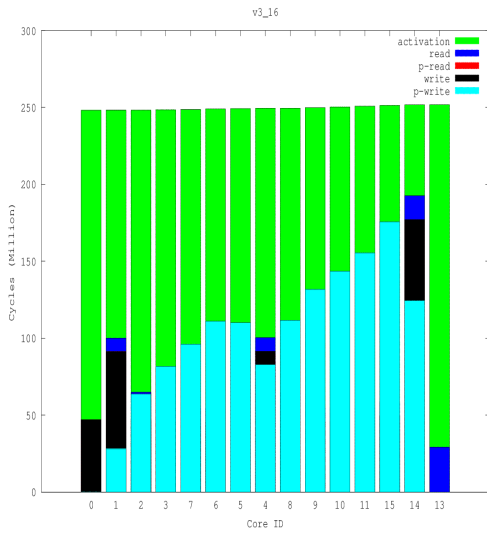
(d) v2 with buffer size 16



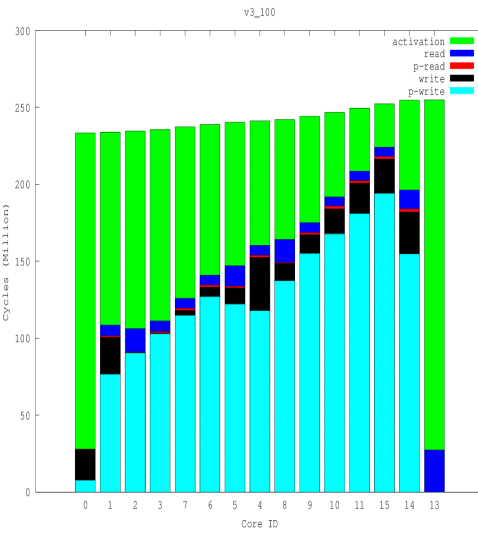
(e) v2 with buffer size 100



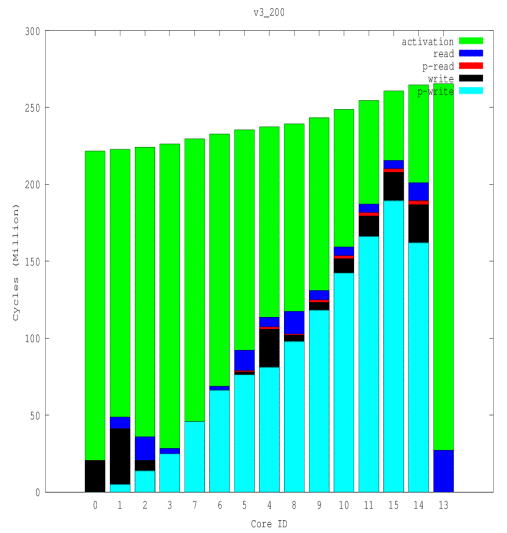
(f) v2 with buffer size 200



(g) v3 with buffer size 16

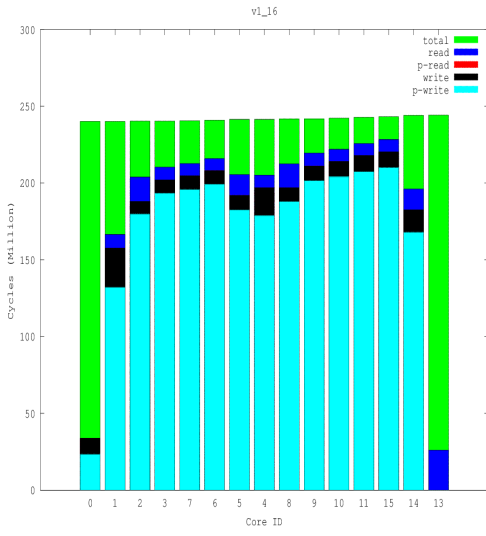


(h) v3 with buffer size 100

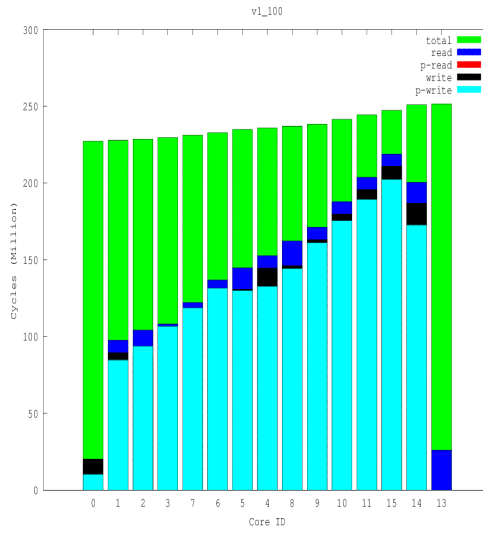


(i) v3 with buffer size 200

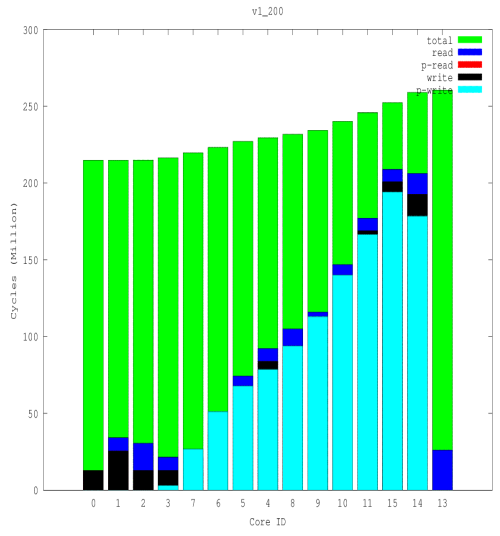
Figure 4.3: Three plain implementations with different buffer size.



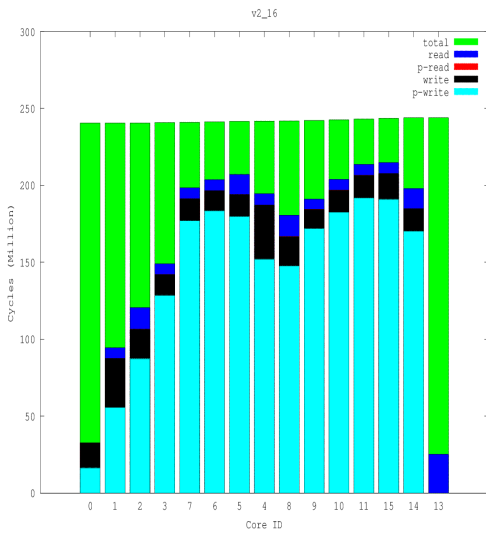
(a) v1o with buffer size 16



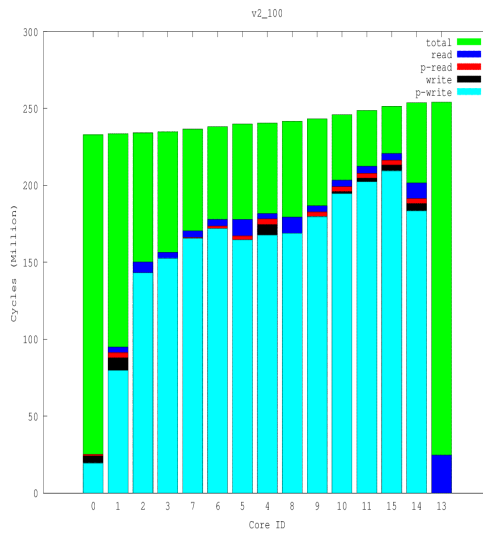
(b) v1o with buffer size 100



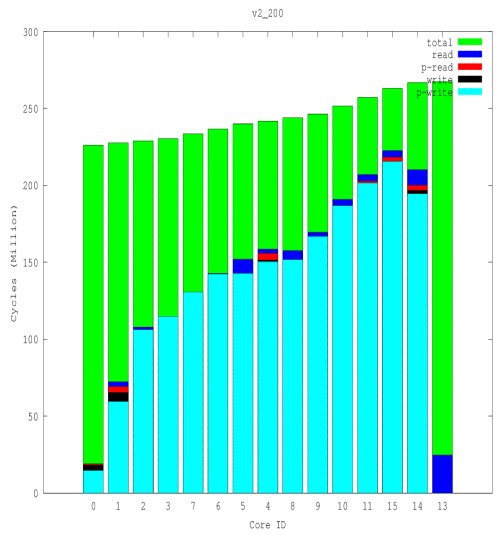
(c) v1o with buffer size 200



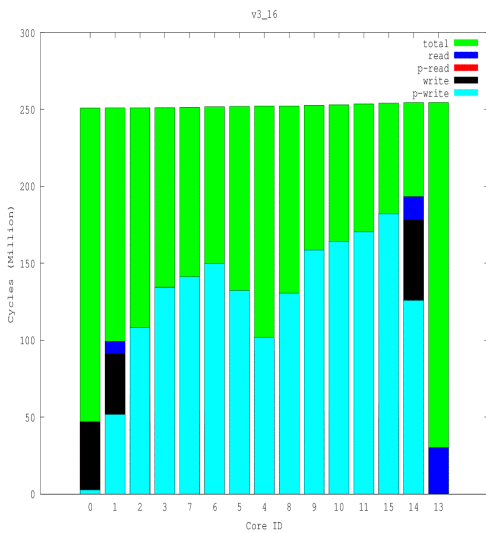
(d) v2o with buffer size 16



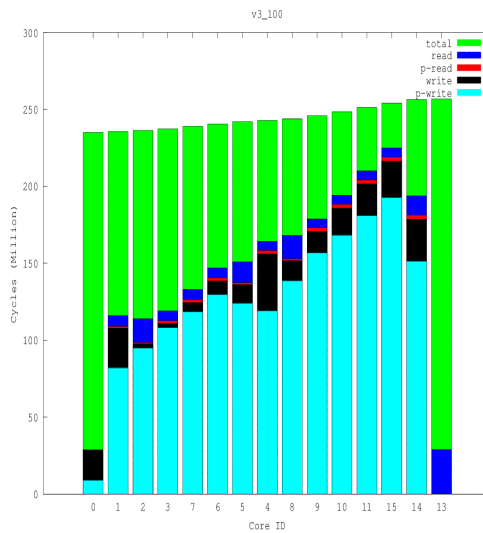
(e) v2o with buffer size 100



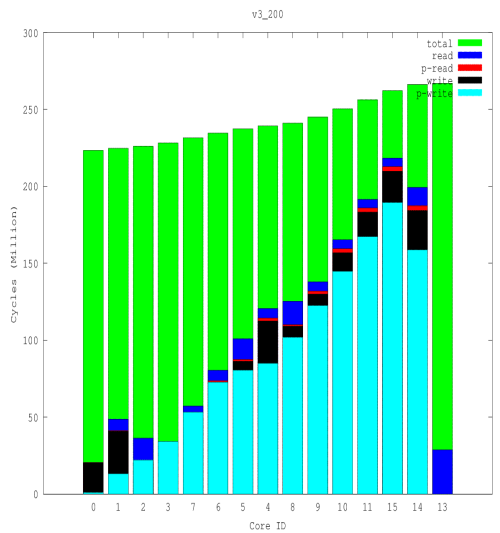
(f) v2o with buffer size 200



(g) v3o with buffer size 16



(h) v3o with buffer size 100



(i) v3o with buffer size 200

Figure 4.4: Applying ‘polling-on-local’ optimization three implementations with different buffer size.

# Chapter 5

## Conclusion and Future Work

In this chapter, some conclusion is drawn from previous result. In addition, the future work is discussed so that further work could be continued on top of this.

### 5.1 Conclusion

This project presents how to construct message passing interface on top of shared memory architecture and generating C code in OOP fashion. Combined with existing front end, the final delivery is one functional code generator for CAL Actor Language targeting Epiphany architecture. Writing concurrent C programs directly could be challenging, but with the existence of CAL, developers could express the concurrency more explicitly and let the code generator to figure it out how to map the declarative statements to low level imperative instructions.

Close look at how communication API performs reveals that buffer utilization is pretty high, since the time spent on polling is almost negligible except for actors which are slowed down by external memory access (which has higher latency compared to internal memory access). Further optimization could be focused on execution inside one actor, and how to isolate actors from direct slow memory access.

### 5.2 Future Work

This section only provides some suggestion from the author's perspective, and how it will evolve is beyond author's imagination.

#### 5.2.1 Multiple instances in one core

As discussed at the very beginning, we constrain ourselves that each core is only responsible for one instance of actor. Such requirement is valid and preferred for prelim-

inary work in one field, to protect us from being overwhelmed by the complexity of the problem. However, this condition makes our deliverable almost infeasible in real world. After all, one of the most significant purposes of compiler is to free developers from various hardware architecture or physical structure, so that they could focus on the problem they are trying to solve. In other words, compilers should be adapted to developers, not the other way around. Therefore, one natural improvement could be to have the ability for multiple instances to share one core. In this way, developers could create as many as instances depending on the problem not depending on this hardware this program's going to run on. We will discuss this issue a little bit, which might shed some light for further development, hopefully.

Considering thread is one concept created to solve the similar problem, we could stand on the shoulder of giants by reusing this idea and tweaking it so that it's not so heavy for embedded systems. Using any thread library would not be feasible due to the large footprint of scheduler and associated libraries to isolate individual thread. It's acceptable to go for cooperative scheduling and push the burden to ensure all instances are working properly (no dead lock or monopolizing one core permanently) to the developers or static analyzers. Considering the limited space, it's better to have one runtime that's as light as possible. Therefore, the scheduler could basically ask each instance, "Are you runnable?". If so, the control would be handled to this instance until it hands the right out (typical cooperative scheduling).

Considering there's no cache in any of the cores, the whole approach, having multiple instances in one core, would introduce one huge penalty in performance. In order to regain the benefit as we did in accessing local variables, we could set up caches in software level. In other words, each core will copy all the data owned by all the instances running (or being runnable) on this core to the local space, which mimics how hardware cache works.

Having decided using thread-like data structure, cooperative scheduling and software cache, one major question left is how to distribute all instances to various cores. Static analysis might not be enough to calculate this distribution, due to the diversity of all the hardwares. In other words, it's quite difficult to determine whether one actor is running faster than another based on pure static analysis. Therefore, we prefer the runtime information in order to achieve load balance.

Using runtime analysis, one simple solution would be putting all available instances in one pool, the scheduler in each core could just ask the pool for the next available instance. However, due the overhead of swapping instances in and out of one core, this approach could be improved dramatically by iterating through the local pool, instead of the global one. Hopefully, in this way, the instances could be quite stable and are not moved from one core to another all the time, inspired by thread affinity. For sure, all kinds of details have to be addressed during implementation, and it could be very challenging; just considering the length required to explain it briefly.

### 5.2.2 Compose instances to form one heavy instance

Another approach to address the similar problem, each is core is under used for one instance, is to compose instances before code generation. One of the promises of actor machine is to compose instances so that the resulting instance is “heavy” enough to be run on one thread.

### 5.2.3 Unit testing

No one can be sure that the program works without testing it, even if it’s proved correct[18]. Testing provides one objective proof that the code is doing what developers have in mind.<sup>1</sup> Another purpose of having tests is to refactor code with confidence. People tend to be very careful while dealing difficult problems, but a little careless for simple tasks. Refactoring is not one complicated process, and some common procedures (mainly syntactic manipulation) are even provided by some IDEs. It’s wise to do incremental refactoring while running testing along the way, so it’s simple to pinpoint where we introduce the bug.

Ideally, unit testing should be the one running while refactoring, but due to the incompetency of the author in writing unit tests for concurrent code, only acceptance tests are conducted in this project, and they serve as the guard while the author is refactoring. It will be great if someone, who is experienced in concurrent programming and TDD, could improve the current situation by adding the unit tests. It would be one pity that implementation becomes legacy code while it’s still one prototype[23].

### 5.2.4 Merging multiple connections into one

Currently, each pair of output and input port form one connection, and all the different FIFO placement we are using affect individual connection. In order to simplify the interface of controlling or tuning the connection, all connections are treated equally. In other words, all the connections are homogeneous, so all connections have the same buffer size, that could be controlled globally. However, this simplification could lead to huge waste if some connections are used rarely. Here’s one trade-off one has to make: either go for specifying the buffer size for each connection, which would require developers to come up with one number for each connection, or go for unified buffer size for all the connections (the one used in this project), which is too inflexible. One workaround, that might mitigate the latter issue to some extent, could be merging connections, that have the same source and destination actor instances, into one channel. The intended scenario is there are two connections, one for options, the other for data, between source and destination actor. The one for options might have quite few tokens compared with the data

---

<sup>1</sup>Incorrect testing code could lie, but that’s another story.

connection. Therefore, having the same buffer size for them is not very space efficient. Since the two connections have the same instance on both ends, it should be possible to mimic two connections using one underlying real connection this project talks about.

### **5.2.5 Out of Order Broadcasting**

As discussed in Chapter Method, broadcasting is supported in all three implementations, but it's implemented in one rather naive way. Basically, the sender iterates over all the receivers one by one in sequential manner, which indicates the sender will not move to the next receivers unless the current receiver receives the data. The drawback of this mechanics becomes apparent especially in the second and third versions, namely 'both-buffer' and 'double-buffer', for DMA transaction requires three conditions to be started and having any of them unsatisfied could hinder the iterating process. One better iterating approach could be viewing all the receives as one set, pick up the first receiver that's ready for data receiving, and start the data transferring.

# Appendix A

## Tweaks to Orcc and d2c

### A.1 Orcc

Orcc provides options for multiple back-ends, but the discussion will be constrained on C back-end. Their web page provides detailed instructions on how to install on Eclipse platform. The documentation of Orcc encourages users to download applications from their website and start with them. However, I don't agree that it's very straightforward, for all the examples provided online are rather large and complicated from the perspective of one end user. I would like to translate some actors I provided, so that I could reuse some techniques. I started with the simplest example, shown in Listing A.1. Unfortunately, it's complaining immediately, saying it's invalid syntax, which is very surprising for this definition comforts the CAL Language Specification, and is one example used on Wikipedia.

Listing A.1: ID Actor

```
actor ID () In ==> Out :  
    action In: [a] ==> Out: [a] end  
end
```

---

Listing A.2: ID Actor(Orcc version)

```
actor ID () int In ==> int Out:  
    action In:[a] ==> Out:[a]end  
end
```

---

Having studied some “valid” code according to Orcc, I was able to update it to Listing A.2 with minor changes. Then, I tried to generate C code based on one dummy network file, which only contains one instance of this actor. Unfortunately, it generated

all the infrastructure code except the code generated from this particular actor. Probably that I missed something in my own project, so let's import one simple project from CAL application repository. Fortunately, this one works quite smoothly, and generates C code for all the actors in the network. Since I have got two projects, one is working, the other is failing mysteriously, I incrementally modify the working one to resemble the failing one so that I could pinpoint the culprit. Unfortunately, the situation wasn't improved even the code for actors and network is identical. All of the sudden, I realized that the project icon was different for these two projects, which turned out to be the real cause. After created one "Orcc" project, I finally could generate C code for actors.

On the first glance, the documentation of Orcc is pretty well written, and the step-by-step instruction is clear as well. However, it could become more user friendly if it could provide some hello-world examples to get started.

Figure A.1 shows the code generation of two instances of one actor in diff view. It could be seen that C files generated by Orcc is instance based. Because of that, there's little difference between these two files. Figure A.2 presents how the action scheduler is implemented in C.

```

60 //////////////////////////////////////////////////
61 // State variables of the actor
62 static i32 toA;
63 static i32 n;
64
65 +- 64 lines: //////////////////////////////////////////////////
129
130
131 //////////////////////////////////////////////////
132 // Token functions
133
134 static void read In() {
135     /* Input port a_In not connected */
136     index_In = 0;
137     numTokens_In = 0;
138 }
139
140 static void read end In() {
141     /* Input port a_In not connected */
142     static void write A() {
143         index_A = a_A->write_ind;
144         numFree_A = index_A + fifo_i32_get_room(a_A, NUM_READERS_A);
145     }
146
147 static void write end A() {
148     a_A->write_ind = index_A;
149 }
150 static void write B() {
151     index_B = a_B->write_ind;
152     numFree_B = index_B + fifo_i32_get_room(a_B, NUM_READERS_B);
153 }

```

```

60 //////////////////////////////////////////////////
61 // State variables of the actor
62 static i32 toA;
63 static i32 n;
64
65 +- 64 lines: //////////////////////////////////////////////////
129
130
131 //////////////////////////////////////////////////
132 // Token functions
133
134 static void read In() {
135     /* Input port b_In not connected */
136     index_In = 0;
137     numTokens_In = 0;
138 }
139
140 static void read end In() {
141     /* Input port b_In not connected */
142     static void write A() {
143         index_A = b_A->write_ind;
144         numFree_A = index_A + fifo_i32_get_room(b_A, NUM_READERS_A);
145     }
146
147 static void write end A() {
148     b_A->write_ind = index_A;
149 }
150 static void write B() {
151     index_B = b_B->write_ind;
152     numFree_B = index_B + fifo_i32_get_room(b_B, NUM_READERS_B);
153 }

```

Figure A.1: Orcc Actor translation for two instances of one actor.

## A.2 d2c

There's one attached report in d2c deliverable, that provides detailed explanation for their work. With this document and the "README" of this project, one should be able to start the build process. Unfortunately, it's not working out of the box; some



```
void a_scheduler(struct schedinfo_s *si) {  
    int i = 0;  
    si->ports = 0;  
  
    read_In();  
  
    write_Out();  
  
    while (1) {  
        if (numTokens_In - index_In >= 1 && isSchedulable_untagged_0()) {  
            int stop = 0;  
            if (stop != 0) {  
                si->num_firings = i;  
                si->reason = full;  
                goto finished;  
            }  
            untagged_0();  
            i++;  
        } else {  
            si->num_firings = i;  
            si->reason = starved;  
            goto finished;  
        }  
    }  
  
finished:  
    read_end_In();  
    write_end_Out();  
}
```

Figure A.2: Orcc actions scheduler

modification has to be made, which is explained in appendix. The build process fails due to illegal access level of one variable, shown in Figure A.3.

```
[javac] /home/albert/Downloads/opendf/contrib/actorsproject/build/java/xdAST/ASTNode.java:330: error: childIndex has private access in ASTNode
[javac]     if(node != null) { node.setParent(this); node.childIndex = i; }
[javac]                  ^
```

Figure A.3: Access level error in build process.

It could be fixed by changing line 279 of file ' /opendf/contrib/actorsproject/build/java/xdAST/ASTNode.java' as shown in List A.3. Since this Java code is generated using compiler compiler technology, it's possible the result code is not compatible to the latest JDK. (The build process is perform using OpenJDK 7.)

Listing A.3: Fix for access level error.

```
// private int childIndex;
public int childIndex;
```

The build process would proceed until another error on illegal access level caused by file ' /opendf/contrib/actorsproject/build/java/xlimAST/ASTNode.java' pops up. The same fix could be applied in this case again, for line 459. By now the build process should finish successfully, and installation could be finished smoothly as well.

```
schedule fsm init:
  init (init) --> run;
  run  (run)  --> run;
end
```

Figure A.4: D2c actions scheduler in CAL

After that one example could be run as instructed, and the followings are results. Figure A.6 shows the same strategy used in Orcc, translating two instances of one actor results into almost identical code. Figure A.5 illustrates the actions scheduler, that's generated from Figure A.4. It's not very obvious that the two are describing the same thing.

```

ART_ACTION_SCHEDULER(SingleDelay_0_action_scheduler) {
    ActorInstance_SingleDelay_0 *thisActor=(ActorInstance_SingleDelay_0*) pBase;
    const int *exitCode=EXIT_CODE_YIELD;
    ART_ACTION_SCHEDULER_ENTER(1,1);
    ART_ACTION_SCHEDULER_LOOP {
        ART_ACTION_SCHEDULER_LOOP_TOP;
        bool_t t26;
        int32_t t17;
        t17=pinAvailOut_int32_t(OUT0_Out);
        t26=(1)==thisActor->s0_s0;
        if (t26) {
            if ((t17>=(1))) {
                ART_FIRE_ACTION(SingleDelay_0_a0_init);
                thisActor->s0_s0=(2);
            }
            else {
                exitCode=exitcode_block_Out_1; goto action_scheduler_exit;
            }
        }
        else {
            bool_t t29;
            t29=(2)==thisActor->s0_s0;
            if (t29) {
                int32_t t10;
                t10=pinAvailIn_int32_t(IN0_In);
                if ((t10>=(1))) {
                    if ((t17>=(1))) {
                        ART_FIRE_ACTION(SingleDelay_0_a1_run);
                        thisActor->s0_s0=(2);
                    }
                    else {
                        exitCode=exitcode_block_Out_1; goto action_scheduler_exit;
                    }
                }
            }
            else {
                exitCode=exitcode_block_In_1; goto action_scheduler_exit;
            }
        }
    }
}

```

Figure A.5: D2c actions scheduler in C

```

+ 38 +- 10 lines: {"Out", sizeof(int32_t)}-----+ 38 +- 10 lines: {"Out", sizeof(int32_t)}-----
40
41 static const ActionDescription actionDescriptions[] = {
42   {"init", portRate_0, portRate_1},
43   {"run", portRate_1, portRate_1}
44 };
45
46 ActorClass ActorClass_SingleDelay_0 = INIT_ActorClass(
47   "SingleDelay",
48   ActorInstance_SingleDelay_0,
49   SingleDelay_0_constructor,
50   0, /* no setParam */
51   SingleDelay_0_action_scheduler,
52   0, /* no destructor */
53   1, inputPortDescriptions,
54   1, outputPortDescriptions,
55   2, actionDescriptions
56 );
57
58
59 ART_ACTION(SingleDelay_0_a0_init, ActorInstance_SingleDelay_0) {
60   ART_ACTION_ENTER(SingleDelay_0_a0_init, 0);
61   pinWrite_int32_t(OUT0_Out, 1);
62   ART_ACTION_EXIT(SingleDelay_0_a0_init, 0);
63 }
64
65 ART_ACTION(SingleDelay_0_a1_run, ActorInstance_SingleDelay_0) {
66   int32_t t2;
67   ART_ACTION_ENTER(SingleDelay_0_a1_run, 1);
68   t2=pinRead_int32_t(IN0_In);
69   pinWrite_int32_t(OUT0_Out, t2);
70   ART_ACTION_EXIT(SingleDelay_0_a1_run, 1);
71 }
72
73 static const int exitcode_block_Out_1[] = {
74   EXITCODE_BLOCK(1), 1, 1
75 };
76
77 static const int exitcode_block_In_1[] = {
78   EXITCODE_BLOCK(1), 0, 1
79 };
80
81 ART_ACTION_SCHEDULER(SingleDelay_0_action_scheduler) {
SingleDelay_0.c [c] 76,0-1 35% 2 SingleDelay_1.c [c] 60,2 35%
SingleDelay_1.c" 133L, 3355C
[0] 1:bash- 2:bash 3:bash 4:bash 5:bash 6:bash 7:vim 8:bash 9:vimdiff+ "albert-laptop" 00:26 17-Feb-13

```

Figure A.6: D2c Actor translation

# Bibliography

- [1] Noflojs.org. n.d.. *NoFlo | Flow-Based Programming for JavaScript*. [online] Available at: <http://noflojs.org/> [Accessed: 3 Aug 2013].
- [2] Eker, J. and Janneck, J. 2003. *CAL language report*. Berkeley: Electronics Research Laboratory, College of Engineering, University of California.
- [3] Orcc.sourceforge.net. n.d.. *Open RVC-CAL Compiler Orcc*. [online] Available at: <http://orcc.sourceforge.net/> [Accessed: 3 Aug 2013].
- [4] Opendf.org. n.d.. *Parkerad hos Loopia*. [online] Available at: <http://opendf.org/index.html> [Accessed: 3 Aug 2013].
- [5] Shuvra S Bhattacharyya, Gordon Brebner, Jörn W Janneck, Johan Eker, Carl Von Platen, Marco Mattavelli, and Mickaël Raulet. 2009. Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News*, 36(5):29–35.
- [6] Ghislain Roquier, Endri Bezati, and Marco Mattavelli. 2012. Hardware and software synthesis of heterogeneous systems from dataflow programs. *Journal of Electrical and Computer Engineering*, 2012:2.
- [7] Actors-project.eu. n.d.. *ACTORS - Adaptivity and Control of Resources in Embedded Systems*. [online] Available at: <http://www.actors-project.eu/> [Accessed: 3 Aug 2013].
- [8] Jörn W Janneck. 2011. A machine model for dataflow actors and its applications. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 756–760. IEEE.
- [9] Southampton.ac.uk. 2013. *Raspberry Pi at Southampton*. [online] Available at: <http://www.southampton.ac.uk/~sjc/raspberrypi/> [Accessed: 3 Aug 2013].
- [10] Adapteva. 2010. *Adapteva*. [online] Available at: <http://www.adapteva.com/> [Accessed: 3 Aug 2013].

- [11] Epiphany SDK Reference
- [12] Epiphany Architecture Reference
- [13] En.wikipedia.org. 2009. *Barrier (computer science)* - *Wikipedia, the free encyclopedia*. [online] Available at: [http://en.wikipedia.org/wiki/Barrier\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Barrier_(computer_science)) [Accessed: 3 Aug 2013].
- [14] Sapir, Y. 2013. *Epiphany SDK 5 Released!*. [online] Available at: <http://www.adapteva.com/announcements/the-new-epiphany-sdk-5-release/> [Accessed: 3 Aug 2013].
- [15] En.wikipedia.org. 2004. *Strategy pattern* - *Wikipedia, the free encyclopedia*. [online] Available at: [http://en.wikipedia.org/wiki/Strategy\\_pattern](http://en.wikipedia.org/wiki/Strategy_pattern) [Accessed: 3 Aug 2013].
- [16] Openocd.sourceforge.net. 2013. *Open On-Chip Debugger*. [online] Available at: <http://openocd.sourceforge.net/> [Accessed: 3 Aug 2013].
- [17] Mecklenburg, R. and Oram, A. 2005. *Managing projects with GNU make*. Beijing: O'Reilly.
- [18] Www-cs-faculty.stanford.edu. 1977. *Knuth: Frequently Asked Questions*. [online] Available at: <http://www-cs-faculty.stanford.edu/~uno/faq.html> [Accessed: 3 Aug 2013].
- [19] En.wikipedia.org. 2012. *Translator (computing)* - *Wikipedia, the free encyclopedia*. [online] Available at: [http://en.wikipedia.org/wiki/Translation\\_\(computing\)](http://en.wikipedia.org/wiki/Translation_(computing)) [Accessed: 3 Aug 2013].
- [20] En.wikipedia.org. 2012. *Source-to-source compiler* - *Wikipedia, the free encyclopedia*. [online] Available at: [http://en.wikipedia.org/wiki/Source-to-source\\_compiler](http://en.wikipedia.org/wiki/Source-to-source_compiler) [Accessed: 3 Aug 2013].
- [21] Donald E Knuth. 1974. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301.
- [22] Coffeescript.org. n.d.. *CoffeeScript*. [online] Available at: <http://coffeescript.org/> [Accessed: 3 Aug 2013].
- [23] Feathers, M. 2004. *Working effectively with legacy code*. Upper Saddle River, N.J.: Prentice Hall PTR.

