Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

A source-to-source compiler for the PRAM language Fork to the REPLICA many-core architecture

by

Cheng Zhou

LIU-IDA/LITH-EX-A--12/042—SE

2012-08-29
A source-to-source compiler for the PRAM language Fork to the REPLICA many-core architecture

by

Cheng Zhou

LIU-IDA/LITH-EX-A--12/042—SE

2012-08-29

Supervisor: Erik Hansson
Examiner: Christoph Kessler
Abstract

This thesis describes the implementation of a source to source compiler that translates Fork language to REPLICA baseline language. The Fork language is a high-level programming language designed for the PRAM (Parallel Random Access Machine) model. The baseline language is a low-level parallel programming language for the REPLICA architecture which implements the PRAM computing model. To support the Fork language on REPLICA, a compiler that translates Fork to baseline is built. The Fork to baseline compiler is built in compatibility with the Fork implementation for SB-PRAM. Moreover, the libraries that support Fork’s features are built using baseline language. The evaluation result verifies that the features of the Fork language are supported in the implementation. The evaluation also shows the scalability of our implementation and shows that the overhead introduced by Fork-to-baseline translation is small.
Acknowledgements

I would like to thank my supervisor Erik Hansson for many advices during my thesis project and immediate help when I encountered problems about the REPLICA assembly and compiler. I also would like to thank my examiner Christoph Kessler for the careful review of my thesis and many suggestions during my thesis project. I also want to thank Martti Forsell, the project leader of the REPLICA project, for the careful review and suggestions on my thesis.
Contents

List of Tables 1
List of Figures 2

1 Introduction 3
  1.1 Contributions ................................................. 3
  1.2 Thesis Outline ............................................... 4

2 Background 5
  2.1 PRAM Model .................................................. 5
  2.2 REPLICA Project .............................................. 7
    2.2.1 REPLICA architecture .................................... 7
    2.2.2 E-Language ............................................... 7
    2.2.3 Baseline language of REPLICA ......................... 8
  2.3 Fork language ................................................ 9
  2.4 Compiler ..................................................... 10
    2.4.1 Source to Source Compiler ............................ 11
  2.5 Problem Statement .......................................... 12
    2.5.1 Choices of Compiler Framework .................... 13
    2.5.2 Goal ....................................................... 14

3 Fork to baseline translation 15
  3.1 Features ..................................................... 15
  3.2 Shared and Private Variables ........................... 16
  3.3 System Variables ........................................... 17
  3.4 Multiprefix Operations .................................... 18
  3.5 Synchronous and Asynchronous Regions .................. 19
  3.6 Synchronicity of Functions ................................ 20
  3.7 Thread Groups ............................................... 21
  3.8 Pointers ...................................................... 25
  3.9 Locks .......................................................... 26
  3.10 Heap .......................................................... 26
  3.11 Fork Statement .............................................. 26
4 Implementation 29
4.1 Overview ........................................ 29
4.2 Cetus ........................................ 30
4.3 Reserved Words in Fork translator ............ 30
4.4 ANTLR Grammar ................................ 31
4.5 Fork IR ......................................... 33
4.6 Fork IR to baseline IR translation ............. 33
4.7 Implementation of the Fork Library ........... 37
  4.7.1 Synchronization ................................ 37
  4.7.2 Locks ....................................... 38
4.8 Memory Management ............................. 39
  4.8.1 Stack ....................................... 39
  4.8.2 Heap ....................................... 40
  4.8.3 Frames ..................................... 42
  4.8.4 Group Splitting .............................. 44
4.9 Jumps ........................................... 45
5 Measurement and Results 47
5.1 Programs in the Fork implementation for SB-PRAM 47
  5.1.1 Porting Fork Programs ........................ 48
  5.1.2 Testing using Fork Programs ............... 48
5.2 Test of scalability of Fork programs on REPLICA 50
  5.2.1 Quicksort .................................... 52
  5.2.2 Mergesort ................................... 54
5.3 Reimplementation of existing baseline programs 56
  5.3.1 Threshold .................................... 56
  5.3.2 Blur ........................................ 58
  5.3.3 Edge ........................................ 60
5.4 Test compilation time ................................ 61
5.5 Summary ........................................ 64
6 Related Work 65
6.1 Languages ........................................ 65
  6.1.1 E-language .................................... 65
  6.1.2 XMTC ....................................... 66
  6.1.3 UPC ......................................... 66
  6.1.4 OpenMP ..................................... 66
  6.1.5 NestStep ..................................... 66
6.2 Compiler and Run-time Libraries for PRAMs .... 67
  6.2.1 FCC .......................................... 67
  6.2.2 E-language libraries .......................... 67
  6.2.3 XMTC Compiler ................................ 67
6.3 Source-to-source compilers ........................ 68
  6.3.1 Fork to CUDA ................................ 68
  6.3.2 NestStep to C compiler ........................ 68
  6.3.3 OpenMP to POSIX threads compiler ........ 68
| 6.3.4 OpenMP to GPGPU compiler   | 69       |
| 6.3.5 G-ADAPT: Adaptive GPU optimization | 69       |
| 6.3.6 hiCUDA                      | 69       |
| 6.3.7 OpenMP to MPI compiler      | 70       |
| 6.3.8 Sequential C++ to Parallelized C++ compiler | 70       |

7 Conclusion, Limitations and Future Work

7.1 Conclusion              71
7.2 Limitations             71
7.3 Future Work             72

8 Bibliography

A Installation

B Implementation Details

B.1 Compiling Flags          82
B.2 Supporting Libraries     82
   B.2.1 Shared Local Variable Allocation 82
   B.2.2 Synchronization         83
   B.2.3 INIT FORK_ENV           84

C Test Programs

C.1 Threshold               85
C.2 Blur                   87
C.3 Edge                  91
C.4 Algorithms in Fork language 94
   C.4.1 Mergesort            94
   C.4.2 Quicksort            103
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>e-language control constructs example</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Features of Fork language</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>System Variables in Fork language and baseline language</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Multiprefix Operations in Fork language and baseline language</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Rules for calling function in each execution mode in Fork</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>Constructs for group creation and group splitting in the baseline language</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Library header files of Fork</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>API for Dynamic Memory Management of Fork</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Tested Features using Fork programs</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution Cycles of quicksort on SB-PRAM simulator</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Execution Cycles of quicksort using 2048 threads on REPLICA-T5-I-512+</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Execution Cycles of mergesort using 2048 threads on REPLICA-T5-I-512+</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison of Execution Cycles of threshold programs in different implementation</td>
<td>58</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of number of lines of the threshold program in different languages</td>
<td>58</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparison of Execution Cycles of blur programs in different implementation</td>
<td>59</td>
</tr>
<tr>
<td>5.8</td>
<td>Comparison of number of lines of the blur program in different languages</td>
<td>59</td>
</tr>
<tr>
<td>5.9</td>
<td>Comparison of Execution Cycles of edge programs in different implementation</td>
<td>61</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison of number of lines of the edge program in different languages</td>
<td>61</td>
</tr>
<tr>
<td>5.11</td>
<td>Compilation time of Fork programs using Fork-to-baseline compiler</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Classification of some parallel programming languages</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A p-processor PRAM (figure from [28]) .......................... 5
4.1 Compilation process of the Fork to baseline compiler .......... 29
4.2 Part of the Fork IR of the example program ................. 35
4.3 Translated baseline IR ............................................. 37
4.4 Heap organization in baseline language to support Fork .... 40
4.5 Memory organization before and after a Fork statement in a thread group ............................................. 44
5.1 Execution Cycles of quicksort on SB-PRAM simulator .... 52
5.2 Execution Cycles of quicksort on REPLICA-T5-4-512+ .... 54
5.3 Execution Cycles of quicksort and mergesort on different datasets on REPLICA-T5-4-512+ ................................. 55
5.4 Comparison of execution cycles of different threshold.c implementations ................................................. 57
5.5 Comparison of execution cycle counts of different blur.c implementations .................................................. 59
5.6 Comparison of execution cycle counts of different edge.c implementations .................................................. 60
5.7 Compilation time of Fork programs using our Fork-to-baseline compiler .................................................... 63
Chapter 1

Introduction

This thesis is part of the project REPLICA (REmoving Performance and programmability Limitations of Chip multiprocessor Architectures) \cite{16, 20}. The REPLICA architecture is a Configurable Emulated Shared Memory Machine (CESM) architecture. It implements a PRAM-NUMA (Parallel Random Access Machine - Non-Uniform Memory Access) model which differs from existing popular Chip Multi-Processor architectures. The NUMA mode of the REPLICA architecture is not considered in this thesis.

In PRAM mode, there are thousands of threads that work synchronously and share a big memory. In REPLICA, the big shared memory is emulated by memory modules distributed across the processors. From the software programmer’s perspective, it is the same as if there were a big shared memory.

To utilize this computing power in REPLICA, a baseline programming language \cite{43} is available. However, the baseline language is a low-level language where the software programmer needs to take care of all the details in order to manage these threads.

Fork is a language that is designed for the PRAM computing model. It uses high-level constructs to manage the threads. In this thesis, we show that it is suitable to use the Fork language in REPLICA by implementing a Fork-to-baseline language compiler.

1.1 Contributions

The contributions of this thesis are as follows.

- A Fork-to-baseline compiler is constructed.
- The baseline libraries that support the Fork language in REPLICA are implemented.
Chapter 1. Introduction

- The evaluation result verifies the scalability of our implementation. The compilation time of the Fork-to-baseline compiler is tested. The overhead of our Fork implementation on REPLICA is also evaluated.

1.2 Thesis Outline

- In Chapter 2, background information for this thesis is presented, which includes the PRAM model, REPLICA, the Fork language, the e-language, the baseline language and compiler technology. Then the problem statement and design choices are discussed.

- In Chapter 3, the features of the Fork language are analyzed in detail and the corresponding translations from the Fork language to baseline language are discussed.

- In Chapter 4, the implementation of the compiler and the implementation of supporting libraries are presented. The memory management functionality that supports group splitting and dynamic memory allocation is discussed.

- In Chapter 5, the implementation is evaluated from different perspectives. Firstly, several programs from the Fork implementation for SB-PRAM [31, 28] are used to verify the features of Fork that are implemented. Then the scalability of our implementation is evaluated. The overhead of the implementation is evaluated by comparing the number of execution cycles of a baseline program and its Fork program on REPLICA simulator. At last, the compilation time of the Fork-to-baseline compiler is tested.

- In Chapter 6, related work is presented.

- In Chapter 7, conclusion, limitations and suggestions for future extensions are presented.
Chapter 2

Background

2.1 PRAM Model

PRAM (Parallel Random Access Machine) is a parallel programming model [28]. It is the parallel version of the Random Access Machine (RAM) model. In PRAM, all the processors share a common memory, see Figure 2.1. Processors can access any memory address of the shared memory in one clock cycle. Therefore, the communication latency between processors is two clock cycles, since processors can communicate by accessing memory. In the PRAM model, the cost of synchronization between processors is ignored. In the implementation of the PRAM model such as SB-PRAM [31] [28] and REPLICA [16] [20], processors could work in synchronous mode in which processors share a common clock and each execute one instruction in one cycle.

![Figure 2.1: A p-processor PRAM (figure from [28])](image-url)
Since the memory latency is deterministic (one clock cycle) and synchronization of processors is implicit, the programmer does not need to take care of memory latency and synchronization between processors. Therefore it is easier to write correct parallel programs in the PRAM model than in most other programming models like MPI (Message Passing Interface) [23].

In the PRAM computing model, when multiple processors write to the same memory cell at the same time step, a write conflict occurs. There are several methods to resolve write conflicts. These methods can be classified into three categories [28].

- **EREW-PRAM (Exclusive Read and Exclusive Write):** only one processor can read or write the same memory cell at the same time step.
- **CREW-PRAM (Concurrent Read and Exclusive Write):** concurrent read is allowed but only one processor can write the same memory cell at the same time step. Since writing is exclusive, simultaneous reading and writing is not allowed.
- **CRCW-PRAM (Concurrent Read and Concurrent Write):** processors can either read or write concurrently at the same memory cell at the same time step. The result of concurrent write depends on the conflict resolution policy that is used. For instance:
  - Arbitrary CRCW-PRAM model: the value written is an arbitrary one among the values to be written in concurrent write.
  - Priority CRCW-PRAM model: the value written by the processor that has the highest priority in concurrent write is kept in the memory cell.
  - Multi-operation CRCW-PRAM model: Accumulative operations such as add, maximum, multiply, are applied to the values written by the processors in concurrent write.

Among these variants, the Multi-operation CRCW-PRAM is most powerful. For example, all the values concurrently written to one memory cell at the same time step can be summed and stored in that memory cell. With this parallel computing architecture, the time complexity of summing an array with \( n \) elements is no longer \( \Theta(\log n) \) but one clock cycle if there are enough threads, since all the threads could write to the same address and get the sum in the next clock cycle. Therefore, writing software programs is easier because the workload of concurrent write is handled by hardware.

Although the PRAM model is powerful, it was considered difficult to be realized in hardware, since it requires many processors to access a large shared memory concurrently and the latency of any memory access is one clock cycle, despite successful early attempts such as SB-PRAM [31, 28]. However, the implementation of a PRAM model architecture becomes more and more possible with the development of technology. In recent years, there are several research projects that implement PRAM style architectures, for
2.2 REPLICA Project

The REPLICA project is a project to ease the programmability for parallel programming by using a strong parallel programming model (PRAM) and realizing it in hardware.

2.2.1 REPLICA architecture

The REPLICA architecture, which derives from the Total Eclipse project, is a Configurable Emulated Shared Memory Machine (CESM) architecture. The REPLICA architecture consists of several multithreaded processors. Each processor has its local memory. All the memories are connected by a network so that they form a large shared memory that can be accessed in one step during which each PRAM processor executes a single instruction. The number of threads is determined by the configuration. All the threads start to run after boot-up of the system.

The threads of the REPLICA architecture can be configured to execute programs either in PRAM mode or NUMA mode at run-time. In PRAM mode, it supports Multi-operation CRCW-PRAM (MCRCW-PRAM). In NUMA mode, each processor can access its local memory with shorter latency than the latency of accessing remote memory. In this thesis, only PRAM mode is considered.

2.2.2 E-Language

The e-language is the language designed for the project Total Eclipse prior to the project REPLICA. It is designed for utilizing Thread Level Parallelism (TLP) on shared memory architectures that implement the PRAM model. The syntax of the e-language is based on the C language with extensions supporting these new features. Since the e-language shares several features with both baseline language and Fork, it influences the implementation in the thesis to support Fork on REPLICA.

- Private/Shared variables
  
  Shared variables are shared by the threads in one group. Private variables are private to a thread.

- Synchronous/Asynchronous Area
  
  In a synchronous area, threads in the same group execute the same code synchronously. In an asynchronous area, threads are not synchronized until the end of the area.

1In the following, threads mean hardware threads and implement PRAM processors.
Some of the control constructs [18] that lead to synchronous or asynchronous areas are listed below in order to show a detail of the e-language.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Calling Area</th>
<th>Create subgroups</th>
<th>Synchronize at the end</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (c) s;</td>
<td>Both</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>if (c,s);</td>
<td>Both</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td>if (c,s);</td>
<td>Synchronous</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>if (c,s);</td>
<td>Synchronous</td>
<td>1</td>
<td>yes</td>
</tr>
</tbody>
</table>

- Thread group hierarchy
  A thread group can be divided into subgroups when the program enters certain control constructs. When creating a subgroup, variables such as _thread_id are saved and updated. When a subgroup ends, the old values are restored.

2.2.3 Baseline language of REPLICA

In the REPLICA project, the baseline language is a low-level C-style programming language with e/Fork-style parallelism [43, 47]. The compiler for the baseline language is built based on LLVM compiler framework[5]. In previous work [47], it is implemented and tested.

The baseline language includes the concept of threads, shared/private variables, and multiprefix operations. It includes inline assembly as a method to use multiprefix instructions. The names of shared variables in baseline language have the symbol ‘_’ as postfix. Some functions are implemented as libraries for the baseline language. For instance, the synchronization of threads is defined as a library function in the baseline language.

In Listing 2.1, a simple baseline language program from [47] is presented. This example program compute the sum of the array into the shared variable sum by the multi-operation MADD.
#include "replica.h"
#define SIZE 8096;
int array_[SIZE]; /* private array with SIZE entries */
int sum_= 0; /* shared variable */
int main ()
{
    unsigned int i :
    for ( i = _thread_id ; i < SIZE ; i += _number_of_threads )
    {
        asm ( "MADD0 %0,%1" : /* no output */
             : "r" (array_[i]) , "r" (&sum_)
             : ) ;
        _synchronize ; /* Wait for all threads */
        _exit ; /* Issue an exit trap to halt the program */
        return 0 ;
    }
}

Listing 2.1: Simple baseline language program

The memory of a baseline language program is organized into three types: program memory space, shared memory space and thread private memory space [47].

The details of the baseline language are described in Chapter 3.

2.3 Fork language

The Fork language [28] is a parallel programming language designed for the MCRCW-PRAM model. It is a SPMD (Single Program Multiple Data) style programming language. Therefore, each processor executes the same program but works on different data. The sequential semantics of the Fork language is based on the C language. For parallel computing, the Fork language includes new keywords and system variables. For example, Fork uses the keyword 'sh' to indicate shared variables and the keyword 'pr' to indicate private variables.

The Fork language has the concept of (hardware) threads and thread groups. Each processor runs one thread. However, the Fork language is not a fork/join style parallel programming language. All the threads start to run at the beginning of the program. The thread group may be split or be unified according to the control flow of the program.

The Fork language defines synchronous and asynchronous program regions. In a synchronous region, threads of the same group execute the same instruction at the same time step. In an asynchronous region, there is no such constraint.

The Fork language supports multiprefix operations which utilize the powerfulness of the MCRCW-PRAM model.

The Fork language supports pointers which are as flexible as in the C language.
The heap memory in Fork is classified into three types: a private heap for each processor, an automatic shared heap for each group and a global, permanent shared heap.

In Listing 2.2, a simple example [28] is presented to illustrate the basics of the Fork language.

```c
#include <fork.h>
#define N 30;
sh int sq[N]; /* shared variable */
sh int p = _STARTED_PROCS_; /* system variable */
sh int sum=0;

void main (void)
{
  pr int i; /* private variable */

  /* synchronous regions*/
  start {
    /* multiprefix operation */
    i=mpadd(&sum, $);

    /* asynchronous regions*/
    farm {
      if ($ < N) {
        sq[$]=i;
      }
    }
  }
}
```

Listing 2.2: Simple Fork language program

The details of the Fork language are illustrated in Chapter 3.

2.4 Compiler

A compiler is a translator that translates a program in one language to another language. For example, the C compiler translates a program written in C language into a program in assembly language.

The compiling process can be divided in two parts: analysis part and synthesis part [11]. In the analysis part, the source program is analyzed for its syntactical and semantic correctness and transformed into intermediate representation (IR). In the synthesis part, the IR is used to do optimizations and generate the target program. The analysis part is also called the front end of the compiler, while the synthesis part is called the back end of the compiler.

The compilation process can also be divided into several phases [11]. The input of each phase is the output of the previous phase. The initial input is the text of a program that is to be compiled. The internal output of each phase is a form of Intermediate Representation (IR) such as Abstract Syntax...
2.4. Compiler

Tree (AST). The final output is the translated program that is written in another language.

Usually, the compiling process is divided into the following phases. A brief introduction about the functionality of each phase is given below. For a detailed introduction, please refer to some text book [11].

- **lexical analysis**
  The input of the lexical analysis is the stream of characters of a program written in the source language. The output is a stream of tokens.

- **syntax analysis**
  The input of syntax analysis is the stream of tokens. The output is the IR of the program in the form of e.g. an AST.

- **semantic analysis**
  The semantic analysis phase checks the semantic correctness regarding to the language definition by analyzing the IR.

- **intermediate code generation**
  After semantic analysis, a compiler usually generates a lower level IR of the program from the higher level IR such as AST. There could be several layers of IR, such as high-level IR, middle-level IR and low-level IR.

- **code optimization**
  The optimization phase performs different kinds of optimizations at the suitable layer of IR.

- **code generation**
  In the code generation phase, the input is the IR. The output is the code that is generated from the IR using the target language.

In the REPLICA project, the initial version of the compiler for the baseline language is accomplished in previous work [47]. The back end of the compiler is implemented in LLVM and the front end is implemented in Clang [2].

### 2.4.1 Source to Source Compiler

A source to source compiler refers to translating a programming language to another programming language that are both at approximately the same abstraction level, for example, OpenMP to GPGPU [37], unoptimized C++ to optimized C++ [18], C++ to CUDA (Compute Unified Device Architecture) [22, 3], etc. The traditional compiler usually translates from a high level programming language to a low level programming language that are at different abstraction levels, for example, from C/C++ to assembly language.
A main difference between a traditional compiler and a source to source compiler is the IR. In a traditional compiler, the higher level IR is transformed down to probably several lower level IRs until it is suitable to generate the target code. In a source to source compiler, since both the source language and target language are at the same abstraction level, the difficulty is to find the best level of IR that could be used to translate from source language to target language. Using the lowest level IR or the assembly language style IR, is possible for source to source translation. However, the generated target program is likely to look very different from the source program and to be difficult to understand.

2.5 Problem Statement

In the REPLICA project, the Fork language is to be supported on the REPLICA platform. In previous work, a first version of a compiler for the baseline language is already implemented. Therefore, there are several alternative methods to accomplish this task.

The first method is to build a front end for the Fork language in LLVM. In this method, the Fork language is translated to the IR of LLVM and then translated to REPLICA assembly language by the baseline back-end compiler implemented in LLVM. Therefore, the solution is integrated very well because the result is a Fork compiler in LLVM for REPLICA. The challenge in this method is that the intermediate representation of LLVM may have to be significantly modified to support the Fork language. Moreover, since the IR of LLVM is low level, it could be difficult to implement and debug.

The second method is to write a source to source compiler to translate Fork language into baseline language. Then the compiler for baseline language is used to compile the program into an assembly language program. Since the Fork language and the baseline language are both based on the C language, the source to source translation seems easier than translating the Fork language to the LLVM IR. On the other hand, the compilation process costs more than the first method because the program needs to go through both Fork compiler and baseline compiler which both consist of front-end and back-end.

Considering that the existing software in baseline language is limited and the time to finish is also under constraint, the second method is chosen to support Fork in REPLICA because this method divides the implementation into different layers and therefore reduces the effort of implementation, and is easier to debug.

Therefore, in this thesis, a source to source compiler that translates Fork to baseline language will be built. The input program written in Fork language will be translated into baseline language first, then be translated into the assembly language of the REPLICA architecture by the baseline language compiler. Several libraries in baseline language are also needed to support Fork on REPLICA.
Manually constructing a compiler is a time-consuming and nontrivial work. Instead, many compiler frameworks are available to be used as the base. To choose a suitable compiler framework, the IR of the compiler framework is considered as an important issue. If the IR is similar to both Fork and baseline language, it would be easier to do the translation.

In the following section, several compiler frameworks that can be used for source to source compilation are discussed.

2.5.1 Choices of Compiler Framework

- **LLVM**

  LLVM \[5\] is a compiler framework that is widely used. Its front-end for the C language is Clang, which is used to build the front-end of the baseline language compiler. The IR of LLVM is a low level, powerful representation which is suitable for many optimizations.

  To use LLVM as a source to source compiler for Fork and baseline language, the front-end, Clang, needs to be modified to support the syntax of Fork. Then a baseline back-end needs to be added so that the output program is in baseline language. Considering the similarity of the baseline language to the C language, it is possible to use the existing C back-end to output target code for the baseline language. Therefore, in this method, the major work is to build a Fork language compiler front-end.

  The challenges in this method are the low-level IR of LLVM and the big code base of LLVM, which both increase the difficulty of implementation.

- **Cetus**

  Cetus \[38\] is a compiler framework for source to source translation of C-style languages. The intermediate representation of Cetus is very similar to the C language. Cetus uses the ANTLR \[1, 45\] parser generator to generate a parser for the compiler.

  Among the research works done based on Cetus, there are translations between OpenMP and the Pthreads API \[25\], OpenMP to GPGPU (General-Purpose computation on Graphics Processing Units) \[37\], and compiling the parallel programming language NestStep to the CELL processor \[24\]. Therefore, it seems that building a compiler to translate Fork language programs to baseline language programs is promising in Cetus. Moreover, the code base of Cetus is very small compared to other alternative compiler frameworks, which is an advantage for implementation.

- **Open64**

  Open64 \[6\] is a compiler that compiles C/C++ and Fortran languages for x86, x86-64, Itanium and some other platforms. Open64’s IR
(WHIRL) has five layers. Open64 is used as a source to source compiler in [39] where a cross-platform OpenMP compiler is built to translate OpenMP programs to C and Fortran programs.

- **Rose**
  
  Rose [8] is a compiler framework that is dedicated to source to source compiling. It supports C/C++, Fortran and OpenMP. It has a high level AST IR with a rich interface for manipulation. Its IR could be printed as a PDF or DOT file to support debugging. Previous work has been done to translate OpenMP to different libraries and to automatically parallelize sequential C++ code [40].

Although each of the above compiler frameworks could be applied in the implementation, Cetus is chosen to implement the Fork to baseline language compiler. This is because Cetus’ IR is similar to both Fork and baseline. Moreover, Cetus has a smaller code base which could be beneficial for implementation and debugging.

### 2.5.2 Goal

The overall goals in this thesis are the following:

- Implement a Fork language to baseline language translator in Cetus.
- Implement necessary libraries for both Fork language and baseline language.
- Test and evaluate the solution.
Chapter 3

Fork to baseline translation

There are many features in the Fork language. These features of the Fork language are to be translated into the baseline language. Since some features in the Fork language are easier to be implemented in the solution and some are difficult, these features are classified into two groups: basic level and advanced level (see Table 3.1).

According to the features defined in Chapter 5 of the book Practical PRAM Programming [28], these features are classified in Table 3.1.

This chapter is written from the point of view of a Fork-to-baseline compiler, since a source-to-source compiler needs to know the detailed translation from each feature of the source language to the target language. Some translations are straight forward, for example the translation of system variables. Some translations such as group splitting and memory management are determined by the REPLICA architecture and baseline language, which are discussed in Chapter 4.

3.1 Features

In the initial approach, the basic features (in Table 3.1) of the Fork language are implemented first, then the advanced features are implemented as many as possible. In this thesis project, all the basic features and most of the advanced features have been implemented.

In this chapter, each feature of the Fork language in Table 3.1 is illustrated. The translation of each feature from Fork language to baseline language is presented.
### Table 3.1: Features of Fork language

<table>
<thead>
<tr>
<th>No.</th>
<th>Features</th>
<th>Level</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shared and private variables</td>
<td>Basic</td>
<td>rename</td>
</tr>
<tr>
<td>2</td>
<td>system variables</td>
<td>Basic</td>
<td>one to one mapping</td>
</tr>
<tr>
<td>3</td>
<td>multiprefix operations</td>
<td>Basic</td>
<td>recognized as operator in Fork compiler, implemented as library functions in baseline</td>
</tr>
<tr>
<td>4</td>
<td>synchronous and asynchronous regions (farm, seq, start)</td>
<td>Basic</td>
<td>translate to baseline statement, implement group_sync()</td>
</tr>
<tr>
<td>5</td>
<td>synchronicity of functions</td>
<td>Adv.</td>
<td>check synchronicity</td>
</tr>
<tr>
<td>6</td>
<td>thread groups</td>
<td>Adv.</td>
<td>support Group Frames for group splitting</td>
</tr>
<tr>
<td>7</td>
<td>pointers</td>
<td>Basic</td>
<td>used to implement shared local variables</td>
</tr>
<tr>
<td>8</td>
<td>heap</td>
<td>Adv.</td>
<td>implement parallel malloc() on shared memory</td>
</tr>
<tr>
<td>9</td>
<td>locks and semaphores</td>
<td>Adv.</td>
<td>simple lock, fair lock</td>
</tr>
<tr>
<td>10</td>
<td>fork</td>
<td>Adv.</td>
<td>support fork() statement</td>
</tr>
</tbody>
</table>

### 3.2 Shared and Private Variables

In the Fork language, the keyword ‘sh’ is used to indicate a shared variable and the optional keyword ‘pr’ is used to indicate a private variable.

```fork
sh int v; /* shared variable */
pr int w; /* private variable */
```

Listing 3.1: Shared and Private variable in Fork

In baseline language, the name of a shared variable ends with a symbol ‘.’.

```baseline
int v.; /* shared variable */
int w;    /* private variable */
```

Listing 3.2: Shared and Private variable in baseline language

Therefore, the shared variables are translated to baseline shared variables by the above renaming. However, this translation is correct only for global shared variables since the baseline language does not support local shared variables. To support local shared variables, private pointers are used instead. Please refer to Section 3.8 for the detailed implementation.
3.3 System Variables

In Fork, there are several system variables. In the baseline language, the built-in variables have the same meaning as the system variables in Fork language. The names of these built-in variables start with a ‘.’.

<table>
<thead>
<tr>
<th>Fork</th>
<th>Baseline</th>
<th>meaning</th>
</tr>
</thead>
</table>
| _STARTED_PROCS_ | _absolute_number_of_threads_              | number of processors/threads
| _PROC_NR_     | _absolute_thread_id_                     | processor/thread ID                      |
| @             | _group_no_                               | Group ID                                  |
| $             | _thread_grid_                            | Group-relative processor ID               |
| $$            | _thread_id_                              | Group rank                                |
| #             | _number_of_threads_                      | the number of threads within one group    |

*aIn the Fork language, each processor executes one thread.*
### 3.4 Multiprefix Operations

In REPLICA assembly, multiprefix instructions are supported, see Listing 3.3 for an example.

```
MPADD0 R2, R1
```

Listing 3.3: A multiprefix instruction in REPLICA assembly

When the above instruction is executed by \(N\) threads in parallel, threads add the value of their register R2 to the value in the memory address provided by R1 automatically in rank order. The value of the prefix-sum is stored in register M0. If the computation of the prefix-sum depends on the thread id, then the prefix-sum of thread \(i\) is the sum of the value in R2 from thread 0 to thread \(i - 1\). Therefore, each thread gets the prefix-sum result in M0 and the total sum result is stored in memory address R1. However, there is no fixed ordering of multiprefix operations in REPLICA. In our implementation, the multiprefix operations in Fork is translated directly to multiprefix operation instructions in REPLICA since many Fork programs do not depend on the fixed order.

In Fork language, the multiprefix operations are recognized as operators, see Listing 3.4. Their priority is the same as Unary Operators, higher than Binary Operators.

```
oid=(-((unsigned int)mpadd(( & p), 1)));
```

Listing 3.4: Example code using a multiprefix operator in Fork

While in baseline language, the multiprefix instructions are implemented as library functions using inline assembly, see Listing 3.5.

```
int replica_mpadd(int a, int *sum)
{
    int pre;
    asm("MPADD0 %0, %1
         ST0 M0,%2": "r"(a),"r"(sum),"r"(&pre));
    return pre;
}
```

Listing 3.5: A multiprefix function in baseline language

The supported Multiprefix Operations in Fork and baseline language are listed in Table 3.3.

---

1. The prefix-sum on REPLICA is actually not depend on the thread id, see [42].
2. This may change in the future. The fixed ordering multiprefix operations could be implemented by software.
### 3.5 Synchronous and Asynchronous Regions

#### farm

The farm statement specifies that the following block is executed in asynchronous mode, see Listing 3.6. At the end of the block, a barrier is used to ensure that the program goes back to synchronous execution mode.

```plaintext
farm {
...
}
```

Listing 3.6: The farm statement in Fork language

In baseline language, the farm statement is translated as in Listing 3.7.

```plaintext
{...
synchronize();
}
```

Listing 3.7: The farm statement in baseline language

#### seq

The seq statement specifies that the following block is executed in asynchronous mode by only one thread, see Listing 3.8. At the end of the block, a barrier is used to ensure that the program goes back to synchronous execution mode.

```plaintext
seq {
...
}
```

Listing 3.8: The seq statement in Fork language

In baseline language, the seq statement is translated as in Listing 3.9.

```plaintext
if (_thread_id == 0)
{
...
}
```

Listing 3.9: The seq statement in baseline language

---

Table 3.3: Multiprefix Operations in Fork language and baseline language

<table>
<thead>
<tr>
<th>Fork</th>
<th>Baseline assembly instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpadd()</td>
<td>MPADD</td>
<td>multiprefix add</td>
</tr>
<tr>
<td>N/A</td>
<td>MPSUB</td>
<td>multiprefix sub</td>
</tr>
<tr>
<td>mpmx()</td>
<td>MPMAX</td>
<td>multiprefix max</td>
</tr>
<tr>
<td>N/A</td>
<td>MPMIN</td>
<td>multiprefix min</td>
</tr>
<tr>
<td>mpor()</td>
<td>MPOR</td>
<td>multiprefix or</td>
</tr>
</tbody>
</table>
Chapter 3. Fork to baseline translation

start

The start statement specifies that the following block is executed in synchronous mode, see Listing 3.10. The Group Relative ID ($) is updated in the start statement.

Listing 3.10: The start statement in Fork language

In baseline language, the start statement is translated as in Listing 3.11. The UPDATE GRID statement is used to update $. The RESTORE GRID is used to restore $ before exiting the start statement.

Listing 3.11: The start statement in baseline language

3.6 Synchronicity of Functions

Functions can be executed in synchronous or asynchronous mode. In Fork, the default execution mode of a function is asynchronous. The main function is in straight mode which is neither asynchronous nor synchronous.

New keywords "async", "sync" and "straight" as type qualifiers in function definition and declaration specify the execution mode of each function. In asynchronous functions, synchronous functions must be called in a start statement. In synchronous functions, asynchronous functions must be called in a farm or seq statement. Straight functions can be called in all execution modes.

The rules of calling function in each execution mode are specified in the following table.
3.7 Thread Groups

Initially, all the threads are in one root group. In synchronous mode, threads are split into subgroups when executing if statement, for loop, while loop and do loop, which includes private conditions. If the condition of these statements is not based on private variables, then all the threads evaluate the same value. Therefore there is no group splitting in non-private condition statements. In asynchronous execution mode, there is no group splitting.

In group creation and group splitting, the old group frame is saved and new group frames are created. The constructs used for group creation and group splitting are shown in Table 3.5.

Table 3.5: Constructs for group creation and group splitting in the baseline language

<table>
<thead>
<tr>
<th>Constructs</th>
<th>Number of subgroups</th>
<th>Used in Fork statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_GROUP</td>
<td>1</td>
<td>if (one-sided)</td>
</tr>
<tr>
<td>BEGIN_GROUP_NO_SYNC</td>
<td>1</td>
<td>for/while/do-while, function call</td>
</tr>
<tr>
<td>BEGIN_GROUP_A</td>
<td>2</td>
<td>if-else</td>
</tr>
<tr>
<td>BEGIN_GROUP_B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW_FORK_GROUPS(n,i)</td>
<td>n</td>
<td>fork</td>
</tr>
</tbody>
</table>

These constructs are similar to each other but each construct is slightly different in its functionality.

For example, the BEGIN_GROUP statement does not need to divide the heap memory comparing to other constructs. The BEGIN_GROUP_NO_SYNC statement does not need to update _thread_id and _number_of_threads comparing to other constructs.

To get a closer look, the implementation of the BEGIN_GROUP statement is presented in Listing 3.12. For detailed information about group frames, please refer to Section 4.8.3.

```c
#define NEW_SINGLE_GROUP SAVE_GFP_FRAME \    
   NEW_GFS_FRAME
#define BEGIN_GROUP NEW_SINGLE_GROUP \      
```
Listing 3.12: The if statement in Fork language

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_update_sub_sync;</td>
<td></td>
</tr>
<tr>
<td>#define END_GROUP</td>
<td>RESTORE_GFP_FRAME</td>
</tr>
</tbody>
</table>

Listing 3.12: The if statement in Fork language
### If statement

```java
if (private condition) {
    ... 
}
```

Listing 3.13: The if statement in Fork language

In baseline language, the if statement is translated in Listing 3.14.

```java
if (private condition)
{
    BEGIN_GROUP;
    ...
    END_GROUP;
} _synchronize();
```

Listing 3.14: The if statement in baseline language

### If-else statement

```java
if (private condition) {
    ...
} else {
    ...
}
```

Listing 3.15: The if-else statement in Fork language

In baseline language, the if-else statement is translated in Listing 3.16.

```java
if (private condition)
{
    BEGIN_GROUP_A;
    ...
    END_GROUP_A;
} else {
    BEGIN_GROUP_B;
    ...
    END_GROUP_B;
} _synchronize();
```

Listing 3.16: The if-else statement in baseline language
do-while statement

```
do {
  ...
} while (private condition);
```

Listing 3.17: The do-while statement in Fork language

In baseline language, the do-while statement is translated in Listing 3.18. The `update_sub_sync` statement is used to update `thread_id` and `number_of_threads` in each iteration in case some thread quits the loop body.

```
BEGIN_GROUP_no_sync;
do {
  update_sub_sync;
  {
    ...
  }
}while(private condition);
END_GROUP;
synchronize();
```

Listing 3.18: The do-while statement in baseline language

while statement

```
while (private condition) {
  ...
}
```

Listing 3.19: The while statement in Fork language

In baseline language, the while statement is translated in Listing 3.20.

```
BEGIN_GROUP_no_sync;
while() {
  update_sub_sync;
  {
    ...
  }
} END_GROUP;
synchronize();
```

Listing 3.20: The while statement in baseline language
3.8. Pointers

Pointers are supported in the baseline language. Since there is no concept of shared local variable in baseline language, pointers in baseline language are used to implement this feature for the Fork language. A shared local variable declared in the Fork language is replaced by a private pointer variable that points to the space allocated on its thread group’s shared_heap.

For example, in the Fork language, a shared local variable is declared in a block or function as in Listing 3.23.

```fork
foo () {
    sh int a;
    a = 1;
}
```

Listing 3.23: A shared local variable in the Fork language

In the baseline language, the shared local variable is translated in Listing 3.24.

```baseline
foo () {
    NEW_SH_LOCAL_VAR(int*, aa, sizeof(int));
    (* aa) = 1;
}
```

Listing 3.24: The shared local variable in the baseline language
Here, a private pointer is used to point to the shared variable allocated on the group’s shared heap. For the implementation details, please refer to Appendix B.2.1.

Function pointers are also supported. A function pointer can point to synchronous or asynchronous functions. In the Fork language, a function call to a synchronous function would create a single subgroup for this function call, and an implicit synchronization is followed. In our implementation, a new group is created for a normal function call to a synchronous function, but the group is not created for function calls by function pointers due to the rareness of this usage.

3.9 Locks

Locks are basic structures in parallel computing. When multiple threads are accessing shared data, locks are used to guarantee that the data is accessed in a correct way. In the Fork language, simple locks and fair locks are provided. In our implementation, locks are implemented as library functions in baseline language. Details about the implementation of locks in baseline language are given in Section 4.7.2.

3.10 Heap

Dynamic memory management is a basic functionality in the Fork language. The heap is divided into Global Shared Heap, Group Shared Heap and Private Heap. Since the support of Private Heap allocation in the baseline language is not available yet, only Global Shared Heap and Group Shared Heap are implemented in our solution.

The Global Shared Heap is shared by all the threads. Every thread can allocate a piece of heap by calling shmalloc(). The shmalloc() is a parallel version of the malloc() function implemented in the baseline language. It supports allocating heap memory in parallel. In parallel allocation, each thread gets one unique piece of heap. The thread that acquires the heap is responsible for releasing the memory by calling shfree();

The Group Shared Heap is shared by each group. It can be allocated by calling shalloc(). Unlike Global Shared Heap, all the threads get a pointer to one same piece of group heap in allocation. After group termination, the Group Shared Heap is freed automatically. Therefore, threads do not need to call any kind of free() function.

More details about the implementation of the heap are presented in Section 4.7.

3.11 Fork Statement

The Fork statement is a construct to divide threads into different subgroups.
The fork statement consists of three expressions.

\[ \text{fork(expr1; expr2; expr3) stmt} \]

Expr1 is the expression that specifies the number of groups.
Expr2 is the expression that specifies the subgroup number of each thread.
Expr3 is the expression that specifies the thread id of each thread in each group.

In the example of Listing 3.25 taken from quicksort in Appendix C.4.2, expr1 is 2, expr2 is \( (@=right) \), expr3 is \( ($=) \).

```text
farm if ($<numofprocesfor0) right = 0; else right = 1;
fork ( 2; @$=right; $=)
{
    qs( subarray[2*0], subn[2*0] );
}
Listing 3.25: The fork statement in Fork language
```

If expr2 is larger than or equal to expr1, the thread does not enter any group. The expr3 is optional. If expr3 is not given, _thread_id is computed by default.

This fork statement is translated to baseline language in Listing 3.26.

```text
{ /* fork( 2; _group_no=right;... ) */
    SAVE_GROUP_NO;
    _group_no=right;
    if ((_group_no<2))
    {
        NEW_FORK_GROUPS(2, _group_no);
        _thread_grid=_thread_grid;
        {
            qs(subarray[(2*group_no)], subn[(2*group_no)]);
        }
        END_FORK_GROUPS;
    }
    _synchronize();
    RESTORE_GROUP_NO;
}
Listing 3.26: The fork statement in baseline language
```

The NEW_FORK_GROUPS and END_FORK_GROUPS are used to setup the heap and stack for all the subgroups. Detailed explanation of group splitting in the fork statement is given in Section 4.8.3.
Chapter 4

Implementation

In this chapter, the implementation of the Fork to baseline language compiler is described firstly. Then the implementation of supporting libraries in baseline language is presented.

4.1 Overview

The compilation process of the Fork to baseline compiler is described in figure 4.1.

![Figure 4.1: Compilation process of the Fork to baseline compiler](image)
The Fork programs used for evaluation are either taken from the Fork distribution’s examples or re-written from baseline programs. The details about the Fork programs are given in Chapter 5 and Appendices.

The preprocessing step consists of two substeps. The first substep is to replace the special symbol `#` with the variable `_number_of_threads` in Fork by using a preprocessing script. The second substep is to replace the special symbol `@` with `group_no` and to insert an `#include` statement in the Fork program by the Cetus preprocessor.

The Fork header files include the file `fork.h` from the Fork implementation for SB-PRAM and other supporting library header files. The Fork-to-baseline compiler will check whether every function and macro is declared in these headers. If not, a warning is generated in compilation.

After preprocessing, the program is given to the parser of the Fork-to-baseline compiler. Then the IR tree of the Fork program is constructed. By transformation of the Fork IR tree to a baseline IR tree, the program is translated to the baseline program. The details of this process are described from Section 4.2 to Section 4.6.

Several features in Fork are implemented in baseline header files and baseline libraries. Synchronization is discussed in Section 4.7.1. Locks are discussed in Section 4.7.2. Memory management including dynamic memory management and group frames is discussed in Section 4.8. Group splitting is discussed in Section 4.8.4.

Finally, the baseline compiler translates the baseline program to a REPLICA assembly program which can be executed on the REPLICA simulator.

4.2 Cetus

Cetus is a compiler framework which is dedicated to source to source translation. It supports C-style language. In this thesis project, Cetus version 1.3 is used.

When Cetus is used as a source to source compiler, it parses the program and constructs an Intermediate Representation (IR) tree. After transforming the IR tree, Cetus outputs the IR tree to a program that is written in another language.

In this thesis, Cetus is extended to support the Fork language. New IR types are added into Cetus to support Fork and baseline language.

4.3 Reserved Words in Fork translator

The Fork language is based on the C language, therefore all the reserved keywords in C language are also reserved words for Fork. Moreover, since multiprefix operations are implemented as operators, see Section 3.3 `mpadd`, `mpmax`, `mpor`, `mpand` are reserved words. The keywords `fork`, `sync`, `async`, `straight`, `start`, `farm`, `seq` are also reserved words.
4.4 ANTLR Grammar

Cetus uses ANTLR \[\text{[1]}\] as its compiler front-end. ANTLR is a framework that is used to construct language recognizers. It is a parser generator that constructs LL(*) parsers. Therefore, left recursion is not allowed in the parser’s grammar.

In Cetus, ANTLR is used as the lexer and parser for Fork language. The ANTLR version 2.7.7 is used in Cetus version 3.0.

The existing grammar for C language in Cetus (NewCParser.g) is used as the base for Fork language, since Fork language is based on C language. Then the grammar of Fork language is added to support features of Fork.

- **Lexer for Fork**

  The input of a lexer (lexical analyzer) is a source program. The output of a lexer is a stream of tokens. The lexer for the Fork language has to recognize tokens of the Fork language. Since Fork language is based on C language, Fork’s tokens are almost the same as those of the C language. The only modification is to add the special symbol @ of Fork language into the Lexer. Then it is replaced with _group_no in the parser.

- **Parser for Fork**

  The input of a parser is a stream of tokens of a program. The output of a parser is the Intermediate Representation (IR) of the program. A parser is defined by a grammar of the language. The grammar specifies how tokens are to be organized in a syntactically valid program.

  Several new grammar rules are added into NewCParser.g to support the Fork language.

  The example in Listing 4.1 shows the extension to recognize multi-prefix operators. This code in NewCParser.g follows ANTLR’s rules of defining a parser.
Chapter 4. Implementation

//Fork multiprefix operations
multiPrefixOperator returns [MultiPrefixOperator code]
    {code = null;}
    
    "mpadd"
    {code = MultiPrefixOperator.MPADD;}
    
    "mpand"
    {code = MultiPrefixOperator.MPAND;}
    
    "mpor"
    {code = MultiPrefixOperator.MPOR;}
    
    "mpmax"
    {code = MultiPrefixOperator.MPMAX;}
    
Listing 4.1: Excerpt of the ANTLR grammar specification for the parser of the Fork language: multiprefix operators

The example in Listing 4.2 shows the extension to recognize a "seq" region as a SyncRegion statement. The SyncRegion statement represents start, farm, seq statements of Fork in our implementation. More detailed information are in Section 4.5.

statement returns [Statement statb]
    {
        Expression stmtb_expr;
        statb = null;
        Expression expr1=null, expr2=null, expr3=null;
        Statement stmt1=null,stmt2=null;
        int a=0;
        int sline = 0;
    } |
    /∗ Fork SyncRegion ∗/
    |
    /∗ SEQ ∗/
    |  tseq:"seq"  
    |
    |  sline = tseq.getLine();
    |
    |  putPragma(tseq,symtab);
    |
    |  stmt1=statement
    |
    |  statb = new SyncRegion(SyncRegion.SyncType.SEQ, stmt1);
    |  statb.setLineNumber(sline);
    |
Listing 4.2: Excerpt of the ANTLR grammar specification for the parser of the Fork language: seq statement
4.5 Fork IR

In Cetus, the constructed IR tree of a C program is similar to the nesting structure of the C statements in the program. To support the Fork language, new IR nodes are added into Cetus to represent new statements introduced by Fork. Each IR type is defined in one file.

- **MultiPrefixOperator.java**
  
  MultiPrefixOperator is the class for representing multiprefix operators presented in Section 3.4.

- **MultiPrefixExpression.java**
  
  MultiPrefixExpression is the class that represents a multiprefix operation expression in the program.

- **SyncRegion.java**
  
  SyncRegion represents the start, seq and farm statements discussed in Section 3.5 by using a different SyncType for each kind of the statements. The statements within seq or farm statement are actually in asynchronous mode. At the end of the seq or farm statement, the execution mode returns to synchronous mode again. These statements are represented in the SyncRegion IR node for simplicity.

- **ExecutionMode.java**
  
  ExecutionMode represents the synchronicity of a function defined by the keywords sync, async and straight.

- **ForkStatement.java**
  
  The ForkStatement class represents the fork statement in the Fork language.

A supporting class ForkBaselineLib.java is added in Cetus IR to provide the names of macros and functions implemented in baseline libraries.

4.6 Fork IR to baseline IR translation

After parsing a Fork program, the corresponding Fork IR tree is constructed. Then the Fork IR tree is transformed to a baseline IR tree. At last, the baseline IR tree yields the baseline program. The translation is divided into several steps.

**Step 1: Preprocessing**

Firstly, a new header file fork_replica.h is inserted into the program. Then the special symbols like ”@” and ”$” are replaced. The implementation is in the file Pre.g in Cetus, in which a simple preprocessor is defined using an ANTLR grammar.
Chapter 4. Implementation

After preprocessing, the Fork IR tree of the program is constructed.

**Step 2: Transform Fork IR to baseline IR**

This work is done in a transformation pass in the file `ForkBaseline-Trans.java`. It consists of several small functions each of which does a specific transformation on the IR of a Fork program.

- **transPredefinedVariable()**
  
  In this transformation, each system variable in Fork is replaced with its counterpart in baseline language according to Section 3.3. For example, `STARTED_PROCS` is replaced with `absolute_number_of_threads`.

- **transVariableDeclaration()**
  
  In this transformation, shared/private variables in Fork are transformed as presented in Section 3.2. Shared local variables are also transformed as described in Section 3.8.

- **transProcedureParameter()**
  
  The shared/private arguments defined in Fork functions are transformed to variables in baseline language.

- **transSynchronicityCheck()**
  
  The synchronicity check described in Section 3.6 is performed in this function.

- **transGroupSplit()**
  
  The transformations that are described in Section 3.5 and Section 3.7 are performed in this function. After this transformation, the group splitting statements in the Fork program are translated to the corresponding constructs in baseline language.

- **transForkEnvSetup()**
  
  The `INIT_FORK_ENV` statement is inserted at the entry of the main function, which reserves global heap memory and initializes group shared frames, see Appendix B.2.3. The function calls `synchronize()` and `exit()` are appended at the end of the main function and before any return statement in the main function.

- **transWorkaround()**
  
  In the last step, some workaround statements are added into the translated baseline program to avoid problems found in baseline compiler and simulator.

After the pass is accomplished, the Fork IR is translated into a baseline IR. In Listing 4.3 is an example that shows how the Fork IR of a Fork program is translated to baseline IR.

The example Fork program is called `mpadd_fork.c`.
4.6. Fork IR to baseline IR translation

Listing 4.3: The example Fork program mpadd_fork.c

```
#include <fork.h>

sh int a[4] = { 2, 1, 3, 1 };
sh int shmemloc;

void main(void) {
    int j;
    start if ($<4) {
        a[8] = mpadd( &shmemloc, a[8]);
    }
}
```

The Fork IR of the example is showed in Figure 4.2. In this picture, the IR nodes for header files and the wrapper CompoundStatement nodes within Procedure, SyncRegion, IfStatement and ExpressionStatement are ignored for simplicity.

![Diagram](image)

Figure 4.2: Part of the Fork IR of the example program

The translated baseline language program is shown in Listing 4.4.
Listing 4.4: The example baseline program translated from mpadd_fork.c

The statement INIT_FORK_ENV_NO_GLOBAL_HEAP is the same as INIT_FORK_ENV except that the global heap is reduced since global heap is not used in this program. This optimization is enabled by the compiling flag -Ofork-global-heap-shmalloc, see Appendix B.1. The statements SAVE_GRID and RESTORE_GRID are used to save and restore the value of group relative id (§).

The translated baseline IR is shown in Figure 4.3
4.7 Implementation of the Fork Library

Some functions need to be implemented in libraries for both Fork and baseline language. The previous implementation of the Fork library for the SB-PRAM system [28, 4] is referenced. The system library header files in the implementation are shown in Table 4.1.

Table 4.1: Library header files of Fork

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fork.h</td>
<td>types.h</td>
<td>multiprefix.h</td>
<td>sync.h</td>
</tr>
<tr>
<td>lock.h</td>
<td>string.h</td>
<td>io.h</td>
<td>stdlib.h</td>
</tr>
</tbody>
</table>

4.7.1 Synchronization

The existing synchronize() function from previous work [17] does not work correctly. On the REPLICA simulator, threads are not synchronized to the same step at the exit of the function. Therefore, a new synchronization function (group sync(), see Appendix B.2.2) is implemented in sync.c using REPLICA assembly. It synchronizes all the threads no matter in what order the threads enter this function. It supports synchronization within each group. Its implementation is based on the group shared variables _group_id.
and _group_barrier. At the exit of this function, all the threads within one group are synchronized to execute the same instruction at the same clock cycle.

The function group_sync() uses group shared variable _group_id and _group_barrier which must be allocated and initialized whenever a new group is created, see Section 4.8.3. At the beginning of a program, these two variables are allocated and initialized in INIT_FORK_ENV in fork_replica.h. These two variables are re-initialized at the exit of group_sync(), therefore explicit re-initialization is not needed.

The function barrier in Fork language is translated to the function group_sync() in baseline language.

4.7.2 Locks

The implementation of Simple Lock and Fair lock are both based on the implementation in the Fork implementation for SB-PRAM.

Simple Lock

A simple lock is based on a single integer variable in shared memory. It allows only one thread to be inside a critical section at one time. Other threads have to wait for the release of the lock. There is no guarantee of the order to enter the critical section. The thread that is the first to enter waiting state may not get the lock first.

The simple lock supports group shared locks where the memory of the lock is in group shared memory.

In the implementation, a multiprefix-max operation is used as an atomic test-and-set instruction to implement Simple Lock.

The API for simple lock is the same as that of the Fork implementation for SB-PRAM, see Listing 4.5:

```
/* Simple Lock API, defined as macros */
typedef int simple_lock, *SimpleLock; /* simple locks */
#define new_SimpleLock() (SimpleLock)shalloc(sizeof(SimpleLock))
#define simple_lock_init( sl ) *(sl) = UNLOCKED
#define simple_lockup( sl )
    while (mpmax((int*)(sl),LOCKED) != UNLOCKED) ;
#define simple_unlock( sl ) *(sl) = UNLOCKED;
```

Listing 4.5: The API of Simple Lock from [28]

Fair Lock

A fair lock is similar to a simple lock. It guarantees that only one thread works inside a critical section at one time. Other threads have to wait for the release of the lock. Moreover, a fair lock guarantees the order to enter
4.8 Memory Management

In this section, we describe the usage and implementation of stack (subsection 4.8.1) and heap (subsection 4.8.2) for Fork on REPLICA baseline language. Frames described in subsection 4.8.3 are implemented to support group creation. In subsection 4.8.4 the implementation of group splitting in Fork is described in detail.

4.8.1 Stack

In Fork, variables can be either private or shared. Shared variables can be global shared or group shared. In baseline language, private variables are allocated on the private stack. Global shared variables are allocated and initialized in global space. However, shared local variables declared inside functions are treated as private variables.

To support shared local variables in baseline language, space for shared local variables is allocated on _shared_stack, and a pointer variable on the private stack which points to that space is used as the shared local variable.

In Fork, there are shared formal parameter variables which are allocated on the shared stack. In our implementation, the shared formal parameter variables are passed as private pointer-typed parameters that point to the shared formal parameter variables, which are allocated on the group shared memory. Since the shared variables are implemented by private pointers, the shared formal parameter variables do not behave in the same way as in Fork. The shared variable passed in by the caller could be modified in the called function but the modification of the pointed-to shared memory

the critical section. The thread that is the first to enter waiting state will get the lock first. Therefore it is more fair than simple lock. Fair Locks can be used in situations where fairness among threads is considered important. However, a fair lock is more costly comparing to a simple lock since it needs more memory and more computation than a simple lock.

Fair Locks also support group shared locks where the memory of the lock is in group shared memory.

In the implementation, a multiprefix-add operation is used as an atomic test-and-set instruction to implement acquisition of a fair lock [28].

The APIs for Fair Lock are the same as that of the Fork implementation for SB-PRAM, see Listing [4.6]:

```c
fair_lock *new_FairLock( void );
#define fair_lock_init( pfl ) \
( pfl )->nextnum = (pfl)->actnum = 0
void fair_lockup( volatile fair_lock *fl );
#define fair_unlock( fl ) ( fl )->actnum++
```

Listing 4.6: The API of Fair Lock
location would remain visible to the caller, which is in conflict with C (and thus Fork) semantics. In the future, this feature could be improved to be compatible with Fork, e.g. by creating a fresh copy of the pointed-to shared variable on the group-shared stack or heap at the call that will be accessed in the called function instead.

4.8.2 Heap

In Fork, the heap can be divided into Global Shared Heap, Group Shared Heap and Private Heap by its usage.

Global Shared Heap is used to allocate memory for access from all threads.

Group Shared Heap is used to allocate memory for access from threads within its group.

Private Heap is used to allocate memory for access from each thread only.

Since the support for Private Heap allocation in the baseline language is not available yet, only Global Shared Heap and Group Shared Heap are implemented in our solution.

![Heap organization in baseline language to support Fork](image)

Figure 4.4: Heap organization in baseline language to support Fork

Dynamic Memory Management

In the baseline language, there is _shared_heap that can be used as Global Shared Heap. At the entry of a program, a certain amount of memory on _shared_heap is reserved in _global_shared_heap for the Global Shared Heap. The remaining memory on _shared_heap is used as Group Shared Heap. In
group splitting, the Group Shared Heap is split and each subgroup gets a part of the Group Shared Heap, see Figure 4.4.

- Group Shared Heap
  Memory on the Group Shared Heap is allocated by calling

  ```c
  void * shalloc(int t)
  ```

  After calling, a space of size \( t \) is allocated from _shared_heap of its own group. The space is allocated for one group, therefore all the threads share one copy. The allocated memory does not need to be freed since it is freed automatically at the end of the current group.

- Global Shared Heap
  Memory on the Global Shared Heap is allocated by calling

  ```c
  void * shmalloc(int t)
  ```

  After calling, a space of size \( t \) is allocated from _global_shared_heap. The space is allocated for each thread, therefore each thread gets one block of memory of size \( t \). The allocated memory must be freed by calling

  ```c
  shfree(void *p)
  ```

  in each thread since it is not freed automatically.

There are several algorithms that improve the performance of parallel memory management \[15, 12\]. However, they are not for the PRAM computing model. The technique that is useful in PRAM is to use multiple areas and assign a thread to an area in round robin style. In the Fork implementation for SB-PRAM, the management of the global heap uses multiple areas and threads are assigned to the area by the size they required. Therefore, threads requiring the same size will contend the lock of one area. Considering the memory allocating pattern in these examples is usually allocating the same size for each thread, an alternative method that assigns a thread an area according to its thread id is used.

The Global Shared Heap is organized in such a way that multiple threads can allocate heap memory in parallel. In the current implementation, the Global Shared Heap is divided into eight pieces or areas. Each thread is hashed into one of these eight areas by its thread id. Before allocating, each thread must acquire the lock of that area. If it is successful, allocating starts. Otherwise, the thread has to wait until the lock is released from other threads.
The allocated memory is organized as a doubly linked list. A doubly linked list is a simple solution since the example Fork programs implemented do not have complex allocating patterns.

In summary, the implementation of shm[][] allocates uses locks and multiple lists to guarantee the correctness of concurrent allocation of the global heap.

In REPLICA T5, the size of whole shared memory is about 30 MB. The Group Shared Heap is about 22 MB and the size of the Global Shared Heap is about 8 MB. If there is no shm[][] allocates function call in the program, the space reserved for Global Shared Heap could be eliminated by specifying a compiler flag, see Appendix B. Therefore, threads get the whole shared memory as the Group Shared Heap.

Table 4.2 summarizes the API for Dynamic Memory Management of Fork.

<table>
<thead>
<tr>
<th></th>
<th>Allocate</th>
<th>Free</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group Shared Heap</td>
<td>shalloc</td>
<td></td>
<td>one piece of memory for one group</td>
</tr>
<tr>
<td>Global Shared Heap</td>
<td>shmalloc</td>
<td>shfree</td>
<td>one piece of memory for each thread</td>
</tr>
</tbody>
</table>

### 4.8.3 Frames

In the Fork implementation for SB-PRAM, there are shared group frame and private group frame to support the group concept. When a new group is created, a shared group frame is allocated on the group’s shared memory space and a private group frame is allocated on each thread’s private memory space. Moreover, there are shared procedure frame and private procedure frame to support calling asynchronous and synchronous functions. When calling synchronous functions, the shared function arguments and shared local variables are store on the shared procedure frame.

In the baseline language, there is no support of a shared procedure frame in the calling conventions. In every function call, a procedure frame is set up on each thread’s private stack. Therefore, shared procedure frame is not supported. In our implementation, the shared function arguments are treated as private arguments. The shared local variables are supported by using private pointers that point to the shared variables allocated on the shared stack.

To support the group concept, the shared group frame and private group frame are supported. When a new group is created, both frames are allocated, as described further below. Our implementation assumes that the baseline compiler does not reorder shared memory accesses.

The implementation of group frames in this thesis is also inspired by the implementation of e-language [17][18].
4.8. Memory Management

Private Group Frame

A private group frame is allocated on each thread’s private stack. The global private variable \_gfp_pointer is added to support stack unwinding for private group frames on private stack. We assume that the baseline compiler will preserve the allocation order of the declared variables in the SAVE_GFP_FRAME statements. Otherwise, the RESTORE_GFP_FRAME statements will be incorrect.

The old values of \_gfp_pointer, \_thread_id, \_number_of_threads, \_group_no, \_shared_heap and \_shared_stack are stored in the private group frame. The old values of shared variables in the shared group frame are also stored in the private group frame. When the group ends, these values are stored back to these variables by a RESTORE_GFP_FRAME statement. See Listing 4.7.

\[
\text{Listing 4.7: Private Group Frame (GFP) Allocation and Destruction}
\]

Shared Group Frame

A shared group frame is allocated on the group’s shared stack. It is used to store the \_group_barrier and \_group_id. These two group shared variables are used in function group_sync() to synchronize all the threads in the group. The values of \_group_barrier and \_group_id are re-initialized to zero after being used in group_sync(). When a group ends, there is no need to free the memory allocated for the two variables since group shared memory is freed automatically.
#define NEW_GFS_FRAME _shared_stack -= 4; \ 
    _group_id = _shared_stack; \ 
    _shared_stack -= 4; \ 
    _group_barrier = _shared_stack;

Listing 4.8: Shared Group Frame (GFS)

4.8.4 Group Splitting

In execution, all threads belong to the root group at the entry of the main function. The group is split when a program running in synchronous mode enters the following constructs: If/For/While/Do-while/Fork statements.

In the Fork statement, group splitting is explicit. Threads enter different subgroups according to the assignment to @ in the fork statement.

In If/For/While/Do-while statements, group splitting is implicit. When the condition of these statements is based on a private variable, each thread may evaluate the condition to a different value. In this situation, the group is split. Otherwise, there is no group splitting.

• Fork Statement

In a Fork statement fork(expr1; expr2; expr3) stmt, the number of groups is specified by expr1. Then the shared memory is split as in Figure 4.5.

Figure 4.5: Memory organization before and after a Fork statement in a thread group
4.9. Jumps

- **If Statement**
  In an If statement, the group is split into two subgroups. The group number is assigned to zero for the subgroup executing the then statement, and to one for the subgroup executing the else statement. The memory organization is similar to the Fork statement except that there are only two subgroups.

- **For/While/Do-while Statement**
  In a For/While/Do-while statement, a new subgroup is created for all threads that enter the loop body. Moreover, the _thread_id and _number_of_threads are updated at each iteration since threads are allowed to leave the loop body in different iterations. The translation is illustrated in Section 3.7.

  The performance cost of updating _thread_id and _number_of_threads in each iteration is not ignorable. Therefore, it is important for the Fork programmer to check whether it is really necessary to use group splitting in For/While/Do-while Statement.

4.9 Jumps

Statements such as `return`, `break`, `continue` and `goto` should be used very carefully since they change the control flow in execution.

In the Fork implementation for SB-PRAM [28], `return` and `break` are correctly handled by the compiler. `Continue` is only allowed in asynchronous mode. In synchronous mode, `Goto` is safe if it is used within one group’s activity range.

In group splitting, the `return` statement jumps out of the current group. Before jumping out of the group, an `END_GROUP` statement is added to restore the group frame. A `synchronize` statement is also inserted to free the waiting threads in the group since the waiting threads do not know how many threads will return. After the synchronization, these waiting threads have to execute an `UPDATE_TID_AND_NTHREADS` statement to update the number of threads in the group. Otherwise, the number of threads in the group is incorrect since some threads have returned.

In our implementation, the `return` statement in nested groups is also supported. Each group splitting/creation statement will insert an `END_GROUP` statement and a `synchronize` statement before the `return` statement. The inserted `END_GROUP` statements here are to unwind the stack of private group frames (GFP) on the private stack.

The `break`, `continue` and `goto` statements are handled in the same way as with the Fork implementation for SB-PRAM. Practically, it is good to avoid using `goto` and `return` in synchronous mode.
Chapter 5

Measurement and Results

The implementation is tested from different perspectives.

Firstly, the example programs in the Fork implementation for SB-PRAM are compiled using the Fork-to-baseline compiler described in this thesis. Then these programs are executed on the REPLICA simulator IPSMSimX86 to verify the results. These programs test each feature implemented and described in Chapter 3.

Secondly, the Fork programs mergesort and quicksort are tested on the REPLICA simulator using different sizes of data. The purpose is to test whether the Fork implementation on REPLICA scales to large size problems.

Thirdly, existing baseline programs are used to compare with programs that have the same functionality but are re-written in Fork language. These Fork programs are firstly translated to a baseline program using the Fork-to-baseline compiler. The translated baseline programs are compared to the existing baseline programs by metrics such as code size, execution clock cycles on simulator, etc.

At last, the compilation time of the Fork-to-baseline compiler is evaluated.

5.1 Programs in the Fork implementation for SB-PRAM

There are about 44 programs in the example directory in the Fork distribution for SB-PRAM [28]. Since the baseline compiler and simulator do not yet support floating point computation fully, these programs using floating point computation are not considered. In this thesis, 12 programs are ported and tested using the Fork-to-baseline compiler. The purpose is to cover all the features of Fork that are described and implemented in Chapter 3 and Chapter 4.
5.1.1 Porting Fork Programs

Some features in the Fork implementation for SB-PRAM are not implemented yet in this thesis. Moreover, there are some differences in the APIs of libraries provided in the Fork implementation for SB-PRAM and the libraries implemented in baseline language for REPLICA. Therefore most of the Fork programs need to be modified in order to be compiled and executed on the REPLICA simulator. These modifications do not change the main structure and algorithms in these programs. The steps of porting Fork programs are summarized below.

1. Functions such as `printf()`, `prS()`, `getct()`, `trace()`, `srand()`, `rand()` are not supported yet. Comment them out.
2. Inline assembly of SB-PRAM in Fork programs should be commented out since it could not be executed on REPLICA platform. If necessary, use inline assembly in baseline library instead.
3. The join construct in Fork is not supported yet, try to replace it with other equivalent statements.
4. Check every function call to `shalloc(int size)` and `shmalloc(int size)`, as the semantics of size is different across platforms. In our implementation, size means the number of bytes other than number of words.
5. Check every call to `memcpy(int size)`, the size means the number of words in the Fork implementation for SB-PRAM. In the implementation, since some baseline programs like threshold and blur use the parameter size as number of bytes, the size in memcpy indicates the size in number of bytes.

5.1.2 Testing using Fork Programs

The programs from the Fork implementation for SB-PRAM are firstly translated to baseline language by our Fork translator and then compiled to REPLICA assembly by the baseline compiler. The compile flags for the baseline compiler are for the T5 architecture variant of REPLICA, -O0 and disabled ”-enable-replica-ilp”. The results of these programs running on the simulator IPSSimX86 are correct.

When using compiling flag -O2 or -enable-replica-ilp, there are some bugs in certain generated REPLICA assembly programs. Therefore, the flag -O0 is used and the flag ”-enable-replica-ilp” is disabled, although the result is slower.

The tested programs are listed below. The features that each program covers are shown in Table 5.1.

- `eratosthenes.c`
  This program tests the Sieve of Eratosthenes algorithm. It is to compute the prime numbers within a given range. It includes many integer multiplications.
- `erewprefix.c`
This program tests a parallel prefix sum algorithm for EREW PRAM. Since multiprefix operations are not available in EREW PRAM, the prefix sum is computed iteratively.

- **glock.c**
  This program tests the group splitting and the fair lock. In this program, there are two groups that would write two shared variables at the same time. A fair lock is used to protect the critical section.

- **group_fork.c**
  This program tests the group splitting and fork statement. Threads enter different subgroups by a fork statement. The result can be verified in the simulator by observing the change of system variables.

- **group_split.c**
  This program tests the group splitting. In this program, the root group is split into two subgroups by an *If* statement with a private condition.

- **hello.c**
  This program tests the group splitting and system variables. In this program, threads enter one of the two subgroups according to their thread id. The result can be verified in the simulator by observing the change of system variables.

- **hello_slock.c**
  This program tests the simple lock. In this program, threads try to get a ticket number at the same time. A simple lock is used to protect the critical section.

- **mergesort.c**
  This program implements a parallel merge sort algorithm. This program is a recursive version of merge sort. A fork statement is used to split the group. The result can be verified in the simulator by observing the sorted array.

- **mpadd.c**
  This program tests the multiprefix-add operation.

- **parprefix.c**
  This program tests a parallel prefix-sum algorithm for CRCW PRAM. Since multiprefix operations are available, it uses MPADD here.

- **qsort.c**
  This program tests the sequential quick sort algorithm. It is a recursive version of quick sort for a single thread.
• **quicksort.c**

This program tests the parallel quick sort algorithm. This program is a recursive version of quick sort. A fork statement is used to split the group. The result can be verified in the simulator by observing the sorted array.

To illustrate the translation, the baseline version and translated Fork version of mergesort.c and quicksort.c are included in Appendix C.4.1 and Appendix C.4.2.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Result</th>
<th>Fork example program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared/Private Variable</td>
<td>Pass</td>
<td>all programs</td>
</tr>
<tr>
<td>System Variable</td>
<td>Pass</td>
<td>all programs</td>
</tr>
<tr>
<td>MultiPrefix Operation</td>
<td>Pass</td>
<td>mpadd.c, parprefix.c, group_split.c, etc.</td>
</tr>
<tr>
<td>Sync/Async Region</td>
<td>Pass</td>
<td>all programs</td>
</tr>
<tr>
<td>Synchronicity of Function</td>
<td>Pass</td>
<td>parprefix.c, erewprefix.c, quicksort.c, etc.</td>
</tr>
<tr>
<td>Thread Group</td>
<td>Pass</td>
<td>hello.c, group_split.c, group_fork.c, etc.</td>
</tr>
<tr>
<td>Shared Pointer</td>
<td>Pass</td>
<td>parprefix.c, quicksort.c, mergesort.c, etc.</td>
</tr>
<tr>
<td>Shared Heap</td>
<td>Pass</td>
<td>parprefix.c, erewprefix.c, quicksort.c, etc.</td>
</tr>
<tr>
<td>Locks</td>
<td>Pass</td>
<td>hello_slock.c, glock.c</td>
</tr>
<tr>
<td>Fork Statement</td>
<td>Pass</td>
<td>group_fork.c, quicksort.c, mergesort.c</td>
</tr>
</tbody>
</table>

**5.2 Test of scalability of Fork programs on REPLICA**

Two programs, mergesort and quicksort, are used to test the scalability of Fork programs on REPLICA.

In the SB-PRAM simulator, the quicksort and mergesort programs are capable of sorting 1024 integers. This limitation is due to the memory size. Since the divide and conquer method is used in quicksort and mergesort in Fork, group splitting occurs when the problem is divided. In group splitting, the heap and stack are halved. Since the total memory size in the Fork implementation for SB-PRAM is by default set to about 30M bytes, the shared heap and stack would be too small to use after several recursive group splits.

In REPLICA-T5, the memory is about 30M bytes which is similar to the Fork implementation for SB-PRAM. Therefore, the mergesort and quicksort programs are expected to have similar computation capability on REPLICA-T5.

However, the cycle count in the REPLICA simulator is different from the SB-PRAM simulator. In the REPLICA simulator, executing one assembly
5.2. Test of scalability of Fork programs on REPLICA

An instruction is usually either 1024 or 512 cycles for 2048 threads. While in the SB-PRAM simulator, the cycle count is much lower since the cycle count in the SB-PRAM (simulator) is actually the number of PRAM steps. Therefore, the direct comparison of cycle count in the REPLICA simulator and the SB-PRAM simulator does not give much information. The execution cycles of quicksort on SB-PRAM simulator are in Table 5.2 and Figure 5.1.

<table>
<thead>
<tr>
<th>number of threads</th>
<th>number of integers</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>128</td>
<td>114223</td>
</tr>
<tr>
<td>1</td>
<td>256</td>
<td>230165</td>
</tr>
<tr>
<td>1</td>
<td>512</td>
<td>507307</td>
</tr>
<tr>
<td>1</td>
<td>1024</td>
<td>1075211</td>
</tr>
<tr>
<td>1</td>
<td>2048</td>
<td>2368765</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
<td>160095</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>298109</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>595917</td>
</tr>
<tr>
<td>2</td>
<td>1024</td>
<td>1228325</td>
</tr>
<tr>
<td>2</td>
<td>2048</td>
<td>2510389</td>
</tr>
<tr>
<td>2048</td>
<td>128</td>
<td>262133</td>
</tr>
<tr>
<td>2048</td>
<td>256</td>
<td>309277</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>340999</td>
</tr>
<tr>
<td>2048</td>
<td>1024</td>
<td>371617</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>373195</td>
</tr>
</tbody>
</table>

To test quicksort and mergesort, the random number generator is needed. In the Fork implementation for SB-PRAM, the random generator generates random numbers that are not unique, which is not suitable for mergesort. Therefore, instead of porting the random generator from SB-PRAM assembly to REPLICA assembly, a new random generator script is used to construct both unique random numbers and non-unique random numbers.
Chapter 5. Measurement and Results

5.2.1 Quicksort

The quicksort program can sort arrays with duplicated entries. In this evaluation, up to 2048 integers are sorted using 2048 threads. The quicksort program crashes when sorting 4096 integers, because the recursive group splitting goes too deep. This is a difference between mergesort and quicksort. In mergesort, integers are divided evenly in each recursion, while in quicksort, integers are divided by the pivot element. If the pivot element is the median, these numbers are divided evenly. Otherwise, the division could be very uneven. Therefore, the group splitting in quicksort also depends on the content of the random generated elements.

The recursive group splitting in quicksort could run much deeper than mergesort. When sorting 4096 integers, some threads recursively go into group splitting more than 18 times. Then the memory in the leaf group would be less than $30M/2^{18}$, about 120 bytes, which is too small. Moreover, the quicksort.c program uses more group shared memory by calling shalloc() than the mergesort.c program.

The results are shown in Table 5.3 and Figure 5.2. The code of quicksort is included in Appendix C.4.2. When sorting 512 integers using one or two
threads, it takes more than 15 minutes to finish. Sorting 1024 integers is possible using one or two threads, however, it is not tested because of its time consumption.

Table 5.3: Execution Cycles of quicksort using 2048 threads on REPLICA-T5-4-512+

<table>
<thead>
<tr>
<th>number of threads</th>
<th>number of integers</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>128</td>
<td>20694016</td>
</tr>
<tr>
<td>1</td>
<td>256</td>
<td>45884928</td>
</tr>
<tr>
<td>1</td>
<td>512</td>
<td>103987200</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
<td>20837888</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>48492032</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>67757056</td>
</tr>
<tr>
<td>2048</td>
<td>128</td>
<td>5719552</td>
</tr>
<tr>
<td>2048</td>
<td>256</td>
<td>7639552</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>13768704</td>
</tr>
<tr>
<td>2048</td>
<td>1024</td>
<td>17251328</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>29030400</td>
</tr>
</tbody>
</table>
5.2.2 Mergesort

The mergesort program in Fork can only sort arrays in which each element is different. In REPLICA-T5, up to 32768 integers can be sorted when using 2048 threads. The simulator IPSMSimX96 ran about 5 minutes to finish the work. The results are in Table 5.4 and Figure 5.3.

When the number of threads reduces to one, mergesort calls the sequential quicksort function defined in baseline library, which is the same as quicksort. Therefore, only 2048 threads are tested.

More threads can be used in mergesort. The limitation is imposed by the memory size, since group splitting halves the memory.

The code of mergesort is included in Appendix C.4.1.
Table 5.4: Execution Cycles of mergesort using 2048 threads on REPLICA-T5-4-512+

<table>
<thead>
<tr>
<th>number of integers</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>3849216</td>
</tr>
<tr>
<td>256</td>
<td>4475904</td>
</tr>
<tr>
<td>512</td>
<td>5130240</td>
</tr>
<tr>
<td>1024</td>
<td>5812224</td>
</tr>
<tr>
<td>2048</td>
<td>6521856</td>
</tr>
<tr>
<td>4096</td>
<td>6920704</td>
</tr>
<tr>
<td>8192</td>
<td>11852800</td>
</tr>
<tr>
<td>16384</td>
<td>22942720</td>
</tr>
<tr>
<td>32768</td>
<td>46563840</td>
</tr>
</tbody>
</table>

Figure 5.3: Execution Cycles of quicksort and mergesort on different datasets on REPLICA-T5-4-512+
5.3 Reimplementation of existing baseline programs

In this section, the implementation is evaluated by comparing the execution of these existing baseline programs and their corresponding Fork programs that are re-written manually. Both versions of these programs are compiled with the same compiling flags of the same version of the baseline compiler. Then each program is executed on the simulator IPSMSimX86 with different configurations. These configurations represent REPLICA architecture variants T5, T7, T11 and T14. The number of functional units within each processor increases when the REPLICA architecture evolves from T5 to T14.

Two metrics are chosen to measure the implementation. The first metric is the execution time in clock cycles, which is retrieved from the statistics of the simulator. This metric measures the execution time of the whole program. It is slightly different from the Execution Time in Clock Cycles (Timer) which measures the clock cycles between the two statements \_start\_timer and \_stop\_timer. Since the translator inserts statements in the entry and exit of main function, the metric Execution Time in Clock Cycles is more suitable for measuring the overhead introduced in the Fork program.

The second metric is the number of lines of code of baseline and Fork programs. Since Fork is a high level language, it is expected that a Fork program is shorter than the same program written in baseline language. This metric shows how many lines are saved by writing a program in Fork other than baseline language.

The three programs that are used in the evaluation are image processing programs. The baseline version of the first two programs, threshold and blur, were used in another report [47]. The program edge is from article [33]. The Fork version of the programs are re-written manually.

5.3.1 Threshold

Threshold.c is a program that transforms an image to white-black style according to the value of each pixel. In threshold.c, there are more than one million addition operations to be executed.

According to the original threshold.c, two Fork implementations are constructed (Appendix C.1). In both Fork implementations, the sync() function from the baseline threshold.c is replaced by group\_sync() because the original sync() can not synchronize all threads in some situation, although the function sync() works correctly in this program. However, after the modification, execution cycles increase because group\_sync() consists of more instructions.

In the first implementation, a for-loop is replaced with a memcpy function call (see Figure 5.4). In the original for-loop, data is copied by bytes, while the memcpy function implemented in the baseline library copies data by words. Therefore, the memcpy function saves CPU cycles when doing
5.3. Reimplementation of existing baseline programs

the same work. In the evaluation result of Table 5.3, this version outperforms the original threshold.c in architecture T7, T11 and T14. This is the reason for constructing the second implementation since the evaluation is to measure the overhead of the Fork implementation not the optimization of modifying the code structure. However, the result is interesting since it shows that the optimization introduced by using memcpy already compensates the overhead of the Fork implementation.

In the second implementation, the original for-loop is preserved. The purpose is to measure the overhead introduced by the implementation by preserving the computation structure of the original program. In figure 5.4, the overhead introduced by the Fork implementation in threshold.c is less than 4% in all REPLICA architectures.

The number of lines of the baseline and Fork threshold.c programs are in Table 5.6.
Table 5.5: Comparison of Execution Cycles of threshold programs in different implementation

<table>
<thead>
<tr>
<th>Architecture</th>
<th>baseline</th>
<th>Fork (memcpy)</th>
<th>Fork (no memcpy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLICA-T5-4-512+</td>
<td>3388416</td>
<td>3413504</td>
<td>3517952</td>
</tr>
<tr>
<td>REPLICA-T7-4-512+</td>
<td>2284032</td>
<td>2248192</td>
<td>2333184</td>
</tr>
<tr>
<td>REPLICA-T11-4-512+</td>
<td>2207232</td>
<td>2171392</td>
<td>2256384</td>
</tr>
<tr>
<td>REPLICA-T14-4-512+</td>
<td>1662976</td>
<td>1615872</td>
<td>1713664</td>
</tr>
</tbody>
</table>

Table 5.6: Comparison of number of lines of the threshold program in different languages

<table>
<thead>
<tr>
<th>Metric</th>
<th>baseline</th>
<th>Fork (memcpy)</th>
<th>Fork (no memcpy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of lines</td>
<td>69</td>
<td>51</td>
<td>56</td>
</tr>
</tbody>
</table>

5.3.2 Blur

Blur is an image processing program that involves many additions and integer divisions. The original baseline program and its Fork version is shown in Appendix C.2. The structure of the program is preserved. Similar to threshold, the sync() function is replaced by group_sync().

The test result is shown in Table 5.7 and Figure 5.5. Initially, the early version Fork program outperforms the original one although there is overhead introduced by the Fork implementation. The reason is that a specifier `const` is added to the following statement: `unsigned int blur_size = 3`.

When `const` is used, the constant values are used in the assembly program instead of loading the value of the variable. Therefore, the assembly instructions are reduced. In the T5 architecture variant, the number of execution cycles of the early version is 34762240 which is 0.6% smaller than that of the original baseline blur in REPLICA T5 (see Table 5.7). This result shows that a simple optimization of using a constant instead of a variable already compensates the Fork implementation in REPLICA T5.

In Table 5.7 and Figure 5.5, the additional specifier `const` is removed. In Figure 5.5, the result shows that the overhead introduced by the Fork implementation is less than 1% in the four REPLICA architectures.

The number of lines of the baseline and Fork blur.c programs are in Table 5.8.
5.3. Reimplementation of existing baseline programs

Figure 5.5: Comparison of execution cycle counts of different blur.c implementations

<table>
<thead>
<tr>
<th>Architecture</th>
<th>baseline</th>
<th>Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLICA-T5-4-512+</td>
<td>34985984</td>
<td>34992640</td>
</tr>
<tr>
<td>REPLICA-T7-4-512+</td>
<td>24678397</td>
<td>24786944</td>
</tr>
<tr>
<td>REPLICA-T11-4-512+</td>
<td>23758333</td>
<td>23866880</td>
</tr>
<tr>
<td>REPLICA-T14-4-512+</td>
<td>15468541</td>
<td>15577088</td>
</tr>
</tbody>
</table>

Table 5.7: Comparison of number of lines of the blur program in different languages

<table>
<thead>
<tr>
<th>Metric</th>
<th>baseline</th>
<th>Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of lines</td>
<td>114</td>
<td>89</td>
</tr>
</tbody>
</table>
5.3.3 Edge

The program edge.c consists of many integer additions and integer multiplications. The original baseline program the Fork version edge.c is in Appendix C.3. The original structure of edge is preserved. The sync() function is replaced by group_sync().

The results in Table 5.9 and Figure 5.6 show that the overhead introduced by the Fork implementation is less than 2% in the four REPLICA architectures.

The number of lines of the baseline and Fork edge.c programs are in Table 5.10.

Figure 5.6: Comparison of execution cycle counts of different edge.c implementations
5.4. Test compilation time

Table 5.9: Comparison of Execution Cycles of edge programs in different implementation

<table>
<thead>
<tr>
<th>Architecture</th>
<th>baseline</th>
<th>Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLICA-T5-4-512+</td>
<td>41922560</td>
<td>42321920</td>
</tr>
<tr>
<td>REPLICA-T7-4-512+</td>
<td>27692544</td>
<td>27949568</td>
</tr>
<tr>
<td>REPLICA-T11-4-512+</td>
<td>26315264</td>
<td>26567168</td>
</tr>
<tr>
<td>REPLICA-T14-4-512+</td>
<td>18497536</td>
<td>18728960</td>
</tr>
</tbody>
</table>

Table 5.10: Comparison of number of lines of the edge program in different languages

<table>
<thead>
<tr>
<th>Metric</th>
<th>baseline</th>
<th>Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of lines</td>
<td>89</td>
<td>81</td>
</tr>
</tbody>
</table>

5.4 Test compilation time

The compilation time of our Fork-to-baseline compiler is evaluated by measuring the translation time of Fork programs. The twelve Fork example programs and three hand-written Fork programs (threshold, blur, merge) are measured. The measurement is carried out on Ubuntu 10.10 with Intel P6100 (two cores). The time (in seconds) spent on each core is summed up. The compilation time of each program is the average compilation time of five measurements.

The compilation time is analyzed against the Lines Of Code (LOC), see Table 5.11 and Figure 5.7. The correlation coefficient of the compilation time and LOC is 0.83778. Since it is close to 1, it indicates that there is a strong positive relation between the compilation time and LOC.
Table 5.11: Compilation time of Fork programs using Fork-to-baseline compiler

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines Of Code</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mergesort</td>
<td>172</td>
<td>1.36960</td>
</tr>
<tr>
<td>quicksort</td>
<td>133</td>
<td>1.33120</td>
</tr>
<tr>
<td>threshold_fork</td>
<td>51</td>
<td>1.09520</td>
</tr>
<tr>
<td>blur_fork</td>
<td>87</td>
<td>0.99360</td>
</tr>
<tr>
<td>edge_fork</td>
<td>81</td>
<td>1.04080</td>
</tr>
<tr>
<td>eratosthenes</td>
<td>66</td>
<td>1.11360</td>
</tr>
<tr>
<td>crewprefix</td>
<td>85</td>
<td>1.28240</td>
</tr>
<tr>
<td>glock</td>
<td>55</td>
<td>1.05280</td>
</tr>
<tr>
<td>group_fork</td>
<td>40</td>
<td>1.08000</td>
</tr>
<tr>
<td>group_split</td>
<td>20</td>
<td>0.94640</td>
</tr>
<tr>
<td>hello</td>
<td>49</td>
<td>1.06480</td>
</tr>
<tr>
<td>hello_slock</td>
<td>24</td>
<td>0.93360</td>
</tr>
<tr>
<td>mpadd</td>
<td>27</td>
<td>1.04240</td>
</tr>
<tr>
<td>parprefix</td>
<td>70</td>
<td>1.15120</td>
</tr>
<tr>
<td>qsort</td>
<td>50</td>
<td>0.97360</td>
</tr>
</tbody>
</table>
5.4. Test compilation time

![Compilation time vs. Lines Of Code (LOC)](image)

Figure 5.7: Compilation time of Fork programs using our Fork-to-baseline compiler
5.5 Summary

In the first part of the evaluation, the Fork implementation is measured by running 12 programs from the example programs of the Fork implementation for SB-PRAM. All these programs are compiled and run correctly on REPLICA architecture T5. Since these 12 programs cover all the features listed in Table 3.1, the result shows that very likely these features of the Fork language are correctly implemented in the implementation. The result also shows that the implementation is compatible to the implementation of Fork for SB-PRAM regarding the features in Table 3.1.

In the second part, the scalability test shows that our solution can solve problems with different data sets on REPLICA. The upper bound of the numbers that can be sorted using parallel quicksort or parallel mergesort is similar to the Fork implementation for SB-PRAM.

In the third part of the evaluation, the Fork implementation is measured by running three programs on four different REPLICA architecture configurations. The result shows that the overhead of the implementation is less than 4% in the three programs. The result also shows that optimizations such as copying by words instead of bytes, or using a constant instead of a variable, could reduce the number of execution clock cycles more than the overhead introduced by the implementation. In the three programs, the length of a Fork program is about 10% shorter than the length of the baseline program.

At last, the compilation time of Fork programs using the Fork-to-baseline compiler is evaluated. The result shows that there is a strong positive correlation between the compilation time and Lines Of Code (LOC).
Chapter 6

Related Work

6.1 Languages

There are many languages designed for parallel computing on multi-core and many-core systems. In this section, an incomplete survey is carried out to classify several parallel programming languages according to the programming model [48, 21] of the programming language. The programming models are Shared-Memory, Message-Passing and Distributed Shared Memory. Some languages are extended to different programming models. For example, OpenMP is compiled into MPI [34, 35], and a UPC runtime system can be built based on MPI and POSIX threads [51]. In the following classification table, only the initial design of the language is considered.

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Hardware</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared-Memory</td>
<td>multi-core</td>
<td>OpenMP, e-language, Fork, XMTC</td>
</tr>
<tr>
<td></td>
<td>many-core</td>
<td></td>
</tr>
<tr>
<td>Distributed Shared Memory</td>
<td>parallel machines</td>
<td>UPC, NestStep</td>
</tr>
<tr>
<td>Message-Passing</td>
<td>clusters</td>
<td>MPI, RCCE [44]</td>
</tr>
<tr>
<td></td>
<td>many-core</td>
<td>CUDA, OpenCL</td>
</tr>
<tr>
<td></td>
<td>GPU</td>
<td></td>
</tr>
</tbody>
</table>

The languages that are in the same category of Fork are considered related and will be discussed in the following section. UPC and NestStep are also considered related since they have a virtual shared memory [14, 29].

6.1.1 E-language

The e-language [17] is the language designed for Total Eclipse, the predecessor of REPLICA. E-language has features such as shared/private variables,
thread groups, group splitting etc. Although e-language is not as high level as Fork, it provides inspirations about the implementation of thread groups on REPLICA. The details of e-language were discussed in Section 2.2.2.

6.1.2 XMTC

The eXplicit Multi-Threading C (XMTC) [20] is the programming language of the Explicit Multi-Threading (XMT) PRAM on-chip architecture [50]. XMT evolves from the PRAM model and aims to provide an easy general-purpose parallel programming model. XMT could execute PRAM-like programs efficiently. XMTC is a Single-Program Multiple-Data (SPMD) extension of the C language with three new constructs: spawn, join and ps (Prefix-Sum). XMT supports the arbitrary CRCW PRAM model. In the spawn statement, threads enter into parallel mode. Unlike the PRAM model, each thread executes at its own speed. At the join statement, threads end the parallel mode and enter into sequential mode.

6.1.3 UPC

Unified Parallel C (UPC) [9, 14, 48] is an extension to the C language that aims to provide a uniform programming model for both shared and distributed memory architectures. The memory address space is partitioned between processors; however, each processor can access any address of the memory. Similar to the Fork language, there are private and shared memory storage classes in UPC. Unlike Fork, each variable in UPC belongs to a specific processor, which is called affinity. In fact, the variable is stored in the physical memory of the processor that it belongs to. The UPC execution model is also SPMD.

6.1.4 OpenMP

OpenMP [7] is an API for parallel computing on shared memory systems. OpenMP consists of compiler directives, library routines and environment variables. The application programmer adds compiler directives in the program. OpenMP compiler directives support shared and private data, parallel constructs, worksharing constructs, task constructs, synchronization, etc. OpenMP is supported on C, C++ and Fortran.

6.1.5 NestStep

NestStep [29, 30] is a programming language designed for the BSP (Bulk-Synchronous-Parallel) programming model [49]. In the BSP model, the parallel computation consists of supersteps and global barriers. Unlike the PRAM model, the BSP model does not assume synchronous execution of
processors. The communication cost in BSP model depends on the underlying communication network, while in PRAM model the communication is writing and reading the shared memory.

By extending BSP, NestStep supports nested supersteps and a group concept. NestStep provides a virtual shared memory from the application programmer’s view. There are shared variables and private variables in NestStep. Within a superstep, a processor reads and writes to its local copies of shared variables. The modifications to shared variables are committed at the end of a superstep.

6.2 Compiler and Run-time Libraries for PRAMs

6.2.1 FCC

The first Fork compiler fcc and the supporting libraries for SB-PRAM \cite{32, 31, 28} is referenced in our implementation. Although the hardware of SB-PRAM is different from REPLICA, the implementation of memory management and group frames are helpful in our implementation to support Fork on REPLICA. The features of the Fork language implemented in this thesis project are compared with the features in fcc. The detailed discussion is given in Chapter 5.

In Section 5.1.2 several example programs from the Fork implementation for SB-PRAM are compiled and tested on the REPLICA simulator. The scalability test of the Fork programs mergesort and quicksort is presented in Section 5.2.

6.2.2 E-language libraries

E-language \cite{17, 18} is designed for the project Total Eclipse which is prior to the project REPLICA (discussed in Section 2.2.2). E-language and the baseline language for REPLICA share some common features such as shared/private variables, global variables, memory organization. Therefore, the implementation of the e-language’s library and header files are also useful in our solution. The implementation of group frames in our solution is inspired by the implementation of the control constructs in e-language. The implementation of synchronization functions in our solution is also based on the related functions (RTL_SYNCHRONIZE,GROUP() and RTL_UPDATE_SUB_SYNC()) in the library of the e-language.

6.2.3 XMTC Compiler

XMTC Compiler (xmcc) is the compiler for the XMTC programming language on the XMT architecture \cite{27, 10}. It consists of three major parts: pre-pass (a source-to-source compiler), core-pass (based on GCC) and post-pass (verifying and linking). It only supports an I/O library including functions
such as printf(). Unlike our implementation of libraries on REPLICA, it does not yet support other libraries such as dynamic memory management. Some other limitations of XMTC are that functions and pointer arithmetics are not supported within spawn statements, and the number of instructions in spawn statements is limited.

6.3 Source-to-source compilers

6.3.1 Fork to CUDA

In [13], it is shown how programs written in Fork can be translated into CUDA programs by a source-to-source compiler. The result shows a speedup of the CUDA program compared to the Fork program running on the SB-PRAM simulator and the sequential program running on CPU. The result also shows the overhead of the source-to-source translation.

6.3.2 NestStep to C compiler

In Holm’s master thesis [24], a source to source compiler that translates the language NestStep to the C language for the CELL processor is constructed. Cetus [38] is used to construct the compiler. It is similar to our work since Cetus is also used to construct the source to source compiler. This work shows that Cetus can be successfully extended for new input languages based on the C language.

In NestStep, there are features like shared and private variables and groups of processors, which are similar to the group concept of Fork. However, there are no synchronous/asynchronous regions, multiprefix operations, etc., since NestStep targets distributed-memory architectures. Although group splitting in supersteps is supported in the language, this compiler does not support nested group splitting. In our implementation, nested group splitting is supported for the Fork language. Moreover, in our implementation, the run-time library has to be built from scratch. Therefore, the language, hardware and runtime system are different between Holm’s work and our work.

6.3.3 OpenMP to POSIX threads compiler

There are many source to source compilers that translate OpenMP programs to POSIX threads based programs. Two of them are discussed here.

The first one, presented in [38], is translating OpenMP to POSIX threads using the Cetus compiler framework. Both OpenMP and POSIX threads are for programming of shared memory systems. In the translation, the IR tree and symbol table are modified to generate target code. This work shows the basic functionalities provided by Cetus.
The second one, presented in [39], is to translate OpenMP programs to a portable thread implementation on several platforms such as Intel Itanium, Intel IA32 and Sun UltraSPARC. The portable thread library is based on POSIX threads. In this work, the method of source-to-source translation is used for these platforms except Itanium. The source-to-source compiler is based on the Open64 compiler framework.

6.3.4 OpenMP to GPGPU compiler

In [37], a source-to-source compiler that translates OpenMP programs to CUDA-based GPGPU programs is constructed to ease the programming for GPGPU. The translation consists of two phases. Firstly, the OpenMP program is translated to an OpenMP program optimized for GPGPU. Then the optimized OpenMP program is translated to a CUDA GPU program. Cetus is used to construct the compiler and apply optimizations in the translation.

Comparing to our source-to-source compilation process, the OpenMP to GPGPU compilation consists of two steps and it focuses on loop translation. In our Fork-to-baseline compilation process, there is only one transforming step and more features are included such as group splitting in if/for/while statements with private conditions.

The OpenMPC programming interface [36] is an extension added to the above work of OpenMP to GPGPU compiler. OpenMPC is based on OpenMP and includes special directives and environment variables for CUDA. OpenMPC interfaces are designed to facilitate the optimization for CUDA in a source-to-source compilation process. The input is an OpenMPC program, and the output is the optimized CUDA program.

6.3.5 G-ADAPT: Adaptive GPU optimization

A framework is proposed in [41] to find the optimal parameters of a CUDA program to achieve best performance. G-ADAPT pragmas are added into the original CUDA program by the programmer to specify the values of each parameter. All the parameters form an optimization space. A source-to-source compiler based on Cetus is used to translate the program using different values of parameters. After recording and analyzing the performance of each set of parameter values, the Regression Tree method is used to learn the best set of parameter values.

In this work, the source-to-source compiler is not only used to recognize G-ADAPT pragmas, but also used to perform data flow analysis, loop analysis, etc.

6.3.6 hiCUDA

The hiCUDA [22] language is designed as a directive-based language for GPGPU programming. The directives are based on the pragma mechanism in C/C++. The hiCUDA source-to-source compiler translates a sequential
Chapter 6. Related Work

program with hiCUDA directives to an equivalent CUDA program. Since the programmer only needs to use the high-level directives in a sequential program, hiCUDA makes programming using CUDA easier. The source-to-source compiler is based on the compiler framework Open64.

6.3.7 OpenMP to MPI compiler

In [34,35], a source-to-source compiler that translates OpenMP programs to MPI programs is constructed using Cetus. The translation consists of two steps. In the first step, the OpenMP program is transformed to SPMD form. In the second step, the communication analysis is applied and communications between MPI processes are added. Therefore, the OpenMP program that is written for a shared memory architecture is translated to a MPI program that can be used in a distributed memory architecture. The result shows that the generated MPI programs perform close to hand-coded MPI programs.

While OpenMP is for shared memory systems, MPI is for distributed memory systems. In our Fork-to-baseline compiler, both Fork and the baseline language are for shared memory systems.

6.3.8 Sequential C++ to Parallelized C++ compiler

In [40], a source-to-source compiler is constructed to parallelize sequential C++ programs by adding OpenMP directives. Both conventional loops and high-level abstractions in Standard Template Library (STL) are parallelized. The implementation is based on the source-to-source compiler framework ROSE.
Chapter 7

Conclusion, Limitations and Future Work

7.1 Conclusion

In this thesis, a source to source compiler that translates Fork language to baseline language is built. The features of the Fork language that are listed in Table 3.1 are implemented. The Fork compiler is constructed to be compatible with the Fork implementation for SB-PRAM \cite{28} as much as possible. The supporting libraries that support functionalities such as group synchronization, locks and memory management are built in baseline language.

The implementation is evaluated from different perspectives. Firstly, each implemented feature of the Fork language is tested and verified. Secondly, scalability of our Fork implementation is tested. It shows that our implementation scales on different problem sizes. Thirdly, the overhead of the implementation is measured and the length of Fork program and baseline program is compared. It shows that the overhead is less than 4% and that the Fork programs are about 10% smaller than the baseline programs.

In conclusion, the Fork language is supported for the REPLICA architecture by building the source to source compiler and supporting libraries.

7.2 Limitations

Some features of Fork are not implemented, such as the join statement, or reader-writer locks.

Some features are different from the Fork implementation for SB-PRAM due to the limitation of our solution, for example, the return statement discussed in Section 4.9.

The stack and heap sizes are both limited on the simulator. Therefore
deep recursion could result in the problem of shared stack overflow. Programs that allocate too much shared memory will fail (discussed in Section 4.8.2).

7.3 Future Work

More features of the Fork language such as the join statement could be added. Library functions such as reader-writer lock support could be added.

The existing implementation of global heap management (shmalloc()) could be improved to reduce the memory fragmentation by using more sophisticated algorithms.

In translation from Fork to baseline, some optimization could be applied to reduce unnecessary overhead. Since our implementation translates a Fork program by translating each construct to the corresponding baseline construct, there are situations where optimizations are possible. Some possible optimizations are as follows.

- Eliminate duplicated synchronization

  For example, when a farm statement is followed immediately by a start statement, there are two group sync() function calls that are executed one after another. One of the function calls can be removed safely.

- Eliminate unnecessary group creation

  In some situations, group creation is not necessary. For example, when calling a synchronous function which does not include early return and does not allocate shared variables or heap memory, the group creation is not necessary. In mergesort.c, the function call to merge() does not need to create a new group.

- using in-line assembly instead of function

  Optimizations such as inline expansion could be applied. For example, the multiprefix operation function calls could be replaced by inline assembly if possible.

A LLVM-based Fork compiler for REPLICA could be developed. The advantage of a Fork compiler for REPLICA is that a Fork program does not need to be compiled twice, firstly by the Fork-to-baseline compiler and then by the baseline compiler. The compiler for the baseline language of REPLICA is implemented in LLVM. To support the Fork language on REPLICA, a Fork program could be translated into the LLVM IR and then use the back-end of the baseline language compiler to generate REPLICA assembly code (discussed in Section 2.3). Both the LLVM IR and the back-end may need to be extended to support Fork. However, since the
Cetus-based Fork-to-baseline compiler is successfully developed in this thesis, a LLVM-based Fork compiler would become easier to develop. The libraries and header files could be re-used when developing a LLVM-based Fork compiler. The implementation of each feature of Fork in LLVM can now reference the corresponding implementation in this thesis.
Chapter 8

Bibliography


Appendix A

Installation

====================================================================
Fork to baseline compiler
====================================================================

====================================================================
1. Installation
====================================================================

The installation is tested on Ubuntu 10.10, Ubuntu 11.10
and Mac OS X version 10.6.8.

The Fork to baseline compiler is in the package cetus.

To use the compiler, firstly, install ANTLR 2.7 on the computer.

Set the variable ANTLR in cetus/build.sh to the path of ANTLR.

Then compile the parser by command ‘‘./build.sh parser’’.

At last, build the compiler by command ‘‘./build.sh bin’’.

====================================================================
2. Directories and Files
====================================================================

The Fork programs and libraries are in package fork_example.

The header files for Fork are in directory fork_example/fork/include.

It includes the original fork.h from the Fork implementation

for SB-PRAM and other new header files.

The baseline header files are in fork_example/include.

The baseline libraries are in fork_example/lib
The script files in package fork_example are useful for Fork programs.

fork_replica_prep.sh
   -- preprocessing of Fork programs to replace symbol '#'
      by '_number_of_threads'.

gen_random_numbers.sh
   -- generate random numbers, maybe duplicated numbers.
      It is useful for mergesort and quicksort

gen_random_numbers_unique.sh
   -- generate random numbers, unique numbers.
      It is useful for mergesort and quicksort.

verify_sorted_numbers.sh
   -- verify the numbers are sorted and the total number is correct.
      It is useful for mergesort and quicksort.

e.h   -- for baseline programs like threshold, blur and edge

3. Compiling a Fork program
   
Set the paths in file common.make correctly.
Then go to any Fork program directory.
Compile the program by command ‘make’.
Appendix B

Implementation Details

B.1 Compiling Flags

```java
options.add(options.TRANSFORM, "fork−baseline", 
    "Perform Fork to baseline translation");
options.add(options.TRANSFORM, 
    "disable−fork−group−creation−functioncall", 
    "Disable Fork group creation in function call to synchronous function");
options.add(options.TRANSFORM, 
    "Ofork−global−heap−shmalloc", 
    "Optimize Fork global heap if shmalloc does not present");
```

Listing B.1: Compiling flags of Fork-to-baseline compiler (Driver.java in Cetus)

B.2 Supporting Libraries

B.2.1 Shared Local Variable Allocation

```c
#define NEW_SH_LOCAL_VAR(spec, name, size) \
    shared_stack = shared_stack - (size);\n    spec name = (spec)shared_stack;
```

Listing B.2: Shared Local Variable Allocation
B.2.2 Synchronization

/* General synchronize function, depends on shared _group_id and _group_barrier */

* Assumptions:
  * 1. _group_id and _group_barrier are allocated and initialized to zero, see INIT_FORK_ENV
  * 2. thread 0 is executed first or last.
  * 3. If this assumption fails, inserted one ‘nop’
  * before the following code: OP0 1 ST0 O0,R3
* Note: _group_id and _group_barrier are re-inited in the function

* Code Explanation (pseudo code):
  * madd(&_group_id, 1);
  * while(_group_id != _number_of_threads);
  * if (_thread_id != 0) goto _LIB6_4;
  * _group_id=0;
  * _group_barrier=1;
  * _group_barrier = 0 and goto _LIB6_5;
  * _LIB6_4:
  * while(_group_barrier!=1);
  * nop;
  * _LIB6_5:
  * threads are synchronized here;

*/

void _group_sync()
{
    asm("OP0 -8 ADD0 R29,00 ST0 R3,A0 WB29 A0 \n
        OP0 _group_barrier ADD0 R32,00 LD0 A0 WB3 M0 \n
        OP0 _group_id ADD0 R32,00 LD0 A0 WB2 M0 \n
        OP0 _number_of_threads ADD0 O0,R32 WB1 A0 \n
        LD0 R1 WB1 M0 \n
        OP0 1 MADD0 O0,R2 \n
        _LIB6_1; while.cond \n
        OP1 _LIB6_1 LD0 R2 SNE M0,R1 BNEZ O1 \n
        _LIB#2: ; while.end \n
        OP0 _thread_id OP1 _LIB6_4 ADD0 O0,R32 LD0 A0 SNE M0,R0 BNEZ O1 WB1 \n
        M0 \n
        _LIB#3: ; if .then \n
        ST0 R0,R2 ;reinit _group_id\n
        OP0 1 ST0 O0,R3 \n
        _LIB6_4; ; while.cond2 \n
        OP1 _LIB6_5 ST0 R0,R3 JMP O1 ;reinit _group_barrier\n
        _LIB6_5; ; sync all threads here \n
        LD0 R29 WB3 M0 \n
        OP0 8 ADD0 R29,00 LD0 A0 WB4 M0 \n
        OP0 8 ADD0 R29,00 WB29 A0 ");
}

Listing B.3: Synchronization
Explanation of the synchronization function

This function is designed for the REPLICA-T5-4-512+ architecture.

The goal of this function is to synchronize threads no matter what order they enter this function. The comments at the beginning of the function provide a pseudo code explanation. This function has two loops: the loop at \_LIB6\_1 and the loop at \_LIB6\_4. The first loop synchronizes threads so that all the threads enter into this function but possibly executing different instructions. The second loop synchronizes threads so that they execute the same instruction after \_LIB6\_5.

If there are more memory units and chaining in the hardware, it is possible to simplify the synchronization function to use only one loop and remove the second loop. The \texttt{forklib\_sync} function in Fork implementation for SB-PRAM [28] has similar structure but uses the built-in “mode bit” of the SB-PRAM instead of the explicit “\_group\_barrier” variable.

B.2.3 INIT\_FORK\_ENV

```c
/* Setup frame at the entry of main() */
#define INIT_FORK_ENV do {
    RESERVE\_GLOBAL\_HEAP
    ALLOC\_GFS\_FRAME
    INIT\_GFS\_FRAME
    INIT\_GRID
} while(0)
```

Listing B.4: INIT\_FORK\_ENV
Appendix C

Test Programs

C.1 Threshold

```c
#include "../e.h"
#define SIZE 307200

struct pixel {
    unsigned char r, g, b;
};

struct image {
    char meta[54];
    struct pixel data[SIZE];
};

struct image inImage;
struct image outImage;
unsigned int sum_in = 0;
unsigned int sum = 0;

int _sync_ = 0;
void sync()
{
    asm("OP0 1 MADD0 O0,%0" : : "r"(&_sync_) :);
    while (_sync_ != _number_of_threads) {}
    asm("OP0 −1 MADD0 O0,%0" : "r"(&_sync_) :);
}

int main()
{
    unsigned int i;
    unsigned int psum = 0;

    _start_timer;
```

if (_thread_id == 0) {
    for (i = 0; i < 54; i++) {
        outImage_meta[i] = inImage_meta[i];
    }
}
sync();

for (i = _thread_id; i < SIZE; i += _number_of_threads) {
    sum = inImage_data[i].r + inImage_data[i].g + inImage_data[i].b;
    asm("MADD0 %0,%1": : "r"(sum), "r"(&sum))
}
sync();

sum = sum / SIZE;

for (i = _thread_id; i < SIZE; i += _number_of_threads) {
    psum = inImage_data[i].r + inImage_data[i].g + inImage_data[i].b;
    if (sum > psum) {
        outImage_data[i].r = outImage_data[i].g = outImage_data[i].b = 0;
    } else {
        outImage_data[i].r = outImage_data[i].g = outImage_data[i].b = 255;
    }
    sync();
    stop_timer;
    exit;
    return 0;
}

Listing C.1: The original threshold.c program

#include "fork.h"
#include "string.h"

#define SIZE 307200

struct pixel {
    unsigned char r, g, b;
};

struct image {
    char meta[54];
    struct pixel data[SIZE];
};

struct image inImage;
volatile struct image outImage;
volatile int sum = 0;
volatile int tsum = 0;

int main() {
unsigned int i;
unsigned int psum = 0;

_Start_timer(
    start {
        seq memcpy((void*) &outImage.meta[0], (void*) &inImage.meta[0], 54);
        farm {
            for (i = _PROC_NR_; i < SIZE; i += _STARTED_PROCS_)
            {
                tsum = inImage.data[i].r + inImage.data[i].g + inImage.data[i].b;
                syncadd(&sum, tsum);
            }
        }
        tsum = sum / SIZE;
        farm {
            for (i = _PROC_NR_; i < SIZE; i += _STARTED_PROCS_)
            {
                psum = inImage.data[i].r + inImage.data[i].g + inImage.data[i].b;
                if (tsum > psum)
                    outImage.data[i].r = outImage.data[i].g = outImage.data[i].b = 0;
                else
                    outImage.data[i].r = outImage.data[i].g = outImage.data[i].b = 255;
            }
        }
    }
)_stop_timer;

return 0;

Listing C.2: The threshold.c program re-written in Fork using memcpy

C.2 Blur

#include "../e.h"

#define XSIZE 640
#define YSIZE 480
#define SIZE 307200

struct pixel {
    unsigned char r, g, b;
};

struct image {
    char meta[54];
    struct pixel data[SIZE];
};

struct image inImage;
struct image outImage;
unsigned int blur_size = 3;

int _sync_ = 0;
void sync()
{
    asm("OP0 1 MADD0 O0,%0" : "r"(&_sync_));
    while (_sync_ != _number_of_threads) {}
    asm("OP0 \-1 MADD0 O0,%0" : "r"(&_sync_));
}

void blur(const unsigned int i)
{
    int j;
    int x = i % XSIZE;
    int y = i / XSIZE;
    int xt;
    unsigned int r = 0;
    unsigned int g = 0;
    unsigned int b = 0;

    r += inImage.data[i].r;
    g += inImage.data[i].g;
    b += inImage.data[i].b;

    for (j = 1; j <= blur_size; j++)
    {
        xt = x - j;
        yt = y; 
        if (xt >= 0 && xt < XSIZE)
        {
            r += inImage.data[xt + yt*XSIZE].r;
            g += inImage.data[xt + yt*XSIZE].g;
            b += inImage.data[xt + yt*XSIZE].b;
        }
        xt = x + j;
        if (xt >= 0 && xt < XSIZE)
        {
            r += inImage.data[xt + yt*XSIZE].r;
            g += inImage.data[xt + yt*XSIZE].g;
            b += inImage.data[xt + yt*XSIZE].b;
        }
        xt = x;
        yt = y - j;
        if (yt >= 0 && yt < YSIZE)
        {
            r += inImage.data[xt + yt*XSIZE].r;
            g += inImage.data[xt + yt*XSIZE].g;
            b += inImage.data[xt + yt*XSIZE].b;
        }
        yt = y + j;
        if (yt >= 0 && yt < YSIZE)
        {
            r += inImage.data[xt + yt*XSIZE].r;
            g += inImage.data[xt + yt*XSIZE].g;
        }
    }
b += inImage.data[xt + yt*XSIZE].b;
}
}

j = 4*blur_size + 1;
outImage.data[i].r = r / j;
outImage.data[i].g = g / j;
outImage.data[i].b = b / j;
}

int main()
{
    unsigned int i;
    unsigned int psum = 0;

    sync();
    _start_timer;

    if (_thread_id == 0)
    {
        for (i = 0; i < 54; i++)
        {
            outImage.meta[i] = inImage.meta[i];
        }
    }
    sync();

    for (i = _thread_id; i < SIZE; i += _number_of_threads)
    {
        blur(i);
    }
    sync();
    _stop_timer;
    _exit;
    return 0;
}

Listing C.3: The original blur.c program

#include <fork.h>

#define XSIZE 640
#define YSIZE 480
#define SIZE 307200

struct pixel {
    unsigned char r, g, b;
};

struct image {
    char meta[54];
    struct pixel data[SIZE];
};
sh struct image inImage;
sh volatile struct image outImage;
sh unsigned int blur_size = 3;

async void blur(const unsigned int i) {
    int x = i % XSIZE;
    int y = i / XSIZE;
    int xt, yt;
    unsigned int r = 0;
    unsigned int g = 0;
    unsigned int b = 0;

    r += inImage.data[i].r;
    g += inImage.data[i].g;
    b += inImage.data[i].b;

    for (j = 1; j <= blur_size; j++) {
        xt = x - j;
        yt = y;
        if (xt >= 0 && xt < XSIZE) {
            r += inImage.data[xt + yt * XSIZE].r;
            g += inImage.data[xt + yt * XSIZE].g;
            b += inImage.data[xt + yt * XSIZE].b;
        }
        xt = x + j;
        if (xt >= 0 && xt < XSIZE) {
            r += inImage.data[xt + yt * XSIZE].r;
            g += inImage.data[xt + yt * XSIZE].g;
            b += inImage.data[xt + yt * XSIZE].b;
        }
        xt = x;
        yt = y - j;
        if (yt >= 0 && yt < YSIZE) {
            r += inImage.data[xt + yt * XSIZE].r;
            g += inImage.data[xt + yt * XSIZE].g;
            b += inImage.data[xt + yt * XSIZE].b;
        }
        yt = y + j;
        if (yt >= 0 && yt < YSIZE) {
            r += inImage.data[xt + yt * XSIZE].r;
            g += inImage.data[xt + yt * XSIZE].g;
            b += inImage.data[xt + yt * XSIZE].b;
        }
    }
    j = 4 * blur_size + 1;

    outImage.data[i].r = r / j;
    outImage.data[i].g = g / j;
    outImage.data[i].b = b / j;
}

int main() {
    unsigned int i;
C.3 Edge

Listing C.4: The blur.c program in Fork language

```c
start {
  start_timer;
  seq {
    for (i = 0; i < 54; i++)
      { outImage.meta[i] = inImage.meta[i]; }
  }
  farm {
    for (i = _PROC_NR_; i < SIZE; i += _STARTED_PROCS_)
      blur(i);
  }
  stop_timer;
}
return 0;
}
```

C.3 Edge

```c
#include "fork_replica.h"
#include "sync.h"
#define XSIZE 640
#define YSIZE 480
#define SIZE 307200
#define HEADER 54

struct pixel {
  unsigned char r, g, b;
};

struct image {
  char meta[HEADER];
  struct pixel data[SIZE];
};

struct image inImage;
struct image greyImage;
// volatile */ struct image outImage; // Change back when compile for REPLICA?
const unsigned int blur_size = 3;

int quick_mask[3][3] = {{−1, 0, −1}, {0, 4, 0}, {−1, 0, −1}};

void edge(const unsigned int i) {
  int j;
  int k;
  int x = i % XSIZE;
  int y = i / XSIZE;
  int xt;
  int yt;
  int r = 0;
```
int g = 0;
int b = 0;
// Threshold value might need some tuning
int const THRESHOLD = 120;

for (j = -1; j <= 1; j++)
    for (k = -1; k <= 1; k++)
    { xt = x + j;
      yt = y + k;
      if ((xt + yt * XSIZE) < SIZE && ((xt + yt * XSIZE) > 0))
      { int inr, in_g, in_b;
        inr = inImage.data[xt + yt * XSIZE].r;
        in_g = inImage.data[xt + yt * XSIZE].g;
        in_b = inImage.data[xt + yt * XSIZE].b;

        r += inImage.data[xt + yt * XSIZE].r * quick_mask[j+1][k+1];
      }
    }

if(r > THRESHOLD)
{ outImage.data[i]. r = 255; //r;
  outImage.data[i]. g = 255; //g;
  outImage.data[i]. b = 255; //b;
}
if(r < 0)
{ outImage.data[i]. r = 0; //r;
  outImage.data[i]. g = 0; //g;
  outImage.data[i]. b = 0; //b;
}

int sync_ = 0;
void sync()
{ asm("OP0 1 MADD0 O0,%0" : : "r"(&sync_) :);
  while (sync_ != number_of_threads) {} 
  asm("OP0 -1 MADD0 O0,500" : "r"(&sync_) :);
}

int main()
{ int i;

  if (_thread_id == 0)
    for(i=0; i < HEADER; i++)
      outImage_meta[i]=inImage_meta[i];
  for (i = _thread_id; i < SIZE; i += _number_of_threads)
    edge(i);

  sync();
  exit();
}
Listing C.5: The original edge.c program

```c
#define XSIZE 640
#define YSIZE 480
#define SIZE 307200
#define HEADER 54

struct pixel {
    unsigned char r, g, b;
};

struct image {
    char meta[HEADER];
    struct pixel data[SIZE];
};

struct image inImage;
struct image greyImage;
/* volatile */ struct image outImage; // Change back when compile for REPLICA?
const unsigned int blur_size = 3;

const unsigned int blur[3][3] = {{-1, 0, -1}, {0, 4, 0}, {-1, 0, -1}};

async void edge(const unsigned int i) {
    int j;
    int k;
    int x = i % XSIZE;
    int y = i / XSIZE;
    int xt;
    int yt;
    int r = 0;
    int g = 0;
    int b = 0;
    //Threshold value might need some tuning
    int const THRESHOLD = 120;
    for (j = -1; j <= 1; j++) {
        for (k = -1; k <= 1; k++) {
            xt = x + j;
            yt = y + k;
            if ((xt + yt * XSIZE) < SIZE && ((xt + yt * XSIZE) > 0)) {
                int in_r, in_g, in_b;
                in_r = inImage.data[xt + yt * XSIZE].r;
                in_g = inImage.data[xt + yt * XSIZE].g;
                in_b = inImage.data[xt + yt * XSIZE].b;

                //Check if we have greyscale picture...
                r += inImage.data[xt + yt * XSIZE].r * quick_mask[j+1][k+1];
            }
        }
    }
    if (r > THRESHOLD)
    {
    }
```

C.3. Edge
Appendix C. Test Programs

outImage.data[i]. r = 255; //r;
outImage.data[i]. g = 255; //g;
outImage.data[i]. b = 255; //b;
}
if(r < 0)
{
  outImage.data[i]. r = 0; //r;
  outImage.data[i]. g = 0; //g;
  outImage.data[i]. b = 0; //b;
}

int main()
{
  int i;

  start {
    seq
      for(i=0; i < HEADER; i++)
        outImage.meta[i]=inImage.meta[i];
    
    farm {
      for (i = thread_id; i < SIZE; i += _number_of_threads)
        edge(i);
    }
  }
}

Listing C.6: The edge.c program re-written in Fork

C.4 Algorithms in Fork language

C.4.1 Mergesort

The example program of mergesort based on mergesort.c in the Fork implementation for SB-PRAM.

/* Parallel Mergesort.  C.W. Kessler 10/97
 * sorts N elements using p processors, synchronous version.
 * Assumes that all elements are different;
 * otherwise the result will be wrong.
 */
#include <fork.h>
#include <assert.h>
#include <io.h>
#include <stdlib.h>
#include <string.h>

#define THRESHOLD 1
// #define NITEM 512

sh int N=NITEM; /* the number of array elements to be sorted */
sh int c[NITEM]; /* depth of recursion of each thread */
```c
int inN[NITEM];
int outN[NITEM];

int simple_lock screen = 0; /* screen output is a critical section */

void print_array(int *a, int n) {
    int i;
    simple_lockup(&screen);
    printf("Array %p of size %d:
", a, n);
    for (i=0; i<n; i++) printf(" %d", a[i]);
    printf("\n");
    simple_unlock(&screen);
}

int compare(void *a, void *b) {
    if ((int*)a < (int*)b) return -1;
    if ((int*)a > (int*)b) return 1;
    return 0;
}

void seq_sort(int *data, int n, int *presult) {
    memcpy(presult, data, n * sizeof(int));
    qsort(presult, n, sizeof(int), compare);
    //qsort(data, n, sizeof(int), compare);
}

void seqsort(int *array, int n, int *outarray) {
    int i, j, temp;
    for (i=0; i<n; i++) outarray[i] = array[i];
    for (i=0; i<n-1; i++)
        for (j=i+1; j<n; j++)
            if (outarray[i] > outarray[j]) {
                /* swap */
                temp = outarray[i];
                outarray[i] = outarray[j];
                outarray[j] = temp;
            }
}

int get_rank(int key, int *array, int n) {
    int left = 0;
    int right = n-1;
    int mid;
```
if (key > array[n−1]) return n;
    //if (key == array[n−1]) return n−1;
if (key <= array[0]) return 0;
while (left < right−1) {
    /*binary searches*/
    /∗always maintain array[left] <= key < array[right]*/
    mid = (right+left)/2;
    if (key < array[mid]) right = mid;
    else left = mid;
}
if (key==array[left]) return left;
else return left+1;
}

/* merge array src1 of size n1 and src2 of size n2
into one array dest of size n1+n2.
* Assertions: p>1, n1*n2>=1, dest + temp array is allocated. */
sync void merge( sh int *src1, sh int n1,
    sh int *src2, sh int n2,
    sh int *dest, sh int *temp)
{
    sh int iter;
    sh int *rank12 = temp;
    sh int *rank21 = temp + n1; /*temp. rank arrays*/
    int i;
    //farm assert(#);
    //farm pprintf(" merge( src1=%p, n1=%d, src2=%p, n2=%d, dest=%p, n=%d, p=%d)\n";
    //src1,n1,src2,n2,dest,n1+n2,#);
    iter = 0;
    farm
    /* self-scheduling par. loop over rank computations: */
    FORALL( i, &iter, 0, n1, 1 )
        rank12[i] = get_rank( src1[i], src2, n2 );
    iter = 0;
    farm
    FORALL( i, &iter, 0, n2, 1 )
        rank21[i] = get_rank( src2[i], src1, n1 );
    farm
    /* copy elements to dest using the rank information */
    forall (i,0,n1,#) dest[i+rank12[i]] = src1[i];
    forall (i,0,n2,#) dest[i+rank21[i]] = src2[i];
}

/* mergesort for an array of size n. */
* The sorted array is to be stored in
* sortedarray which is assumed to be allocated.
* temp is an allocated temporary buffer of length n.
*/
sync void mergesort( sh int *array, sh int n, sh int *sortedarray, sh int *temp )
{
    //farm pprintf(" mergesort( array=%p, n=%d, p=%d, $s=%d)/n", array, n, #, $s);
    c[absolute_thread_id]++;
}
if (n < THRESHOLD) {
    seq seq_sort( array, n, sortedarray );
    return;
}

if (# == 1) {
    farm seq_sort( array, n, sortedarray );
    return;
}

fork (2; @ =$@%2;)
   mergesort( array + @*(n/2), (1-@)*((n/2) + @*(n-n/2), temp + @*(n/2),
            sortedarray + @*(n/2));
merge( temp, n/2, temp+n/2, n-n/2, sortedarray, array );
}

void main( void )
{
    int j;
    start {
        sh int *temp;
        // initTracing(100000);
        // seq {
        // printf("Enter N = ");
        // scanf("%d", &N);
        // }  
        // a = (int *) shalloc( N * sizeof(int) );
        // b = (int *) shalloc( N * sizeof(int) );
        temp = (int *) shalloc( N * sizeof(int) );
        // seq {
        //     for (j=0; j<N; j++)
        //     a[j] = rand()%8192; /* set arrays */
        // }  
        // seq print_array( a, N );
        // startTracing();
        mergesort(inN, N, outN, temp);
        // stopTracing();
        // seq print_array( b, N );
        // writeTraceFile( "mergesort.trv", "mergesort" );
    }
    exit(0);
}

Listing C.7: The Fork program of mergesort

The translated baseline program of mergesort.

#include <fork_replica.h>

/*
Parallel Mergesort. C.W. Kessler 1097
* sorts N elements using p processors, synchronous version.
* Assumes that all elements are different;
* otherwise the result will be wrong.
*/

#include <fork.h>
#include <assert.h>
#include <io.h>
#include <stdlib.h>
#include <string.h>

/*
#define NITEM 512 */
int N; = NITEM;
/* the number of array elements to be sorted */
int c[NITEM];
/* depth of recursion of each thread */
inN[NITEM];
outN[NITEM];
/* screen output is a critical section */
/* print an array a of size n sequentially */

void print_array(int *a, int n)
{
    int i;
    simple_lockup(( & screen_));
    printf("Array %p of size %d:
", a, n);
    for (i=0; i<n; i++)
    {
        printf(" %d", a[i]);
    }
    printf("\n");
    ( * ( & screen_))=0;
    WORKAROUND_BUG_R31;
}

int compare(void *a, void *b)
{
    if ((( * ((int*)a))<(( * ((int*)b))))
    {
        return (-1);
    }
    if ((( * ((int*)a))>( * ((int*)b))))
    {
        return 1;
    }
    return 0;
    WORKAROUND_BUG_R31;
}

void seq_sort(int *data, int n, int *result)
{
    memcpy(result, data, (n*sizeof(int)));
    qsort(result, n, sizeof(int), compare);
    /* qsort(data, n, sizeof(int), compare); */
    WORKAROUND_BUG_R31;
}

/*
Alternative: sequential sorting, using bubble sort.
Assumes that outarray is allocated.
*/
C.4. Algorithms in Fork language

```c
void seqsort(int *array, int n, int *outarray)
{
    int i, j;
    int temp;
    for (i=0; i<n; i++)
    {
        outarray[i]=array[i];
    }
    for (i=0; i<(n-1); i++)
    {
        for (j=(i+1); j<n; j++)
        {
            if ((outarray[i]>outarray[j]))
            {
                /* swap */
                temp=outarray[i];
                outarray[i]=outarray[j];
                outarray[j]=temp;
            }
        }
    }
    WORKAROUND_BUG_R31;
}

/*
 * in sequential compute the rank of key within
 * array of size n, i.e. number_of_threads array_elements < key
 */
int get_rank(int key, int *array, int n)
{
    int left = 0;
    int right = (n-1);
    int mid;
    if ((key>array[(n-1)]))
    {
        return n;
    }
    /* if (key == array[n-1]) return n-1; */
    if ((key<=array[0]))
    {
        return 0;
    }
    while (left<(right-1))
    {
        /* binary search */
        /* always maintain array[left] <= key < array[right] */
        mid=((right+left)/2);
        if ((key<array[mid]))
        {
            right=mid;
        }
        else
        {
            left=mid;
        }
    }
}
```
if ((key==array[left]))
{
    return left;
}
else
{
    return (left+1);
}
WORKAROUND_BUG_R31;

merge array src1 of size n1 and src2 of size n2
into one array dest of size n1+n2.
* Assertions: p>1, n1*n2>=1, dest + temp array is allocated.

*/

void merge(int * src1_, int n1_, int * src2_, int n2_, int * dest_, int * temp_)
{
    NEW_SH_LOCAL_VAR(int*, iter_, sizeof (int));
    NEW_SH_LOCAL_VAR(int* *, rank12_, sizeof (int * ));
    ( * rank12_)=temp_;
    NEW_SH_LOCAL_VAR(int* *, rank21_, sizeof (int * ));
    ( * rank21_)=(temp_+n1_);
    /∗ temp. rank arrays ∗/
    int i;
    /∗ farm assert(number_of_threads); ∗/
    /∗ farm printf(“merge( src1=%p, n1=%d, src2=%p, n2=%d, dest=%p, n=%d, p
    =$d)\n”, ∗
    src1,n1,src2,n2,dest,n1+n2,number_of_threads); ∗/
    ( * iter_)=0;
    /∗ self−scheduling par. loop over rank computations: ∗/
    {
        /∗ farm ∗/
        {
            for ((( * iter_)=0), barrier (), (i=mpadd(( & (∗ iter_)), 1))); i<n1_; i=
            mpadd(( & (∗ iter_)), 1))
            {
                (∗ rank12_)[i]=get_rank(src1_[i], src2_, n2_);
            }
            synchronize();
        } ( * iter_)=0;
        {
            /∗ farm ∗/
            {
                for ((( * iter_)=0), barrier (), (i=mpadd(( & (∗ iter_)), 1))); i<n2_; i=
                    mpadd(( & (∗ iter_)), 1))
                {
                    (∗ rank21_)[i]=get_rank(src2_[i], src1_, n1_);
                }
                synchronize();
            }
            /∗ farm ∗/
        }
    }
C.4. Algorithms in Fork language

```c
{ /* copy elements to dest using the rank information */
  for (i=(thread_id+0); i<n1; i+=number_of_threads)
    { dest_[(i+(*rank12)[i])] = src1_[i]; }
  for (i=(thread_id+0); i<n2; i+=number_of_threads)
    { dest_[(i+(*rank21)[i])] = src2_[i]; }
  } 
  _synchronize(); 
  WORKAROUND_BUG_R31; 
} /*
  * mergesort for an array of size n.
  * The sorted array is to be stored in
  * sortedarray which is assumed to be allocated.
  * temp is an allocated temporary buffer of length n.
  */
void mergesort(int *array, int n, int *sortedarray, int *temp) 
{ /* farm pprintf( " mergesort( array=%p, n=%d, p=%d, $$
   \_number_of_threads, $$\) ; */
  c_[absolute_thread_id] ++ ;
  if ((n<=1))
    { 
      /* seq */
      if ((thread_id==0))
        { seq_sort(array, n, sortedarray); }
      _synchronize();
      return ; 
    }
  if ((number_of_threads==1))
    { BEGIN_GROUP; 
      { 
        /* farm */
        seq_sort(array, n, sortedarray); 
      } 
      _synchronize();
      END_GROUP; 
      return ; 
    }
  END_GROUP; 
} 
```
Appendix C. Test Programs

/* fork( 2; _group_no=(thread_id%2);... ) */
SAVE_GROUP_NO;
_group_no=(thread_id%2);
if ((_group_no<2))
{
    NEW_FORK_GROUPS(2, _group_no);
}

{ SAVE_GRID;
    mergesort((array+( _group_no*(n/2) )), ((1−_group_no)*(n/2))+( _group_no*(n−(n/2)) )),(temp_+( _group_no*(n/2) )),(sortarray_+( _group_no*(n/2) )));
    RESTORE_GRID;
}
} END_FORK_GROUPS;
{synchronize() ;
    RESTORE_GROUP_NO;
}
{ SAVE_GRID;
    merge(temp_−(n/2), (temp_+(n/2)), (n−(n/2)), sortarray_−, array_);
    RESTORE_GRID;
} WORKAROUND_BUG_R31;

int main(void )
{
    int j;
    INIT_FORK_ENV_NO ,GLOBAL_HEAP;
    {
    /* start */
    synchronize();
    { SAVE_GRID;
        NEW_SH_LOCAL_VAR(int *, temp_, sizeof (int *) );
        /* initTracing(100000); */
        /* seq { */
        /*   printf("Enter N = "); */
        /*   scanf("%d", &N); */
        /* } */
        /* a = (int) shalloc( N * sizeof(int) ); */
        /* b = (int) shalloc( N * sizeof(int) ); */
        /* ( * temp_)=((int *)shalloc((N_*sizeof(int) )));
        /* seq { */
        /*   for (j=0; j<N; j++) */
        /*   a[j] = rand()%8192; set array */
        /* } */
        /* seq print_array( a, N ); */
        /* startTracing(); */
        { SAVE_GRID;
            mergesort(inN_, N, outN_, ( * temp_));
            RESTORE_GRID;
        }
        /* stopTracing(); */
C.4. Algorithms in Fork language

C.4.2 Quicksort

The example program of quicksort based on quicksort.c in the Fork implementation for SB-PRAM.

```c
#include <fork.h>
#include <assert.h>
#include <io.h>
#include <stdlib.h>
#include <math.h>

#define NITEM 512

int N=NITEM; /* the number of array elements to be sorted */
int c[2048]; /* depth of recursion of each thread */
int gright = 0;
int gnpr = 0;
int *inN[NITEM];
int *outN[NITEM];

async int cmp( void *a, void *b )
{
    if (*((int *)a) < *((int *)b)) return -1;
    else if (*((int *)a) > *((int *)b)) return 1;
    else return 0;
}

async void qs( sh int *array, sh int n )
{
    sh int subn[3]; /* size of subarrays */
    sh int *subarray[3]; /* subarrays to be recursively sorted */
    sh int *subindex = (int *) shalloc(n*sizeof(int)); /* temporary storage*/
    sh int *subarrayindex = (int *) shalloc(n*sizeof(int)); /* temporary storage*/
    sh int numofprocsfor0;
}
```

Listing C.8: The translated baseline program of mergesort
Appendix C. Test Programs

```c
int pivot;
int p = 0;
pr int j;

c[ absolute_thread_id ] ++;
// seq pprintf(" qs(%d,%d)/n", n, p);
$=mpadd( &p, 1); /* renumber $ and compute p */

if (n<=1) return; /* trivial */
if (n==2) {
  /* simple: */
  if (array[0]>array[1]) { /* swap: */
    pivot=array[0]; array[0]=array[1]; array[1]=pivot;
  }
  return;
}

if (p==1) {
  /* sequential computation */
  seqsort( array, n ); /*
  farm qsort( array, n, 1, cmp );
  return;
  }

pivot = array[0];

farm
  for (j=$; j < n; j+=p) { /*in parallel select new subgroups for array elements*/
    if (array[j]<pivot)
      { subarrayindex[j]=0; subindex[j]=mpadd(&subn[0],1); }
    else if (array[j]==pivot)
      { subarrayindex[j]=1; subindex[j]=mpadd(&subn[1],1); }
    else
      { subarrayindex[j]=2; subindex[j]=mpadd(&subn[2],1); }
  }
/* now subn[k] holds the number of elements to be copied to subarray k */

  /* allocate subarrays and copy elements to them in parallel: */
  subarray[0] = (int *) shalloc( subn[0]+sizeof(int) );
  subarray[1] = (int *) shalloc( subn[1]+sizeof(int) );
  subarray[2] = (int *) shalloc( subn[2]+sizeof(int) );

  farm
    for (j=$; j < n; j+=p)
      subarray[subarrayindex[j]] [subindex[j]] = array[j];

if (subn[0]>1 && subn[2]>1) { /* the general case */
  int right;
  numofprocsfor0 = (subn[0]+p)/(subn[0]+subn[2]);
  if (!numofprocsfor0) numofprocsfor0 = 1; /*correction*/
  if (numofprocsfor0==p) numofprocsfor0 = p-1; /*correction*/
  farm if ($<numofprocsfor0) right = 0;
    else right = 1;
  fork ( 2; 0=right; $=$)
    qs( subarray[2*0], subn[2*0] );

  } else
```
C.4. Algorithms in Fork language

```c
if (subn[0] > 1) qs(subarray[0], subn[0]);
else if (subn[2] > 1) qs(subarray[2], subn[2]); /* else do nothing */
/* now concatenate sorted subarrays: */
for (j = 0; j < subn[0]; j += p)
    array[j] = subarray[0][j];
for (j = 0; j < subn[1]; j += p)
    array[j + subn[0]] = subarray[1][j];
for (j = 0; j < subn[2]; j += p)
    array[j + subn[0] + subn[1]] = subarray[2][j];
}
}

void main(void)
{
    int j;
    unsigned int startime, stoptime;
    N = NITEM;
    _start_timer;
    if (T < NTHREADS) {
        qs(inN, N);
    }
    if (T == 0) {
        for (j = 0; j < N; j++) outN[j] = inN[j];
    }
    _stop_timer;
}
```

Listing C.9: The Fork program of quicksort

The translated baseline program of quicksort.

```c
#include <fork_replica.h>
/*
CRCW Quicksort using fork DC strategy. C.W. Kessler 1295
* sorts N elements using p processors
* in expected time O((N/p) log N) and space O(N log N).
* synchronous version.
*/
#include <fork.h>
#include <assert.h> *
#include <io.h>
#include <stdlib.h>
#include <math.h> *
#define NITEM 512 *
int N = NITEM;
/* the number of array elements to be sorted */
int c[1024];
/* depth of recursion of each thread */
int gright = 0;
```
# Appendix C. Test Programs

```c
int gapr = 0;
int inN[NITEM];
int outN[NITEM];
/
∗ compare function used by the sequential qsort() routine ∗/
int cmp(void * a, void * b) {
  if ((( * ((int *)a)) < ( * ((int *)b))))
    return (-1);
  else
    if ((( * ((int *)a)) > ( * ((int *)b))))
      return 1;
    else
      return 0;
}

WORKAROUND BUG R31;
/
∗ quicksort n elements using p processors in place ∗/
void qs(int * array, int n) {
  NEW_SH_LOCAL_VAR(int, subn, sizeof (int)*3);
  /* size of subarrays */
  NEW_SH_LOCAL_VAR(int *, subarray, sizeof (int *)*3);
  /* subarrays to be recursively sorted */
  NEW_SH_LOCAL_VAR(int *, subindex, sizeof (int *));
  ( * subindex)=((int *)shalloc((n*sizeof (int))));
  /* temporary storage */
  NEW_SH_LOCAL_VAR(int *, subarrayindex, sizeof (int *));
  ( * subarrayindex)=((int *)shalloc((n*sizeof (int))));
  /* temporary storage */
  NEW_SH_LOCAL_VAR(int *, numofprocsfor0, sizeof (int));
  NEW_SH_LOCAL_VAR(int *, pivot, sizeof (int));
  NEW_SH_LOCAL_VAR(int *, p, sizeof (int));
  ( * p)=0;
  int j;
  c[absolute_thread_id] ++ ;
  /* seq pprintf(" qs(%d,%d)\n", n,p); */
  _thread_grid=mpadd( & ( * p), 1);
  /* renumber $ and compute p */
  if ((n<=1))
    return ;
  /* trivial */
  if ((n==2))
    /* simple */
    if (((array[0]>array[1]))
      {
        /* swap */
```
C.4. Algorithms in Fork language

```
( * pivot_0)=array_0[0];
array_0[0]=array_0[1];
array_1[1]=( * pivot_1);
}
return ;
}
if ((( * p_0)==1))
{
  /* sequential computation */
  /* seqsort( array, n ); */
  /* farm qsort( array, n, 1, cmp ); */
  {
    /* farm */
    {
      qsort(array_, n_, 4, cmp);
    }
    synchronize();
    return ;
  }
subn_0[0]=subn_1[1]=(subn_2[2]=0);
( * pivot_0)=array_0[0];
{
  /* farm */
  {
    for (j=thread_grid; j<n; j+=(* p_0))
    {
      /* in parallel select new subgroups for array elements */
      if ((array_0[j]<(* pivot_0))
      {
        ( * subarrayindex_0)[j]=0;
        ( * subindex_0)[j]=mpadd(( & subn_0[0]), 1);
      }
    else
      
      if ((array_0[j]==(* pivot_0))
      {
        ( * subarrayindex_0)[j]=1;
        ( * subindex_0)[j]=mpadd(( & subn_0[1]), 1);
      }
    else
      
      ( * subarrayindex_0)[j]=2;
      ( * subindex_0)[j]=mpadd(( & subn_0[2]), 1);
    }
  }
  synchronize();
  /* now subn[k] holds the number of elements to be copied to subarray k */
  /* allocate subarrays and copy elements to them in parallel: */
  subarray_0=((int *)shalloc ((subn_0[0]+sizeof (int))));
  subarray_1=((int *)shalloc ((subn_1[1]+sizeof (int))));
  subarray_2=((int *)shalloc ((subn_2[2]+sizeof (int))));
  {
    /* farm */
```
{ 
  for (j=thread_grid; j<n; j+=(*p))
  { 
    subarray[(*subarrayindex)[j]][(*subindex)[j]]=array_[j];
  }
  _synchronize();
}
if (((subn[0]>1)&&(subn[2]>1))){
  /* the general case */
  int right;
  (*numofprocsfor0)=((subn[0]*(*p))/(subn[0]+subn[2]));
  if (! (*numofprocsfor0))
  { (*numofprocsfor0)=1;
  } /* correction */
  if (((*numofprocsfor0)==(*p))){
    (*numofprocsfor0)=((*p)-1);
  } /* correction */
  /* farm */
  if (((thread_grid<(*numofprocsfor0))))
  { 
    right=0;
  }
  else
  { 
    right=1;
  }
  _synchronize();
  { 
    /* fork( 2; _group_no=right;... ) */
    SAVE_GROUP_NO;
    _group_no=right;
    if ((_group_no<2))
    { 
      NEW_FORK_GROUPS(2,_group_no);
      thread_grid=_thread_grid;
      { 
        BEGIN_GROUP_NO_SYNC;
        SAVE_GRID;
        cs(subarray[2*_group_no], subn[(2*_group_no)]);
        _synchronize();
        RESTORE_GRID;
        END_GROUP;
      }
    }
  }
END_FORK_GROUPS;
C.4. Algorithms in Fork language

```c
} 
    synchronize();
    RESTORE_GROUP_NO;
},
else
{  
    if {((subn_[0]>1))}
    {
        BEGIN_GROUP_NO_SYNC;
        SAVE_GRID;
        qs(subarray_[0], subn_[0]);
        synchronize();
        RESTORE_GRID;
        END_GROUP;
    }
    else
    {  
        if {((subn_[2]>1))}
        {
            BEGIN_GROUP_NO_SYNC;
            SAVE_GRID;
            qs(subarray_[2], subn_[2]);
            synchronize();
            RESTORE_GRID;
            END_GROUP;
        }
    }
}  
/* else do nothing */
/* now concatenate sorted subarrays */
{
    /* farm */
    /* in parallel copy sorted subarrays into old array */
    for (j=thread_grid; j<subn_[0]; j+=(p_))
    {
        array_[j]=subarray_[0][j];
    }
    for (j=thread_grid; j<subn_[1]; j+=(p_))
    {
        array_[(j+subn_[0])]=subarray_[1][j];
    }
    for (j=thread_grid; j<subn_[2]; j+=(p_))
    {
        array_[((j+subn_[0])+subn_[1])]=subarray_[2][j];
    }
    synchronize();
    WORKAROUND_BUG_R31;
```
int main(void) {
    int j;
    unsigned int starttime, stoptime;
    INIT_FORK_ENV_NOGLOBAL_HEAP;
    N=NITEM;
    _start_timer ;
    { /* start */
        _synchronize();
        { SAVE_GRID;
        }
        if ((_thread_grid<NTHREADS))
        { BEGIN_GROUP;
        { BEGIN_GROUP_NO_SYNC;
            SAVE_GRID;
            qs(inN_, N_);
            _synchronize();
            RESTORE_GRID;
            END_GROUP;
        }
        END_GROUP;
        } _synchronize();
    }
    if ((_thread_grid==0))
    { for (j=0; j<N_; j ++ )
    { outN_[j]=inN_[j];
    }
    _stop_timer; /* sync and exit */
    _synchronize();
    _exit();
    }
}

Listing C.10: The translated baseline program of quicksort
This thesis describes the implementation of a source to source compiler that translates Fork language to REPLICA baseline language. The Fork language is a high-level programming language designed for the PRAM (Parallel Random Access Machine) model. The baseline language is a low-level parallel programming language for the REPLICA architecture which implements the PRAM computing model. To support the Fork language on REPLICA, a compiler that translates Fork to baseline is built. The Fork to baseline compiler is built in compatibility with the Fork implementation for SB-PRAM. Moreover, the libraries that support Fork’s features are built using baseline language. The evaluation result verifies that the features of the Fork language are supported in the implementation. The evaluation also shows the scalability of our implementation and shows that the overhead introduced by Fork-to-baseline translation is small.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/