POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

A Hybrid Autotuning Framework for Performance Optimization of Heterogeneous Systems

Advisor:
Prof. Cristina Silvano

Co-Advisor:
Prof. Gianluca Palermo
Dr. Amir H. Ashouri

Master Thesis of:
Mahdi Fani-Disfani (841258)
Puya Amiri (835886)

Academic Year 2016-2017

To all those who supported us...

Abstract in English

The increasing complexity of modern multi and manycore hardware design makes performance tuning of the applications a difficult task. While the aid of the successful past automatic tuning has been the execution time minimization, the new performance objectives have emerged comprise of energy consumption, computational cost, and area.

Automatic Tuning approaches range from the relatively non-intrusive (e.g., by using compiler options) to extensive code modifications that attempt to exploit specific architectural features. Intrusive techniques often result in code changes that are not easily reversible, which can negatively impact readability, maintainability, and performance on different architectures.

Therefore, more sophisticated methods capable of exploiting and identifying the trade-offs among these goals are required. We introduce a Hybrid Optimization framework to optimize the code for two main mutually competing criteria, e.g., execution time and resource usage in several layers starting from the original source code to a high-level synthesis level. Several effective tools and optimizations are involved, i.e., OpenTuner framework for building domain-specific multi-objective program autotuners, Annotation-based empirical tuning system called Orio, and a high-level synthesis tool named LegUp are the optimization components of our framework. The framework aims at improving both performance and productivity over a semi-automated procedure.

Our chain supports both architecture-independent and architecture-specific code optimization and can be adapted to any hardware platform architecture. After identifying the application's optimization parameters through OpenTuner, we pass the annotated code as input to Orio which generates many tuned versions and returns the version with the best performance. Furthermore, LLVM performs a number of optimization passes according to the Orio's result and finally, LegUp will use the LLVM output to synthesis for a particular target platform adding its optimizations.

We show that our automated approach can improve the execution time and resource usage on HLS through different optimization levels.

Key Words: Optimization, Performance, Autotuning, HLS, Compiler

Sommario

La crescente complessità del moderno design hardware multi e manycore rende l'ottimizzazione delle prestazioni delle applicazioni un compito difficile. Mentre l'aiuto della sintonizzazione automatica conclusa con successo è stata la riduzione dei tempi di esecuzione, sono emersi i nuovi obiettivi di prestazione che comprendono il consumo di energia, il costo computazionale e l'area.

Gli approcci di ottimizzazione automatica spaziano dal relativamente non intrusivo (ad esempio, utilizzando le opzioni del compilatore) alle estese modifiche del codice che tentano di sfruttare specifiche caratteristiche architettoniche. Le tecniche intrusive spesso portano a modifiche del codice che non sono facilmente reversibili, il che può avere un impatto negativo sulla leggibilitÃ , sulla manutenibilità e sulle prestazioni su diverse architetture.

Pertanto, sono necessari metodi più sofisticati in grado di sfruttare e identificare i trade-off tra questi obiettivi. Introduciamo una struttura di ottimizzazione ibrida per ottimizzare il codice per due criteri principali che si confrontano reciprocamente, ad es. Tempo di esecuzione e utilizzo delle risorse in diversi livelli, a partire dal codice sorgente originale fino ad un livello di sintesi di alto livello. Sono coinvolti diversi strumenti e ottimizzazioni efficaci, ovvero il framework OpenTuner per la creazione di autotuner di programmi multi-obiettivo specifici del dominio, il sistema di ottimizzazione empirica basato su Annotation chiamato Orio e uno strumento di sintesi di alto livello denominato LegUp sono i componenti di ottimizzazione del nostro framework . Il framework mira a migliorare sia le prestazioni che la produttività attraverso una procedura semi-automatica.

La nostra catena supporta l'ottimizzazione del codice indipendente dall' architettura e l'architettura specifica e può essere adattata a qualsiasi architettura di piattaforma hardware. Dopo aver identificato i parametri di ottimizzazione dell'applicazione tramite OpenTuner, passiamo il codice annotato come input a Orio che genera molte versioni ottimizzate e restituisce la versione con le migliori prestazioni. Inoltre, LLVM esegue un numero di passaggi di ottimizzazione in base al risultato di Orio e, infine, LegUp utilizzerà l'output LLVM per la sintesi di una determinata piattaforma target aggiungendo le sue ottimizzazioni.

Dimostriamo che il nostro approccio automatizzato può migliorare i

tempi di esecuzione e l'utilizzo delle risorse su HLS attraverso diversi livelli di ottimizzazione.

Parole chiave: Ottimizzazione, prestazioni, autotuning, HLS, compilatore

Acknowledgement

This Master of Science thesis has been carried out at the Department of Electronics and Computer (DEIB) at Politecnico di Milano University. The work has been performed within the HEAP laboratory of professors Cristina Silvano and Gianluca Palermo.

Being part of HEAP was a great and an invaluable experience. We take many memorable and enjoyable moments we spent with our colleagues while investigating the state-of-the-art problems. We thrived from all the moments and would like to appreciate all our teammates there.

First and foremost, we would very much like to thank our supervisors at Politecnico di Milano, Cristina Silvano and Gianluca Palermo who always had the answer to our questions and guided us towards the right way. Their constant encouragement and support throughout the project made it possible for us to complete the work.

We would also like to thank Dr. Amir H. Ashouri, a former Ph.D. student of HEAP and a current postdoctoral researcher at University of Toronto, Canada; It was a pleasure having his advice and excellent experiences in the field of computer architecture especially, compiler field.

Moreover, we would like to appreciate lifetime support of our perfect families whom always been there for us during the hard-times and good-times. Thank you so much for stop-less giving us the positive energy to carry-on and thanks for urging us to choose this path for our life.

Finally, we would like to thank everyone in Politecnico di Milano University circle, from our colleagues, secretaries to the professors, who got involved in such a way to let this checkpoint of our life happens.

Thank you all,

Contents

Abstract in English I

Abstract in Italian III

Acknowledgement V

List of Figures IX

List of Tables XIV

Listings XVI

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Contribution . . . 5
- 1.3 Thesis Outline . . . 6

2 Background 9
- 2.1 High Level Synthesis . . . 10
- 2.2 HLS Optimizations . . . 10
  - 2.2.1 Simple Loop Architecture Variations . . . 11
  - 2.2.2 Optimization: Merging Loops . . . 12
  - 2.2.3 Nested Loops . . . 13
  - 2.2.4 Optimization: Flattening Loops . . . 14
  - 2.2.5 Pipelining Loops . . . 15
- 2.3 External Tools and Frameworks . . . 17
  - 2.3.1 Profiling Tools . . . 17
  - 2.3.2 Orio . . . 18
  - 2.3.3 OpenTuner . . . 20
  - 2.3.4 LegUp . . . 22
- 2.4 Target Architectures . . . 27
  - 2.4.1 MIPS . . . 27
  - 2.4.2 Emulation Environments . . . 29
  - 2.4.3 Development Board . . . 29

3 State of the Art **33**
3.1 Related Works 37

4 Proposed Methodology **41**
4.1 Hybrid Optimization Framework details 42
4.1.1 Tool chain and platforms 42
4.1.2 Code/Application Autotuning 42
4.1.3 HLS Tuning 47

5 Experimental Results **51**
5.1 Introduction 51
5.1.1 Preliminary Definitions 55
5.2 Block-Wise Matrix Multiply Benchmark 57
5.2.1 Code Annotation 59
5.2.2 Result Tables 64
5.2.3 Performance Diagrams 66
5.3 AXPY Benchmark 72
5.3.1 Result Tables 72
5.3.2 Performance Diagrams 73
5.4 Matrix Multiply Benchmark 75
5.4.1 Result Tables 75
5.4.2 Performance Diagrams 76
5.5 DFMUL Benchmark 78
5.5.1 Result Tables 79
5.5.2 Performance Diagrams 79
5.6 GSM Benchmark 82
5.6.1 Result Tables 83
5.6.2 Performance Diagrams 83
5.7 DFADD Benchmark 86
5.7.1 Result Tables 86
5.7.2 Performance Diagrams 88
5.8 ADPCM Benchmark 90
5.8.1 Result Tables 91
5.8.2 Performance Diagrams 91
5.9 MIPS Benchmark 94
5.9.1 Result Tables 94
5.9.2 Performance Diagrams 95
5.10 DFDIV Benchmark 97
5.10.1 Result Tables 97
5.10.2 Performance Diagrams 100
5.11 DFSIN Benchmark 102
5.11.1 Result Tables 102
5.11.2 Performance Diagrams 103
5.12 JPEG Benchmark 105

5.12.1 Result Tables 105
5.12.2 Performance Diagrams 107

6 Conclusion and Future Works 111
6.1 Conclusion 111
6.2 Future Works 112

Bibliography 115

A Mandelbrot 121

B Loop Transformation Modeling 141
B.1 Source to Source Transformation 141
B.2 Theory of code transformation 142
B.2.1 Polyhedral Model for nested loops 142
B.3 Loop Transformation 144

C Codes 147
C.1 AXPY 147
C.2 ADPCM 149
C.3 DFADD 151
C.4 DFMUL 156
C.5 DFSIN 159
C.6 DFDIV 163
C.7 GSM 167
C.8 MIPS 168
C.9 JPEG 170
C.10 Matrix Multiplication 173
C.11 Bash Scripts 175
C.12 LLVM-opt Recipes for HLS 181

List of Figures

1.1 Mandelbrot Area Delay on different HLS approaches 4
1.2 Hybrid Optimization Framework Overview 6

2.1 Extraction of addition loop into datapath and control logichow transformations affect the hardware implementation. . 11
2.2 Consecutive loops for addition and multiplication within a function 13
2.3 Merged addition and multiplication loops 14
2.4 Example code for nested loops adding the elements of two dimensional arrays 14
2.5 Control flow through the matrix addition, without flattening (clock cycles in circles) 16
2.6 Overview of Orio's code generation and empirical tuning process. 18
2.7 Integration of Pluto and Orio 19
2.8 OpenTuner structure 21
2.9 OpenTuner components 21
2.10 LegUp Design Flow 24
2.11 Target System Architecture 24
2.12 Design flow with legup 26
2.13 MIPS datapath with control unit. 28
2.14 Terasic , DE1 SoC 30
2.15 Altera's Cyclon V SoC Architecture 31

3.1 ASV triangles for the conventional and auto-tuned approaches to programming. 34
3.2 High-level discretization of the autotuning optimization space. 34
3.3 Visualization of three different strategies for exploring the optimization space: (a) Exhaustive search, (b) Heuristically-pruned search, and (c) hill-climbing. Note, curves denote combinations of constant performance. The gold star represents the best possible performance. 35

3.4 Comparison of traditional compiler and autotuning capabilities. Low-level optimizations include loop transformations and code generation.†only via OpenMP pragmas. 36

4.1 Methodology Diagram 43
4.2 HOF Tool Chain 44
4.3 Methodology Diagram, first section 46
4.4 Methodology Diagram, second section 48

5.1 Brief description and source of the CHStone benchmark programs [1]. 53
5.2 Source-level characteristics [1]. 54
5.3 The number of states and resource utilization in RTL description [1]. 55
5.4 Pareto Curve 56
5.5 Graphical interpretation of blocked matrix multiply The innermost (j, k) loop pair multiplies a $1 * bsize$ sliver of A by a $bsize * bsize$ block of B and accumulates into a $1 * bsize$ sliver of C 58
5.6 The BWMM *Wall-Clock Time* diagram of the original code vs. different optimization levels 66
5.7 BWMM Speedup w.r.t baseline 67
5.8 BWMM Logic Utilization Efficiency w.r.t baseline 67
5.9 Area Delay of baseline vs. HOF 68
5.10 BWMM cache miss rate w.r.t baseline 69
5.11 BWMM different phases speedup 70
5.12 BWMM different phases logic utilization efficiency 71
5.13 BWMM different phases Area Delay 71
5.14 AXPY Speedup diagram 73
5.15 AXPY logic utilization efficiency diagram 74
5.16 AXPY Area Delay diagram 74
5.17 Matrix Multiply Speedup diagram 76
5.18 Matrix Multiply Utilization diagram 77
5.19 Matrix Multiply Area Delay diagram 77
5.20 DFMUL Profiling. The Call-graph depicting calling relationships and computational needs of each function 78
5.21 DFMUL Speedup diagram 80
5.22 DFMUL Utilization diagram 81
5.23 DFMUL Area Delay diagram 81
5.24 GSM Profiling. The Call-graph depicting calling relationships and computational needs of each function 82
5.25 GSM Speedup diagram 84
5.26 GSM Utilization diagram 85
5.27 GSM Area Delay diagram 85

5.28 DFADD Profiling. The Call-graph depicting calling relationships and computational needs of each function. 86
5.29 DFADD Speedup diagram 88
5.30 DFADD Utilization diagram 89
5.31 DFADD Area Delay diagram 89
5.32 ADPCM Profiling, ADPCM Call-graph depicting calling relationships and computational needs of each function 90
5.33 ADPCM Speedup diagram 92
5.34 ADPCM Utilization diagram 93
5.35 ADPCM Area Delay diagram 93
5.36 MIPS Profiling, the Call-graph depicting calling relationships and computational needs of each function 94
5.37 MIPS Speedup diagram 96
5.38 MIPS Utilization diagram 96
5.39 MIPS Area Delay diagram 97
5.40 DFDIV Profiling, The Call-graph depicting calling relationships and computational needs of each function 98
5.41 DFDIV Speedup diagram 100
5.42 DFDIV Utilization diagram 101
5.43 DFDIV Area Delay diagram 101
5.44 DFSIN Profiling. The call-graph depicting calling relationships and computational needs of each function 102
5.45 DFSIN Speedup diagram 104
5.46 DFSIN Utilization diagram 104
5.47 DFSIN Area Delay diagram 105
5.48 JPEG Profiling. The Call-graph depicting calling relationships and computational needs of each function 106
5.49 JPEG Speedup diagram 108
5.50 JPEG Utilization diagram 108
5.51 JPEG Area Delay diagram 109

6.1 Normalized Execution Time of All Benchmarks w.r.t Baseline 112
6.2 Speedup of all benchmarks besides their corresponding logic utilization efficiency w.r.t original 113

A.1 Pareto Curve 124
A.2 Mandelbrot call graph 126
A.3 Mandelbrot scheduling details 127
A.4 Loop Pipelining 127
A.5 Loop Pipelining Scheduling 129
A.6 Schedule for loop lp1 in the transformed Mandelbrot code. . . 133
A.7 Mandelbrot Fmax on different HLS approaches 136
A.8 Mandelbrot Wall Clock Time on different HLS approaches . . 137
A.9 Mandelbrot Clock Period on different HLS approaches 137

A.10 Mandelbrot Clock cycles on different HLS approaches 138
A.11 Mandelbrot DSP blocks on different HLS approaches 139
A.12 Mandelbrot Area*Delay on different HLS approaches 139
A.13 Mandelbrot ALMs on different HLS approaches 140

B.1 An Example of Array Access Pattern 143
B.2 An An example of column access pattern. There are 4 patterns with different traverse directions. 143
B.3 An example of diagonal access pattern with *slope = 1*. There are 4 patterns with different traverse directions. 145

List of Tables

3.1 Summary of selected related projects using autotuning 37

5.1 OpenTuner in HLS Framework 65
5.2 Pure Hardware AXPY HLS Analysis 72
5.3 Pure Software AXPY HLS Analysis 72
5.4 Hardware/Software AXPY HLS Analysis 73
5.5 Pure Hardware Matrix Multiply HLS Analysis 75
5.6 Pure Software Matrix Multiply HLS Analysis 75
5.7 Hardware/Software Matrix Multiply HLS analysis 76
5.8 Pure Hardware DFMUL HLS Analysis 79
5.9 Pure Software DFMUL HLS Analysis 79
5.10 Hardware/Software DFMUL HLS Analysis 80
5.11 Pure Hardware GSM HLS Analysis 83
5.12 Pure Software GSM HLS Analysis 83
5.13 Hardware/Software GSM HLS Analysis 84
5.14 Pure Hardware DFADD HLS Analysis 87
5.15 Pure Software DFADD HLS Analysis 87
5.16 Hardware/Software DFADD HLS Analysis 87
5.17 Pure Hardware ADPCM HLS Analysis 91
5.18 Pure Software ADPCM HLS Analysis 91
5.19 Hardware/Software ADPCM HLS Analysis 92
5.20 Pure Hardware MIPS HLS Analysis 95
5.21 Pure Software MIPS HLS Analysis 95
5.22 Pure Hardware DFDIV HLS Analysis 99
5.23 Pure Software DFDIV HLS Analysis 99
5.24 Hardware/Software DFDIV HLS Analysis 100
5.25 Pure Hardware DFSIN HLS Analysis 103
5.26 Pure Software DFSIN HLS Analysis 103
5.27 Pure Hardware JPEG HLS Analysis 107
5.28 Pure Software JPEG HLS Analysis 107

A.1 Mandelbrot HLS Analysis 122

C.1 Three sets of favourable LLVM-opt flags for adpcm 182

C.2 Three sets of favourable LLVM-opt flags for blowfish 183
C.3 Three sets of favourable LLVM-opt flags for dfadd 184

Listings

2.1 Example code for a loop adding the elements of two arrays . 11
5.1 BWMM original code 58
5.2 Autotuner program for OpenTuner to find optimum parameters for BWMM 59
5.3 Annotated BWMM 60
5.4 BWMM Orio transformation specification 61
5.5 BWMM tuned and transformed code 62
A.1 Mandelbrot Kernel 123
A.2 Transformed Mandelbrot Kernel 130
A.3 Threaded Mandelbrot Kernel 132
C.1 Annotated AXPY code 147
C.2 AXPY Orio specification code to use for S2S transformation . 147
C.3 Tuned and transformed AXPY code after S2S transformation by Orio 148
C.4 Annotated ADPCM code 149
C.5 ADPCM Orio specification code to use for S2S transformation 149
C.6 Tuned and transformed ADPCM code after S2S transformation by Orio 150
C.7 Annotated DFADD code 151
C.8 DFADD Orio specification code to use for S2S transformation 151
C.9 Tuned and transformed DFADD code after S2S transformation by Orio 152
C.10 Annotated DFMUL code 156
C.11 DFMUL Orio specification code to use for S2S transformation 156
C.12 Tuned and transformed DFMUL code after S2S transformation by Orio 157
C.13 Annotated DFSIN code 159
C.14 DFSIN Orio specification code to use for S2S transformation 160
C.15 Tuned and transformed DFSIN code after S2S transformation by Orio 160
C.16 Annotated DFDIV code 163
C.17 DFDIV Orio specification code to use for S2S transformation 163
C.18 Tuned and transformed DFDIV code after S2S transformation by Orio 164

C.19 Annotated GSM code 167
C.20 GSM Orio specification code to use for S2S transformation . 167
C.21 Tuned and transformed GSM code after S2S transformation by Orio 168
C.22 Annotated MIPS code 169
C.23 MIPS Orio specification code to use for S2S transformation . 169
C.24 Tuned and transformed MIPS code after S2S transformation by Orio 169
C.25 Annotated JPEG code 170
C.26 JPEG Orio specification code to use for S2S transformation . 171
C.27 Tuned and transformed JPEG code after S2S transformation by Orio 171
C.28 Annotated Matrix Multiplication code 173
C.29 Matrix Multiplication Orio specification code to use for S2S transformation 173
C.30 Tuned and transformed Matrix Multiplication code after S2S transformation by Orio 174
C.31 GUI and command line tool for the main chain tests 175
C.32 GProf profiler 178
C.33 Perf profiler 179
C.34 Valgrind profiler 180

Chapter 1

Introduction

The performance of a software application crucially depends on the quality of its source code. The increasing complexity and multi/many-core nature of hardware design have transformed code generation, whether done manually or by a compiler, into a complex, time-consuming, and error-prone task which additionally suffers from a lack of performance portability [2].

To mitigate these issues, a new research field, known as autotuning, has gained increasing attention. Autotuners are an effective approach to generate high-quality portable code. They produce highly efficient code versions of libraries or applications by generating many code variants which are evaluated on the target platform, often delivering high-performance code configurations which are unusual or not intuitive [2].

1.1 Motivation

Earlier autotuning approaches were mainly targeted at improving the execution time of a code [3]. However, recently other optimization criteria such as energy consumption or computing costs are gaining interest. In this new scenario, a code configuration that is found to be optimal for low execution time might not be optimal for another criterion or vice-versa.

Therefore, there is no single solution to this problem that can be considered optimal, but a set, namely the Pareto set, of solutions (i.e., code configurations) representing the optimal trade-off among the different optimization criteria. Solutions within this set are said to be non-dominated: any solution within it is not better than the others for all the considered criteria.

This multi-criteria scenario requires a further development of autotuners, which must be able to capture these trade-offs and offer them using either the whole Pareto set or a solution within it. Although there is a growing amount of related work considering the optimization of several criteria [4, 5, 6, 7, 8], most of them consider two criteria simultaneously at most, and many fail in

capturing the trade-off among the objectives.

There is two factor in common to most of these autotuning methods and frameworks, first is that they focus exclusively on a single optimization technique and objective such as execution time, memory behavior or resource consumption. Only little support exists for code optimizers that deal with trade-offs between multiple, conflicting goals. Second, they do not consider the effect of optimization mixture in different application execution levels from software code to hardware synthesis; in particular, the effect of loop transformation on HLS which has been highlighted as the most effective optimization in Mandelbrot set [9] benchmark. The Mandelbrot set and the result of performance evaluation of different possible HLS optimizations is extensively described in Appendix A.

Figure A.12 shows a key plot that motivates the proposed framework in this thesis. Taking into account the importance of *Area-Delay* (indicating the combination of execution time an resource consumption), the figure shows that *Loop Transformations* or generally *Code Transformations* yields the best Area-delay product meaning that the concept of source to source compilation to reform the programming structure not only affects in the context of sequential software but also extremely impacts on *High-Level Synthesis* and subsequent circuitries in terms of area and speed.

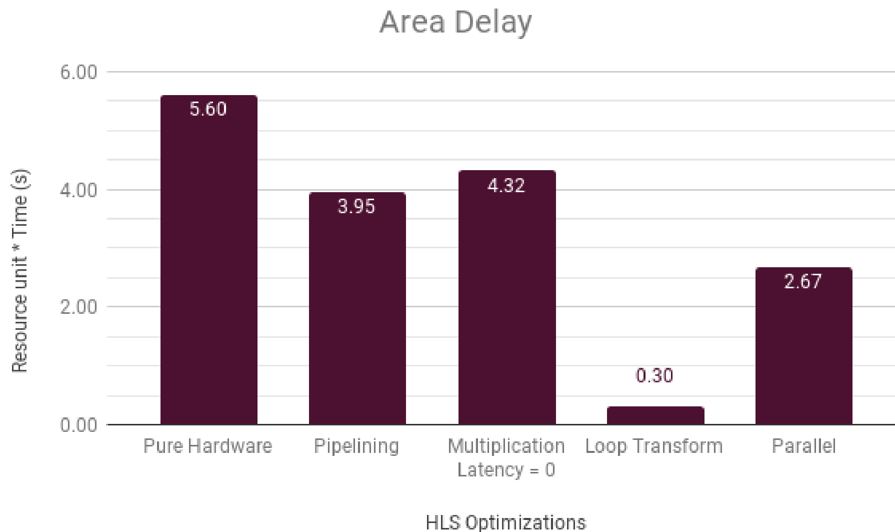

Figure 1.1: Mandelbrot Area Delay on different HLS approaches

In this thesis, we introduce a novel multi-objective hybrid optimization framework to optimize two different criteria: execution time, resource usage. The basic idea is to integrate different tools and semi-automatically find an effective set of optimizations with proper parameter settings (e.g., tile sizes

and unrolling factors) for individual code regions. It is based on the application parameter tuning, Source-to-Source transformation, compiler and High-Level Synthesis optimizations. We examine the obtained results in detail by analyzing and illustrating the interaction between software optimizations and hardware synthesis. Our approach is generic and can be applied to arbitrary transformations and parameter settings. We demonstrate our implementation by exploring the trade-off between execution time and resource efficiency when tuning loop transformations applied.

1.2 Contribution

Our method, which is based on a combination of software level optimizations and hardware level optimization techniques, is described in chapter 3. In this section, we illustrate an overview of our solution, highlighting the five main components: the code analyzer and profiler, application parameter autotuning, source-to-source code transformation, compiler Optimizations and the HLS optimization techniques. Each of them can be individually customized or exchanged by alternative implementations.

The labels in Figure 1.2 follow the processing of a program within our framework. An input code will be loaded by our Profiler Valgrind, analyzed and autotuned by application parameter autotuner. Then annotated by S2S coded transformer Orio, transformed and passed to our compiler LLVM to optimize and used as the input of the HLS with further optimization before synthesis.

For the region of the hot function which is the most time intensive part of the application found by the profiler, the application parameter autotuner determines the best value for the candidate parameters. The autotuned code will be annotated and transformed by the source-to-source code transformer which describes generic sequences of code transformations using unbound parameters for tunable properties (e.g., tile sizes, unrolling factors or vectorization). The associated transformation skeletons and some (optional) parameter constraints, will be evaluated (executed) on the target system automatically by the transformer and the best transformation with chosen parameter configuration is passed on to the compiler. At this point, the compiler conducts code optimization by using efficient compiler flags iteratively selecting standard sets of flags. After the compilation flow, corresponding to the target architecture for synthesis, some High-Level Synthesis optimizations will be applied to the code through synthesis tool.

In the end, our framework will generate the report including details regarding the represented trade-off between execution time and resource usage of the input application with the optimized code and its optimization configuration in each step which is application specific and architectural-dependent. However, the ultimate method to utilize automatically all the

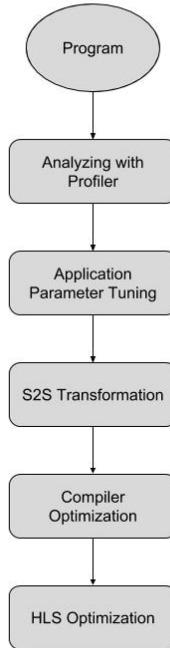

Figure 1.2: Hybrid Optimization Framework Overview

steps of the framework and gained the opportunity of dynamically customizing attributes and using better strategy to choose compiler flags like machine learning is beyond the scope of this thesis and left for future research.
The major contributions of this work include:

- The design of a novel hybrid optimization framework facilitating the consideration of multiple conflicting criteria simultaneously by passing through a multi-objective optimization chain

- The combination of the application parameter tuning and the search for optimal loop transformation to minimize execution time and resource usage efficiency

- The development of a hybrid optimization framework capable of solving the combined problem using an effective number of optimization steps

1.3 Thesis Outline

The thesis is structured as follows:

- In Chapter 2 there is the description of the background and the external tools and frameworks on which this work is based on. It introduces related concepts and works, and target architecture of this thesis.
- Chapter 3 gives an overview of the state-of-the-art regarding methodologies and techniques developed in this thesis, focusing on the description of the OpenTuner autotuner, Orio S2S transformer, and the LegUp HLS framework.
- In Chapter 4 there is the detailed description of the framework implemented, in terms of components and behavior. We describe the methodology proposed step by step and the logic structure developed for building Hybrid Optimization Framework.
- In Chapter 5 contains all the experiments we made to validate our proposed methodology and to gather information on the framework behavior.
- Chapter 6 contains the conclusions of this work, with the description of both benefits and limitations, and the list of possible future works that can be done in order to improve and expand this thesis.
- Appendix A elaborates Mandelbrot algorithm and evaluates the performance of different synthesis approaches. The results explain the main motivation behind proposing the framework.
- Appendix B is about modeling the transformations. As the *PluTo* integrated into *Orio* implements loop transformations based on polyhedral modeling, also taking into account various loop optimizations that can be enabled by LLVM *opt* command, in this appendix we describe the theory of source code optimization techniques.
- Appendix C covers almost all the codes used for experiments, excluding the unmodified baselines[1]. There you can find annotated codes, specifications, Tuned codes and framework scripts.

Finally, there is the bibliography.

[1]Original unmodified codes

Chapter 2

Background

The size and complexity of scientific computations are increasing at least as fast as the improvements in processor technology. Programming such scientific applications are hard, and optimizing them for high performance is even harder. This situation results in a potentially large gap between the achieved performance of applications and the available peak performance, with many applications achieving 10 percent or less of the peak. A greater concern is the inability of existing languages, compilers, and systems to deliver the available performance for the application through fully automated code optimizations.

Delivering performance without degrading productivity is crucial to the success of scientific computing. Scientific code developers generally attempt to improve performance by applying one or more of the following three approaches: manually optimizing code fragments; using tuned libraries for key numerical algorithms; and, less frequently, using compiler-based source transformation tools for loop-level optimizations. Manual tuning is time-consuming and impedes readability and performance portability.

Tuned libraries often deliver great performance without requiring significant programming effort, but then can provide only limited functionality. General-purpose source transformation tools for performance optimizations are few and have not yet gained popularity among computational scientists, at least in part because of poor portability and steep learning curves.

Profiling could be an alternative solution to fill those gaps and is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler). Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.

2.1 High Level Synthesis

" High-level synthesis raises the design abstraction level and allows rapid generation of optimized RTL hardware for performance, area, and power requirements ..."

Tim Cheng

High-level synthesis (HLS), sometimes referred to as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavioral synthesis, is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior

High-level synthesis (HLS) raises the level of abstraction for hardware design by allowing software programs written in a standard language to be automatically compiled into hardware. First proposed in the 1980s, HLS has received renewed interest in recent years, notably as a design methodology for field-programmable gate arrays (FPGAs). Although FPGA circuit design historically has been the realm of hardware engineers, HLS offers a path toward making FPGA technology accessible to software engineers, where the focus is on using FPGAs to implement accelerators that perform computations with higher throughput and energy efficiency relative to standard processors. We believe, in fact, that FPGAs (rather than ASICs) will be the vehicle through which HLS enters the mainstream of IC design, owing to their reconfigurable nature. With custom ASICs, the silicon area gap between human-designed and HLS-generated RTL leads directly to (potentially) unacceptably higher IC manufacturing costs, whereas with FPGAs, this is not the case, as long as the generated hardware fits within the available target device.

2.2 HLS Optimizations

How code transformations affect the hardware implementation?

Loops are used extensively in software programming, and constitute a very succinct and natural method of expressing operations that are repetitive in some way. They are also used in a similar manner in HDLs, for example, to iteratively instantiate and connect circuit components. However, an important difference is that the designer can prompt the loop(s) to be synthesized in different ways. This contrasts with the use of loops in HDL, where code expressing loops is directly converted into hardware, usually resulting in prescribed and fixed architectures.

Usually, the repetitive operations described by the loop are realized by a single piece of hardware implementing the body of the loop. To take a simple

illustrative example, if a loop was designed to add the individual elements of two, 12 element arrays, then conceptually the implementation would involve a single adder (the body of the loop), shared 12 times according to the number of loop iterations.There is some latency associated with each iteration of the loop, and in this case, the latency is affected by interactions with the memory interfaces at the inputs and output of the function.

Additional clock cycles are also required for entering and exiting the loop. Code for this example is provided in 2.1. Analysis of HLS synthesis with default settings shows that the overall latency is 26 clock cycles: 2 cycles each for 12 iterations (including reading the inputs from memory, adding, and writing the output to memory), and a further two clock cycles for entering and exiting the loop. Execution is illustrated in Figure 2.1.

Listing 2.1: Example code for a loop adding the elements of two arrays

```
void add_array (short c[12], short a[12], short b[12])
{
    short j;
     // loop variable
add_loop: for (j=0;j<12;j++) {
        c[j] = a[j] + b[j];
    }
}
```

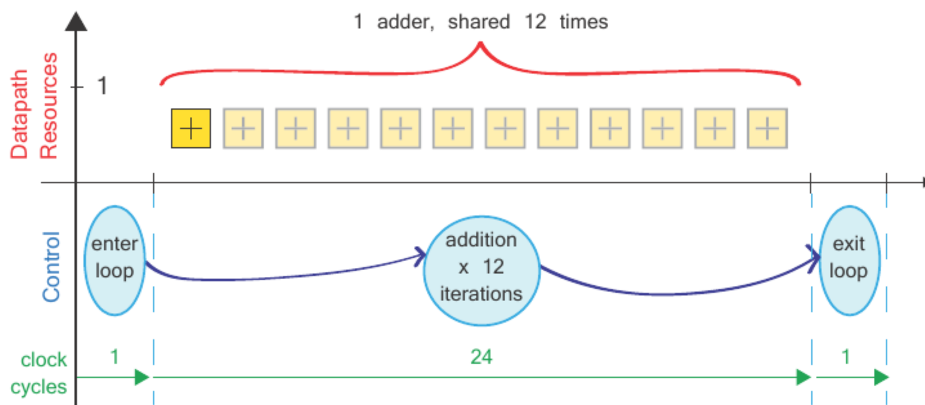

Figure 2.1: Extraction of addition loop into datapath and control logichow transformations affect the hardware implementation.

2.2.1 Simple Loop Architecture Variations

The default, rolled loop implementation may not always be desirable, but there are alternatives for implementing a loop

- **Unrolled:** In a rolled implementation, a single instance of the hardware is inferred from the loop body and shared to the maximum extent. Unrolling a loop means that instead the hardware inferred from the loop body is created up to N times, where N is the number of loop iterations. In practice, the number of instances may be lower than N if other limiting factors are identified in the design, such as memory operations. The clear disadvantage of the unrolled version is that it consumes much larger area on the device than the rolled design, but the advantage is increased throughput.

- **Partially unrolled** This constitutes a compromise between rolled and unrolled and is typically used when a rolled implementation does not provide high enough throughput. If a rolled architecture represents minimal hardware cost but maximal time sharing (lowest throughput), and an unrolled architecture represents maximal hardware cost but minimal sharing (highest throughput), then we may try to find a different balance between the two. Control is exerted by the use of directives, and a number of different positions in the trade-off may be possible.

With reference to the upper section of 2.1, which depicts a rolled architecture, fully or partially unrolling the loop would cause the number of datapath resources (adders) to increase, but to be shared to a lesser extent. Meanwhile, in the lower section of the diagram, the large, central state constituting the addition of array elements would require fewer clock cycles to complete. When fully unrolled, the implementation effectively does not contain a loop, and as a result, the loop entry and exit clock cycles are saved, too.

With these observations in mind, the decision to select a rolled, unrolled, or partially unrolled implementation for the loop will be based on the specific requirements of the application, particularly in terms of the target throughput and any constraint on area utilization that may apply.

2.2.2 Optimization: Merging Loops

In some cases, there might be two loops occurring one after the other in the code. For instance, the addition loop in the example of 2.1 might be followed by a similar loop which multiplies the elements of the two arrays. Assuming that the loops are both rolled (according to the default mode), a possible optimization, in this case, would be to merge the two loops, such that there is only one loop, with both the addition and multiplication operations being conducted within the single loop body.

The advantage of merging loops may not be immediately obvious, but in fact, it relates to the control aspect of the design. Control is realized in the form of a Finite State Machine (FSM), with each loop corresponding

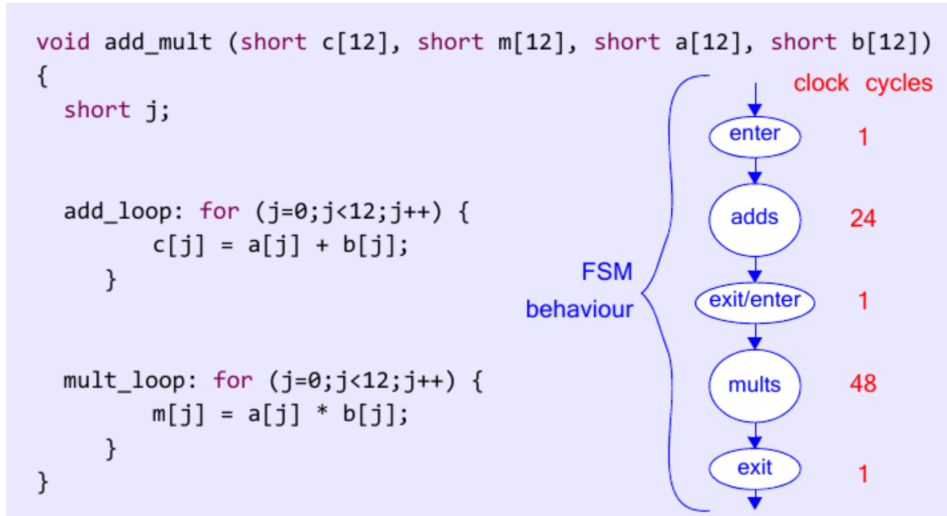

Figure 2.2: Consecutive loops for addition and multiplication within a function

to at least one state; thus the FSM can be simplified due to the merging of loops, as this results in fewer loops overall, and thus fewer FSM states. This is demonstrated by the code examples in Figures 2.2 and 2.3. The first example shows two separate loops, one each for addition and multiplication of the arrays, while the second shows the effect of merging the loops to create a single loop.

The add_loop represents 12 iterations of an addition operation (which takes 2 clock cycles), while the mult_loop represents 12 iterations of a multiplication operation (which takes 4 clock cycles). Therefore, the overall latencies of the two loops are 24 and 48 clock cycles, respectively. Merging the loops has the effect that the latency of the new, combined loop reduces the length of the original two loops, i.e. 48 clock cycles. One further clock cycle is saved due to the removed loop transition, i.e. the 'exit/enter' state in Figure 2.2.

2.2.3 Nested Loops

Another common configuration is to nest loops, i.e., place one loop inside another. There may even be multiple levels of nesting. To give an example of a 2-level nested loop, suppose we extend our array addition example from linear arrays to 2-dimensional arrays. Mathematically, this is equivalent to adding two matrices, as shown in 2.1.

$$\begin{bmatrix} f_{00} & f_{01} & f_{02} & f_{03} \\ f_{10} & f_{11} & f_{12} & f_{13} \\ f_{20} & f_{21} & f_{22} & f_{23} \end{bmatrix} = \begin{bmatrix} d_{00} & d_{01} & d_{02} & d_{03} \\ d_{10} & d_{11} & d_{12} & d_{13} \\ d_{20} & d_{21} & d_{22} & d_{23} \end{bmatrix} + \begin{bmatrix} e_{00} & e_{01} & e_{02} & e_{03} \\ e_{10} & e_{11} & e_{12} & e_{13} \\ e_{20} & e_{21} & e_{22} & e_{23} \end{bmatrix} \tag{2.1}$$

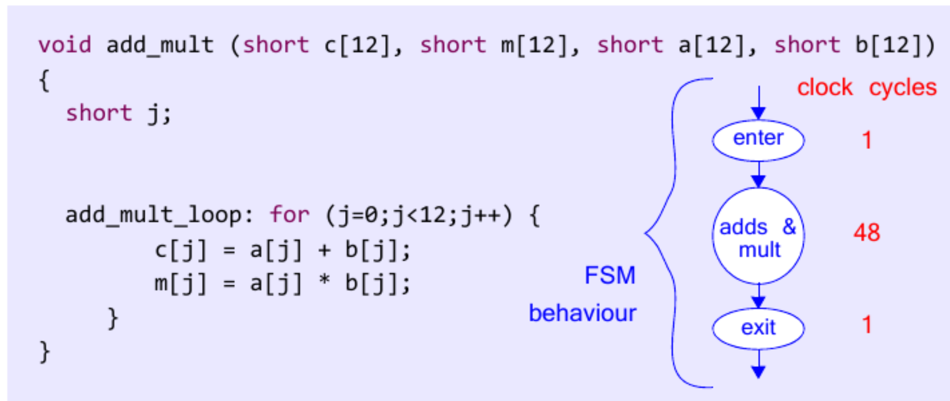

Figure 2.3: Merged addition and multiplication loops

```
void add_matrix (short f[3][4], short c[3][4], short d[3][4])
{
  short j,k;                           // loop variable

  row_loop: for (j=0;j<3;j++) {        // iterate through rows
       column_loop : for (k=0;k<4;k++) {  // iterate through columns
          f[j][k] = c[j][k] + d[j][k];    // addition operation
      }
  }
}
```

Figure 2.4: Example code for nested loops adding the elements of two dimensional arrays

Now, in order to add the arrays, we must iterate through the rows, and for each row, iterate through the columns, adding together the two values for each array element. Coding the matrix addition operation requires an outer and an inner loop to cycle through the rows and columns, respectively, as shown by the code example in 2.4. According to 2.1, there are 3 rows and 4 columns, and this determines the limits of the nested loops (note that indexing starts as zero, as per the usual convention). Extending this idea, it would be possible to work with three dimensional arrays, or even higher dimensions, by increasing the levels of nesting in the loop structure.

2.2.4 Optimization: Flattening Loops

In the case of nested loops, there is an option to perform 'flattening'. This means that the loop hierarchy is effectively removed during high-level synthesis while preserving the algorithm, i.e. all of the operations performed

by the loop(s). The advantage of flattening is similar to merging: the additional clock cycles associated with transitioning into or out of a loop are avoided, meaning that the overall duration of algorithm execution reduces, thus improving the achievable throughput.

In order to explain flattening in further detail, it is necessary to clarify the terms *loop* and *loop body*. By loop, we refer to an entire code structure of a set of statements repeated a defined number of times. The statements inside the loop, i.e. the statements that are repeated, are the loop body. For instance, column_loop is a loop, and the statements contained within column_loop correspond to the loop body.

When loops are nested, and again taking the example of a 2-level nested structure, the outer loop body contains another loop, i.e. the inner loop. The outer loop body (including the inner loop) is executed a certain number of times; for instance, row_loop has 3 repetitions in the example of Figure 2.4, and hence there are 3 executions of the inner loop, column_loop. Each execution of the inner loop involves repeating the inner loop body a specified number of times, as well: in our example, a statement to calculate the matrix element $f[j][k]$ is executed 4 times, where j is the row index and k is the column index.

The overhead of loop transitioning means that two additional clock cycles are required each time the inner loop is executed, i.e. one to enter the inner loop, and one to exit from it.

To clarify this point, a diagram depicting the control flow for our matrix addition example, and associated clock cycles, is provided in 2.5. This represents the original loop structure. The process of flattening 'unrolls' the inner loop, and as a consequence, reduces the number of clock cycles associated with transitioning into and out of loops; specifically, the 'enter_inner' and 'exit_inner' states in 2.5 are removed. These would have been repeated 3 times, hence 6 clock cycles are saved in total in this case.

In our simple 3×4 matrix addition example, the saving equates to 6 clock cycles, but in other examples, this could be considerably higher (in particular where the outer loop iterates many times, or there are several layers of nesting), and thus there is a clear motivation to flatten loops. Similar to merging of loops, flattening can be achieved by a directive and does not involve explicit unrolling of the loop by manually changing the code. However, depending on its initial form, some manual restructuring may also be required in order to achieve a loop structure optimal for flattening.

2.2.5 Pipelining Loops

The direct interpretation of a loop written in C code is that executions of the loop occur consecutively, i.e. each loop iteration cannot begin until the previous one has completed. In hardware terms, this translates to a single set of hardware (as inferred from the loop body) that is capable of executing

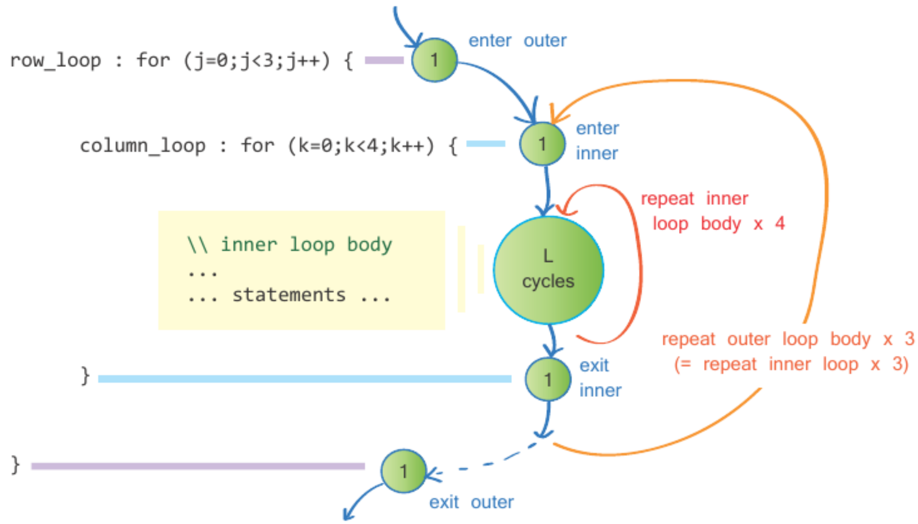

Figure 2.5: Control flow through the matrix addition, without flattening (clock cycles in circles)

the operations of only one loop iteration at any particular instant, and which is shared over time according to the number of loop iterations. Throughput is limited when a set of operations are grouped together into a processing stage. In the case of loops, the loop body (i.e. the set of repeated operations) forms such a stage, and without pipelining, this would result in all stages operating in a sequential manner, and within them, all operations executing in a sequential manner. In effect, all operations from all iterations of the loop body would occur one after the other. The total number of clock cycles to complete the execution of a loop, N loop, would therefore be:

$$N_{loop} = (J \times N_{body}) + N_{control} \tag{2.2}$$

where J is the number of loop iterations, N+body is the number of clock cycles to execute all operations in the loop body, and N control represents the overhead of transitioning into and out of the loop.

The insertion of pipelining into the loop means that registers separate the implemented operators within the loop body. Given that the loop body is repeated several times, this carries the implication that operations within loop iteration $j + 1$ can commence before those of the loop iteration j have completed. In fact, at any instant, operations corresponding to several different iterations of the loop may be active.

As a consequence of pipelining the loop, the hardware required to implement the loop body is more fully utilized, and loop performance is improved in terms of both throughput and latency. The effect of adding a pipelining directive can, therefore, be considerable, especially where there are multiple

operations within the loop body, and many iterations of the loop are performed. When working with nested loops, it is useful to consider at which level of hierarchy pipelining should be applied. Pipelining at a certain level of the hierarchy will cause all levels below (i.e. all nested loops) to be unrolled, which may produce a more costly implementation than intended. Therefore, a good balance between performance and resource utilization may be achieved by pipelining only at the level of the innermost loop (for instance, column_loop in Figure 2.5).

For more information about theoretical and mathematical fundamentals of loop transformations, based on the polyhedral model, refer to Appendix B.

2.3 External Tools and Frameworks

2.3.1 Profiling Tools

As indicated in chapter 4, to find the critical section of the code which is considered as computational or memory intensive should be identified before applying any optimizations. it is crucial to tune, transform and modify the section of the code which impacts critically on computations, CPU interrupts, and memory accesses.

Throughout this research different profiling tools got tested to obtain required performance metrics. Amongst them three candidates were *Perf*, *GProf* and *Valgrind*. As *Perf* and *GProf* were unable to measure performance metrics for small programs and also due to compatibility problems, we selected **Valgrind** as the main profiling tool in the framework. We also wrote three scripts to automate profiling via different tools. To pinpoint the critical section of the code, three execution features are taken into account, the *execution time* of a particular function,the *number of the times a function is called* and ($Execution_time \times \#function_call$). Alongside these performance measurements, we also investigate call graphs, generated by *Callgrind* to decide about the hot function of the code.

Framework's high-level Profiler

Kcachegrind is the highest level profiler integrated into the framework which is actually a profile data visualization. *Callgrind* uses runtime instrumentation via the Valgrind framework for its cache simulation and call-graph generation. This way even shared libraries and dynamically opened plugins can be profiled. The data files generated by *Callgrind* can be loaded into *Kcachegrind* for browsing the performance results. All experiments and benchmarks in this thesis are profiled by *Kcachegrind*. For each code the above-mentioned performance measurements were captured, meaning that we had at most three candidates function to apply software optimiza-

tions, code transformation and hardware accelerator selection for HW/SW designs(Co-designs).

> *"In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization"*

2.3.2 Orio

Orio [10] is an empirical performance-tuning system that takes annotated C source code as input, generates many optimized code variants of the annotated code, and empirically evaluates the performance of the generated codes, ultimately selecting the best-performing version to use for production runs. Orio also supports automated validation by comparing the numerical results of the multiple transformed versions.

The Orio annotation approach differs from existing annotation- and compiler-based systems in the following significant ways. First, through designing an extensible annotation parsing architecture, it is not committing to a single general-purpose language. Thus, Orio can define annotation grammars that restrict the original syntax, enabling more effective performance transformations (e.g., disallowing pointer arithmetic in a C-like annotation language); furthermore, it enables the definition of new high-level languages that retain domain-specific information normally lost in low-level C or Fortran implementations. This feature, in turn, expands the range of possible performance-improving transformations.

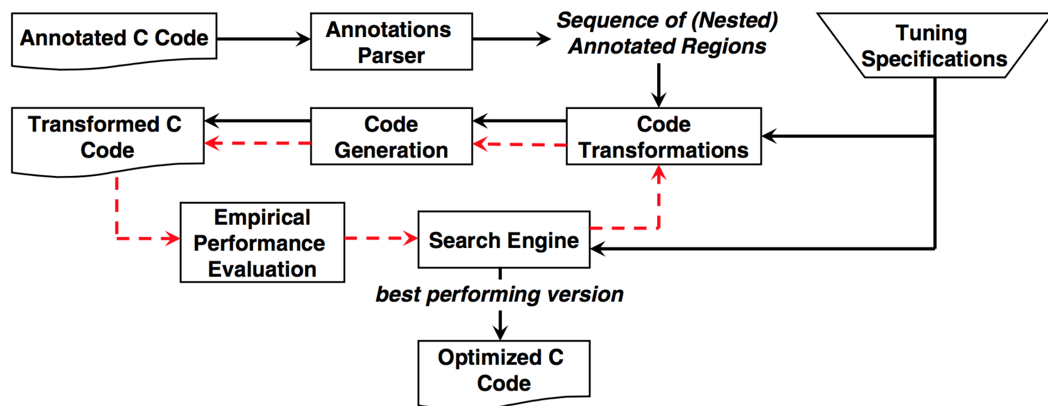

Figure 2.6: Overview of Orio's code generation and empirical tuning process.

Second, Orio was conceived and designed with the following requirements in mind: portability (which precludes extensive dependencies on external packages), extensibility (new functionality must require little or no change

to the existing Orio implementation, and interfaces that enable integration with other source transformation tools must be defined), and automation (ultimately Orio should provide tools that manage all the steps of the performance tuning process, automating each step as much as possible). Third, Orio is usable in real scientific applications without requiring reimplementation. This ensures that the significant investment in the development of complex scientific codes is leveraged to the greatest extent possible.

Figure 2.6 shows a high-level graphical depiction of the code generation and tuning process implemented in Orio. Orio can be used to improve performance by source-to-source transformations such as loop unrolling, loop tiling, and loop permutation. The input to Orio is C code containing structured comments that include a simplified expression of the computation, as well as various performance-tuning directives. Orio scans the marked-up input code and extracts all annotation regions. Each annotation region is then processed by transformation modules. The code generator produces the final C code with various optimizations that correspond to the specified annotations. Unlike compiler approaches, we do not implement a full-blown C compiler; rather, we use a pre-compiler that parses only the language-independent annotations.

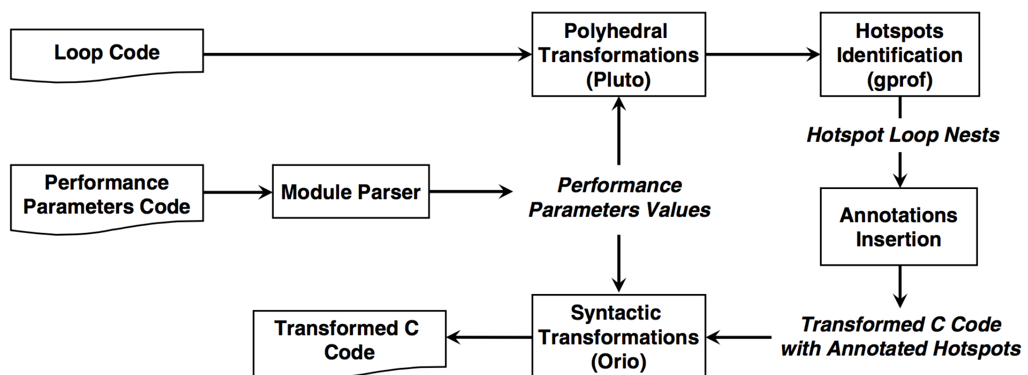

Figure 2.7: Integration of Pluto and Orio

Orio can also be used as an automatic performance-tuning tool. The code transformation modules and code generator produce an optimized code version for each distinct combination of performance parameter values. Then each optimized code version is executed and its performance evaluated. After iteratively evaluating all code variants, the best-performing code is picked as the final output of Orio. Because the search space of all possible optimized code versions can be huge, a brute-force search strategy is not always feasible. Hence, Orio provides various search heuristics for reducing the size of the search space and thus the empirical testing time.

A number of source-to-source transformation tools for performance optimization exist. Using these tools to achieve (near) optimal performance on different architectures, however, is still a nontrivial task that requires significant architectural and compiler expertise. Orio has been extended with an external transformation program, called Pluto [11]. This integration also demonstrates the easy extensibility of Orio and the ability to leverage other source transformation approaches.

Pluto is a source-to-source automatic transformation tool aimed at optimizing a sequence of nested loops for data locality and coarse-grained parallelism simultaneously. Pluto employs a polyhedral model of arbitrary loop nests, where the dynamic instance (iteration) of each statement is viewed as an integer point in a well-defined space called the statement's polyhedron.

This statement representation and a precise characterization of data dependencies enable Pluto to construct mathematically correct complex loop transformations. Pluto's polyhedral-based transformations result in improved cache locality and loops parallelized for multi-core architecture. Figure 2.7 outlines the overall structure of the Pluto-Orio integrated system, which is implemented as a new optimization module in Orio.

2.3.3 OpenTuner

OpenTuner [12] is a framework for building domain-specific program autotuners. OpenTuner features an extensible configuration and technique representation able to support complex and user-defined data types and custom search heuristics. It contains a library of predefined data types and search techniques to make it easy to setup a new project. Thus, OpenTuner solves the custom configuration problem by providing not only a library of data types that will be sufficient for most projects, but also extensible data types that can be used to support more complex domain specific representations when needed. A core concept in OpenTuner is the use of ensembles of search techniques. Figure 2.8 illustrates the structure of OpenTuner.

Many search techniques (both built-in and user-defined) are run at the same time, each testing candidate configurations. Techniques which perform well by finding better configurations are allocated larger budgets of tests to run, while techniques which perform poorly are allocated fewer tests or disabled entirely. Techniques are able to share results using a common results database to constructively help each other in finding an optimal solution.

Algorithms add results from other techniques as new members of their population. To allocate tests between techniques we use an optimal solution to the multi-armed bandit problem using the area under the curve credit assignment. Ensembles of techniques solve the large and complex search space problem by providing both robust solutions to many types of large search spaces and a way to seamlessly incorporate domain-specific search

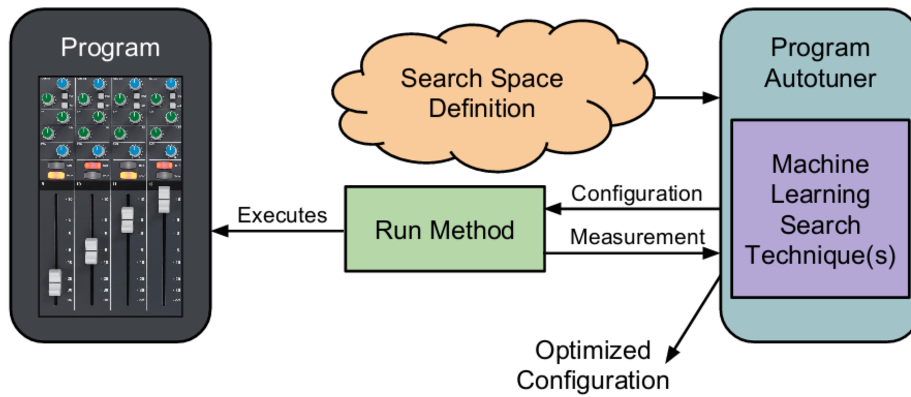

Figure 2.8: OpenTuner structure

techniques.

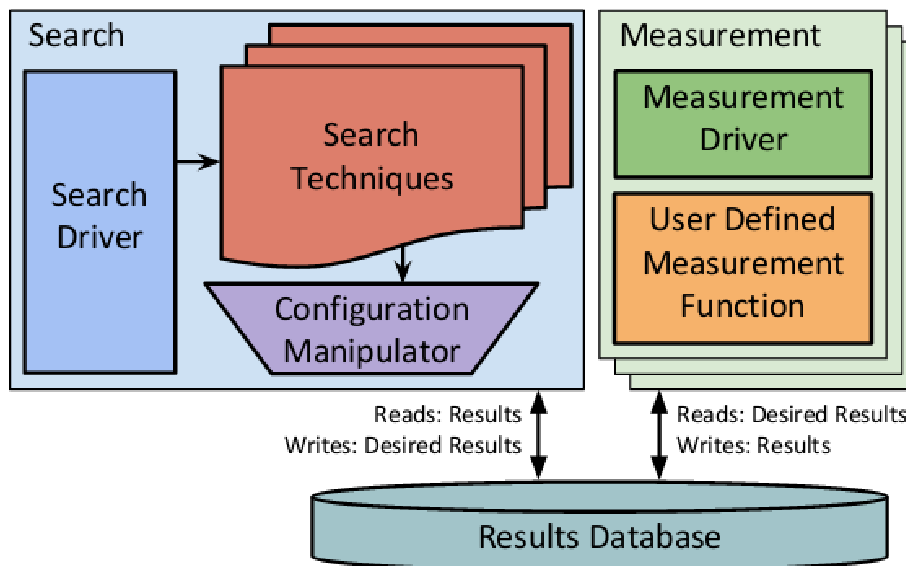

Figure 2.9: OpenTuner components

Figure 2.9 provides an overview of the major components in OpenTuner. The search process includes techniques, which use the user-defined configuration manipulator in order to read and write configurations. The measurement processes evaluate candidate configurations using a user-defined measurement function. These two components communicate exclusively through a results database used to record all results collected during the tuning process, as well as providing the ability to perform multiple measurements in parallel.

OpenTuner and HLS

As already mentioned in methodology description, the two-step code modifications impact positively on different design methodologies. Application auto-tuning increase the multiplication block size which makes the corresponding synthesized hardware make use of LEs more efficiently. Depending on the type of FPGA this size approaches to an optimum value which makes the multiplication outperform both in HW and SW.

It is important to understand the differences between Orio and OpenTuner. Orio is a source to source compiler which also reports some optimum compiler flags. But OpenTuner use application parameter tuning, it can tune some algorithmic parameters embedded inside the code and it is independent of the source to source compilation flow.

2.3.4 LegUp

LegUp [13] is an open-source high-level synthesis (HLS) tool that accepts a C program as input and automatically synthesizes it into a hybrid system. The hybrid system comprises an embedded processor and custom accelerators that realize user-designated compute-intensive parts of the program with improved throughput and energy efficiency.Embedded system designers can achieve energy and performance benefits by using dedicated hardware accelerators.

However, implementing custom hardware accelerators for an application can be difficult and time intensive. LegUp is an open-source high-level synthesis framework that simplifies the hardware accelerator design process. With LegUp, a designer can start from an embedded application running on a processor and incrementally migrate portions of the program to hardware accelerators implemented on an FPGA. The final application then executes on an automatically-generated software/hardware co-processor system.

LegUp includes a soft processor because not all program segments are appropriate for hardware implementation. Inherently sequential computations are well-suited for software (e.g. traversing a linked list); whereas, other computations are ideally suited for hardware (e.g. addition of integer arrays). Incorporating a processor also offers the advantage of increased high-level language coverage.

Program segments that use restricted C language constructs can execute on the processor (e.g. recursion). LegUp is written in modular C++ to permit easy experimentation with new HLS algorithms. It leverages the state-of-the-art LLVM (low-level virtual machine) compiler framework for high-level language parsing and its standard compiler optimizations [14].

LegUp Design flow

The LegUp design flow comprises first compiling and running a program on a standard processor, profiling its execution, selecting program segments to target to hardware, and then re-compiling the program to a hybrid hardware/software system.

Figure 2.10 illustrates the detailed flow. Referring to the labels in the figure, at step 1, the user compiles a standard C program to a binary executable using the LLVM compiler. At 2, the executable is run on an FPGA-based MIPS processor. It makes use of Tiger MIPS processor from the University of Cambridge [University of Cambridge 2010], but it is also possible to migrate to other processors such as ARM core.

The MIPS processor has been augmented with extra circuitry to profile its own execution. Using its profiling ability, the processor is able to identify sections of program code that would benefit from hardware implementation, improving program throughput and power. Specifically, the profiling results drive the selection of program code segments to be re-targeted to custom hardware from the C source. Having chosen program segments to target custom hardware, at step 3 LegUp is invoked to compile these segments to synthesizable Verilog RTL.

Presently, LegUp HLS operates at the function level: entire functions are synthesized to hardware from the C source. Moreover, if a hardware function calls other functions, such called functions are also synthesized to hardware. In other words, we do not allow a hardware-accelerated function to call a software function. The RTL produced by LegUp is synthesized to an FPGA implementation using standard commercial tools at step 4. As illustrated in the figure, LegUp's hardware synthesis and software compilation are part of the same LLVM-based compiler framework.

In step 5, the C source is modified such that the functions implemented as hardware accelerators are replaced by wrapper functions that call the accelerators (instead of doing computations in software). This new modified source is compiled to a MIPS binary executable. Finally, in step 6 the hybrid processor/accelerator system executes on the FPGA.

Figure 2.11 elaborates on the target system architecture. The processor connects to one or more custom hardware accelerators through a standard on-chip interface. As our initial hardware platform is based on Cyclone II FPGA, the Altera Avalon interface for processor/accelerator communication is used. A shared memory architecture is used, with the processor and accelerators sharing an on FPGA data cache and off-chip main memory. The on-chip cache memory is implemented using block RAMs within the FPGA fabric (M4K blocks on Cyclone II). Access to memory is handled by a memory controller. The architecture in Figure 2.11 allows processor/accelerator communication across the Avalon interface or through memory.

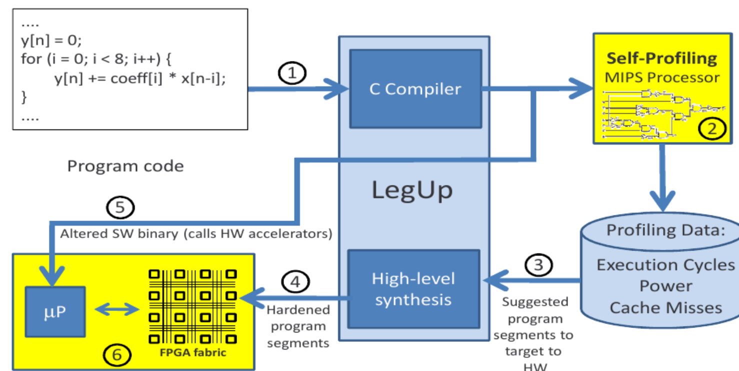

Figure 2.10: LegUp Design Flow

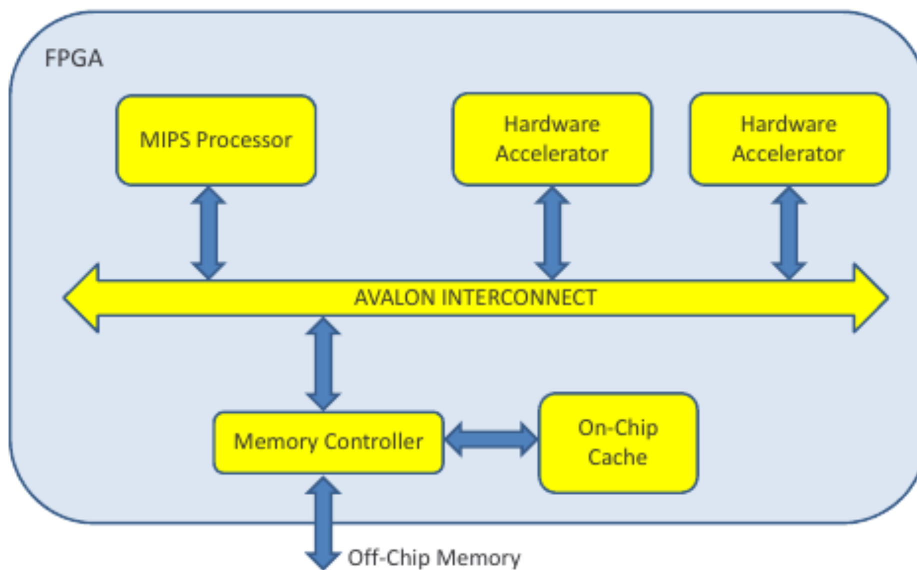

Figure 2.11: Target System Architecture

LegUp High-Level Hardware Synthesis

High-level synthesis has traditionally been divided into three steps:

- allocation
- scheduling
- binding

Allocation determines the amount of hardware available for use (e.g., the number of adder functional units), and also manages other hardware constraints (e.g., speed, area, and power). Scheduling assigns each operation in the program being synthesized to a particular clock cycle (state) and generates a finite state machine.

Binding assigns an application operation to specific hardware units. The decisions made by binding may imply sharing functional units between operations, and share registers/memories between variables. All the abovementioned stages are integrated into LegUp procedures.

The LegUp open-source HLS tool is implemented within the LLVM compiler framework, which is used in both industry and academia. LLVM's front-end, clang, parses the input C source and translates it into LLVM's intermediate representation (IR). The IR is essentially machine-independent assembly code in static-single assignment (SSA) form, composed of simple computational instructions (e.g., add, shift, multiply) and control-flow instructions (e.g., branch). LLVM's opt tool performs a sequence of compiler optimization passes on the program's IR, each such pass directly manipulates the IR, accepting an IR as input and producing a new/optimized IR as output.

A high-level diagram of the LegUp Hardware synthesis flow is shown in Figure 2.12. The LegUp HLS tool is implemented as back-end passes of LLVM that are invoked after the standard compiler passes. LegUp accepts a program's optimized IR as input and goes through the four stages shown in Figure 2.12 (1) Allocation, (2) Scheduling, (3) Binding, and (4) RTL generation) to produce a circuit in the form of synthesizable Verilog HDL code.

The allocation stage allows the user to provide constraints to the HLS algorithms, as well as data that characterizes the target hardware. Examples of constraints are limits on the number of hardware units of a given type that may be used, the target circuit critical path delay, and directives pertaining to loop pipelining and resource sharing. The hardware characterization data specifies the speed (critical path delay) and area estimates (number of FPGA logic elements) for each hardware operator (e.g., adder, multiplier) for each supported bandwidth (typically 8, 16, 32, and 64 bit). The characterization data is collected only once for each FPGA target family using automated

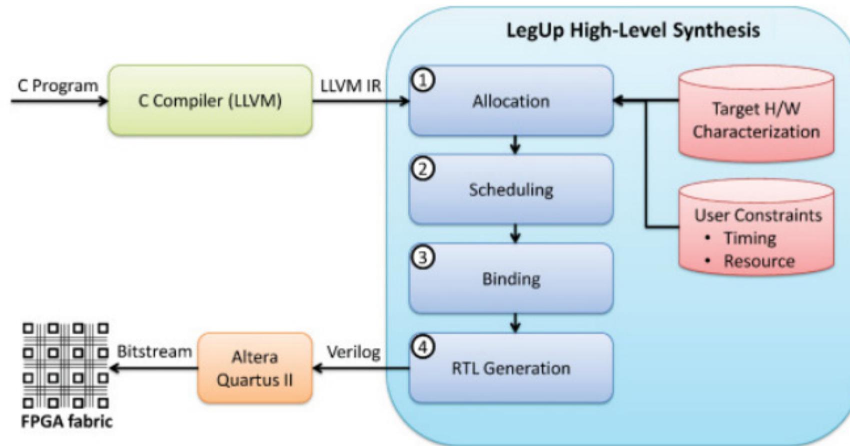

Figure 2.12: Design flow with legup

scripts. The scripts synthesize each operation in isolation for the target FPGA family to determine their speed and area.

At the scheduling stage, each operation in the program (in the LLVM IR) is assigned to a particular clock cycle (state) and an FSM is generated. The LegUp scheduler, based on the SDC formulation, operates at the basic block level, exploiting the available parallelism between instructions in a basic block. A basic block is a sequence of instructions that has a single entry and exit point.

The scheduler performs operation chaining between dependent combinational operations when the combined path delay does not exceed the clock period constraint-chaining refers to the scheduling of dependent operations into a single clock cycle. Chaining can reduce hardware latency (number of cycles for execution) and save registers without impacting the final clock period.

The binding stage assigns operations in the program to specific hardware units. When multiple operators are assigned to the same hardware unit, multiplexers are generated to facilitate the sharing. Multiplexers require a significant amount of area when implemented in an FPGA logic fabric.

Consequently, there is no advantage to sharing all but the largest functional units, namely multipliers, dividers, and recurring patterns of smaller operators. Multiplexers also contribute to circuit delay, and thus they are used judiciously by the HLS algorithms. LegUp also recognizes cases where there are shared inputs between operations, which allows hardware units to be shared without creating multiplexers. Last, if two operations with non-overlapping lifetime intervals are bound to the same functional unit, then a

single output register is used for both operations. This optimization saves a register as well as a multiplexer.

The RTL generation stage produces synthesizable Verilog HDL register transfer level code. One Verilog module is generated for each function in the C source program. Results show that LegUp produces solutions of comparable quality to a commercial

2.4 Target Architectures

MIPS architecture is the main target that we followed due to the compatibility of the Orio and LegUp with the architecture and also, using the common reference architecture in computer science.

2.4.1 MIPS

As the MIPS processor is part of the framework to implement software execution and also code tuning is applied based on this architecture, we roughly illustrate the processor organization. Figure 2.13 shows the datapath fellow with the control unit of MIPS architecture. The key concepts of the original MIPS architecture are:

- Five-stage execution pipeline: fetch, decode, execute, memory-access, write-result
- Regular instruction set, all instructions are 32-bit
- Three-operand arithmetical and logical instructions
- 32 general-purpose registers of 32-bits each
- No status register or instruction side-effects
- No complex instructions (like stack management, string operations, etc.)
- Optional coprocessors for system management and floating-point
- Only the load and store instruction access memory
- Flat address space of 4 GBytes of main memory ($2^{3}2$ bytes)
- Memory-management unit (MMU) maps virtual to actual physical addresses
- Optimizing C compiler replaces hand-written assembly code
- Hardware structure does not check dependencies - not "foolproof"

- But software toolchain knows about hardware and generates correct code

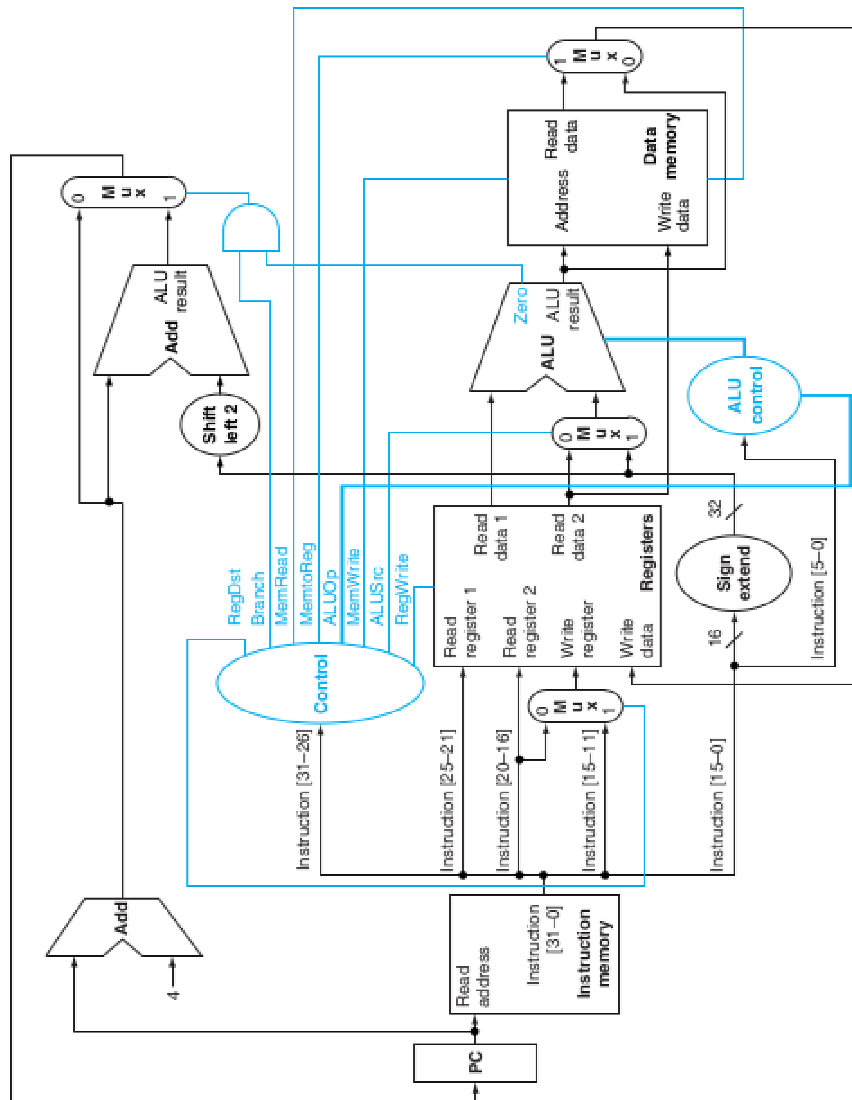

Figure 2.13: MIPS datapath with control unit.

One of the key features of the MIPS architecture is the regular register set. It consists of the 32-bit wide program counter (PC), and a bank of 32 general-purpose registers called r0..r31, each of which is 32-bit wide. All general-purpose registers can be used as the target registers and data sources for all logical, arithmetical, memory access, and control-flow instructions. Only r0 is special because it is internally hardwired to zero. Reading r0

always returns the value 0x00000000, and a value written to r0 is ignored and lost.

Note that the MIPS architecture has no separate status register. Instead, the conditional jump instructions test the contents of the general-purpose registers, and error conditions are handled by the interrupt/trap mechanism. Two separate 32-bit registers called HI and LO are provided with the integer multiplication and division instructions.

2.4.2 Emulation Environments

The proposed methodology needs a MIPS/ARM processor to optimize architectural dependent parameters and to find the best-transformed version of the code which outperforms among others. To find a versatile and applicable clue, we considered several approaches including using an actual MIPS/ARM processor, Open Virtual Platform(OVP), and QEMU. To tackle compatibility issues with other elements of the framework such as Orio and OpenTuner, and also the potential to access the emulator remotely using a virtual private network and a secure shell, QEMU was chosen to emulate our MIPS environment. In case of ARM platform, we used our DE1-SoC development board which provides an embedded arm processor (Hard-core) on a Cyclon V FPGA.

QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.

When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, 64-bit POWER, S390, 32-bit and 64-bit ARM, and MIPS guests.

2.4.3 Development Board

At the end of compilation and synthesis process, binary and SRAM object files are downloaded on an FPGA. we used DE1-SoC evaluation board to implement the designs. Figure 2.14 shows the structure of the board. Figure 2.15 shows the architecture of Cyclon V which consist of the logic part, namely FPGA, and a hard Processing System called HPS which is actually an ARM core.

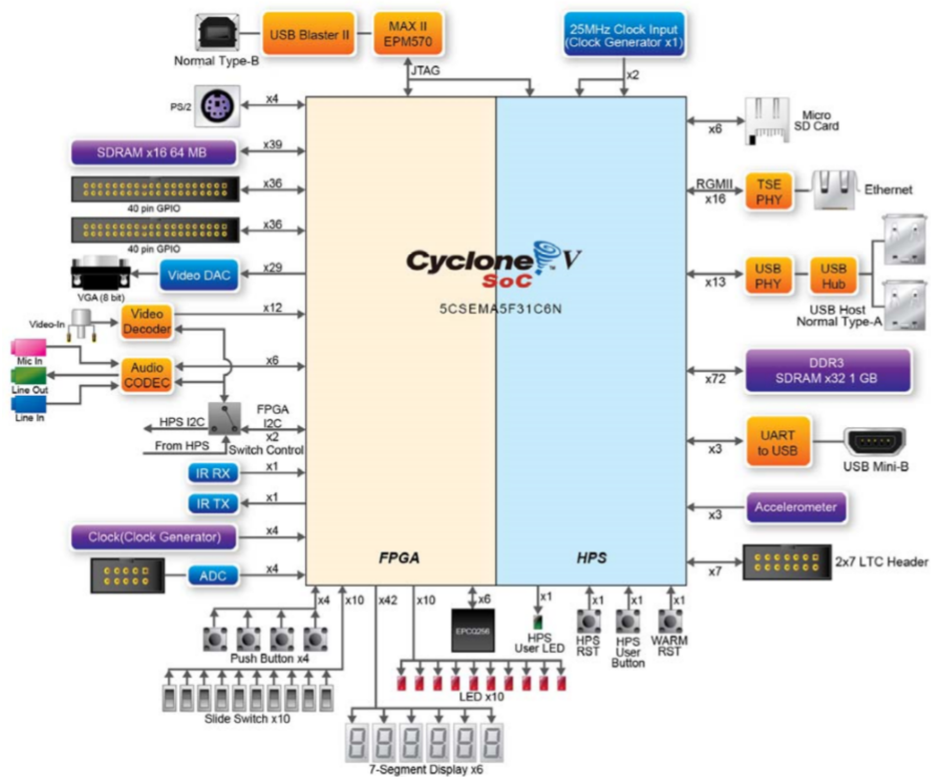

Figure 2.14: Terasic , DE1 SoC

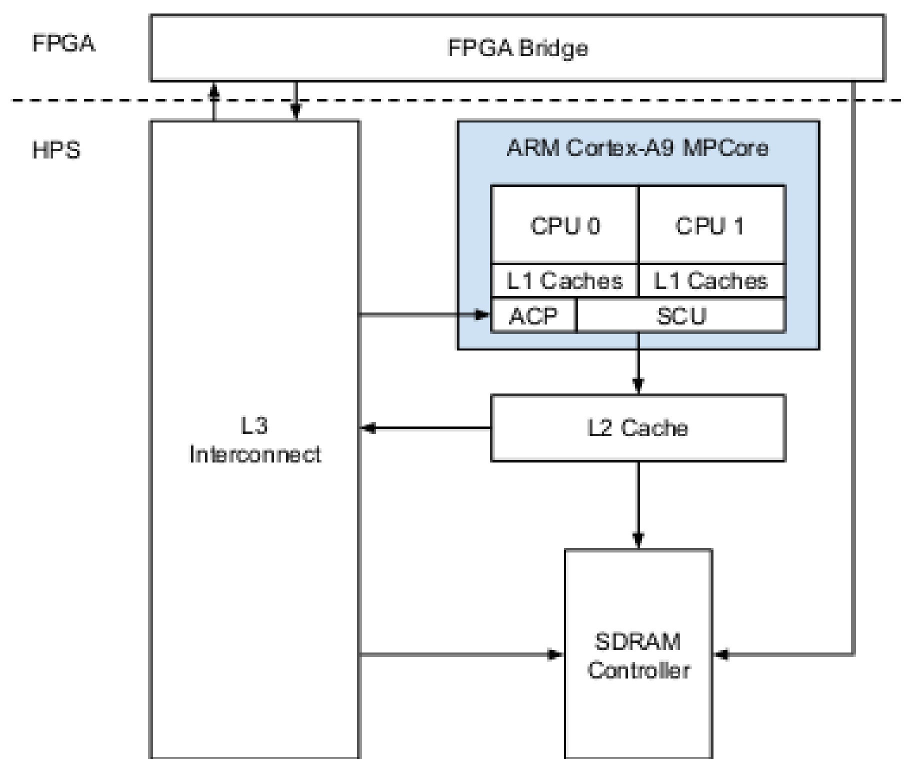

Figure 2.15: Altera's Cyclon V SoC Architecture

Chapter 3

State of the Art

Automatic Performance Tuning or "Auto-tuning", is an empirical, feedback-driven performance optimization technique designed to maximize performance across a wide variety of architectures without sacrificing portability or productivity. Over the years, autotuning has expanded from simple loop tiling and unroll-and-jam to encompass transformations to data structures, parallelization, and algorithmic parameters. Automated tuning or autotuning has become a commonly accepted technique used to find the best implementation for a given kernel on a given single-core machine [15, 16, 17, 18, 19]. Figure 3.1 compares the traditional and autotuning approaches to programming. 3.1(a) shows the common Alberto Sangiovanni-Vincentelli (ASV) triangle [20]. A programmer starts with a high-level operation or kernel he wishes to implement. There is a large design space of possible implementations that all deliver the same functionality.

However, he prunes them to a single C program representation. In doing so, all high-level knowledge is withheld from the compiler which in turn takes the C representation and explores a variety of safe transformations given the little knowledge available to it. The result is a single binary representation. Figure 3.1(b) presents the autotuning approach. The programmer implements an autotuner that rather than generating a single C-level representation, generates hundreds or thousands. The hope is that in generating these variants some high-level knowledge is retained when the set is examined collectively. The compiler then individually optimizes these C kernels producing hundreds or machine language representations.

The autotuner then explores these binaries in the context of the actual data set and machine. There are three major concepts with respect to autotuning: the optimization space, code generation, and exploration. First, a large optimization space is enumerated. Then, a code generator produces C code for those optimized kernels. Finally, the autotuner proper explores the optimization space by benchmarking some or all of the generated kernels searching for the best performing implementation. The resultant configura-

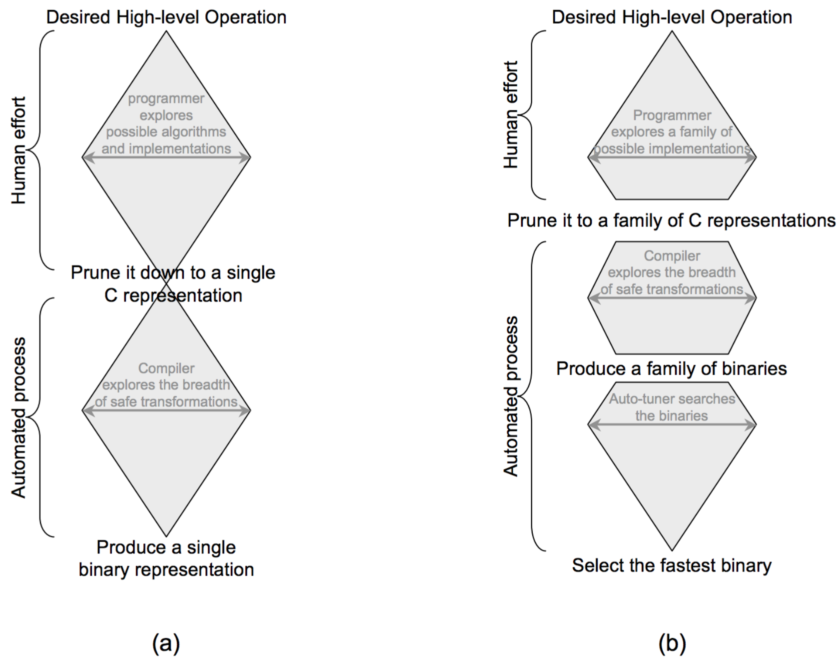

Figure 3.1: ASV triangles for the conventional and auto-tuned approaches to programming.

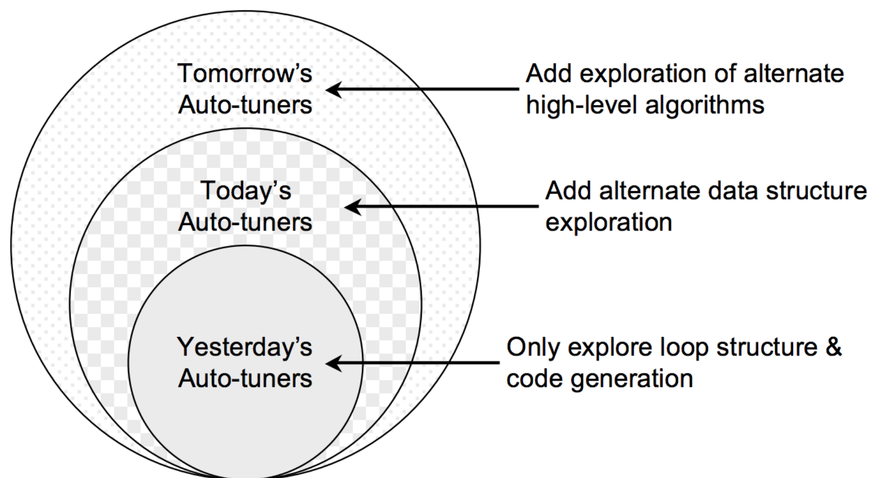

Figure 3.2: High-level discretization of the autotuning optimization space.

tion is an autotuned kernel.

Figure 3.2 on the next page shows the high-level discretization of the optimization space. The simplest autotuners only explore low-level optimizations like unrolling, reordering, restructuring loops, eliminating branches, explicit SIMDization or using cache bypass instructions. These are all optimizations compilers claim to be capable of performing, but often cannot due to the lack of information conveyed in a C program. More advanced autotuners will also explore different data types, data layouts, or data structures. Compilers have no hope of performing these optimizations.

Finally, the most advanced autotuners also explore different algorithms that produce the same solution for the high-level problem being solved. For example, an auto-tuner might implement a Barnes-Hut-like particle-tree method instead of a full N2 particle interaction. The second aspect of autotuning is code generation. The simplest strategy is for an expert to write a Perl or similar script to generate all possible kernels as enumerated by the optimization space. A more advanced code generator could inspect C or FORTRAN code, and in the context of a specific motif generate all valid optimizations through a regimented set of transformations [21].

Another aspect of autotuning is the exploration of the optimization space. There are several strategies designed to cope with the ever-increasing search space such as ReSPIR [22], DeSpErate++ [23] and OSCAR [24]. The most basic approach is an exhaustive search of all parameters for all optimizations and for each optimization, an appropriate parameter is selected.

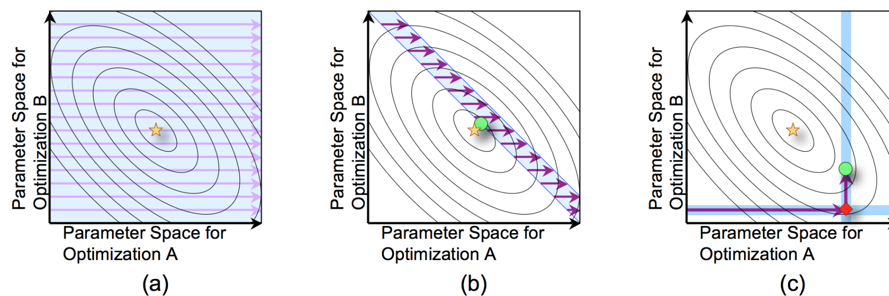

Figure 3.3: Visualization of three different strategies for exploring the optimization space: (a) Exhaustive search, (b) Heuristically-pruned search, and (c) hill-climbing. Note, curves denote combinations of constant performance. The gold star represents the best possible performance.

Finally, in a hill-climbing approach, optimizations are examined in isolation. An optimization is selected. Performance is benchmarked for all parameters for that optimization and the best-known parameter for all other optimizations. The best configuration for that optimization is determined. The process continues until all optimizations have been explored once.

Unfortunately, this approach may still require thousands of trials. Figure 3.3 visualizes the three different strategies for exploring the optimization space. For clarity, we have restricted the optimizations space to two optimizations, each with their own independent range of parameters. In Figure 3.3(a), an exhaustive approach searches every combination of every possible parameter for all optimizations. Clearly, this approach is a very time consuming, but is guaranteed to find the best possible performance for the implemented optimizations.

Figure 3.3(b) heuristically-prunes the search space, and exhaustively searches the resultant region. Clearly, this will reduce the tuning time, but might not find the best performance as evidenced by the fact that the resultant performance (green circle) is close but not equal to the best performance (gold star).

Figure 3.3 uses a one-pass hill-climbing approach. Starting from the origin, the parameter space for optimization A is explored. The local maximum performance is found (red diamond). Then, using the best-known parameter for optimization A, the parameter space for optimization B is explored. The result (green circle) is far from the best performance (gold star), but the time required for tuning is very low. Perhaps the most important aspect of an autotuner's exploration of the parameter space is that it is often done in conjunction with real data sets. That is, one provides either a training set or the real data to ensure the resultant optimization configuration will be ideal for real problems.

	Optimizations Explored				**Exploration of the Optimization Space**			
					Data-oblivious		**Data-aware**	
	Low Level	**Data Structure**	**Other Algorithms**	**multi-core**	**Heuristics**	**Search**	**Heuristics**	**Search**
Traditional Compilers	✓			†	✓			
"auto-tuning" Compilers	✓			†	✓	✓		
Yesterday's Auto-tuners	✓							✓
Today's Auto-tuners	✓	✓					✓	✓
Tomorrow's Auto-tuners	✓	✓	✓	✓			✓	✓

Figure 3.4: Comparison of traditional compiler and autotuning capabilities. Low-level optimizations include loop transformations and code generation.†only via OpenMP pragmas.

Figure 3.4 provides a comparison of the capabilities of compilers and autotuners. Traditional compilers can only perform the most basic optimization and choose the parameters in a data-oblivious fashion. Nevertheless, these compilers are limited by the input C program and are oblivious to the actual data set. As a result, even yesterday's traditional autotuner is more capable. Today, some autotuners explore alternate data structures and use heuristics to make exploration of the search space tractable. Tomorrow's au-

totuners will likely also explore algorithmic changes. In doing so, they may trade vastly improved computational complexity for slightly worse efficiency. As a result, the time to the solution will be significantly improved.
Broadly speaking, autotuning is premised on three fundamental research topics:

- creating novel application- and architecture-relevant optimizations and their associated parameter space
- creating novel application- and architecture-relevant optimizations and their associated parameter space
- automating the generation of code for this optimization space efficiently searching this optimization space (a mathematical optimization problem)

3.1 Related Works

Table 3.1: Summary of selected related projects using autotuning

Package	Domain	Search Method
Active Harmony [25]	Runtime System	Nelder-Mead
ATLAS [26]	Dense Linear Algebra	Exhaustive
FFTW [27]	Fast Fourier Transform	Exhaustive/Dynamic Prog.
Insieme [28]	Compiler	Differential Evolution
OSKI [29]	Sparse Linear Algebra	Exhaustive+Heuristic
PATUS [30]	Stencil Computations	Nelder-Mead or Evolutionary
PetaBricks [31]	Programming Language	Bottom-up Evolutionary
Sepya [32]	Stencil Computations	Random-Restart Gradient Ascent
SPIRAL [33]	DSP Algorithms	Pareto Active Learning

A number of offline empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains; selected projects and techniques used are summarized in Table 3.1. ATLAS [26] utilizes empirical autotuning to produce an optimized matrix multiply routine. FFTW [27] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPIRAL [33] for digital signal processing PATUS [30] and Sepya [32] for stencil computations, and OSKI [29] for sparse matrix kernels and recent application parameter autotuner OpenTuner [12] investigating the best parameter values for the desired parameters. The area of iterative compilation contains many projects that use different machine learning techniques to optimize lower level compiler optimizations [34, 35, 36, 37, 38, 39, 40].

These projects change both the order that compiler passes are applied and the types of passes that are applied. In the dynamic autotuning space, there have been a number of systems developed [41, 42, 43, 44, 45, 46] that focus on creating applications that can monitor and automatically tune themselves to optimize a particular objective. Many of these systems employ control systems based autotuner that operates on a linear model of the application being tuned. For example, PowerDial [41] converts static configuration parameters that already exist in a program into dynamic knobs that can be tuned at runtime, with the goal of trading QoS guarantees for meeting performance and power usage goals.

The system uses an offline learning stage to construct a linear model of the choice configuration space which can be subsequently tuned using a linear control system. The system employs the heartbeat framework [47] to provide feedback to the control system. A similar technique is employed in [42], where a simpler heuristic-based controller dynamically adjusts the degree of loop perforation performed on a target application to trade QoS for performance.

These automatic or hand-tuned approaches can deliver performance that can be five times as fast as that produced by many optimizing compilers. The library approach, however, is limited by the fact that optimizations are highly problem- and machine-dependent. Furthermore, at this time, the functionality of the automated tuning systems is quite limited.

General-purpose tools for optimizing loop performance are also available. Loop Tool supports annotation-based loop fusion, unroll/jamming, skewing, and tiling. The Matrix Template Library uses template meta-programs to tile at both the register and cache levels. A new tool, POET, also supports a number of loop transformations.

Other research efforts whose goal, at least in part, is to enable optimizations of source code to be augmented with performance-related information include the X language (a macro C-like language for annotating C code), the Broadway compiler, and telescoping languages, and various meta-programming techniques.

Emerging annotation-based tools like Orio [10] are normally designed by compiler researchers and thus the interfaces are not necessarily based on concepts accessible to computational scientists. The complexity of existing annotation languages and lack of common syntax for transformations (e.g., loop unrolling) result in steep learning curves and the inability to take advantage of more than one approach at a time. Furthermore, at present, there is no good way for users to learn about the tools available and compare their capabilities and performance.

Other research efforts whose goal, at least in part, is to enable optimizations of source code to be augmented with performance-related information include the X language (a macro C-like language for annotating C code), the Broadway compiler, and telescoping languages.

Besides, improving the execution time and resource consumption through optimizing HLS SPIRIT [48], LegUp [13] using compiler optimizations [49] is broadly interesting.

Compilers in HLS

Modern HLS tools are implemented within software compiler frameworks. For example, Altera's OpenCL compiler for FPGAs[50], Xilinx's Vivado HLS tool[51], ROCCC HLS from the University of California at Riverside, and LegUp from the University of Toronto [13] are implemented within the LLVM framework [14]. Similarly, GAUT[52] from the Universite de Bretange Sud is implemented within GCC.

Chapter 4

Proposed Methodology

This chapter describes the components of our hybrid optimization framework. We provide an abstract view of the whole framework then tend into fine-grained details.

In general, the framework optimizes the performance of input code in different design methods which includes software, hardware, hybrid, and parallel. In case of software implementation, it acts as an *application/-code/compiler tuning* framework. In case of hardware, it acts as a *high performance HLS* framework. Also, it normally finds the best-transformed code for all types of design methods.

In our framework, we make use of two other autotuning frameworks: Orio [10] and OpenTuner [12]. They are extensible and portable frameworks for empirical performance tuning. Orio takes as input an annotated C code specifying potential code transformations (see Table I for the transformations used in our experiments), their possible parameter values, and a search strategy. The search algorithm in Orio generates multiple versions of the source code based on the different code transformations and runs the resulting configurations on a target machine in order to find the one with the best run time. We refer the reader to [10] for details about the annotation parsing and code-generation schemes in Orio. OpenTuner is a framework for building domain-specific program autotuners. It provides extensible infrastructure to support complex and user-defined data types and custom search heuristics. It runs a number of search techniques at the same time; those that perform well are allocated larger budgets systematically using optimal budget allocation methods. Unlike Orio, it does not provide automatic code transformation recipes; typically the user has to write the code-specific transformation modules. Nonetheless, it is well suited for tuning compiler and command line parameters for full applications.

4.1 Hybrid Optimization Framework details

The main input to the Hybrid Optimization Framework (HOF) is a C code, testing the correctness of the input through compiling by GCC is the start point of the chain. C is the popular programming language among HLS developers, also the most famous benchmarks in the field of computer architecture and HLS are written in C.Our framework contains two main parts, Code/Application Autotuning which is related to all the optimizations in the software section and HLS tuning which is related to the hardware aspects of the framework and subsequent different design methods. The final output of HOF is a binary file to program MIPS/ARM, or/and an SRAM object file to program FPGA. Beside necessary files for actual Board-level implementation, it is also possible to simulate the whole design using generated files. Moreover, some performance measurements are reported accordingly. Figure 4.1 illustrates the complete flowchart and details.

4.1.1 Tool chain and platforms

The framework is made from several stacked platforms. It is accessible through online through VPN. due to consistency and architectural dependent optimizations, there are levels of data transfer between different platforms. Figure 4.2 show the order the platforms are stacked, it also illustrates the location of each component of the framework. As the processors embedded inside the HOF are either MIPS or ARM, most of the code transformation and tuning processes are executed on MIPS/ARM platforms. These platforms run Debian as the os and reside on DE1-SoC board or emulated MIPS on QEMU. Following sections will provide further explanation of the tool interactions.

4.1.2 Code/Application Autotuning

In this section, we explain all the software section optimizations of the framework.

Profiler The profiler is playing a vital role in our framework and after testing several candidates, the Valgrind profiler has been chosen as the best appropriate profiler for our framework. At the first step, the original code will be evaluated by the Valgrind to investigate the hot function of the application to be considered as an assumption in the next steps. The hot function is important for three stages:

- First, we consider it when OpenTuner autotuner is tuning the parameter of the application in which we will search the best value for the parameters in hot function of the application instead of the whole application.

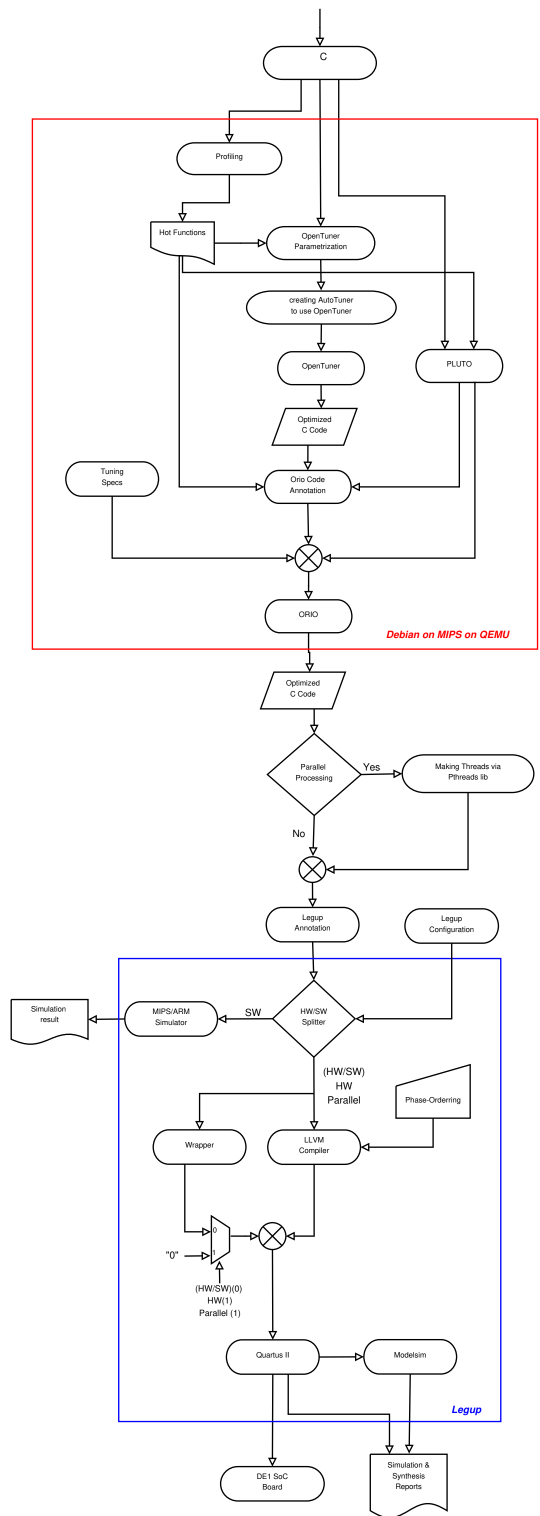

Figure 4.1: Methodology Diagram

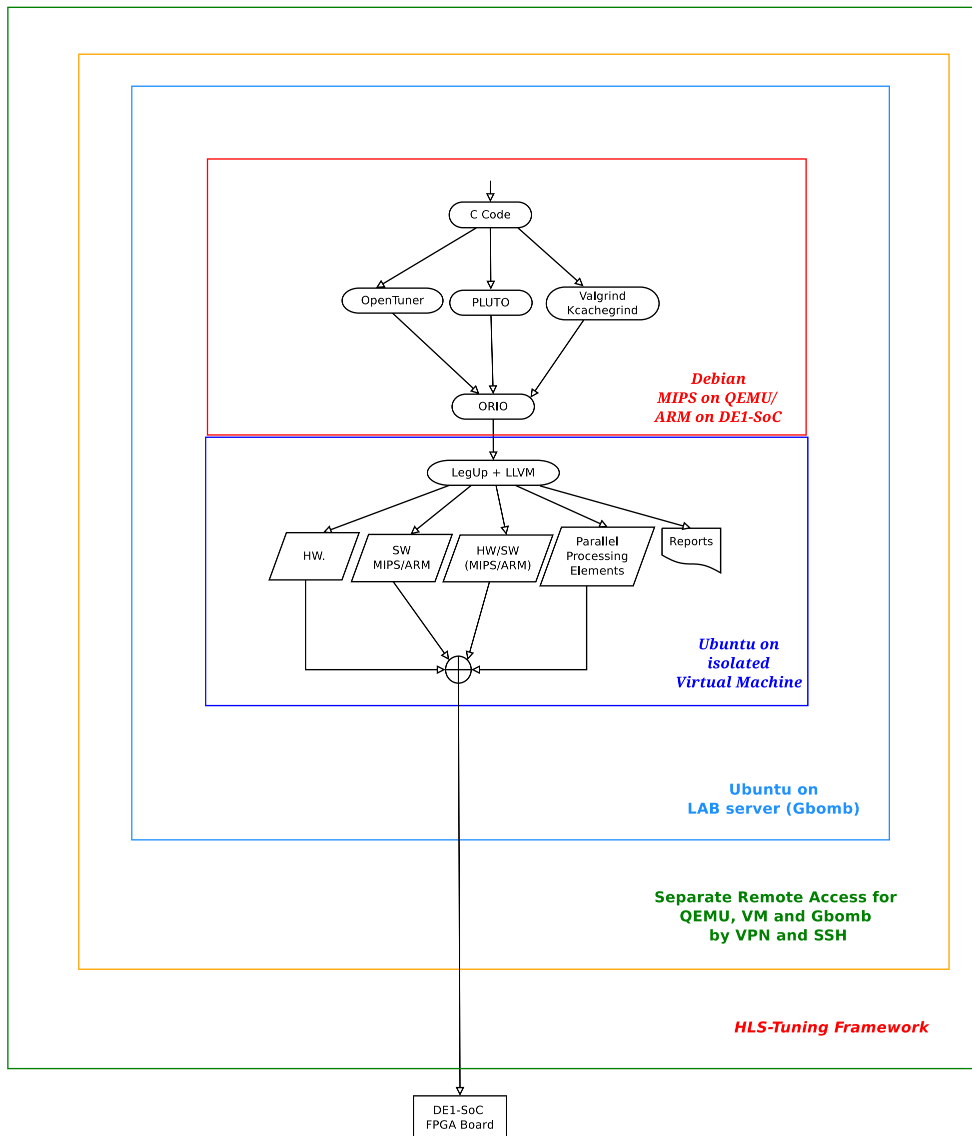

Figure 4.2: HOF Tool Chain

- Second, when we prepare the code for S2S[1] transformation by Orio, we annotate the most important part of the application which is the hot function in order to have the highest performance improvement.
- Third, in hybrid[2] test, and synthesis, we use this prior knowledge for picking the part of the code to be executed on the accelerator to result in a better performance.

This section is automatically executed by Valgrind and we manually pass its result to OpenTuner by focusing on the explored hot function.

OpenTuner After finding the hot function as the most important part of the application to be parametrized by OpenTuner, it tunes the critical parameter of the hot function with the best value to increase the execution performance of the code. In our experimental result, we used OpenTuner to find the best value for nested loop iterations in Block-Wise Matrix Multiplication[3] benchmark. Choosing the parameter to be tuned is carried out manually and the autotuning phase is done automatically by OpenTuner.

Orio After the parameterization phase, we write the specification code for the hot function and annotate it in the original code according to the Orio style. The specification code is written by us according to the original code. Despite it seems a time-consuming task to write specification file for all the codes but, we have written a template specification file that one needs to change the input parameter of the code and the target parameter besides choosing the desired search algorithm. However, the annotation of the hot function code must be manually carried out according to the valid Orio style. After determining the specification file and code annotation, we run the Orio framework by using the following command:

```
$ orcc -v -s test.spec test-nospec.c
```

The output of the above command is the best-transformed code with the best loop-unroll factor, it also provides the optimum GCC compiler optimization flag, but due to clang integration issues in Orio framework, we cannot get the corresponding Clang/LLVM flag for our LLVM-based HLS framework. Figure 4.3 illustrates the described procedure.

[1]Source to source
[2]Hardware/Software Co-design method,
[3]from here on we call it BWMM

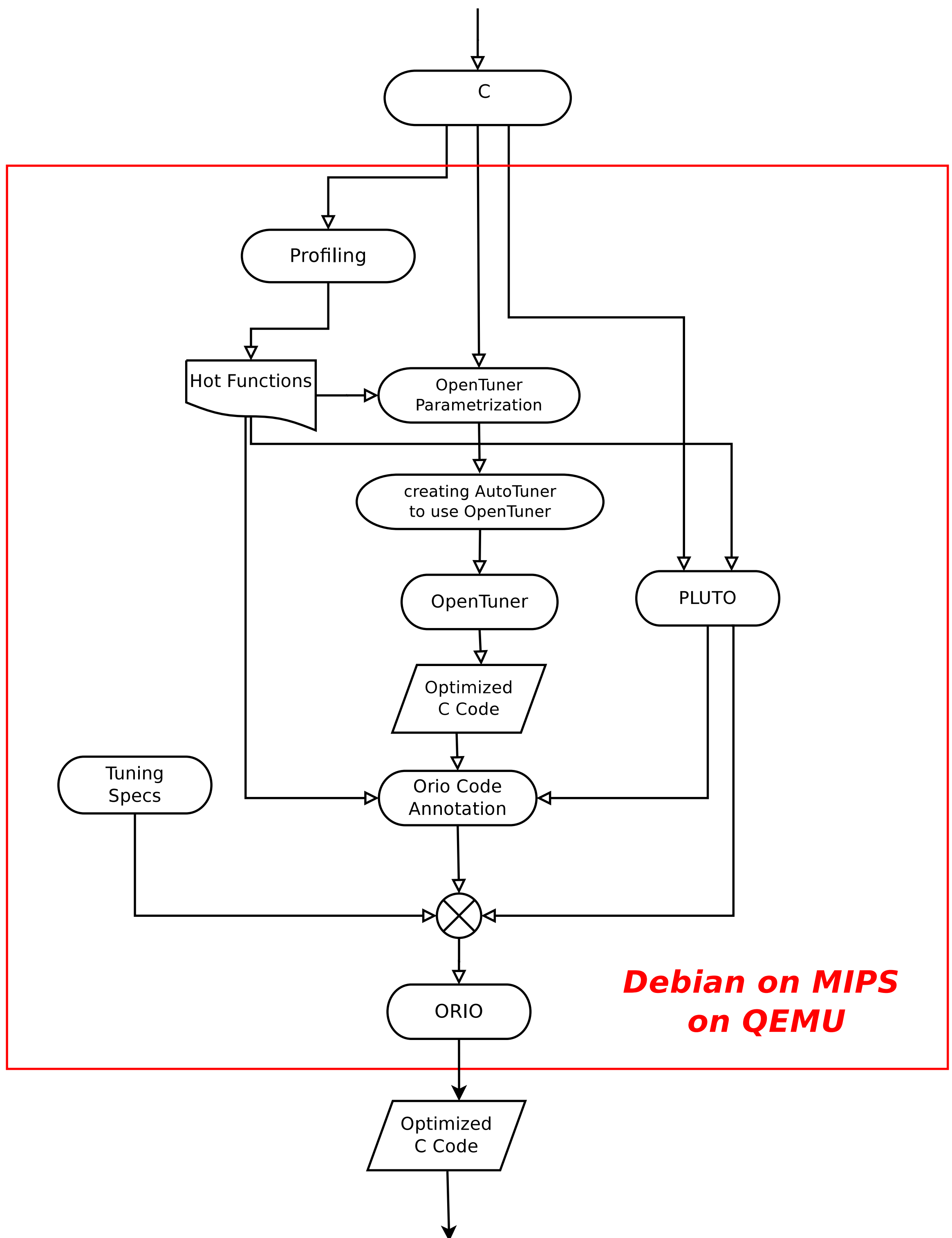

Figure 4.3: Methodology Diagram, first section

4.1.3 HLS Tuning

The second part of the framework flow as depicted in 4.4 is about different design methodologies. LegUp supports *parallel processing* exploiting POSIX threads. It converts each thread to a standalone accelerator which implies *data-parallel* type of parallel processing. Obviously, if we decide to have a parallel design, the selected code should be modified and rewritten using Pthreads, the number of threads is another tunable parameter. If we choose to run the optimized code on a processor a MIPS/ARM compiler make the binary file the MIPS simulator is invoked and the binary code is executed and some simulation results like the number of clock cycles are reported. In case of ARM processor, the binary code is executed on the actual ARM core on DE1-SoC board. As the Tiger-MIPS is synthesizable there is a possibility to synthesize the CPU on the FPGA and measure required are and maximum frequency, the same measurements are applicable for ARM processor.

If we decide on Pure hardware or HW/SW implementation, we annotate the code with some labels to signal pure hardware design method such as labels for loop pipelining or accelerator selections. Also, there is a config file named *config.tcl* which defines HLS optimizations, scheduling, binding, resource constraints, such multipliers latency, and data-path refactorizations, such as changing division unit.

Then several layers of makefiles are modified, optionally, to apply LLVM compiler optimization. These optimizations are applied on *Intermediate Representations*[4] and as investigated by [Jason Anderson et al. 2016][49] can drastically impact the HLS performance. we also observed the same similar behavior on your hybrid HLS tool, but the fine-grained exploration was not conducted in this research. Therefore the default *-O3* flag applied to the experiment. There is the possibility to benefit from phase-ordering, further explanation is available in 6.Then the modified IRs are processed by LegUp according to the selected design methodology it includes adding a wrapper for CPU and accelerators for HW/SW method. The result of this step is binary and HDL codes which feed Quartus II, there interfacing modules, called AVALON are added to project and ModelSim in invoked for Simulation. It also can synthesize the whole project for Cyclon V FPGA to generate SRAM Object File to program the FPGA on DE1-SoC development board. The simulation and synthesis reports are used for performance improvement.

Compiler Optimization Component

Compilers perform their optimizations in passes, where each pass is responsible for a specific code transformation. Examples of passes include dead-code

[4] IR

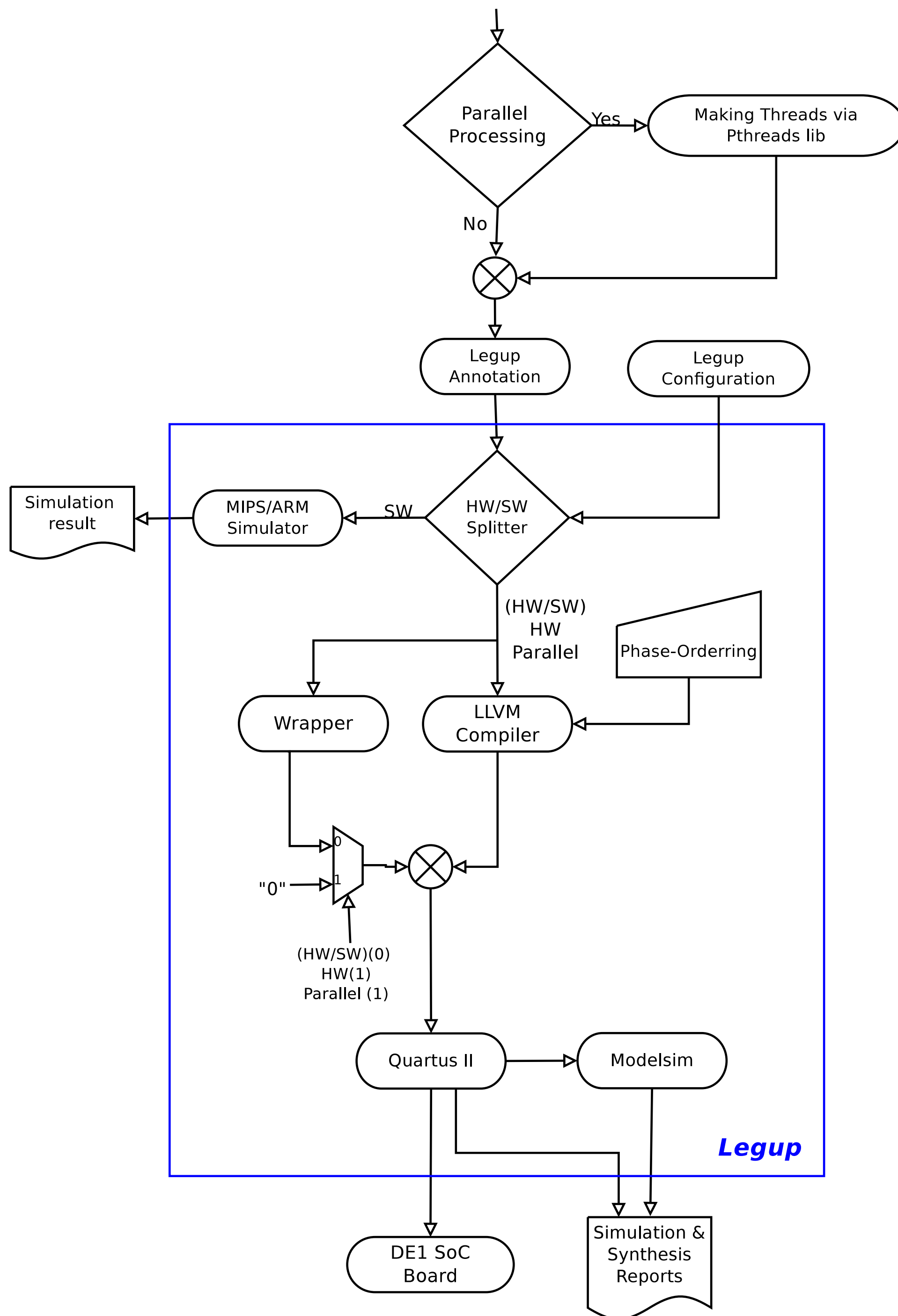

Figure 4.4: Methodology Diagram, second section

elimination, constant propagation, loop unrolling, and loop rotation. LLVM contains 56 such optimization (transform) passes that may alter the program, as well as many other passes that analyze the code to provide decision-making data for transform passes (see http://llvm.org/docs/Passes.html). The familiar command-line optimization levels (e.g., -O3) correspond to a particular set and sequence of compiler passes. The compiler passes within LLVM were intended to optimize software programs that run on a microprocessor.

Compilers such as LLVM and GCC provide standard optimization levels that can be selected by the user. Higher optimization levels typically cause the compiler to perform more passes in an attempt to better optimize the generated result. The level is normally set by a compiler parameter, as in -O1, -O2, and -O3. The particular optimizations applied at each level are chosen to benefit the average results for a collection of benchmark programs. However, it is not guaranteed that a higher optimization level will give a better result for a specific program. This has led the (software) compiler community to consider selecting a particular set of compiler optimization passes on a per-program (or even per code segment) basis. An example of this code-specific passes is available in Appendix C.

Despite the conventional compiler passes which are usually applied on IR[5], we also used code transformation on a higher level abstraction, meaning that the framework tries to tune the code for different design method via different code transformation in different levels of abstractions.

[5]Intermediate Representation

Chapter 5

Experimental Results

5.1 Introduction

In this section, we discuss the several experiments' performance results of Hybrid Optimization Framework in different design methods, Pure Hardware, Pure Software and Hardware/Software. High-Level Synthesis, or behavioral synthesis, is the technology which automatically translates behavioral level design descriptions or C software, into register-transfer level (RTL) counterpart.

In this experiments, our target platform for optimizations is an emulated MIPS machine implemented on QEMU as explained in Chapter 2, Target Architecture section. Our testing data-set is some well-known pieces of codes and CHStone benchmark suite which includes several well-known algorithms. All selected codes are widely used in the field of computer architectures, but as Hybrid Optimization Framework is based on both software autotuning and HLS optimizations we selected a mixture benchmarks pertaining to both domains.

All codes were compiled with the LLVM compiler using the *-O3* optimization flag to enable auto-vectorization and other advanced optimizations while the vectorization of LLVM is not supported by current LegUp free version, and they are synthesized by default LegUp options.

In the following sections, we refer to the input application code as "Original"[1], the Hybrid Optimization Framework as "HOF" and High-Level Synthesis as "HLS". The code tuned by OpenTuner and OpenTuner plus Orio are referred to as "OT-Tuned" and "OT-OR-Tuned", respectively. Apart from the first benchmark, you can find all the annotated and specification codes of all the benchmarks in Appendix C.

- Block-wise Matrix Multiply It is possible to use a block partitioned matrix product that involves only algebra on sub-matrices of the fac-

[1] In some research papers, it is known as the baseline.

tors. The partitioning of the factors is not arbitrary, however, and requires "conformable partitions" between two matrices A and B such that all sub-matrix products that will be used are defined. A detailed explanation is available in 5.2.

- AXPY Basic Linear Algebra Subprograms is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products and linear combinations.

- MATRIX MULTIPLY Matrix multiplication (MM) of two matrices is one of the most fundamental operations in linear algebra. The algorithm for MM is very simple, it could be easily implemented in any programming language, and its performance significantly improves when different optimization techniques are applied.

- ADPCM ADPCM (Adaptive Differential Pulse Code Modulation) implements the CCITT G.722 ADPCM algorithm for voice compression. It includes both encoding and decoding functions, which can be pipelined. The two functions can be also used as independent benchmark programs.

- DFADD DFADD implements IEC/IEEE-standard double-precision floating-point addition using 64-bit integer numbers. A number of the control statements such as *if* and *goto* statements are used. No loop exists except one *for* statement used as a testbench which is added by the authors. This program can be pipelined.

- DFDIV DFDIV implements IEC/IEEE-standard double-precision floating-point division using 64-bit integer numbers. A number of the control statements such as *if* and *goto* statements are used. DFDIV has several common functions with DFADD. DFDIV contains data-dependent loops, which make it difficult to be pipelined.

- DFMUL DFMUL implements IEC/IEEE-standard double-precision floating-point multiplication using 64-bit integer numbers. A number of the control statements such as *if* and *goto* statements are used. No loop exists except one *for* statement, used as a testbench, which is added by the authors. DFMUL has several common sub-functions which are also used in DFADD and DFDIV. This program can be pipelined.

- DFSIN DFSIN implements double-precision floating-point sine function using 64-bit integer numbers. A number of the control statements such as *if* and *goto* statements are used. It calls DFADD, DFMUL, and DFDIV, which are also included in CHStone.

- GSM This is a program for LPC (Linear Predictive Coding) analysis of GSM (Global System for Mobile Communications), which is a communication protocol for mobile phones. Only lossy sound compression GSM is implemented.

- MIPS This program describes instruction-level behaviors of a simplified MIPS processor which has 30 types of instructions. A sorting program is served as test vectors. Depending on synthesis options, HLS tools may synthesize a sequential processor or a pipelined one from the program.

- JPEG JPEG (Joint Photographic Experts Group) transforms a JPEG image into a bit-mapped image. This program is mainly composed of three parts: *huffman*, *idct*, and *inverse quantization*. An intelligent behavioral synthesis tool may pipeline the three functions. Alternatively, the three functions can be used as individual benchmark programs.

In Figure 5.1, the CHStone benchmark suite programs have been briefly described. Figure 5.2 and Figure 5.3 illustrate the characteristics of each program and resource utilization in RTL, respectively.

Application domain	Name	Description	Source
Arithmetic	DFADD	Double-precision floating-point addition	SoftFloat
	DFDIV	Double-precision floating-point division	SoftFloat
	DFMUL	Double-precision floating-point multiplication	SoftFloat
	DFSIN	*Sine* function for double-precision floating-point numbers	Authors' group, SoftFloat
Microprocessor	MIPS	Simplified MIPS processor	Authors' group
Media processing	ADPCM	Adaptive differential pulse code modulation decoder and encoder	SNU
	GSM	Linear predictive coding analysis of global system for mobile communications	MediaBench
	JPEG	JPEG image decompression	Authors' group, The Portable Video Research Group

Figure 5.1: Brief description and source of the CHStone benchmark programs [1].

	Representative	Lines of		Variables		Operations						Statements					
	data type	C code	Functions	Scalar	Array	Addition/Subtraction	Multiplication	Division	Comparison	Shift	Logic	*if*	*switch*	*while*	*for*	*goto/break*	*assignment*
DFADD	64-bit integer	526	17	121	4	38			78	65	146	87			1	26	299
DFDIV	64-bit integer	436	19	111	4	45	8	2	50	56	73	47		2	1	11	220
DFMUL	64-bit integer	376	16	92	4	28	4		34	41	61	38			1	9	159
DFSIN	64-bit integer	755	31	285	3	141	17	2	196	214	357	216		3	1	58	864
MIPS	32-bit integer	232	1	32	5	17	2		12	22	23	3	3	1	3	34	66
ADPCM	Array of 32-bit integers	541	15	269	26	156	69	2	73	81	24	52	24		25	97	792
GSM	Array of 16-bit integers	393	12	150	10	251	53		110	44	41	95		1	17	30	492
JPEG	Array of 32-bit integers	1,692	30	390	48	1,029	148	6	242	277	132	213	64	27	90	228	2,666

Figure 5.2: Source-level characteristics [1].

	DFADD	DFDIV	DFMUL	DFSIN	MIPS	ADPCM	GSM	JPEG
No. of states	35	52	25	121	19	155	236	815
32-bit Adders/Subtracters	4	4	4	4	3	16	18	14
Multipliers				1		4	1	4
Dividers						1		1
Comparators *1	9	8	6	13	33	9	4	10
Shifters					2	1	2	2
64-bit Adders/Subtracters	5	3	3	6				
Multipliers		4	4	4	2 *2			
Dividers		1		1				
Comparators	8	5	5	10				
Shifters	6	2	2	6				
ROMs (bits)	17,024	12,416	12,032	12,800	1,920	17,664	7,296	65,712
SRAMs (bits)					3,072	9,408	3,408	160,448
Registers (bits)	1,917	1,862	574	4,935	491	3,698	1,409	5,032

Figure 5.3: The number of states and resource utilization in RTL description [1].

5.1.1 Preliminary Definitions

In our experimental result, the performance metrics are the following:

- ***Cycle Latency:*** number of clock cycles for execution
- ***FMax:*** Maximum possible hardware clock frequency to run the circuit
- ***Clock Period:*** The inverse of clock $Fmax$, $\frac{1}{Fmax}$
- ***Wall-Clock Time:*** It is the key performance metric for HLS, computed as the product of cycle latency and the clock period.$cyclelatency \times clockperiod$
- ***ALMs:***The number of used Adaptive Logic Module (ALM) which indicates the area consumed on the FPGA
- ***DSP Block:*** The number of used Digital Signal Processing units on the FPGA
- ***Area Delay:*** HLS projects are considered as multi-objective optimization problems in which a trade-off among several metrics is taken into account. *Delay* and *Area* are two prominent performance criteria that are related inversely, meaning that, assuming a reasonable design, the design space,mathematically called feasible region, includes solutions which introduce faster circuits but bulkier and vice versa. Evidently it is designers' decision to select between different optimum extremes according to desired specifications.Whenever there is a multi-objective problem there is not a unique optimum solution.But,evidently, there is a point in the space of contrasting optimum solutions where both metrics can have a minimum value, relatively. In theory the curve of different optimums is called Pareto

front. In the context of HLS problem we are interested in smaller but faster solutions, hence considering different design approach we calculate $AREA \times DELAY$ to find our desired optimum which is actually the minimum value of this metric. figure A.1 shows the Pareto front for two conflicting objective function. In this context axis are labeled by *Area* and *Cycle latency*.

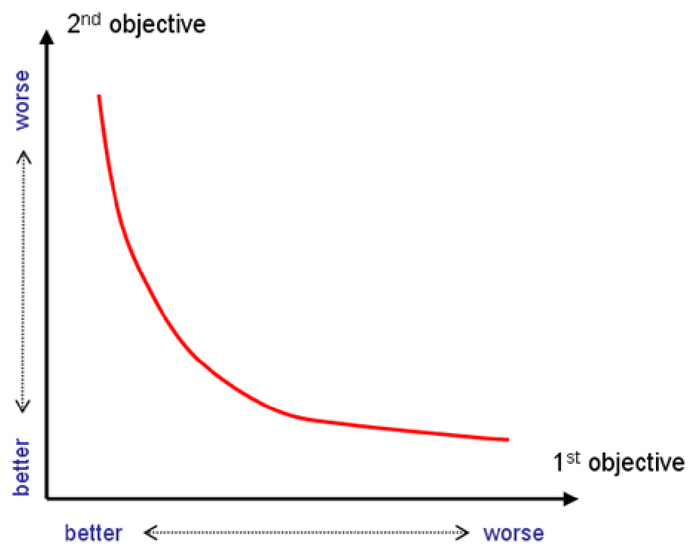

Figure 5.4: Pareto Curve

5.2 Block-Wise Matrix Multiply Benchmark

Since the Block-Wise Matrix Multiplication (BWMM) has only one main function and we know a priori the hot function thereby we skip the profiler step and directly apply OpenTuner to find the optimal value of the block size parameter, making sure to take the block size as a compile-time constant to the program.

Blocking a matrix multiply routine works by partitioning the matrices into sub-matrices and then exploiting the mathematical fact that these sub-matrices can be manipulated just like scalars. For example, suppose we want to compute $C = A * B$, where A, B, and C are each $8 * 8$ matrices. As shown in equation 5.1,we can partition each matrix into four $4 * 4$ sub-matrices, Figure 5.5 demonstrate the process graphically and Listing 5.1 indicates the BWMM original code written in C++.It shows one version of blocked matrix multiplication which we call the b_{ijk} version. The basic idea behind this code is to partition A and C into $1 \times bsize$ row slivers and to partition B into $bsize \times bsize$ blocks. The innermost (j, k) loop pair multiplies a sliver of A by a block of B and accumulates the result into a sliver of C. The i loop iterates through n row slivers of A and C, using the same block in B.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{11} &= A_{11}B_{11} + A_{12}B_{21} \end{aligned} \tag{5.1}$$

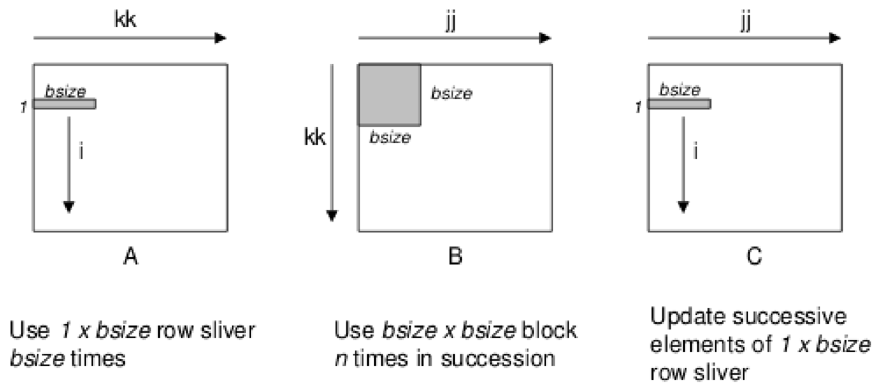

Figure 5.5: Graphical interpretation of blocked matrix multiply The innermost (j, k) loop pair multiplies a $1 * bsize$ sliver of A by a $bsize * bsize$ block of B and accumulates into a $1 * bsize$ sliver of C

Listing 5.1: BWMM original code

```
#include <stdio.h>
#include <cstdlib>
#define N 100

int main(int argc, const char** argv)
{
  int n = BLOCK_SIZE * (N/BLOCK_SIZE);
  int a[N][N];
  int b[N][N];
  int c[N][N];
  int sum=0;
  for(int k1=0;k1<n;k1+=BLOCK_SIZE)
  {
      for(int j1=0;j1<n;j1+=BLOCK_SIZE)
      {
          for(int k1=0;k1<n;k1+=BLOCK_SIZE)
          {
              for(int i=0;i<n;i++)
              {
                  for(int j=j1;j<j1+BLOCK_SIZE;j++)
                  {
                      sum = c[i][j];
                      for(int k=k1;k<k1+BLOCK_SIZE;k++)
                      {
                          sum += a[i][k] * b[k][j];
                      }
                      c[i][j] = sum;
                  }
              }
          }
      }
  }
  return 0;
```

```
}
```

5.2.1 Code Annotation

Listing 5.2 shows the program written for tuning BWMM. This code consists of several components, it creates an instance of class *GccFlagsTuner*, which tunes specified parameters using *opentuner.class GccFlagsTuner (MeasurementInterface)*. The manipulator method defines the variable search space by specifying parameters that should be tuned by this instance of *GccFlagsTuner*.

The run method actually runs OpenTuner under the given configuration and returns the calculated performance under this configuration. In this example, the *BLOCK_SIZE* parameter to be tuned is input as a compile-time constant that takes on a value within the specified range each time it is run. However, OpenTuner also supports other methods of specifying these parameters that may be preferred in different use cases. The manipulator method defines the variable search space by specifying parameters that should be tuned by this instance of *GccFlagsTuner*.

Listing 5.2: Autotuner program for OpenTuner to find optimum parameters for BWMM

```
#!/usr/bin/env python

import adddeps
import opentuner
from opentuner import ConfigurationManipulator
from opentuner import IntegerParameter
from opentuner import MeasurementInterface
from opentuner import Result

class GccFlagsTuner(MeasurementInterface):

  def manipulator(self):
    """
    Define the search space by creating a
    ConfigurationManipulator
    """
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      IntegerParameter('blockSize', 1, 10))
    return manipulator

  def run(self, desired_result, input, limit):
    """
    Compile and run a given configuration then
    return performance
    """
    cfg = desired_result.configuration.data

    gcc_cmd = 'g++ mmm_block.cpp '
```

```
    gcc_cmd += '-DBLOCK_SIZE='+ cfg['blockSize']
    gcc_cmd += '␣-o␣./tmp.bin'

    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0

    run_cmd = './tmp.bin'

    run_result = self.call_program(run_cmd)
    assert run_result['returncode'] == 0

    return Result(time=run_result['time'])

  def save_final_config(self, configuration):
    """called at the end of tuning"""
    print "Optimal␣block␣size␣written␣to␣mmm_final_config.json:",
        configuration.data
    self.manipulator().save_to_file(configuration.data,
                                    'mmm_final_config.json')

  if __name__ == '__main__':
      argparser = opentuner.default_argparser()
      GccFlagsTuner.main(argparser.parse_args())
```

Running the tunning program on the proposed platform, which is an emulated MIPS, results in finding the *BLOCK_SIZE=10* as the best size for BWMM code. This code then modified to be applicable for Orio tuning which consists of code annotations and specification as shown in Listing 5.3 and Listing 5.4 respectively. The code has been modified a bit syntactically to become adaptable for LegUp synthesis. Listing 5.5, you can find the final tuned and transformed code by Orio.

Listing 5.3: Annotated BWMM

```
#include <stdio.h>
#define N 50
#define BLOCK_SIZE 10

int main()
{
  int main_result;
  int n = BLOCK_SIZE * (N/BLOCK_SIZE);
  int a[N][N];
  int b[N][N];
  int c[N][N];
  int sum=0;
  int k1;
  for(k1=0;k1<n;k1+=BLOCK_SIZE)
  {
    int j1;
    for(j1=0;j1<n;j1+=BLOCK_SIZE)
    {
      int k1;
```

```
    for(k1=0;k1<n;k1+=BLOCK_SIZE)
    {
      int i;
      for(i=0;i<n;i++)
      {
        int j;
        for(j=j1;j<j1+BLOCK_SIZE;j++)
          {
             sum = c[i][j];
             int k;
             /*@ begin Loop (
               transform Composite(
               unrolljam = (['i'],[UF]),
               vector = (VEC, ['ivdep','vector always']))

               for(k=k1;k<=k1+BLOCK_SIZE-1;k=k+1)
               {
                 sum = sum + a[i][k] * b[k][j];
               }
             ) @*/

             for(k=k1;k<k1+BLOCK_SIZE;k++)
             {
               sum += a[i][k] * b[k][j];
             }
             /*@ end @*/

             c[i][j] = sum;
          }
        }
      }
    }
  }
  return main_result;
}
```

Listing 5.4: BWMM Orio transformation specification

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,10);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0','-O1','-O2','-O3'];
 }
 def input_params {
   param BLOCK_SIZE[] = [10];
```

```
    param N[] = [50];
  }
  def input_vars {
    decl dynamic int a[N][N] = random;
    decl dynamic int b[N][N] = random;
    decl int sum = random;
  }
  def search {
    arg algorithm = 'Exhaustive';
  }
}
```

Listing 5.5: BWMM tuned and transformed code

```
/**-- (Generated by Orio)
Best performance cost:
  [6.15e-07, 2.46e-07, 2.12e-07, 2.14e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.12e-07, 2.14e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.14e-07, 2.13e-07, 2.14e-07, 2.13e-07,
      2.14e-07, 2.14e-07, 2.13e-07, 2.14e-07, 2.13e-07, 2.14e-07,
      2.13e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07,
      2.13e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.14e-07, 2.13e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.12e-07, 2.14e-07, 2.13e-07, 2.12e-07, 2.13e-07,
      2.12e-07, 2.13e-07, 2.14e-07, 2.13e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.13e-07, 2.14e-07, 2.13e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.13e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.13e-07, 2.14e-07,
      2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.14e-07, 2.13e-07,
      2.14e-07, 2.14e-07, 2.13e-07, 2.14e-07, 2.14e-07, 2.14e-07,
      2.14e-07, 2.13e-07, 2.14e-07, 2.14e-07]
Tuned for specific problem sizes:
  BLOCK_SIZE = 10
  N = 50
Best performance parameters:
  CFLAGS = -O2
  UF = 3
  VEC = True
--**/

#include <stdio.h>
#define N 50
#define BLOCK_SIZE 10

int main()
{
  int main_result;
  int n = BLOCK_SIZE * (N/BLOCK_SIZE);
  int a[N][N];
  int b[N][N];
  int c[N][N];
  int sum=0;
```

```
  int k1;
  for(k1=0;k1<n;k1+=BLOCK_SIZE)
  {
    int j1;
    for(j1=0;j1<n;j1+=BLOCK_SIZE)
    {
      int k1;
      for(k1=0;k1<n;k1+=BLOCK_SIZE)
      {
        int i;
        for(i=0;i<n;i++)
        {
          int j;
          for(j=j1;j<j1+BLOCK_SIZE;j++)
          {
            sum = c[i][j];
            int k;
            /*@ begin Loop (
              transform Composite(
              unrolljam = (['i'],[UF]),
              vector = (VEC, ['ivdep','vector always']))

              for(k=k1;k<=k1+BLOCK_SIZE-1;k=k+1)
              {
                sum = sum + a[i][k] * b[k][j];
              }
            ) @*/
           {
            register int cbv_1;
            cbv_1=(k1+BLOCK_SIZE)-2;
#pragma ivdep
#pragma vector always
            for (k=k1; k<=cbv_1; k=k+2)
            {
              sum=sum+a[i][k]*b[k][j];
              sum=sum+a[i][k+1]*b[k+1][j];
            }
            register int cbv_2, cbv_3;
            cbv_2=(k1+BLOCK_SIZE)-((k1+BLOCK_SIZE-(0))%2);
            cbv_3=(k1+BLOCK_SIZE)-1;
#pragma ivdep
#pragma vector always
            for (k=cbv_2; k<=cbv_3; k=k+1)
            {
              sum+=a[i][k]*b[k][j];
            }
           }
           /*@ end @*/
            c[i][j] = sum;
          }
        }
      }
    }
  }
```

```
  return main_result;
}
```

5.2.2 Result Tables

After S2S transformation, the LLVM optimizations are applied. In the next step, different LegUp synthesis options have been considered to observe their effect over all the type of synthesis methods.

Table 5.1 indicates the result of applying different LegUp synthesis options on variant *HLS* methods, Pure Hardware, Pure Software and Hardware/Software. The first column of the table shows the measured performance indexes such as *Cycles*, *FMax*, *Wall-Clock Time* or execution time, *ALMS* or the number of computational units, *DSP Blocks*, and *Area Delay* which is the production of the execution time multiply by number of resource units as the main criterion.

The Second column is the result of *original* code without any optimization synthesis as the baseline to determine HOF performance improvement. For this benchmark and to have a better vision about our framework, we added our optimization levels step by step. As you can see, the second and third columns represent the synthesis result of original code after adding OpenTuner and then adding Orio to it as the second and third HOF optimization levels called *OT-Tuned* and *OT-OR-Tuned* respectively.

Each column contains three sub sections representing three different LegUp synthesis options as the last level of tuning parameters. By activating *Pipelining* option, LegUp pipelines the labeled loop and by activating *Mul Lat=0* option, we assigned the multiplication latency to zero.

Since we used the free version of LegUp we had the constrains preventing us to have the pipelining for *OT-OR-Tuned* case and consequently, its result is Not Available on the table. *Wall-Clock Time* and *Area-Delay* are highlighted as the most important rows.

FMax is generally affected by circuit critical path, the longer critical path which is the result of a more complex combinational circuit needs a longer clock period to finish the computations, hence fewer cycles in the time steps. Usually, multipliers play a significant role in prolonging clock period thereby synthesizing a complex multiplier needing fewer clock cycles makes the FMax less too.

Table 5.1: OpenTuner in HLS Framework

	Original			OT-Tuned			OT-OR-Tuned		
	Pure HW	Pipelining	Mul lat[a]=0	Pure HW	Pipelining	Mul lat=0	Pure HW	Pipelining	Mul lat=0
Cycles	6353322	3353322	3102222	2825462	1137962	1075312	1575462	N/A	1262812
FMax	116	116	79.82	113	97.46	84.46	125.64	N/A	102.07
Clock Period	8.62	8.62	12.53	8.85	10.2	11.8	7.96	N/A	9.80
Wall-Clock Time	54770.02	28907.95	38865.22	25004.09	11607.21	12688.68	12539.49	N/A	12372.02
ALMs	374	374	378	368	383	371	335	N/A	325
DSP Blocks	4	4	4	4	4	4	2	N/A	4
Area*Delay	20483986	10811573	14691054	9201505	4445562	4707501	4200730	N/A	4020906

[a] Multiplication latency

5.2.3 Performance Diagrams

In this section, we evaluate the performance improvement of HOF with respect to the baseline from different aspects in variant diagrams. Considering the trend on 5.6 shows the fact that a more parallel design decreases the wall clock time. As the clock frequencies in different design approaches are different, other metrics such as clock cycles and clock periods cannot reveal the aggregated speedup of the synthesized circuit. Clock Period is the inverse of frequency as mentioned before, decreasing the multiply latency results in longer clock period. By the way, the clock period alone is not a good performance measurement metric.

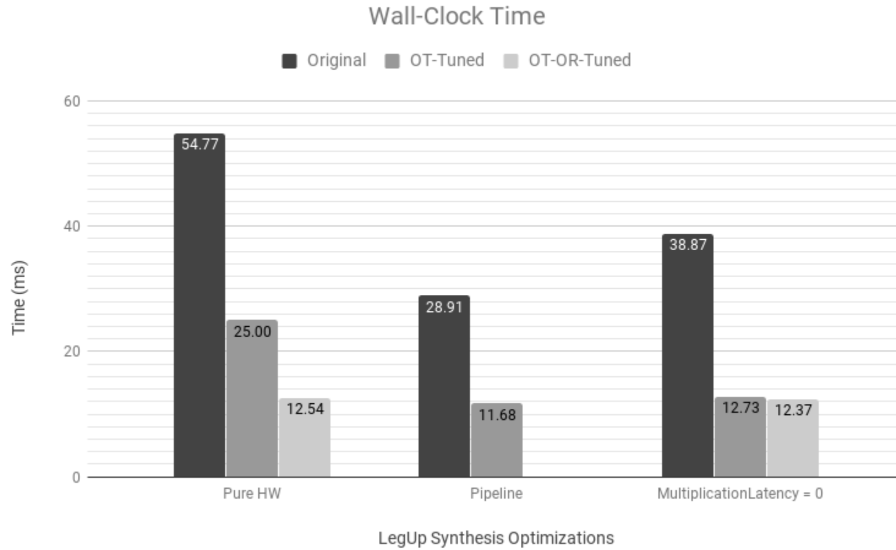

Figure 5.6: The BWMM Wall-Clock Time diagram of the original code vs. different optimization levels

The speedup result in Figure 5.7 shows that optimizations using LegUp options, *loop pipelining* and *multiplication latency=0*, besides OpenTuner and Orio tuning achieve much better performance than original code.

Reviewing 5.1 shows that there is a definite decrease on the number of clock cycles as moving form pure hardware to other synthesis optimizations. There is five times improvement but as the clock frequencies on different approached are not the same, this performance improvement is biased. The actual performance improvement is visible on 5.7. But, all in all, a more parallel hardware design is considered as a spatial decomposition and parallelization which always reduces the number of clock cycles.

The logic utilization efficiency for both *OT-Tuned* and *OT-OR-Tuned* versions is shown in Figure 5.8 that indicates the closer we are to the HOF

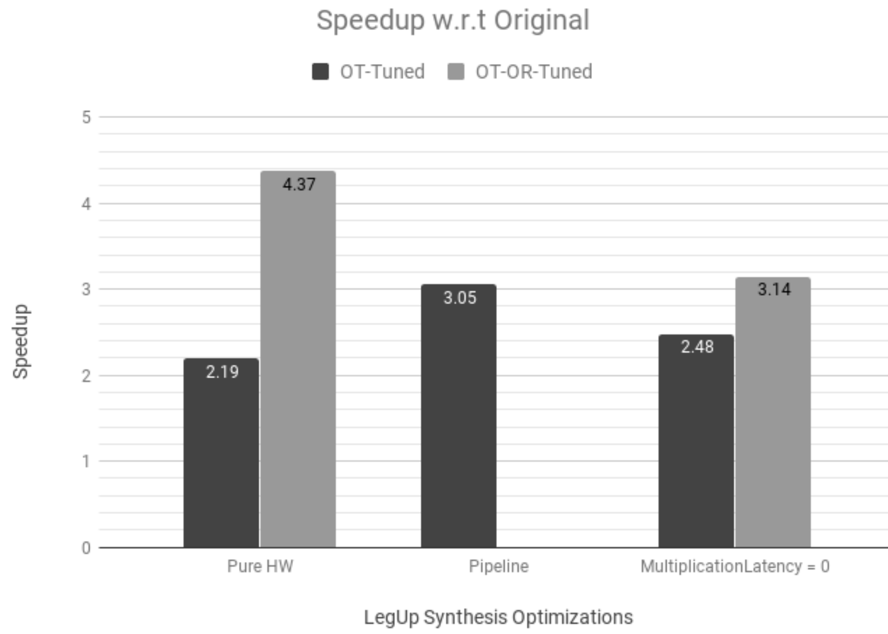

Figure 5.7: BWMM Speedup w.r.t baseline

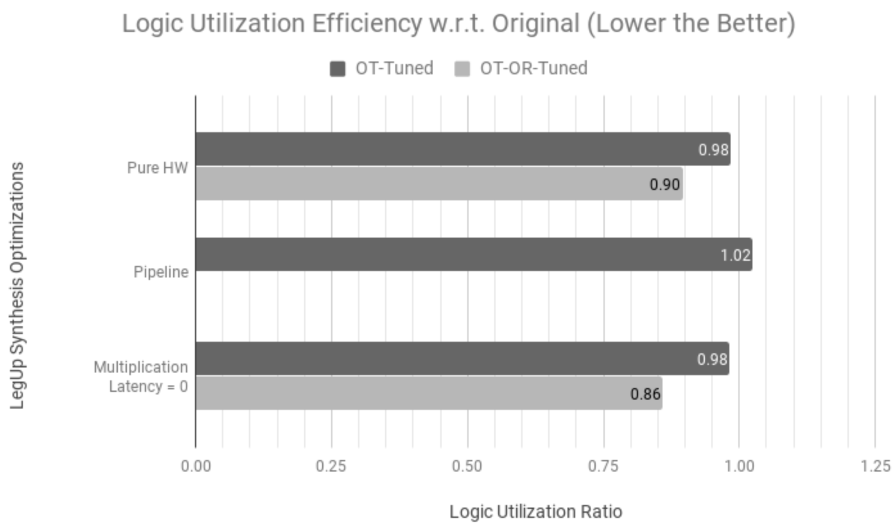

Figure 5.8: BWMM Logic Utilization Efficiency w.r.t baseline

full design the better we consume the computational resources.

Figure 5.9 contains the Area-Delay result, which shows that applying both *OT-Tuned* and *OT-OR-Tuned* or complete HOF toolchain on the original code always delivers performance boosts especially the latter one, which ranges from 100 percent to 387 percent. Furthermore, the performance of the OT-tuned[2] code and OT-OR-Tuned[3] are almost equivalent in some cases but the performance difference between them becomes more significant as more optimizations per phase are applied to facilitate the toolchain.

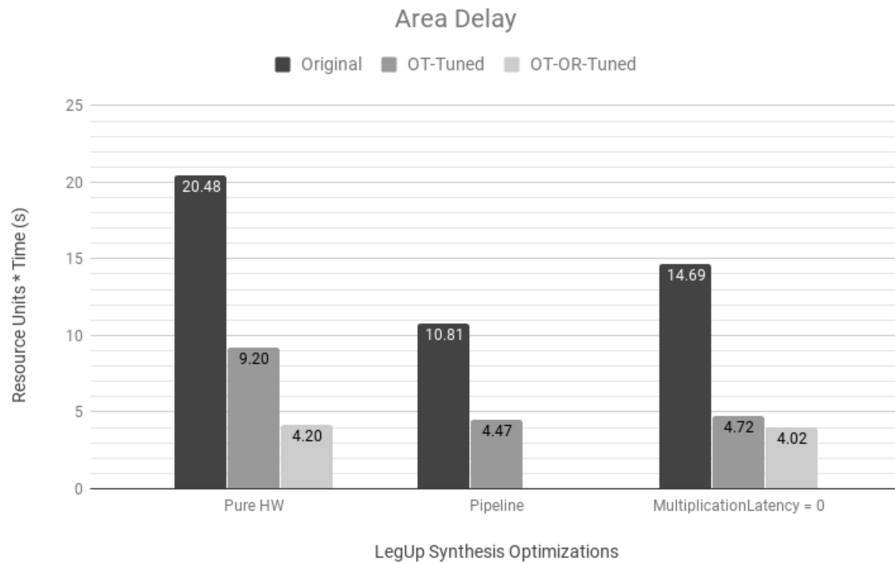

Figure 5.9: Area Delay of baseline vs. HOF

Not even the execution time but also the cache missing rate is enhanced as we applied the HOF on the original code illustrated in Figure 5.10. Exploiting the concept of the cache which increases the temporal and spatial localities in CPUs, we can explain the effect of code transformation on HLS. In the BWMM experiment, larger block size decreases the cache miss rate resulting larger parallel data flows wherein the domain of HLS infrastructure, a larger data flow normally makes use of more parallel resources which yields less execution time. Also, taking into account, the loop unrolling as the main effect of Orio code transformations, we discovered that unrolled loops make more parallel computational units. Besides, using LegUp pipeline option decreases the execution time as the code is executed in more parallel manner.

[2]Tuned only by OpenTuner
[3]Tuned by OpenTuner and Orio

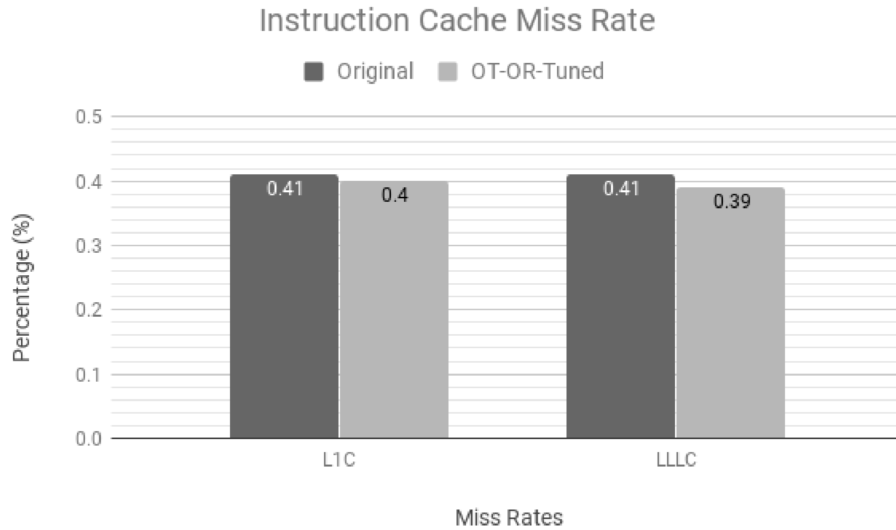

Figure 5.10: BWMM cache miss rate w.r.t baseline

Different Phases

After comparing different optimizations, we introduce different phases completing our toolchain levels. Figure 5.11 demonstrates the different phases speedup w.r.t the baseline through adding HOF pieces in a cumulative manner.

To demonstrate the HOF tuning effect on performance in details, in this experiment, we ran the HOF in eight phases by adding optimization levels incrementally as described below:

- Original: Original code without optimization
- Phase 1: Original code with multiplication latency=0 LegUp option
- Phase 2: Original code with pipelining LegUp option
- Phase 3: Original code with OpenTuner tuning
- Phase 4: Original code with OpenTuner tuning and multiplication latency=0 LegUp option
- Phase 5: Original code with OpenTuner tuning and pipelining LegUp option
- Phase 6: Original code with OpenTuner and Orio tuning
- Phase 7: Original code with OpenTuner and Orio tuning, and multiplication latency=0 LegUp option

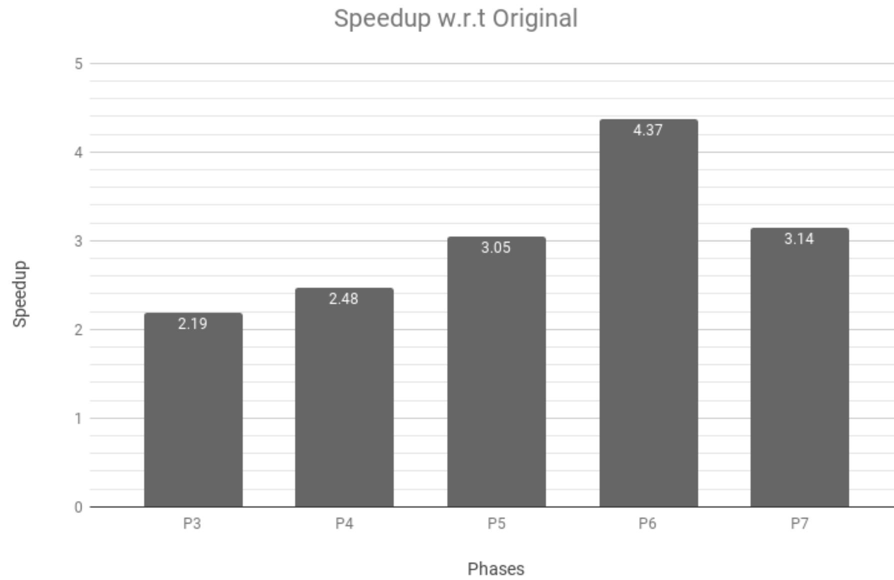

Figure 5.11: BWMM different phases speedup

The code is tuned in each optimization stage, it consistently outperforms passing each phase in which the last phase representing the complete HOF structure has the best performance among the others. Apart from having a better speedup as we walk through the phases, Figure 5.12 reveals the logic utilization efficiency compared to the original code where the HOF tuned code in the last phase has the most efficient computational unit usage.

Figure 5.13 contains the Area-Delay result as the main metric, which shows that applying complete HOF toolchain on the original code delivers the best performance boost, around 400 percent.

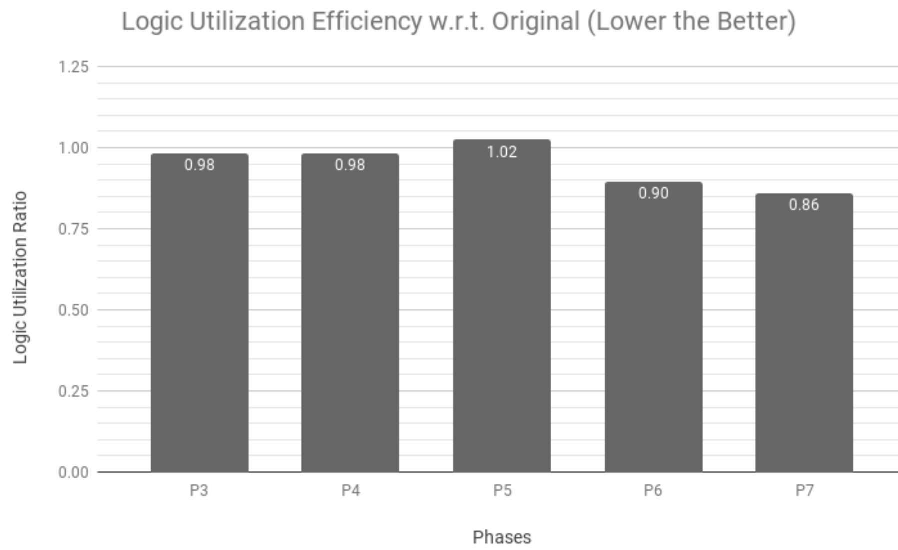

Figure 5.12: BWMM different phases logic utilization efficiency

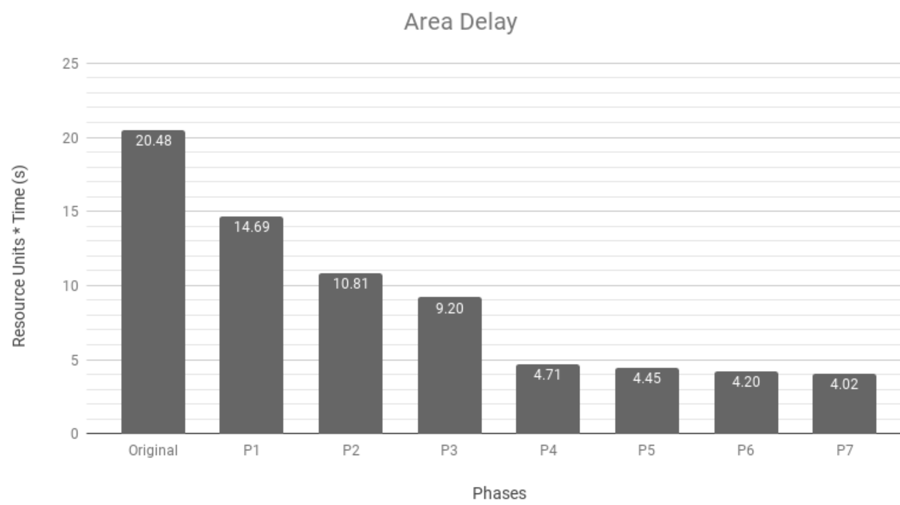

Figure 5.13: BWMM different phases Area Delay

5.3 AXPY Benchmark

In this experiment, we tuned the performance of the AXPY operation. Since it has only one main function and does not have any parameter to be tuned by OpenTuner so, we skipped the first two steps and used Orio directly. We measured the performance of two scenarios: the original code without any optimization and using HOF.

5.3.1 Result Tables

After S2S transformation, the LLVM optimization would be applied to the code which is implemented internally inside the LegUp. The results are shown in Tables 5.2, 5.3 and 5.4, representing *Pure Hardware*, *Pure Software* and *Hardware/Software* synthesis result, respectively.

Metrics	Original	With FW
Cycles	404	164
FMax (MHz)	127.31	125.44
Clock Period (ns)	7.855	7.972
Wall-clock Time (us)	3.17	1.31
ALMs	368	766
Area*Delay (us)	1167	1001
DSP Blocks	0	0
RAM Blocks	1	2
Blocks Memory Bits	3200	3200
Registers	200	605

Table 5.2: Pure Hardware AXPY HLS Analysis

Metrics	Original	With FW
Cycles	14040	12860
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	195.33	178.91
ALMs	4735	4735
Area*Delay (us)	924866	847136
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.3: Pure Software AXPY HLS Analysis

Metrics	Original	With FW
Cycles	10958	10958
FMax (MHz)	72.66	74
Clock Period (ns)	13.76	13.51
Wall-clock Time (us)	150.81	148.08
ALMs	5156	6070
Area*Delay (us)	777587	898852
DSP Blocks	16	16
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	6410	7891

Table 5.4: Hardware/Software AXPY HLS Analysis

5.3.2 Performance Diagrams

The performance results are shown in Figure 5.14 indicate that the code tuned by Orio consistently outperforms the original code without optimization. We observed that even for a simple algebraic operation, such as the composed AXPY routines, the compiler alone is unable to yield performance comparable to the HOF tuned version.

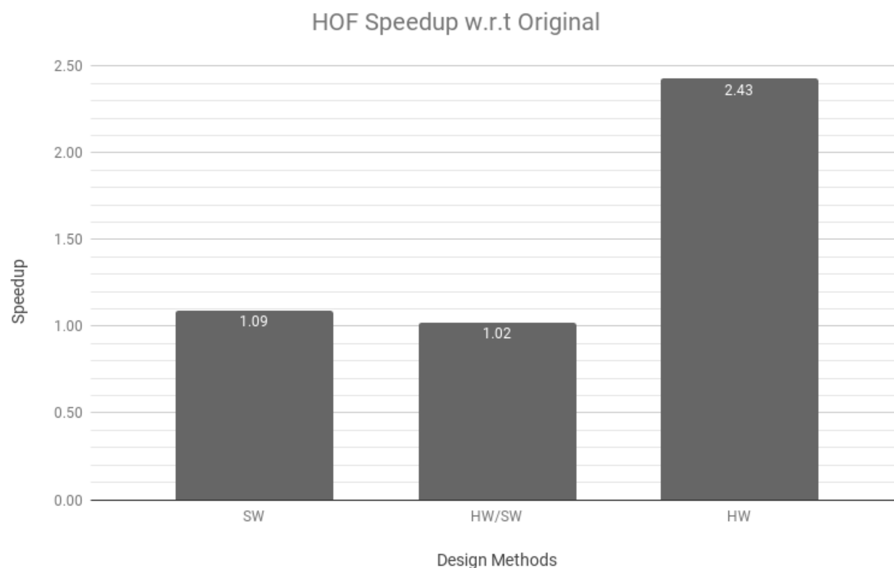

Figure 5.14: AXPY Speedup diagram

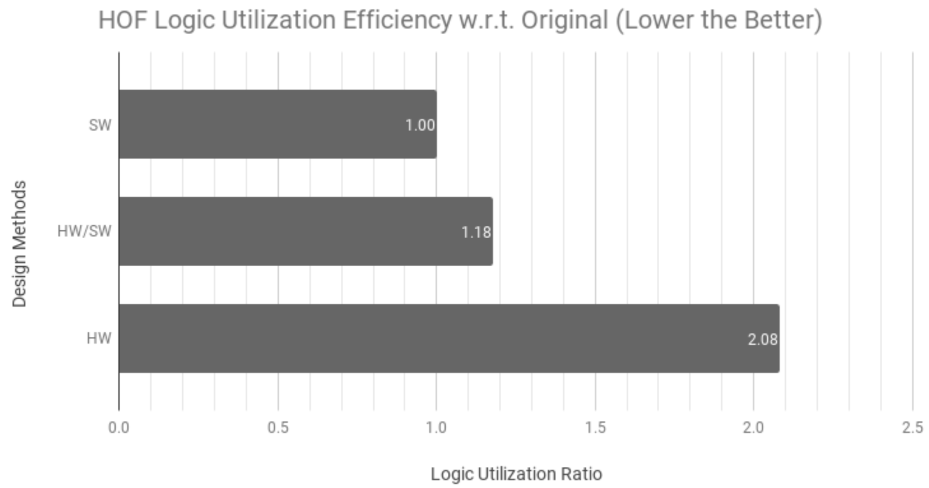

Figure 5.15: AXPY logic utilization efficiency diagram

We observed that in Pure Hardware method the logic utilization became doubled compare to the baseline Figure 5.15, on the other hand, it results in 2.5 times speedup and on aggregation 15% Area Delay improvement shown in Figure 5.16.

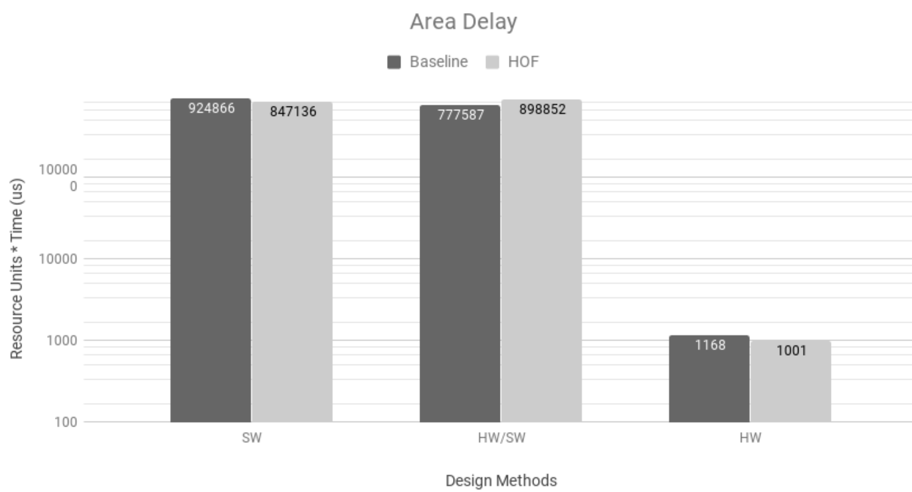

Figure 5.16: AXPY Area Delay diagram

5.4 Matrix Multiply Benchmark

In this section, we examine the effectiveness of HOF in optimizing by semi-automated tuning toolchain on one of heavily used benchmark Matrix Multiplication. Matrix Multiplication dominates the performance of various scientific applications. We used Orio to automatically select the best loop-unroll factor for only the loop that iterates over the matrices' elements and computes the production.

5.4.1 Result Tables

Respectively, tables 5.5, 5.6 and 5.7 indicate the results of variant HLS synthesis methods Pure Hardware, Pure Software and Hardware/Software.

Metrics	Original	With FW
Cycles	33243	10907
FMax (MHz)	112.75	145.71
Clock Period (ns)	8.869	6.863
Wall-clock Time (us)	294.84	74.85
ALMs	375	2005
Area*Delay (us)	110564	150083
DSP Blocks	2	2
RAM Blocks	6	4
Blocks Memory Bits	38400	25600
Registers	599	3846

Table 5.5: Pure Hardware Matrix Multiply HLS Analysis

Metrics	Original	With FW
Cycles	231332	151698
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	3218.31	2110.43
ALMs	4735	4735
Area*Delay (us)	15238690	9992905
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.6: Pure Software Matrix Multiply HLS Analysis

Metrics	Original	With FW
Cycles	111847	110858
FMax (MHz)	73.38	73.39
Clock Period (ns)	13.63	13.63
Wall-clock Time (us)	1524.22	1510.53
ALMs	5060	5038
Area*Delay (us)	7712535	7610064
DSP Blocks	8	8
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	6236	6209

Table 5.7: Hardware/Software Matrix Multiply HLS analysis

5.4.2 Performance Diagrams

We tested the performance of all the synthesis methods HW, SW and HW/SW includes Speedup, Logic Utilization Efficiency and Area Delay shown in Figure 5.17, Figure 5.18 and Figure 5.19, respectively.

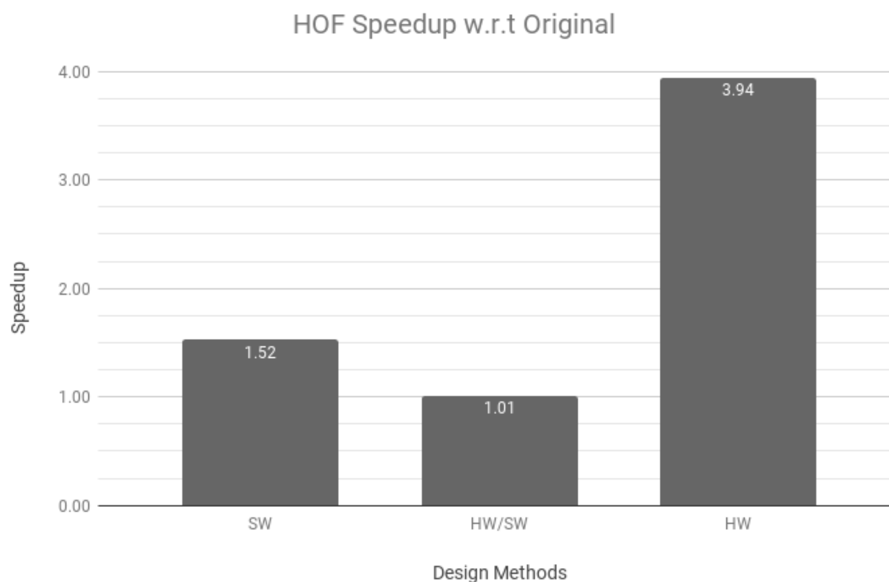

Figure 5.17: Matrix Multiply Speedup diagram

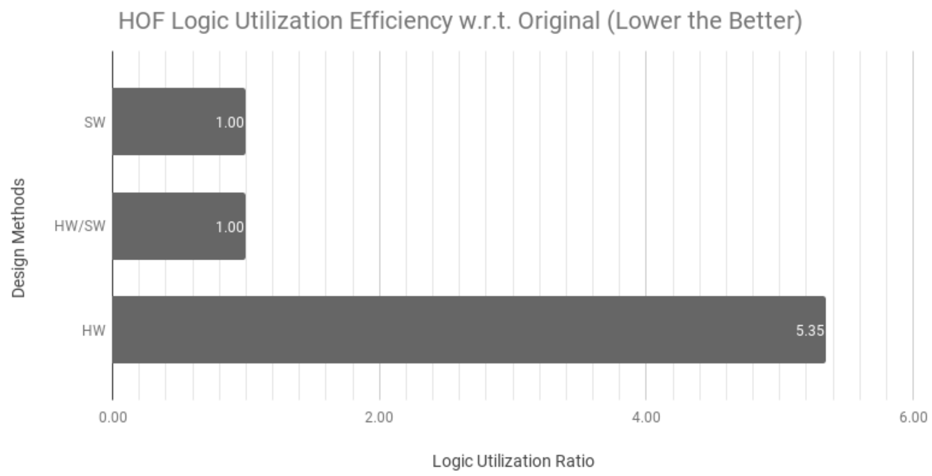

Figure 5.18: Matrix Multiply Utilization diagram

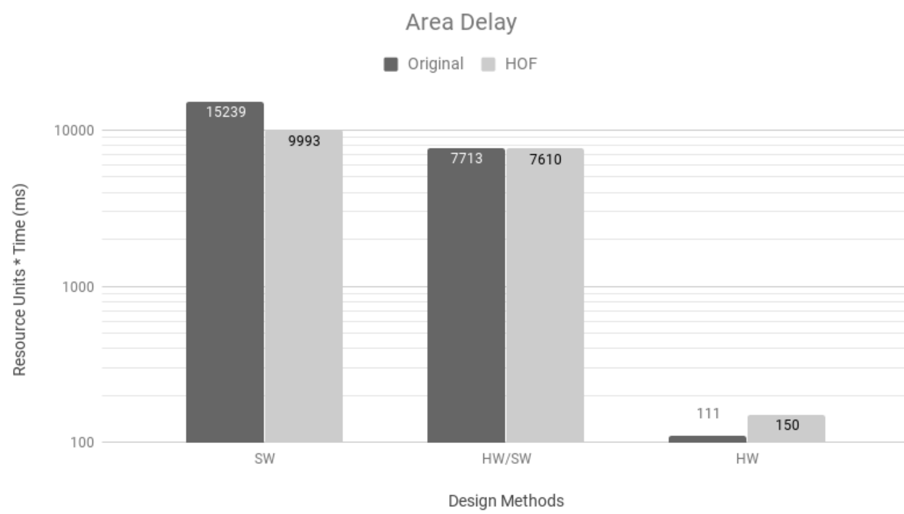

Figure 5.19: Matrix Multiply Area Delay diagram

5.5 DFMUL Benchmark

This experiment discusses the effect of HOF on DFMUL benchmark. First, the profiler performed the analysis over the original code and detected *float64_mul* as the hot function for this benchmark. Then, we tuned the hot function in the next step and used it for hybrid synthesis as the selected part for accelerating transformation. Figure 5.20 graphically demonstrates the result of Valgrind profiler.

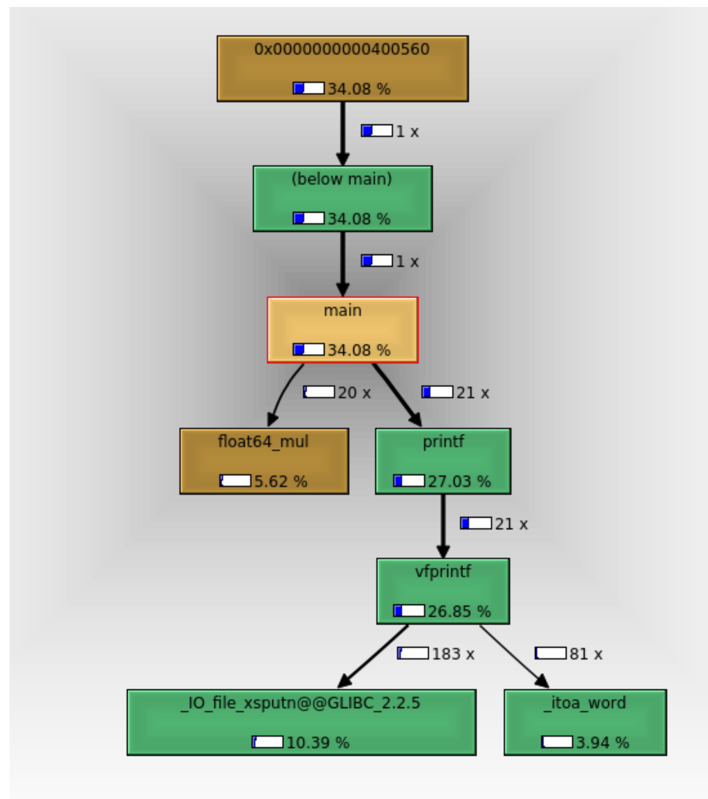

Figure 5.20: DFMUL Profiling. The Call-graph depicting calling relationships and computational needs of each function

5.5.1 Result Tables

Respectively, tables 5.8, 5.9 and 5.10 indicate the results of variant HLS synthesis methods Pure Hardware, Pure Software and Hardware/Software.

Metrics	Original	With FW
Cycles	216	134
FMax (MHz)	100.86	127.49
Clock Period (ns)	9.915	7.844
Wall-clock Time (us)	2.14	1.05
ALMs	1043	951
Area*Delay (us)	2234	1000
DSP Blocks	32	32
RAM Blocks	1	0
Blocks Memory Bits	8192	0
Registers	1364	1707

Table 5.8: Pure Hardware DFMUL HLS Analysis

Metrics	Original	With FW
Cycles	96551	8699
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	1343.22	121.02
ALMs	4735	4735
Area*Delay (us)	6360170	573035
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.9: Pure Software DFMUL HLS Analysis

5.5.2 Performance Diagrams

This section discusses the performance evaluation of the DFMUL benchmark. We compare the performance of the code tuned by HOF with the original code.

The speedup Figure 5.21 shows that optimizations using HOF achieve dramatically better performance than the original code, up to 15.45 times faster. We have also logic utilization efficiency, Figure 5.22, where we used 10% less computational units in Pure Hardware synthesis, improved the execution time by more 2 times.

Metrics	Original	With FW
Cycles	98188	6458
FMax (MHz)	69.82	70.93
Clock Period (ns)	14.32	14.10
Wall-clock Time (us)	1406.30	91.05
ALMs	5744	5766
Area*Delay (us)	8077798	524980
DSP Blocks	6	6
RAM Blocks	24	24
Blocks Memory Bits	152768	152768
Registers	7077	7076

Table 5.10: Hardware/Software DFMUL HLS Analysis

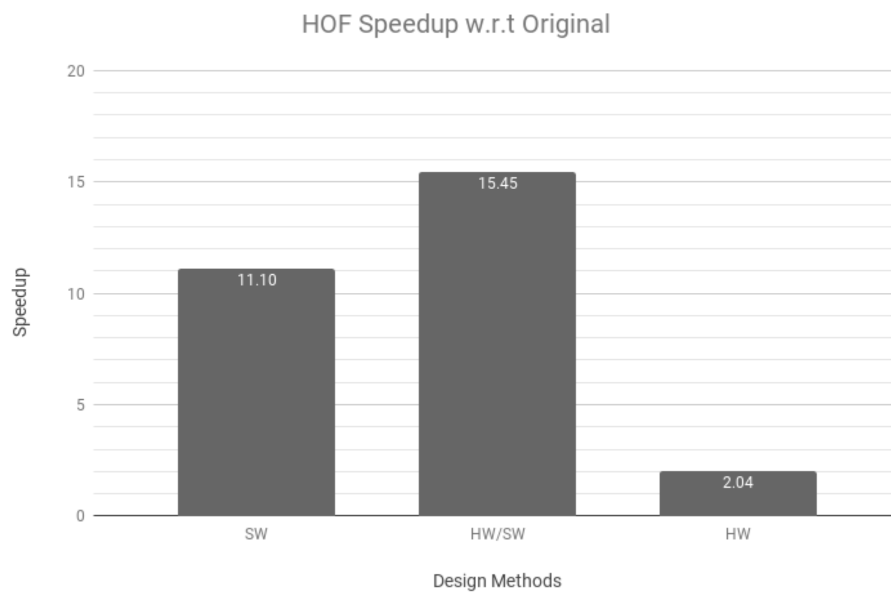

Figure 5.21: DFMUL Speedup diagram

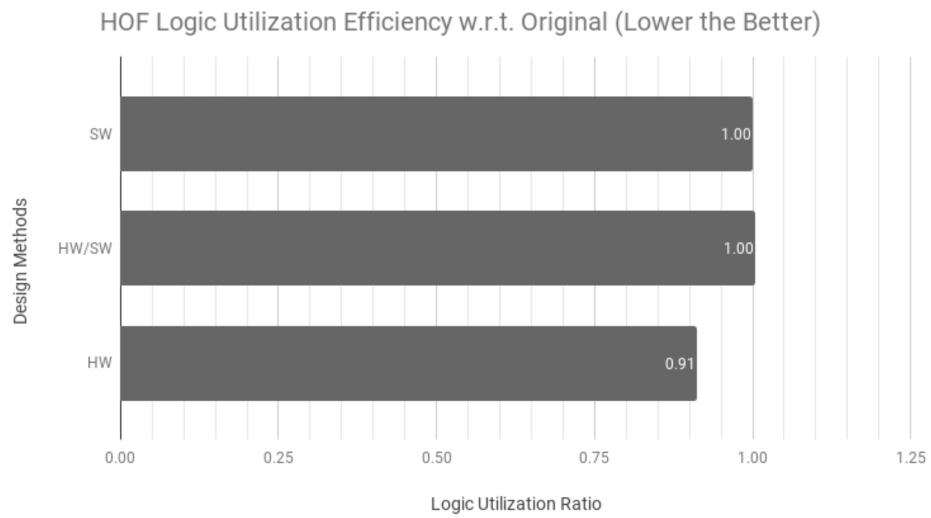

Figure 5.22: DFMUL Utilization diagram

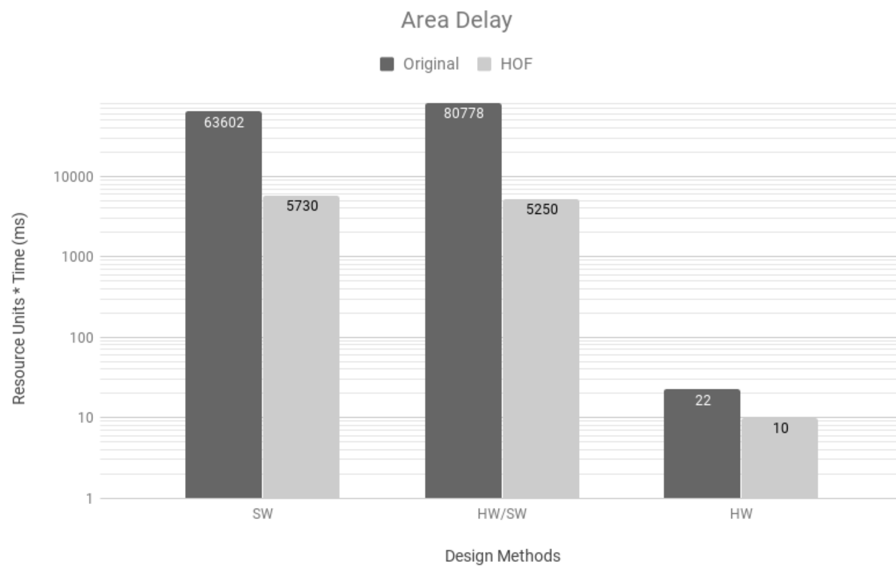

Figure 5.23: DFMUL Area Delay diagram

HOF further improves the Area-Delay and the performance of the original version by a factor of 2 to more than 15 times shown in Figure 5.23.

5.6 GSM Benchmark

In this experiment, we tuned the performance of the GSM benchmark by HOF optimizations. Figure 5.24 shows the output of Valgrind profiler indicating the possible hot functions *Autocorrelation* and *gsm-mult-r* in this benchmark. By considering the number of calls of each candidate, we have chosen the latter one as the selected function.

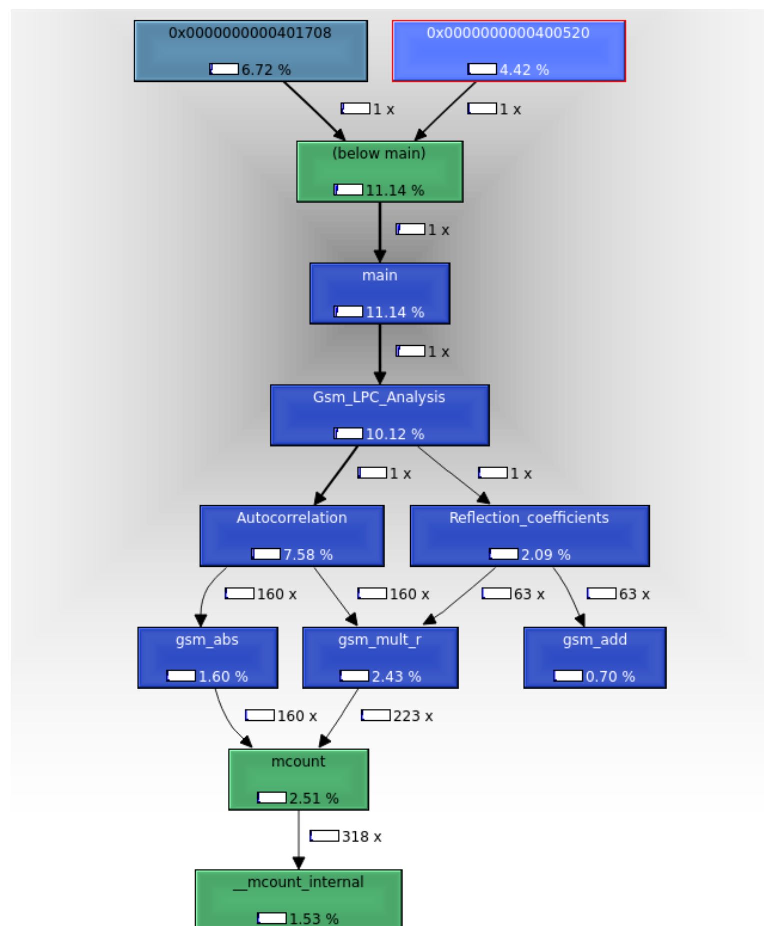

Figure 5.24: GSM Profiling. The Call-graph depicting calling relationships and computational needs of each function

5.6.1 Result Tables

Respectively, tables 5.11, 5.12 and 5.13 indicate the results of variant HLS synthesis methods Pure Hardware, Pure Software and Hardware/Software.

Metrics	Original	With FW
Cycles	4763	4441
FMax (MHz)	98.09	96.84
Clock Period (ns)	10.195	10.326
Wall-clock Time (us)	48.56	45.86
ALMs	3236	3271
Area*Delay (us)	157132	150005
DSP Blocks	54	54
RAM Blocks	7	7
Blocks Memory Bits	10144	10144
Registers	5190	5225

Table 5.11: Pure Hardware GSM HLS Analysis

Metrics	Original	With FW
Cycles	47970	47143
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	667.36	655.86
ALMs	4735	4735
Area*Delay (us)	3159960	3105483
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.12: Pure Software GSM HLS Analysis

5.6.2 Performance Diagrams

We tested the performance of all the synthesis methods HW, SW and HW/SW includes Speedup, Utilization Efficiency and Area Delay shown in Figure 5.25, Figure 5.26 and Figure 5.27, respectively.

Metrics	Original	With FW
Cycles	32059	30934
FMax (MHz)	72.47	72.47
Clock Period (ns)	13.80	13.80
Wall-clock Time (us)	442.38	426.85
ALMs	7095	7095
Area*Delay (us)	3138659	3028518
DSP Blocks	52	52
RAM Blocks	22	22
Blocks Memory Bits	156800	156800
Registers	9664	9664

Table 5.13: Hardware/Software GSM HLS Analysis

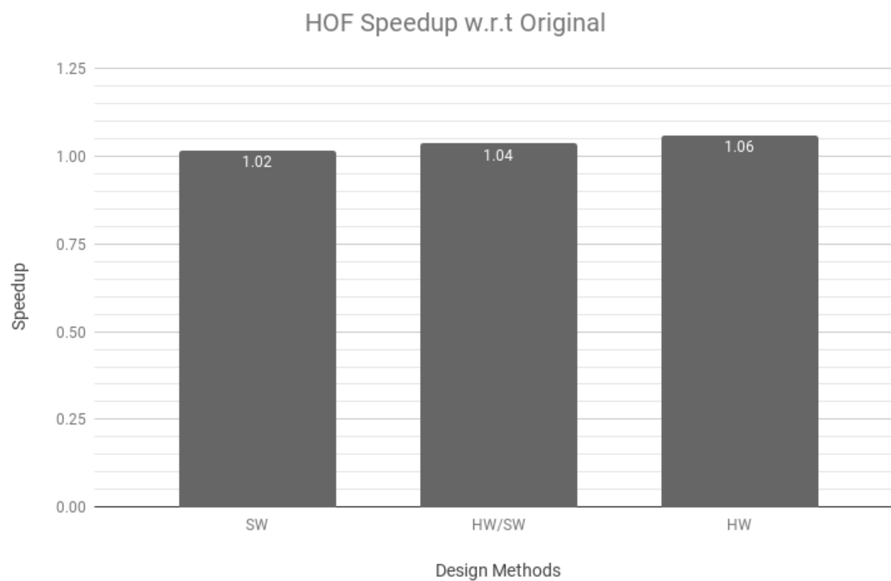

Figure 5.25: GSM Speedup diagram

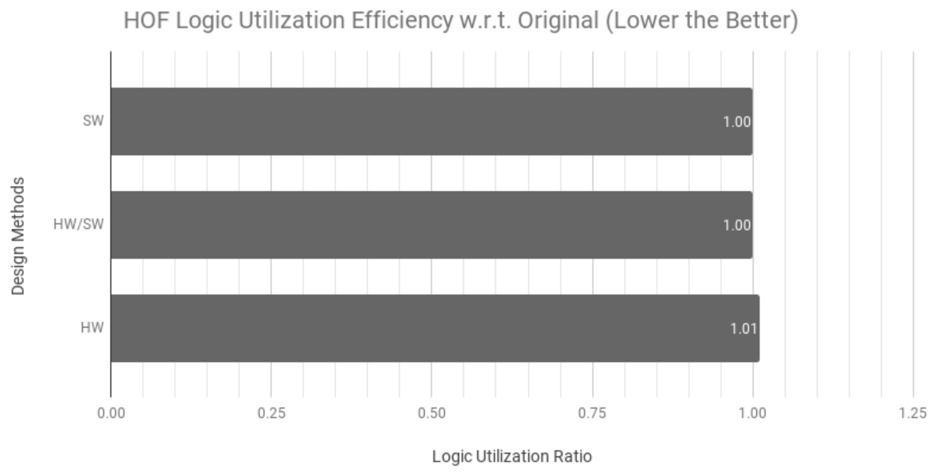

Figure 5.26: GSM Utilization diagram

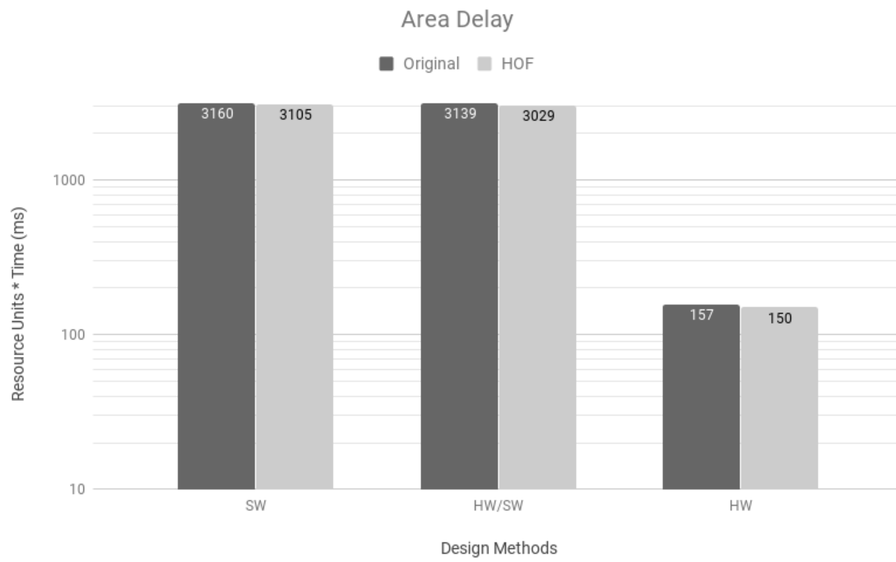

Figure 5.27: GSM Area Delay diagram

5.7 DFADD Benchmark

In this section we examine the effectiveness of HOF in optimizing DFADD benchmark. Figure 5.28 illustrates the output of Valgrind profiler indicating the possible hot functions *main*, *addfloaf64sigs* and *extractFloat64Exp* in this benchmark. By considering the number of calls of each candidate, we have chosen the last one as the selected function.

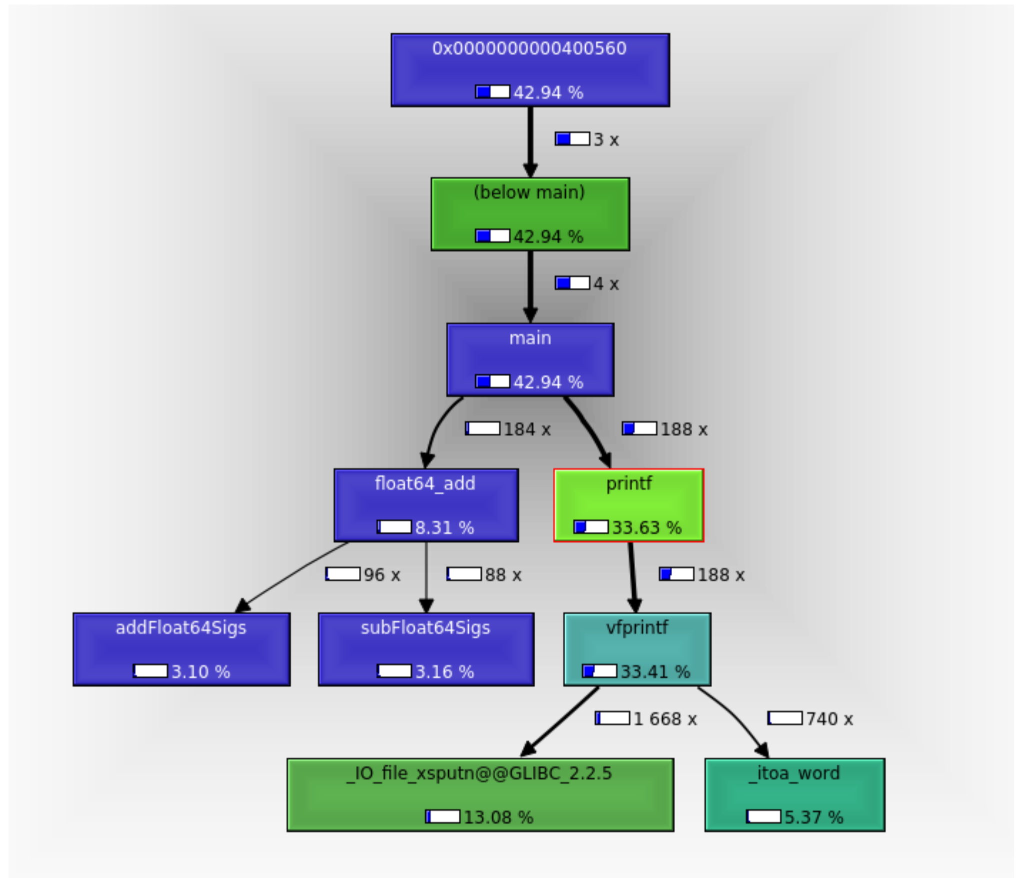

Figure 5.28: DFADD Profiling. The Call-graph depicting calling relationships and computational needs of each function.

5.7.1 Result Tables

Table 5.14 indicates the result of the Pure Hardware HLS synthesis. Respectively, Table 5.15 illustrates the DFADD HLS synthesis for pure software compilation and Table 5.16 shows the result of hybrid one.

Metrics	Original	With FW
Cycles	643	337
FMax (MHz)	134.43	133.28
Clock Period (ns)	7.439	7.503
Wall-clock Time (us)	4.78	2.53
ALMs	1326	1328
Area*Delay (us)	6342	3358
DSP Blocks	0	0
RAM Blocks	1	1
Blocks Memory Bits	8192	8192
Registers	1822	1808

Table 5.14: Pure Hardware DFADD HLS Analysis

Metrics	Original	With FW
Cycles	19685	12914
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	273.86	179.66
ALMs	4735	4735
Area*Delay (us)	1296723	850693
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.15: Pure Software DFADD HLS Analysis

Metrics	Original	With FW
Cycles	16897	14534
FMax (MHz)	68.38	75.76
Clock Period (ns)	14.62	13.20
Wall-clock Time (us)	247.10	191.84
ALMs	7264	5767
Area*Delay (us)	1794966	1106357
DSP Blocks	6	6
RAM Blocks	32	24
Blocks Memory Bits	152896	152768
Registers	8917	7040

Table 5.16: Hardware/Software DFADD HLS Analysis

5.7.2 Performance Diagrams

To evaluate the performance of the tuned Matrix Multiplication, we conducted the experiment on both the original and HOF-tuned codes. Figures 5.29, 5.30 and 5.31 illustrate the speedup, logic utilization efficiency and Area Delay, in order.

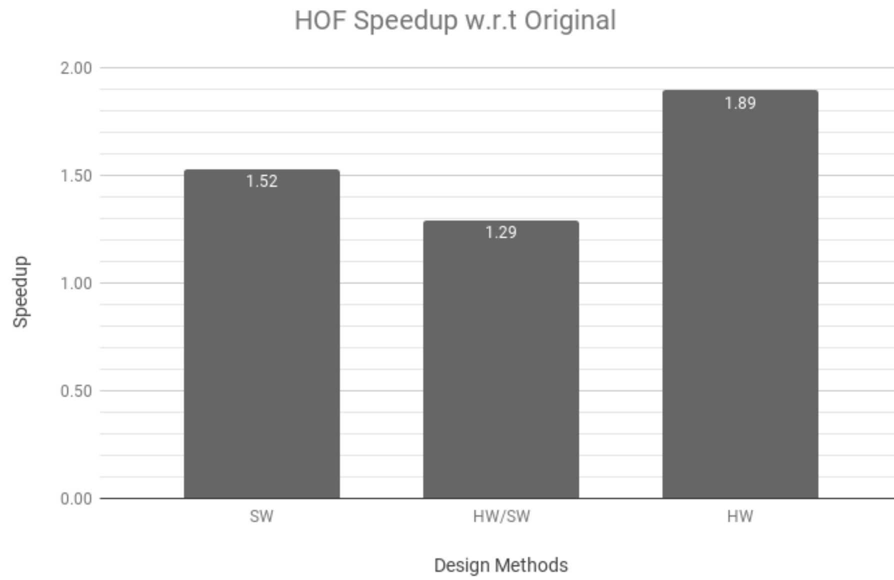

Figure 5.29: DFADD Speedup diagram

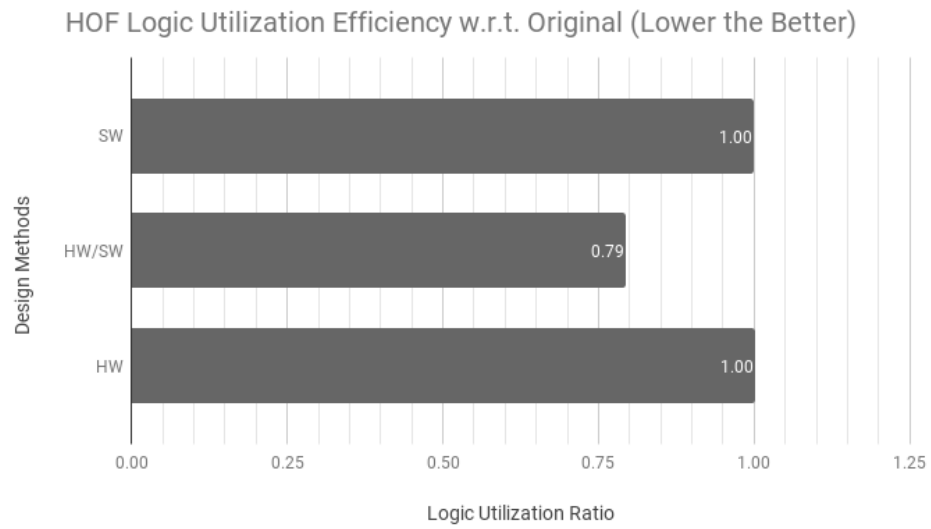

Figure 5.30: DFADD Utilization diagram

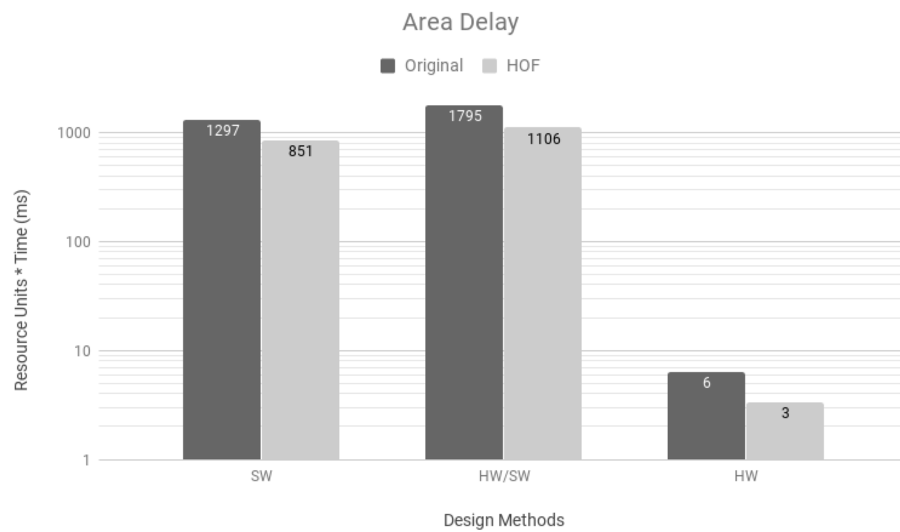

Figure 5.31: DFADD Area Delay diagram

5.8 ADPCM Benchmark

This experiment discusses the effect of HOF on ADPCM benchmark. First, the profiler performed the analysis over the original code and detected *upzero* as the hot function for this benchmark. Then, we tuned the hot function in the next step and used it for hybrid synthesis as the selected part for accelerating transformation. Figure 5.32 graphically demonstrates the result of Valgrind profiler.

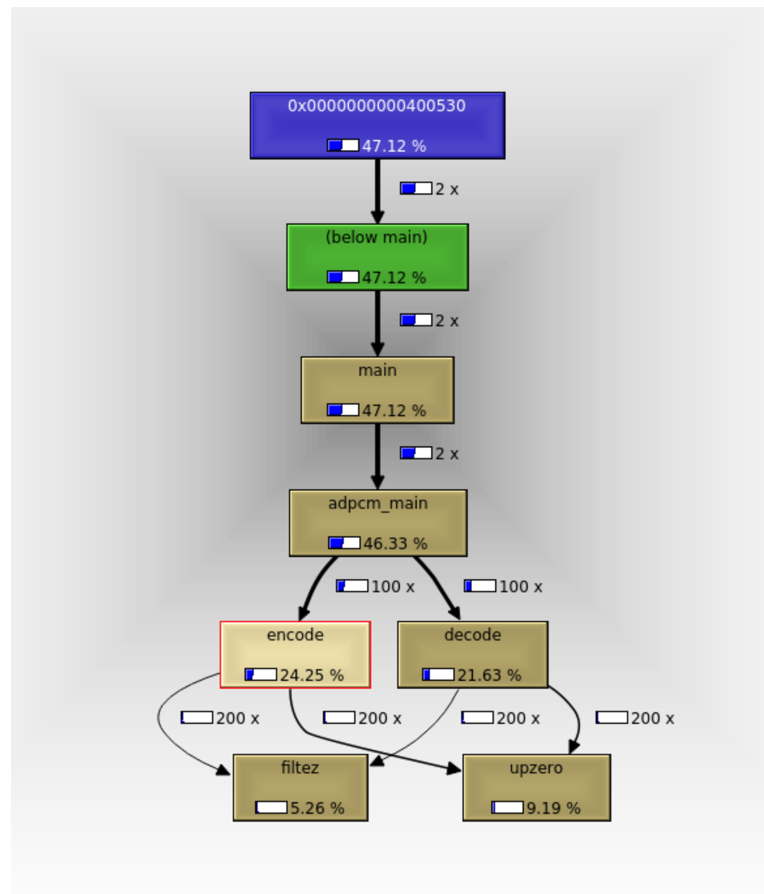

Figure 5.32: ADPCM Profiling, ADPCM Call-graph depicting calling relationships and computational needs of each function

5.8.1 Result Tables

Table 5.17 indicates the result of the Pure Hardware HLS synthesis. Respectively, Table 5.18 illustrates the ADPCM HLS synthesis for pure software compilation and Table 5.19 shows the result of hybrid one.

Metrics	Original	With FW
Cycles	13521	13421
FMax (MHz)	85.38	86.05
Clock Period (ns)	11.712	11.621
Wall-clock Time (us)	158.36	155.97
ALMs	5299	5330
Area*Delay (us)	839163	831307
DSP Blocks	87	87
RAM Blocks	7	7
Blocks Memory Bits	9152	9152
Registers	9133	9145

Table 5.17: Pure Hardware ADPCM HLS Analysis

Metrics	Original	With FW
Cycles	196607	197151
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	2735.21	2742.78
ALMs	4735	4735
Area*Delay (us)	12951226	12987062
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.18: Pure Software ADPCM HLS Analysis

5.8.2 Performance Diagrams

We tested the performance of all the synthesis methods HW, SW and HW/SW includes Speedup, Utilization Efficiency and Area Delay shown in Figure 5.33, Figure 5.34 and Figure 5.35, respectively.

Metrics	Original	With FW
Cycles	99834	99362
FMax (MHz)	68.32	71.05
Clock Period (ns)	14.64	14.07
Wall-clock Time (us)	1461.27	1398.48
ALMs	11390	11544
Area*Delay (us)	16643871	16144052
DSP Blocks	76	76
RAM Blocks	34	34
Blocks Memory Bits	153334	153334
Registers	14616	14643

Table 5.19: Hardware/Software ADPCM HLS Analysis

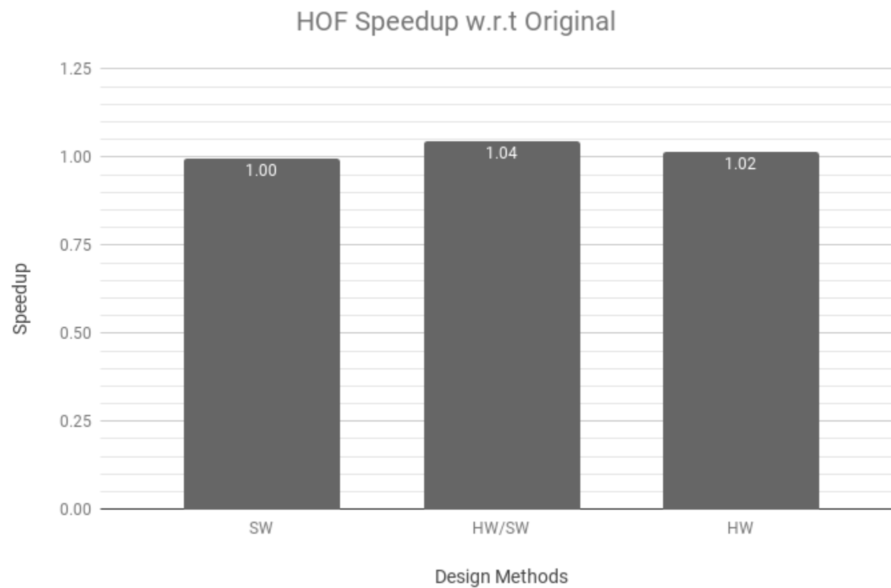

Figure 5.33: ADPCM Speedup diagram

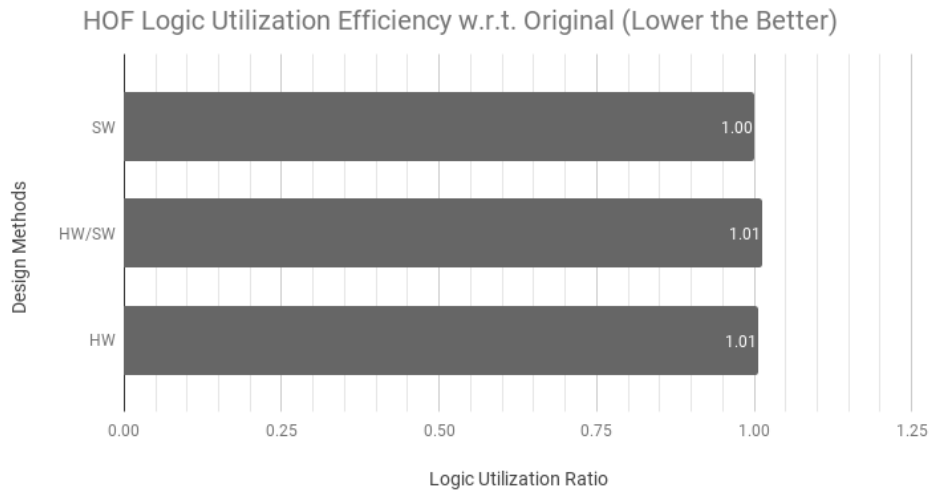

Figure 5.34: ADPCM Utilization diagram

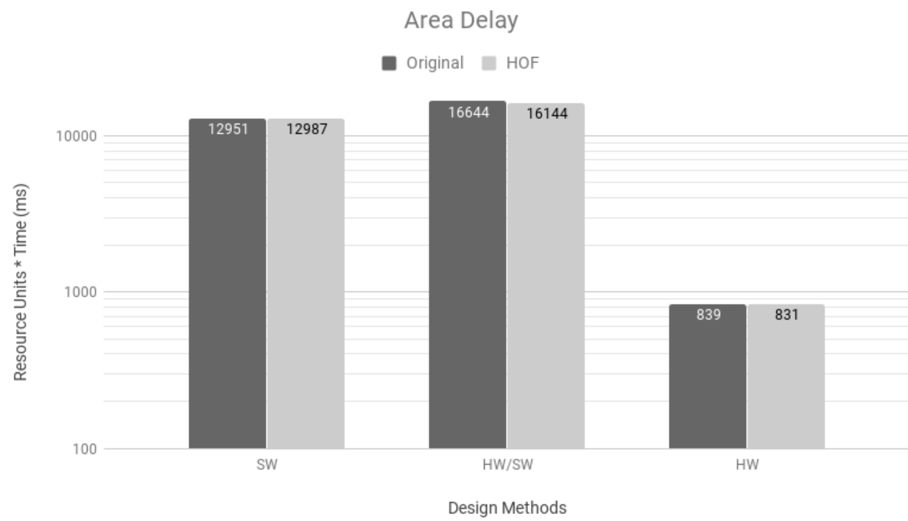

Figure 5.35: ADPCM Area Delay diagram

5.9 MIPS Benchmark

This program describes instruction-level behaviors of a simplified MIPS processor which has 30 types of instructions. A sorting program is served as test vectors. Depending on synthesis options, HLS tools may synthesize a sequential processor or a pipelined one from the program. Figure 5.36 indicates the result of the Valgrind profiler in which the main function is the hot function and most time-intensive part of the code.

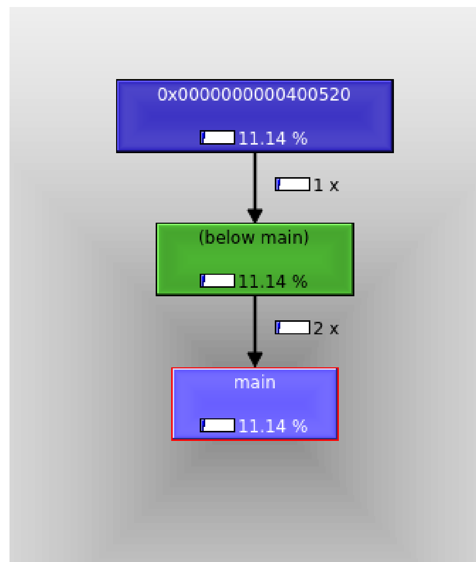

Figure 5.36: MIPS Profiling, the Call-graph depicting calling relationships and computational needs of each function

5.9.1 Result Tables

Respectively, tables 5.20 and 5.21 indicate the results of variant HLS synthesis methods Pure Hardware and Pure Software.

Metrics	Original	With FW
Cycles	5121	5060
FMax (MHz)	84.78	86.18
Clock Period (ns)	11.795	11.604
Wall-clock Time (us)	60.40	58.71
ALMs	1102	1065
Area*Delay (us)	66565	62531
DSP Blocks	6	6
RAM Blocks	3	4
Blocks Memory Bits	3072	3072
Registers	1136	995

Table 5.20: Pure Hardware MIPS HLS Analysis

Metrics	Original	With FW
Cycles	50341	49898
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	700.35	694.18
ALMs	4735	4735
Area*Delay (us)	3316147	3286965
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.21: Pure Software MIPS HLS Analysis

5.9.2 Performance Diagrams

On MIPS architecture, the performance gap between the hardware and the software synthesis is now larger than in the other experiment due to MIPS division operation constraints. Figures 5.37, 5.38 and 5.39 indicate the speedup, utilization efficiency and Area Delay, respectively.

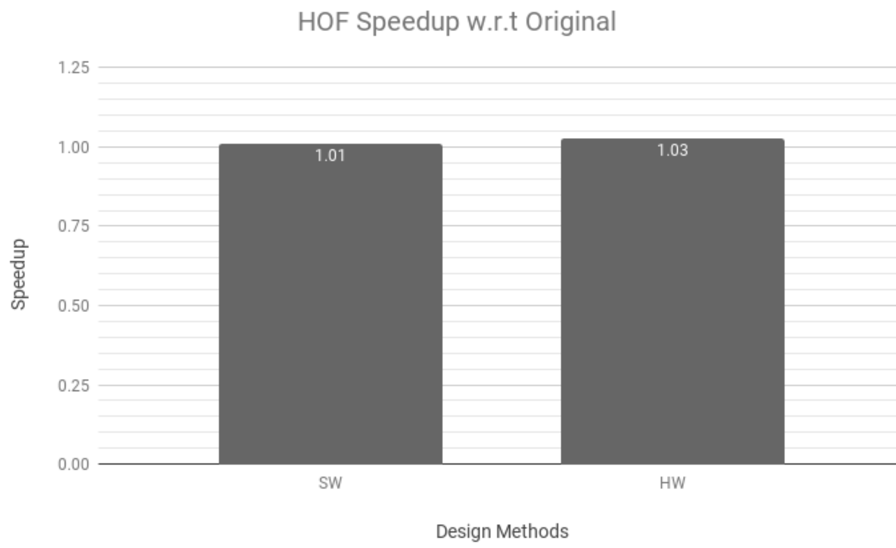

Figure 5.37: MIPS Speedup diagram

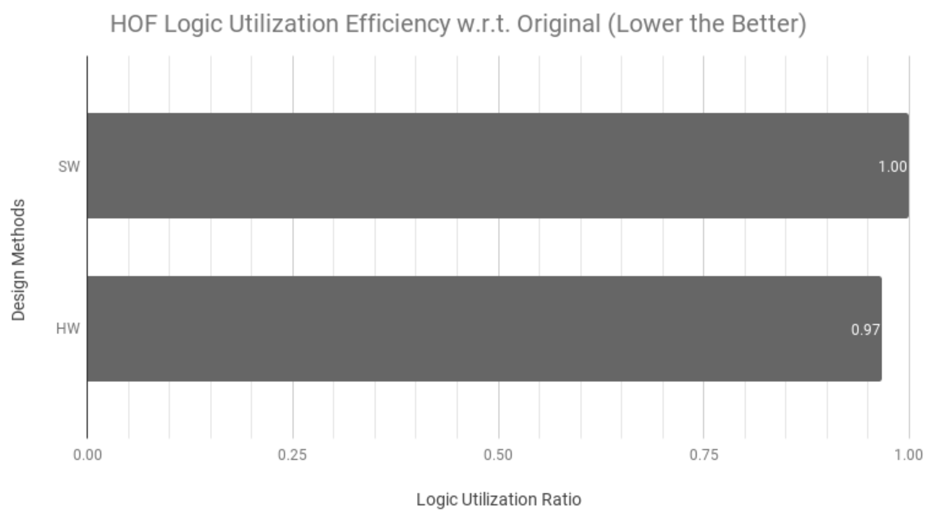

Figure 5.38: MIPS Utilization diagram

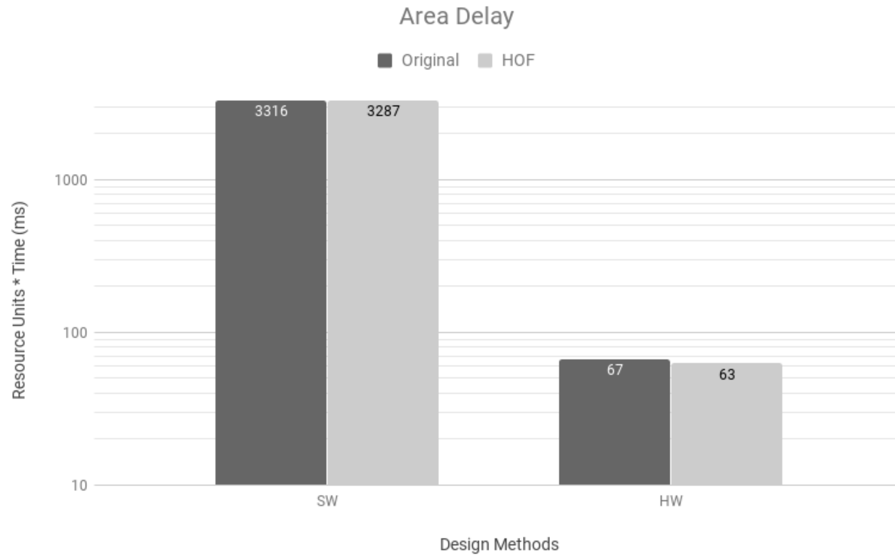

Figure 5.39: MIPS Area Delay diagram

5.10 DFDIV Benchmark

This experiment discusses the effect of HOF on DFDIV benchmark. First, the profiler performed the analysis over the original code and detected *float64div* as the hot function for this benchmark. Then, we tuned the hot function in the next step and used it for hybrid synthesis as the selected part for accelerating transformation. Figure 5.32 graphically demonstrates the result of Valgrind profiler.

5.10.1 Result Tables

Table 5.22 indicates the result of the Pure Hardware HLS synthesis. Respectively, Table 5.23 illustrates the DFDIV HLS synthesis for pure software compilation and Table 5.24 shows the result of hybrid one.

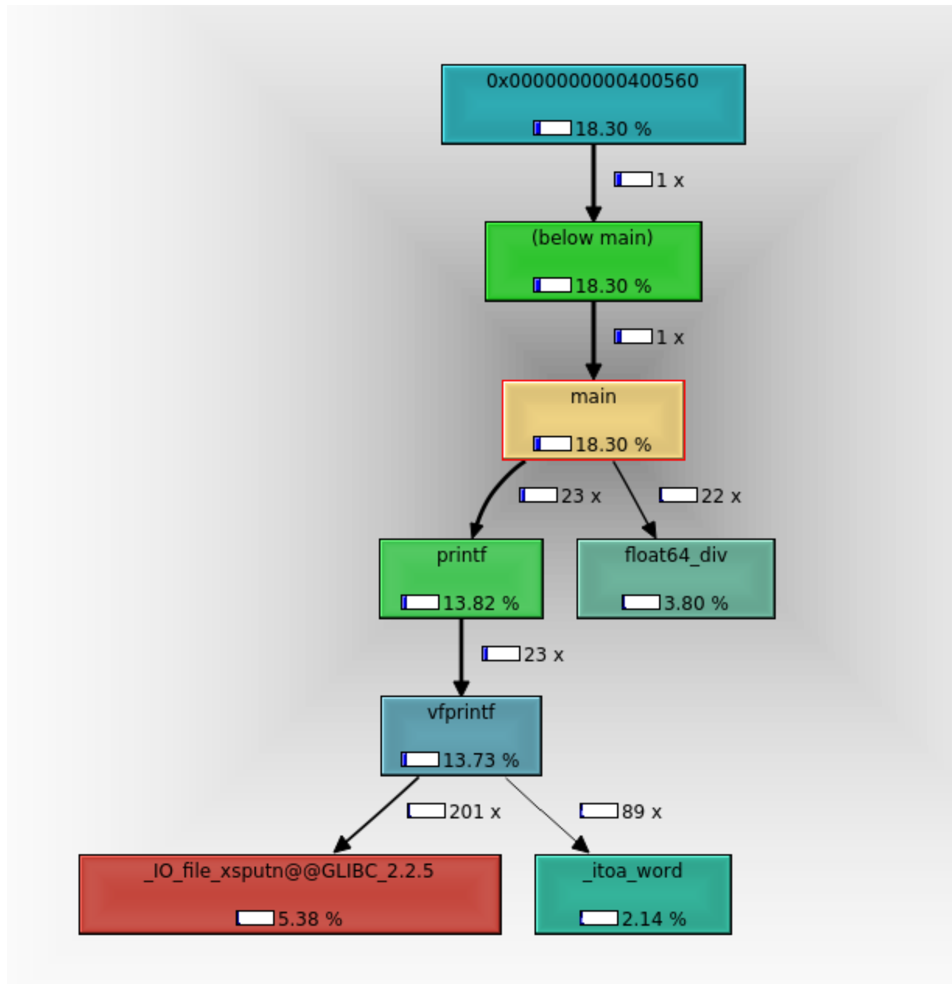

Figure 5.40: DFDIV Profiling, The Call-graph depicting calling relationships and computational needs of each function

Metrics	Original	With FW
Cycles	1884	1846
FMax (MHz)	104.13	102.88
Clock Period (ns)	9.603	9.720
Wall-clock Time (us)	18.09	17.94
ALMs	3143	3422
Area*Delay (us)	56866	61402
DSP Blocks	48	48
RAM Blocks	2	2
Blocks Memory Bits	1535	1535
Registers	6698	7793

Table 5.22: Pure Hardware DFDIV HLS Analysis

Metrics	Original	With FW
Cycles	93988	93210
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	1307.57	1296.74
ALMs	4735	4735
Area*Delay (us)	6191335	6140086
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.23: Pure Software DFDIV HLS Analysis

Metrics	Original	With FW
Cycles	9730	11804
FMax (MHz)	61.5	64.99
Clock Period (ns)	16.26	15.39
Wall-clock Time (us)	158.21	181.63
ALMs	10939	10944
Area*Delay (us)	1730674	1987736
DSP Blocks	70	70
RAM Blocks	73	73
Blocks Memory Bits	171009	171009
Registers	16690	16730

Table 5.24: Hardware/Software DFDIV HLS Analysis

5.10.2 Performance Diagrams

We observed that loop unrolling for DFDIV can result in worse performance than original code because the number of division operations can be higher than MIPS capacity. We tested the performance of all the synthesis methods HW, SW and HW/SW includes Speedup, Utilization Efficiency and Area Delay shown in Figure 5.41, Figure 5.42 and Figure 5.43, respectively.

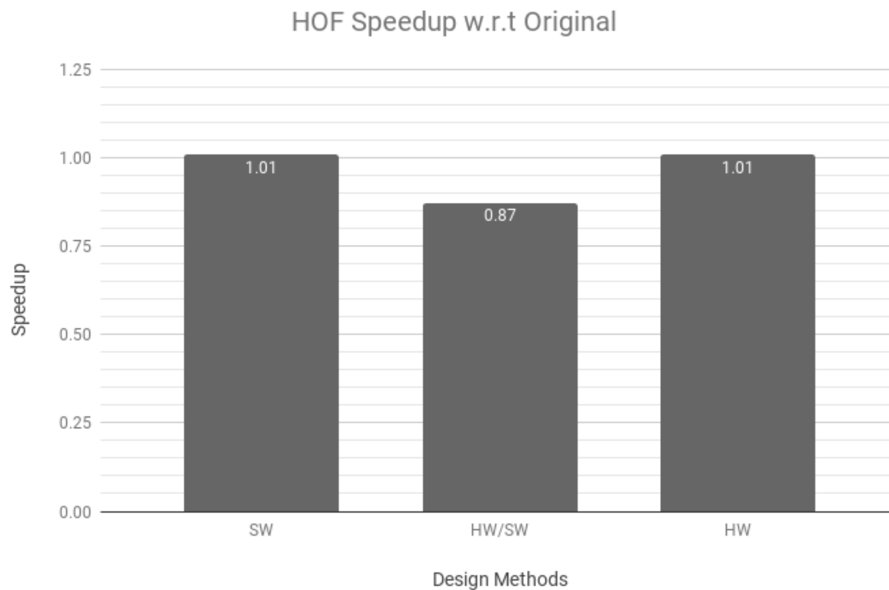

Figure 5.41: DFDIV Speedup diagram

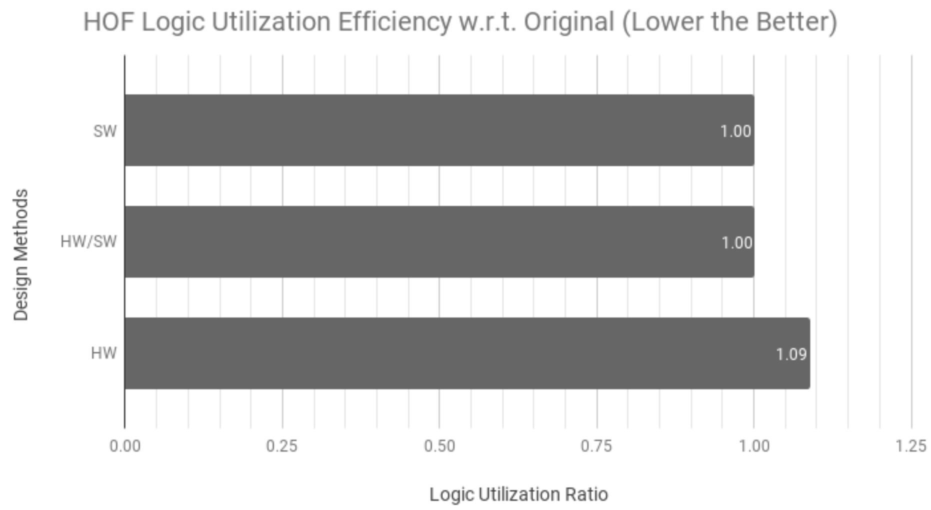

Figure 5.42: DFDIV Utilization diagram

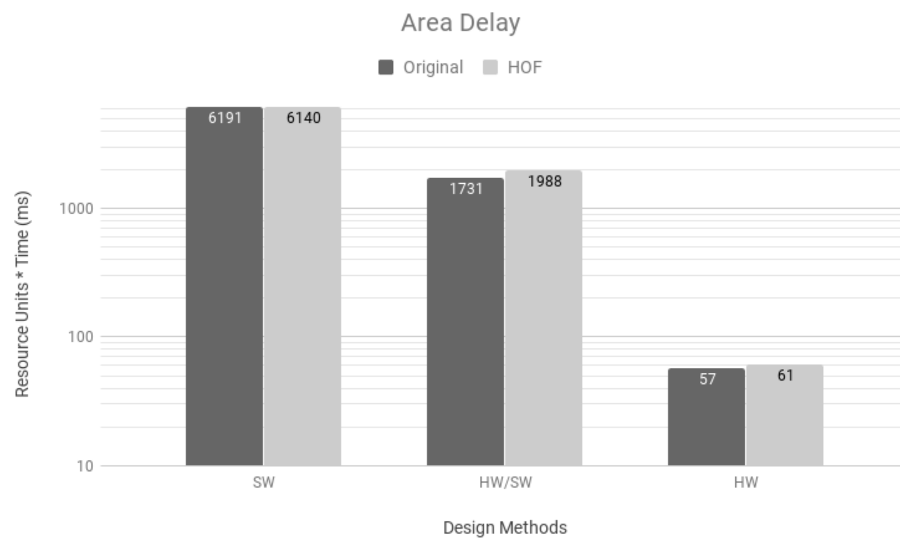

Figure 5.43: DFDIV Area Delay diagram

5.11 DFSIN Benchmark

In this section we examine the effectiveness of HOF in optimizing DFSIN benchmark. Figure 5.44 illustrates the output of Valgrind profiler indicating the possible hot functions *roundAndPackFloat64* and *extractFloat64EXP* in this benchmark. By considering the number of calls of each candidate, we have chosen the latter one as the selected function.

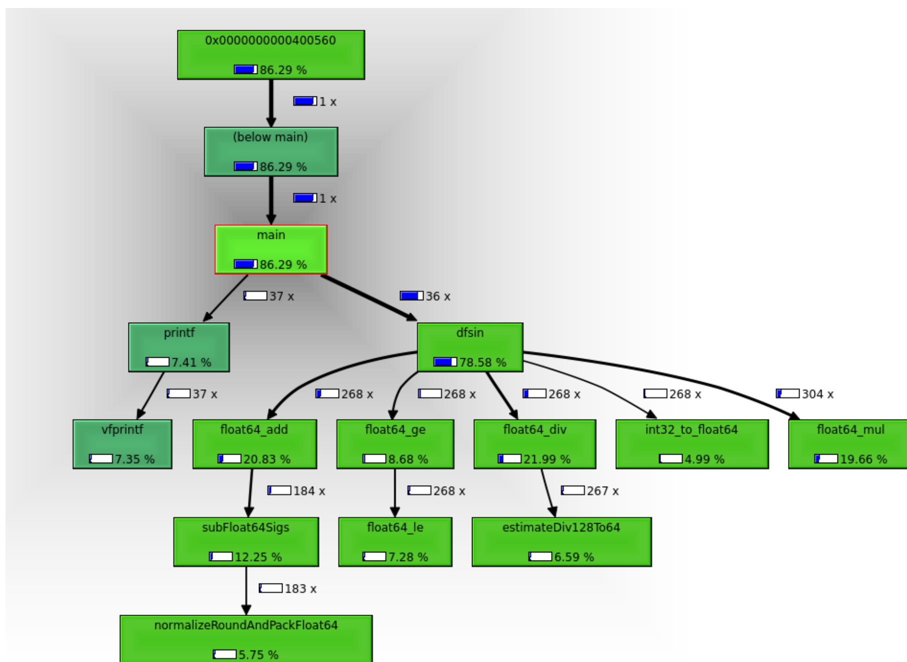

Figure 5.44: DFSIN Profiling. The call-graph depicting calling relationships and computational needs of each function

5.11.1 Result Tables

Table 5.25 indicates the result of the Pure Hardware HLS synthesis. Respectively, Table 5.26 illustrates the DFSIN HLS synthesis for pure software compilation.

Metrics	Original	With FW
Cycles	56739	56811
FMax (MHz)	95.7	97.05
Clock Period (ns)	10.449	10.304
Wall-clock Time (us)	592.88	585.38
ALMs	8392	9556
Area*Delay (us)	4975483	5593879
DSP Blocks	82	82
RAM Blocks	5	5
Blocks Memory Bits	9377	9377
Registers	13850	16762

Table 5.25: Pure Hardware DFSIN HLS Analysis

Metrics	Original	With FW
Cycles	4049788	3799757
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	56340.96	52862.51
ALMs	4735	4735
Area*Delay (us)	266774432	250303970
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.26: Pure Software DFSIN HLS Analysis

5.11.2 Performance Diagrams

This section discusses the performance evaluation of the DFMUL benchmark. We compare the performance of the code tuned by HOF with the original code.

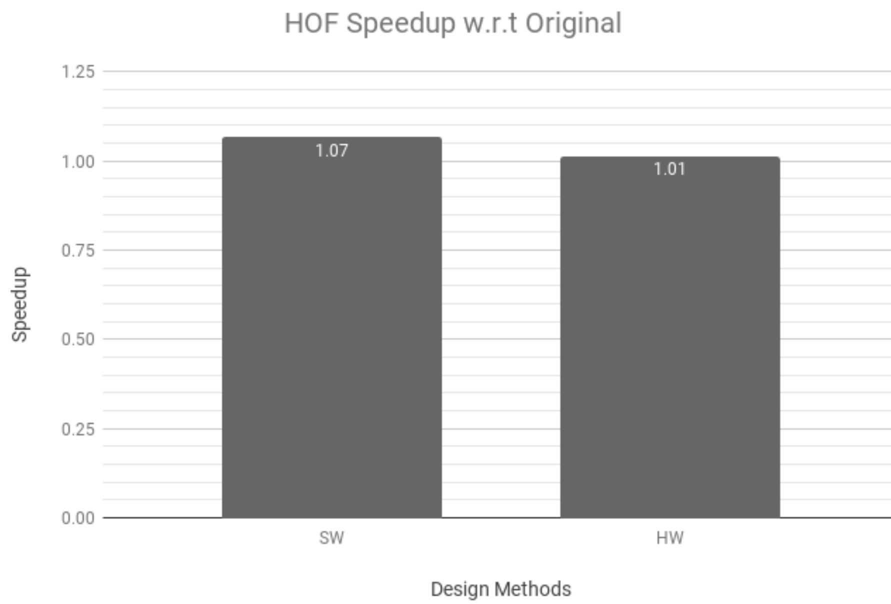

Figure 5.45: DFSIN Speedup diagram

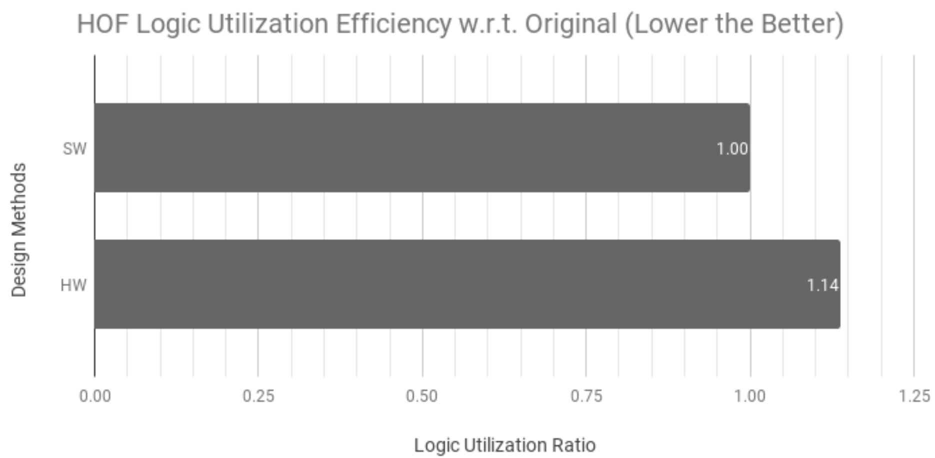

Figure 5.46: DFSIN Utilization diagram

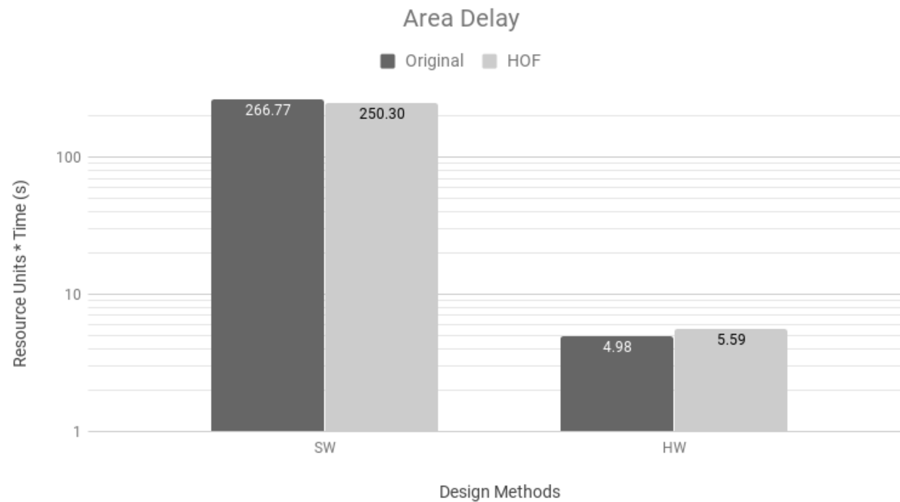

Figure 5.47: DFSIN Area Delay diagram

5.12 JPEG Benchmark

In this experiment, we tuned the performance of the JPEG benchmark as the most complex application among all other benchmarks in this thesis with the highest execution time in general. Figure 5.48 illustrates the output of Valgrind profiler indicating the possible hot functions *YuvToRgb* and *buf_getb* in this benchmark. By considering the number of calls of each candidate, we have chosen the latter one as the selected function.

5.12.1 Result Tables

Similarly to BWMM results, the HOF-optimized code is more efficient than the original code when the input applications are more complex. For large problem sizes, HOF enhances further with its toolchain optimizations, resulting in performance consistently and significantly higher than the original code, up to almost 8 times faster and more utilization efficiency for JPEG. Table 5.27 indicates the result of the Pure Hardware HLS synthesis. Respectively, Table 5.28 illustrates the JPEG HLS synthesis for pure software compilation.

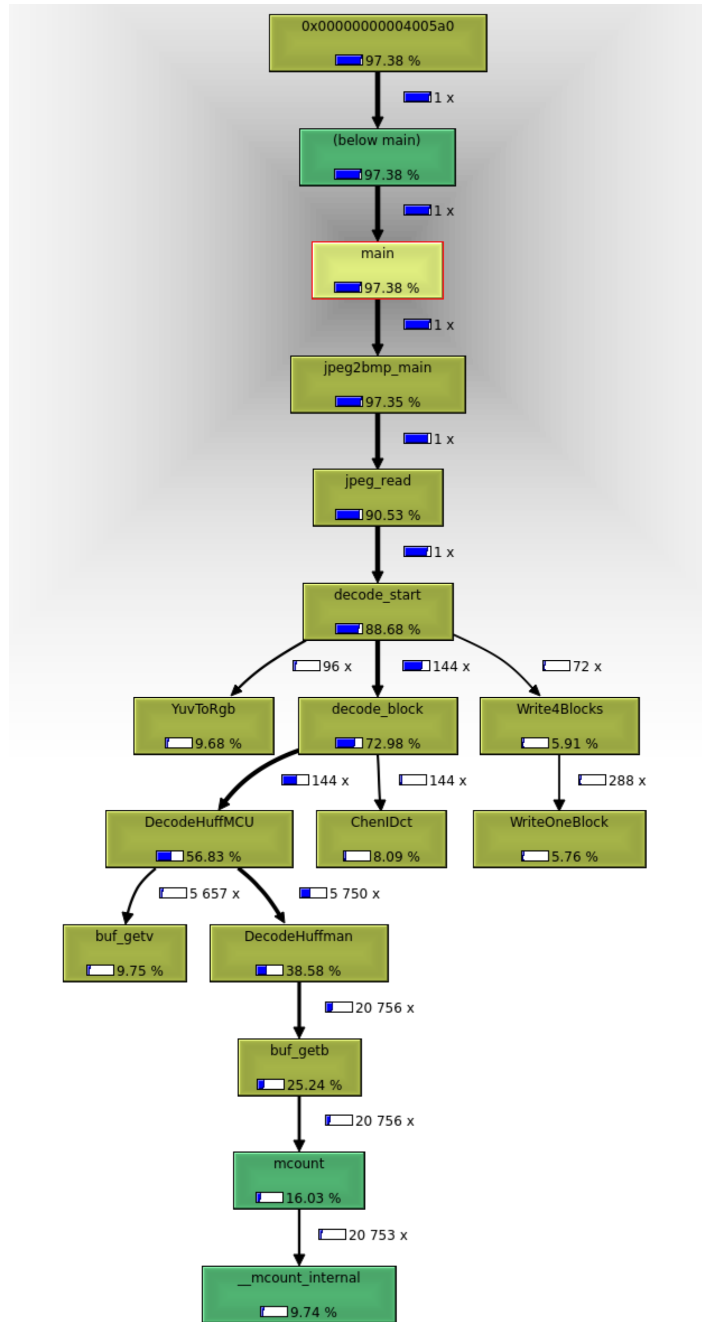

Figure 5.48: JPEG Profiling. The Call-graph depicting calling relationships and computational needs of each function

Metrics	Original	With FW
Cycles	1301530	1296217
FMax (MHz)	9.15	71.62
Clock Period (ns)	109.290	13.963
Wall-clock Time (us)	142243.72	18098.53
ALMs	16665	16616
Area*Delay (us)	2370491525	300725240
DSP Blocks	87	87
RAM Blocks	83	83
Blocks Memory Bits	461176	461176
Registers	20233	20404

Table 5.27: Pure Hardware JPEG HLS Analysis

Metrics	Original	With FW
Cycles	5187081	5146408
FMax (MHz)	71.88	71.88
Clock Period (ns)	13.91	13.91
Wall-clock Time (us)	72163.06	71597.22
ALMs	4735	4735
Area*Delay (us)	341692105	339012825
DSP Blocks	6	6
RAM Blocks	20	20
Blocks Memory Bits	152704	152704
Registers	5556	5556

Table 5.28: Pure Software JPEG HLS Analysis

5.12.2 Performance Diagrams

We tested the performance of all the synthesis methods HW, SW and HW/SW includes Speedup, Utilization Efficiency and Area Delay shown in Figure 5.49, Figure 5.50 and Figure 5.51, respectively.

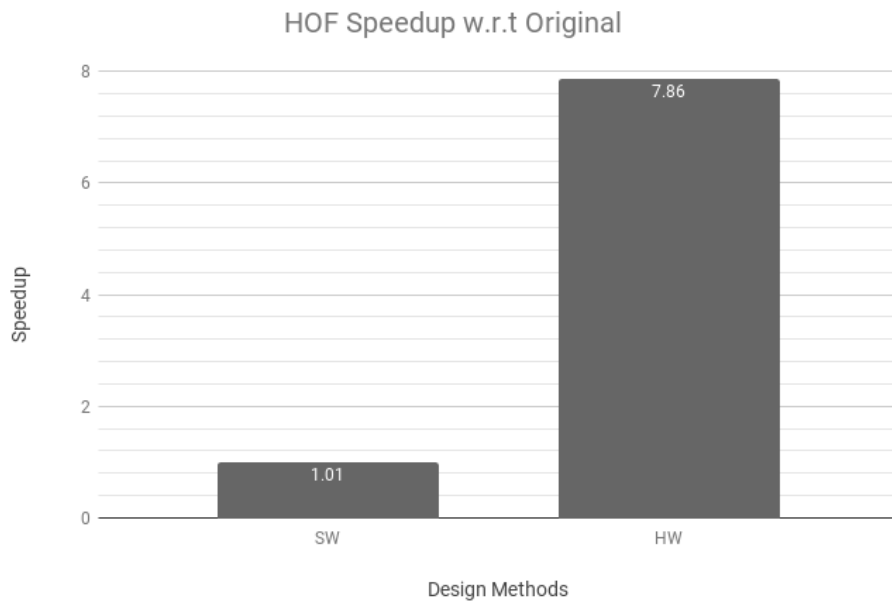

Figure 5.49: JPEG Speedup diagram

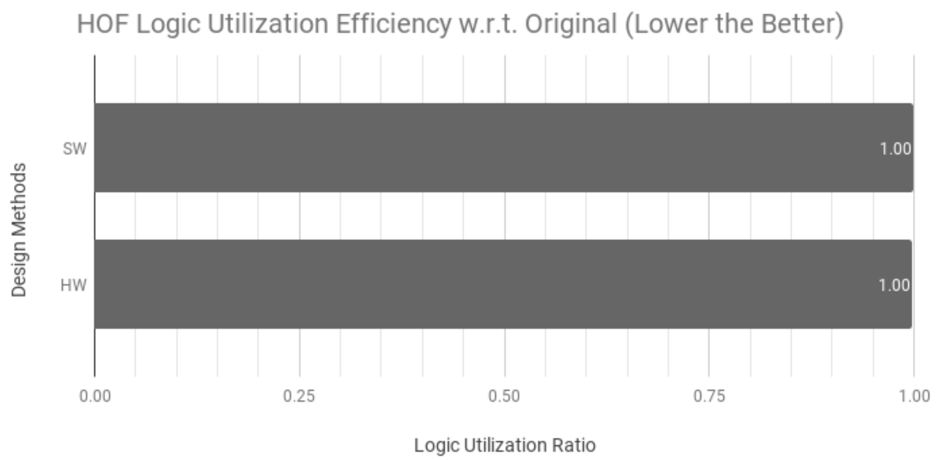

Figure 5.50: JPEG Utilization diagram

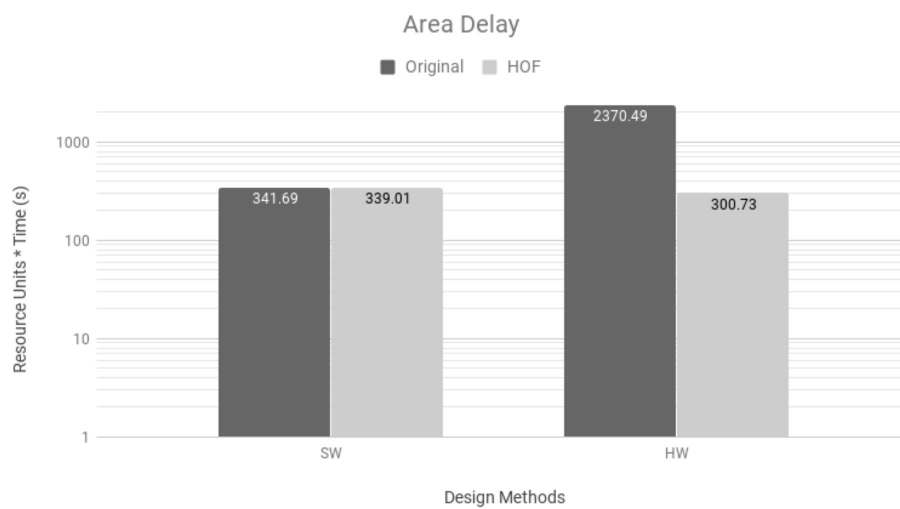

Figure 5.51: JPEG Area Delay diagram

Chapter 6

Conclusion and Future Works

6.1 Conclusion

In this work, we explored the effect of code autotuning, code transformation, application autotuning, compiler optimization, HLS directives and HLS optimizations n different design method from software to parallel processing systems. We showed that our framework outperforms on almost all design performance measurements and keeps the critical performance index called, Area-Delay-Product lower than baseline. We experimentally showed the benefits of our framework by optimizing CHStone benchmark on different design methods. The results show that not only code transformations such as loop unrolling benefits the software performance but also it impacts positively on HLS and subsequent circuits.

We indicate that applying a combination of application autotuning and optimization levels by our Hybrid Optimization Framework, for all the benchmarks in which we conducted the experiments on both the original and HOF-tuned codes, revealed performance improvement and resource usage efficiency. The results indicate that whereas LegUp is unable to use LLVM vectorization, HOF successfully optimizes the codes and enhanced the execution time and resource usage. This is also reflected in the best S2S transformations found by HOF, where loop unrolling always turn out to be better than other optimizations. The HOF-optimized code performs better than the original version when the number of computations per iteration increases and the number of branches decreases. The speedup result obtained from tuning the original code with HOF is significant.

Application optimizations and source-to-source transformation by HOF improved the execution time from 2% to 700% which leads up to a 7× speedup factor w.r.t baseline as shown in Figure 6.1 in which we normalized to one all the execution times. Moreover, we observe that because of MIPS

architecture limitations, the speedup of HOF w.r.t original code slightly decreases when the number of division operation used is increasing. The Figure 6.1 shows that semi-automated optimization HOF tuning produces the best performance for all cases.

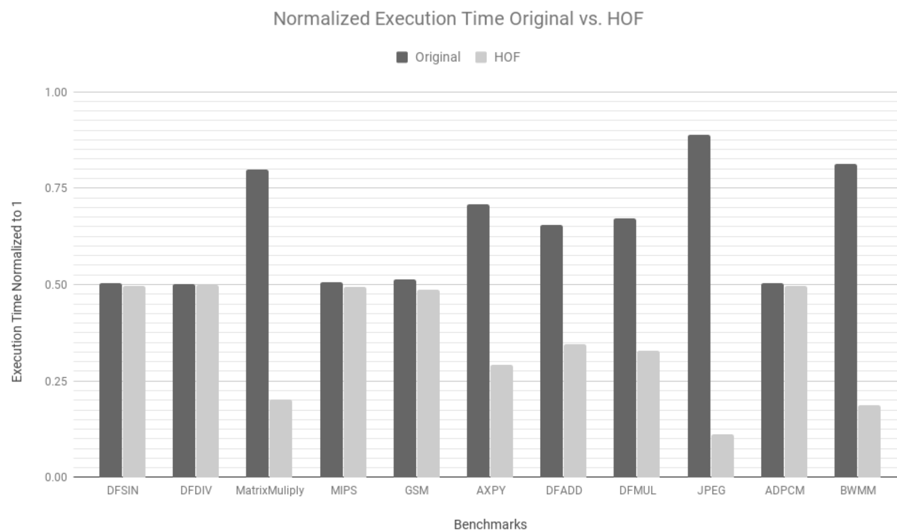

Figure 6.1: Normalized Execution Time of All Benchmarks w.r.t Baseline

Additionally, we observed that LegUp free version fails to pipeline the code and run the HW/SW synthesis, whereas HOF is able to profile the hot function of original code and improves the performance for other hardware and software flows. The sequential performance results are shown in Figure 6.2 for all used benchmarks in this experiment contains the Area-Delay result, which shows that applying HOF on the original code always delivers logic utilization efficiency boosts, which range from 3 percent to 700 percent.

In certain cases, HOF did not have a significant performance improvement; nevertheless, the HOF version noticed to perform better than the original implementations in all instances. Also, the positive Logic Utilization Efficiency was improved even for the cases that a significant execution time improvement was not noticed. Thus, we optimized the two important criterion meaning that having less execution time by using less computational units.

6.2 Future Works

Throughout developing the framework, we also worked on LLVM IR-level optimizations. Although Orio reports the best compiler optimization flags for the SW platform, we cannot always make use of them because:

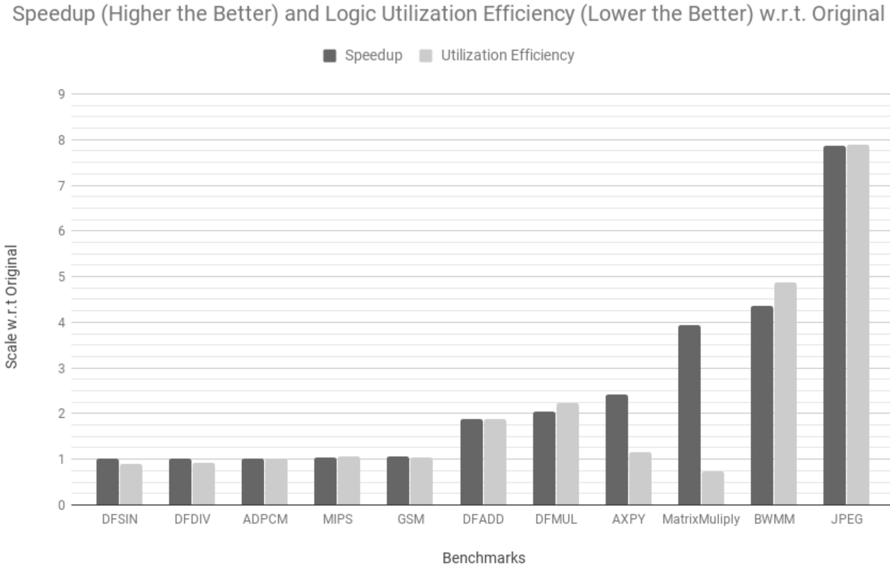

Figure 6.2: Speedup of all benchmarks besides their corresponding logic utilization efficiency w.r.t original

- Orio is integrated with GCC and still not fully adapted to exploit clang, taking into account that compiler optimization parameters which are wrapped in *-O1 -O2 -O3* are different in GCC and Clang-LLVM, therefore integrating *-O* flags reported by Orio are not always reliable

- The hybrid essence of our framework does not allow us to be always versatile on compiler optimization flags. in the current version of the framework, after many experiments, we realized that -O3 can outperform on different design methods.

Moreover as already investigated by other researchers in University of Toronto [49], there are customized sets of LLVM flags which outperform on pure hardware design method, meaning that compiling HLS by some particular flags rather than conventional -O optimizations can achieve a better result on HLS-based implementations.

Also, the order that those flags are applied on IRs can significantly change the performance of final synthesized circuits. Our early experiments on several codes show that there is the possibility to optimize them on different design methods by applying custom compiler flags. Evidently, the best order to apply the optimizations is not unique and there is no a single recipe for all codes but the customized recipe of flags can impact positively on some performance metrics, this information helps to prune the vast design space of our framework regarding HLS on pure hardware method. For

the sake of completeness, three customized LLVM-Clang recipe, which we believe can upgrade our framework are appended in appendix C. Also, the complete list of flags is available at:

`http://janders.eecg.toronto.edu/pdfs/fccm13/recipes_summary.htm`.

Regarding the HLS design space, considering the recent progress in Artificial Neural Network and the concept of Deep Neural Network and its applicability using Google *Tensorflow* platform, we believe that fully automated approaches to selecting compiler optimizations on a per-program basis are practical, and will be of keen interest to FPGA users seeking high design performance. such approaches also appear to be a useful mechanism for narrowing the gap between HLS-generated hardware and manually designed hardware.

Last but not least, there are many different code transformations that Orio provides. In this research, we managed to utilize some, mostly loop unrolling. We wrote the tuning specifications in a way to guide Orio directive generation algorithm to vectorizing for different segments of a code. However, the free version of LegUp we used for this research was unable to apply vectorization on compilation stage. Meaning that, by developing this feature inside the tool-chain, the HOF is expected to achieve better results in the future.

Bibliography

[1] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195, May 2008.

[2] Juan J. Durillo Philipp Gschwandtner and Thomas Fahringer. Multi-objective auto-tuning with insieme: Optimization and trade-off analysis for time, energy and resource usage. *Euro-Par 2014: Parallel Processing*, 8632:87–98, 2014.

[3] Chung I. Hollingsworth J. Tapus, C. Active harmony: Towards automated performance tuning. *IEEE Conference on Supercomputing*, 2002.

[4] de Supinski B.R. Schulz M. et al. Li, D. Strategies for energy-efficient resource management of hybrid programming models. *Parallel and Distributed Systems, IEEE Transactions*, 24(1):144–157, 2013.

[5] Laurenzano M. Carrington L. et al. Tiwari, A. Auto-tuning for energy usage in scientific applications. *Euro-Par 2011: Parallel Processing Workshops*, 2012.

[6] Eeckhout L.: Cole Hoste, K. compiler optimization level exploration. *Proc. of the 6th Intl. Symposium on Code generation and optimization*, 2008.

[7] Guo J. Bhat A. et al. Rahman, S. Studying the impact of application-level optimizations on the power consumption of multi-core architectures. *Proc. of the 9th conference on Computing Frontiers*, 2012.

[8] Chen J. Yang X. etal. Dong, Y. Energy-oriented openmp parallel loop scheduling. *Parallel and Distributed Processing with Applications, ISPA'08. International Symposium on*, 2008.

[9] Mandelbrot set (http://mathworld.wolfram.com/mandelbrotset.html).

[10] Boyana Norris Albert Hartono and P. Sadayappan. Annotation-based empirical performance tuning using orio. *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, 2009.

[11] J. Ramanujam U. Bondhugula, A. Hartono and P. Sadayappan. A practical automatic polyhedral program optimization system. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2008.

[12] Kalyan Veeramachaneni Jonathan Ragan-Kelley Jeffrey Bosboom Una-May O'Reilly Jason Ansel, Shoaib Kamil and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, August 2014.

[13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[14] Llvm compiler project (http://www.llvm.org), 2010.

[15] C.W. Chin J. Bilmes, K. Asanovi'c and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. *the International Conference on Supercomputing, ACM SIGARC, Vienna, Austria*, July 1997.

[16] A. Petitet R. C. Whaley and J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

[17] Matteo Frigo and Steven G. Johnson. Fftw: An adaptive software architecture for the fft. *In Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, 3:1381–1384, 1998.

[18] J. Demmel R. Vuduc and K. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *In Proc. of SciDAC 2005, J. of Physics: Conference Series*, June 2005.

[19] Eric Allen Brewer. Portable high-performance supercomputing: high-level platform-dependent optimization. *PhD thesis, Massachusetts Institute of Technology*, 1994.

[20] T.C. Meyerowitz. Single and multi-cpu performance modeling for embedded systems. *PhD thesis, University of California, Berkeley, Berkeley, CA, USA*, April 2008.

[21] K. Datta S. Williams-J. Shalf L. Oliker S. Kamil, C. Chan and K. Yelick. In-place auto-tuning of structured grid kernels. *http://www.cs.berkeley.edu/ skamil/stencilautotunerposter.ppt*, December 2008.

[22] Cristina Silvano Gianluca Palermo and Vittorio Zaccaria. Respir: A response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829, Dec. 2009.

[23] Vittorio Zaccaria Cristina Silvano Giovanni Mariani, Gianluca Palermo. Desperate++: An enhanced design space exploration framework using predictive simulation scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(2):293–306, February 2015.

[24] Vittorio Zaccaria Cristina Silvano Giovanni Mariani, Gianluca Palermo. Oscar: an optimization methodology exploiting spatial correlation in multi-core design space. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):740–753, May 2012.

[25] I.-H. Chung C. Tapus and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. *in In Proceedings from the Conference on High Performance Networking and Computing*, 2003.

[26] Top 500 supercomputer sites (http://www.top500.org/), 2010.

[27] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *IEEE*, 93(2), February 2005.

[28] J. J. Durillo S. Pellegrini P. Gschwandtner T. Fahringer H. Jordan, P. Thoman and H. Moritsch. A multi-objective auto-tuning framework for parallel codes. *in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC '12*, 2012.

[29] J. W. Demmel R. Vuduc and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *in Scientific Discovery through Advanced Computing Conference, San Francisco, CA*, June 2005.

[30] O. Schenk M. Christen and H. Burkhart. A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. *in IPDPS. IEEE*, 2011.

[31] S. Amarasinghe J. Ansel, M. Pacula and U.-M. O'Reilly. An efficient evolutionary algorithm for solving bottom up problems. *in Annual Conference on Genetic and Evolutionary Computation, Dublin, Ireland*, July 2011.

[32] S. A. Kamil. Productive high performance parallel programming with auto-tuned domain-specific embedded languages. *Ph.D. dissertation, EECS Department, University of California, Berkeley*, Jan 2013.

[33] B. Singer J. Xiong J. R. Johnson D. A. Padua M. M. Veloso M. Puschel, J. M. F. Moura and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *IJHPCA*, 18(1), 2004.

[34] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):21, 2016.

[35] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):29, 2017.

[36] Amir Hossein Ashouri. *Compiler Autotuning Using Machine Learning Techniques*. PhD thesis, Politecnico di Milano, Italy, 2016. http://hdl.handle.net/10589/129561.

[37] A. Grosul T. J. Harvey S. W. Reeves D. Subramanian L. Torczon L. Almagor, K. D. Cooper and T. Waterman. Finding effective compilation sequences. *in LCTES'04*, pages 231–239, 2004.

[38] O. Temam M. Namolaru E. Yom-Tov A. Zaks B. Mendelson E. Bonilla J. Thomson H. Leather C. Williams M. O'Boyle P. Barnard E. Ashton E. Courtois G. Fursin, C. Miranda and F. Bodin. Milepost gcc: machine learning based research compiler. *in Proceedings of the GCC Developers' Summit*, Jul 2008.

[39] J. Cavazos A. Cohen E. Park, L.-N. Pouche and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *in CGO'11*, pages 119–129.

[40] J. Cavazos B. Franke G. Fursin-M. F. P. O'boyle J. Thomson M. Toussaint F. Agakov, E. Bonilla and C. K. I. Williams. Using machine learning to focus iterative optimization. *in CGO'06*, pages 295–305, 2006.

[41] M. Carbin S. Misailovic A. Agarwal H. Hoffmann, S. Sidiroglou and M. Rinard. Power-aware computing with dynamic knobs. *in ASPLOS*, 2011.

[42] S. Sidiroglou A. Agarwal H. Hoffmann, S. Misailovic and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. *Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2209-042*, Sep 2009.

[43] J. Sztipanovits G. Peceli G. Simon G. Karsai, A. Ledeczi and T. Kovacshazy. An approach to self-adaptive software based on supervisory control. *in International Workshop in Self-adaptive software*, 2001.

[44] . Hua Liu M. Khandekar N. Kandasamy V. Bhat, M. Parashar and S. Abdelwahed. Enabling self-managing applications using model based online control strategies. *in International Conference on Autonomic Computing, Washington, DC*, 2006.

[45] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4, March 2001.

[46] W. Baek and T. Chilimbi. Green: A framework for supporting energy conscious programming using controlled approximation. *in PLDI*, June 2010.

[47] M. D. Santambrogio J. E. Miller H. Hoffmann, J. Eastep and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. *in ICAC, New York, NY*, 2010.

[48] Zaccaria Cristina Silvano Sotirios Xydis, Gianluca Palermo. Spirit: Spectral-aware pareto iterative refinement optimization for supervised high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(1):155–159, January 2015.

[49] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 8(3):14:1–14:26, May 2015.

[50] Altera(intel) fpgas (http://www.altera.com), 2017.

[51] Xilinx vivado (http://www.xilinx.com), 2017.

[52] E. Martin, O. Sentieys, H. Dubois, and J. L. Philippe. Gaut: An architectural synthesis tool for dedicated signal processors. In *Proceedings of EURO-DAC 93 and EURO-VHDL 93- European Design Automation Conference*, pages 14–19, Sep 1993.

[53] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through

polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 9–18, New York, NY, USA, 2013. ACM.

Appendix A

Mandelbrot

""The Mandelbrot Set is the most complex object in mathematics, its admirers like to say. An eternity would not be enough time to see it all, its disks studded with prickly thorns, its spirals and filaments curling outward and around, bearing bulbous molecules that hang, infinitely variegated, like grapes on God's personal vine.""

[James Gleick, "Chaos: Making a New Science"]

Mandelbrot set is a set of complex numbers defined in the following way:

$$\begin{aligned} M = \{c \in \mathbb{C} | \lim_{x \to \infty} Z_n \neq \infty\} \\ Z_0 = c \\ Z_{n+1} = Z_n^2 + c \end{aligned} \tag{A.1}$$

That is, the Mandelbrot set is the set of all complex numbers which fulfill the condition described above, that is, if the value of the (recursive) function Zn for the value c is not infinite when n approaches infinity, then c belongs to the set.

As with many other fractal functions, it is said that this function has an attractor, which in this case is located at infinity.

Attractors are related to the "orbit" of the function. This orbit is defined by the path formed by the values of Z at each step n. The orbit of Z for a certain value c either tends towards the attractor or not. In this type of fractals a value c causing the orbit of Z to go to the attractor point is considered to be outside the set.

The attractor of a fractal may not always be at infinity, and there even may not be just one attractor. For example the so-called magnet1 fractal has two attractors, one at infinity and the other at 1+0i.

Metrics	Pure HW[1]	Pipelining	Mul lat=0	Loop Trs[2]	Parallel
Cycles	759939	559235	354435	237710	108608
II[3]	N/A	2	1	1/1/1	2
FMax	111.77MHz	116 .35MHz	64.16MHz	83.73MHz	109.4MHz
Clock Period	8.9ns	8.5ns	15ns	11ns	9.1ns
W-C time[4]	6763us	4753us	5316us	2614us	988us
ALMs	828	832	812	115	2703
DSP blocks	18	18	18	18	68
Area×Delay	5599764	3954496	4316592	300610	2670564

Table A.1: Mandelbrot HLS Analysis

Performance Metrics:

cycle latency: number of clock cycles for execution
Fmax: Maximum possible hardware clock frequency to run the circuit
Clock Period: The inverse of clock $Fmax$, $\frac{1}{Fmax}$
Wall-clock time: It is the key performance metric for HLS, computed as the product of cycle latency and the clock period.$cyclelatency \times clockperiod$
ALMs:The number of used Adaptive Logic Module (ALM) which indicates the area consumed on the FPGA
DSP block: The number of used Digital Signal Processing units on the FPGA
Area-Delay: HLS projects are considered as multi-objective optimization problems in which a trade-off among several metrics is taken into account. *Delay* and *Area* are two prominent performance criteria that are related inversely, meaning that, assuming a reasonable design, the design space,mathematically called feasible region, includes solutions which introduce faster circuits but bulkier and vice versa. Evidently it is designers' decision to select between different optimum extremes according to desired specifications.Whenever there is a multi-objective problem there is not a unique optimum solution.But,evidently, there is a point in the space of contrasting optimum solutions where both metrics can have a minimum value, relatively. In theory the curve of different optimums is called pareto front. In the context of HLS problem we are interested in smaller but faster solutions, hence considering different design approach we calculate $AREA \times DELAY$ to find our desired optimum which is actually the minimum value of this metric. figure A.1 shows the pareto front for two conflicting objective function. In this context axis are labeled by *Area* and *Cycle latency.*

Pure Hardware Analysis:

Listing A.1: Mandelbrot Kernel

```
#define DECIMAL_PLACES 28
#define int2fixed(num) ((num) << DECIMAL_PLACES)
#define fixedmul(a, b) ((((long long)a) * \
((long long)b)) >> DECIMAL_PLACES)
#define fixed2int(num) ((num) >> DECIMAL_PLACES)

#define WIDTH 64
#define HEIGHT 64
#define MAX_ITER 50

int mandelbrot()
{
int i, j;
int count = 0;
```

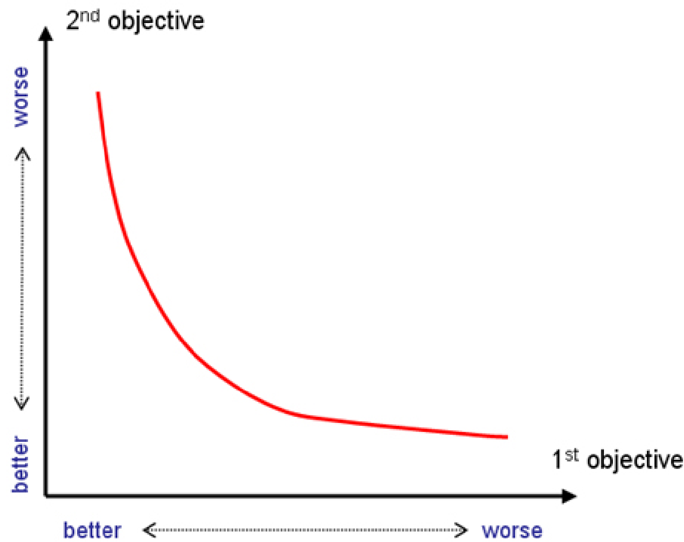

Figure A.1: Pareto Curve

```
for (j = 0; j < HEIGHT; j++)
{
for (i = 0; i < WIDTH; i++)
{
int x_0 = int2fixed(-2) + (((((3 << 20) * i)/WIDTH) ) << 8);
int y_0 = int2fixed(-1) + (((((2 << 20) * j)/HEIGHT) ) << 8);
int x = 0;
int y = 0;
int xtmp;
unsigned char iter;
unsigned char fiter = 0;

loop: for (iter = 0; iter < MAX_ITER; iter++)
{
long long abs_squared = fixedmul(x,x) + fixedmul(y,y);
xtmp = fixedmul(x,x) - fixedmul(y,y) + x_0;
y = fixedmul(int2fixed(2), fixedmul(x,y)) + y_0;
x = xtmp;
fiter  +=  abs_squared <= int2fixed(4);
}
unsigned char colour = (fiter >= MAX_ITER) ? 0 : 1;
count += colour;
}
}

return count;
}
```

In this experiment we compiled the Mandelbrot C code shown in A.1 to hardware without any structual modifications. As this code is compiled by

llvm-clang in the legup hierarchy, a main() function to call and assert the above-mentioned algorithm is added while applying the experiment. HLS executes and then produces two Verilog file named *mandelbrot.v* and *main_tb.v.* main_tb module is called a testbench, and it is used for simulation of the main Verilog module (called top) that implements Mandelbrot. The testbench is quite simple: it simulates the main Verilog module with a clock (clk), as well as asserts start and reset signals, and waits for a finish signal to become high when Mandelbrot has finished its work computing 64×64 pixels. A.2 shows the mandelbrot call graph, BB_12 (12th basic block) indicates the inner-most loop which is computationally intensive. This loop is scheduled into three clock cycles. On A.3 you can see the name of LLVM instructions and the way they are scheduled on corresponding resultant hardware.since we observe that the inner-most loop has been scheduled with 3 clock cycles, and since that loop body executes $64 \times 64 \times 50 = 204800$ times, we expect the total number of cycles spent executing the hardware to be roughly $204800 \times 3 = 614400$ cycles. Of course, this doesn't count some of the overhead operations outside of the inner-most loop. Simulating the hardware by ModelSim reveals the actual number of cycles which is 759K cycles. The Simulation is done in Register Transfer level (RTL) and called functional simulation. we make use of Altera Quartus II to map the verilog code to Altera Cyclone V FPGA this procedure includes synthesizing,placement,routing and timing analysis. Results of our interest are reported in A.1.

Loop Pipelining Analysis:

The main concept of loop pipelining has been discussed in previous chapters. here we roughly review the main idea to analyze the performance of the mandelbrot exploiting loop pipelining. Loop pipelining allows a new iteration of the loop to be started before the current iteration has finished. By allowing the execution of the loop iterations to be overlapped, higher throughput can be achieved. The amount of overlap is controlled by the initiation interval. The initiation interval (II) indicates how many cycles are taken before starting the next loop iteration. Thus an II of 1 means a new loop iteration can be started every clock cycle, which is the best case. The II needs to be larger than 1 in other cases, such as when there is a resource contention (multiple loop iterations need the same resource in the same clock cycle) or when there are loop-carried dependencies (the output of a previous iteration is needed as an input to the subsequent iteration).
A.4 shows an example of loop pipelining . A.4(A) shows the sequential loop, where the II=3, and it takes 8 clock cycles for the 3 loop iterations before the final write is performed. A.4 (B) shows the pipelined loop. In this case, there are no resource contentions or data dependencies. Hence the II=1, and it takes 4 clock cycles before the final write is performed. You can see that

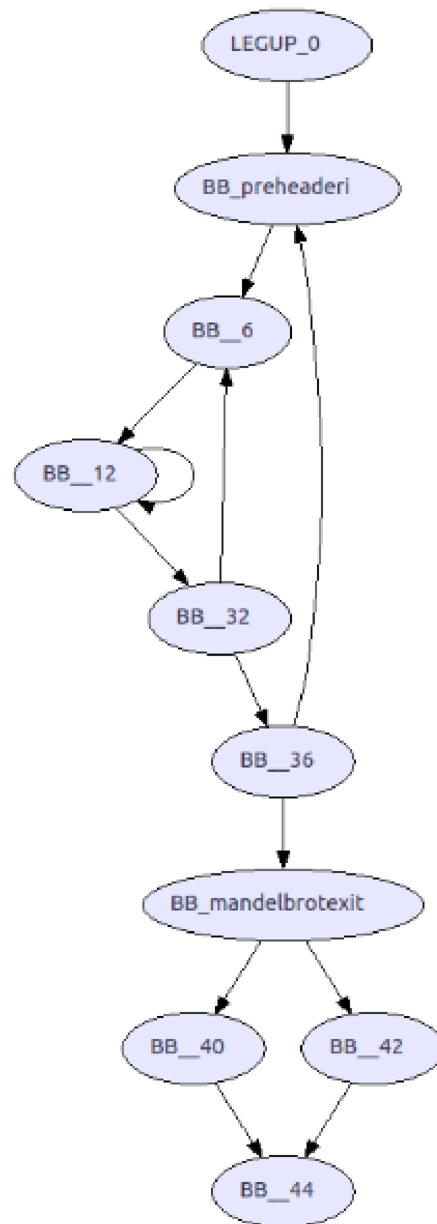

Figure A.2: Mandelbrot call graph

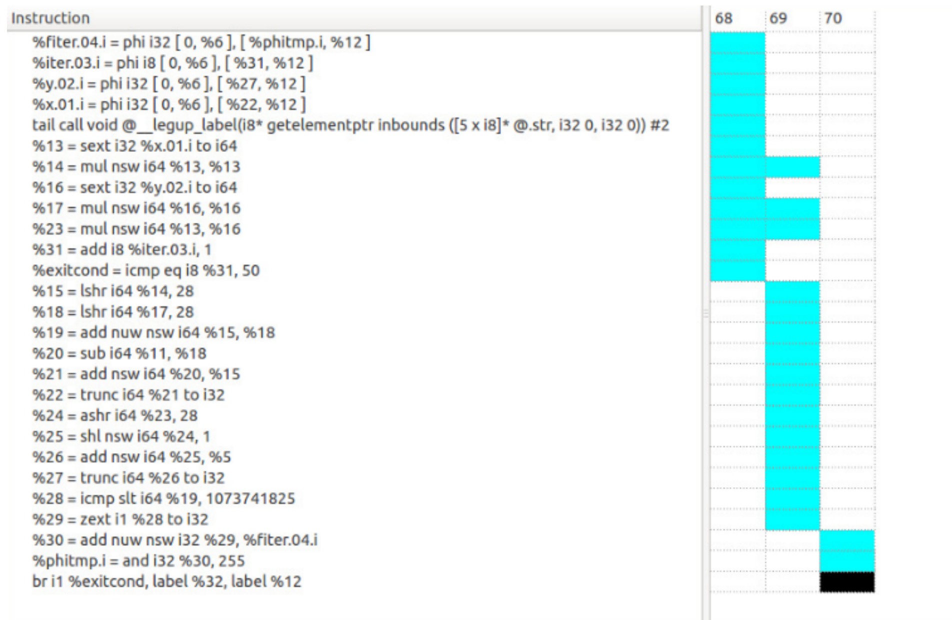

Figure A.3: Mandelbrot scheduling details

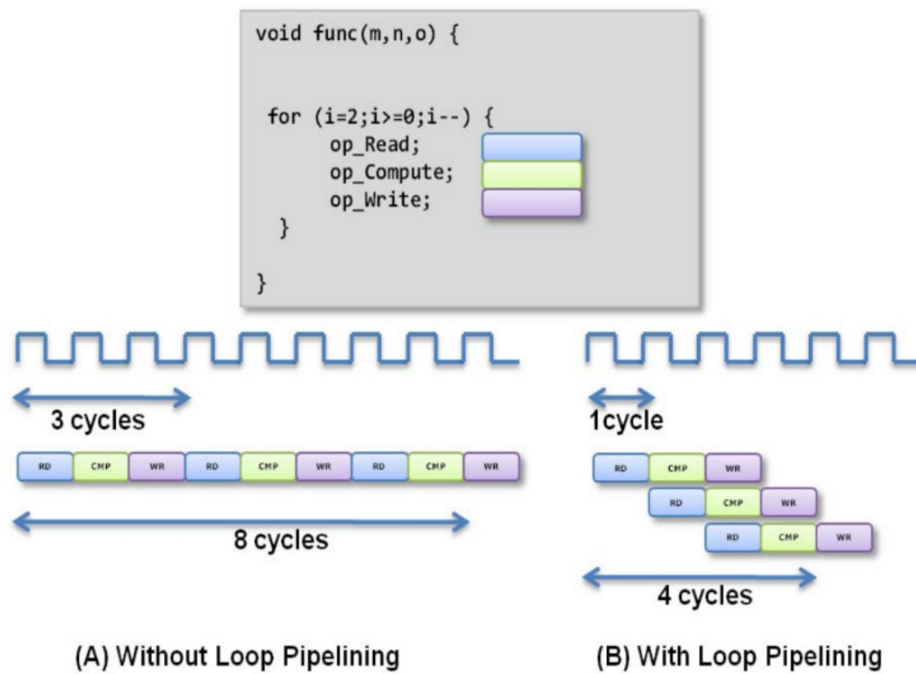

Figure A.4: Loop Pipelining

loop pipelining can significantly improve the performance of your circuit, especially when there are no data dependencies or resource contentions. In A.1 there is a loop nest, we can only pipeline the inner-most loop. In A.1 you can observe the performance measurements while pipelining the loop.An II of 2 was achieved for the loop, and the pipeline length is 3. in A.5 a visual view for the pipelined schedule is available. we see that the II of the loop is 2, and that the length of the pipeline is 3 cycles. The dark black rectangle illustrates what the pipeline looks like in the steady state. In the steady state, two iterations of the loop are "inflight" at once. Since we observe that the loop II is 2, and the loop executes 64*64*50 = 204800times, the total time spend in the inner-most loop is roughly 204800*2 = 409600 cycles. This is an approximation, as it doesn't include the initial time to fill the pipeline, nor does it include the time to flush the pipeline for each pixel. ModelSim simulation of the design reveals that that loop pipelining has dramatically improved the cycle latency for the design, reducing it from 759K cycles to 559K cycles in total. Invoking synthesis tool-chain shows the other performance metrics available in A.1

Multipliers Latency:

Certain computations in hardware may take multiple clock cycles to complete. For example, a load from memory takes two cycles, and a store to memory takes one cycle. Other operations, however, are completely combinational - they take zero cycles to complete and may be chained together with other zero-latency operations in a single clock cycle. In LegUp, for example,add, subtract, and all logical operations are zero-latency operations. By default in LegUp, multiplication operations take a single cycle to complete. The rationale for this default setting is to improve the FMax of the circuits produced. We have observed with zero-latency multiplies, that multiplies are often on the critical path of the resultant circuits. So, setting the latency of multiplies to 1 is generally good for FMax. On the other hand, non-zero-latency operations are generally worse for cycle latency, as they cannot be chained with other operations in the same clock cycle. In the Mandelbrot example, the inner loop is heavy on multiplies, thereby lengthening the number of clock cycles needed for each inner loop iteration. LegUp provides a mechanism to change the cycle latency of an operation. In this case, we set the multiply latency to 0 cycles and synthesize the design again (with loop pipelining),we see that an initiation interval of 1 has been achieved for the inner loop.looking at the loop pipeling schedule, we observed an iteration of the loop starting each cycle but this lower II impacts on cycle latency for the entire hardware execution, it is observed by simulating and the synthesizing the project.As reported in A.1 lower II has drastically reduced cycle latency in comparison to the prior steps.Also, the

Figure A.5: Loop Pipelining Scheduling

FMax has dropped to about 64MHz. By reducing the multiplier latency to 0, we reduced its cycle latency; however, the circuit's critical path is considerably worse. This is a key trade-off in high-level synthesis: cycle latency can typically be traded-off with FMax - by lengthening the clock period (worse FMax), we can normally reduce cycle latency by chaining operations together in a clock cycle.

Loop Transformation:

The code for the Mandelbrot example contains a triply-nested loop. The outer loop walks over the rows; the first inner loop walks over the columns; the inner-most loop iterates through the iterations for a particular pixel. If you look carefully at the inner-most loop, you can see it contains a loop-carried dependency: the ith iteration of the loop depends on the $i-1^{\text{th}}$ iteration. Because of this dependency, loop pipelining could not achieve an II of 1 without reducing the multiplier latency to 0, as you did in the prior step of the lab. For this step, you will modify the C code to remove the dependency entirely, by using an approach called loop interchange. The basic idea is to interchange the first inner loop with the inner-most loop, thereby producing an inner-most loop without any loop-carried dependency. Conceptually, in the original version of the code, each pixel was considered in turn. That is, the code computed whether a given pixel was in the Mandelbrot set,and then moved onto determine whether the next pixel was in the Mandelbrot set. With loop interchange,the order in which things are computed will be changed fundamentally: the inner-most loop will compute z_i for an entire row of pixels. After this is completed, the next run of the inner-most loop will compute $z_i + 1$for the same row of pixels. In essence, the Mandelbrot set computations for an entire row of pixels will be "in flight". To make this work, we need to store the z_i values for all such pixels that are in flight: an entire row of pixels.In the code belowA.2, the first few lines of the function declare arrays that allow us to store intermediate data for an entire row of the image. Following these declarations, observe that the outer loop is over the rows of the image (as before). Within the outer loop, the original functionality has been split into three sections: In the first section, labelled *lp1*, there is an inner loop that initializes all data for a row of the image. The second section, has a doubly nested loop, where the loop interchange has been implemented. The top-level loop of this nest corresponds to the inner-most loop of the original code; the inner-most loop here, labelled *lp2*, iterates over all the columns of a row. The third and final section of the code below, labelled *lp3*, performs the accumulation to count for all pixels in a row. The code below implements loop interchange, and also loop fission (also sometimes called loop distribution), where the triply nested loop in the original code has been split into multiple separate loops.

Listing A.2: Transformed Mandelbrot Kernel

```
#define DECIMAL_PLACES 28
#define int2fixed(num) ((num) << DECIMAL_PLACES)
#define fixedmul(a, b) ((((long long)a) *\
((long long)b)) >> DECIMAL_PLACES)
#define fixed2int(num) ((num) >> DECIMAL_PLACES)

#define WIDTH 64
#define HEIGHT 64
#define MAX_ITER 50

int mandelbrot() {
int i, j;
int count = 0;

int x_0 [WIDTH] = {0};
int y_0 [WIDTH] = {0};
int x [WIDTH] = {0};
int y [WIDTH] = {0};
unsigned char fiter[WIDTH] = {0};

for (j = 0; j < HEIGHT; j++) {

lp1:for (i = 0; i < WIDTH; i++)
{
        x_0[i] = int2fixed(-2) +
    (((((3 << 20) * i)/WIDTH) ) << 8);
        y_0[i] = int2fixed(-1) +
    (((((2 << 20) * j)/HEIGHT) ) << 8);
        x[i]=0;
        y[i]=0;
        fiter[i]=0;
}

int xtmp;
unsigned char iter;
for (iter = 0; iter < MAX_ITER; iter++)
{

lp2:for (i=0; i < WIDTH; i++)
{
long long abs_squared = fixedmul(x[i],x[i]) +
fixedmul(y[i],y[i]);
xtmp = fixedmul(x[i],x[i]) -
fixedmul(y[i],y[i]) + x_0[i];
y[i] = fixedmul(int2fixed(2), fixedmul(x[i],y[i])) +
y_0[i];
x[i] = xtmp;
fiter[i]  +=  abs_squared <= int2fixed(4);
}

}
```

```
lp3:for (i=0; i < WIDTH; i++)
{
unsigned char colour = (fiter[i] >= MAX_ITER) ? 0 : 1;
count += colour;
}
}

return count;
}
```

we apply loop pipelining to the loop where interchange has been applied: that labelled *lp2*, then we take into account other possible pipelinings which are labeled by *lp1* and *lp3*. Also multiplier latency is set back to its original value of 1. Simulating and synthesis results are reported in A.1. Evidently, a significant reduction in cycle latency has been achieved. In A.6 loop pipeline scheduling is reported visually. three basic blocks with the names:wait_lp1_1,wait_lp2_1,and wait_lp3_1, corresponding to the three pipelined loops. The schedule for lp1 is particularly interesting: the II of the loop is 1, however, the length of the pipeline is over 30 cycles long. The reason for this is that lp1 contains a division operation, which itself is heavily pipelined by LegUp to improve the FMax of the design.

Parallel Processing:

In the parallel processing approach, we exploit a key advantage of moving computations into hardware: the ability to exploit spatial parallelism. We parallelize the software code using Pthreads, and then synthesize the parallel code into parallel hardware. At a high-level, the idea is to launch four threads, each of which responsible for computing the pixel values for 1/4 of the rows of the image. The HLS tool then synthesize four Mandelbrot "cores" in the hardware, that operate concurrently with one another. Since each core is responsible for only 1/4 of the original work, the total cycle latency is reduced to roughly 25% of the original latency. Naturally, the resultant circuit takes up about 4x the area on the FPGA. In essence, we are trading off area for time- an optimization approach that is very typical in computer hardware design.

Listing A.3: Threaded Mandelbrot Kernel

```
#include <pthread.h>

#define DECIMAL_PLACES 28
#define int2fixed(num) ((num) << DECIMAL_PLACES)
#define fixedmul(a, b) ((((long long)a) *\
((long long)b)) >> DECIMAL_PLACES)
#define fixed2int(num) ((num) >> DECIMAL_PLACES)

#define WIDTH 64
```

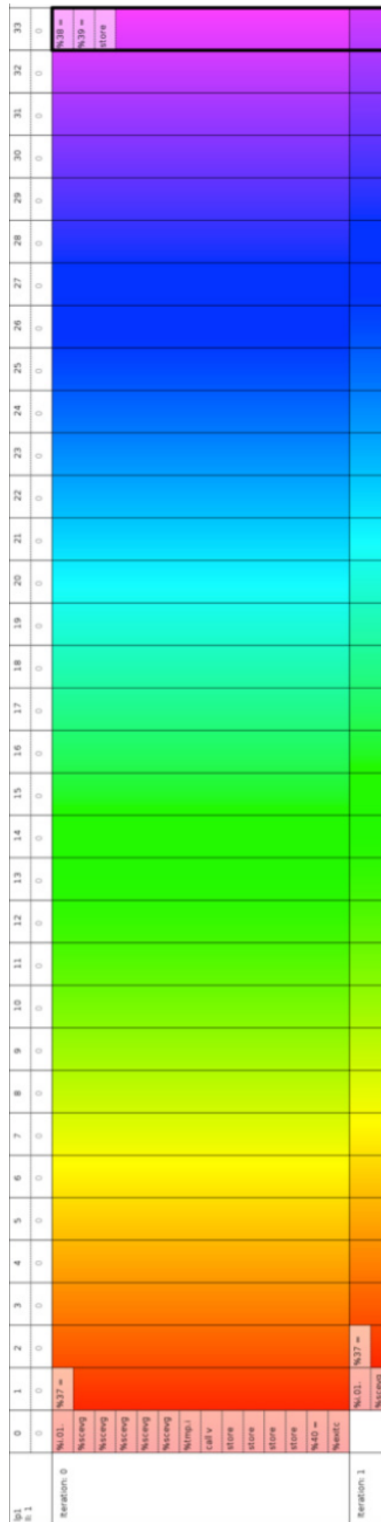

Figure A.6: Schedule for loop lp1 in the transformed Mandelbrot code.

```
#define HEIGHT 64
#define MAX_ITER 50
#define NUM_ACCEL 4
#define OPS_PER_ACCEL HEIGHT/NUM_ACCEL
#define OMP_ACCEL 4

struct thread_data{
        int  startidx;
        int  maxidx;
};

volatile unsigned char img[WIDTH][HEIGHT];

void *mandelbrot(void *threadarg) {
int i, j, tid;
        int count = 0;
        struct thread_data* arg = (struct thread_data*) threadarg;
        int startidx = arg->startidx;
        int maxidx = arg->maxidx;

for (j = startidx; j < maxidx; j++)
{
for (i = 0; i < WIDTH; i++)
{

int x_0 = int2fixed(-2) + ((((3 << 20) * i/WIDTH) ) << 8);
int y_0 = int2fixed(-1) + ((((2 << 20) * j/HEIGHT) ) << 8);

int x = 0;
int y = 0;
int xtmp;
unsigned char iter;
unsigned char fiter = 0;

loop:for (iter = 0; iter < MAX_ITER; iter++) {
long long abs_squared = fixedmul(x,x) + fixedmul(y,y);
xtmp = fixedmul(x,x) - fixedmul(y,y) + x_0;
y = fixedmul(int2fixed(2), fixedmul(x,y)) + y_0;
x = xtmp;
fiter  +=  abs_squared <= int2fixed(4);
}

unsigned char colour = (fiter >= MAX_ITER) ? 0 : 1;
                        count += colour;
                        img[i][j] = colour;
}
}

pthread_exit((void*)count);
}


int main() {
        int final_result = 0;
```

```
        int i, j;
        int count[NUM_ACCEL];
        pthread_t threads[NUM_ACCEL];
        struct thread_data data[NUM_ACCEL];

        //initialize structs to pass into accels
        for (i=0; i<NUM_ACCEL; i++) {
                data[i].startidx = i*OPS_PER_ACCEL;
                if (i == NUM_ACCEL-1) {
                        data[i].maxidx = HEIGHT;
                } else {
                        data[i].maxidx = (i+1)*OPS_PER_ACCEL;
                }
        }

        //launch threads
for (i=0; i<NUM_ACCEL; i++)
{
pthread_create(&threads[i], NULL, mandelbrot, (void *)&data[i]);
}

//join the threads
for (i=0; i<NUM_ACCEL; i++)
{
        pthread_join(threads[i], (void**)&count[i]);
}

//sum the results
for (i=0; i<NUM_ACCEL; i++)
{
        final_result += count[i];
}

}
```

In A.3 you can find the same mandelbrot code that is written in threaded structure using pthred library. A C structure is defined at the top of the code with two fields: *startidx* and *maxidx*. For each thread, these integers hold the starting row and ending row of the image that each thread is responsible for. The mandelbrot function is nearly identical to A.1 however, in this case, the function parameter is a pointer to the structure (cast to void*). The first thing the function does is to extract the starting and ending rows from the structure. These are then the bounds of the outer loop (for j).
Looking down in the code, within the main function, a code snippet, sets up the work for each thread (the set of rows it is responsible for). Here,$NUMACCEL$ is the number of accelerators that should be synthesized (4 in this case). Further down in the code, there are calls to *pthread_create* and *pthread_join*, which create the threads and wait for them to complete, respectively. Executing HLS then simulating and synthesizing the threaded code, results the

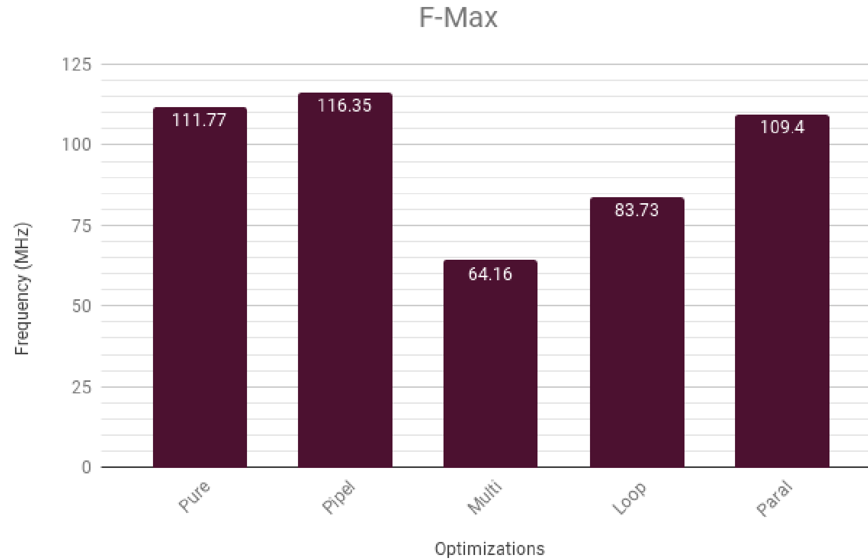

Figure A.7: Mandelbrot Fmax on different HLS approaches

measurements reported in A.1. It is observed that the parallel approach has the least clock cycle and delay but the most area. There is no such thing as free lunch!

A discussion on HLS strategies:

Fmax is generally affected by circuit critical path, the longer critical path which is the result of a more complex combinational circuit needs a longer clock period to finish the computations ,hence less cycles in the time steps. Usually multipliers play a significant role in prolonging clock period thereby synthesizing a complex multiplier needing less clock cycles makes the Fmax less too.

Considering a trend line on A.8 shows the fact that a more parallel design decreases the wall clock time. There is a huge difference between Pure hardware implementation and a parallel approach which indicates direct hardware synthesis is not always optimally parallel hence an extra effort is needed to re-design the system in data parallel to task parallel fashion accordingly. Moreover, as the clock frequencies in different design approaches are different, other metrics such as clock cycles and clock periods cannot reveal the aggregated speed up of the synthesized circuit

Clock Period is the inverse of frequency as mentioned before, decreasing the multiply latency results in longer clock period. By the way the clock period alone is not a good performance measurement metric.

Reviewing A.10 shows that there is a definite decrease on the number of

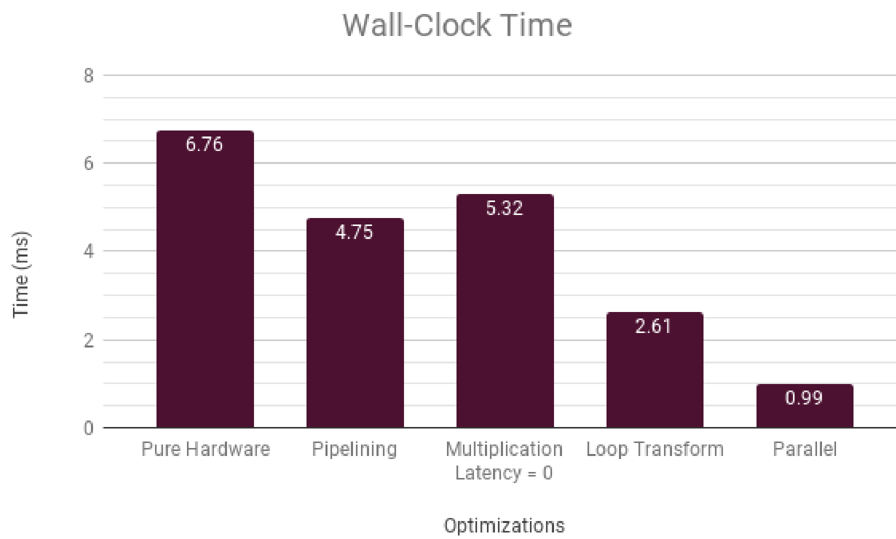

Figure A.8: Mandelbrot Wall Clock Time on different HLS approaches

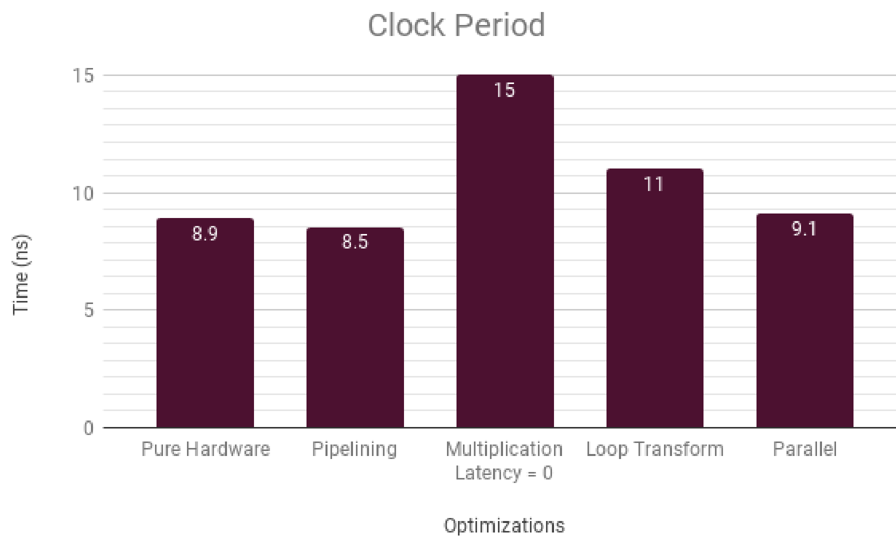

Figure A.9: Mandelbrot Clock Period on different HLS approaches

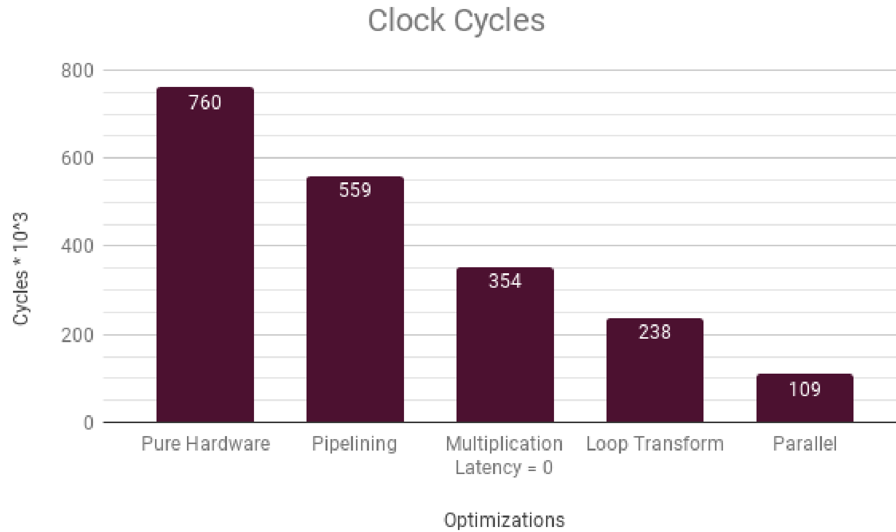

Figure A.10: Mandelbrot Clock cycles on different HLS approaches

clock cycles as moving form pure hardware to other synthesis optimizations and finally a parallel approach which indicates the smallest value. It is 85% improvement but as the clock frequencies on different approached are not the same, this performance improvement is biased. The actual performance improvement is visible on A.8. But,all in all, a more parallel hardware design is considered as a spatial decomposition and parallelization which always reduces the number of clock cycles.

It is obvious in A.11 that only the parallel design increases the number of DSP blocks consumed.

Figure A.12 shows a key plot that motivates the proposed framework in this thesis. Taking into account the importance of Area*Delay, the figure shows that *Loop Transformations* or generally *Code Transformations* yields the best Area-delay product meaning that the concept of source to source compilation to reform the programming structure not only affects in the context of sequential softwares but also extremely impacts on *High Level Synthesis* and subsequent circuitries in terms of area and speed. In a nutshell, code transformations normally result in a pareto optimum point.

A.13 shows that Code Transformation have conveys the best gain in resource consumption while the parallel approach is the worst. It is obvious that spatially replicating the hardware modules needs more logic elements.

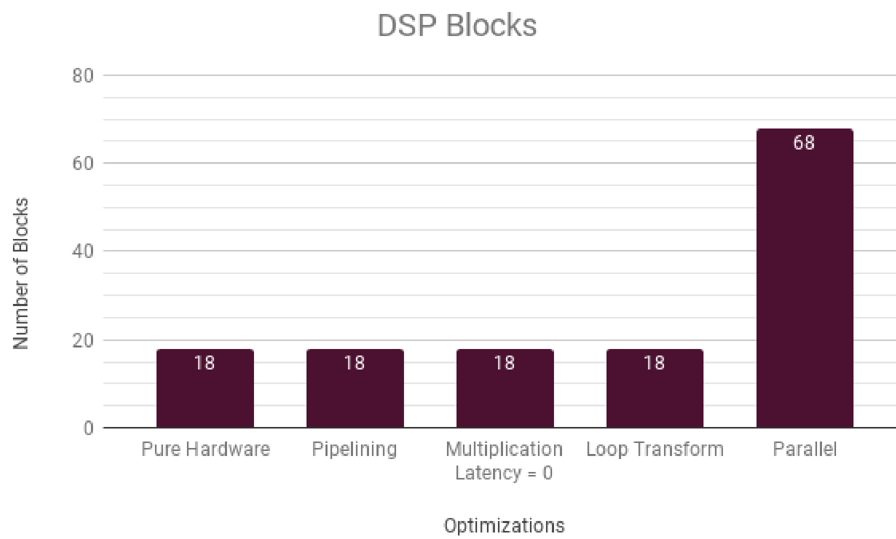

Figure A.11: Mandelbrot DSP blocks on different HLS approaches

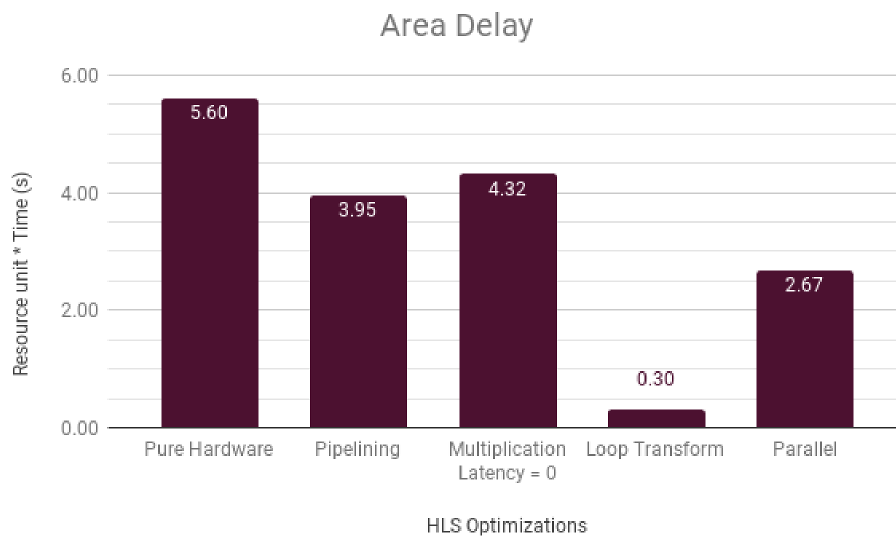

*Figure A.12: Mandelbrot Area*Delay on different HLS approaches*

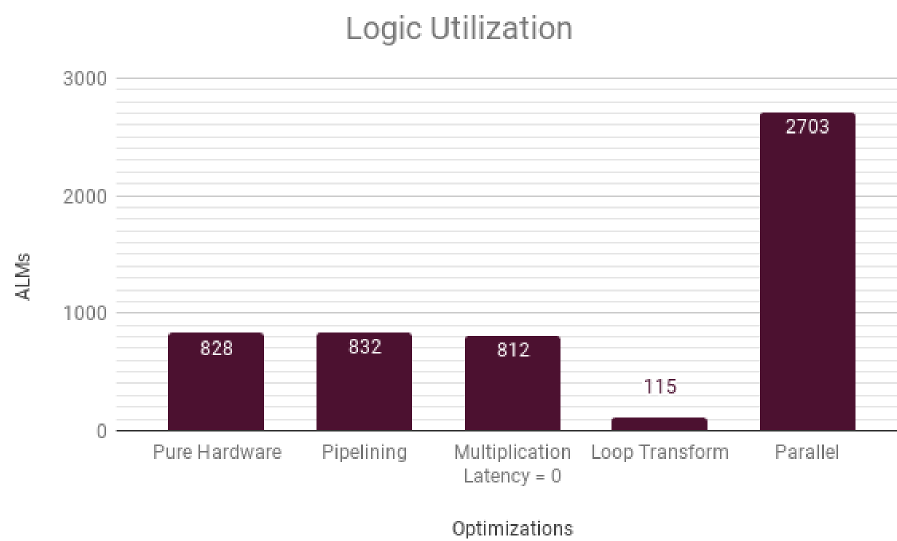

Figure A.13: Mandelbrot ALMs on different HLS approaches

Appendix B

Loop Transformation Modeling

B.1 Source to Source Transformation

As the *PluTo* integrated in *Orio* implements loop transformations based on polyhedral modeling, also taking into account various loop optimizations that can be enabled by LLVM *opt* command, in this appendix we describes the theory of source code optimization techniques.

HLS has lifted the design abstraction from RTL to C/C++, but in practice extensive source code rewriting is often required to achieve a good design using HLS especially when the design space is too large to determine the proper design options in advance. In addition, this code rewriting requires not only the knowledge of hardware microarchitecture design, but also familiarity with the coding style for the high-level synthesis tools. Automatic source-to-source transformation techniques have been applied in software compilation and optimization for a long time. They can also greatly benefit the FPGA accelerator design in a high-level synthesis design flow. In general, source-to-source optimization for FPGA will be much more complex and challenging than that for CPU software because of the much larger design space in microarchitecture choices combined with temporal/spatial resource allocation. The goal of source-to-source transformation is to reduce or eliminate the design abstraction gap between software/algorithm development and existing HLS design flows. This will enable the fully automated FPGA design flows for software developers, which is especially important for deploying FPGAs in data centers, so that many software developers can efficiently use FPGAs with minimal effort for acceleration.

B.2 Theory of code transformation

B.2.1 Polyhedral Model for nested loops

Polyhedral models can be used to represent execution information of a program's loop nests, such as the loop iteration domain, statement/iteration dependencies, array access functions, and scheduling functions (execution order). Mathematical definitions of this model are described `http://polyhedral.info/` and related articles.

Here,as a case study, we roughly introduce mathematical modeling of loop transformations via polyhedral modeling [53]. To do so, first Array Access Patterns are introduced.

Classification of Array Access Patterns

We model the array access patterns using the polyhedral model, thus we assume that all array accesses are affine expressions of loop indices and constants. The program inputs are composed of multiple data-dependent blocks where each block contains a single multi-dimensional loop nest. Let us consider a loop nest of dimensionality D that accesses an N-dimensional array. The array access pattern is defined by matrix M whose size is $N \times D$, where the rows (i) represent the data access pattern in dimension i of the data array, and columns (j) represent the access pattern in the loop level j. Given the array access pattern $\boldsymbol{M}$, loop iteration vector $\vec{i}$, and constant offset vector $\vec{o}$, the array access vector $\vec{s}$ is defined as

$$\vec{s} = \mathbf{M}\vec{i} + \vec{o}$$

$\vec{s}$ is column vector of size N , where each row (i)represents array accesses in dimension i, and the offset vector is a constant offset into that dimension. Figure **??** shows an example of array access pattern and vector for the writes to array B; similar access patterns and vectors could be derived for the reads from array A. In the following, we demonstrate the data access patterns for 2-dimensional arrays and 2-dimensional loop-nests. For ease of illustration, we classify the access patterns below.The polyhedral model can be easily extended to handle a wide variety of additional access patterns, and our work can use any additional access patterns to estimate the performance benefit.Now we will describe how to define the array access patterns using **M**. Let

$$M = \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix}$$

We classify the array access patterns based on the values of a_1, a_2, b_1 and b_2. For array accesses with non-unit loop strides, we perform loop normalization as a preprocessing step so that our analysis can assume unit loop stride.

```
for(i =0; i < N; i++)
  for(j=0; j < N; j++)
   B[i][j] = A[i][j] + A[i - 1][j];
```

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \vec{s} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Figure B.1: An Example of Array Access Pattern

Column and Reverse Column.

$$\begin{pmatrix} \pm 1 & 0 \\ 0 & \pm 1 \end{pmatrix}$$

The array access vector can be obtained by $M\vec{i}$. Figure 5 shows the four patterns in this category, where the signs of a_1 and b_2 determine traversal direction. For example, with outer loop index i and inner loop index j, if $a_1 = 1$ and $b_2 = 1$, then the outer loop traverses increasing values of i, and the inner loop traverses increasing values of j.

Row and Reverse Row

$$\begin{pmatrix} 0 & \mp 1 \\ \mp 1 & 0 \end{pmatrix}$$

Similar to column and reverse column, there are four patterns in this category, and the signs of $b_1$and a_2 determine the traversal directions.

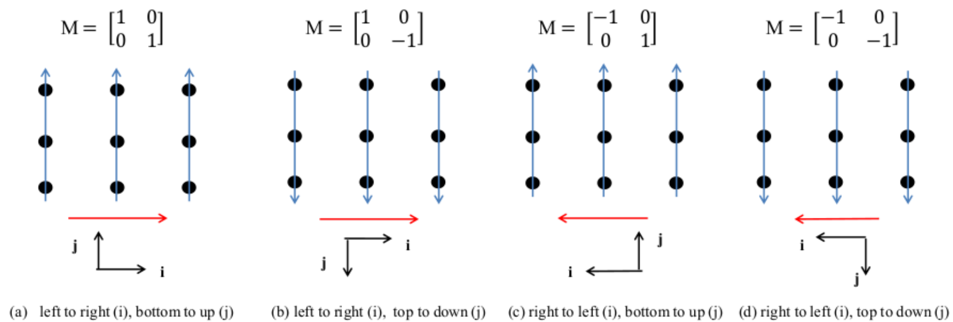

Figure B.2: An An example of column access pattern. There are 4 patterns with different traverse directions.

Diagonal Access

In this category, the loop traverses in a diagonal line fashion. We further divide this into two cases based on the slopes of the diagonal lines.

- *slope* > 1.

$$\begin{pmatrix} \pm 1 & N > b_1 \geq 1 \\ 0 & \pm 1 \end{pmatrix}$$

  The array access vector can be obtained by $\vec{i}$. The slope of the diagonal line is determined by b_1 , and the signs of a_1 and b_2 determine the traversal directions.When b 1 $\geq N$, the traversal order reduces to one of the column access orders. Figure B.3 shows the four data access patterns for *slope = 1.*

- *slope* < 1.

$$\begin{pmatrix} N > a_1 \geq 1 & \mp 1 \\ \mp 1 & 0 \end{pmatrix}$$

  The slope of the diagonal line is determined by $\frac{}{1}a_1$, and the signs of a_2 and b_1 determine the traversal directions. When $a_1 \geq N$,the traversal order reduces to one of the row access orders.

All of the array access patterns defined above are unimodular matrices where $\|a_1 \times b_2 - a_2 \times b_1\| = 1$. Thus, all of these access patterns can be achieved by unimodular loop transformations of the block(s).

B.3 Loop Transformation

Loop transformations can change the schedule (e.g., execution order) of loop iterations such that the data access pattern can be changed. Thus, here we derive the loop transformation given a desired data access pattern.
THEOREM 3.1. *The transformation function T required for the desired data access pattern M_{des} can be obtained by*

$$T = M_{des}^{-1} M_{ori} F_{ori}^{-1}$$

where M_{ori} and F_{ori} are the data access pattern and schedule function of the source code without transformation.
PROOF. Let $\vec{i'}$ be the loop iterator vector after transformation. Thus, the desired array access vector after transformation is $M_{des}\vec{i'}$. Let us assume the schedule function and data access pattern of the original code are F_{ori} and M_{ori},respectively. Thus, the schedule function after transformation is TF_{ori} . Schedule function maps the original loop iterations to a new ordering of the loop iterations. Thus,

$$TF_{ori}\vec{i} = \vec{i'}$$

Thus,

$$\vec{i} = F_{ori}^{-1} T^{-1} \vec{i'}$$

The data accessed by the iterations before and after transformation is the same

$$M_{ori} F_{ori}^{-1} T^{-1} \vec{i'} = M_{des} \vec{i'}$$

Thus

$$T = M_{des}^{-1} M_{ori} F_{ori}^{-1}$$

The data access pattern and loop transformations are all unimodular matrix. For unimodular matrix, we can always derive its reverse matrix.

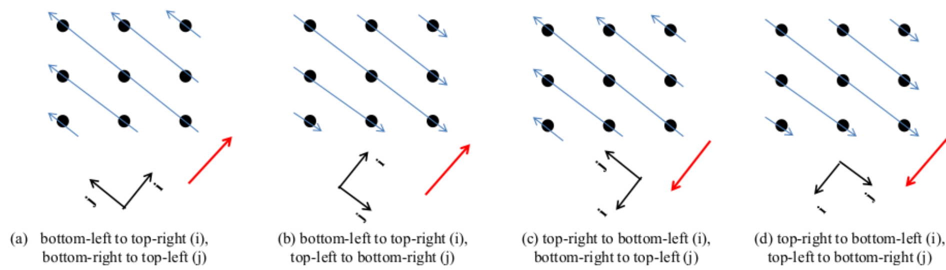

Figure B.3: An example of diagonal access pattern with slope $= 1$. *There are 4 patterns with different traverse directions.*

Appendix C

Codes

C.1 AXPY

Listing C.1: Annotated AXPY code

```
/*@ begin Loop (
    transform Composite(
      unrolljam = (['i'],[UF]),
      vector = (VEC, ['ivdep','vector always'])
     )
  for (i=0; i<=N-1; i=i+1){
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i]+a5*x5[i];
  }
) @*/
  for (i=0; i<=N-1; i=i+1){
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i]+a5*x5[i];
  }
/*@ end @*/
```

Listing C.2: AXPY Orio specification code to use for S2S transformation

```
PerfTuning (spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,100);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0','-O1','-O2','-O3'];
 }
 def input_params {
   param N[] = [100000];
 }
```

```
 def input_vars {
   decl dynamic int x1[N] = random;
   decl dynamic int x2[N] = random;
   decl dynamic int x3[N] = random;
   decl dynamic int x4[N] = random;
   decl dynamic int x5[N] = random;
   decl dynamic int y[N] = 0;
   decl int a1 = random;
   decl int a2 = random;
   decl int a3 = random;
   decl int a4 = random;
   decl int a5 = random;
 }
 def search {
   arg algorithm = 'Exhaustive';
 }
}
```

Listing C.3: Tuned and transformed AXPY code after S2S transformation by Orio

```
@*/
/**-- (Generated by Orio)
Tuned for specific problem sizes:
  N = 100000
Best performance parameters:
  CFLAGS = -O3
  UF = 7
  VEC = True
--**/

/*@ begin Loop (
    transform Composite(
      unrolljam = (['i'],[UF]),
      vector = (VEC, ['ivdep','vector always'])
     )
  for (i=0; i<=N-1; i=i+1){
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i]+a5*x5[i];
  }
) @*/
{
  int i;
  register int cbv_1;
  cbv_1=N-7;
#pragma ivdep
#pragma vector always
  for (i=0; i<=cbv_1; i=i+7) {
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i]+a5*x5[i];
    y[(i+1)]=y[(i+1)]+a1*x1[(i+1)]+a2*x2[(i+1)]+a3*x3[(i+1)]+a4*x4
        [(i+1)]+a5*x5[(i+1)];
    y[(i+2)]=y[(i+2)]+a1*x1[(i+2)]+a2*x2[(i+2)]+a3*x3[(i+2)]+a4*x4
        [(i+2)]+a5*x5[(i+2)];
    y[(i+3)]=y[(i+3)]+a1*x1[(i+3)]+a2*x2[(i+3)]+a3*x3[(i+3)]+a4*x4
        [(i+3)]+a5*x5[(i+3)];
    y[(i+4)]=y[(i+4)]+a1*x1[(i+4)]+a2*x2[(i+4)]+a3*x3[(i+4)]+a4*x4
```

```
        [(i+4)]+a5*x5[(i+4)];
    y[(i+5)]=y[(i+5)]+a1*x1[(i+5)]+a2*x2[(i+5)]+a3*x3[(i+5)]+a4*x4
        [(i+5)]+a5*x5[(i+5)];
    y[(i+6)]=y[(i+6)]+a1*x1[(i+6)]+a2*x2[(i+6)]+a3*x3[(i+6)]+a4*x4
        [(i+6)]+a5*x5[(i+6)];
  }
  register int cbv_2, cbv_3;
  cbv_2=N-((N-(0))%7);
  cbv_3=N-1;
#pragma ivdep
#pragma vector always
  for (i=cbv_2; i<=cbv_3; i=i+1)
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i]+a5*x5[i];
}
/*@ end @*/
```

C.2 ADPCM

Listing C.4: Annotated ADPCM code

```
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
     for (i=0; i <= (IN_END-1); i=i+2)
       {
         j = i/2;
         compressed[j] = encode (test_data[i], test_data[i + 1]);
       }
  ) @*/

  for (i = 0; i <= (IN_END-1); i=i+2)
    {
      j = i/2;
      compressed[j] = encode (test_data[i], test_data[i + 1]);
    }
 /*@ end @*/
```

Listing C.5: ADPCM Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,2);
   param VEC[] = [False,True];
```

```
   param CFLAGS[] = ['-O0','-O1','-O2','-O3'];
 }
 def input_params {
   param SIZE[] = [100];
   param IN_END[] = [100];
 }
 def input_vars {
   decl dynamic int accumc[11] = random;
   decl dynamic int accumd[11] = random;
   decl dynamic int tqmf[24] = random;
   decl dynamic int delay_bpl[6] = random;
   decl dynamic int delay_bph[6] = random;
   decl dynamic int dec_del_bpl[6] = random;
   decl dynamic int dec_del_bph[6] = random;
   decl dynamic int delay_dltx[6] = random;
   decl dynamic int delay_dhx[6] = random;
   decl dynamic int dec_del_dltx[6] = random;
   decl dynamic int dec_del_dhx[6] = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.6: Tuned and transformed ADPCM code after S2S transformation by Orio

```
  /**-- (Generated by Orio)
Best performance cost:
  [9.99e-07, 4.14e-07, 3.54e-07, 3.55e-07, 3.53e-07, 3.55e-07,
      3.56e-07, 3.59e-07, 3.54e-07, 3.56e-07, 3.52e-07, 3.57e-07,
      3.52e-07, 3.54e-07, 3.54e-07, 3.55e-07, 3.56e-07, 3.57e-07,
      3.54e-07, 3.52e-07, 3.53e-07, 3.56e-07, 3.54e-07, 3.56e-07,
      3.57e-07, 3.55e-07, 3.58e-07, 3.56e-07, 3.54e-07, 3.53e-07,
      3.57e-07, 3.57e-07, 3.56e-07, 3.54e-07, 3.54e-07, 3.52e-07,
      3.54e-07, 3.8e-07, 3.54e-07, 3.56e-07, 3.81e-07, 3.55e-07,
      3.49e-07, 3.54e-07, 3.57e-07, 3.58e-07, 3.55e-07, 3.56e-07,
      3.74e-07, 3.56e-07, 3.57e-07, 3.56e-07, 3.54e-07, 3.54e-07,
      3.74e-07, 3.56e-07, 3.54e-07, 3.53e-07, 3.55e-07, 3.56e-07,
      3.54e-07, 3.54e-07, 3.53e-07, 3.57e-07, 3.58e-07, 3.58e-07,
      3.53e-07, 3.52e-07, 3.56e-07, 3.54e-07, 3.54e-07, 3.54e-07,
      3.55e-07, 3.54e-07, 3.54e-07, 3.56e-07, 3.55e-07, 3.59e-07,
      3.56e-07, 3.56e-07, 3.56e-07, 3.54e-07, 3.54e-07, 3.55e-07,
      3.54e-07, 3.53e-07, 3.54e-07, 3.55e-07, 3.57e-07, 3.55e-07,
      3.54e-07, 3.56e-07, 3.56e-07, 3.54e-07, 3.54e-07, 3.56e-07,
      3.54e-07, 3.56e-07, 3.54e-07, 3.55e-07]
Tuned for specific problem sizes:
  IN_END = 100
  SIZE = 100
Best performance parameters:
  CFLAGS = -O1
  UF = 1
  VEC = True
--**/
```

```
/*@ begin Loop (
    transform Composite(
      unrolljam = (['i'],[UF]),
      vector = (VEC, ['ivdep','vector always'])
     )
   for (i=0; i <= (IN_END-1); i=i+2)
     {
       j = i/2;
       compressed[j] = encode (test_data[i], test_data[i + 1]);
     }
) @*/
{
  register int cbv_1;
  cbv_1=IN_END-1;
#pragma ivdep
#pragma vector always
  for (i=0; i<=cbv_1; i=i+2) {
    j=i/2;
    compressed[j]=encode(test_data[i],test_data[i+1]);
  }
}
```

C.3 DFADD

Listing C.7: Annotated DFADD code

```
/*@ begin Loop (
    transform Composite(
      unrolljam = (['i'],[UF]),
      vector = (VEC, ['ivdep','vector always'])
     )
for (i = 0; i <= N-1; i=i+1)
      {
        float64 result;
        x1 = a_input[i];
        x2 = b_input[i];
        result = float64_add (x1, x2);
        main_result += (result == z_output[i]);
      }
) @*/
    for (i = 0; i <= N-1; i=i+1)
      {
        float64 result;
        x1 = a_input[i];
        x2 = b_input[i];
        result = float64_add (x1, x2);
        main_result += (result == z_output[i]);
      }
/*@ end @*/
```

Listing C.8: DFADD Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
```

```
 def build {
   arg build_command = 'gcc';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,32);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0', '-O1','-O2','-O3'];
   constraint divisible_by_two = (UF % 8 == 0);
 }
 def input_params {
   param N[] = [46];
 }
 def input_vars {
   decl static int a_input[46] = random;
   decl static int b_input[46] = random;
   decl static int z_input[46] = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.9: Tuned and transformed DFADD code after S2S transformation by Orio

```
/**-- (Generated by Orio)
Best performance cost:
  [8.65e-07, 3.68e-07, 2.75e-07, 2.79e-07, 2.76e-07, 2.81e-07,
      2.75e-07, 2.77e-07, 2.74e-07, 2.97e-07, 2.78e-07, 2.74e-07,
      2.92e-07, 2.75e-07, 2.76e-07, 2.78e-07, 2.78e-07, 2.79e-07,
      2.78e-07, 2.77e-07, 2.79e-07, 2.77e-07, 2.92e-07, 2.75e-07,
      2.76e-07, 2.93e-07, 2.76e-07, 2.95e-07, 2.73e-07, 2.77e-07,
      2.76e-07, 2.72e-07, 2.76e-07, 2.74e-07, 2.76e-07, 2.76e-07,
      2.75e-07, 2.77e-07, 2.76e-07, 2.77e-07, 2.76e-07, 2.77e-07,
      2.77e-07, 2.74e-07, 2.75e-07, 2.71e-07, 2.76e-07, 2.75e-07,
      2.77e-07, 2.77e-07, 2.77e-07, 2.75e-07, 2.74e-07, 2.73e-07,
      2.75e-07, 2.76e-07, 2.75e-07, 2.74e-07, 2.76e-07, 2.76e-07,
      2.75e-07, 2.74e-07, 2.75e-07, 2.77e-07, 2.79e-07, 2.81e-07,
      2.75e-07, 2.74e-07, 2.77e-07, 2.74e-07, 2.77e-07, 2.76e-07,
      2.8e-07, 2.75e-07, 2.75e-07, 2.75e-07, 2.94e-07, 2.76e-07,
      2.92e-07, 2.77e-07, 2.74e-07, 2.76e-07, 2.75e-07, 2.74e-07,
      2.76e-07, 2.73e-07, 2.75e-07, 2.75e-07, 2.75e-07, 2.76e-07,
      2.75e-07, 2.75e-07, 2.75e-07, 2.78e-07, 2.76e-07, 2.75e-07,
      2.76e-07, 2.76e-07, 2.74e-07, 2.77e-07]
Tuned for specific problem sizes:
  N = 46
Best performance parameters:
  CFLAGS = -O1
  UF = 24
  VEC = True
```

```
--**/
/*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
  for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          x1 = a_input[i];
          x2 = b_input[i];
          result = float64_add (x1, x2);
          main_result += (result == z_output[i]);
        }
  ) @*/
  {
    int i;
    register int cbv_1;
    cbv_1=N-24;
#pragma ivdep
#pragma vector always
    for (i=0; i<=cbv_1; i=i+24) {
            float64 result;
;
      x1=a_input[i];
      x2=b_input[i];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[i]);
      ;
      x1=a_input[(i+1)];
      x2=b_input[(i+1)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+1)]);
      ;
      x1=a_input[(i+2)];
      x2=b_input[(i+2)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+2)]);
      ;
      x1=a_input[(i+3)];
      x2=b_input[(i+3)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+3)]);
      ;
      x1=a_input[(i+4)];
      x2=b_input[(i+4)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+4)]);
      ;
      x1=a_input[(i+5)];
      x2=b_input[(i+5)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+5)]);
      ;
```

```
x1=a_input[(i+6)];
x2=b_input[(i+6)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+6)]);
;
x1=a_input[(i+7)];
x2=b_input[(i+7)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+7)]);
;
x1=a_input[(i+8)];
x2=b_input[(i+8)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+8)]);
;
x1=a_input[(i+9)];
x2=b_input[(i+9)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+9)]);
;
x1=a_input[(i+10)];
x2=b_input[(i+10)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+10)]);
;
x1=a_input[(i+11)];
x2=b_input[(i+11)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+11)]);
;
x1=a_input[(i+12)];
x2=b_input[(i+12)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+12)]);
;
x1=a_input[(i+13)];
x2=b_input[(i+13)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+13)]);
;
x1=a_input[(i+14)];
x2=b_input[(i+14)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+14)]);
;
x1=a_input[(i+15)];
x2=b_input[(i+15)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+15)]);
;
x1=a_input[(i+16)];
x2=b_input[(i+16)];
result=float64_add(x1,x2);
main_result=main_result+(result==z_output[(i+16)]);
```

```
      ;
      x1=a_input[(i+17)];
      x2=b_input[(i+17)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+17)]);
      ;
      x1=a_input[(i+18)];
      x2=b_input[(i+18)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+18)]);
      ;
      x1=a_input[(i+19)];
      x2=b_input[(i+19)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+19)]);
      ;
      x1=a_input[(i+20)];
      x2=b_input[(i+20)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+20)]);
      ;
      x1=a_input[(i+21)];
      x2=b_input[(i+21)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+21)]);
      ;
      x1=a_input[(i+22)];
      x2=b_input[(i+22)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+22)]);
      ;
      x1=a_input[(i+23)];
      x2=b_input[(i+23)];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[(i+23)]);
    }
    register int cbv_2, cbv_3;
    cbv_2=N-((N-(0))%24);
    cbv_3=N-1;
#pragma ivdep
#pragma vector always
    for (i=cbv_2; i<=cbv_3; i=i+1) {
            float64 result;
;
      x1=a_input[i];
      x2=b_input[i];
      result=float64_add(x1,x2);
      main_result=main_result+(result==z_output[i]);
    }
  }
/*@ end @*/
```

C.4 DFMUL

Listing C.10: Annotated DFMUL code

```
/*@ begin Loop (
    transform Composite(
      unrolljam = (['i'],[UF]),
      vector = (VEC, ['ivdep','vector always'])
     )
    for (i = 0; i <= N-1; i=i+1)
      {
        float64 result;
        x1 = a_input[i];
        x2 = b_input[i];
        result = float64_mul (x1, x2);
        main_result += (result == z_output[i]);
      }
) @*/
    for (i = 0; i <= N-1; i=i+1)
      {
        float64 result;
        x1 = a_input[i];
        x2 = b_input[i];
        result = float64_mul (x1, x2);
        main_result += (result == z_output[i]);
      }
/*@ end @*/
```

Listing C.11: DFMUL Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,20);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0', '-O1','-O2','-O3'];
   constraint divisible_by_two = (UF % 8 == 0);
 }
 def input_params {
   param N[] = [20];
 }
 def input_vars {
   decl static int a_input[N] = random;
   decl static int b_input[N] = random;
   decl static int z_input[N] = random;
 }
```

```
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.12: Tuned and transformed DFMUL code after S2S transformation by Orio

```
/**-- (Generated by Orio)
Best performance cost:
  [6.41e-07, 2.77e-07, 2.5e-07, 2.49e-07, 2.48e-07, 2.49e-07, 2.53
      e-07, 2.53e-07, 2.48e-07, 2.49e-07, 2.5e-07, 2.5e-07, 2.5e
      -07, 2.49e-07, 2.47e-07, 2.52e-07, 2.5e-07, 2.51e-07, 2.5e
      -07, 2.48e-07, 2.51e-07, 2.49e-07, 2.5e-07, 2.49e-07, 2.49e
      -07, 2.48e-07, 2.46e-07, 2.5e-07, 2.48e-07, 2.48e-07, 2.49e
      -07, 2.47e-07, 2.5e-07, 2.5e-07, 2.51e-07, 2.52e-07, 2.48e
      -07, 2.48e-07, 2.5e-07, 2.48e-07, 2.48e-07, 2.52e-07, 2.49e
      -07, 2.49e-07, 2.46e-07, 2.5e-07, 2.49e-07, 2.46e-07, 2.49e
      -07, 2.46e-07, 2.52e-07, 2.51e-07, 2.5e-07, 2.51e-07, 2.5e
      -07, 2.49e-07, 2.48e-07, 2.48e-07, 2.47e-07, 2.5e-07, 2.48e
      -07, 2.47e-07, 2.5e-07, 2.49e-07, 2.52e-07, 2.49e-07, 2.5e
      -07, 2.49e-07, 2.5e-07, 2.5e-07, 2.46e-07, 2.49e-07, 2.47e
      -07, 2.51e-07, 2.49e-07, 2.5e-07, 2.49e-07, 2.5e-07, 2.48e
      -07, 2.49e-07, 2.52e-07, 2.48e-07, 2.5e-07, 2.5e-07, 2.5e
      -07, 2.48e-07, 2.49e-07, 2.5e-07, 2.49e-07, 2.5e-07, 2.5e
      -07, 2.53e-07, 2.5e-07, 2.52e-07, 2.47e-07, 2.51e-07, 2.66e
      -07, 2.44e-07, 2.52e-07, 2.49e-07]
Tuned for specific problem sizes:
  N = 20
Best performance parameters:
  CFLAGS = -O1
  UF = 16
  VEC = True
--**/
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
      for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          x1 = a_input[i];
          x2 = b_input[i];
          result = float64_mul (x1, x2);
          main_result += (result == z_output[i]);
        }
  ) @*/
  {
    int i;
    register int cbv_1;
    cbv_1=N-16;
#pragma ivdep
#pragma vector always
    for (i=0; i<=cbv_1; i=i+16) {
```

```
            float64 result;
;
      x1=a_input[i];
      x2=b_input[i];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[i]);
      ;
      x1=a_input[(i+1)];
      x2=b_input[(i+1)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+1)]);
      ;
      x1=a_input[(i+2)];
      x2=b_input[(i+2)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+2)]);
      ;
      x1=a_input[(i+3)];
      x2=b_input[(i+3)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+3)]);
      ;
      x1=a_input[(i+4)];
      x2=b_input[(i+4)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+4)]);
      ;
      x1=a_input[(i+5)];
      x2=b_input[(i+5)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+5)]);
      ;
      x1=a_input[(i+6)];
      x2=b_input[(i+6)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+6)]);
      ;
      x1=a_input[(i+7)];
      x2=b_input[(i+7)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+7)]);
      ;
      x1=a_input[(i+8)];
      x2=b_input[(i+8)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+8)]);
      ;
      x1=a_input[(i+9)];
      x2=b_input[(i+9)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+9)]);
      ;
      x1=a_input[(i+10)];
      x2=b_input[(i+10)];
```

```
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+10)]);
      ;
      x1=a_input[(i+11)];
      x2=b_input[(i+11)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+11)]);
      ;
      x1=a_input[(i+12)];
      x2=b_input[(i+12)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+12)]);
      ;
      x1=a_input[(i+13)];
      x2=b_input[(i+13)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+13)]);
      ;
      x1=a_input[(i+14)];
      x2=b_input[(i+14)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+14)]);
      ;
      x1=a_input[(i+15)];
      x2=b_input[(i+15)];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[(i+15)]);
    }
    register int cbv_2, cbv_3;
    cbv_2=N-((N-(0))%16);
    cbv_3=N-1;
#pragma ivdep
#pragma vector always
    for (i=cbv_2; i<=cbv_3; i=i+1) {
            float64 result;
;
      x1=a_input[i];
      x2=b_input[i];
      result=float64_mul(x1,x2);
      main_result=main_result+(result==z_output[i]);
    }
  }
  /*@ end @*/
```

C.5 DFSIN

Listing C.13: Annotated DFSIN code

```
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
```

```
      for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          result = dfsin (test_in[i]);
          main_result += (result == test_out[i]);
        }
  ) @*/
      for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          result = dfsin (test_in[i]);
          main_result += (result == test_out[i]);
        }
  /*@ end @*/
```

Listing C.14: DFSIN Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,36);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0', '-O1','-O2','-O3'];
   constraint divisible_by_two = (UF % 8 == 0);
 }
 def input_params {
   param N[] = [36];
 }
 def input_vars {
   decl static int test_in[36] = random;
   decl static int test_out[36] = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.15: Tuned and transformed DFSIN code after S2S transformation by Orio

```
  @*/
/**-- (Generated by Orio)
Best performance cost:
  [7.38e-07, 2.54e-07, 2.46e-07, 2.49e-07, 2.48e-07, 2.48e-07,
      2.48e-07, 2.5e-07, 2.5e-07, 2.68e-07, 2.52e-07, 2.5e-07,
      2.46e-07, 2.49e-07, 2.52e-07, 2.48e-07, 2.5e-07, 2.49e-07,
      2.5e-07, 2.52e-07, 2.49e-07, 2.47e-07, 2.48e-07, 2.47e-07,
      2.48e-07, 2.47e-07, 2.53e-07, 2.5e-07, 2.47e-07, 2.47e-07,
```

```
      2.51e-07, 2.47e-07, 2.48e-07, 2.49e-07, 2.48e-07, 2.48e-07,
      2.47e-07, 2.5e-07, 2.45e-07, 2.52e-07, 2.47e-07, 2.69e-07,
      2.48e-07, 2.49e-07, 2.46e-07, 2.48e-07, 2.49e-07, 2.5e-07,
      2.48e-07, 2.54e-07, 2.53e-07, 2.53e-07, 2.49e-07, 2.49e-07,
      2.5e-07, 2.49e-07, 2.51e-07, 2.51e-07, 2.48e-07, 2.47e-07,
      2.48e-07, 2.51e-07, 2.5e-07, 2.49e-07, 2.47e-07, 2.49e-07,
      2.47e-07, 2.47e-07, 2.52e-07, 2.5e-07, 2.48e-07, 2.49e-07,
      2.5e-07, 2.52e-07, 2.47e-07, 2.49e-07, 2.5e-07, 2.46e-07,
      2.49e-07, 2.48e-07, 2.51e-07, 2.49e-07, 2.46e-07, 2.47e-07,
      2.5e-07, 2.48e-07, 2.5e-07, 2.53e-07, 2.48e-07, 2.5e-07, 2.5
      e-07, 2.5e-07, 2.5e-07, 2.47e-07, 2.49e-07, 2.47e-07, 2.5e
      -07, 2.48e-07, 2.5e-07, 2.49e-07]
Tuned for specific problem sizes:
  N = 36
Best performance parameters:
  CFLAGS = -O3
  UF = 24
  VEC = True
--**/
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
      for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          result = dfsin (test_in[i]);
          main_result += (result == test_out[i]);
        }
  ) @*/
  {
    int i;
    register int cbv_1;
    cbv_1=N-24;
#pragma ivdep
#pragma vector always
    for (i=0; i<=cbv_1; i=i+24) {
            float64 result;
;
      result=dfsin(test_in[i]);
      main_result=main_result+(result==test_out[i]);
      ;
      result=dfsin(test_in[(i+1)]);
      main_result=main_result+(result==test_out[(i+1)]);
      ;
      result=dfsin(test_in[(i+2)]);
      main_result=main_result+(result==test_out[(i+2)]);
      ;
      result=dfsin(test_in[(i+3)]);
      main_result=main_result+(result==test_out[(i+3)]);
      ;
      result=dfsin(test_in[(i+4)]);
      main_result=main_result+(result==test_out[(i+4)]);
```

```
;
result=dfsin(test_in[(i+5)]);
main_result=main_result+(result==test_out[(i+5)]);
;
result=dfsin(test_in[(i+6)]);
main_result=main_result+(result==test_out[(i+6)]);
;
result=dfsin(test_in[(i+7)]);
main_result=main_result+(result==test_out[(i+7)]);
;
result=dfsin(test_in[(i+8)]);
main_result=main_result+(result==test_out[(i+8)]);
;
result=dfsin(test_in[(i+9)]);
main_result=main_result+(result==test_out[(i+9)]);
;
result=dfsin(test_in[(i+10)]);
main_result=main_result+(result==test_out[(i+10)]);
;
result=dfsin(test_in[(i+11)]);
main_result=main_result+(result==test_out[(i+11)]);
;
result=dfsin(test_in[(i+12)]);
main_result=main_result+(result==test_out[(i+12)]);
;
result=dfsin(test_in[(i+13)]);
main_result=main_result+(result==test_out[(i+13)]);
;
result=dfsin(test_in[(i+14)]);
main_result=main_result+(result==test_out[(i+14)]);
;
result=dfsin(test_in[(i+15)]);
main_result=main_result+(result==test_out[(i+15)]);
;
result=dfsin(test_in[(i+16)]);
main_result=main_result+(result==test_out[(i+16)]);
;
result=dfsin(test_in[(i+17)]);
main_result=main_result+(result==test_out[(i+17)]);
;
result=dfsin(test_in[(i+18)]);
main_result=main_result+(result==test_out[(i+18)]);
;
result=dfsin(test_in[(i+19)]);
main_result=main_result+(result==test_out[(i+19)]);
;
result=dfsin(test_in[(i+20)]);
main_result=main_result+(result==test_out[(i+20)]);
;
result=dfsin(test_in[(i+21)]);
main_result=main_result+(result==test_out[(i+21)]);
;
result=dfsin(test_in[(i+22)]);
main_result=main_result+(result==test_out[(i+22)]);
```

```
      ;
      result=dfsin(test_in[(i+23)]);
      main_result=main_result+(result==test_out[(i+23)]);
    }
    register int cbv_2, cbv_3;
    cbv_2=N-((N-(0))%24);
    cbv_3=N-1;
#pragma ivdep
#pragma vector always
    for (i=cbv_2; i<=cbv_3; i=i+1) {
            float64 result;
;
      result=dfsin(test_in[i]);
      main_result=main_result+(result==test_out[i]);
    }
  }
  /*@ end @*/
```

C.6 DFDIV

Listing C.16: Annotated DFDIV code

```
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
      for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          x1 = a_input[i];
          x2 = b_input[i];
          result = float64_div (x1, x2);
          main_result += (result == z_output[i]);
        }
  ) @*/
      for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          x1 = a_input[i];
          x2 = b_input[i];
          result = float64_div (x1, x2);
          main_result += (result == z_output[i]);
        }
  /*@ end @*/
```

Listing C.17: DFDIV Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
```

```
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,22);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0', '-O1','-O2','-O3'];
   constraint divisible_by_two = (UF % 8 == 0);
 }
 def input_params {
   param N[] = [22];
 }
 def input_vars {
   decl static int a_input[N] = random;
   decl static int b_input[N] = random;
   decl static int z_input[N] = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.18: Tuned and transformed DFDIV code after S2S transformation by Orio

```
/**-- (Generated by Orio)
Best performance cost:
  [6.76e-07, 2.66e-07, 2.54e-07, 2.54e-07, 2.52e-07, 2.51e-07,
      2.53e-07, 2.53e-07, 2.52e-07, 2.53e-07, 2.51e-07, 2.56e-07,
      2.5e-07, 2.54e-07, 2.53e-07, 2.5e-07, 2.53e-07, 2.5e-07,
      2.51e-07, 2.52e-07, 2.53e-07, 2.52e-07, 2.52e-07, 2.5e-07,
      2.49e-07, 2.52e-07, 2.52e-07, 2.53e-07, 2.5e-07, 2.52e-07,
      2.5e-07, 2.47e-07, 2.55e-07, 2.54e-07, 2.52e-07, 2.52e-07,
      2.54e-07, 2.52e-07, 2.49e-07, 2.5e-07, 2.5e-07, 2.5e-07,
      2.52e-07, 2.48e-07, 2.51e-07, 2.49e-07, 2.52e-07, 2.5e-07,
      2.5e-07, 2.51e-07, 2.51e-07, 2.85e-07, 2.52e-07, 2.51e-07,
      2.51e-07, 2.5e-07, 2.54e-07, 2.49e-07, 2.51e-07, 2.58e-07,
      2.53e-07, 2.53e-07, 2.54e-07, 2.53e-07, 2.5e-07, 2.5e-07,
      2.53e-07, 2.53e-07, 2.49e-07, 2.5e-07, 2.53e-07, 2.53e-07,
      2.51e-07, 2.54e-07, 2.5e-07, 2.5e-07, 2.5e-07, 2.51e-07,
      2.55e-07, 2.53e-07, 2.53e-07, 2.53e-07, 2.54e-07, 2.54e-07,
      2.52e-07, 2.51e-07, 2.51e-07, 2.51e-07, 2.53e-07, 2.54e-07,
      2.5e-07, 2.5e-07, 2.5e-07, 2.52e-07, 2.48e-07, 2.5e-07, 2.48
      e-07, 2.53e-07, 2.49e-07, 2.5e-07]
Tuned for specific problem sizes:
  N = 22
Best performance parameters:
  CFLAGS = -O2
  UF = 16
  VEC = True
--**/
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
```

```
        vector = (VEC, ['ivdep','vector always'])
       )
      for (i = 0; i <= N-1; i=i+1)
        {
          float64 result;
          x1 = a_input[i];
          x2 = b_input[i];
          result = float64_div (x1, x2);
          main_result += (result == z_output[i]);
        }
  ) @*/
  {
    int i;
    register int cbv_1;
    cbv_1=N-16;
#pragma ivdep
#pragma vector always
    for (i=0; i<=cbv_1; i=i+16) {
            float64 result;
;
      x1=a_input[i];
      x2=b_input[i];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[i]);
      ;
      x1=a_input[(i+1)];
      x2=b_input[(i+1)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+1)]);
      ;
      x1=a_input[(i+2)];
      x2=b_input[(i+2)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+2)]);
      ;
      x1=a_input[(i+3)];
      x2=b_input[(i+3)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+3)]);
      ;
      x1=a_input[(i+4)];
      x2=b_input[(i+4)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+4)]);
      ;
      x1=a_input[(i+5)];
      x2=b_input[(i+5)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+5)]);
      ;
      x1=a_input[(i+6)];
      x2=b_input[(i+6)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+6)]);
```

```
      ;
      x1=a_input[(i+7)];
      x2=b_input[(i+7)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+7)]);
      ;
      x1=a_input[(i+8)];
      x2=b_input[(i+8)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+8)]);
      ;
      x1=a_input[(i+9)];
      x2=b_input[(i+9)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+9)]);
      ;
      x1=a_input[(i+10)];
      x2=b_input[(i+10)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+10)]);
      ;
      x1=a_input[(i+11)];
      x2=b_input[(i+11)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+11)]);
      ;
      x1=a_input[(i+12)];
      x2=b_input[(i+12)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+12)]);
      ;
      x1=a_input[(i+13)];
      x2=b_input[(i+13)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+13)]);
      ;
      x1=a_input[(i+14)];
      x2=b_input[(i+14)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+14)]);
      ;
      x1=a_input[(i+15)];
      x2=b_input[(i+15)];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[(i+15)]);
    }
    register int cbv_2, cbv_3;
    cbv_2=N-((N-(0))%16);
    cbv_3=N-1;
#pragma ivdep
#pragma vector always
    for (i=cbv_2; i<=cbv_3; i=i+1) {
            float64 result;
;
```

```
      x1=a_input[i];
      x2=b_input[i];
      result=float64_div(x1,x2);
      main_result=main_result+(result==z_output[i]);
    }
  }
  /*@ end @*/
```

C.7 GSM

Listing C.19: Annotated GSM code

```
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
      for (i = 0; i <= N-1; i++)
        so[i] = inData[i];
  ) @*/

      for (i = 0; i <= N-1; i++)
        so[i] = inData[i];

  /*@ end @*/
```

Listing C.20: GSM Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,10);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0', '-O1','-O2','-O3'];
 }
 def input_params {
   param N[] = [160];
   param M[] = [8];
 }
 def input_vars {
   decl dynamic int inData[160] = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.21: Tuned and transformed GSM code after S2S transformation by Orio

```
/**-- (Generated by Orio)
Best performance cost:
  [7.74e-07, 2.85e-07, 2.66e-07, 2.64e-07, 2.68e-07, 2.67e-07,
      2.68e-07, 2.65e-07, 2.7e-07, 2.65e-07, 2.67e-07, 2.66e-07,
      2.68e-07, 2.67e-07, 2.66e-07, 2.62e-07, 2.63e-07, 2.66e-07,
      2.66e-07, 2.66e-07, 2.66e-07, 2.67e-07, 2.66e-07, 2.66e-07,
      2.67e-07, 2.69e-07, 2.84e-07, 2.68e-07, 2.68e-07, 2.65e-07,
      2.66e-07, 2.68e-07, 2.65e-07, 2.66e-07, 2.69e-07, 2.66e-07,
      2.67e-07, 2.67e-07, 2.66e-07, 2.67e-07, 2.68e-07, 2.67e-07,
      2.68e-07, 2.67e-07, 2.64e-07, 2.64e-07, 2.66e-07, 2.67e-07,
      2.67e-07, 2.67e-07, 2.69e-07, 2.66e-07, 2.63e-07, 2.67e-07,
      2.66e-07, 2.65e-07, 2.68e-07, 2.68e-07, 2.65e-07, 2.7e-07,
      2.66e-07, 2.66e-07, 2.66e-07, 2.64e-07, 2.68e-07, 2.69e-07,
      2.67e-07, 2.68e-07, 2.66e-07, 2.66e-07, 2.66e-07, 2.69e-07,
      2.64e-07, 2.66e-07, 2.66e-07, 2.7e-07, 2.66e-07, 2.69e-07,
      2.7e-07, 2.68e-07, 2.67e-07, 2.64e-07, 2.66e-07, 2.65e-07,
      2.69e-07, 2.67e-07, 2.68e-07, 2.65e-07, 2.7e-07, 2.67e-07,
      2.68e-07, 2.66e-07, 2.68e-07, 2.66e-07, 2.66e-07, 2.67e-07,
      2.67e-07, 2.67e-07, 2.68e-07, 2.69e-07]
Tuned for specific problem sizes:
  M = 8
  N = 160
Best performance parameters:
  CFLAGS = -O1
  UF = 6
  VEC = False
--**/
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
      for (i = 0; i <= N-1; i++)
        so[i] = inData[i];
  ) @*/
  {
    int i;
    for (i=0; i<=N-6; i=i+6) {
      so[i]=inData[i];
      so[(i+1)]=inData[(i+1)];
      so[(i+2)]=inData[(i+2)];
      so[(i+3)]=inData[(i+3)];
      so[(i+4)]=inData[(i+4)];
      so[(i+5)]=inData[(i+5)];
    }
    for (i=N-((N-(0))%6); i<=N-1; i=i+1)
      so[i]=inData[i];
  }
  /*@ end @*/
```

C.8 MIPS

Listing C.22: Annotated MIPS code

```
/*@ begin Loop (
    transform Composite(
      unrolljam = (['i'],[UF]),
      vector = (VEC, ['ivdep','vector always'])
     )
    for (i = 0; i <= N-1; i=i+1)
      {
        reg[i] = 0;
      }
) @*/
    for (i = 0; i <= N-1; i=i+1)
      {
        reg[i] = 0;
      }
/*@ end @*/
```

Listing C.23: MIPS Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,10);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0', '-O1','-O2','-O3'];
 }
 def input_params {
   param N[] = [32];
 }
 def input_vars {
   decl dynamic int reg[32] = random;
   decl dynamic int out_key[5200] = random;
   decl int i;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.24: Tuned and transformed MIPS code after S2S transformation by Orio

```
/**-- (Generated by Orio)
Best performance cost:
  [0.004, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Tuned for specific problem sizes:
  N = 32
Best performance parameters:
  CFLAGS = -O0
  UF = 8
  VEC = False
--**/
  /*@ begin Loop (
      transform Composite(
        unrolljam = (['i'],[UF]),
        vector = (VEC, ['ivdep','vector always'])
       )
      for (i = 0; i <= N-1; i=i+1)
        {
          reg[i] = 0;
        }
  ) @*/
  {
    int i;
    for (i=0; i<=N-8; i=i+8) {
      reg[i]=0;
      reg[(i+1)]=0;
      reg[(i+2)]=0;
      reg[(i+3)]=0;
      reg[(i+4)]=0;
      reg[(i+5)]=0;
      reg[(i+6)]=0;
      reg[(i+7)]=0;
    }
    for (i=N-((N-(0))%8); i<=N-1; i=i+1)
      reg[i]=0;
  }
  /*@ end @*/
```

C.9 JPEG

Listing C.25: Annotated JPEG code

```
  /*@ begin Loop (
    transform Composite(
    unrolljam = (['i'],[UF]),
    vector = (VEC, ['ivdep','vector always'])
    )
    for(i=0; i<=RGB_NUM-1; i=i+1){
      for(j=0; j<=BMP_OUT_SIZE-1; j=j+1){
        if(OutData_comp_buf[i][j] == hana_bmp[i][j]){
          sum(1);
        }
      }
        }
  ) @*/
```

```
        for(i=0; i<=RGB_NUM-1; i=i+1){
                for(j=0; j<=BMP_OUT_SIZE-1; j=j+1){
                        if(OutData_comp_buf[i][j] == hana_bmp[i][j
                            ]){
                                sum(1);
                        }
                }
        }
   /*@ end @*/
```

Listing C.26: JPEG Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
 }
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,10);
   param VEC[] = [False,True];
   param CFLAGS[] = ['-O0', '-O1','-O2','-O3'];
 }
 def input_params {
   param BMP_OUT_SIZE[] = [5310];
   param RGB_NUM[] = [3];
 }
 def input_vars {
   decl static int OutData_comp_buf[RGB_NUM][BMP_OUT_SIZE] =
       random;
   decl static int hana_bmp[RGB_NUM][BMP_OUT_SIZE] = random;
   decl int main_result = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.27: Tuned and transformed JPEG code after S2S transformation by Orio

```
/**-- (Generated by Orio)
Best performance cost:
  [1.6925e-05, 1.778e-06, 1.564e-06, 1.566e-06, 1.528e-06, 1.524e
      -06, 1.531e-06, 1.483e-06, 1.479e-06, 1.473e-06, 1.47e-06,
      1.481e-06, 1.473e-06, 1.499e-06, 1.489e-06, 1.487e-06, 1.474
      e-06, 1.479e-06, 1.481e-06, 1.457e-06, 1.461e-06, 1.462e-06,
       1.428e-06, 1.447e-06, 1.465e-06, 1.449e-06, 1.446e-06,
      1.472e-06, 1.459e-06, 1.489e-06, 1.441e-06, 1.444e-06, 1.457
      e-06, 1.432e-06, 1.513e-06, 1.455e-06, 1.433e-06, 1.454e-06,
       1.433e-06, 1.442e-06, 1.459e-06, 1.482e-06, 1.456e-06, 1.45
```

```
      e-06, 1.464e-06, 1.436e-06, 1.438e-06, 1.476e-06, 1.429e-06,
       1.475e-06, 1.421e-06, 1.462e-06, 1.422e-06, 1.455e-06,
      1.404e-06, 1.414e-06, 1.424e-06, 1.434e-06, 1.448e-06, 1.446
      e-06, 1.449e-06, 1.433e-06, 1.426e-06, 1.41e-06, 1.396e-06,
      1.463e-06, 1.431e-06, 1.418e-06, 1.399e-06, 1.439e-06, 1.427
      e-06, 1.446e-06, 1.478e-06, 1.442e-06, 1.448e-06, 1.454e-06,
       1.444e-06, 1.424e-06, 1.463e-06, 1.447e-06, 1.425e-06,
      1.414e-06, 1.428e-06, 1.392e-06, 1.359e-06, 1.441e-06, 1.473
      e-06, 1.454e-06, 1.389e-06, 1.444e-06, 1.413e-06, 1.421e-06,
       1.431e-06, 1.4e-06, 1.41e-06, 1.441e-06, 1.462e-06, 1.428e
      -06, 1.4e-06, 1.429e-06]
Tuned for specific problem sizes:
  BMP_OUT_SIZE = 5310
  RGB_NUM = 3
Best performance parameters:
  CFLAGS = -O1
  UF = 2
  VEC = False
--**/
  /*@ begin Loop (
    transform Composite(
    unrolljam = (['i'],[UF]),
    vector = (VEC, ['ivdep','vector always'])
    )
    for(i=0; i<=RGB_NUM-1; i=i+1){
      for(j=0; j<=BMP_OUT_SIZE-1; j=j+1){
        if(OutData_comp_buf[i][j] == hana_bmp[i][j]){
          sum(1);
        }
      }
    }
  ) @*/
 {
   int i;
   for (i=0; i<=RGB_NUM-2; i=i+2) {
     for (j=0; j<=BMP_OUT_SIZE-1; j=j+1) {
       if (OutData_comp_buf[i][j]==hana_bmp[i][j]) {
         sum(1);
       }
       if (OutData_comp_buf[(i+1)][j]==hana_bmp[(i+1)][j]) {
         sum(1);
       }
     }
   }
   for (i=RGB_NUM-((RGB_NUM-(0))%2); i<=RGB_NUM-1; i=i+1)
     for (j=0; j<=BMP_OUT_SIZE-1; j=j+1) {
       if (OutData_comp_buf[i][j]==hana_bmp[i][j]) {
         sum(1);
       }
     }
 }
   /*@ end @*/
```

C.10 Matrix Multiplication

Listing C.28: Annotated Matrix Multiplication code

```
/*@ begin Loop (
                transform Composite(
                unrolljam = (['i'],[UF]),
                vector = (VEC, ['ivdep','vector always'])
                )
                for(i = 0; i <= SIZE-1; i=i+1) {
                        for(j = 0; j <= SIZE-1; j=j+1) {
                                count = count + multiply(i,j);
                        }
                }
        )
        @*/

        for(i = 0; i <= SIZE-1; i=i+1) {
                for(j = 0; j <= SIZE-1; j=j+1) {
                         count = count + multiply(i,j);
                 }
        }
/*@ end @*/
```

Listing C.29: Matrix Multiplication Orio specification code to use for S2S transformation

```
spec unroll_vectorize {
 def build {
   arg build_command = 'gcc␣-O0';
   arg libs = '-lrt';
}
 def performance_counter {
   arg method = 'basic␣timer';
   arg repetitions = 100;
 }
 def performance_params {
   param UF[] = range(1,20);
   param VEC[] = [False,True];
   param CFLAGS[] = ['O0','-O1','-O2','-O3'];
 }
 def input_params {
   param SIZE[] = [20];
 }
 def input_vars {
   decl dynamic int A1[SIZE][SIZE] = random;
   decl dynamic int B1[SIZE][SIZE] = random;
   decl dynamic int resultAB1[SIZE][SIZE] = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Listing C.30: Tuned and transformed Matrix Multiplication code after S2S transformation by Orio

```
/**-- (Generated by Orio)
Best performance cost:
  [9.05e-07, 3.87e-07, 3.31e-07, 3.24e-07, 3.26e-07, 3.3e-07, 3.29
      e-07, 3.27e-07, 3.3e-07, 3.31e-07, 3.28e-07, 3.34e-07, 3.3e
      -07, 3.27e-07, 3.29e-07, 3.3e-07, 3.29e-07, 3.28e-07, 3.26e
      -07, 3.26e-07, 3.3e-07, 3.29e-07, 3.28e-07, 3.3e-07, 3.3e
      -07, 3.29e-07, 3.5e-07, 3.33e-07, 3.3e-07, 3.32e-07, 3.3e
      -07, 3.31e-07, 3.28e-07, 3.33e-07, 3.29e-07, 3.27e-07, 3.26e
      -07, 3.29e-07, 3.27e-07, 3.3e-07, 3.29e-07, 3.29e-07, 3.29e
      -07, 3.31e-07, 3.27e-07, 3.28e-07, 3.3e-07, 3.25e-07, 3.28e
      -07, 3.29e-07, 3.27e-07, 3.27e-07, 3.29e-07, 3.29e-07, 3.29e
      -07, 3.28e-07, 3.31e-07, 3.29e-07, 3.32e-07, 3.27e-07, 3.29e
      -07, 3.31e-07, 3.29e-07, 3.28e-07, 3.28e-07, 3.29e-07, 3.32e
      -07, 3.29e-07, 3.29e-07, 3.3e-07, 3.32e-07, 3.31e-07, 3.29e
      -07, 3.28e-07, 3.28e-07, 3.31e-07, 3.29e-07, 3.3e-07, 3.3e
      -07, 3.3e-07, 3.28e-07, 3.29e-07, 3.29e-07, 3.31e-07, 3.31e
      -07, 3.3e-07, 3.27e-07, 3.29e-07, 3.3e-07, 3.34e-07, 3.28e
      -07, 3.31e-07, 3.3e-07, 3.29e-07, 3.27e-07, 3.27e-07, 3.26e
      -07, 3.3e-07, 3.28e-07, 3.3e-07]
Tuned for specific problem sizes:
  SIZE = 20
Best performance parameters:
  CFLAGS = -O3
  UF = 4
  VEC = True
--**/
   /*@ begin Loop (
                transform Composite(
                unrolljam = (['i'],[UF]),
                vector = (VEC, ['ivdep','vector always'])
                )
                for(i = 0; i <= SIZE-1; i=i+1) {
                        for(j = 0; j <= SIZE-1; j=j+1) {
                                count = count + multiply(i,j);
                        }
                }
        )
        @*/
      {
        int i;
        for (i=0; i<=SIZE-4; i=i+4) {
          register int cbv_1;
          cbv_1=SIZE-1;
#pragma ivdep
#pragma vector always
          for (j=0; j<=cbv_1; j=j+1) {
            count = count + multiply(i,j);
            count = count + multiply((i+1),j);
            count = count + multiply((i+2),j);
            count = count + multiply((i+3),j);
          }
```

```
        }
        for (i=SIZE-((SIZE-(0))%4); i<=SIZE-1; i=i+1) {
          register int cbv_2;
          cbv_2=SIZE-1;
#pragma ivdep
#pragma vector always
          for (j=0; j<=cbv_2; j=j+1) {
            count = count + multiply(i,j);
          }
        }
      }
  /*@ end @*/
```

C.11 Bash Scripts

In this section, you can find all our generated bash scripts to make for the automated part of the framework and can be expanded to have the fully automated framework as one of the future works.

Listing C.31: GUI and command line tool for the main chain tests

```
#!/bin/bash

RED='\033[0;31m'
NC='\033[0m' # No Color

INPUT="$1"

if [ -z $INPUT ]; then

        CODES="$(zenity␣--list␣--checklist␣--separator␣", "␣--text
            ␣"Please select the code(s) that you want to test:"␣--
            column␣"Select"␣--column␣"Code"␣FALSE␣adpcm␣FALSE␣aes␣
            FALSE␣blowfish␣FALSE␣gsm␣FALSE␣dfadd␣FALSE␣dfdiv␣FALSE
            ␣dfmul␣FALSE␣dfsin␣FALSE␣jpeg␣FALSE␣mips␣FALSE␣motion␣
            FALSE␣sha)"

        if [ "$?" -ne "0" ]; then
                echo -e "${RED}WARNING:${NC}␣You␣cannot␣run␣the␣
                    tool␣in␣graphical␣mode,␣please␣try␣again␣with␣
                    -c␣flag"
                exit 1
        fi

        TESTS="$(zenity␣--list␣--checklist␣--separator␣", "␣--text
            ␣"Please select the test(s) that you want to apply:"␣
            --column␣"Select"␣--column␣"Test"␣FALSE␣Hardware␣FALSE
            ␣Software)"

        IFS=', ' read -r -a NAMES <<< "$CODES"

        case $TESTS in
                Software)
```

```
                        TYPES="sw"
                        ;;
                Hardware)
                        TYPES="hw"
                        ;;
                Hardware*)
                        TYPES="swhw"
                        ;;
                * )
                        echo -e "${RED}WARNING:${NC}␣Test␣type␣is␣
                            wrong,␣please␣try␣again␣and␣choose␣the
                            ␣correct␣one."
                        exit 1
                        ;;
        esac

elif [ $INPUT = "-c" ]; then
        echo
        echo "Please␣insert␣the␣name␣of␣the␣codes␣that␣you␣want␣to
            ␣test."
        echo "(You␣can␣select␣single␣or␣multiple␣codes␣by␣using␣
            ',' separator␣like␣'adpcm'␣or␣'adpcm,gsm,sha')"
        echo "Available␣Codes:"
        echo "␣␣adpcm␣|␣aes␣|␣blowfish␣|␣dfadd␣|␣dfdiv␣|␣dfmul␣|␣
            dfsin␣|␣gsm␣|␣jpeg␣|␣mips␣|␣motion␣|␣sha"
        echo
        read -p "Insert␣Code␣name(s):␣" input

        echo
        echo "Please␣insert␣the␣type␣of␣test:"
        echo "(options:␣sw␣|␣hw␣|␣swhw)"
        read -p "Insert␣test␣type:␣" types

        IFS=',' read -r -a NAMES <<<"$input"

        case $types in
                sw)
                        TYPES="sw"
                        ;;
                hw)
                        TYPES="hw"
                        ;;
                swhw)
                        TYPES="swhw"
                        ;;
                * )
                        echo -e "${RED}WARNING:${NC}␣Test␣type␣is␣
                            wrong,␣please␣try␣again␣and␣choose␣the
                            ␣correct␣one."
                        exit 1
                        ;;
        esac
else
        echo "Please␣run␣the␣tool␣with␣the␣following␣option:"
```

```
        echo "␣␣-c␣␣␣␣␣␣to␣run␣in␣command␣line␣mode"
fi

CLEAN="make␣clean"
MAKE="make"
MAKEV="make␣v"
MAKESW="make␣sw"
MAKESWSIM="make␣swsim"

if [ ${#NAMES[@]} -eq 0 ]; then
        echo -e "${RED}WARNING:${NC}␣No␣code␣has␣been␣selected,␣
            please␣try␣again␣and␣choose␣at␣least␣one␣code."
        exit 1
fi

for NAME in "${NAMES[@]}"
do
        if [ $TYPES = "hw" ] || [ $TYPES = "swhw" ]; then
                # Test Original Code HW Test
                echo
                echo "******************************␣$NAME␣
                    Hardware␣Test␣
                    *********************************"
                FOLDER="cd␣$NAME/"
                echo "$(date)" >> $NAME/hardware_test_cycle.out
                echo "Hardware␣test␣of␣$NAME␣code␣has␣been␣started
                    ,␣please␣wait..."
                $($FOLDER; $CLEAN; $MAKE; $MAKEV | grep "Cycle" >>
                     hardware_test_cycle.out; echo >>
                    hardware_test_cycle.out)
                $($FOLDER; $CLEAN)
                echo "------------------------------␣OUTPUT␣HW␣
                    --------------------------------"
                cat "${NAME}/hardware_test_cycle.out"
                echo "*******************************␣END␣$NAME␣
                    HW␣*********************************"
                echo

                # Test Annotated Code HW Test
                echo "******************************␣_$NAME␣
                    Hardware␣Test␣
                    *********************************"
                FOLDER="cd␣_$NAME/"
                echo "$(date)" >> _$NAME/hardware_test_cycle.out
                echo "Hardware␣test␣of␣$NAME␣annotated␣code␣has␣
                    been␣started,␣please␣wait..."
                $($FOLDER; $CLEAN; $MAKE; $MAKEV | grep "Cycle" >>
                     hardware_test_cycle.out; echo >>
                    hardware_test_cycle.out)
                $($FOLDER; $CLEAN)
                echo "------------------------------␣OUTPUT␣HW␣
                    --------------------------------"
                cat "_${NAME}/hardware_test_cycle.out"
```

```
			echo "******************************␣END␣_$NAME␣
			    HW␣*********************************"
			echo
		fi

		if [ $TYPES = "sw" ] || [ $TYPES = "swhw" ]; then
			# Test Original Code SW Test
			echo
			echo "******************************␣$NAME␣
			    Software␣Test␣
			    *********************************"
			FOLDER="cd␣$NAME/"
			echo "$(date)" >> $NAME/software_test_cycle.out
			echo "Software␣test␣of␣$NAME␣code␣has␣been␣started
			    ,␣please␣wait..."
			$($FOLDER; $CLEAN; $MAKESW; $MAKESWSIM | grep "
			    counter␣=" >> software_test_cycle.out; echo >>
			     software_test_cycle.out)
			$($FOLDER; $CLEAN)
			echo "-------------------------------␣OUTPUT␣SW␣
			    ---------------------------------"
			cat "${NAME}/software_test_cycle.out"
			echo "******************************␣END␣$NAME␣
			    SW␣*********************************"
			echo

			# Test Annotated Code HW Test
			echo "******************************␣_$NAME␣
			    Software␣Test␣
			    *********************************"
			FOLDER="cd␣_$NAME/"
			echo "$(date)" >> _$NAME/software_test_cycle.out
			echo "Software␣test␣of␣$NAME␣annotated␣code␣has␣
			    been␣started,␣please␣wait..."
			$($FOLDER; $CLEAN; $MAKESW; $MAKESWSIM | grep "
			    counter␣=" >> software_test_cycle.out; echo >>
			     software_test_cycle.out)
			$($FOLDER; $CLEAN)
			echo "-------------------------------␣OUTPUT␣SW␣
			    ---------------------------------"
			cat "_${NAME}/software_test_cycle.out"
			echo "******************************␣END␣_$NAME␣
			    SW␣*********************************"
			echo
		fi
done

#awk -F'                 ' 'NF > 1 {print $2}'  gsm/
    hardware_test_cycle.out > hw_cycles.out
#awk -F'                ' 'NF > 1 {print $2}'  gsm/
    software_test_cycle.out > sw_cycles.out
```

Listing C.32: GProf profiler

```
#!/bin/bash
clear
echo "______.::␣$1␣Gprof␣Analysis␣::.______"
cd $1
ls *.c *.h
read -p "Select␣the␣main␣source␣code␣(without␣suffix)␣to␣profile:"
     file
gcc -Wall -pg -g -lm ${file}.c -o ${file}
echo "$file␣successfully␣compiled␣with␣debuging␣flag␣enabled"
chmod 755 $file
./$file
echo ""
echo "Profiling..."
gprof $file gmon.out > gprof_analysis.txt
echo "Generating␣call-graph␣in␣png"
gprof2dot gprof_analysis.txt > gprof_call_graph.dot
dot -Tgif gprof_call_graph.dot -o gprof_call_graph.png
echo "Done!"
```

Listing C.33: Perf profiler

```
#!/bin/bash
clear
echo "______.::␣$1␣Gprof␣Analysis␣::.______"
cd $1
ls *.c *.h
read -p "Select␣the␣main␣source␣code␣(without␣suffix)␣to␣profile:"
     file
gcc -Wall -pg -g -lm ${file}.c -o ${file}
echo "$file␣successfully␣compiled␣with␣debuging␣flag␣enabled"
chmod 755 $file
./$file
echo ""
echo "Profiling..."
gprof $file gmon.out > gprof_analysis.txt
echo "Generating␣call-graph␣in␣png"
gprof2dot gprof_analysis.txt > gprof_call_graph.dot
dot -Tgif gprof_call_graph.dot -o gprof_call_graph.png
echo "Done!"

legup@legup-vm:~/legup-4.0/examples/chstone/orio$ cat perf_r.sh
#!/bin/bash
clear
echo "______.::␣$1␣Perf␣Analysis␣::.______"
cd $1
ls *.c *.h
read -p "Select␣the␣main␣source␣code␣(without␣suffix)␣to␣profile:"
     file
gcc -Wall -pg -g -lm $file.c -o $file
echo "$file␣successfully␣compiled␣with␣debuging␣flag␣enabled"
chmod 755 $file
./$file
#perf record ./$file
perf record -g -s -d ./$file
```

```
perf report --stdio --sort comm,dso
#perf report
#perf top
#perf annotate
```

Listing C.34: Valgrind profiler

```
#!/bin/bash
clear
echo "______.:: $1 Gprof Analysis ::.______"
cd $1
ls *.c *.h
read -p "Select the main source code (without suffix) to profile:"
     file
gcc -Wall -pg -g -lm ${file}.c -o ${file}
echo "$file successfully compiled with debuging flag enabled"
chmod 755 $file
./$file
echo ""
echo "Profiling..."
gprof $file gmon.out > gprof_analysis.txt
echo "Generating call-graph in png"
gprof2dot gprof_analysis.txt > gprof_call_graph.dot
dot -Tgif gprof_call_graph.dot -o gprof_call_graph.png
echo "Done!"

legup@legup-vm:~/legup-4.0/examples/chstone/orio$ cat perf_r.sh
#!/bin/bash
clear
echo "______.:: $1 Perf Analysis ::.______"
cd $1
ls *.c *.h
read -p "Select the main source code (without suffix) to profile:"
     file
gcc -Wall -pg -g -lm $file.c -o $file
echo "$file successfully compiled with debuging flag enabled"
chmod 755 $file
./$file
#perf record ./$file
perf record -g -s -d ./$file
perf report --stdio --sort comm,dso
#perf report
#perf top
#perf annotate
legup@legup-vm:~/legup-4.0/examples/chstone/orio$ cat valprof.sh
#!/bin/bash
clear
echo "______.:: $1 Valgrind Analysis ::.______"
cd $1
ls *.c *.h
read -p "Select the main source code (without suffix) to profile:"
     file
gcc -Wall -pg -g -lm $file.c -o $file
echo "$file successfully compiled with debuging flag enabled"
```

```
chmod 755 $file
./$file
valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --
    collect-jumps=yes ./$file
kcachegrind
```

C.12 LLVM-opt Recipes for HLS

Table C.1: Three sets of favourable LLVM-opt flags for adpcm

adpcm
List1
-scalarrepl -instcombine -break-crit-edges -gvn -inline -lowerswitch -loop-rotate -early-cse -simplifycfg -tailduplicate -partial-inliner -sink -codegenprepare -jump-threading -indvars -licm -loop-unswitch -loop-simplify -loop-unroll -loop-deletion -block-placement -strip-nondebug -strip -simplify-libcalls -reassociate -lowerinvoke -lcssa -globalopt -functionattrs -adce -constmerge -correlated-propagation -dse -globaldce -loop-idiom -lower-expect -memcpyopt -scalarrepl-ssa -sccp -tailcallelim -scalarrepl -instcombine -break-crit-edges -gvn -inline -lowerswitch -early-cse -simplifycfg -tailduplicate -partial-inliner -sink -codegenprepare -jump-threading -indvars -licm -loop-unswitch -loop-simplify -loop-unroll
List 2
-loop-rotate -loop-reduce -scalarrepl -scalarrepl -scalarrepl-ssa -scalarrepl-ssa -instcombine -functionattrs -simplify-libcalls -inline -simplifycfg -globalopt -tailduplicate -sink -strip -sink -partial-inliner -break-crit-edges -licm -early-cse -inline -simplify-libcalls -globaldce -lowerinvoke -tailcallelim -functionattrs -reassociate -jump-threading -loop-reduce -indvars -loop-unroll -tailcallelim -simplify-libcalls -scalarrepl-ssa -functionattrs -inline -partial-inliner -lowerswitch -constmerge -globaldce -lowerinvoke -adce -dse -sccp -memcpyopt -strip-nondebug -globalopt -block-placement -loop-deletion -loop-unswitch -lcssa -gvn -instcombine -reassociate -instcombine -tailduplicate -simplifycfg -break-crit-edges -loop-simplify -codegenprepare -block-placement -jump-threading -licm -strip-nondebug -reassociate -loop-simplify -early-cse -loop-unroll -loop-deletion -loop-idiom -loop-unswitch -loop-rotate -lcssa -correlated-propagation -gvn -codegenprepare -break-crit-edges -licm -jump-threading -loop-simplify -early-cse -loop-unswitch -indvars -loop-rotate -lcssa
List 3
-simplifycfg -simplify-libcalls -loop-rotate -partial-inliner -break-crit-edges -loop-reduce -inline -constmerge -scalarrepl-ssa -functionattrs -scalarrepl -instcombine -partial-inliner -inline -sink -partial-inliner -scalarrepl-ssa -tailduplicate -scalarrepl -simplifycfg -lcssa -constmerge -licm -functionattrs -loop-reduce -sccp -loop-rotate -break-crit-edges -codegenprepare -early-cse -inline -simplify-libcalls -globaldce -scalarrepl -lowerinvoke -globaldce -lowerinvoke -tailcallelim -functionattrs -reassociate -loop-unroll -loop-deletion -loop-unswitch -indvars -gvn -jump-threading -sink -lcssa -correlated-propagation -loop-simplify -strip-nondebug -indvars -loop-reduce -globaldce -lowerinvoke -globalopt -block-placement -gvn -tailduplicate -simplifycfg -break-crit-edges -block-placement -jump-threading -codegenprepare -early-cse -instcombine -licm -block-placement -loop-deletion -memcpyopt -loop-unroll -sccp -correlated-propagation -loop-unswitch -loop-simplify -reassociate -strip-nondebug -instcombine -loop-idiom -reassociate -jump-threading -loop-unswitch -lcssa -licm -early-cse -loop-unroll -loop-deletion -loop-rotate -indvars -sink -gvn -strip-nondebug -loop-simplify -codegenprepare

Table C.2: Three sets of favourable LLVM-opt flags for blowfish

blowfish
List1
-simplify-libcalls -functionattrs -constmerge -functionattrs -simplifycfg -loop-unswitch -jump-threading -loop-simplify -early-cse -instcombine -loop-reduce -break-crit-edges -inline -indvars -scalarrepl-ssa -scalarrepl -scalarrepl-ssa -scalarrepl -loop-unroll -simplify-libcalls -scalarrepl-ssa -sink -loop-idiom -licm -codegenprepare -lcssa -reassociate -loop-reduce -break-crit-edges -loop-reduce -loop-rotate -break-crit-edges -jump-threading -strip-nondebug -globalopt -block-placement -loop-deletion -sink -globaldce -lowerinvoke -globaldce -lowerinvoke -globaldce -lowerinvoke -gvn -loop-unroll -tailduplicate -simplifycfg -sink -loop-unswitch -jump-threading -tailduplicate -block-placement -reassociate -early-cse -loop-idiom -instcombine -loop-deletion -reassociate -early-cse -strip-nondebug -licm -loop-rotate -loop-simplify -functionattrs -lcssa -gvn -loop-unroll -codegenprepare -loop-idiom -licm -instcombine -loop-unswitch -simplifycfg -loop-rotate -tailduplicate -strip-nondebug -block-placement -loop-simplify -codegenprepare -tailcallelim -lcssa -gvn
List 2
-simplifycfg -functionattrs -early-cse -instcombine -reassociate -reassociate -break-crit-edges -loop-unroll -loop-deletion -loop-idiom -indvars -loop-unswitch -loop-rotate -loop-simplify -inline -simplify-libcalls -jump-threading -dse -gvn -licm -functionattrs -inline -partial-inliner -block-placement -constmerge -loop-reduce -codegenprepare -simplifycfg -instcombine -tailduplicate -simplify-libcalls -strip-nondebug -break-crit-edges -loop-simplify -jump-threading -loop-rotate -loop-reduce -globalopt -globaldce -lowerinvoke
List 3
-simplify-libcalls -functionattrs -constmerge -functionattrs -simplifycfg -loop-unswitch -jump-threading -loop-simplify -early-cse -instcombine -loop-reduce -break-crit-edges -inline -indvars -scalarrepl-ssa -scalarrepl -scalarrepl-ssa -scalarrepl -loop-unroll -simplify-libcalls -scalarrepl-ssa -sink -loop-idiom -licm -codegenprepare -lcssa -reassociate -loop-reduce -break-crit-edges -loop-reduce -loop-rotate -break-crit-edges -jump-threading -strip-nondebug -globalopt -block-placement -loop-deletion -sink -globaldce -lowerinvoke -globaldce -lowerinvoke -globaldce -lowerinvoke -gvn -loop-unroll -tailduplicate -simplifycfg -sink -loop-unswitch -jump-threading -tailduplicate -block-placement -reassociate -early-cse -loop-idiom -instcombine -loop-deletion -reassociate -early-cse -strip-nondebug -licm -loop-rotate -loop-simplify -functionattrs -lcssa -gvn -loop-unroll -codegenprepare -loop-idiom -licm -instcombine -loop-unswitch -simplifycfg -loop-rotate -tailduplicate -strip-nondebug -block-placement -loop-simplify -codegenprepare -tailcallelim -lcssa -gvn

Table C.3: Three sets of favourable LLVM-opt flags for dfadd

dfadd
List1
-scalarrepl -instcombine -gvn -inline -lowerswitch -loop-rotate -early-cse -tailduplicate -partial-inliner -sink -codegenprepare -jump-threading -indvars -licm -loop-unswitch -loop-simplify -loop-unroll -loop-deletion -block-placement -strip-nondebug -strip -simplify-libcalls -reassociate -lowerinvoke -lcssa -globalopt -functionattrs -adce -constmerge -correlated-propagation -dse -globaldce -loop-idiom -lower-expect -memcpyopt -scalarrepl-ssa -sccp -tailcallelim -scalarrepl -instcombine -gvn -inline -lowerswitch -loop-rotate -early-cse -simplifycfg -tailduplicate -partial-inliner -sink -codegenprepare -jump-threading -indvars -licm -loop-unswitch -loop-simplify -loop-unroll -loop-deletion -block-placement -strip-nondebug -strip -simplify-libcalls -reassociate
List 2
-globalopt -scalarrepl -jump-threading -scalarrepl-ssa -loop-reduce -scalarrepl-ssa -break-crit-edges -jump-threading -sccp -simplifycfg -sccp -sccp -instcombine -correlated-propagation -early-cse -functionattrs -simplify-libcalls -simplify-libcalls -simplify-libcalls -inline -adce -tailcallelim -scalarrepl-ssa -scalarrepl -jump-threading -tailduplicate -partial-inliner -strip -partial-inliner -constmerge -lowerswitch -functionattrs -inline -reassociate -correlated-propagation -sink -loop-rotate -loop-rotate -loop-rotate -indvars -lcssa -loop-reduce -gvn -block-placement -strip -globalopt -partial-inliner -constmerge -lowerswitch -strip-nondebug -block-placement -tailduplicate -simplifycfg -break-crit-edges -codegenprepare -indvars -early-cse -instcombine -sink -reassociate -strip-nondebug -indvars -correlated-propagation -simplifycfg -tailduplicate -block-placement -instcombine -reassociate -lcssa -gvn -codegenprepare -early-cse -lcssa -gvn -lowerswitch -codegenprepare -strip-nondebug -globaldce -lowerinvoke -globaldce -lowerinvoke -globaldce -lowerinvoke
List 3
-loop-rotate -simplify-libcalls -scalarrepl -tailduplicate -globalopt -scalarrepl -sccp -break-crit-edges -simplifycfg -instcombine -loop-reduce -early-cse -inline -partial-inliner -scalarrepl-ssa -functionattrs -lowerswitch -simplify-libcalls -inline -adce -tailcallelim -scalarrepl-ssa -strip -globalopt -jump-threading -partial-inliner -reassociate -constmerge -correlated-propagation -loop-rotate -indvars -lcssa -gvn -adce -sink -lowerswitch -strip-nondebug -block-placement -simplifycfg -break-crit-edges -codegenprepare -indvars -early-cse -instcombine -tailduplicate -sink -reassociate -strip-nondebug -correlated-propagation -lcssa -gvn -codegenprepare -jump-threading -block-placement -globaldce -lowerinvoke -globaldce -lowerinvoke