DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Evaluating Unison's Speedup Estimation

An experimental study of the estimated speedup of code generated by Unison for the Hexagon DSP

MARTIN PERSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

KTH Information and
Communication Technology

Evaluating Unison's Speedup Estimation

An experimental study of the estimated speedup of code generated by Unison for the Hexagon DSP

MARTIN PERSSON

Master's Thesis at KTH Information and Communication Technology
Supervisor: Roberto Castañeda Lozano
Examiner: Christian Schulte

TRITA-ICT-EX-2017:75

Abstract

Unison is a code generator integrated into the LLVM compilation chain which solves the problem of instruction scheduling and register allocation in conjunction as a constraint optimization problem, producing possibly optimal code. The solution from LLVM of the very same problem acts as a baseline for the code generated by Unison, and along with an improved version of the code is an estimation of the speedup compared to the baseline given by Unison.

This thesis investigates the accuracy of this estimation for the Hexagon v4 DSP by executing the code produced by LLVM and Unison on a simulation platform. The results show that the estimations is not accurate, but there is a strong correlation between the given estimation and the observed speedup. The main reason for inaccuracy is hardware stalls of which the code produced by Unison suffers more of compared to the code produced by LLVM.

Referat

Unison är en kodgenerator integrerad i LLVMs kompileringskedja som löser problemen med instruktionsschemaläggning och registerallokering tillsammans som ett optimeringsproblem, vilket möjligen resulterar i optimal kod. Lösningen av LLVM för samma problem agerar måttstock för koden som Unison genererar, och tillsammans med en förbättrad version av koden ger Unison ifrån sig en uppskattning om hur mycket snabbare dess kod är.

Den här avhandligen undersöker precisionen för denna uppskattning för Hexagon v4 DSP genom exekvering av kod producerad av LLVM och Unison på en simuleringsplattform. Resultaten visar att upskattningen är oriktig, men att det är en stark korrelation mellan en given uppskattning och observerad förbättring. Den största orsaken till felaktigheter i uppskattningen beror på uppehåll i hårdvaran, vilket kod från Unison lider mer av jämfört med kod från LLVM.

Acknowledgments

I would like to thank Roberto Castañeda Lozano for all the guidance and help I have received during this project. I have learned so much and all your feedback has been priceless.

I am grateful to Christian Schulte for giving me the opportunity to do my thesis in such an interesting project and environment, and for encouragement when I needed it. Furthermore I would like to thank Martin Nilsson, Lars Rasmusson, Magnus Boman and Anders Holst for taking their time to discuss statistical analysis with me.

At last I would like to thank Filip Forsby and Alexander Östman for great support not just during the degree project, but for our time together at KTH.

Contents

Chapter 1

Introduction

1.1 Background

Compilers are programs that transform source code of a programming language into assembly code for a specific processor. Typically, compilers are structured into a front-end and a back-end. The front-end translates the source code into an intermediate representation (IR), which the back-end operates on to produce assembly code. The back-end has the responsibility of selecting and scheduling instructions corresponding to the IR, as well as allocating temporaries (program variables) to registers. Traditionally, these actions are performed in stages where heuristic algorithms are used to locally solve each stage with the object to minimize execution time or code size. This approach does not take the dependencies between the stages into account, which prevents it from generating optimal code. Unison, a result of a research collaboration between KTH and Swedish Institute of Computer Science (SICS), is a tool that solves the problem of instruction scheduling and register allocation in conjunction using combinatorial optimization, producing possibly optimal code.

Unison derives a constrained optimization problem from the generated IR, extended with selected instructions, together with a description of the target processor. The problem consists of allocating temporaries to registers and scheduling of instructions while respecting the semantics of the source code and the limitations of the processor, together with an optimization criterion.

The optimization criterion in Unison, when optimizing for performance, is to minimize a cost function based on latencies of instructions for the target processor and an estimation on how often the instructions will execute. The cost function does not capture the unpredictable features of processors such as varying memory access times or branch misprediction.

This project proposal builds on the work by Roberto Castañeda Lozano, described in his licentiate thesis [1]

1.2 Problem Statement

Experiments utilizing Unison to produce code for Hexagon, a digital signal processor included in Qualcomm's Snapdragon system-on-chip, estimates that the generated code could sometimes result in a speedup of as much as 40% compared with traditional approaches. However, due to the complexity of solving combinatorial problems, the compile time to generate code of same quality as traditional approaches grows in average quadratically with the size of the input program, and in the worst case exponentially.

The main problem investigated in this project is that speedup estimations given by Unison for its generated code is not experimentally verified, and the impact on the estimation from the unpredictable features of a processor is not studied. The estimation does not account for these unpredictable features which adds uncertainty about the accuracy of the estimation. Moreover, the estimation relies on static speculative information of how often certain parts of the code will execute, which could affect the estimated speedup as well. An experimental verification of the estimation will add confidence, and possibly reveal flaws in the estimation.

1.3 Goals

1. The main goal is to experimentally verify the estimation of the speedup given by Unison.

   This goal is considered successful if the actual speedup has been accurately compared to the estimation. In case of deviations between the estimations and the observed speedups, suggesting inaccuracies in the estimations, the project will continue with Goal 2.

2. Thoroughly study deviations to identify sources of errors and possibly develop a proposal for a solution. It will be considered successful if the sources of deviations are identified and their impact on the error is quantified.

Goal 1 is the mandatory goal of this thesis whereas Goal 2 is a desirable outcome.

1.4 Ethics and sustainability

From a sustainability point of view, it is desirable to have as efficient use of the hardware as possible, which Unison strives to achieve. By omitting instructions not relevant to the objective of the software and increasing the number of instructions per cycle, the need for faster and more power-hungry processors decreases. The contribution to environmental sustainability from this work is twofold:

1. By comparing run time measurements of the code generated by the compilers, an evaluation of the estimation by Unison can be made. Providing data from

experiments along the estimation could affect the interest and expansion of the project in a positive way.

2. Revealing factors that contribute to inaccuracy in the estimation is the first step in improving Unison, by presenting what should be modeled for better accuracy. This could result in a more accurate estimation and maybe most important, code of higher quality.

The work carried out in this project complies with the IEEE Code of Ethics [2], where the second item in the list is worth reflecting on.
"To avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist"
Unison is an outcome of the doctoral studies by Roberto Castañeda Lozano, who is also the supervisor for this thesis and blurs the line of being an independent study. Although it might appear to be a conflict of interest, the common goal is to validate or reveal flaws in Unison's estimation in order to improve the estimator.

1.5 Delimitations

The project described in this thesis is delimited to a specific processor, Hexagon V4. When compiling code for the target processor, only a single version of clang, LLVM and Unison is studied. Using benchmarks and experiments to conclude results implies a limitation in a sense that other benchmark suites might lead to slightly different results. This emphasizes the importance of having appropriate benchmarks for the target field of application, since it is not possible to capture all possible aspects of programs by experiments. The execution platform is delimited to the simulator provided in Hexagon SDK 2.0. The impact of adjusting the estimated execution frequencies of basic blocks is limited to a few case studies. The functions selected for compilation are restricted to functions containing less than 1000 instructions, due to the complexity of solving combinatorial problems.

1.6 Outline

The rest of this thesis is structured as follows:
Chapter 2 contains the theoretical background of the techniques and technologies applied in this thesis. Chapter 3 describes the methodology related to Goal 1, the process from compilation to data collection and definitions of variables used throughout this thesis. Chapter 4 presents the results related to Goal 1. Chapter 5 describes the methodology applied to study and quantifying the sources of divergence, related to Goal 2. Chapter 6 presents results related to Goal 2, where the impact of the identified sources of divergence have been eliminated. Chapter 7 summarizes the thesis with conclusions and suggestions for future work.

Chapter 2

Background

2.1 Compilers

A compiler is a program that transforms code written in a source language into a target language [3]. Two fundamental principles are defined which a compiler must respect:

- *"The compiler must preserve the meaning of the program being compiled"*
- *"The compiler must improve the input program in some discernible way"*

The semantics of the program described in the source language must be correctly reflected in the target language, and the output needs to have some additional value compared to the input.
A typical compiler is structured with a front-end which produces an internal Intermediate Representation (IR) of the program, an optimizer which improves the IR, and a back-end which transforms the IR into the target language [3, 4]. The target language is typically assembly code for a specific processor, which will be assumed for the rest of this document.

This separation of duties allows the same front-end to be used with different back-ends to target multiple processor, and vice-versa, the same back-end with different front-ends to support multiple languages. By having the optimizer running on the IR, the same optimizations can be performed independently of source/target language. However, source and target-dependent optimization might still be performed on the front- and back-end respectively.

The back-end of the compiler has the responsibility of selecting hardware instructions corresponding to the IR, (*Instruction Selection*), scheduling the instructions, (*Instruction Scheduling*), and deciding which data will reside in registers and which data will reside in memory, (*Register Allocation*) [3].

2.1.1 Instruction Selection

Instruction selection maps the code expressed in the compiler's IR form to the form of the target processor [3]. The instruction sets of modern processors often have multiple instructions which could implement a single statement in the IR and instructions which correspond to multiple statements with different *latencies.* E.g. to multiply a value with 2 could be achieved with either an ADD or an MUL instruction. Generally, an ADD instruction perform better, but depending on the state of the corresponding functional unit, busy or not, it is not always the case. The states of the functional units of the processor are not captured during this stage, but during instruction scheduling.

2.1.2 Instruction Scheduling

Instruction scheduling is the task of ordering instructions of a code sequence efficiently while respecting the semantics of the source program and taking the limitations of the hardware into account.

Processors typically issue instructions to the available functional units as quickly as possible, overlapping execution time of instructions to speed up the execution of a program [3]. A problem arises when subsequent instructions depend on a result from a previous instruction or when the functional unit is busy with a former instruction. Processors can either wait on the instruction causing the delay by stalling the dependent operation, reducing performance, or issue the instruction regardless, changing the semantics of the program. Regardless of which approach the processor uses to handle these situations, the issue is mitigated by efficient instruction scheduling.

Assume executing code to add three variables X, Y, Z which all reside in main memory on an imaginary processor with a load/store unit for memory transactions and an Arithmetic Logic Unit (ALU). The processor is capable of issuing one instruction every cycle and a load/store takes two cycles to finish whereas an ALU-operation takes one cycle. Table 2.1 shows a hypothetical schedule of code corresponding to this task, together with which temporary values, *temporaries*, are live at every program point.

What can be observed from the different ordering of the instructions is that:

(a) The scheduled code finishes one cycle ahead of the original code.

(b) The scheduled code requires usage of three registers instead of two in the original code; at Issue Cycle 5, the scheduled code has three live temporaries whose values must be preserved

Assuming there are sufficient registers available, the scheduled code finishes one cycle earlier than the original code. The scheduled code, however, puts additional pressure on the register allocator by introducing the need for an additional register. Typically, instruction scheduling is done using greedy algorithms [3].

Table 2.1: Difference between original code and scheduled code on an imaginary processor

Issue Cycle	Original Code	Live	Scheduled Code	Live
1	$t_1 \leftarrow$loadw, @M_X	t_1	$t_1 \leftarrow$loadw, @M_X	t_1
2		t_1		t_1
3	$t_2 \leftarrow$loadw, @M_Y	t_1,t_2	$t_2 \leftarrow$loadw, @M_Y	t_1,t_2
4		t_1,t_2		t_1,t_2
5	$t_3 \leftarrow$add, t_1,t_2	t_3	$t_4 \leftarrow$loadw, @M_Z	t_1,t_2,t_4
6	$t_4 \leftarrow$loadw, @M_Z	t_3,t_4	$t_3 \leftarrow$add, t_1,t_2	t_3,t_4
7		t_3,t_4	$t_5 \leftarrow$add, t_3,t_4	t_5
8	$t_5 \leftarrow$add, t_3,t_4	t_5		

2.1.3 Register Allocation

Register allocation is the task of assigning temporary values, *temporaries*, to a given set of registers by the target processor [3]. The registers in a processor are the fastest locations in the memory hierarchy, typically hundreds of times faster than main memory and multiple times faster than the cache memory, and efficient usage of these is crucial for the performance of the program [5]. The register allocator must decide on which temporary is stored in which register for each given program point. In cases where it is not possible to have all of the currently live temporaries reside in registers, the allocator resorts to *spilling*. Spilling is the operation of moving a temporary from register to memory by inserting spill code; instructions to store and load a temporary in memory. Spilling involves decisions of what temporary to spill to minimize the number of spilled temporaries and where to insert the spill code for minimal impact on the performance.

2.2 Constraint Programming

Constraint programming is an alternative approach of programming where the programmer is restricted to model a problem in terms of variables and constraints upon them [6]. The model is fed to a constraint solver which by **propagation** and **searching** tries to find a solution to the problem, a Constraint Satisfaction Problem (CSP). A solution to the problem consists of an assignment of the variables where all the constraints are satisfied.

An example of a simple problem, depicted in Figure 2.1 is utilized to explain the concepts.

- **Propagation** is a phase in the constraint solver where constraints are enforced upon the variable in order to remove assignments which are not part of any solution. A function used to enforce a specific constraint is called a propagator, and during the propagation phase all propagators are executed as needed

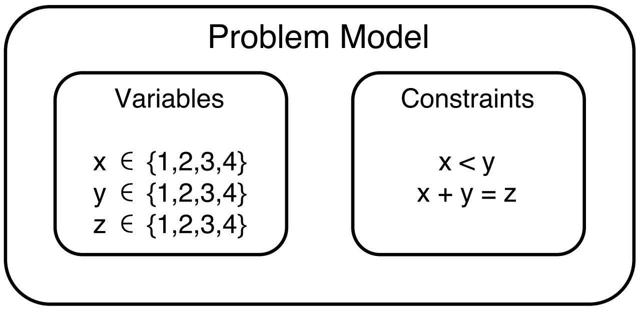

Figure 2.1: A simple Constraint Satisfaction Problem

until no further assignments can be removed. Table 2.2 depicts the reduction of possible assignments of variables in the example after propagation of the constraints.

Table 2.2: Applying propagation to example

Stage	x	y	z
Initial problem	{1,2,3,4}	{1,2,3,4}	{1,2,3,4}
After propagation of x<y	{1,2,3}	{2,3,4}	{1,2,3,4}
After propagation of x+y=z	{1,2}	{2,3}	{3,4}

- **Searching** is a phase employed by the the constraint solver when no further assignments can be removed from the variables in the model. Searching is typically achieved by trying out assignments for a variable, in order to trigger additional propagation [1]. Figure 2.2 visualizes the searching process applied to the example depicted in Figure 2.1 after the propagation described in Table 2.2. The solver resorts to searching in the topmost node, where it explores the assignment of $y = 2$ vs. $y \neq 2$. Depending on the order of which the solver explores this example, either of the assignments in the bottom most nodes could be returned as solution.

- **Constrained Optimization Problem** A CSP extended with a cost function to quantify the quality of the solution is called a Constrained Optimization Problem (COP). Whenever a solution is encountered by the solver, it will continue searching for other solution with the additional constraint that the solution must be of greater quality according to the cost function. Extending the same example with an optimization criteria to maximize the value of z, would yield the assignment in the rightmost solution in Figure 2.2 no matter in what order the solver explores the assignment during the search phase.

Instruction scheduling and register allocation are both combinatorial problems, which might lead to an explosion of nodes needed to be explored in the search

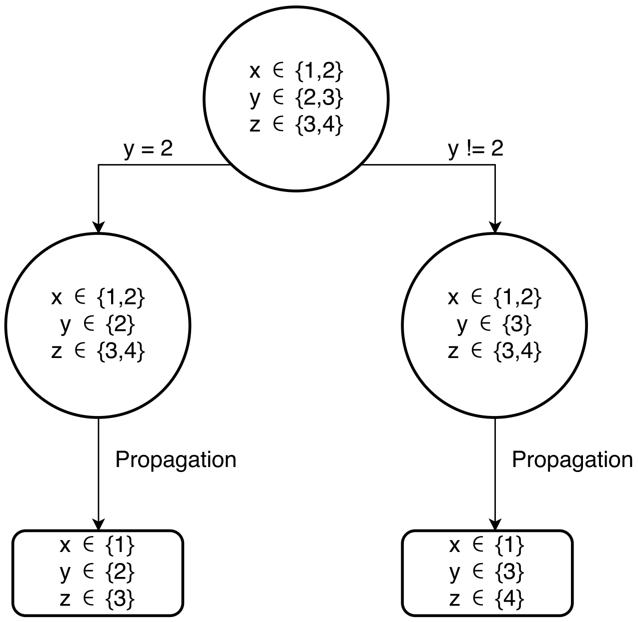

Figure 2.2: A search for a solution of the problem in Figure 2.1 after the propagation depicted in Table 2.2. Circular and rectangular nodes corresponds to intermediate stages and solutions respectively.

tree in order to find the optimal solution. In the worst case, the search tree grows exponentially with the number variables in the problem, which in this case is the number of temporaries and instructions in the input program.

2.3 LLVM Project

The LLVM Project is a collection of modular tools which together compose a complete compiler chain for common source languages and platforms [7]. Moreover, it provides a framework and libraries which enable development and customization throughout the chain. The framework is structured like a traditional compiler where a front-end for a source language translates the code to the intermediate representation, LLVM-IR, upon which independent optimizations are performed on [4].

LLVM features a target-independent code generator (back-end) which, together with a description of the target platform and the memory model used, transforms the LLVM-IR to the target language [8].

The code generator consists of multiple stages [8], depicted in Figure 2.3:

1. **Instruction selection** selects instructions of the target instruction set to efficiently express the input IR.

2. **Scheduling and formation** determines the order between the selected instructions.
3. **Machine code optimizations** performs various optimizations on the generated code from the scheduling stage.
4. **Register allocation** determines what data will be stored in which register for every given program point. In cases where the data is unable to fit in the registers, it determines what data to be stored on the stack in memory, *spilling*, and adds instructions for these memory transactions.
5. **Prologue/epilogue code insertion** add instructions used to maintain the software stack upon execution of subroutines.
6. **Late machine code optimizations** executes optimizations which operate on the final machine code.
7. **Code emission** outputs the generated code in the target assembler format.

Each stage consists of multiple passes, where a pass could have dependencies of other passes.

2.4 Unison

Unison is a code generator that models the problem of allocating registers to temporaries and instruction scheduling as a single *Constrained Optimization Problem.* This model is fed to the constraint solver provided in Gecode [9] which produces a possibly optimal solution to the unified problem of register allocation and instruction scheduling [10]. The input to Unison is a program expressed in the LLVM-Machine IR format (LLVM-MIR) [11] where [12]:

- instructions of the target processor have been selected,
- registers are not yet allocated and
- instructions are not yet scheduled.

While it is not a part of the LLVM-MIR format, the input file might contain statically estimated information about execution frequency for each basic block [12].

Unison takes as an optional input another file in LLVM-MIR format corresponding to the same source code, where instruction scheduling and register allocation already have been performed by LLVM. This file is used as a baseline for the solution which Unison produces. The interaction with LLVM is depicted in Figure 2.3 [10].

- **The modeling stage** consists of multiple passes preparing the input program to be used with the constraint solver. It creates a constraint model of the input program, respecting the limitations of the processor and the semantics of the program.

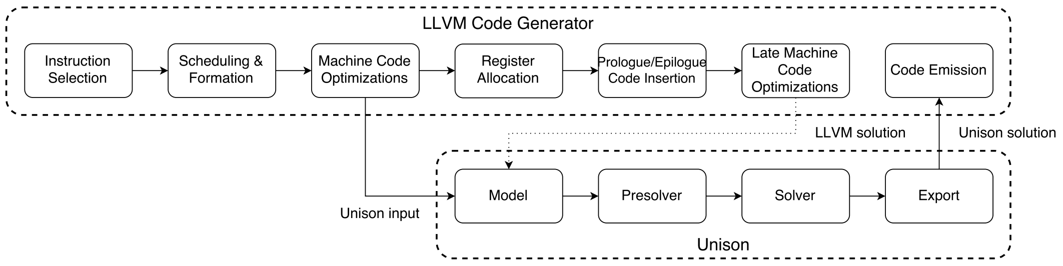

Figure 2.3: Unison integration with LLVM

- **The Presolver** improves the robustness of the solving process by reformulating the combinatorial problem [13, 14]. By generating implied constraints, removing redundant solutions and computing lower bounds for solutions it improves propagation and reduces the search tree for the solver.
- **The Solver** solves the combinatorial problem by propagation and exploring the search tree, Section 2.2. The cost of the solution created by LLVM acts as a baseline for the code quality, which the solution of Unison must improve. Once a solution is found it continues searching for a better solution until an optimal solution is found or a timeout has been reached. Once the final solution is decided, it is compared with the code produced by LLVM to give an estimated improvement of the code quality. Another possible outcome is that Unison proves absence of a better solution than the baseline, in which case the code from LLVM is exported.
- **Export** returns the solution to LLVM for assembly code emission.

Before returning the solution to LLVM, Unison gives an estimated improvement of the quality of code. The estimation is based on the cost function which Unison uses to compare solutions, compared with the solution from LLVM. When optimizing for speed the cost is calculated by [10]:

$$\sum_{bb \in \text{function}} \text{frequency}(bb) * \text{cost}(bb) \tag{2.1}$$

where frequency is the estimated executions frequency of a basic block (bb), given by LLVM, and cost is the maximum issue cycle relative to the beginning of the block of the instructions in a basic block.

The estimated code quality improvement is given as a ratio:

$$\text{Estimation} = \frac{solution_{LLVM}}{solution_{Unison}} - 1 \tag{2.2}$$

The code generated by Unison suffers from non-determinism in two different ways. Timeouts might result in code of different quality from run to run, depending on the platform and operating system used for compilation. The second reason for non-determinism is that Unison utilizes a parallel search strategy when exploring the search tree of the combinatorial problem. The code generated depends on which thread that is first to report the optimal or the solution with the lowest cost. While it might lead to different solution from run-to-run, the solutions are of the same quality according to the cost function.

2.5 Profiling

Profiling information contains an overall view of the execution of a program [15]. It may contain information about time/cycles spent in different sections of the program, call graphs to present which sections invokes other sections and even how many times each instruction is executed. Profiling information is utilized in this project to study the impact of which the estimated execution frequencies of basic blocks have on the estimation given by Unison.
There are two distinct techniques to create profiling information, Program Counter (PC)-sampling and instrumentation.

2.5.1 PC-sampling

PC-sampling relies on periodic interrupts to trigger an Interrupt Service Routine (ISR), which collects sample data of the execution. Each time the ISR is triggered, it collects information from the stack memory to determine the state of the program before entering the ISR. After execution an estimation of the portion spent in different sections of the program can be done by dividing the number of samples recorded towards a specific section by the total number of samples recorded during execution. A higher interrupt frequency or a longer execution of the program results in a more accurate estimation.

2.5.2 Instrumentation

Instrumentation involves extending each basic-block with code to count the number of executions of that particular basic block. Before termination the execution counters of the basic blocks is outputted which forms a histogram over executions per basic block. Instrumentation is typically more accurate than PC-sampling and also provides reproducible results due to it is not dependent on periodic timers [16]. Another advantage that comes with this additional accuracy is that the profiling information can be used for code coverage analysis and finding dead code.

2.6 Hexagon DSP

The Hexagon Digital Signal Processor (DSP) is a Very Long Instruction Word (VLIW) processor, optimized for mobile multimedia and communication [17]. A VLIW-processor is an in-order processor which can issue multiple instructions each cycle to one ore more execution units, encoded in the same instruction *bundle*. An in-order processor issues instructions in the order in which they appear in the instruction stream, and if an instruction results in a stall, no subsequent instructions are issued until the stall is resolved [5]. Hexagon features two load/store units for memory transactions and two 64 bits single-instruction, multiple-data (SIMD) execution units. The load/store units also support execution of 32-bit ALU instructions. Each instruction bundle may contain up to 4 instructions to be executed in the different units. In a VLIW-architecture the burden of exploiting instruction level parallelism (ILP) is on the compiler, by determining and scheduling independent instructions in the same bundle.

The Hexagon v4 features one processor which is shared by three hardware threads [17]. These threads are scheduled in a round-robin approach, where each thread is given opportunity to issue instructions every third clock cycle. The number of hardware threads is chosen to match the depth of the pipeline. This choice makes the previously issued instructions by a thread complete before a new instruction bundle is issued by the same thread.

The advantage of this design is that the compiler does not need to schedule latencies between dependent instructions. A downside to this approach is that when a hardware thread is idle, its time slot is unused.

Qualcomm's Snapdragon 800 System-on-Chip series, a processor widely used in smartphones, features two Hexagon DSPs. One dedicated for modem processing and one for multimedia acceleration [17].

2.6.1 Hexagon Simulator

The Hexagon SDK provides a simulator to execute code compiled for the Hexagon architecture [18]. The following features are implemented in the simulator:

- Execution profiling which creates a call graph of the execution of a program and the amount of CPU time spent in each function. The profiling employed in the simulator measures clock cycles rather than time units or samples, which provides more accurate run time information than a regular profiler [19].

- Execution statistics outputs collected statistics of the execution of a program. The statistics contain cycle/instruction counters, characteristics of instruction bundles, cache and stall statistics and bus traffic.

- Timer interface enables developers to clock program sections by making source code modifications to start, stop and read timers.

- Regional profiling enables a limited set of execution statistics to be counted separately for a region of interest in the code, defined by a memory address range.

- Timing modeling enables the simulator to model caches and processor stalls.

According to the user guide is the simulator cycle-approximate with an accuracy within 2% from cycle-accurate with a projected performance of less than 1% difference [18].

2.7 Benchmarking

This section describes concepts considered and applied in terms of benchmarking. Section 2.7.1 describes methods of subsetting benchmark suites while Section 2.7.2 summarizes a paper [20], describing the impact of which not commonly reported variables might have on benchmark results.

2.7.1 Benchmark selection

There exist methods to derive a smaller but still representative subset of benchmarks from a benchmark suite, in order to reduce the number of benchmarks to study [21]. By characterizing benchmarks by specific properties it is possible to replace multiple benchmarks with similar properties by a single benchmark. Benchmarks may be characterized in several ways:

Microarchitecture-dependent measurements

Microarchitecture-dependent measurements characterize programs by collecting run time data of an execution of a program [21, 22]. The data is typically collected by hardware performance counters which track hardware events occurring during execution. These events constitute the characteristics of the benchmark. Example of properties derived using this approach:

- **Mix of executed instructions** - the distribution between different types of instructions (branches, memory operations, integer/floating-point operations) as recorded during execution.
- **IPC** - the number of executed instructions per cycle.
- **Cache miss ratio** - the ratio of cache misses for data/instruction caches respectively.
- **Branch misprediction ratio** - the ratio of branch misprediction by the predictor.

This approach to capture the native properties of a benchmark might be misleading, due to the fact that these measurements are biased by the instruction set, the compiler used as well as the target hardware [22]. However, due to the availability of these performance counters in recent architectures, these measurements are often used since retrieving them adds no overhead [21].

Microarchitecture-independent measurements

Microarchitecture-independent measurements characterize programs by properties of the its corresponding assembly code [21, 23]. Examples of such properties is:

- **Instruction mix** - the distribution between different types of instructions, as they appear in the assembly code.
- **Basic block size** - the number of instructions in a basic block.
- **Data working set size** - the number of unique memory locations used in the data stream of the program.
- **Branch characterization** - the ratio of forward/backwards branches and/or the ratio of taken/not taken branches.

While microarchitecture-independent measurements are not biased by the target hardware, they are still influenced by the compiler and instruction set.

2.7.2 Omitted-Variable Bias

In [20] the authors describe the effect of Omitted-Variable Bias (OVB) when evaluating compiler optimization. OVB is the effect caused by variables not included in the result of a study which causes the results to not be reproducible or useful. They identify two variables which are commonly not presented in compiler optimizations studies, altering the memory layout of the executing program:

1. **Environment size** - The environment variables occupy size in memory for every program executing under an operating system providing this functionality. The size of the environment will have an impact on the alignment of stack variables.
2. **Link order** - The order in which object files are linked together alters the placement of the code in the resulting executable.

As an experiment they compare the speedup of benchmarks compiled with -O3 versus -O2 using **gcc** when changing the link order and altering the size of the environment for different benchmarks. The results show that the effect of altering these variables affects the outcome from the experiments significantly throughout multiple platforms with different compilers.

2.8 Statistical Analysis

This section describes the statistical tests employed in this thesis.

2.8.1 Normality tests

Several tests exist to whether or not a given set of data could stem from a normal distribution [24], out of which two are briefly presented below.

- **Kolmogorov-Smirnov Goodness-of-Fit test**
  The Kolmogorov-Smirnov Goodness-of-Fit (KS) test is based on a comparison between a set of ordered data and a fully specified distribution [25]. From the set of data, the empirical cumulative distribution function (ECDF) is derived, and the maximum distance between the ECDF and the specified distribution function is calculated. This distance is compared with a table of pre-calculated critical values depending on the size of the data set and significance level, to tell whether or not the given set of data could be an observation of the distribution. The null hypothesis that there is no significant difference between the data and the specified distribution is rejected if the maximum distance is greater than the critical value.

- **Anderson-Darling test**
  The Anderson-Darling (AD) test is a modification of the KS-test to test whether a sample could stem from a specified distribution [25]. The statistic variable used to test for a distribution depends on the distance between the ECDF and the distribution function of the specified distribution just like the KS-test [24]. Instead of using the maximum value it is based on the sum of the squared difference between the ECDF and the distribution function for the specified distribution to test for. It differs from the KS-test by giving more weight to the tails of the distributions.

A comparison of normality tests concluded the AD test was more powerful than the KS test by comparing their ability to reject the null hypotheses for non-normal distributions [24].

2.8.2 Spearmans Rank-Order Correlation

Spearmans rank correlation coefficient is a distribution-free measurement of a monotonic relationship between two variables [26, 27]. It does not require the variables to come from a normal distribution to be utilized [26], but requires the variable to be ordinal [28]. That is to be able to tell which of two observations of the same variable is the greater or if they are equal.

When calculating Spearmans rank correlation coefficient, the ranks are used rather than the raw values [27]. Let $(x_1, y_1), \dots, (x_n, y_n)$ be pairwise observations of variables X and Y where (x_i, y_i) are both retrieved from the same observation i. The rank of a variable is denoted $R(x_i)$ and corresponds to the index of x_i if all $x \in X$ were to be ordered in an ascending order. In case of ties, that is $x_i = x_j$ for $i \neq j$ the rank of x_i and x_j will both be the mean of the relevant ranks.

Table 2.3: Spearman correlation coefficient for a population of n=3

$R(X)$	$R(Y_1)$	$R(Y_2)$	$R(Y_3)$	$R(Y_4)$	$R(Y_5)$	$R(Y_6)$
1	1	1	2	2	3	3
2	2	3	1	3	1	2
3	3	2	3	1	2	1
ρ_s	1	0.5	0.5	-0.5	-0.5	-1

The correlation coefficient is calculated by:

$$\rho_s = \frac{\sum_{i=1}^{n}[R(x_i) - \bar{R}_x][R(y_i) - \bar{R}_y]}{\sqrt{\sum_{i=1}^{n}[(R(x_i) - \bar{R}_x)]^2 \sum_{i=1}^{n}[(R(y_i) - \bar{R}_y)]^2}} \tag{2.3}$$

where $\bar{R}$ denotes the mean value of the ranks.

ρ_s is defined in the interval $[-1, 1]$ where $\rho_s = 1$ corresponds to a perfect positive monotonic correlation between the variables whereas $\rho_s = -1$ corresponds to a perfect negative monotonic correlation. A positive monotonic correlation means that whenever x increases so does y while a negative correlation means that y decreases whenever x increases. A $\rho_s = 0$ corresponds to no monotonic relationship between the variables. Significance for the Spearmans coefficient is determined by comparing the obtained coefficient to the distribution of a true random variable P_s [29].

As an example consider all possible pairs of ranks between the variables $x_1, \ldots, x_3, y_1, \ldots, y_3$ in Table 2.3, and let the null hypothesis H_0 = "There is no correlation between the variables". In this example, the rank $R(X)$ is ordered, and all the possible permutations are achieved by reordering the ranks in the sets of different observations Y. The number of permutations are the $n!$ different way of allocating the ranks of $Y, R(Y)$ to different indexes $R(X)$ which can be done in $n!$ different ways.

By enumerating the correlation coefficient for the whole population, the probability density function for the random variable P_s can be derived.

$$p(P_s = \rho_s) \begin{cases} \frac{1}{6} & \rho_s = 1 \\ \frac{2}{6} & \rho_s = 0.5 \\ \frac{2}{6} & \rho_s = -0.5 \\ \frac{1}{6} & \rho_s = -1 \end{cases}$$

By comparing the obtained coefficient to the likelihood of a random variable adopting the same or a more extreme value, $p(P_s \geq \rho_s)$, significance can be determined. For $n \geq 19$ the density function for P_s can be approximated with a t- or normal-distribution whereas there exists tables with pre-calculated critical values for smaller n [28]. This example is adapted from [29].

Distribution-free correlation coefficients are frequently used in ecology [30] and may be preferable for software metrics, which tend not to be normally distributed [26].

Distribution-free rank tests have the benefit of being less sensitive to outliers and do not make any assumption of the underlying distribution [30].

Chapter 3

Accuracy study - methodology

The methodology presented in this chapter is primarily aimed towards Goal 1 of the thesis. The chapter describes the process of workload selection, compilation and processing of data, and the how the collected data is used to evaluate the accuracy of Unison's estimation.

3.1 Workload selection

To fairly evaluate the performance of Unison, it is desirable to use it in a realistic environment. In this case, an environment consists of a target processor and appropriate workload for this architecture.
Unison supports few architectures, of which Hexagon is one of them. Hexagon is an interesting architecture to study for compiler design optimizations, due to its architecture where it is up to the compiler to exploit ILP.

3.1.1 Selection of benchmark suite

When selecting a workload for benchmarking, it is important that the workload is relevant for the target field of application [21]. As the target field of application can be rather huge (possible programs that might be executed on a target system), researchers have composed *benchmark suites*, collections of benchmarks designed to exercise relevant properties of the system. Several benchmark suits exists for different application domains. Mediabench [31] is a benchmark suite focusing on embedded multimedia and communication systems, which fits well with the application of the target architecture. Compared to a general-purpose benchmark suite, SPECint, it exhibits a higher mean Instructions Per Cycle (IPC) value, higher instruction and data cache read hit rate and a lower bus utilization. The benchmarks included in MediaBench are written entirely in C, which allows portability between architectures without source code modifications.

Even though the purpose of this study is to investigate the accuracy of Unison's estimation which should not be related to the application domain, the results are believed to be magnified by having a realistic environment where the application and compiler can make use of the underlying hardware.

3.1.2 Selecting benchmarks

As Unison estimates code quality improvement per function, the task of selecting tests is not only to select benchmarks, but functions in benchmarks. In Section 2.7.1, two methods of subsetting benchmark suites are described. Both of these methods are biased by the compiler used, and the properties used to subset the benchmark suite are either in terms of characteristics of the assembly code or statistics from the hardware. A more general approach was developed that overlooks these fine grained properties, but tries to maintain the native characteristics of the benchmark.

A rule of thumb states that approximately 90% of the execution time is spent in 10% of the code, which can also be applied for DSPs where most of the CPU time is spent in arithmetic-intensive loops [32].
To capture the characteristics from the benchmarks, an assumption is made.

Assumption. A benchmark designed to exercise property P of a processor will exercise P where the benchmark spends most of its time.

From this assumption, the process of selecting functions for evaluation is the following.

Each benchmark is compiled with LLVM for the Hexagon architecture, simulated and profiled to collect run time information. The profiling information is inspected and functions, excluding library functions, are sorted in descending order based on the cycles they consume. From this list, the cumulative cycle coverage ratio is derived. This ratio describes the portion of total cycles captured by taking X of the top functions for this evaluation. The results are shown in Table 3.1, where library functions and functions with less than 0.01% are excluded. The geometric mean as well as the average gain in coverage is calculated. It was decided to select additional functions as long as $\textit{average gain} \geq 0.01$.

Based on these results, the top 6 functions for each benchmark are selected for evaluation with Unison, with exception of **rawcaudio** and **rawdaudio** where only two functions are available. Selecting additional functions would only yield a small improvement in terms of executed cycles, but a great cost of compiling and benchmarking. Worth mentioning about the benchmarks **epic** and **unepic** is that they spend most of the cycles executing floating point operations, which in the Hexagon v4 architecture must be done in software. The code implementing the floating point operations is provided through precompiled libraries, which are not studied in this project.

Table 3.1: Cumulative cycle coverage ratio depending on number of selected functions

Benchmark	Top X functions in benchmark									
	1	2	3	4	5	6	7	8	9	10
cjpeg	0.2193	0.4372	0.6146	0.7701	0.8594	0.9003	0.9069	0.9122	0.9148	0.9173
djpeg	0.3145	0.5817	0.7254	0.8534	0.8905	0.9059	0.9108	0.9136	0.9156	0.9174
epic	0.0627	0.0646	0.0655	0.0662	0.0667	0.0670	0.0673	0.0675	0.0677	0.0677
unepic	0.0131	0.0224	0.0296	0.0358	0.0392	0.0423	0.0438	-	-	-
g721_decode	0.3917	0.6175	0.7400	0.7706	0.7906	0.8041	0.8150	0.8223	0.8294	0.8364
g721_encode	0.4093	0.6345	0.7571	0.7934	0.8134	0.8323	0.8437	0.8532	0.8608	0.8678
rawcaudio	0.9786	0.9793	-	-	-	-	-	-	-	-
rawdaudio	0.9746	0.9753	-	-	-	-	-	-	-	-
geo-mean	0.2329	0.3292	0.3829	0.4167	0.4327	0.4428	0.4472	0.4490	0.4504	0.4516
avg. gain	0.2329	0.0963	0.0538	0.0338	0.0160	0.0101	0.0044	0.0018	0.0014	0.0013

3.2 Evaluation technique

To investigate the estimated code quality by Unison, a simulation environment of the hardware is used as described in Section 2.6.1. Simulation enables the collection of additional run-time execution statistics without altering the source program, enables further control over experiment variables and gives better insight into the results [33, 34].

3.2.1 Compiling

Each of the selected benchmarks was compiled with both LLVM and Unison. The selected functions in the benchmarks were compiled separately with Unison, resulting in one executable for every function selected from the benchmark. This helps to isolate any effect which the compilation with Unison might result in by having the processor in the same state, in terms of contents of cache memories and state of functional units and branch predictor, upon entry of a function. Moreover it makes sure that the function is placed in the same memory location compared to LLVM; if Unison extends the code size required for a function, any subsequent code might have an altered memory address. The effect of altering memory location of code is described in Section 2.7.2.

To achieve comparable results between the compilers and to isolate the effect of the instruction scheduling and register allocation, several optimization passes are disabled in the LLVM compilation chain (Appendix B).

The process used to produce executable code for the Hexagon DSP is depicted in Figure 3.1

- **clang** [35], a C language family front-end for LLVM, is used to compile the

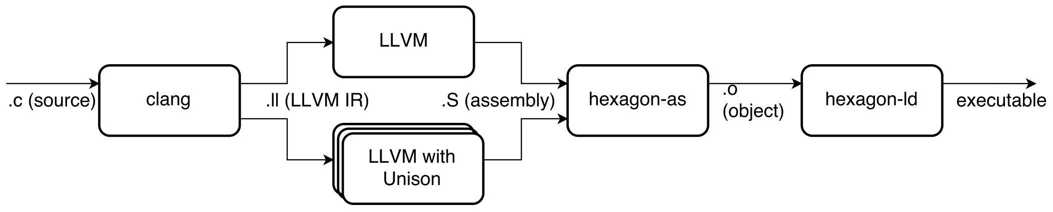

Figure 3.1: Compilation chain used to produce executable code

source .c-files into LLVM IR. clang is invoked with flags to optimize for speed as well as no function inlining. With function inlining enabled, the optimized code by Unison might not be executed if the front-end decides to inline the function where the function is called.

- **LLVM** is invoked multiple times per set of source files for a benchmark. It is invoked once for every function selected for evaluation with Unison, with the purpose of only having just one function compiled with Unison in each produced binary. This isolates the effects that compilation of a single function with Unison carries. LLVM is invoked once more without Unison, to create the baseline with which Unison is to be compared with. For every compilation with Unison, the estimated speedup is collected.

- **hexagon-as**, an assembler for Hexagon provided in the SDK translates the assembly files to machine code in object files.

- **hexagon-ld**, a linker for Hexagon combines every generated object file to an executable file.

From the selected 40 functions for investigation, one (unepic_main) got excluded due to that it could not be fully automated but required manual interaction to get it through the compile chain. When emitting assembly code for function unepic_main, regardless if compiled with Unison or not, references to global variables *temp_int* and *temp_short* got denoted improperly.

3.2.2 Data collection

Several methods exist to retrieve run-time data from the simulator. This section discusses the alternatives and describes the selected approach. Additionally it explains the process of utilizing the selected approach.

To retrieve run-time statistics from the simulator, the regional profiling feature described in Section 2.6.1 is utilized. The main benefits with this approach compared to using the timing interface in the simulator is that no source code modifications are required and it does not introduce additional cycles. An early experiment concluded that an initial call to the pair of functions used to start and stop the timer added 69 cycles as overhead, and subsequent calls to the same pair of functions could add less than that when simulating with hardware timings. In order to get the number of cycles that a functions consumes, one would need to investigate how many times the function is executed and use an approximation of the overhead cycles.

To utilize the regional profiling feature, a memory range and a unique name must be specified for the function of interest. The Hexagon SDK includes a utility program which lists object file symbols and the size of these from a binary file, hexagon-nm [36]. By parsing the output of hexagon-nm, the memory address range for a specific function can be calculated. An example output when using regional profiling is depicted in Figure 3.2, where statistics from the function H2V2_FANCY_UPSAMPLE are collected.

```
------REGION PROFILING------
Total cycles profiled 334960
Region 'H2V2_FANCY_UPSAMPLE': (0x00013220 - 0x00013854)
        Total cycles        : 334960
        Total packets       : 312150
        Total instructions  : 867750
        User cycles         : 0
        Guest cycles        : 0
        Supervisor cycles   : 312150
        Exception cycles    : 0
        Wait cycles         : 0
        Stall cycles        : 22810
        Bus Read bytes      : 2816
        Bus Write bytes     : 0
        ICache: 378752 hits, 60 misses
        DCache: 285750 hits, 743 misses
```

Figure 3.2: Example output from the simulator utilizing regional profiling

3.3 Data processing

To answer the question whether or not the estimations by Unison are accurate, test variables are derived from data of pairwise simulations of functions compiled with Unison and LLVM respectively. This section introduces the definitions of Improvement and Error of Estimation which are used throughout this thesis, as well how the results are filtered and aggregated.

3.3.1 Definition of Improvement and Error of Estimation

Almdahl's Law defines speedup as [21]

$$\text{Speedup} = \frac{\text{Execution time without improvement}}{\text{Execution time with improvement}} \tag{3.1}$$

An improved version of some program executing twice as fast as the original version, would result in a speedup of 2, whereas an "improved version" executing twice as slow, would have a speedup of 0.5. Generally, an improved version performing better than the original version, is defined by the interval $]1, \infty[$, and worse by the interval $]0, 1[$. A problem occurs when comparing this ratio with an estimation, where the ratio of a speedup and a slowdown uses different ranges; the speedup ratio is asymmetric. This is circumvented by transforming the values of a slowdown to a negative representation of a speedup, defined in the range of $] - 1, -\infty[$, by negating and inverting the ratio. To have it comparable with the estimation and continuous over zero, 1 is subtracted from the speedup before negating the ratio.

The transformed value is defined as *Improvement* and given by the calculation:

$$\text{Improvement} = \begin{cases} \frac{cycles(LLVM)}{cycles(Unison)} - 1 & \text{if } cycles(LLVM) \geq cycles(Unison) \\ -(\frac{cycles(Unison)}{cycles(LLVM)} - 1) & \text{otherwise} \end{cases} \tag{3.2}$$

The absolute value corresponds to how much faster one execution is to the other, while the sign tells which compiler that produced the faster code. This transformation is visualized by an example in Figure 3.3, where a speedup and a slowdown is compared to the estimation.

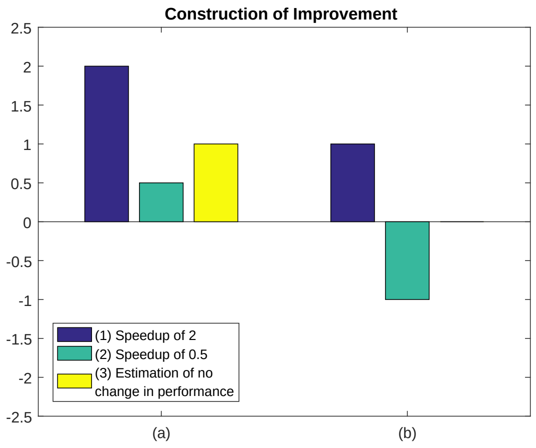

Figure 3.3: Calculating the improvement:
(a) corresponds to the initial values
(b) is the representation of the Improvement

From this definition, the *Error of Estimation* (EoE) is defined as:

$$\text{Error of Estimation} = \text{Improvement} - \text{Estimation} \tag{3.3}$$

In Table 3.2, the Error of Estimation is calculated for the series in the example visualized in Figure 3.3. The improvement series, (b), has the desired property that the absolute value of the Estimation Error is not affected by whether or not it is derived from a speedup or a slowdown. The Error of Estimation can be interpreted as the inaccuracy of the estimation in terms of percentage points.

Table 3.2: Error of Estimation of example

Series	Error of Estimation	
	(1)-(3)	(2)-(3)
(a)	1	-0.5
(b)	**1**	**-1**

3.3.2 Filtering

To prevent samples based on incorrectly generated code, code that does not correspond to the input program, polluting the results, the results are filtered in three different ways.

1. The output from each simulation of the code generated by Unison for the selected functions in the benchmarks is compared to the output from the same benchmark compiled with LLVM. The output from LLVM acts as a baseline and any difference in the output leads to disqualification of the sample.
2. If a case study, (Section 5.5), reveals that the generated code is incorrect, samples related to this code will be disregarded.
3. Any function resulting in less than 10 samples after filtering out possible erroneous values is disregarded as a whole.

The first and the second point are there to mitigate the risks of having erroneous samples in the results while the purpose of the third point is to protect the number of samples when aggregating data, described in Section 3.3.3.

3.3.3 Data aggregation

After collecting data from the simulations of the functions compiled with Unison, it is desirable to consolidate the measurements in order to give a generalized statement regarding the estimation of Unison. Ideally, one would have one single experiment representing every application to be executed on this kind of processor, exercising all the relevant properties with correct weights among them. For this evaluation, this one experiment corresponds to the benchmark suite of choice, MediaBench.

To be able to aggregate the values retrieved from simulations, an additional assumption must be made.

Assumption. MediaBench is representative for the typical workload of Hexagon; it captures all the relevant properties of programs to be executed on the processor with representative frequencies among them.

With this assumption, an aggregation of data obtained from the different functions corresponds to observations from a sample of a typical application for Hexagon.

A problem with taking all measurements from the various functions is that the number of values differs from function to function. This would skew the importance of the different properties. As an example imagine the functions F1 and F2 which exhibits the properties X and Y respectively. F1 manages to produce a result 10/10 times whereas F2 only produces a result 3/10 times. Aggregating all the 13 results together would skew the occurrences of the properties in the results, giving the importance of $X > Y$.

This example violates the assumption that the properties of the selected functions reflects the properties of the benchmark suite. A solution to this problem would be to take a representative single value, such as the mean or the median, for each function when aggregating data. Taking a single value representing a greater set of measurements hides any variability in the set of measurements. To circumvent this issue, a number of samples is randomly chosen from every function without replacement when aggregating the data. The number of samples to pick is equal to the number of measurement of the function which managed to produce the least amount of results.

This technique maintains the characteristics of the selected benchmark suite and captures variance in the results.

3.3.4 Data evaluation

This section describes how the obtained data is used to draw conclusion regarding the estimation given by Unison. Figure 3.4 depicts the process used to evaluate the data.

- The Error of Estimation visualization summarizes the data in a box plot with the purpose of giving an overview of the accuracy of the estimation.
- The EoE of the aggregated data is visually inspected and tested for normality using the Anders-Darlington test, see Section 2.8.1. If the data turns out to be normally distributed a statistical test for zero mean is employed, otherwise a study of the correlation between the estimation and improvement per function is performed. Depending on the skew of the data for individual functions, either the mean or the median is chosen to represent the data for the corresponding function.
- To investigate the relationship between the estimation and improvement, Spearman's correlation coefficient will be used, (Section 2.8.2). Spearman's coefficient is a measurement of monotonic relationship between pairwise data, which does not make any assumption of what distributions the data may

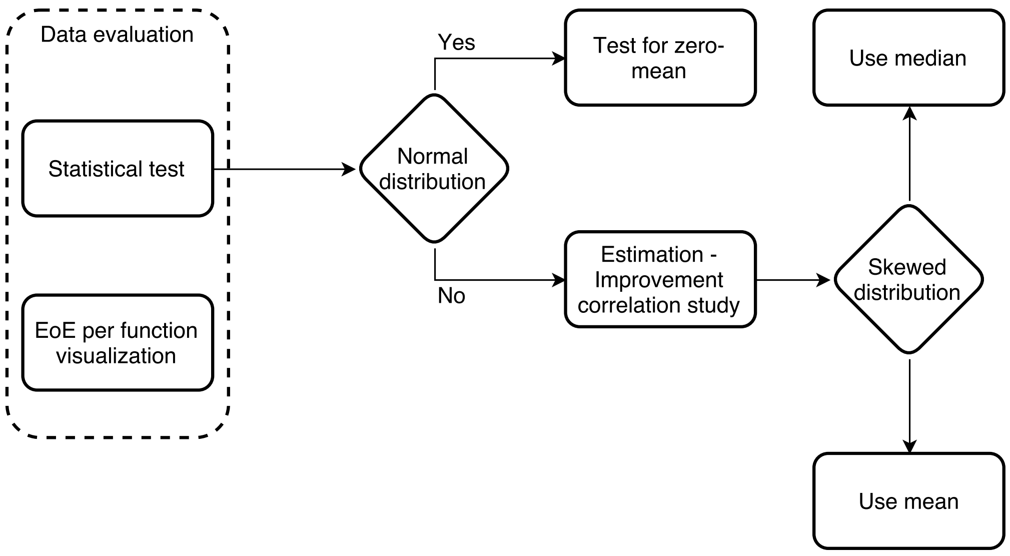

Figure 3.4: Process of evaluating data

stem from. The monotonic relationship is relevant due to how Unison operates; Unison outputs code of greater and greater quality according to the cost function, until optimality is proven or a timeout occurs. A positive monotonic relationship between the estimation and improvement means that code with a greater estimation yields greater improvement.

3.4 Discussion

MediaBench was selected as benchmark suite due to its target field of application is in line with the Hexagon DSP. There exist other benchmark suites including benchmarks of similar target of application such as the EEMBC [37] and MiBench [38]. Previous internal work done at SICS with MediaBench and Unison suggest that by choosing MediaBench the effort of compiling the benchmarks would decrease.

Out of 19 benchmarks included in MediaBench benchmark suite, only 8 where studied in this thesis. It is hard to argue that these 8 benchmarks are representative for the whole benchmark suite, causing the assumptions stated in Section 3.3.3 not to hold. The initial idea was to study additional benchmarks but due to time constraints were not manageable.

The accuracy of the simulator used in this project is according to the manual cycle-approximate with a performance within 2% of cycle-accurate with modeling of hardware stalls. These errors do not appear to be random, since simulation of a specific byte code always yields the same amount of cycles. A question regarding what type of errors that constitutes these 2% of inaccuracy was posted to the official

Hexagon developer forum, but no explanation was given.

Section 3.3.1 defines Improvement, which has the property of being symmetric around zero in contrast to the speedup ratio, and is used throughout the thesis. An alternative approach would be to take the logarithmic result of the speedup ratio, which hold the same symmetric property. However, when comparing the logarithmic ratio to the logarithmic estimation, the raw values are believed to be of less meaning than the values of the constructed variables.

Chapter 4

Accuracy study - results

This chapter presents the results associated with the main goal of the thesis. The naming convention used henceforth for a function appearing in a benchmark is {benchmark-name}_{function-name}. The first part of this chapter, Section 4.1, presents an overview of the obtained results together with interesting observations whereas the second part, Section 4.2 studies the correlation. The significance level used for all statistical tests are 5%.
The results are based on 130 compilations and simulations with Unison. A list of excluded functions and the reason for exclusion is provided in Appendix D.

4.1 Overview

The results from the simulations of the separate functions are visualized in Figure 4.1 where the Error of Estimation and improvement are plotted side by side. A positive Error of Estimation means that Unison underestimate the quality of the code produced whereas a negative values implies an overestimation of the code quality. The first part of the plot, Error of Estimation, shows that Unison tend to overestimate the quality of its generated code. What can be observed by looking at the bottommost part of the box plot, depicting the improvement of the functions, is that Unison produces code of higher quality in most of the cases. By studying the size of the inter-quantile range, the size of the boxes, one can conclude that there is a variance in the variability of the improvement between the different functions. For both parts of the plots, where there is a variance in the values, the median is more than often not centered in the box. That means that there is a skew in the results for that specific function, which is verified by studying the distributions for individual functions in Appendix A.

Another thing worth pointing out is that the characteristics of the improvement of each function are maintained when subtracting the estimation; the boxes have a similar appearance in the topmost and the bottommost part of the graph. This indicates that the estimation by Unison is somewhat constant for each function.

Table 4.1: Unique estimations by functions

Function	Unique estimations
rawcaudio_main	2
all other	1

Table 4.1 lists the number of unique estimations by functions, which agrees with this observation. This means that only rawcuadio_main is subject of getting code of different qualities from Unison's perspective, caused by timeouts in the solver.

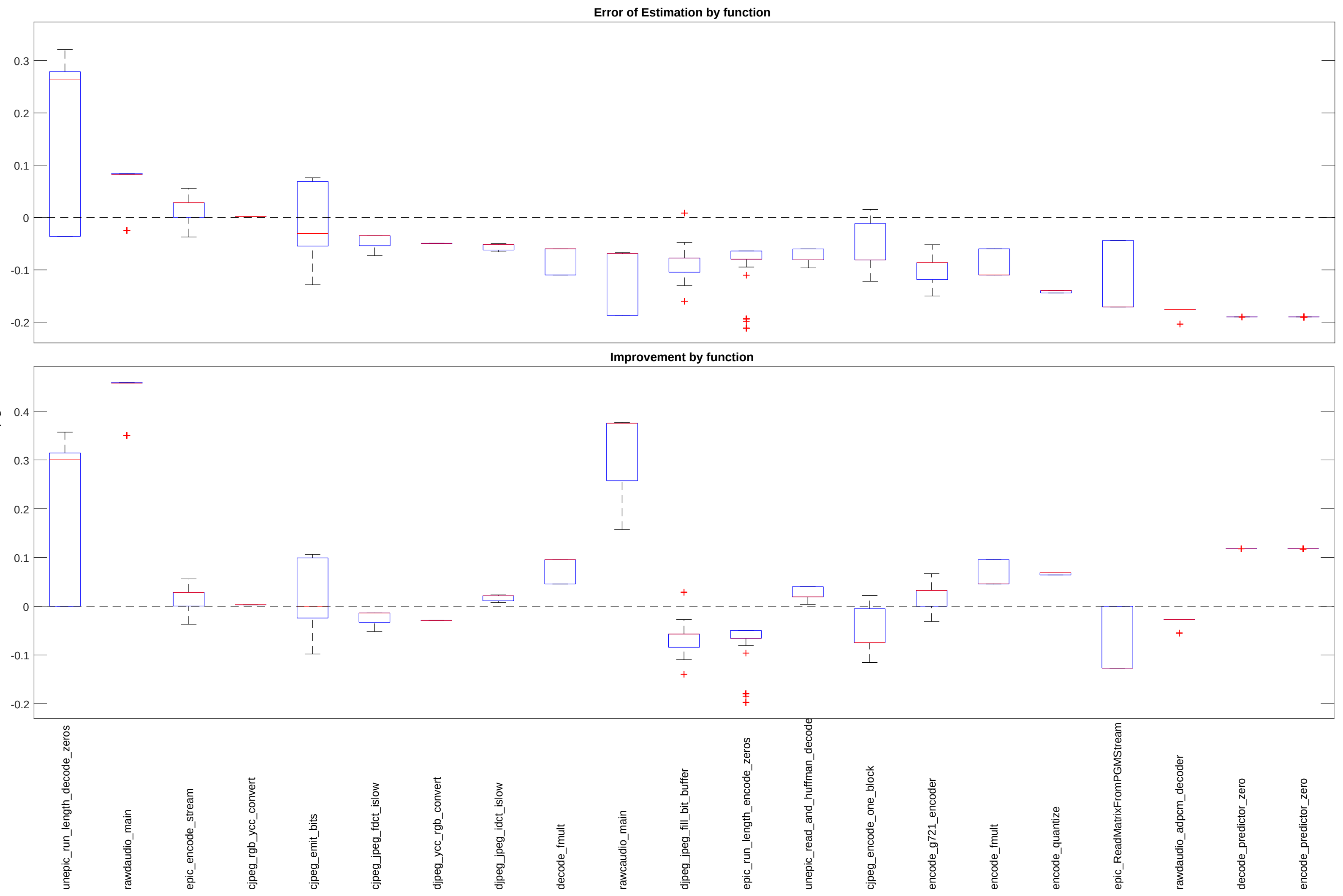

Figure 4.1: Error of Estimation/Improvement by function

The observation of Unison overestimating the quality of the code is studied further in Figure 4.2, where the distribution of the Error of Estimation is visualized in a histogram. The data used for this graph is derived by random sampling as described in Section 3.3.3. The histogram strengthens the thesis of Unison over-

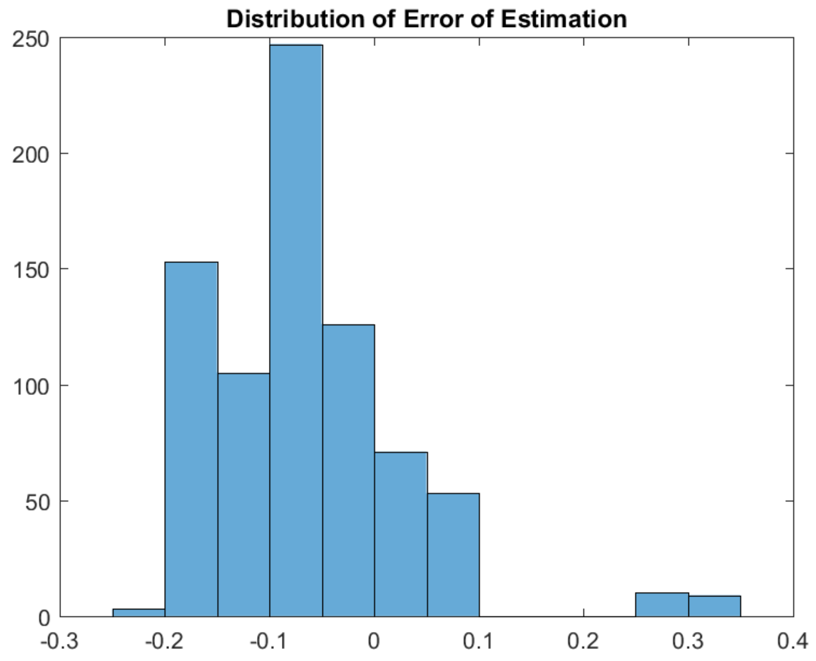

Figure 4.2: Distribution of Error of Estimation

estimating its generated code quality. By looking at the tails of the histogram, it can be seen that it is subject of outliers. Furthermore, the histogram has a shape which does not immediately reject the underlying distribution of being normally distributed. However, the Anderson-Darlington, described in Section 2.8.1, rejected the hypothesis that this data set could stem from a normal distributed variable with a probability value of $p \leq 0.0005$. Due to its non-normality, a test for zero mean is not carried out, but the Figures 4.1 and 4.2 suggest that the estimation by Unison is not accurate.

4.2 Correlation

Instead of trying to prove accuracy of the estimation, the relationship between the estimation and improvement is studied. In Figure 4.3 the median of the estimation and improvement are plotted as different bars for every function. The median is selected for representing the results of a function due to skew in the values.

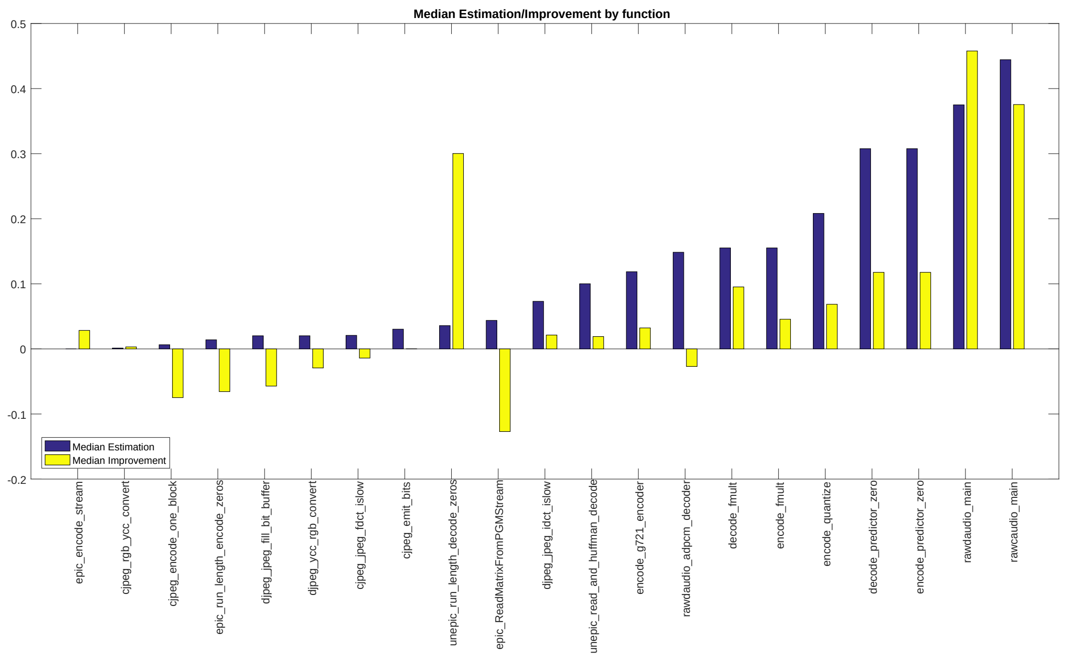

Figure 4.3: Comparison Estimation/Improvement by function

The functions are sorted by the estimation in an ascending order, and even though it is hard to argue for accuracy there is a relationship between the estimation and improvement. A Spearman's correlation coefficient was calculated with the pairwise data from Figure 4.3 which reported a strong positive monotonic relationship between the estimation and the improvement, $\rho_s = 0.72, p = 0.0002$.

Even though the estimations tend to be on the optimistic side of the improvement, Unison produces solution of greater quality as it tries to find the optimal solution. This correlation means that even if the estimation is not accurate, a subsequent solution produced for the same code should result in greater quality than the previous when executing the code.

Chapter 5

Divergence study - methodology

The previous chapter presented results of the simulations and the relation between the estimation and the improvement. It did not cover reasons for the observed divergence nor the impact of the sources of error. This chapter is devoted to presenting the motivation and the methodology used to dissect the results from the previous chapter.

5.1 Motivation

In order for Unison to make a more accurate estimation it is fundamental to understand where the divergence originates from. Isolating and quantifying the sources of error should reveal how sources of divergence contribute to the total, exposing what should be taken into account when improving the model used by Unison. Ultimately, if all possible sources of divergence are eliminated accurately, the results could act as an experimental verification of the cost model used by Unison for the Hexagon v4 DSP.

5.2 Reasons for divergence

To identify sources of divergence, the dependencies of the Error of Estimation are visualized in Figure 5.1.

The Error of Estimation is calculated as the difference between the improvement and the estimation.
The estimation in turn depends on the cost-function used by Unison which determines a cost for the code of a function, together with the statically estimated execution frequencies of basic blocks.
The improvement depends on the ratio between the observed cycles when executing the code produced by the different compilers, which may be affected differently by hardware stalls.

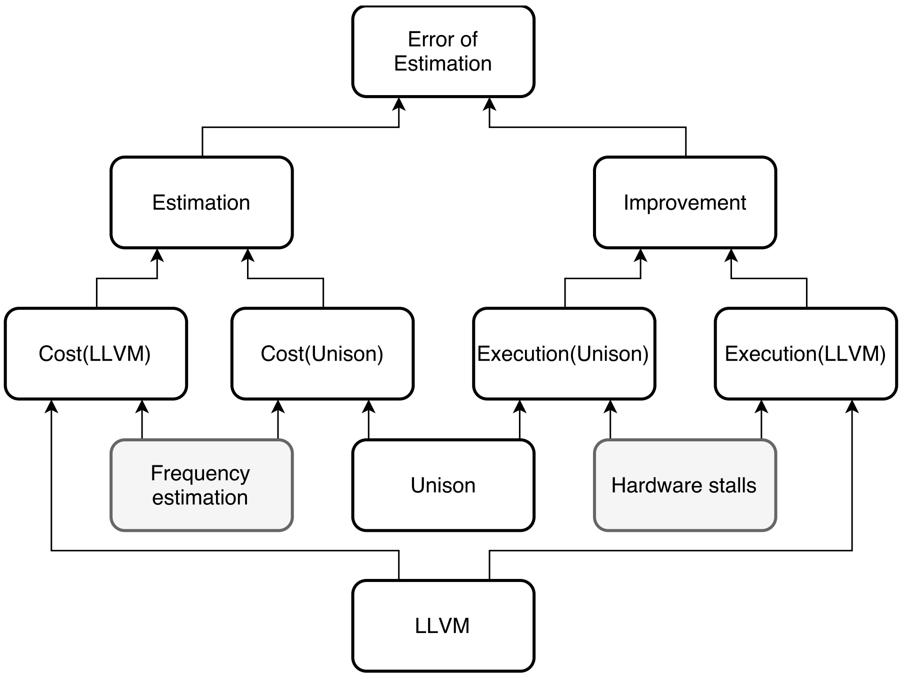

Figure 5.1: Dependencies of Error of Estimation
Unison denotes the code with instruction scheduling and register allocation done by Unison and **LLVM** corresponds to the code produced by LLVM.

The identified potential sources of divergence in the Error of Estimation from Figure 5.1, are the following:

1. The frequency estimation of how often a basic block will execute affects the performance of Unison in two ways.

   a) It affects the constraint solver by giving basic blocks with higher frequency greater impact on the cost of a function (Section 2.4), thus affecting the improvement of the code.

   b) It is used when estimating the cost of the code produced by LLVM and Unison resulting in the estimation given by Unison.

2. Hardware stalls affect the execution of the code produced by LLVM as well as Unison, which could have an impact on the improvement thus affecting the Error of Estimation. Even if both the code by LLVM and Unison are subject to an equal amount of stalls, k, the ratio will still be affected:

$$\frac{cycles(a)}{cycles(b)} \geq \frac{cycles(a)+k}{cycles(b)+k} \quad \text{if } cycles(a) \geq cycles(b), k \geq 0 \tag{5.1}$$

5.3 Process

The method used to eliminate sources of divergence is depicted in Figure 5.2. Hardware stalls will be eliminated from all measurements whereas **case studies** will be performed to study the impact of the estimated execution frequency on the estimation, Section 5.2 1b.

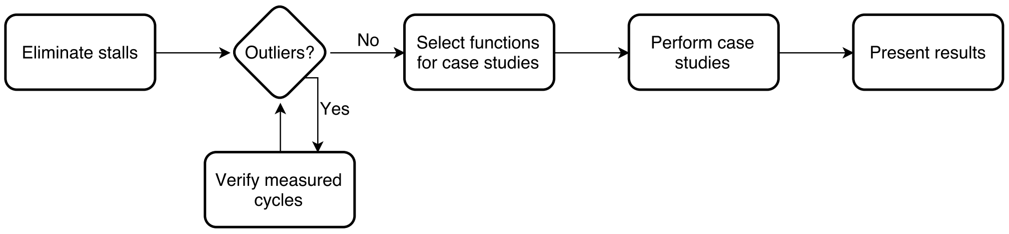

Figure 5.2: Process of dissecting the Error of Estimation

- Elimination of hardware stalls will be performed on all functions appearing in the results from the first part of this thesis. Any outliers in the intermediate results without hardware stalls will be investigated further, using profiling data to track cycle consumption to individual instructions.

- Selection of functions for case studies will be done based on the value of the Error of Estimation excluding hardware stalls. The top three functions with a positive EoE and the top three functions with a negative EoE will be selected for case studies. The reason behind taking the functions with the most extreme values is to see if there are other sources of divergence, or if the incorrect frequencies are accountable for the remaining error.

  Assumption. If there exists other sources of divergence, they are more likely to appear in functions with the greatest unaccounted Error of Estimation.

- The results will be presented in different ways:

  1. The measurements without hardware stalls will be visualized in a box plot along the measurements including the stalls. The purpose of this plot is to visualize the impact of the stalls per function.
  2. The ratio of stall cycles/total cycles will be visualized in pie charts for Unison and LLVM respectively. If stall cycles affect code by one compiler more than the other, this plot should tell.
  3. The impact of stall cycles on the Error of Estimation will be summarized by taking the arithmetic average of absolute value of the median of each function including/excluding stall cycles. In [39], the authors describe that the results are meaningless by summarizing normalized numbers using arithmetic mean, which is circumvented by using the geometric mean.

However, since the Error of Estimation is the difference between two ratios, interpreted as percentage points and not percentages, summarizing using the arithmetic average is believed to be suitable.

4. The results from the case studies will be presented as a breakdown of the EoE per function, depicting the contribution from hardware stalls, the statically estimated execution frequencies of basic blocks, and possible any remaining unidentified source of divergence.

5.4 Hardware stalls

To eliminate the stalls caused by the hardware a function $cycles_{ns}(X)$ is defined, which corresponds to the number of cycles consumed by an execution of a code X with **no stalls** included:

$$cycles_{ns}(X) = cycles(X) - stall\_cycles(X) \tag{5.2}$$

These values are part of the data collected with the regional profiling feature presented in Section 3.2.2.
The Improvement$_{ns}$ is defined as:

$$\text{Improvement}_{ns} = \begin{cases} \frac{cycles_{ns}(LLVM)}{cycles_{ns}(Unison)} - 1 & \text{if } cycles_{ns}(LLVM) \geq cycles_{ns}(Unison) \\ -(\frac{cycles_{ns}(Unison)}{cycles_{ns}(LLVM)} - 1) & \text{otherwise} \end{cases} \tag{5.3}$$

The Error of Estimation excluding stall cycles, EoE$_{ns}$, is then defined as:

$$\text{EoE}_{ns} = \text{Improvement}_{ns} - \text{Estimation} \tag{5.4}$$

When inspecting the improvements from the results excluding hardware stalls, some outliers appeared. Appendix E describes the applied procedure of identifying and correcting these outliers.

This finding does not rule out that there might be additional measurements suffering from the same problem, resulting in a verification of the measurements obtained by regional profiling against profiling information from the simulations. This verification is done for every unique observation of $cycles_{ns}$ for any function compiled with Unison and LLVM, and adjusted to include all cycles consumed by instructions belonging to the function of interest.

The results presented in Chapter 4 have been adjusted according to this method. As described in Appendix E, this adjustment adds an uncertainty to the observed cycles; it does not account for any stalls that might occur by the instructions incorrectly included/excluded by the regional profiling feature. These uncertainties are not present in results excluding the stalls.

5.5 Case studies

The idea of the case studies is to eliminate the uncertainties that come with the estimated execution frequencies of the basic blocks by LLVM. By feeding back the observed execution frequencies of basic blocks to the estimator used by Unison, a more accurate estimation can be given. The estimator operates on code expressed in the LLVM MIR Format [11], which requires altering the execution frequencies of each basic block in the intermediate code by both Unison and LLVM.

The procedure used to retrieve an improved estimation is depicted in Figure 5.3.

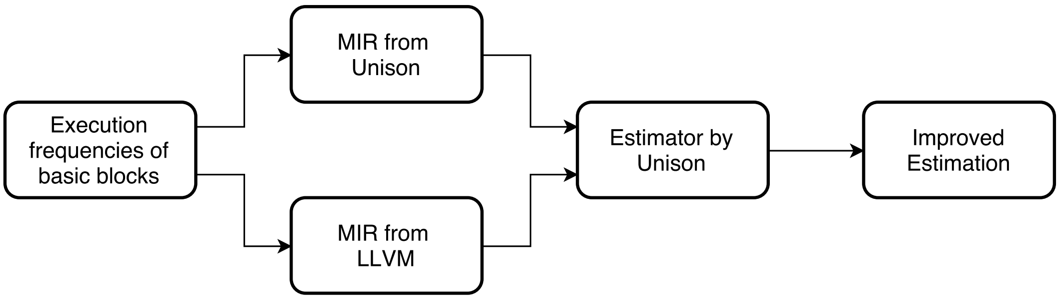

Figure 5.3: Retrieving an improved estimation based on observed execution frequencies

5.5.1 Retrieving execution frequencies of basic blocks

Execution frequencies can be retrieved in two different ways (Section 2.5). The utilized approach is based on the profiling data collected by the simulator. The simulator record every single cycle when using the profiling feature of the simulator, which is used to determine the execution frequencies of the basic blocks. More specifically, the profiling information contains name-value pairs in the format: *{Program Counter - number of cycles consumed}*. By running the simulator without modeling of hardware stalls, the *number of cycles consumed* = *number of executions*.

The mappings between the basic blocks in the MIR-files and the assembly code is not necessarily of a one-to-one relationship. Two adjacent basic blocks are merged in the assembly code if the first basic block is the only predecessor to the subsequent basic block. This prevents usage of the symbol table from the binaries to retrieve the corresponding basic block from a memory address. Instead, the disassembly is used to fill in the missing gaps to be able to do the mapping between basic blocks in the MIR-file and the execution frequency from the profiling information. The disassembled code contains the mappings between the instructions and their memory addresses. This procedure is depicted in Figure 5.4.

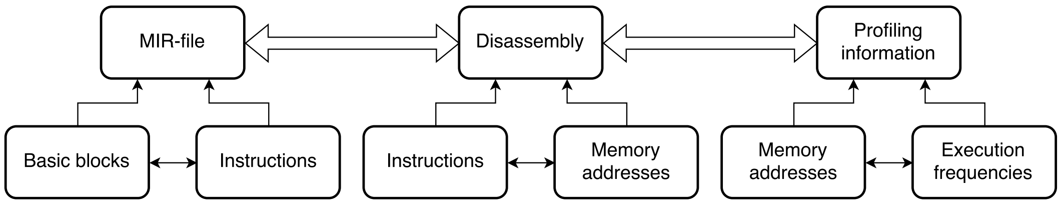

Figure 5.4: Establishing the mapping between basic blocks in the MIR file and execution frequencies from simulation

5.6 Discussion

The discovery of cycles associated with instructions incorrectly included/excluded in a defined memory range, in the measurements from the regional profiling feature turned up late into the project, (Section 5.4). Measurements subject to this error were adjusted according to the cycle measurements using the profiling feature without modeling of stalls. This adjustment does not take any stall cycles caused by the incorrectly included/excluded instruction into account. By the time the discovery was made, it was not feasible with regards to the limited time frame to re-simulate all of the compiled benchmarks with a different approach of collecting data.

Chapter 6

Divergence study - results

This chapter presents the results corresponding to the secondary goal of this thesis. The first part, Section 6.1, presents the results excluding the stall cycles with a comparison against the previous results including stalls. The second part, Section 6.2, presents the results from the case studies, by breaking down the contribution to the total EoE for the selected functions.

6.1 Excluding stall cycles

Figure 6.1 shows box plots comparing the Error of Estimation and improvement including/excluding stall cycles, EoE vs. EoE$_{ns}$ and Improvement vs. Improvement$_{ns}$ respectively. What can be observed by comparing the size of the boxes, is that the variance in the results is eliminated by exclusion of stalls cycles. This means that code of same quality according to cost model in fact are of the same quality when executing the generated code excluding stall cycles, and that the stalls are the reason for varying performance from different compilations.

Having code of different quality from different compilations of the same code is undesirable in an compiler, especially for embedded systems where deadlines and worst-case execution time could be important. If all hardware stalls could be captured in the constraint model employed by Unison, not only would the variance in the results disappear but the quality of the code would increase. The variance in the improvement is for all but one function, rawcaudio_main, code of the same quality according to the cost model, but if the stalls could be modeled they would be of different quality. This means that Unison could select only the code resulting in the greatest improvement for each function.

Looking at the bottommost graph, (Improvement comparison), the median values for the series excluding hardware stalls are generally greater than their counterparts including stalls. For five of the functions the opposite is true, exclusion of the stalls results in a lower improvement. Four of these functions appear in the leftmost

part of the graph, which corresponds to functions that Unison underestimated the improvement for. Furthermore, it appears that the functions in the rightmost part of the graph, corresponding to functions that Unison overestimates the improvement for, have a greater gain comparing the series including/excluding stalls.
With the exclusion of stalls, only three functions have a negative improvement compared the eight functions in the series including stalls.

By looking at the median values for each function in the top most graph, Error of Estimation comparison, it can be observed that the series excluding the stalls are closer to zero. Only for one of the functions, djpeg_fill_bit_buffer, is the absolute value of the Error of Estimation greater when excluding the stalls. This function is studied further in Section 6.2, where an explanation to this phenomena is presented. The observed change in the improvement has a positive impact on the Error of Estimation, for all but this one function.

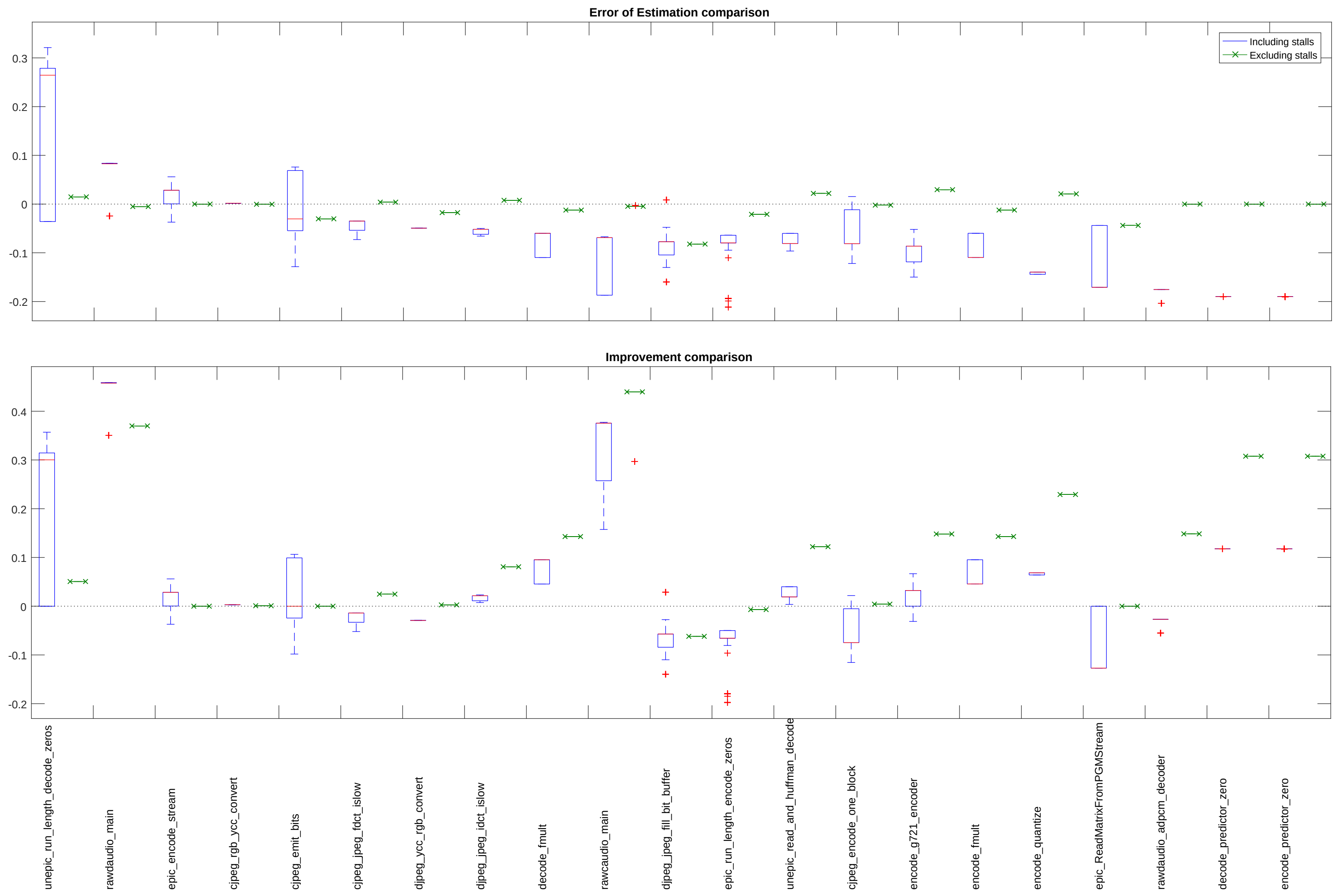

Figure 6.1: Comparison of EoE and Improvement including/excluding stall cycles

In Figure 6.2, the impact of stall cycles on the code generated by Unison and LLVM is presented. Stall cycles appear more frequently in code generated by Unison, constituting 8% of the total cycles compared to 3% in code generated by LLVM.

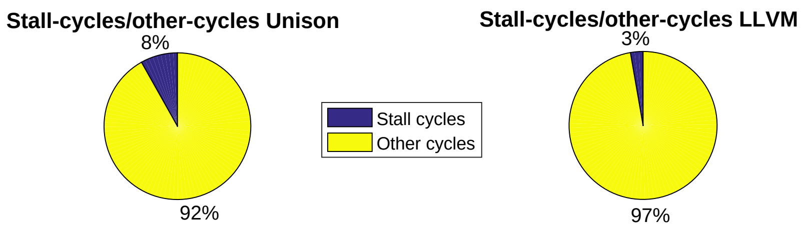

Figure 6.2: Ratio of stall cycles/other cycles comparison

By inspecting the underlying values of the pie chart, presented in Table 6.1, the apparent reason for the increased frequency of stall cycles is due to an increased number of stall cycles together with an overall lower cycle consumption by Unison.

Table 6.1: Comparison of stall/other cycles between code generated by Unison and LLVM

	Stall cycles	Other cycles
Unison	$7.1781 * 10^8$	$8.0830 * 10^9$
LLVM	$2.6239 * 10^8$	$9.2911 * 10^9$

To summarize the impact of stall cycles on the Error of Estimation, the arithmetic mean, as discussed in Section 5.3, of the absolute values of the median for each function is presented in Table 6.2, together with the maximum and minimum observed error. This data suggests that by exclusion of stall cycles, the average error is reduced by roughly 84% and the maximum absolute error by roughly 69%.

Table 6.2: Summary of the impact of stall cycles on EoE

	Average EoE	Maximum EoE	Minimum EoE
EoE	0.0978	0.2645	-0.1901
EoE_{ns}	0.0157	0.0295	-0.0821

The data presented in this section suggests that the hardware stalls, which have a greater impact on code generated by Unison compared to LLVM, are accountable for the majority of the observed EoE and the variance in in the improvement. Unison does not capture the dynamic behavior of the processor in the constraint model in order to find the optimal solution. If some of the dynamic features could be modeled with confidence, the varying performance would be reduced and the improvement would increase.

6.1.1 Correlation

In Figure 6.3, the median estimation and improvement for each function are presented side by side in the same fashion as in Section 4.1. A visual inspection of the graph suggests that the correlation should be stronger than its counterpart including stalls in Figure 4.3. This is verified by calculating the Spearman's correlation coefficient, which reported a very strong positive monotonic relationship, $\rho_s = 0.90, p \leq 0.0001$. By comparing the correlation coefficients including/excluding stalls, it shows that code of greater quality is more likely to perform better than code of lower quality when the impact of stalls is eliminated.

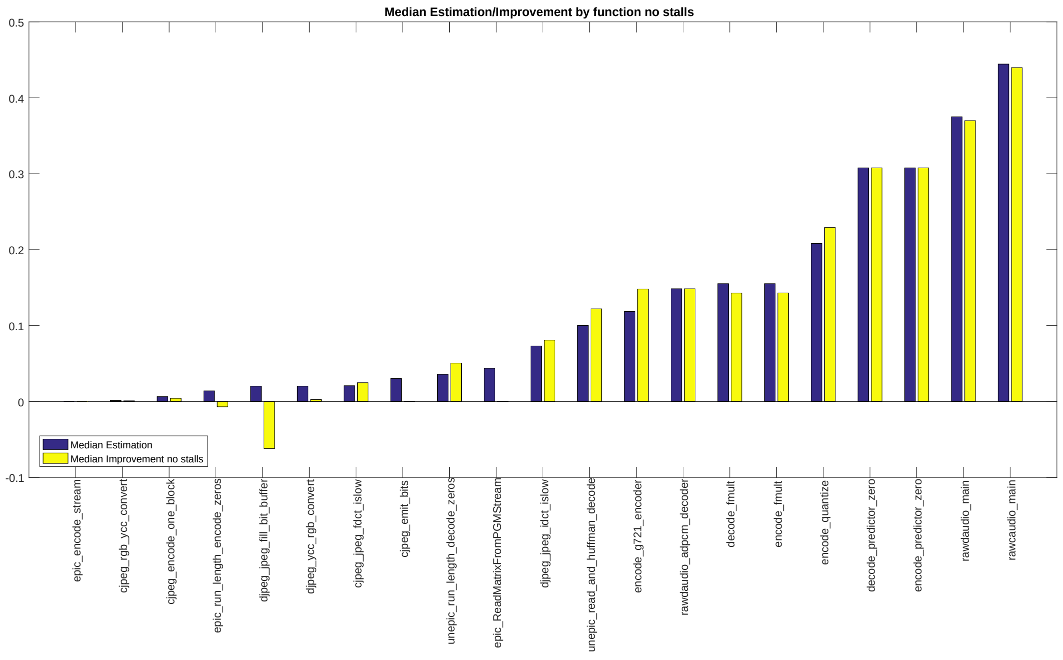

Figure 6.3: Comparison of estimation/improvement by function excluding stalls

6.2 Results from case studies

Figure 6.4 shows the breakdown of the EoE for the case studies. By eliminating the hardware stalls and employing the improved estimation based on correct execution frequencies, no additional errors in the estimation were recorded for any of the functions selected for case studies. This can be seen by absence of Unaccounted EoE in either of the bars in the figure. As discussed in Section 5.3, functions where chosen for case studies by selecting three functions with the greatest positive and negative EoE with stalls removed, with the primary goal to maximize the impact of adjusting the frequencies. Even though this selection process should emphasize the impact of EoE due to incorrect frequencies, the contribution from the stalls is hard to ignore for four of the functions.

The total value of the bars might be greater than median of the EoE in Figure 6.1. This is due to the error contribution by hardware stalls and estimations based on incorrect frequencies cancels out each other. E.g., a contribution to the total error of 0.5 by hardware stalls and -0.2 by incorrect frequencies results in an observed EoE of 0.3.

While the results in Chapter 4 suggested that the estimations given by Unison were inaccurate, it is strengthened further by showing that elimination of factors of error yields an accurate estimation.

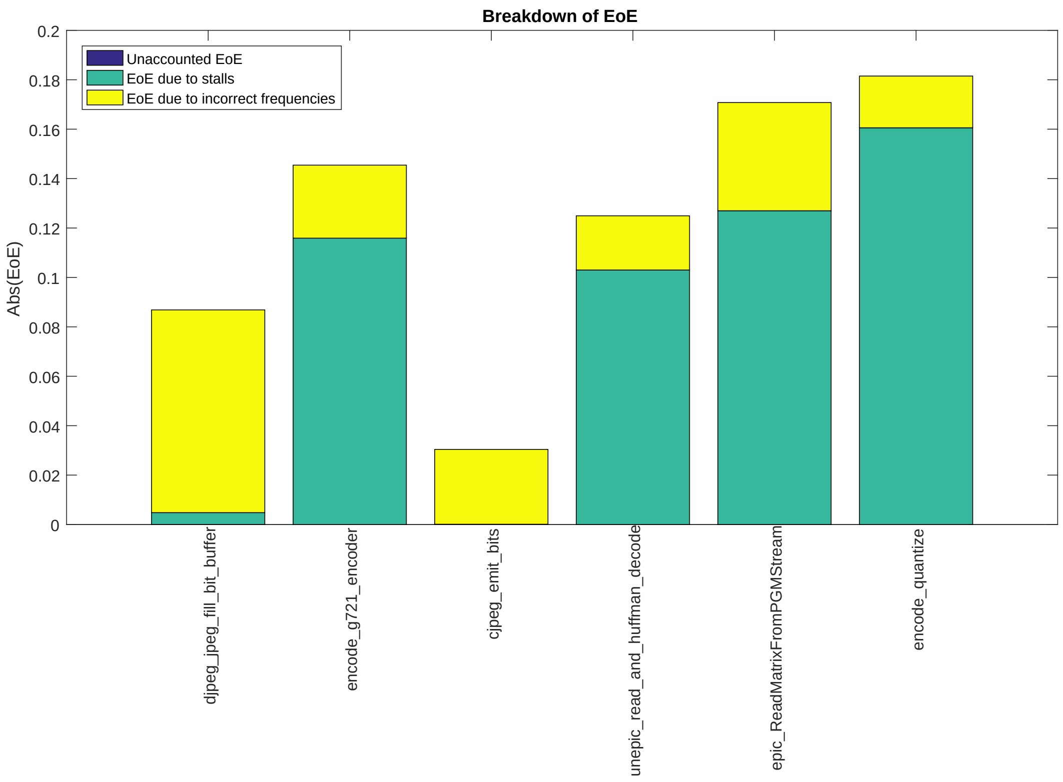

Figure 6.4: Breakdown of EoE

Chapter 7

Conclusions and future work

7.1 Conclusions

This thesis shows evidence of that the estimation of the improved code quality by Unison is not accurate, which was the primary goal of this thesis. Although this statement is not backed up by a statistical test, it is strengthened by showing that elimination of sources of divergence makes the estimation accurate. The main factor for the inaccuracy is due to hardware stalls, of which the code generated by Unison suffers more of compared to the code from LLVM. The stalls have a varying impact on code of the same quality by Unison, which could be reduced if sources of stalls could be captured in the constraint model. Despite the influence of stalls, there is a strong monotonic relationship between the estimation and the observed improvement. This correlation strengthens the idea that running Unison for a longer time implies code of greater quality.

The results of the case studies performed suggest that there is no other source of error than what is discussed in this thesis. Despite choosing case studies to maximize the impact of incorrect frequencies, the stalls where the dominant factor for inaccuracy for most of the case studies. The breakdown of the Error of Estimation and the quantification of contributing factors is a coarse grained result mapped towards the secondary goal of the thesis.

In addition to the contributions presented above, this thesis presents methods for quantifying the impact of dynamic processor features and static estimations by compilers.

7.1.1 Side effects

A side effect of carrying out this project has been the gain in maturity of Unison. 20 bugs have been reported during this thesis, where 18 of them can be viewed at the issue tracker for the code base [40]. The remaining two are summarized below.

- A non leaf function, a function that might call other functions, did not properly allocate memory on the stack for its local variables.
- A conditional jump, where the condition is evaluated in the same instruction bundle must contain a hint whether or not the jump is taken. This hint was not provided by Unison.

Out of the 20 reported bugs, all but four are fixed. Without correction of these bugs this evaluation would not be able progress, but it still emphasizes the importance of the reported bugs.

7.2 Future work

This thesis concludes that hardware stalls is the main factor for divergence in the estimation given by Unison. The next step to improve the estimation for this specific processor, would be to track the underlying sources of these stalls. The execution statistics by the simulator contains a breakdown of stalls occurrences for an execution of a program as a whole, and not just a specific function. By differentiating the stall statistics between code generated by LLVM and Unison, an indication of what kind of additional stall cycles Unison suffers from can be derived.
If it is not obvious why or what instruction that causes the additional stall cycles, the correlation between micro-architecture independent measurements and these stalls could be analyzed. This might provide additional insight on how the stalls could be captured in the constraint model.

The amount of bugs found by assembling and executing the code generated by Unison for the functions studied in this thesis suggests that there might be additional bugs hidden. By performing similar experiments on functions for other benchmarks additional bugs might be revealed, resulting in Unison being a more mature and reliable compiler to use with the Hexagon processor.

Bibliography

[1] Roberto Castañeda Lozano. *Integrated Register Allocation and Instruction Scheduling with Constraint Programming.* Information and Communication Technology, KTH Royal Institute of Technology, 2014.

[2] 7.8 IEEE code of ethics. `http://www.ieee.org/about/corporate/governance/p7-8.html`. [Online; accessed 29-May-2017].

[3] Keith Cooper and Linda Torczon. *Engineering a compiler.* Elsevier, 2011.

[4] Amy Brown. *The architecture of open source applications, volume ii*, volume 2. Kristian Hermansen, 2012.

[5] David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[6] Krzysztof Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[7] The LLVM compiler infrastructure. `https://llvm.org`. [Online; accessed 6-February-2017].

[8] The LLVM target-independent code generator'. `http://llvm.org/docs/CodeGenerator.html`. [Online; accessed 6-February-2017].

[9] Gecode - an open, free, efficient constraint solving toolkit. `http://www.gecode.org/`. [Online; accessed 8-February-2017].

[10] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Combinatorial spill code optimization and ultimate coalescing. In *ACM SIGPLAN Notices*, volume 49, pages 23–32. ACM, 2014.

[11] Machine IR (MIR) format reference manual. `http://llvm.org/docs/MIRLangRef.html`. [Online; accessed 8-February-2017].

[12] Roberto Castañeda Lozano. *The Unison Manual.* [accessed 25-January-2017].

[13] Erik Ekström. Implied constraints for the unison presolver, 2015. TRITA-ICT-EX-2015:74.

[14] Mikael Almgren. Evaluation and implementation of dominance breaking presolving techniques in the unison compiler back-end, 2015. TRITA-ICT-EX-2015:73.

[15] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.

[16] Clang compiler user's manual. `https://clang.llvm.org/docs/UsersManual.html`. [Online; accessed 22-May-2017].

[17] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, 34(2):34–43, 2014.

[18] Inc. Qualcomm Technologies. Hexagon simulator user guide, 2013. 80-N2040-17 Rev. A.

[19] Inc. Qualcomm Technologies. Hexagon gprof profiler, 2011. 80-N2040-4 Rev. C.

[20] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. The effect of omitted-variable bias on the evaluation of compiler optimizations. *Computer*, 43(9):62–67, 2010.

[21] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010.

[22] Aashish Phansalkar, Ajay Joshi, and Lizy K John. Subsetting the SPEC CPU2006 benchmark suite. *ACM SIGARCH Computer Architecture News*, 35(1):69–76, 2007.

[23] Aashish Phansalkar, Ajay Joshi, Lieven Eeckhout, and Lizy Kurian John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 10–20. IEEE, 2005.

[24] Nornadiah Mohd Razali, Yap Bee Wah, et al. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics*, 2(1):21–33, 2011.

[25] NIST Sematech. Engineering statistics handbook. *NIST SEMATECH*, 2006.

[26] NE Fenton. Software metrics: a rigorous approach. 1991.

[27] Georgy L Shevlyakov and Hannu Oja. *Robust Correlation: Theory and Applications*, volume 3. John Wiley & Sons, 2016.

[28] Peter Y Chen and Paula M Popovich. *Correlation: Parametric and nonparametric measures.* Number 137-139. Sage, 2002.

[29] Vijay K Rohatgi and AK Md Ehsanes Saleh. *An introduction to probability and statistics.* John Wiley & Sons, 2015.

[30] Catherine Potvin and Derek A Roff. Distribution-free and robust statistical methods: viable alternatives to parametric statistics. *Ecology*, 74(6):1617–1628, 1993.

[31] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[32] Allan Frederiksen, Rasmus Christiansen, Jeff Bier, and Peter Koch. An evaluation of compiler-processor interaction for DSP applications. In *Signals, Systems and Computers, 2000. Conference Record of the Thirty-Fourth Asilomar Conference on*, volume 2, pages 1684–1688. IEEE, 2000.

[33] Jakob Engblom. Using simulation tools for embedded software development. In *Embedded System Conference, San Jose, CA, USA*, 2008.

[34] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, 1990.

[35] clang: a C language family frontend for llvm. `https://clang.llvm.org/`. [Online; accessed 21-Mars-2017].

[36] Inc. Qualcomm Technologies. Hexagon binutils - a gnu manual, 2013. 880-N2040-1 Rev. J.

[37] Jason A Poovey, Thomas M Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE micro*, 29(5), 2009.

[38] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[39] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[40] Issues - unison-code/unison - GitHub. `https://github.com/unison-code/unison/issues`. [Online; accessed 28-May-2017].

[41] unison-code/llvm - GitHub. `https://github.com/unison-code/llvm`. [Online; accessed 20-June-2017].

[42] unison-code/unison - GitHub. `https://github.com/unison-code/unison`. [Online; accessed 20-June-2017].

Appendix A

Distribution of Error of Estimation

Figure A.1 and A.2 depicts the distributions of the Error of Estimation per function. The bin width used in the histograms are selected by Matlab to the present the distributions of each of the function.

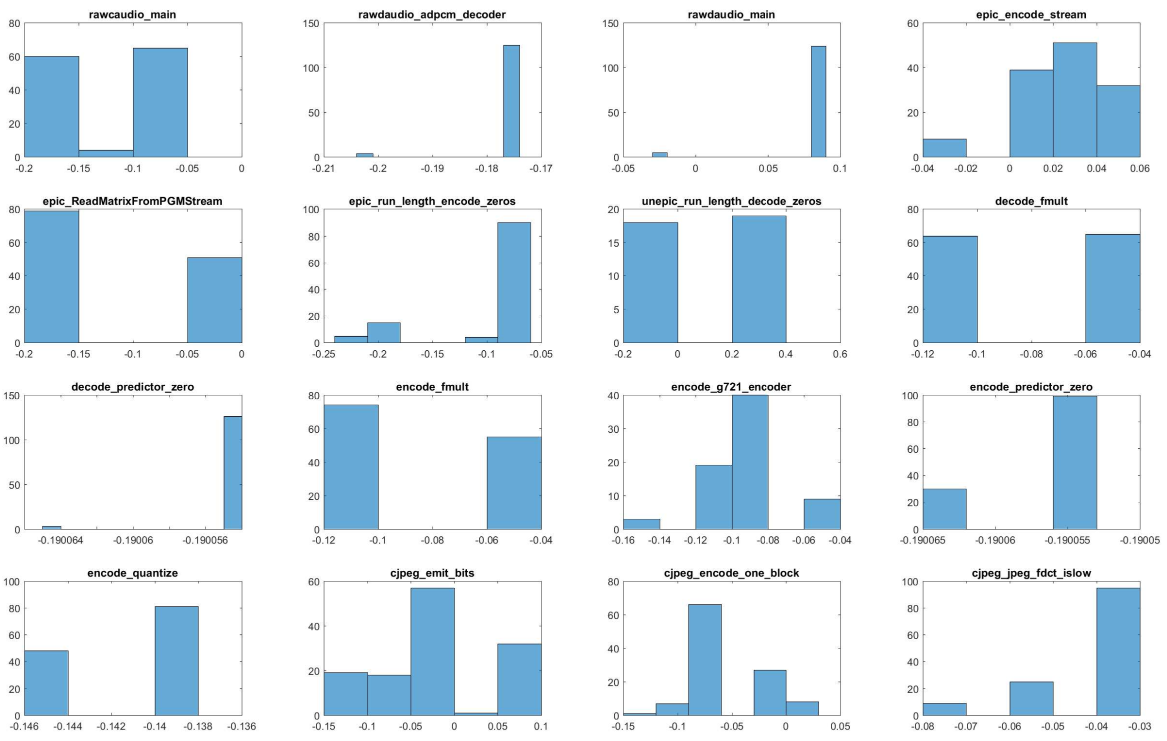

Figure A.1: Error of Estimation per function

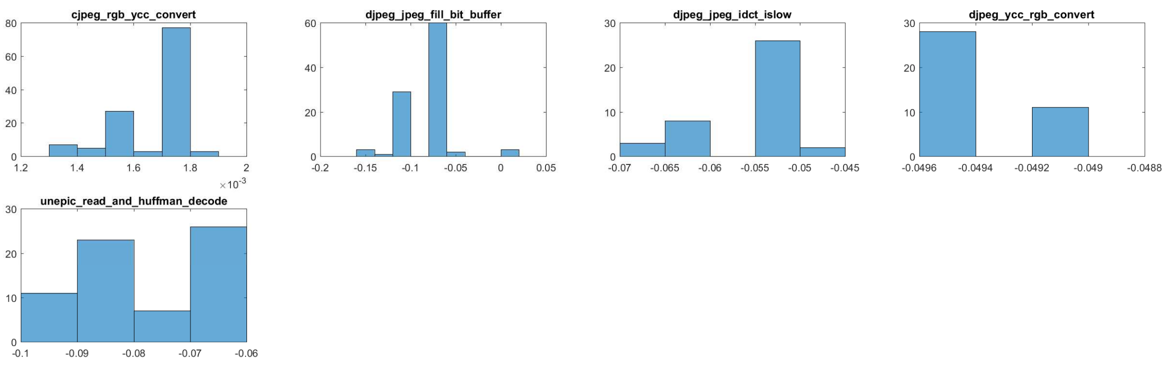

Figure A.2: Error of Estimation per function continued

Appendix B

Compilation flags and software used

This appendix presents the software used along with the version number and parameters used.

B.1 clang

The version of the compiler front-end clang used in this project is 3.8.1. It is invoked with the following parameters:

```
-S -O3 -fno-inline --target=hexagon -emit-llvm -std=c11
```

B.2 llc

llc compiles source files in MIR-format to assembly code for the target processor. The version used in this project is a copy of the original repository, including a driver to invoke Unison for targeted functions [41]. The branch used is `release_38-unison`, commit `cd985fba0f336ec5a71205954527adf7a50eeb4b` (2016-10-14). It is invoked with the following parameters:

```
-O3 -unison-no-clean
   -unison-import-flags="--simplifycontrolflow=false"
   -unison-normalize-flags="--simplifycontrolflow=false"
   -unison-verbose
   -unison-solver-flags="--emit-improvement"
   -unison-max-instrs=1000
   -unison-model-flags="--strictlybetter=false"
   -march=hexagon -mcpu=hexagonv4
```

The following parameters are also passed to llc, with the purpose of disabling optimization passes to isolate the effect of instruction scheduling and register allocation:

```
-disable-post-ra -disable-tail-duplicate
   -disable-branch-fold -disable-block-placement
   -disable-phi-elim-edge-splitting
   -disable-hexagon-cfgopt -disable-if-conversion
```

B.3 Unison

Unison is available from its public repository [42]. The version used in this project is compiled from commit `0a6b75cec651347d1327af4bdab8a116ac44cae9` (2017-03-13). It is invoked from llc by passing the parameter:
`-unison-single-function=<function-name>`

B.4 Gecode

The constraint solver used by Unison is Gecode [9]. Gecode is invoked by Unison during compilation. The version used in this project is 5.0.1.

Appendix C

Benchmarks in MediaBench

Table C.1 describes the benchmarks used in this study and which parameters they are invoked with.

Table C.1: Selected benchmarks from MediaBench

Benchmark	Description	Execution parameters
rawcaudio	Adaptive differential pulse code modulation encoder	`< clinton.pcm > test_caudio.adpcm`
rawdaudio	Adaptive differential pulse code modulation decoder	`< clinton.adpcm > test_daudio.pcm`
epic	An experimental image compression encoder	`test_image -b 25 -o test_image.E`
unepic	An experimental image compression decoder	`test_image.E test_image.E.U`
encode	G.721 voice compression encoder	`-4 -l < clinton.pcm > clinton.pcm.g721`
decode	G.721 voice compression decoder	`-4 -l < clinton.g721 > clinton.g721.pcm`
cjpeg	JPEG image compression	`-dct int -outfile testout.jpg testimg.ppm`
djpeg	JPEG image decompression	`-dct int -ppm -outfile testout.ppm testorig.jpg`

Appendix D

Excluded functions

Table D.1 lists all the functions that was selected for evaluation but are not present in the results.

Table D.1: Functions excluded from the results

Function	Reason for exclusion
rawcaudio_adpcm_coder	Bug #15 prevented code emission from LLVM
encode_quan	Produced output differed from output by LLVM
encode_update	Produced output differed from output by LLVM
decode_g721_decoder	Unison could not find a better solution before timeout
decode_quan	Produced output differed from output by LLVM
decode_update	Produced output differed from output by LLVM
decode_unpack_input	Produced output differed from output by LLVM
cjpeg_forward_DCT	Unison could not find a better solution before timeout
cjpeg_h2v2_downsample	Unison could not find a better solution before timeout
djpeg_decode_mcu	Unison could not find a better solution before timeout
djpeg_decompress_onepass	Produced output differed from output by LLVM
djpeg_h2v2_fancy_upsample	Case study revealed that execution pattern differed between Unison and LLVM
epic_internal_filter	Too many instructions (2769)
epic_quantize_image	Unison could not find a better solution before timeout
epic_reflect1	Bug #15 prevented code emission from LLVM
unepic_collapse_pyr	Too many instructions (3166)
unepic_unquantize_image	Proven absence of a better solution
unepic_internal_int_transpose	Unison could not find a better solution before timeout
unepic_main	32-bits constant operands in assembly code denoted improperly

Appendix E

Measurement errors

When eliminating hardware stalls from the measurements, the variance in the improvement disappeared for each function but four:

- unepic_read_and_huffman_decode
- decode_fmult
- encode_fmult
- encode_quantize

Inspecting the distribution of the consumed cycles with the hardware stalls removed, Figure E.1, showed that each function had a total of two unique observations of consumed cycles without stalls.

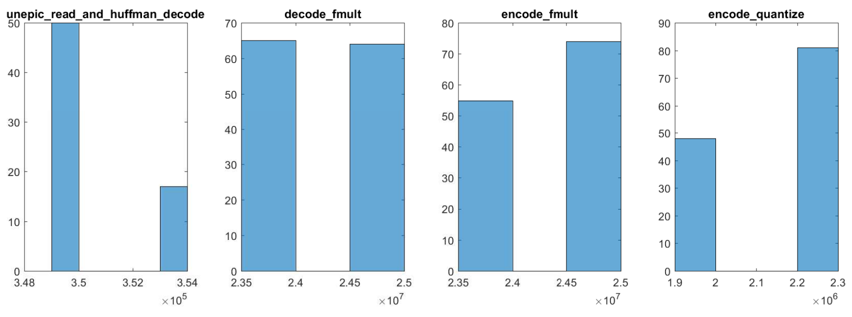

Figure E.1: Distribution of consumed cycles for functions with variance in Improvement without hardware stalls

The different observed cycles and the difference between the observations are presented in Table E.1.

From each function, two different binaries were selected; one that generated an observation of the lowest cycle count and one that generated the highest cycle count. These binaries where simulated without hardware stalls and with profiling, as well

Table E.1: Unique observed consumed cycles without stalls per function

Function	Observation lowest	Observation highest	Difference
unepic_read_and_huffman_decode	349511	**353009**	3498
decode_fmult	23601046	**24781206**	1180160
encode_fmult	23587066	**24767226**	1180160
encode_quantize	**1931406**	2216668	285262

as disassembled with *hexagon-objdump* [36]. The disassembly of the binaries verified the memory range specified for the regional profiling; the code of the function were covered by the ranged defined for regional profiling. The profiling information from the execution without hardware stalls contains information of how the cycles are spent by the processor per instruction address.

An example is presented in Figure E.2 where profiling data from encode_fmult have been reduced to only include the function of interest, fmult. By summing together all the cycles spent in the function, a verification of the data collected with the regional profiling feature can be done.

```
PC = 0x00005ae0,count = 1180160
PC = 0x00005aec,count = 1180160
PC = 0x00005af8,count = 1180160
PC = 0x00005b04,count = 1180160
PC = 0x00005b10,count = 1180160
PC = 0x00005b1c,count = 1180160
PC = 0x00005b2c,count = 1177471
PC = 0x00005b30,count = 1177471
PC = 0x00005b3c,count = 1177471
PC = 0x00005b44,count = 1177471
PC = 0x00005b48,count = 1177471
PC = 0x00005b54,count = 1177471
PC = 0x00005b58,count = 1180160
PC = 0x00005b68,count = 1180160
PC = 0x00005b74,count = 1180160
PC = 0x00005b80,count = 1180160
PC = 0x00005b8c,count = 1180160
PC = 0x00005b9c,count = 1180160
PC = 0x00005ba8,count = 1180160
PC = 0x00005bb0,count = 1180160
PC = 0x00005bbc,count = 1180160
```

Figure E.2: Subset of profiling data from encode_fmult. The cycles spent in the function add up to 24767226 cycles

Both of the binaries for each of the functions investigated, resulted in the same number of cycles when manually summarizing the cycles spent within the memory address belonging to the function of interest. The result of this manual cycle calculations are marked with **bold** in Table E.1.

Note that the difference in number of cycles between the lowest and highest observation for encode_fmult is equal to the number of cycles spent in the last instruction bundle, 0x00005bbc, which was the case for 3 out of 4 functions studied. The outcome of this study is that the results from these functions are adjusted to match the manual cycle calculation by adding or subtracting the difference of cycles in samples that suffered from this measurement error. This adjustment does not take into account any hardware stalls that might occur by the instructions that are incorrectly included/excluded by the regional profiling.

TRITA TRITA-ICT-EX-2017:75

www.kth.se