LLVM COMPILER BACKEND FOR VHDL AND POWERPC AND APPLICATION TO DSP ALGORITHMS

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY

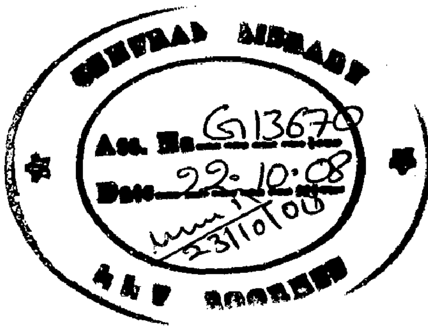
in

ELECTRICAL ENGINEERING

(With Specialization in Measurements and Instrumentation)

By

ADITYA VISHNUBHOTLA VIJAY



DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

ROORKEE - 247 667 (INDIA)

JUNE, 2008

I.D. NO.-M.T.

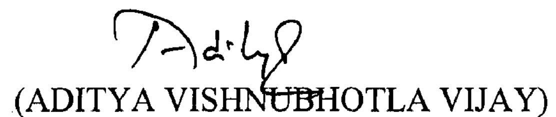
CANDIDATE'S DECLARATION

I hereby declare that the work presented in this dissertation entitled, "**LLVM Compiler Backend for VHDL and PowerPC and Application to DSP Algorithms**" submitted in the partial fulfillment of the requirements for the award of the degree of **Master of Technology** in **Electrical Engineering** with specialization in **Measurements and Instrumentation**, to the **Department of Electrical Engineering, IIT Roorkee**, Roorkee is an authentic record of my own work carried out during the period from September 2007 to June 2008 at TU Darmstadt, under the supervision of **Dr. Ing. Sorin A. Huss**, Professor, Integrated Circuits and Systems Lab, Department of Computer Science, TU Darmstadt and at IIT Roorkee, under the supervision of **Dr. H. K. Verma**, Professor, Department of Electrical Engineering, IIT Roorkee, Roorkee.

The matters embodied in this report have not been submitted by me for the award of any other degree or diploma.

Date: 30/6/08

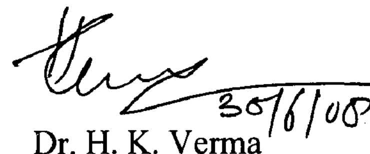
Place: Roorkee



(ADITYA VISHNUBHOTLA VIJAY)

CERTIFICATE

This is to certify that above statement made by the candidate is correct to the best of my knowledge.



Dr. H. K. Verma
Professor,
Electrical Engineering Department
Indian Institute of Technology, Roorkee
Roorkee-247667 (India)

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Dr.-Ing. S. A. Huss

Integrierte Schaltungen und Systeme

Fachbereich 20 Informatik
Fachbereich 18 Elektrotechnik

Hochschulstr. 10
64289 Darmstadt
Telefon (06151) 16-4882
Telefax (06151) 16-4810

e-mail: huss@iss.tu-darmstadt.de

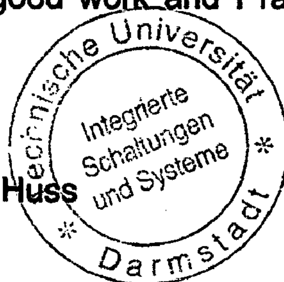
TUD · ISS · Hochschulstr. 10 · 64289 Darmstadt

Darmstadt, den 28. Mai 2008

Certificate

This is to certify that Mr. Aditya Vishnubhotla Vijay has successfully worked on the Master's thesis entitled "LLVM Compiler Backend for VHDL and PowerPC and Application to DSP Algorithms" under my supervision in the Integrated Circuits and Systems Laboratory, Department of Computer Science, Technische Universität, Darmstadt, Germany during the period of September 2007 to May 2008 under the DAAD - IIT Masters Sandwich Programme.

He did a remarkable good work and I rate therefore his thesis with1.3, which corresponds to A-.



Prof. Dr. –Ing. Sorin A. Huss

ACKNOWLEDGEMENT

I would like to express my deep sense of gratitude to my project guides *Prof. Dr. H.K. Verma* and *Prof. Dr. Ing. Sorin A. Huss*, without whom this dissertation could not have been written. I owe my most sincere gratitude to *Prof. Dr. H.K. Verma* for his advice and guidance. I gratefully acknowledge his whole-hearted encouragement and support. I acknowledge my obligation to *Prof. Dr. Ing. Sorin A. Huss* for having me in Integrated Circuits and Systems Group, which enriched my knowledge and kindled the aspiration to learn deeper.

I acknowledge my heartfelt obligation *Dipl. Ing. Tim Sander*, who not only mentored me but also challenged and encouraged me throughout the project work. It was he who opened the door to the beautiful world of codesign and compilers. I am indebted to him more than he knows.

My special thanks to *Dipl. Ing. Gregor Molter*, *Dipl. Ing. Felix Madelener*, *Frau Maria Tiedemann* and *Dr. Abdul Shoufan* who never failed to help me however small the matter may be. It was a pleasure working with them at Integrated Circuits and Systems Group.

I wish to express my warm and heartfelt gratitude to *Frau Kleinschmidt*, *Frau Ulla Nothnagel* for their loving support, which made me feel at home in a foreign land.

I am gratefully acknowledge the *German Academic Exchange Service (DAAD)* for providing the scholarship for my stay in Germany.

My loving thanks to my IIT Roorkee friends and my DAAD-IIT friends from Darmstadt who are my pillars of support.

Lastly, I owe my loving thanks to my parents *Mr. V V Sastry* and *Mrs. V Padmaja* and my sweet little brother *Mr. V M Charan* for their inseparable love, faith and persistant confidence in me.

ABSTRACT

Hardware/Software (HW/SW) codesign of an application is a popular design methodology employed to meet the performance needs under design costs and time-to-market constraints. It stands for the design of a system or application using both hardware and software components. The advantages of software based design are flexibility, low cost and less development time and the advantages of hardware based design are liability (backup) and performance. HW/SW codesign is employed to achieve an optimal system design taking the advantage of individual designs.

HW/SW partitioning is a primary step of HW/SW codesign which involves splitting of the application into hardware and software divisions based on the initial design constraints such as performance, costs and hardware area. Automatic HW/SW partitioning is the automatic identification and sectioning of the critical code segments of the application which are to be implemented in hardware. There are a few works which have already implemented automatic HW/SW partitioning using various tools and techniques, but the present partitioning work is based on Low Level Virtual Machine (LLVM) compiler framework and a novelty of its kind.

The LLVM is a compiler infrastructure designed to support life-long program analysis and transformation for arbitrary programs. LLVM provides tools and libraries used to build compilers, optimizers, code generators and many other compiler related programs.

The present work is aimed as an extension to the LLVM tool chain which automatically creates a HW/SW division of the input algorithm. The input algorithm is lowered by the LLVM compiler frontend into the language and target independent LLVM Intermediate Representation (LLVM IR) and the backend modules written, modify the LLVM IR to create the HW/SW partitioned output.

The backend modules comprise a Binning Pass and a Selection Pass. The LLVM Pass is a set of classes used to analyze, transform or output the LLVM IR. The Binning Pass is an analyze pass which employs heuristics to gather data flow and control flow statistics of the input algorithm written in C/C++ programming languages and empirical feature analysis is done to decide the parts of the algorithm to be implemented in HW/SW and the partitioning decision is taken after the above analysis.

The Selection Pass is a transform pass created to partition the input algorithm into HW/SW sections. The Selection Pass exploits the features collected by the Binning Pass to partition the algorithm for implementation in HW/SW. It modifies the LLVM IR of the input algorithm by replacing the set of the instructions which are not to be executed by the curent backend with intrinsic function calls. The above mentioned intrinsic instructions are defined as extensions to the LLVM language to keep the control and dataflows of the algorithm intact and are inserted at each context computing switch.

The Selection Pass generates the LLVM IR of the hardware and software sections. The hardware section is sent to the LLVM VHDL backend which transforms the LLVM IR into a VHDL RTL representation. The software section is sent to the LLVM PowerPC backend which transforms the LLVM IR into a PowerPC assembly language output. The backends are adapted to process the above mentioned intrinsic instructions. The work is based on the existing C/C++ frontends, VHDL and PowerPC backends of the compiler.

The above passes have been tested and validated using input algorithms from the Digital Signal Processing (DSP) domain.

Contents

1 Introduction 1

1.1 Overview 1

1.2 Motivation 2

1.3 Introduction to LLVM 2

1.3.1 The LLVM Compiler 3

1.3.2 LLVM Intermediate Representation 3

1.3.3 LLVM Type System 4

1.3.4 LLVM Pass 4

1.4 Description 5

2 Binning Pass 7

2.1 Data Flow and Control Flow in LLVM 7

2.1.1 Data Flow 7

2.1.2 Control flow 8

2.2 Basic Block Level Analysis 9

2.2.1 Example: Combinational implementation for constant data memory accesses 10

2.3 Function Level Analysis 12

2.4 Two Dimensional Discrete Fourier Transform Example 13

2.5 Binning Pass Results 15

3 Selection Pass 20

3.1 Description 20

3.2 Selection Pass Intrinsics 21

3.2.1 Intrinsic Functions in LLVM [6] 21

3.2.2 Selection Pass Intrinsics 22

3.3 Deps.h header file description . . . 23
3.4 Additions to Intrinsics.td file . . . 24
3.4.1 LLVM Intrinsics.td TableGen File [6] . . . 24
3.4.2 Additions to Intrinsics.td File . . . 26
3.5 Data Flow Dependencies . . . 28
3.6 Control Flow Dependencies . . . 29
3.7 Walsh Transform Example . . . 29
3.8 Selection Pass Results . . . 31

4 Adaptations to the PowerPC and VHDL Backends 33
4.1 LLVM Code Generator and Additions to SelectionDAGISel.cpp file . . . 33
4.1.1 LLVM Target-Independant Code Generator [6] . . . 33
4.1.2 Additions to SelectionDAGISel.cpp file . . . 35
4.2 Possible future extensions to SelectionDAGISel.cpp file . . . 37
4.3 PowerPC Backend Results . . . 37
4.4 VHDL Backend Results . . . 39
4.5 Hardware/Software Integration . . . 40

5 Conclusion 43
5.1 Summary . . . 43
5.2 Future Extensions . . . 44
5.2.1 Possible Extensions to the Binning Pass . . . 44
5.2.2 Possible Extensions to the Selection Pass . . . 44

References 47

Appendix A. Binning Pass Header File 49

Appendix B. Binning Pass 51

Appendix C. Selection Pass 58

Appendix D. Deps Header File 65

Appendix E. Two Dimensional DFT Program 70

Appendix F. Walsh-Hadamard Transform Program 72

List of Tables

2.1 Load Instruction Truth Table 10

2.2 Binning Pass basic block statistics 17

List of Figures

1.1 LLVM GCC Compiler Block Diagram [5] 3

1.2 Block Diagram 5

2.1 Load Instruction combinational circuit implementation 11

2.2 Data flow driven basic block 15

2.3 Control flow driven basic block 16

2.4 Control Flow Graph of twod_dft function 18

3.1 Data Dependencies 28

3.2 Modified basic block with intrinsics 31

3.3 Hardware and software sections 32

4.1 PowerPC Assembly Language Output 38

4.2 HW/SW Integration 41

Chapter 1

Introduction

1.1 Overview

The goal of the dissertation is to develop a tool for the Low Level Virtual Machine(LLVM) compiler suite, which automatically creates a Hardware/Software (HW/SW) division of the input algorithm. The input algorithm is lowered by the LLVM compiler frontend into the language and target independent LLVM Intermediate Representation(LLVM IR) and the backend modules written, modify the LLVM IR to create the HW/SW partitioned output. The work is based on the existing C/C++ frontends, VHDL and PowerPC backends of the compiler.

The results of the work are:

1. Binning Pass to collect Data/Control flow statistics

2. Selection Pass for Hardware/Software mapping

3. Adaptations to the VHDL backend

4. Adaptations to the PowerPC backend

5. Creation of example application

1.2 Motivation

The main motive of this work is to develop an extension to the LLVM tool chain for automatic HW/SW partitioning. HW/SW partitioning is a primary step and an integral part of HW/SW codesign process. HW/SW codesign is the design of a system or application using both hardware and software components. As mentioned in [1], the advantages of software based design are flexibility, low cost and less development time and the advantages of hardware based design are liability(backup) and performance. HW/SW codesign is employed to take into account the above mentioned advantages, to achieve an optimal system design. HW/SW partitioning is the splitting of the application into hardware and software divisions based on the initial design constraints such as performance, costs and hardware area. Automatic HW/SW partitioning is the automatic identification and sectioning of the critical code segments of the application which are to be implemented in hardware. There are a few works which have already implemented automatic HW/SW partitioning using various tools and techniques such as [2], [3] etc., but the present work is based on LLVM compiler framework and a novel of its kind.

1.3 Introduction to LLVM

The LLVM is a compiler framework designed to support life-long program analysis and transformation for arbitrary programs [4]. As mentioned in [5], the "LLVM is a compiler infrastructure that provides modular and reusable components to build compilers, optimizers, code generators and many other compiler related programs." A compiler is a translator which translates a program written in high level or source language into an equivalent program in machine level or object language. The traditional C compiler (GCC), Java Just-In-Time(JIT) compiler etc. are some examples of compilers.

1.3.1 The LLVM Compiler

The figure 1.1 below shows the workflow of a LLVM-toolsuite compiler run.

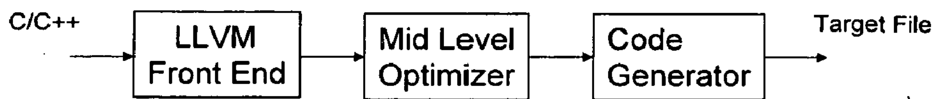


Figure 1.1: LLVM GCC Compiler Block Diagram [5]

The language specific front end lowers the code to LLVM IR. The LLVM IR is a well-defined intermediate representation of programs which provides for the capability of representing all high level languages. The LLVM includes, besides others, C and C++ front ends. The front ends comprise scanners, parsers and intermediate code generators. The scanner or lexical analyser recognises the patterns in the source code text and groups them into tokens and GCC parsers check the syntactic and semantic validity of the tokens. The LLVM disassembler converts the source code into LLVM IR.

LLVM is equipped with a mid-level optimiser which performs various standard scalar and loop optimisations. The link time is a natural place for interprocedural (cross functional) optimisations. The LLVM has static backends to generate code for X86, X86-64, PowerPC 32/64, ARM, SPARC and various other architectures.

1.3.2 LLVM Intermediate Representation

The LLVM defines a low-level code representation or Intermediate Representation (IR) in Static Single Assignment (SSA Form) which serves as a common code representation used throughout all phases of the LLVM compilation strategy. As mentioned in the [6], the LLVM code representation is designed to be used in three different forms all of which are equivalent. They are :

1. In-memory compiler IR
2. On-disk bitcode representation
3. Human readable assembly language representation

This allows LLVM to provide a intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations.

1.3.3 LLVM Type System

As mentioned in [7], the LLVM instruction set is fully typed, using a low-level, source language independant type system. The type system consists of primitive types with predefined sizes and derived types. The LLVM instructions have strict type rules and there are no mixed type operations. An explicit cast instruction is the sole manner to convert a value from one type to other type. As mentioned in [8], the LLVM type system lays a good foundation for aggressive optimisations.

1.3.4 LLVM Pass

LLVM Passes perform the transformations and optimizations that make up the compiler. The LLVM Pass is a set of classes used to analyse, transform or output the LLVM IR. As mentioned in [6], optimizations in LLVM are implemented as Passes that traverse some portion of a program to either collect information or transform the program. Passes in LLVM can depend on other passes etc. The Passes in LLVM are classified into three types:

1. Analysis Passes which compute information for usage in other passes
2. Transform Passes which modify the program
3. Utility Passes which provide some utility
4. Output passes which output the LLVM IR

The LLVM Passes are to be registered to obtain desired functionality. The present section on LLVM presents only a brief and relevant description. The citation [6], can be referred for a detailed reading on LLVM.

1.4 Description

The various stages of the proposed work are depicted in the following figure 1.2

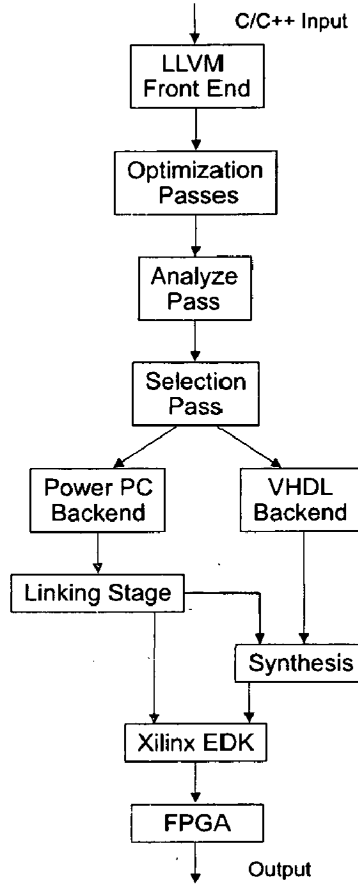


Figure 1.2: Block Diagram

The input algorithm is a code chunk in C or C++ programming languages. The input algorithm is lowered into language independant LLVM IR by the LLVM front end. A series of optimizations such as loop unrolling etc. are run on the algorithm with the help of pre-existing optimization passes.

The Binning pass which is an Analysis Pass 1 on the preceding pageis created to identify the sections of the algorithm to be sent to HW/SW. There are already various approaches for HW/SW partitioning like the Deterministic approach, Statistical approach and the usage of Profiling techniques [9]. But the present Pass extracts various features from the LLVM IR of the input algorithm. The features, for instance, include the memory accesses or different controlflow/dataflow features of the algorithm. Heuristics are created and empirical feature analysis is done to decide the parts of the algorithm to be implemented in HW/SW and the partitioning decision is taken after the above analysis.

The Selection Pass which is a Transform Pass 2 on page 4 is created to partition the input algorithm into HW/SW sections. The Selection Pass exploits the features collected by the Binning Pass to partition the algorithm for implementation in HW/SW. It modifies the LLVM IR of the input algorithm by replacing the set of the instructions which are not to be executed by the curent backend with intrinsic function calls. The above mentioned intrinsic instructions are defined as extensions to the LLVM language to keep the control and dataflows of the algorithm intact and are inserted at each context computing switch.

The Selection Pass generates the LLVM IR of the hardware and software sections. The hardware section is sent to the LLVM VHDL backend which transforms the LLVM IR into a VHDL RTL representation. The software section is sent to the LLVM PowerPC backend which transforms the LLVM IR into a PowerPC assembly language output. The backends are adapted to process the above mentioned intrinsic instructions.

The hardware(VHDL RTL) and software(PowerPC) sections are to be integrated using an existing bus interface[10] and a HW/SW interaction program written in C programming language. This program comprises the definitions for the above mentioned intrinsic functions is linked to the execution process in the linking stage. The program with the help of the bus interface

1. Moves the input data to hardware
2. Waits till the end of the instruction execution in the hardware
3. Retrieves the results obtained from hardware

The bus interface which is used for data transfer between HW/SW and the hardware section of the input algorithm are to be synthesized on Xilinx Virtex-II FPGA and the software section of the input algorithm is to be executed on the PowerPC microprocessor present on the same FPGA . The Xilinx Embedded Development Kit (EDK) is employed for the co-synthesis. Test examples from DSP domain are used to test the created passes.

Chapter 2

Binning Pass

The Binning Pass is an Analysis Pass which categorises the control flow driven and data flow driven sections of the LLVM IR of the test algorithm at basic block and function levels.

2.1 Data Flow and Control Flow in LLVM

LLVM instruction set provides explicit data flow and control flow information of the input algorithm which is explained in the subsequent sections.

2.1.1 Data Flow

Data flow is the movement of data or information through the sections of the algorithm. The sections of an algorithm are processed depending on the availability of data input. Data flow information can be gathered by setting up and solving systems of equations at various points within the program. As mentioned in the [11], a typical data flow equation has the form

$$out[S] = gen[S] \cup (in[S] - kill[S]) \text{ [11]}$$

and can be read as, " the information at the end of the end of a statement is either generated within the statement or nters at the beginning and is not killed as control flows through the statement."

As mentioned in the [6], LLVM uses a low-level object code representation in SSA form that uses simple RISC-like instructions, but provides rich, language-independent,

type information and dataflow information about operands. The Static Single Assignment is a form of instruction representation used by compilers, in which each variable is assigned exactly once during its life-time and PHI functions are included at the places where the program flow joins and the the value of the PHI function depends on the path taken by the program at run-time.

As mentioned in [7], SSA form is used in LLVM to handle data flow problems efficiently and because SSA form provides for explicit def-use chains. To represent SSA form directly in the code, LLVM uses an explicit phi instruction to merge values at control flow join points. The SSA form of LLVM IR and the usage of def-use and use-def chains provide the information regarding the data flowing in and out of a code segment(e.g. basic block or function). The type of operations(e.g. arithmetic or control flow) performed on the incoming data and the size of the code segment determine the amount of data processing being performed in the code segment.

Data flow reduces the dependency between the sections of code in an algorithm and provides for parallelization. Sections of the algorithm having heavy data flow indicate the computationally intensive nature of the algorithm and hence they are preferred to be executed in hardware.

2.1.2 Control flow

Control flow in general refers to the order in which the statements of an algorithm are executed. Control flow constructs are statements or instructions which when executed cause the subsequent flow of control to differ from the natural sequential order of execution. These are generally branching instructions, function calls and function return instructions.

As mentioned in [7], control flow instructions such as branch, multi-way branch or switch, function return, invoke or unwind instructions are grouped as terminator instructions in LLVM. LLVM also provides explicit Control Flow Graphs and hence the above features are considered to determine the extent of control flow in LLVM code segments.

The sections of the algorithm with complex control flow features, for instance with nested conditions and loops, when implemented as automatons lead to exponential increase in the number of states and eventually occupy huge space on hardware. Hence, control flow driven sections of the algorithm are preferred to be executed on microprocessors.

2.2 Basic Block Level Analysis

The Control flow features of the algorithm at basic block level are identified by the following control flow constructs:

1. Conditional and unconditional branching statements
2. Function calls
3. Nested conditions or loops
4. Terminator instructions

The Data flow features of the algorithm at basic block level are identified by the following data flow constructs:

1. Arithmetic instructions
2. Number of phi nodes
3. Number of instructions

Memory accesses in the algorithm play a significant role in determining the sections of the program to be executed in hardware and software. Memory access features at basic block level are identified by the following memory access constructs:

1. Memory allocation instructions
2. Instructions transferring data to memory
3. Instructions retrieving data from memory
4. Instructions clearing the allocated memory

Frequent memory accesses are inefficient for hardware implementation but accesses to fixed size memory locations storing constant values can be effectively implemented on hardware. using simple combinational circuits.The following simple combinational circuit implementation of a Load instruction, illustrates the above statement.

2.2.1 Example: Combinational implementation for constant data memory accesses

Let the fixed size global array of constant data stored in memory be {1, 5, 2, 8}. Let us consider a Load Instruction which retrieves a data element from the above mentioned array. The instruction can be implemented in hardware as follows

The values in memory represent the output. The biggest number is eight and therefore four output binary digits are sufficient to represent all the four output values. Two input binary digit address is sufficient to access the four values.

The following table 2.1 is the truth table for the circuit and below that are the output equations and figure 2.1 depicts the combinational circuit implementation of the load instruction.

A. Truth Table:

Table 2.1: Load Instruction Truth Table

x	y	f_1	f_2	f_3	f_4
0	0	0	0	1	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	1	0	0	1

B. Output equations:

$f_1 = x*y$

$f_2 = x*y'$

$f_3 = x'$

$f_4 = x+y$

C. Circuit:

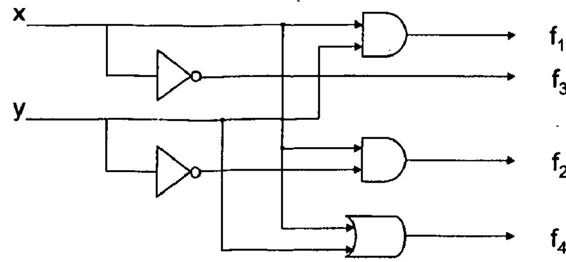


Figure 2.1: Load Instruction combinational circuit implementation

The execution of the constant array memory accesses on hardware is many a times faster than the execution of the instruction cycle for the instruction on a microprocessor.

Hence the Binning Pass detects the presence of global variables in the test algorithm and stores the pointers to the memory locations storing the constant global arrays in a Standard Template Library Vector Container. The basic blocks of the algorithm are tested to find whether there are memory accesses to the locations stored of the Vector Container, which is used to categorise the parts of the test algorithm having memory accesses to constant global arrays under data flow driven section. The simple empirical relations formed from the above detected features are described below.

The first relation calculates the total number of memory access instructions excluding the constant global array memory access instructions and control flow instructions. The basic block is believed to be more of control flow driven when the above calculated instructions in a basic block are greater than a specified threshold.

The second relation is a check for the data flow within a basic block. The presence of phi instructions indicates that the values required for the execution of basic block instructions are available at run-time and cannot be decided prior. Therefore an ideal data flow basic block has no phi instructions and has only one or no successors. The above condition is checked and the boolean result is stored in an indicator variable.

The third relation is a comparison between the data flow and control flow within a basic block. The number of phi instructions gives a measure of data flow into the basic block. The number of successors for a basic block gives a measure of the flow of control to the subsequent sections of the algorithm. Greater number of successors indicates that the basic block has more of control flow and the boolean result is stored in an indicator variable.

The final condition determines whether the basic block has self loop. The presence of a self loop indicates heavy control flow.

Appropriate weights are given to the input indicator variables and the result is calculated. The first relation which tries to classify the blocks on the number of control flow constructs, is of prime importance and hence given maximum weight. The calculated resultant is compared with a pre-defined constant to determine whether the basic block is a data flow driven or a control flow driven basic block and near to expected results are obtained using the relation.

2.3 Function Level Analysis

A measure of control flow at function level is identified by the following features:

1. Number of basic blocks [1].
2. Number of successors for each basic block [2].
3. Number of control flow driven basic blocks.

A measure of data flow at function level is identified by the following features

1. Total number of instructions in the function
2. Total number of phi nodes
3. Number of data flow driven basic blocks

The following simple empirical relations are formed from the above features.

The count of data flow driven basic blocks and control flow driven basic blocks determines the categorization of the function.

The ratio between edges and nodes of the function is taken into consideration. A function is believed to be more of control flow when its CFG has much greater number of edges compared to the number of nodes.

The number of basic blocks gives us a measure of number of times the instruction execution varies from the natural sequential order of execution and can be taken as a

[1] nodes of CFG of the function
[2] edges of CFG of the function

measure of control flow and huge basic blocks indicate greater execution of data and indicate data flow and hence a comparison is made between the above features.

The presence of backedges in the CFG of the function is taken into consideration as it is an indicator of heavy control flow within the function

Appropriate weights are given to the ratios formed above to decide the result. The calculated result is checked whether it is within specific pre-determined threshold levels. This is used to determine whether a function is data flow driven or control flow driven or mid way between them.

The empirical constants of all the relations mentioned above are determined by testing and validating the Binning Pass with a number of input test algorithms.

The input test algorithms include Control flow driven Data Structures like linked lists,stacks and queues and simple data flow driven digital signal processing algorithms which perform convolution, correlation, discrete fourier transforms, fast fourier transforms of input sequences and various other algorithms which are partly data flow driven and partly control flow driven.

Two Standard Template Library maps are created one each for basic blocks and functions respectively and the above calculated features stored in the maps for further usage.The Binning Pass does not modify the LLVM IR. The Pass is included in the Appendices A and B on page 49 and on page 51 correspondingly, for further reference.

2.4 Two Dimensional Discrete Fourier Transform Example

A simple two dimensional Discrete Fourier Transform is written to provide a test input to the Binning Pass. A two dimensional DFT computes the transform of a two-dimensional data set. The definition of the transform for the data set is given below:

$$y_{\mathrm{k1,k2}} = \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{\mathrm{j1,j2}} \exp^{-2\Pi i\left[\frac{j1k1}{n1}+\frac{j2k2}{n2}\right]}$$

where k1 = 0,1,...,n1-1 and k2 = 0,1,...,n2-1

n1and n2 are dimensions of input rectangular matrix.

The program written calculates the transform for floating point square matrices and is provided in the Appendix E on page 70 for further reference.

2.5 Binning Pass Results

The figure 2.2 below shows the statistics collected by the Pass for a data flow driven basic block of the algorithm:

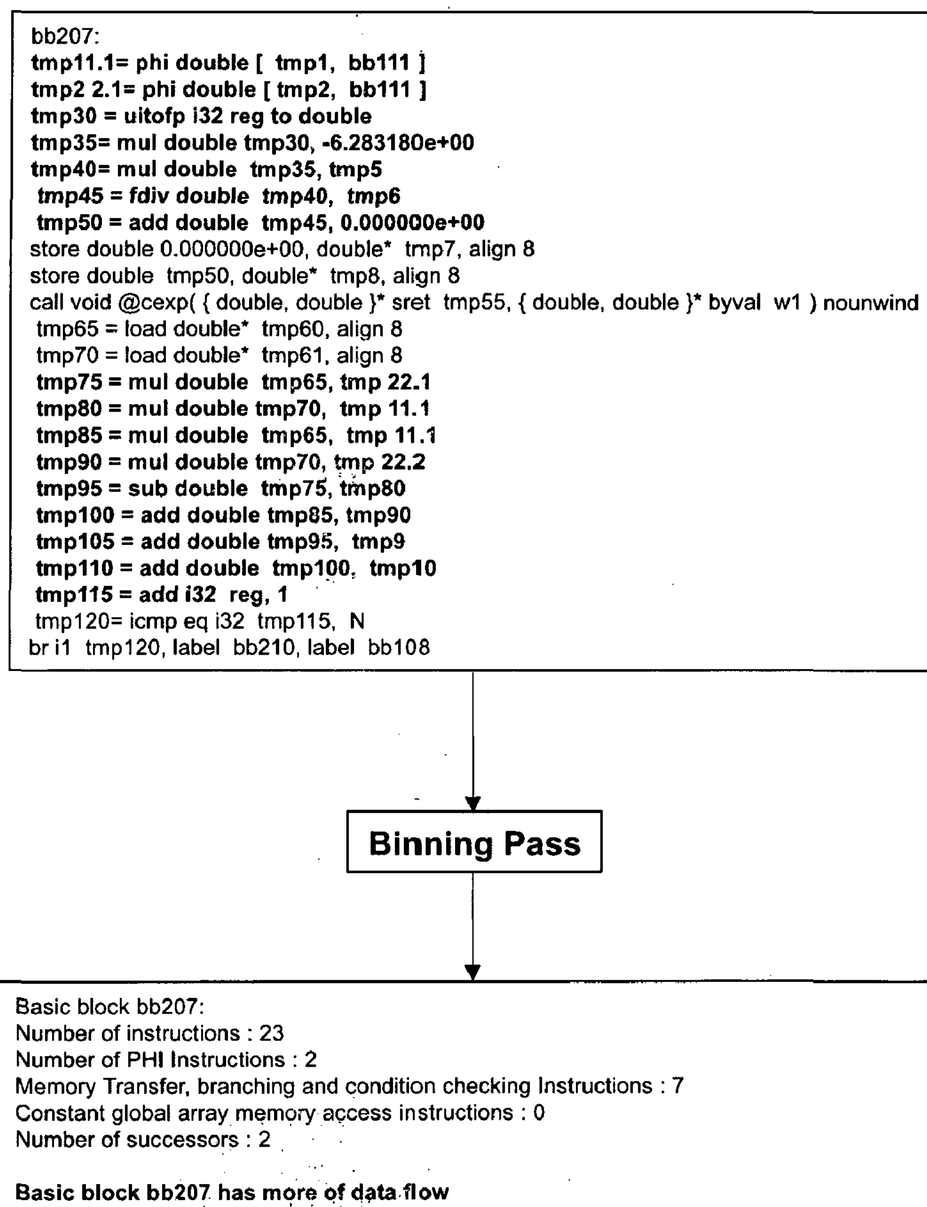


Figure 2.2: Data flow driven basic block

The upper block shows the LLVM IR of a basic block of the DFT example and the lower block shows the corresponding output of the Binning Pass which gathers statistics of the basic block. The statistics show that the basic block performs significant amount of arithmetical operations and data processing and hence the basic block is termed as data flow driven basic block.

The figure 2.3 below shows the statistics collected by the Pass for a control flow driven basic block of the algorithm:

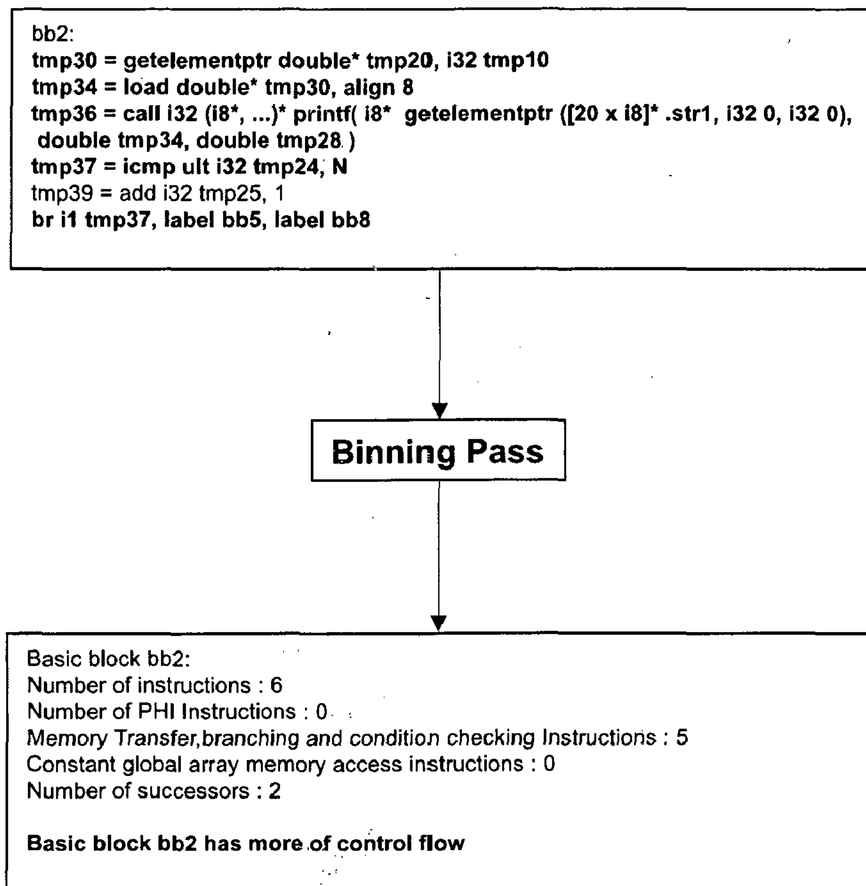


Figure 2.3: Control flow driven basic block

The output of the Pass shows that there are many branching instructions and memory access instructions and no constant global memory accesses as can be seen in the LLVM IR. Hence the basic block is termed as control flow driven basic block.

Similarly all the basic blocks of all the functions in the input test algorithm are classified with the help of the Pass. The table below shows that statistics collected for all the basic blocks of the 2d_dft transform function 70 written in the test algorithm.

The table 2.2 below shows the statistics collected by Binning Pass for 2d_dft function at basic block level:

Table 2.2: Binning Pass basic block statistics

Basic Block Name	Instructions	PHI Instructions	Memory Transfer and Control Flow Instructions	Constant Memory Accesses	Successors	Result
entry	11	0	9	0	2	Control Flow
bb372.preheader	11	0	9	0	1	Control Flow
bb372.outer	16	1	7	0	1	Data Flow
bb105	3	1	1	0	2	Data Flow
bb108.preheader	2	0	1	0	1	Data Flow
bb108	7	3	2	0	1	Data Flow
bb111	24	3	9	0	2	Data Flow
bb207	23	2	7	0	2	Data Flow
bb293.loopexit	3	2	1	0	1	Control Flow
bb293	13	2	10	0	2	Control Flow
bb326	6	0	5	0	2	Control Flow
bb347	8	3	4	0	1	Control Flow
bb372	3	1	2	0	2	Control Flow
bb105.preheader	1	0	1	0	1	Control Flow
bb378.loopexit	4	3	1	0	1	Control Flow
bb378.loopexit46	1	0	1	0	1	Control Flow
bb378	4	0	3	0	2	Control Flow
bb388.loopexit	1	0	1	0	1	Control Flow
bb388	2	0	2	0	0	Control Flow

The output of the Pass for the 2d_dft function 70 in the program is given below:

```
Number of instructions in function : 143
PHI Instructions in function : 21
Number of control flow basic blocks in function : 11
Number of data flow basic block in function : 8
Size of average basic block in function : 7
Number of nodes in function : 19
Number of edges of function : 26
The function has more of control flow
```

The validity of the categorization can be checked with the help of the figure 2.4 below which is the Control Flow Graph for the 2d_dft function

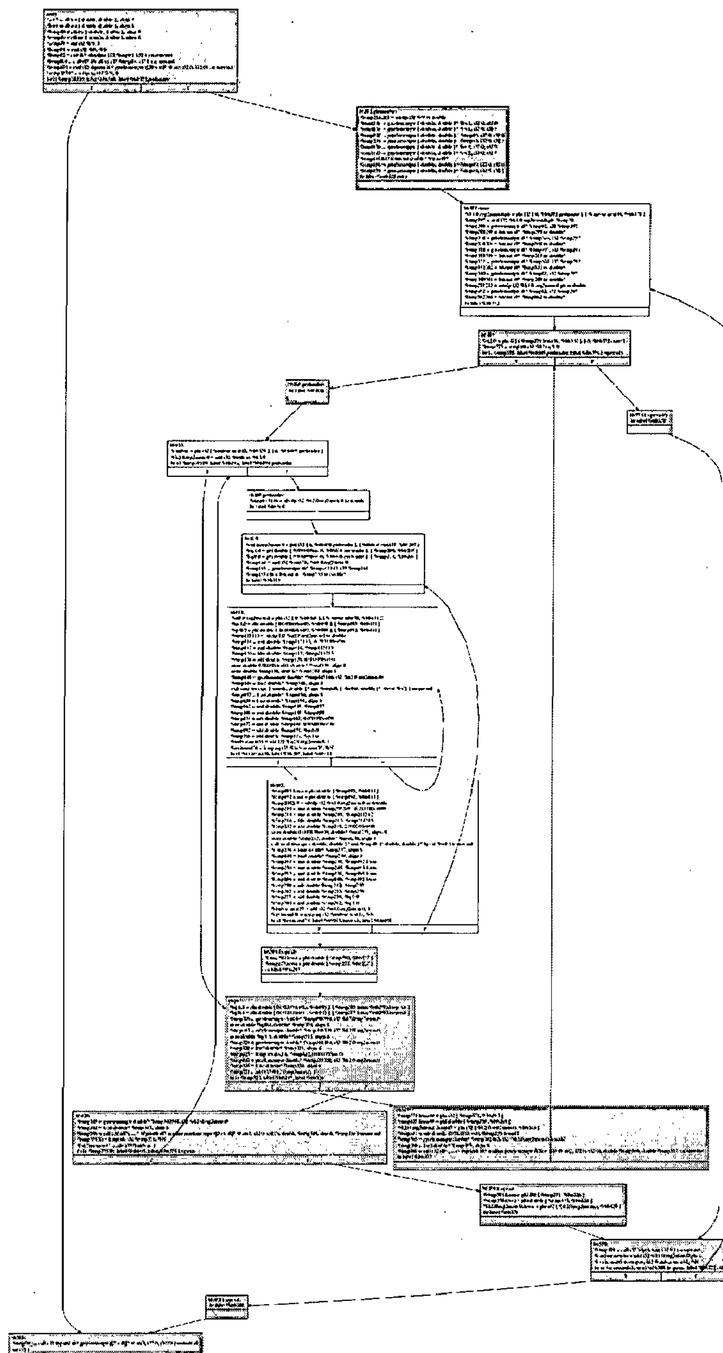


Figure 2.4: Control Flow Graph of twod_dft function

The shaded blocks in the image are control flow driven basic blocks and the unshaded blocks are data flow driven basic blocks. The image above shows that though there are a few basic blocks in which extensive data processing is done, the rest of the blocks are

small in size and the control flow driven basic blocks are more and edges are more than the nodes indicating that it is a control flow driven function.

Chapter 3

Selection Pass

3.1 Description

The Selection Pass is a Transform Pass developed for hardware/software mapping. The Pass utilises the statistics gathered by the Binning Pass to partition the test algorithm into data flow and control flow driven sections. The data flow driven section is mapped to hardware using the VHDL backend for LLVM and the control driven section is mapped to the software using PowerPC backend for LLVM. These sections are made independant of each other for parallel execution on both of the above mentioned backends.

The Selection Pass modifies the LLVM IR to select only the instructions which are to be mapped either to software or to hardware and replaces the other instructions by intrinsic instructions. These intrinsic replacement instructions are used to represent the parts of the algorithm partitioned to a different computing context. These newly created intrinsic instructions are place holders for the instructions which should not be executed by the backend and are supposed to keep the control and data flow dependencies persistant.

The Selection Pass ensures that the results of the Binning Pass are computed before the Pass is executed. The statistics of each basic block are accessed from the map created in the Binning Pass to store the basic block information. This information is used to identify the data flow driven and control flow driven basic blocks.

The Selection Pass is run twice using different command line arguments to create the required two sections. The control flow section is created from the test algorithm LLVM IR by replacing the dataflow driven basic blocks of the code with intrinsics and the data flow section is created by replacing the control flow driven basic blocks with intrinsics.

3.2 Selection Pass Intrinsics

Selection Pass modifies the LLVM IR to achieve the required functionality by adding new intrinsics to the LLVM. The phi and terminator instructions of the basic blocks being modified are retained and the rest of the instructions are replaced by migrate_begin and migrate_end intrinsic functions.

3.2.1 Intrinsic Functions in LLVM [6]

The intrinsic functions represent an extension mechanism to the LLVM language. Generally, all extensions to LLVM start as an intrinsic function and then be turned into an instruction if warranted. Intrinsic function names start with an "llvm." prefix. This prefix is reserved in LLVM for intrinsic names and hence function names may not begin with this prefix.and they may only be used in call or invoke statements. Intrinsic functions are always external functions i.e the body of intrinsic functions cannot be defined and they are to be enumerated in the LLVM Intrinsics.td Table Generator file.

A. Overloaded Intrinsics

Overloaded intrinsics represent a family of functions that perform the same operation but on different data types. For instance, overloading can be used to allow an intrinsic function to operate on any integer type as there are numerous integer types in LLVM. One or more of the argument types or the result type can be overloaded to accept any integer type. Argument types may also be defined as exactly matching a previous argument's type or the result type. This allows an intrinsic function which accepts multiple arguments, but needs all of them to be of the same type, to only be overloaded with respect to a single argument or the result.

Overloaded intrinsics will have the names of its overloaded argument types added as suffixes to the function name. For instance, the pre-defined llvm.ctpop function takes an integer of any width and returns an integer of exactly the same integer width. This leads to a family of functions such as i16 @llvm.ctpop.i16(i16 %val) and i43 @llvm.ctpop.i43(i43 %val). In this example only one type, the return type, is overloaded, and only one type suffix is required as the argument's type is matched against the return type, it does not require its own name suffix.

The following are the categories of pre-existing intrinsics in LLVM

1. Variable Argument Handling Intrinsics.
2. Accurate Garbage Collection Intrinsics.
3. Code Generator Intrinsics.
4. Standard C Library Intrinsics.
5. Bit Manipulation Intrinsics.
6. Debugger Intrinsics.
7. Exception Handling Intrinsics.
8. Trampoline Intrinsics.
9. Atomic Intrinsics.
10. General Intrinsics.

3.2.2 Selection Pass Intrinsics

LLVM does not support the functionality expected to obtain in its current incarnation with the pre-existing intrinsic functions previously mentioned and hence new intrinsics are created in the Selection Pass to add the required functionality to LLVM.

A. migrate_begin intrinsic

The migrate_begin intrinsic instruction marks the beginning of the set of instructions of the basic block which are not executed by the current backend and is created for every basic block which is to be modified. The intrinsic is created to handle the incoming data dependencies of the basic block. These data dependencies are the values external to the present basic block, required for the execution of the above mentioned set of basic block instructions.

The arguments for the intrinsic function are given by the Deps.h header file included in the Selection Pass. The intrinsic call sends the arguments to the corresponding migrate_end intrinsic function.

B. migrate_end intrinsic

The migrate_end intrinsic instructions replace the set of instructions which are not to be executed by the current backend. A migrate_end intrinsic function call is created for every instruction having outgoing dependencies. The outgoing dependencies are the instructions external to the basic block which have the present instruction as an operand. The intrinsic is also created for an instruction which has uses in the PHI and terminator instructions present in the same basic block.

The return value of the Call instruction is the value of the instruction replaced by it. The data dependencies of the external instructions on the basic block are satisfied by iterating over the def-use chain of each instruction and replacing the uses of instruction with the corresponding migrate_end Call instruction.The arguments for the intrinsic functions are given by the Deps.h header file.

3.3 Deps.h header file description

The Deps class is created for providing arguments to intrinsic functions in Selection Pass and for further application in the creation of VHDL RTL output from VHDL backend. The Deps.h header file comprises the dependencies class Deps. The class gathers the arguments required by the migrate_begin and migrate_end intrinsic functions. The arguments to the migrate_begin intrinsic function are the following:

1. Incoming data dependencies of the basic block [1]
2. The phi instructions of the basic block.
3. The count of the arguments mentioned in 1 and 2.
4. The enumeration of the intrinsic within the function.

The incoming data dependencies are gathered by iterating over uses and definitions (use-def) chain of each instruction of the basic block. This chain iterates over the operands of the instruction. The global values required for execution of the basic block are also taken into consideration. The arguments mentioned are stored in a vector container. The arguments to the migrate_end intrinsics are the following:

[1] Teminator instruction incoming data dependencies not included.

1. The enumeration of the migrate_begin intrinsic of the basic block
2. The position of the instruction within the basic block which is being replaced by the migrate_end intrinsic.

The migrate_end intrinsics are enumerated by iterating over the definitions and uses (def-use) chain of each instruction. This chain iterates over the uses of each instruction. The arguments for the migrate_end intrinsics are stored in a map. The class also implements functions for providing the collected arguments to the callee. In this way the collected arguments are sent to the migrate_begin and migrate end intrinsics. The class is provided in the Appendix D on page 66 for further reference.

3.4 Additions to Intrinsics.td file

3.4.1 LLVM Intrinsics.td TableGen File [6]

Intrinsics.td is a TableGen file which defines all LLVM intrinsic functions. A TableGen assists in developing and maintaining records of domain-specific information. As there may be a large number of such records, TableGen is specifically designed to allow writing flexible descriptions and for common features of the records to be factored out. This reduces the amount of duplication in the description, reduces the chance of error, and makes it easier to structure domain specific information.

The following specifications are to be provided to define an intrinsic in the file:

1. Intrinsic Property
2. Intrinsic Type
3. Intrinsic Definition

A. Intrinsic Property

Intrinsic Properties are the memory properties of the intrinsic. An intrinsic is allowed to have exactly one of these properties set. The properties are listed from the most aggressive (best to use if correct) to the least aggressive.

1. IntrNoMem

2. IntrReadArgMem

3. IntrReadMem

4. IntrWriteArgMem

5. IntrWriteMem

IntrNoMem property states that the intrinsic does not access memory.

IntrReadArgMem property states that the intrinsic reads only from memory that one of its arguments points to, but may read an unspecified amount.

IntrReadMem property states that intrinsic reads from unspecified memory and hence it cannot be moved across stores. However, it can be reordered otherwise and can be deleted if dead.

IntrWriteArgMem property states that the intrinsic reads and writes only from memory that one of its arguments points to, but may access an unspecified amount. It has no other side effects. This may only be used if the intrinsic doesn't "capture" the argument pointer (e.g. storing it someplace).

IntrWriteMem property states that the intrinsic may read or modify unspecified memory or has other side effects. This is the default if the intrinsic has no other intrinsic memory property.

The above memory access properties are considered for optimization of the intrinsic functions.

B. Intrinsic Type

Intrinsic Types are the LLVM types used by the intrinsic. The types to used by the intrinsics are to be defined in the file prior to their usage in the intrinsic definition. The types range from simple integer, float and pointer data types to complex vector data types.

C. Intrinsic Definition

The Intrinsic Definition provides for the manner in which an intrinsic is defined in the file. The following is the syntax of a general intrinsic definition

def **intrinsic definition name** : Intrinsic<[**types**], [**intrinsic property**], "**LLVM intrinsic name**" >;

intrinsic definition name should start with "int_" which indicates that the intrinsic entry is actually an enumeration. The name then should match with the LLVM intrinsic name with the "llvm." prefix removed and all "." characters turned into "_" characters. For instance, int_bswap_i32 is the intrinsic definition name for the LLVM intrinsic name llvm.bswap.i32.

types is the function type of the intrinsic. It includes the return type and the argument types expected for the intrinsic respectively.

intrinsic property describes the memory behaviour property of the intrinsic function.

LLVM intrinsic name is the name given to the intrinsic during its creation in the LLVM Pass added with "llvm." prefix.

Finally, all the different Passes that are intended to use the extension made to LLVM are updated.

3.4.2 Additions to Intrinsics.td File

The newly created migration intrinsics in the Selection Pass are enumerated in the Intrinsics.td TableGen file to be recognized by all the libraries and tools present in LLVM. The following are the additions to the file:

```
def int_migrate_begin:
Intrinsic<[llvm_i32_ty,llvm_vararg_ty],
[IntrWriteMem],"llvm.migrate_begin">;
def int_migrate_end_int :
Intrinsic<[llvm_anyint_ty,llvm_i32_ty,llvm_i32_ty],
[IntrWriteMem],"llvm.migrate_end_int">;
def int_migrate_end_float :
Intrinsic<[llvm_anyfloat_ty,llvm_i32_ty,llvm_i32_ty],
```

```
[IntrWriteMem],"llvm.migrate_end_float">;
def int_migrate_end_ptr_8 :
Intrinsic<[llvm_ptr_ty,llvm_i32_ty,llvm_i32_ty],
[IntrWriteMem],"llvm.migrate_end_ptr_8">;
def int_migrate_end_ptr_ptr :
Intrinsic<[llvm_ptrptr_ty,llvm_i32_ty,llvm_i32_ty],
[IntrWriteMem],"llvm.migrate_end_ptr_ptr">;
def int_migrate_end_ptr_32 :
Intrinsic<[llvm_ptr32_ty,llvm_i32_ty,llvm_i32_ty],
[IntrWriteMem],"llvm.migrate_end_ptr_32">;
def int_migrate_end_ptr_ptr32 :
Intrinsic<[llvm_ptrptr32_ty,llvm_i32_ty,llvm_i32_ty],
[IntrWriteMem],"llvm.migrate_end_ptr_ptr32">;
def int_migrate_end_ptr_float :
Intrinsic<[llvm_ptrfloat_ty,llvm_i32_ty,llvm_i32_ty],
[IntrWriteMem],"llvm.migrate_end_ptr_float">;
def int_migrate_end_ptr_double :
Intrinsic<[llvm_ptrdouble_ty,llvm_i32_ty,llvm_i32_ty],
[IntrWriteMem],"llvm.migrate_end_ptr_double">;
```

The following are the entries used from the list of types in the file, used for the above definitions:

```
def llvm_void_ty : LLVMType<isVoid>;
def llvm_i8_ty : LLVMType<i8>;
def llvm_i32_ty : LLVMType<i32>;
def llvm_anyint_ty : LLVMType<iAny>;
def llvm_anyfloat_ty : LLVMType<fAny>;
def llvm_vararg_ty : LLVMType<isVoid>;
def llvm_ptr_ty : LLVMPointerType<llvm_i8_ty>;
def llvm_ptrptr_ty : LLVMPointerType<llvm_ptr_ty>;
```

The following are the new entries made for the additional types used in the intrinsic definitions:

```
def llvm_ptr32_ty : LLVMPointerType<llvm_i32_ty>;
def llvm_ptrptr32_ty : LLVMPointerType<llvm_ptr32_ty>;
```

```
def llvm_ptrdouble_ty :  LLVMPointerType<llvm_double_ty>;
def llvm_ptrfloat_ty :  LLVMPointerType<llvm_float_ty>;
```

The migrate_begin intrinsic has an integer return type. The "llvm_vararg_ty" handles the types of the variable number of arguments to the migrate_begin intrinsic function.

Owing to the strict type system of the LLVM, eight migrate_end intrinsics are created to handle the varying return types of the instructions being replaced by the intrinsics.

Instructions with integer return types are mapped to the intrinsic with return type "iAny" and instructions with float return types are mapped to the intrinsic with return type "fAny". The above two are overloaded intrinsics and have the return types of the instructions as suffixes to the instruction names.

The pointer type intrinsics correspond to the instructions with different pointer return types.

The default intrinsic property is used define the memory access behaviour of the intrinsics.

3.5 Data Flow Dependencies

The migrate_begin and migrate_end intrinsic instructions ensure that the data dependencies are maintained. The following figure 3.1 explains the handling of data dependencies by the intrinsic instructions.

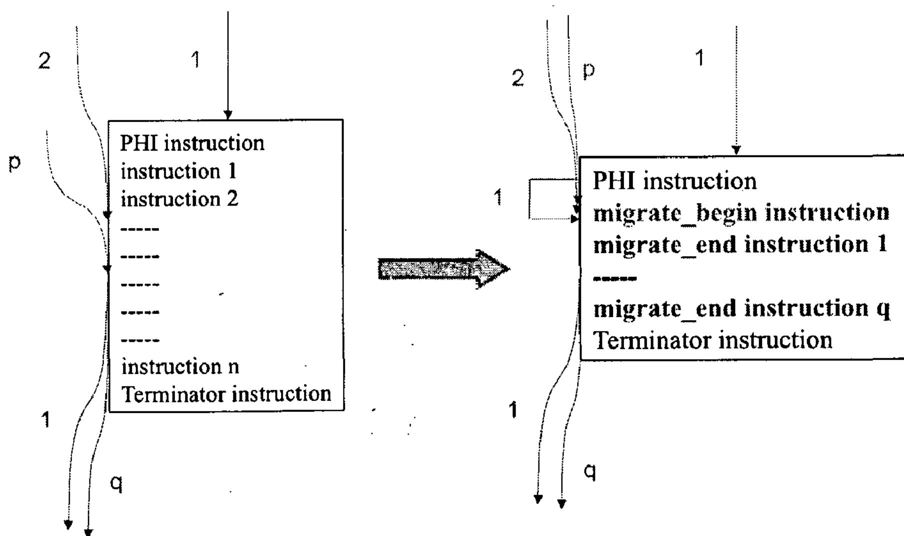


Figure 3.1: Data Dependencies

The left block is the basic block with "n" instructions prior to the introduction of intrinsic instructions and the right block is the same basic block after the addition of intrinsics. The incoming arrows indicate the data dependencies entering into the basic blocks and the outward arrows indicate the data going out of the basic block.

From the block on the right side, it can be noticed that the migrate_begin intrinsic gathers all the incoming dependencies into the basic block. The migrate_end intrinsics are created for "q" instructions (q<=n) having outgoing dependencies.

The PHI and Terminator instructions are retained and the references made by the other instructions in the basic block are dropped to ensure that the instructions do not have internal uses remaining. The instructions are discarded after verifying that the uses list of each instruction to be deleted is empty.

3.6 Control Flow Dependencies

The control flow dependencies are satisfied by placing the newly created migrate_end intrinsic instructions in the basic block in the same order as that of the instructions replaced by them. The phi instructions are placed first followed by the migrate_begin intrinsic call instruction and the migrate_end intrinsic calls are placed after the migrate_begin instruction and the terminator instruction marks the end of the basic block.

The newly created intrinsic instructions implicitly define the interface between sections of the algorithm to be mapped separately to software and hardware.

3.7 Walsh Transform Example

A program is written which computes the natural ordered Walsh-Hadamard transform of integer input sequence. The definition of the transform is given below:

$$y = WHT_N * x$$

where

x - input sequence

y - transformed sequence

WHT_N- Walsh Hadamard matrix where $N = 2^n$.

$WHT_N = WHT_2 \otimes WHT_2 \ldots \otimes WHT_2$ n times.

$$WHT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{ and}$$

$$WHT_4 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

The iterative procedure given below is used in the development of the program.

$$WHT_N = \prod_{i=1}^{n} (I_{2\hat{(i-1)}} \otimes WHT_2 \otimes I_{2\hat{(n-i)}})[12]$$

The program is provided in the Appendix F on page 72 for further reference.

3.8 Selection Pass Results

The results shown by the figure below depict the manner in which a basic block of the the Walsh transform input algorithm is modified by the Filter Pass to include the intrinsics

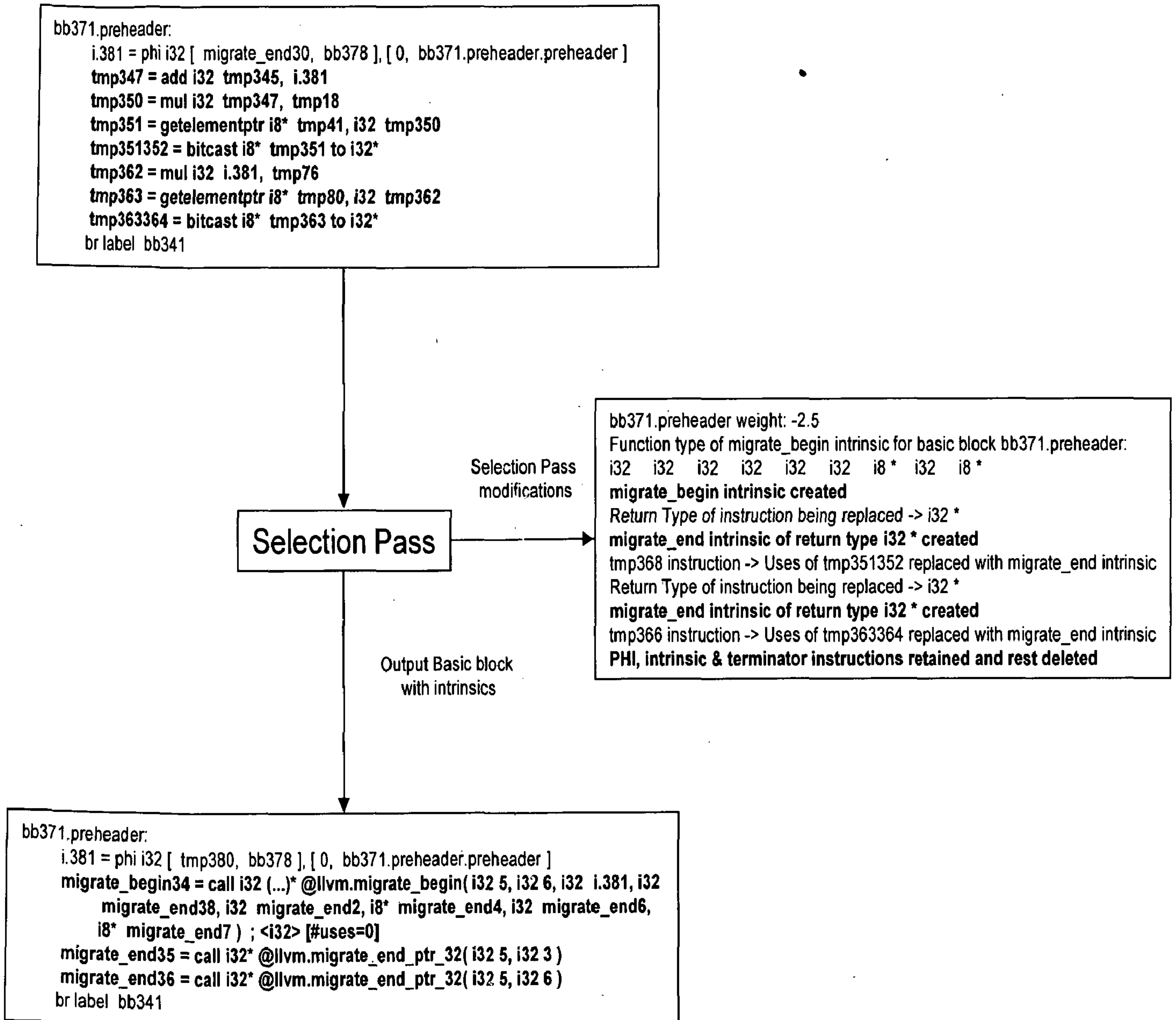


Figure 8: Modified basic block with intrinsics

The upper block shows the LLVM IR of a basic block of the Walsh Tranform algorithm. The block to the right shows the analysis and modifications being made to the LLVM IR. The block below shows the modified LLVM IR of the basic block. The figure clearly shows that the data and control flow dependencies of the basic block are maintained by the newly created intrinsic.

The figure below shows the sectioning of the input algorithm by the Selection Pass.

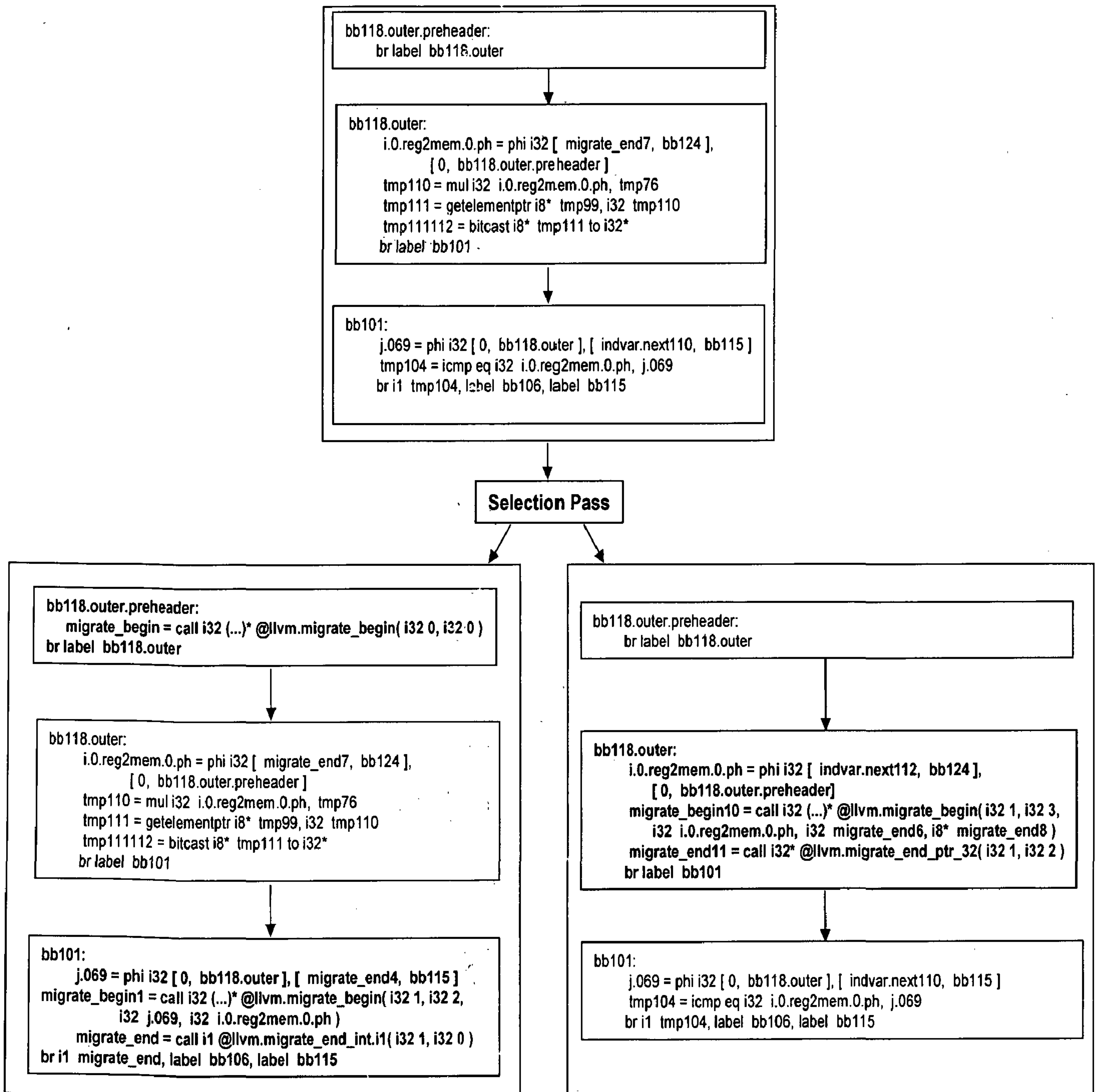


Figure 9: Hardware and software sections

The upper block of the figure shows a set of basic blocks of the Walsh transform algorithm. The lower blocks are the software and hardware sections of the algorithm correspondingly. The image clearly indicates that the code segment being sent to hardware has the data flow driven basic blocks of the main algorithm intact and the rest of the basic blocks are filled with dummy intrinsic instructions and the code segment being sent to software has the control flow driven basic blocks of the main algorithm intact and the remaining basic blocks are filled with intrinsic instructions.

Chapter 4

Adaptations to the PowerPC and VHDL Backends

The newly added intrinsics are extensions to LLVM language and support is to be provided for them to enable code generation on target backends. This is achieved by making additions to the LLVM Target Independant Code Generator.

4.1 LLVM Code Generator and Additions to SelectionDAGISel.cpp file

4.1.1 LLVM Target-Independant Code Generator [6]

A. Description

The LLVM target-independent code generator is a framework that provides a suite of reusable components for translating the LLVM IR to the machine code for a specified target, either in assembly form (suitable for a static compiler) or in binary machine code format (usable for a JIT compiler).

The LLVM target-independent code generator consists of five main components:

1. Abstract target description interfaces which capture important properties about various aspects of the machine.

2. Machine Code Representation classes used to represent the machine code being generated for a target. These classes are intended to be abstract enough to represent the machine code for any target machine.

3. Target independent algorithms used to implement various phases of native code generation such as register allocation, scheduling, stack frame representation, etc.

4. Implementations of the abstract target description interfaces for particular targets. These machine descriptions make use of the components provided by LLVM, and can optionally provide custom target-specific passes, to build complete code generators for a specific target. The LLVM currently supports ARM, SPARC, PowerPC and Intel x86 architectures.

5. The target independent JIT components.

The target independant code generation algorithms implement the high level design of the Code Generator and enable code generation for the intrinsic instructions, independant of target backends.

B. The High Level Design of LLVM Code Generator

The LLVM target-independent code generator is designed to support efficient and quality code generation for standard register-based microprocessors. Code generation in this model is divided into the following stages:

1. Instruction Selection
2. Scheduling and Formation
3. SSA-based Machine Code Optimizations
4. Register Allocation
5. Prolog/Epilog Code Insertion
6. Late Machine Code Optimizations
7. Code Emission

Instruction Selection is the process of translating LLVM code presented to the code generator into target-specific machine instructions. This step turns the LLVM code into a Directed-Acyclic- Graph(DAG) of target instructions. LLVM uses a SelectionDAG based instruction selector for translation.The SelectionDAG is a Directed-Acyclic-Graph

whose nodes are instances of the SDNode class. The SelectionDAG is a Directed-Acyclic-Graph whose nodes are instances of the LLVM SDNode class. The following example explains SelectionDAG.

C. SelectionDAG Example

Consider the following LLVM IR:

tmp1 = mul float W, X

tmp2 = add float tmp1, Y

tmp3 = sub float tmp2, Z

The SelectionDAG for the above LLVM code is given below

(fsub:f32 (fadd:f32 (fmul:f32 W, X), Y), Z)

The SelectionDAG changes depending on the operations supported by the target. For instance the PowerPC supports floating point multiply-and-add(FMA) operations. The SelectionDAG for PowerPC is given below

(FSUBS (FMADDS W, X, Y), Z)

4.1.2 Additions to SelectionDAGISel.cpp file

The initial SelectionDAG is constructed from the LLVM input by the SelectionLoweringClass in SelectionDAGISel.cpp file of LLVM Code Generator. The intent of this file is to expose the low-level, target-specific details to the SelectionDAG to the maximum extent possible. SelectionDAGLowering Class is used for common target independant lowering implementation and has methods to lower the Call instructions to the intrinsic functions. The LLVM code representing the intrinsic function Calls is lowered in the visitIntrinsicCall method to SelectionDAG operators that the target instruction selector can accept natively.

The following are the switch instruction cases included to the visitIntrinsicCall method of the SelectionDAGISel.cpp file in LLVM.

```
const char *SelectionDAGLowering::visitIntrinsicCall(CallInst &I, unsigned
Intrinsic) {
    switch (Intrinsic) {
```

```
case Intrinsic::migrate_begin: {
return "migrate_begin";
}
case Intrinsic::migrate_end_int: {
return "migrate_end";
}
case Intrinsic::migrate_end_float: {
return "migrate_end_float";
}
case Intrinsic::migrate_end_ptr_8: {
return "migrate_end";
}
case Intrinsic::migrate_end_ptr_32: {
return "migrate_end";
}
case Intrinsic::migrate_end_ptr_ptr: {
return "migrate_end";
}
case Intrinsic::migrate_end_ptr_ptr32: {
return "migrate_end";
}
case Intrinsic::migrate_end_ptr_float: {
return "migrate_end";
}
case Intrinsic::migrate_end_ptr_double: {
return "migrate_end";
}
}
}
}
```

Normally the intrinsic call instructions in the SelectionDAGISel.cpp file are lowered to methods present within the file and present and then null value is returned in the above shown "switch-case" statements but, as it is required to emit the migrate intrinsics as calls to named external functions, the symbol or the function name of the migrate intrinsic is returned as shown above.

The switch case statements clearly indicate that the created intrinsics are enumerations in the Intrinsic namespace and migrate_begin and migrate_end are the symbols added in LLVM to lower the intrinsics.

4.2 Possible future extensions to SelectionDAGISel.cpp file

New additions are to be made to the SelectionDAGISel.cpp file whenever the Filter Pass is extended by creating new migrate_end intrinsics in the Filter Pass.

For instance, if migrate_end_ptr_ptr_float and migrate_end_ptr_ptr_double intrinsics are created to cater to the double pointer to float(**f32) and double pointer to double(**f64) data types, the following are the additions to be made to the file:

```
case Intrinsic::migrate_end_ptr_ptr_float:  {
return "migrate_end";
}
case Intrinsic::migrate_end_ptr_ptr_double:  {
return "migrate_end";
}
```

The present intrinsic lowering is performed at higher levels of abstraction of the LLVM Code Generator and completely independant of the target backends. Target specific lowering of intrinsics can also be done by making additions to the TableGen files describing the instruction formats of the target architectures. Various procedures are followed to lower intrinsics into LLVM and the procedure discussed in this section is one among them.

4.3 PowerPC Backend Results

The LLVM infrastructure is equipped with static backends both for both PowerPC 32 bit and 64 bit architectures. Once the intrinsics are adapted to the target backends, the bit code output of software section of the Filter Pass is compiled into the PowerPC assembly language by pre-existing PowerPC-32 backend. The following figure 4.1 shows the results obtained from the backend.

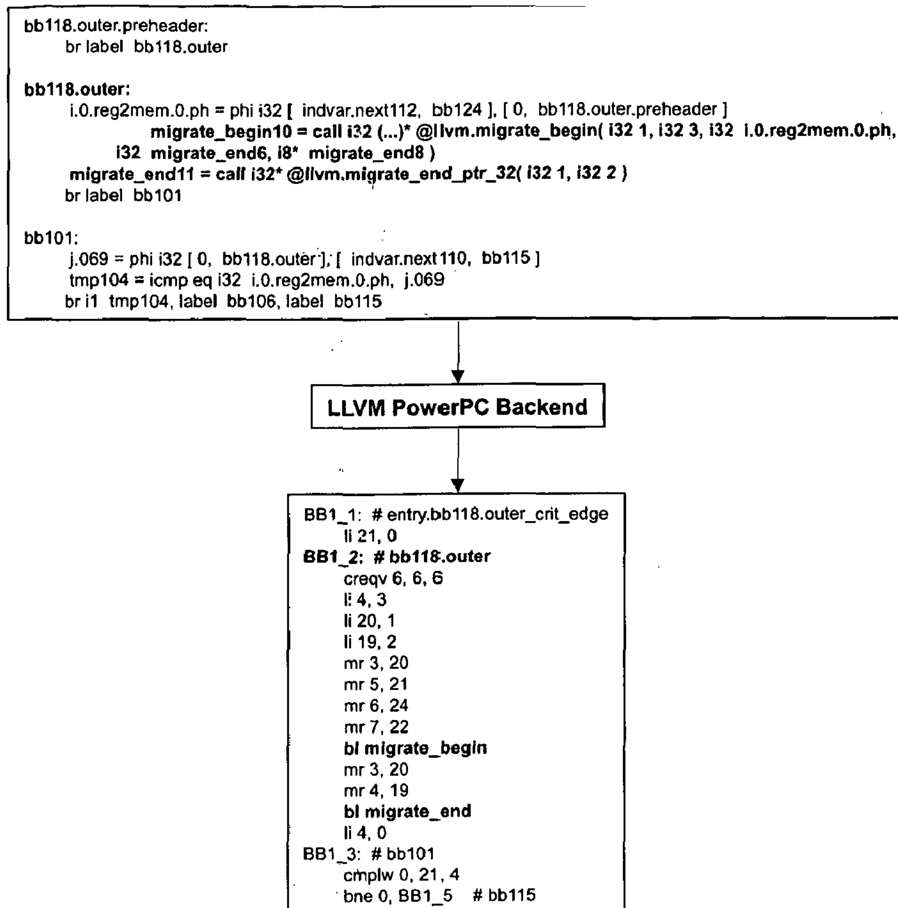


Figure 4.1: PowerPC Assembly Language Output

The image shows the PowerPC assembly language output obtained from the PowerPC-32 Backend for a basic block from the software section LLVM IR of the Selection Pass containing intrinsics. It can be seen from the output that the newly created intrinsics have been adapted to the PowerPC backend.

4.4 VHDL Backend Results

The VHDL backend for LLVM is under construction. The backend has been adapted to the newly created intrinsic functions and the following are results of lowering the LLVM IR of the hardware section to VHDL RTL. The following is a basic block from the hardware section LLVM IR of the Selection Pass.

```
define i32 blub(i32 a, i32 b) {
entry:
tmp6 = mul i32 b,a
tmp3 = add i32 b, a
tmp8 = add i32 tmp3, tmp6
tmp11 = mul i32 tmp8, a
tmp13 = add i32 tmp11,tmp8
tmp16 = mul i32 tmp13, b
tmp18 = add i32 tmp16, tmp13
ret i32 tmp18
}
```

The following shows the corresponding VHDL representation obtained from the VHDL Backend.

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;

entity blub_entry is
port (
rst :  in std_logic;
SyncI : in std_logic;
SyncO : out std_logic;
a:  in std_logic_vector(31 downto 0);
b:  in std_logic_vector(31 downto 0);
tmp18:  out std_logic_vector(31 downto 0);
clk :  in std_logic
);
end entity;
```

```
architecture LLVM_VHDL of blub_entry is
signal entry_sync_0_5:  std_logic_vector( 0 to 6-1);
signal tmp6:std_logic_vector(32-1 downto 0);
signal tmp3:std_logic_vector(32-1 downto 0);
signal tmp8:std_logic_vector(32-1 downto 0);
signal tmp11:std_logic_vector(32-1 downto 0);
signal tmp13:std_logic_vector(32-1 downto 0);
signal tmp16:std_logic_vector(32-1 downto 0);
--tmp18 in port list.
begin
p_entry:  process (clk) is
begin
if(clk'event and clk='1') then
tmp6 <= STD_LOGIC_VECTOR(UNSIGNED(b) * UNSIGNED(a));
tmp3 <= STD_LOGIC_VECTOR(UNSIGNED(b) + UNSIGNED(a));
tmp8 <= STD_LOGIC_VECTOR(UNSIGNED(tmp3) + UNSIGNED(tmp6));
tmp11 <= STD_LOGIC_VECTOR(UNSIGNED(tmp8) * UNSIGNED(a));
tmp13 <= STD_LOGIC_VECTOR(UNSIGNED(tmp11) + UNSIGNED(tmp8));
tmp16 <= STD_LOGIC_VECTOR(UNSIGNED(tmp13) * UNSIGNED(b));
tmp18 <= STD_LOGIC_VECTOR(UNSIGNED(tmp16)+ UNSIGNED(tmp13));
end if;
end process;
end architecture;
```

The set of instructions of the each data flow driven basic block are represented in VHDL as *entity* and *architecture* units which are to be synthesized in hardware.

4.5 Hardware/Software Integration

The hardware output(VHDL RTL) obtained from the VHDL Backend and the software output(PowerPC assembly) are integrated using the Xilinx EDK tool with the help of an existing bus interface and a HW/SW interaction program written in C programming language.

The existing bus interface which worked at funtion level data transfers was adapted to meet the basic block level data transfers required for the present work. The VHDL RTL

output and the bus interface are synthesized on Xilinx Virtex II FPGA. The PowerPC assembly is executed on the PowerPC microprocessor present on the same FPGA. All the files required are imported into the Xilinx EDK environment where the co-synthesis is performed. The HW/SW integration is performed with the help of the bus interface and the HW/SW interaction program. The HW/SW interaction program contains the definitions for the migrate_begin and migrate_end functions.

The following figure 4.2 describes the HW/SW integration and the manner in which the input application is executed

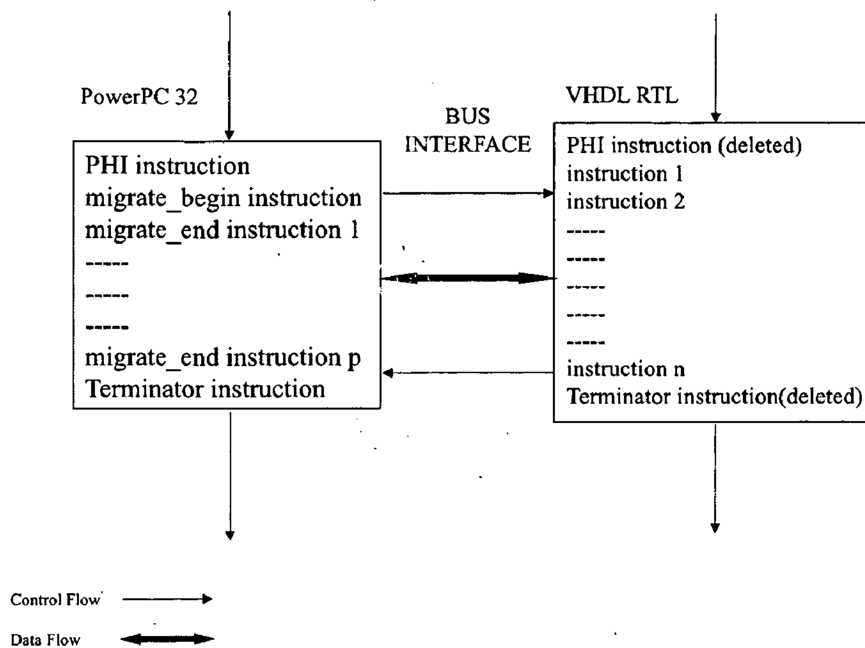


Figure 4.2: HW/SW Integration

The image shows the hardware and software sections and the transfer of data and control betweem them through the bus interface. The execution is carried out on the PowerPC of Virtex II till the migrate intrinsics are encountered. As soon as a function call to the migrate_begin intrinsic is encountered, the control is transferred to the function definition of the migrate_begin intrinsic provided in the HW/SW interaction program. The same call instruction transfers the arguments to the migrate_begin funtion.

The arguments of migrate_begin instruction are the following

1. Incoming data dependencies of the basic block having the intrinsics [1]

2. The phi instructions of the basic block.

[1]Teminator instruction incoming data dependencies not included.

3. The count of the arguments mentioned in 1 and 2.

4. The enumeration of the intrinsic within the parent function.

The first and second arguments provide the input data necessary for the execution of the subsequent instructions on hardware. The fourth argument is for the provision of a new memory segment for each new migrate_begin intrinsic and the third argument is for the amount of memory space required for the storage of input arguments or data. The migrate_begin function loads these arguments into the memory locations specified by the bus interface and waits for the execution of the susequent instructions on hardware. As soon as the execution of the instructions is finished in hardware the control is returned back and the PowerPC proceeds with the execution of remaining instructions. The migrate_end instructions are place holders for instructions having external dependencies. When a migrate_end function call is encountered, the control and the arguments are again transferred to the definition of the migrate_end intrinsic function definition provided in the HW/SW interaction program. The arguments of the migrate end function are the following

1. The enumeration of the migrate_begin intrinsic of the basic block

2. The position of the instruction within the basic block which is being replaced by the migrate_end intrinsic.

The values of the instructions executed are placed in consecutive memory locations. The first and second arguments of the migrate_end intrinsic provide for the identification of the base address and offset of the memory location in which the required value of the instruction is present. This value is returned back to the function call and the PowerPC proceeds in the same manner with the execution of the remaining migrate_end instructions if present. In this way the application is executed in hardware/software.

Chapter 5

Conclusion

5.1 Summary

1. LLVM Passes (Binning Pass and Selection Pass) are implemented to enable automatic hardware/software partitioning of input algorithms in C/C++ programming languages.

2. The VHDL and PowerPC target backend outputs are obtained for the created HW/SW sections of the algorithm in LLVM IR.

3. The implemented Passes are tested and verified using the DSP algorithms (Two dimensional DFT program and Walsh Hadamard Transform program) written.

4. The present Passes support integer, float data types and only a few kinds of pointer data types of the identifiers present in input algorithms.

5. The VHDL Backend used for the presented work is still in development stages and hence does not support the identifiers of float data type present in input algorithms.

6. The co-synthesis of the VHDL/PowerPC outputs is to be performed for the execution of the input algorithms and to analyze the performance and communication costs of the co-design implementation.

5.2 Future Extensions

5.2.1 Possible Extensions to the Binning Pass

Various new features which identify control flow and data flow can be detected and utilised in the construction of the Binning Pass. For instance, the following control flow features can be considered:

1. Number of immediate predecessors for each basic block
2. Number of arguments of each phi instruction
3. Number of possible loops in a function
4. Loop unrolling feature detection

The empirical relations framed may also depend on various other ratios formed using the presently identified features or new features. For instance the new ratio Number of immediate predecessors / Number of phi nodes can be considered. The number of immediate predecessors gives us the incoming branches to the basic block and is a measure of control flow. The number of phi nodes represents the number of data values coming into the basic block and is a measure of data flow. The Pass can utilise the Control Flow Graph classes supported by the LLVM libraries for identification of the control flow sections but the present Binning pass is simple and sufficient for a considerably large section of algorithms.

It is also possible that the profiling information can be used to identify the sections of the algorithm to be sent to HW/SW instead of the present Binning Pass. Profiling is done for performance analysis and evaluation of the execution time of the program. It finds the critical code segments of the program such as loops or iterative processes, where the processor spends most of its execution time. These code segments can be identified and sent to hardware for speedier execution. LLVM-GCC compiler has a tool called llvm-prof which reads in the bitcode file of the input program and determines the hotspots of the program. This profiling tool can also be used for the categorization of HW/SW sections.

5.2.2 Possible Extensions to the Selection Pass

The following are the possible extensions for the Selection Pass

1. Type support for intrinsics
2. Support for multiple return values
3. Support for hyperblocks

The return type of an LLVM instruction can be broadly classified into integer, floating point and pointer data types. The pointer data type covers complex types including data types of identifiers like pointers,arrays, structures and functions.

The migrate_end intrinsics created in the Selection Pass handle the integer and floating point data types efficiently but all the pointer data types are not supported as LLVM's intrinsic overloading mechanism does not currently support overloading on pointer types. Therefore, additional checks need to be performed to determine the exact return type of the LLVM instruction in the Selection Pass and the corresponding new migrate_end intrinsic is to be created. This also involves adding entries to the new intrinsics and the adding the types required to create these intrinsics in the Intrinsics.td file.

The additional migrate_end intrinsics with new pointer return types can be created depending on the requirement of the input test algorithm.

These may include the various return types such as

- Double pointers to float data types(**f32).
- Double pointers to double data types(**f64).

The presently created migrate_end intrinsics cover a major section of the possible return types of an LLVM IR instruction.

LLVM functions do not currently support multiple return values unlike some C programming language functions which can return multiple values in registers (e.g. "conj" function which returns the conjugate of a complex number, with the real and imaginary components, in two different registers). This multiple return value support is expected to be included in the upcoming releases(i..e next to llvm-2.2 version). The migrate_end intrinsics of the Selection Pass can then be modified to handle multiple return values.

As mentioned in [13], hyperblocks support the provision of instruction level parallelism in program codes. A hyperblock is formed by combining basicblocks having different execution paths and hence the terminator instructions might be present at places

other than the end of the block. The present Selection Pass only supports the sectioning of basic blocks with single terminator instruction at the end of the block and support is to be provided to hyperblocks with multiple terminator instructions.

The presented work is a step in the direction of hardware/software codesign of applications using LLVM compiler framework.

References

[1] Laurent Maillet-Contoz. Co Design Hardware/software partitioning is key. In *IEEE Potentials*, pages 13–14, Oct/Nov 1997.

[2] Giovanni Busonera, Salvatore Carta, Andrea Marongiu, and Luigi Raffo. Automatic Application Partitioning on FPGA/CPU Systems Based on Detailed Low-Level Information. In *DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 265–268, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Joerg Henkel and Rolf Ernst. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(2):273–290, 2001.

[4] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Chris Lattner. Introduction to the LLVM Compiler Infrastructure. In *2006 Gelato Itanium Conference and Expo (ICE)*, San Jose, California, Apr 2006.

[6] The LLVM Team. Documentation for the llvm system. http://www.llvm.org/docs/.

[7] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 205, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. http://llvm.cs.uiuc.edu.

[9] Jerzy Rozenblit and Buchenrieder Klaus, editors. *CoDesign Computer Aided Software/Hardware Engineering*, chapter Codesign: An Overview. IEEE Press, Piscataway, NJ, USA, 1994.

[10] Marc Stoettinger. Implementierung eines Bus Interfaces fur das LLVM VHDL Backend, jan 2008. Assignment.

[11] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[12] J. Johnson and M. Puschel. In search of the optimal Walsh-Hadamard transform. In *ICASSP '00: Proceedings of the Acoustics, Speech, and Signal Processing, 2000. on IEEE International Conference*, pages 3347–3350, Washington, DC, USA, 2000. IEEE Computer Society.

[13] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Patti. Hyperblock formation: a power/energy perspective for high performance VLIW architectures. In *ISCAS (4)*, pages 4090–4093. IEEE, 2005.

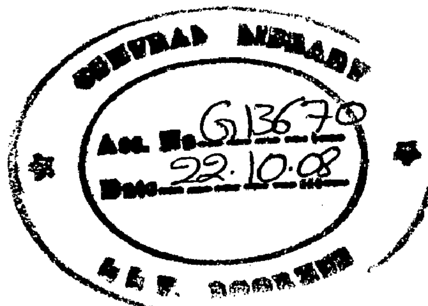
Appendix A

Binning Pass Header File

```
//Header file for Binning Pass

#ifndef CDFBINNING_H
#define CDFBINNING_H
#include "llvm/Pass.h"
#include "llvm/Function.h"
#include "llvm/BasicBlock.h"
#include "llvm/InstrTypes.h"
#include "llvm/Instructions.h"
#include "llvm/Analysis/LoopInfo.h"
#include <string>
#include<map>
using namespace llvm;
using namespace std;
namespace llvm {
class MyFunctionInfo //Function Information
{
public:
unsigned int ni,phi,nodes,edges,loop,dfbb,cfbb;
int y1,y2,y3;
float weight;
string name;
};
class MyBlockInfo //BasicBlock Information
```

```
{
public:
unsigned int mci,phi,arr,size,succ;
int x1,x2,x3;
float weight;
string name;
};
class CDFbinning :  public FunctionPass
{
public:
static char ID;
//map to store function information
map<Function* , MyFunctionInfo*> FInfo;
//map to store basic block information
map<BasicBlock* , MyBlockInfo*> BBInfo;
vector<Value*> GVInfo;
CDFbinning() :  FunctionPass((intptr_t)&ID) {}
//To specify that LoopInfo Pass is required for current transformation
void getAnalysisUsage(AnalysisUsage &AU) const {
AU.setPreservesAll();
AU.addRequired<LoopInfo>();
}
virtual bool runOnFunction(Function &F);
};
}
#endif
```



Appendix B

Binning Pass

```
//CDFbinning Pass -> Determines the extent of
//control flow and data flow
//at function and basic block levels

#include "llvm/Pass.h"
#include "llvm/Module.h"
#include "llvm/Function.h"
#include "llvm/BasicBlock.h"
#include "llvm/InstrTypes.h"
#include "llvm/Instructions.h"
#include "llvm/Support/CFG.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/Constants.h"
#include <map>
#include <vector>
#include "CDFbinning.h"
//constants used for framing empirical relations
#define kb1 0.6
#define kf1 0.7
#define kf2 1.5
#define kf3 1.2
#define kf4 0.3
#define p1 2
```

```
#define p2 0.7
#define p3 0.5
#define p4 0.5
#define q1 1
#define q2 0.5
#define q3 0.5
#define q4 0.5
using namespace llvm;
using namespace std;
namespace llvm {
//Iterating over all the functions in the input algorithm
bool CDFbinning::runOnFunction(Function &F) {
Module *M = F.getParent();
cerr<<"\n-------CDFbinning Pass Output-------\n\n";
llvm::cerr << "Function ->\t"<<F.getName() << ":\n\n";
cerr<<"\nglobal values:  "<<endl;
LoopInfo &LI=getAnalysis<LoopInfo>();
Loop *loop;
for(Module::global_iterator gi=M->global_begin(),gie=M->global_end();gi!=gie;gi++) {
Value *val1= dyn_cast<Value>(gi);
//get first encapsulated type from PointerType
const Type *t=gi->getType()->getElementType();
if(t->getTypeID()==Type::ArrayTyID) {
const ArrayType *at=cast<ArrayType>(t);
if(at) {
int notConstant=0;
User *inner=dyn_cast<User>(gi->getOperand(0)); //get array pointer
if(inner) {
//checks if all array values are constants
for (User::op_iterator opi = inner->op_begin(), ope = inner->op_end();
opi != ope; ++opi) {
Constant *c=dyn_cast<Constant>(opi);
if(!c){
notConstant=1;
break;
}
}
if(!notConstant) {//stores arrayvalues in map if array values are constants
GVInfo.push_back(val1);
cerr<<val1->getName()<<endl;
```

```
}
}
else
notConstant=1;
}
}
}
cerr<<endl;
MyFunctionInfo* MFI;
Function* pF = &F;
unsigned int nif=0,bb=1,phif=0,edgesf=0,dfbb=0,cfbb=0,tbnum=0;
int y1=0,y2=0,y3=0,y4=0;
float wf=0;
//Iterating over all the basic blocks of the function
for (Function::iterator b = F.begin(), be = F.end(); b != be; ++b,++bb){
MyBlockInfo* MBI;
BasicBlock *pb = b;
unsigned int phibb =0,mcibb=0,global=0;
int x1=-1,x2=0,x3=-1,x4=-1;
float wbb=0;
llvm::cerr << "Basic block " << pb->getName() << ":\n";
for ( BasicBlock::iterator i = pb->begin(), ie = pb->end(); i != ie; ++i){
//Checks whether GEP instruction accesses Global constant integer arrays memory using
map
if(isa<GetElementPtrInst>(i)){
GetElementPtrInst *GEPinst = dyn_cast<GetElementPtrInst>(i);
for( vector<Value*>::iterator iter = GVInfo.begin(); iter != GVInfo.end(); iter++ )
{
if(*iter == GEPinst->getPointerOperand()){
global++;
//iterating over def use chain of GEP instruction to find memory accesses dependant
// on GEP instruction
for(Value::use_iterator ui = GEPinst->use_begin(),uie = GEPinst->use_end();ui!=uie;++ui){
if(Instruction *piuser=dyn_cast<Instruction>(ui)) {
if((isa<LoadInst>(piuser)) || (isa<StoreInst>(piuser)) ){
global++;
}
}
}
}
```

```
}
}
//Checks whether Load instruction accesses Global constant integer arrays memory using
map
else if(isa<LoadInst>(i)){
LoadInst *Linst = dyn_cast<LoadInst>(i);
for( vector<Value*>::iterator iter =GVInfo.begin(); iter != GVInfo.end(); iter++ ) {
if(*iter == Linst->getPointerOperand())
global++;
}
}
//Checks whether
//Store instruction accesses Global constant integer arrays memory using map
else if(isa<StoreInst>(i)){
StoreInst *Sinst = dyn_cast<StoreInst>(i);
for( vector<Value*>::iterator iter =GVInfo.begin();
iter != GVInfo.end(); iter++ ) {
if(*iter ==Sinst->getPointerOperand())
global++;
}
}
if(isa<CallInst>(i) ||isa<GetElementPtrInst>(i)
|| isa<StoreInst>(i) || isa<TerminatorInst>(i) || isa<AllocationInst>(i) ||
isa<LoadInst>(i) || isa<FreeInst>(i))
mcibb++;//count of control constructs and memory access instructions
if(isa<PHINode>(i))
++phibb; //count of phi instructions
}
TerminatorInst *te = pb->getTerminator();
const BasicBlock *BB = pb;
loop = LI.getLoopFor(BB);
unsigned int benum=0;
if(LI.isLoopHeader(pb)){
benum= loop->getNumBackEdges();
tbnum+= benum;
}
llvm::cerr << "Number of instructions :\t" << pb->size()<< "\n";
//gives number of instructions from begin to first non phi
llvm::cerr << "Number of PHI Instructions :\t" << phibb<< "\n";
llvm::cerr << "Control flow and memory transfer instructions :\t" << mcibb << "\n";
```

```
llvm::cerr << "Constant global array memory access instructions :\t" << global << "\n";
llvm::cerr << "Number of successors :\t" <<te->getNumSuccessors() << "\t" <<"\n";
llvm::cerr << "Number of backedges :\t" << benum << "\t"<<"\n";
for(unsigned int nt=0;nt<(te->getNumSuccessors());++nt){
if(pb == te->getSuccessor(nt)){
x4=x4+2;
llvm::cerr << "Basic Block has self loop" <<"\n";
break;
}
}
edgesf+=te->getNumSuccessors();
phif+=phibb;
nif+= pb->size();
//empirical relations for determining extent of control
//and data flows at basic block level
//check for control flow - mem. and control instructions are more
if((mcibb-global) > kb1 * ((pb->size())-phibb))
x1 = 1;
//check for data flow - phi instructions absent and less than or
//equal to one successor
if(phibb == 0 && (te->getNumSuccessors() <= 1) )
x2 = -1;
// check for control flow - phi instructions are less than successors
if( phibb!=0 && phibb < te->getNumSuccessors())
x3 = 1;
wbb = p1 * x1 + p2 * x2 + p3 * x3 + p4 * x4;
if(wbb > 0){
llvm::cerr << "Basic block " << pb->getName()<< " has more of control flow\n\n";
cfbb++;
}
else{
llvm::cerr << "Basic block " << pb->getName()<< " has more of data flow\n\n";
dfbb++;
}
//storing the basicblock information in map
MBI = new MyBlockInfo();
MBI->mci = mcibb;
MBI->arr = global;
MBI->phi = phibb;
MBI->size = pb->size();
```

```
MBI->weight = wbb;
MBI->x1 = x1;
MBI->x2 = x2;
MBI->x3 = x3;
MBI->name = pb->getName();
MBI->succ = te->getNumSuccessors();
BBInfo[pb] = MBI;
}
llvm::cerr << "Number of instructions in function :\t" << nif<<"\n" ;
llvm::cerr << "PHI Instructions in function :\t" << phif <<"\n";
llvm::cerr << "Data flow basic blocks in function :\t" << dfbb<< "\n";
llvm::cerr << "Control flow basic blocks in function :\t" <<cfbb << "\n";
llvm::cerr << "Average basic block size :\t"<<nif/F.size()<<endl;
llvm::cerr << "Number of back edges in function:  \t" <<tbnum<<"\n" ;
llvm::cerr << "Number of nodes in function:  \t" << F.size()<<"\n" ;
llvm::cerr << "Number of edges of function:  \t" << edgesf<<"\n\n" ;
//empirical relations for determining extent of control and data
//flows at function level
//check for data flow - number of data flow blocks are more
if(dfbb > kf1 * F.size())
y1 = -1;
//check for control flow - number of control flow blocks are more
else if(cfbb > kf1 * F.size())
y1 = 1;
//check for control flow - edges by node ratio is more
if(edgesf > kf2 * F.size())
y2 = 1;
//check for data flow - edge by node ratio is very less
else if(edgesf < kf3 * F.size())
y2 = -1;
//check for control flow - number of nodes greater than avg.  basic block size
if ( F.size() > (nif/F.size()) )
y3 = 1;
//check for data flow - avg.  basic block size greater than number of nodes.
else if ((nif/F.size()) > F.size())
y3 = -1;
if( tbnum > kf4 * edgesf)
y4 = 1;
wf = q1 * y1 + q2 * y2 + q3 * y3+ q4 * y4;
if(wf>=0)
```

```
llvm::cerr << "This function has more of control flow\n\n\n";
else
llvm::cerr << "This function has more of data flow\n\n\n";
//storing the function information in map
MFI = new MyFunctionInfo();
MFI->ni = nif;
MFI->phi = phif;
MFI->nodes = F.size();
MFI->edges = edgesf;
MFI->loop = tbnum;
MFI->dfbb = dfbb;
MFI->cfbb = cfbb;
MFI->y1 = y1;
MFI->y2 = y2;
MFI->y3 = y3;
MFI->weight = wf;
MFI->name = F.getName();
FInfo[pF] = MFI;
GVInfo.clear();
return false; // Return 'false' when LLVM IR is not modified.
}
char CDFbinning::ID = 0; //CDFbinning Pass ID
}
namespace {
//Registering CDFbinning Pass
RegisterPass<CDFbinning> X("CDFbinning", "CDFbinning Analysis Pass");
}
```

Appendix C

Selection Pass

```
//Filter Pass -> Creates Intrinsics as place holders
//for instructions
//for target specific execution

#include <string>
#include "llvm/Intrinsics.h"
#include "llvm/Pass.h"
#include "llvm/Function.h"
#include "llvm/BasicBlock.h"
#include "llvm/InstrTypes.h"
#include "llvm/Instructions.h"
#include "llvm/Value.h"
#include "llvm/User.h"
#include "llvm/Support/CFG.h"
#include <map>
#include <vector>
#include "../CDFbinning/CDFbinning.h"
#include "llvm/Type.h"
#include "llvm/IntrinsicInst.h"
#include "llvm/DerivedTypes.h"
#include "llvm/Support/CommandLine.h"
#include "Deps.h"
using namespace llvm;
using namespace std;
namespace llvm{
```

```
static cl::opt<bool> filter_hw("filter-hw", cl::desc("filter for
dataflow intensive stufe (hw generation only)"));
struct Filter : public FunctionPass {
static char ID;
Filter() : FunctionPass((intptr_t)&ID) {}
//To specify that CDFbinning Pass is required for current transformation
void getAnalysisUsage(AnalysisUsage &AU) const {
AU.addRequired<CDFbinning>();
}
//Iterating over all the functions in the input algorithm
virtual bool runOnFunction(Function &F) {
CDFbinning &CDF = getAnalysis<CDFbinning>();
cerr<<"-------Filter Pass Output-------"<<endl;
cerr<< "\n\nFunction-> " << F.getName() << "\n\n";
cerr<<"non positive 'weight' => basic block with more data flow\n";
//Iterating over all the basic blocks of the function
for(Function::iterator b=F.begin(),be=F.end();b!=be;++b){
llvm::cerr << CDF.ID;
BasicBlock* bbp = dyn_cast<BasicBlock>(&*b);
llvm::cerr << endl<<bbp->getName()<< " weight:"<< (CDF.BBInfo[bbp])->weight <<"\n";
//Check for basic block with more of data flow
if((not filter_hw and CDF.BBInfo[bbp]->weight <=0)
or (filter_hw and CDF.BBInfo[bbp]->weight >0)){
//instantiating an object of Deps class in Deps.h file
Deps *dp = new Deps(bbp);
vector<Value*>
migrate_end_arglist;//Vector Container to handle values required for migrate_end
intrinsic creation
llvm::cerr<<"\nFunction type of migrate_begin intrinsic for basic block "
<< bbp->getName() << ":\n";
//Tys** points to returntype and operand types
const Type **Tys=(const Type**)calloc((dp->NrIncoming())+1,sizeof(Type*));
unsigned int i=0;
//Return Type of migrate_begin intrinsic
Tys[i]=Type::Int32Ty;
cerr << *Tys[i++]<<"\t";
for(vector<Value*>::iterator it=dp->mb_begin(),e=dp->mb_end();it!=e;it++) {
Tys[i]=(*it)->getType();
llvm::cerr<<*Tys[i++]<<"\t";
}
```

```
llvm::cerr<<"\n";
Module *M = bbp->getParent()->getParent();
// migrate_begin intrinsic
Function *FMigBegin = Intrinsic::getDeclaration(M, Intrinsic::migrate_begin,Tys);
intrinsic creation
// Creates and inserts intrinsic call before terminator instruction
CallInst *CI= new CallInst(FMigBegin,dp->mb_begin(),dp->mb_end(),
"migrate_begin",bbp->getTerminator());
cerr<<"migrate_begin intrinsic created"<<endl;
//PHI,terminator and intrinsic instructions not considered for
//migrate_end intrinsic creation
for(BasicBlock::iterator bi= bbp->getFirstNonPHI(),ie=CI;bi!=ie;++bi){
Instruction* pinst = bi;//Converting BasicBlock iterator to Instruction pointer.
CallInst* NewCI=0;
unsigned int k=0;
for(Value::use_iterator ui = pinst->use_begin(),uie = pinst->use_end();
((ui!=uie) && (k==0));++ui){//iterating over def use chain
if(Instruction *piuser=dyn_cast<Instruction>(ui)) {
//create migrate_end intrinsic only when instruction has
//phi, terminator or external dependencies.
if((piuser->getParent() != bbp) || ((piuser->getParent() == bbp) &&
((piuser->isTerminator()) || isa<PHINode>(piuser)))){
++k;
const Type *Ty = pinst->getType();
cerr<<"Return Type of instruction being replaced -> "<<*Ty<<endl;
Function *FMigEnd=0;
Value* cint1= dp->lookupOutgoing(pinst);//fetching x_enum from Deps.h
const Type *Tys[] = {pinst->getType(),Type::Int32Ty,Type::Int32Ty};
if(dyn_cast<IntegerType>(Ty)) {
//migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M, Intrinsic::migrate_end_int,Tys,1);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< " created"<<endl;
}
else
if(Ty->isFloatingPoint()) {
//migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M, Intrinsic::migrate_end_float,Tys,1);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< " created"<<endl;
```

```
}
else
if(isa<PointerType>(Ty)) {
const PointerType *PTy = dyn_cast<PointerType>(Ty);
const IntegerType *it=dyn_cast<IntegerType>(PTy->getContainedType(0));
const PointerType *itp=dyn_cast<PointerType>(PTy->getContainedType(0));
const ArrayType *ita=dyn_cast<ArrayType>(PTy->getContainedType(0));
if(it){
if(it->getBitWidth() == 8){
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_8,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else if(it->getBitWidth() == 32){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_32,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else
printf("\n create suitable pointer to integer type intrinsic\n");
else if((PTy->getContainedType(0))->isFloatingPoint()){
const Type *Ty = PTy->getContainedType(0);
if(Ty->getTypeID() == 1){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_float,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else if(Ty->getTypeID() == 2){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_double,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else
printf("\n create suitable pointer to floating point type intrinsic\n");
}
else if(itp){
const IntegerType *it1=dyn_cast<IntegerType>(itp->getContainedType(0));
```

```
if(it1 and it1->getBitWidth() == 32){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_ptr32,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else
printf("\n create suitable pointer to pointer type intrinsic\n");
}
else if(ita){
const IntegerType *it1=dyn_cast<IntegerType>(ita->getContainedType(0));
if(it1 and it1->getBitWidth() == 32){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_ptr32,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else if(it1 and it1->getBitWidth() == 8){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_ptr,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else
printf("\n create suitable pointer to array type intrinsic\n");
}
else
printf("\n create suitable pointer type intrinsic\n");
}
else if(isa<ArrayType>(Ty)){
const ArrayType *ita=dyn_cast<ArrayType>(Ty);
const IntegerType *it1=dyn_cast<IntegerType>(ita->getContainedType(0));
cerr<<"C Contained Type -> "<<*it1 <<"Size -> "<< it1->getBitWidth()<<endl;
if(it1 and it1->getBitWidth() == 32){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M, Intrinsic::migrate_end_ptr_8,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< " created"<<endl;
}
else if(it1 and it1->getBitWidth() == 8){
```

```
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M, Intrinsic::migrate_end_ptr_32,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< " created"<<endl;
}
else if((ita->getContainedType(0))->isFloatingPoint()){
const Type *Ty = ita->getContainedType(0);
if(Ty->getTypeID() == 1){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_float,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else if(Ty->getTypeID() == 2){
// migrate_end intrinsic creation
FMigEnd = Intrinsic::getDeclaration(M,Intrinsic::migrate_end_ptr_double,Tys);
const Type *MET = FMigEnd->getReturnType();
cerr<<"migrate_end intrinsic of return type " << *MET<< "created"<<endl;
}
else
printf("\n create suitable pointer to floating point type intrinsic\n");
}
else
printf("\n create suitable array type intrinsic\n");
}
else
printf("\n create suitable migrate_end intrinsic\n");
Value* cint= dp->mb_enum;//fetching mb_enum from Deps.h
migrate_end_arglist.push_back(cint);
migrate_end_arglist.push_back(cint1);
// Creates and inserts intrinsic call before terminator instruction
NewCI = new
CallInst(FMigEnd,migrate_end_arglist.begin(),migrate_end_arglist.end(),
"migrate_end",bbp->getTerminator());
migrate_end_arglist.clear();
}
}
}
//iterating over def use chain of current instruction
for(Value::use_iterator ui =pinst->use_begin(),uie = pinst->use_end();ui!=uie;){
```

```
if(Instruction *piuser=dyn_cast<Instruction>(ui)) {
//increment iterator early to avoid voiding it by deleting the instruction
++ui;
//replace dependencies(uses) only for phi nodes,terminator instructions
//and users outside basic block
if((piuser->getParent() != bbp) || ((piuser->getParent() == bbp) &&
((piuser->isTerminator()) || isa<PHINode>(piuser)))){
cerr<< piuser->getName()<< "instruction -> Uses of "<< pinst->getName()
<<" replaced with migrate_end intrinsic\n";
piuser->replaceUsesOfWith(pinst,NewCI);
}
}
else
cerr<<"Use not an instruction and hence not replaced!  "<<endl; //Error Message
}
}
//PHI, intrinsic & terminator instructions retained
for(BasicBlock::iterator bi=bbp->getFirstNonPHI(),ie=CI;bi!=ie;){
Instruction* pinst = bi;//Converting BasicBlock iterator to Instruction pointer.
//increment iterator early to avoid voiding it by deleting the instruction
++bi;
pinst->dropAllReferences(); //drop all references of each instruction
}
int k=0;
//PHI, intrinsic & terminator instructions retained
for(BasicBlock::iterator bi= bbp->getFirstNonPHI(),ie=CI;bi!=ie;){
Instruction* pinst = bi;//Converting BasicBlock iterator to Instruction pointer.
//increment iterator early to avoid voiding it by deleting the instruction
++bi;
if(pinst->use_empty()) //Delete instruction only when its uses are empty.
pinst->eraseFromParent(); //discard instructions from basic block
else{
++k;
cerr<<pinst->getName()<<" -> Uses not empty!\n "<< k <<endl; //Error Message
}
}
if(k==0)
cerr<<"\nPHI, intrinsic & terminator instructions retained and rest deleted\n";
delete dp;
}
```

```
}
return true; // Return 'true' when LLVM IR is modified.
}
};
char Filter::ID = 0; //Filter Pass ID
}
namespace{
RegisterPass<Filter> X("Filter", "Filter Pass"); //Registering Filter Pass
}
```

Appendix D

Deps Header File

```
//Header file to collect arguments for intrinsic functions

#ifndef DEPS_H
#define DEPS_H
#include "llvm/Pass.h"
#include "llvm/Module.h"
#include "llvm/Function.h"
#include "llvm/BasicBlock.h"
#include "llvm/InstrTypes.h"
#include "llvm/Instructions.h"
#include "llvm/Value.h"
#include <vector>
#include "llvm/Constants.h"
#include<map>
using namespace llvm;
using namespace std;
namespace llvm {
class Deps
{
public:
vector<Value*>mb_arglist;//migrate_begin arglist vector for each basic block
map<Instruction*,Value*>me_arglist;//migrate_end arglist map for each basic block
Value* mb_enum;
Deps(BasicBlock *bbp) {//FILL incoming vector here
```

```
static unsigned int migrate_begin_counter=0;
//add this number to the migrate_end_arglist
mb_enum=ConstantInt::get(Type::Int32Ty,migrate_begin_counter);
migrate_begin_counter++;
mb_arglist.push_back(mb_enum);
mb_arglist.push_back(mb_enum);
Function *func = bbp->getParent();
//adding function arguments to migrate_begin argument list
for(Function::arg_iterator agi=func->arg_begin(),agie= func->arg_end();agi!=agie;++agi){
Value *farg = dyn_cast<Value>(agi);
for(Value::use_iterator ui1 = farg->use_begin(),uie1 = farg->use_end();ui1!=uie1;++ui1){
if(Instruction *farguser=dyn_cast<Instruction>(ui1)) {
if(farguser->getParent()== bbp){
mb_arglist.push_back(farg);
cerr<<"function argument -> " <<farg->getName()<<"\n";
break;
}
}
}
}
for(BasicBlock::iterator i= bbp->begin(), ie=bbp->getTerminator(); i!=ie;++i){
Instruction* pinst = i;//Converting BasicBlock iterator to Instruction pointer.
if(isa<PHINode>(pinst))
mb_arglist.push_back(pinst);//storing phi values
else{
//Iterating over use def chain of each instruction of BB
for(User::op_iterator opi = pinst->op_begin(),opie = pinst->op_end();
opi!=opie;++opi){
Value *v = *opi;
//Check whether operand is an instruction
if(Instruction *vinst=dyn_cast<Instruction>(v)) {
if(vinst->getParent() != bbp){//Check for external dependencies of basic block.
mb_arglist.push_back(v); //storing operands in vector container
}
}
}
}
}
//add this number to the migrate_end_arglist
Value *mig_begin_num=ConstantInt::get(Type::Int32Ty,(mb_arglist.size()-2));
```

```
mb_arglist[1]=mig_begin_num;
//FILL map here
//position of migrate_end statement.
int migrate_end_position=0;
//PHI,terminator and intrinsic instructions not considered for migrate_end intrinsic
creation
for(BasicBlock::iterator bi= bbp->getFirstNonPHI(),ie=bbp->getTerminator();bi!=ie;++bi){
Instruction* pinst = bi;//Converting BasicBlock iterator to Instruction pointer.
unsigned int k=0;
for(Value::use_iterator ui = pinst->use_begin(),uie = pinst->use_end();((ui!=uie) &&
(k==0));++ui){//iterating over def use chain
if(Instruction *piuser=dyn_cast<Instruction>(ui)) {
//create migrate_end
//intrinsic only when instruction has phi, terminator or external dependencies.
if((piuser->getParent() != bbp) || ((piuser->getParent() == bbp) &&
((piuser->isTerminator()) || isa<PHINode>(piuser)))){
++k;
Value *cint1=ConstantInt::get(Type::Int32Ty,migrate_end_position);
me_arglist[pinst]=cint1;
}
}
}
migrate_end_position++;
}
};
typedef vector<Value*>::iterator mb_iterator;
typedef map<Instruction*,Value*>::iterator me_iterator;
unsigned int NrIncoming() {
return mb_arglist.size();
}
unsigned int NrOutgoing() {
return me_arglist.size();
}
Value* lookupOutgoing(Instruction *I) {
me_iterator mei=me_arglist.find(I);
if(mei==me_arglist.end()) {
Value *cint2=ConstantInt::get(Type::Int32Ty,-1);
return cint2;
}
else
```

```
return me_arglist[I];
}
mb_iterator mb_begin() {
return mb_arglist.begin();
}
mb_iterator mb_end() {
return mb_arglist.end();
}
me_iterator me_begin() {
return me_arglist.begin();
}
me_iterator me_end() {
return me_arglist.end();
}
};
}
#endif
```

Appendix E

Two Dimensional DFT

```
//Two Dimensional Discrete Fourier Transform

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<complex.h>
int twod_dft(unsigned int N,double a[][N]){
complex double w1,w2,p,q;
unsigned int n1,n2,k1,k2;
double (*real)[N] = (double (*)[N]) calloc(N * N,sizeof(double));
double (*imag)[N] = (double (*)[N]) calloc(N * N,sizeof(double));
printf("\n DFT of 2D array :\n");
for(k1=0;k1<N;++k1){
for(k2=0;k2<N;++k2){
q=0;
for(n1=0;n1<N;++n1){
p=0;
for(n2=0;n2<N;++n2){
w2 = -1 * (2 * 3.14159 * n2 * k2)/N * I;
p = p + a[n1][n2] * cexp(w2) ;
}
w1 = -1 * (2 * 3.14159 * n1 * k1)/N * I;
q = q + p * cexp(w1) ;
}
```

```
real[k1][k2] = creal(q);
imag[k1][k2] = cimag(q);
if(imag[k1][k2] >= 0)
printf("\t%5.3lf + i %5.3lf\t",real[k1][k2],imag[k1][k2]);
else
printf("\t%5.3lf - i %5.3lf\t",real[k1][k2],-1 * imag[k1][k2]);
}
printf("\n");
}
printf("\n\n");
return 1;
}
int main(){
unsigned int i,j,num;
printf("\nenter length(or breadth) of 2D array:  ");
scanf("%d",&num);
double (*p)[num] = (double (*)[num]) calloc(num * num,sizeof(double));
printf("\nenter elements :\n ");
for(i=0;i<num;++i)
for(j=0;j<num;++j)
scanf("\n%lf",&p[i][j]);
twod_dft(num,p);
free(p);
printf("\n\n");
return 1;
}
```

Appendix F

Walsh-Hadamard Transform

//Walsh Transform

```
#include<stdio.h>
#include<stdlib.h>
int walsh_transform(unsigned int N,unsigned int n,int *p){
unsigned int a,b,i,j,k,k1 = N/2,k2= 1;
int val;
int *q = (int *)calloc(N,sizeof(int));
int (*WHT)[N] = (int (*)[N])calloc(N * N,sizeof(int));
int (*A)[N] = (int (*)[N])calloc(N * N,sizeof(int));
int (*B)[N] = (int (*)[N])calloc(N * N,sizeof(int));
int (*M)[k1] = (int (*)[k])calloc(k1 * k1,sizeof(int));
int (*Ik)[k1] = (int (*)[k1])calloc(k1 * k1,sizeof(int));
for(i=0;i<k1;++i){//Ik - identity matrix
for(j=0;j<k1;++j){
if(i==j)
Ik[i][j] = 1;
}
}
for(i=0;i<k1;++i){//first product term - WHT=WHT2*Ik1
for(j=0;j<k1;++j){
WHT[i][j]=Ik[i][j];
WHT[i][j+k1]=Ik[i][j];
WHT[i+k1][j]=Ik[i][j];
```

```
WHT[i+k1][j+k1]=-Ik[i][j];
}
}
for(a=2;a<=n;++a){
k1=k1/2;
k2=k2*2;
if(a!=n){
for(i=0;i<k1;++i){//M = WHT2*Ik1
for(j=0;j<k1;++j){
M[i][j]=Ik[i][j];
M[i][j+k1]=Ik[i][j];
M[i+k1][j]=Ik[i][j];
M[i+k1][j+k1]=-Ik[i][j];
}
}
for(b=0;b<k2;++b){//A=Ik2*M
for(i=0;i<2*k1;++i){
for(j=0;j<2*k1;++j){
A[i+b*2*k1][j+b*2*k1]=M[i][j];
}
}
}
}
else{
for(i=0;i<(2*k2);i=i+2){//last product term - A=Ik2*WHT2
A[i][i]=1;
A[i][i+1]=1;
A[i+1][i]=1;
A[i+1][i+1]=-1;
}
}
for(i=0;i<N;++i){ //B=WHT*A
for(j=0;j<N;++j){
val=0;
for(k=0;k<N;++k){
val = val + WHT[i][k] * A[k][j];
}
B[i][j]=val;
}
}
```

```
for(i=0;i<N;++i){ //WHT=B && A=0
for(j=0;j<N;++j){
WHT[i][j]=B[i][j];
A[i][j]=0;
}
}
}
for(i=0;i<N;++i){//print Walsh Matrix
for(j=0;j<N;++j)
printf("%d ",WHT[i][j]);
printf("\n");
}
printf("\n\n");
for(i=0;i<N;++i){//q=WHT*p
for(j=0;j<N;++j)
q[i] = q[i] + WHT[i][j] * p[j];
}
printf("walsh transform of sequence:  \n");
for(i=0;i<N;++i)//print Tansform
printf("%d\t",q[i]);
printf("\n\n");
free(WHT);
free(A);
free(B);
free(M);
free(Ik);
return 1;
}
int main(){
unsigned int i,j,l,num,N=1,n=0;
int *p;
printf("\nenter number(num(>0)) = 2^n) of elements in sequence:  ");
scanf("%d",&num);
while(1){
if(num <= N)
break;
else{
N=N*2;
n++;
}
```

```
}
p = (int *)calloc(N,sizeof(int));
printf("\nenter sequence :\n ");
for(i=0;i<num;++i)
scanf("\n %d",&p[i]);
walsh_transform(N,n,p);
return 1;
}
```