Structural Optimization of Numerical Programs for High-Level Synthesis

Xitong Gao

Supervisor George A. Constantinides

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London and the Diploma of Imperial College London

September 20, 2016
Xitong Gao

Structural Optimization of Numerical Programs for High-Level Synthesis

September 20, 2016

Supervisor: George A. Constantinides

Imperial College London

Circuits and Systems Research Group

Department of Electrical and Electronic Engineering

South Kensington Campus

SW7 2AZ and London
Abstract

This thesis introduces a new technique, and its associated tool SOAP, to automatically perform source-to-source optimization of numerical programs, specifically targeting the trade-off among numerical accuracy, latency, and resource usage as a high-level synthesis flow for FPGA implementations. A new intermediate representation, MIR, is introduced to carry out the abstraction and optimization of numerical programs. Equivalent structures in MIRs are efficiently discovered using methods based on formal semantics by taking into account axiomatic rules from real arithmetic, such as associativity, distributivity and others, in tandem with program equivalence rules that enable control-flow restructuring and eliminate redundant array accesses. For the first time, we bring rigorous approaches from software static analysis, specifically formal semantics and abstract interpretation, to bear on program transformation for high-level synthesis. New abstract semantics are developed to generate a computable subset of equivalent MIRs from an original MIR. Using formal semantics, three objectives are calculated for each MIR representing a pipelined numerical program: the accuracy of computation and an estimate of resource utilization in FPGA and the latency of program execution. The optimization of these objectives produces a Pareto frontier consisting of a set of equivalent MIRs. We thus go beyond existing literature by not only optimizing the precision requirements of an implementation, but changing the structure of the implementation itself. Using SOAP to optimize the structure of a variety of real world and artificially generated arithmetic expressions in single precision, we improve either their accuracy or the resource utilization by up to 60%. When applied to a suite of computational intensive numerical programs from PolyBench and Livermore Loops benchmarks, SOAP has generated circuits that enjoy up to a $12 \times$ speedup, with a simultaneous $7 \times$ increase in accuracy, at a cost of up to $4 \times$ more LUTs.
Contents

1 Introduction ............................................. 17
  1.1 Thesis Organization .................................. 21
  1.2 Contributions ....................................... 23
  1.3 Publications ........................................ 24

2 Background ............................................. 25
  2.1 Field-Programmable Gate Arrays ................. 26
    2.1.1 FPGA Architecture ............................. 28
    2.1.2 RTL Design Flow .............................. 30
  2.2 High-Level Synthesis ............................... 32
    2.2.1 HLS Design Flow ............................... 33
    2.2.2 Loop Pipelining ............................... 36
    2.2.3 Modulo SDC Scheduling ....................... 38
    2.2.4 Obstacles in Adoption ......................... 44
  2.3 Program Analysis and Abstract Interpretation .... 46
    2.3.1 Data-Flow Analysis Framework ............... 47
    2.3.2 Least Fixpoint Solution to a Data-Flow Analysis Problem 49
    2.3.3 Abstract Interpretation with Intervals ........ 53
    2.3.4 Abstract Domains ............................. 59
  2.4 Intermediate Representations ...................... 64
    2.4.1 Static Single Assignment Form and Control-Flow Graph 64
    2.4.2 Equality Saturation ............................ 68
  2.5 Discovering Equivalent Programs .................. 72
    2.5.1 Improving Performance by Rewriting Arithmetic Expressions 73
    2.5.2 Rewriting Arithmetic Expressions for Accuracy .... 76
# 4.5.2 Discovering Equivalent Structures Efficiently

4.6 Code Generation

4.7 Evaluation

4.7.1 Results

4.7.2 Quality of Resource Estimation

4.8 Summary

# 5 Accurate and Resource Efficient Pipelining of Numerical Programs

5.1 Motivation

5.2 Syntax Definition

5.3 Extending MIRs with Arrays

5.4 Structural Optimization

5.4.1 Improved Algorithm

5.4.2 Transformation Rules

5.5 Performance Analysis

5.5.1 Latency Analysis

5.5.2 Resource Utilization Analysis

5.5.3 Accuracy Analysis

5.6 Evaluation

5.6.1 Results

5.6.2 Discussion

5.7 Summary

# 6 Conclusion

6.1 Future Prospects

6.2 Final Remarks

# Bibliography

# Appendices

A Sound Acceleration of Equivalent Expression Discovery

B Formal Definitions of Equivalent MIR Discovery

C Benchmark Source Code
# List of Figures

2.1 A high-level block diagram of an ALM in Stratix V, from Stratix V Device Handbook [Alt15d]. .......................................................... 29

2.2 Quartus II design flow. .................................................... 31

2.3 The LegUp design flow, adapted from [Can+13] and [LU]. .............. 34

2.4 A simple dot-product example which calculates the dot-product of two arrays \( A \) and \( B \), each with 1024 elements. ........................................ 36

2.5 The resulting schedule of the example program in generated. ............ 37

2.6 The dependence graph formed by the data-dependences in the loop body of the dot-product example in Figure 2.4. The dashed edge highlights the inter-iteration dependence. .............................................. 39

2.7 A simple program example to be statically analyzed. ....................... 47

2.8 The CDFG of `simple` in Figure 2.7. .................................... 48

2.9 The compiled and optimized LLIR output from the dot-product example in Figure 2.4. .................................................. 65

2.10 The CFG of the LLIR code in Figure 2.9. .................................. 66

2.11 A simple loop which computes the factorial of 10, and the resulting PEG. This example and its PEG, showing computations that lead to the final \( x \) and \( y \), is taken from [Tat+]. ................................................. 69

2.12 An simple E-PEG example, taken from [Tat+]. ............................ 72

2.13 An example APEG for the expression \(((a + a + b) \times c\), from [Mar12]. ... 79

3.1 Our algorithm to compute \( cl_N f(\epsilon) \), which discovers a set of equivalent expressions with a \( \cup \)-distributive EEG \( f \) from an initial set of equivalent expressions \( \epsilon \). ................................................................. 98
3.2 The algorithm used to compute $fr(\epsilon)$, i.e. the Pareto frontier from a set of equivalent expressions $\epsilon$. .................................................. 99
3.3 The DFG for the sample expression $(a + b) \times (a + b)$. .................. 100
3.4 The DFG for finding equivalent expressions of $(a + b) \times (a + b)$. ........ 100
3.5 The alternative DFG for $(a + b) \times (a + b)$. ................................. 102
3.6 Optimization of $(a + b)^2$. ......................................................... 108
3.7 Simultaneous optimization of both $e_1$ and $e_2$. ............................... 108
3.8 Varying the mantissa width of Figure 3.7. ...................................... 109
3.9 The Taylor expansion of $\sin(x + y)$. .......................................... 110
3.10 The Motzkin polynomial $e_m$. .................................................... 110
3.11 Accuracy of Area Estimation. ....................................................... 111

4.1 A simple program, basel, written with our syntax definition. .............. 118
4.2 Two pairs of programs that are equivalent but syntactically different. .... 118
4.3 A simple program which exhibits common subexpressions reuse across nested MIRs. ................................................................. 125
4.4 The accuracy analysis of a fixpoint expression. ................................. 132
4.5 The sharing of conditional expressions in a simple program. ............... 133
4.6 The sharing of composition expressions. ........................................ 134
4.7 The sharing of fixpoint expressions in a simple example program. ........ 134
4.8 Example branch fusion creates a cycle. ......................................... 135
4.9 A transformed while loop with different partial loop unroll depths. ....... 139
4.10 The Pareto while loop. ................................................................. 148
4.11 The quality of resource estimation. ............................................... 148

5.1 An overview of our automatic program optimization process. The shaded region shows our internal tool flow. ........................................ 152
5.3 The optimized program using only arithmetic equivalences. ................. 154
5.4 The optimized program using arithmetic equivalences in tandem with control-flow restructuring and memory access optimization. .............. 155
5.5 A simple example program and its corresponding MIR. .................................. 158
5.6 An example to illustrate how MIRs are partitioned. ........................................ 159
5.7 The algorithm used to sample the Pareto frontier. ........................................... 161
5.8 A sample MIR transformation using the independent accesses rule. ................. 163
5.9 The optimized program that computes the Fibonacci sequence. It reduces latency of the original in Figure 5.5a by half and improves accuracy by 50%. 163
5.10 The MIR with edges labelled with latency attributes. ....................................... 165
5.11 Comparisons of our estimated LUT counts against actual LUT counts from VHLS. .................................................................................................................. 172
5.12 Comparisons of our estimated latency statistics against actual latency from VHLS. .................................................................................................................. 173
5.13 Pareto-optimal variants of the Seidel stencil program from Figure 5.2. Each graph shows a 2D projection of the 3D Pareto frontier. In each graph, the original program is marked ×, and the lowest-latency variant obtained by arithmetic transformations alone is marked by the red circle. .................... 176

6.1 Kahan’s compensated summation algorithm to accurately compute the sum of n elements \( \sum_{i=0}^{n-1} x_i \). ................................................................. 183

C.1 simple ................................................................. 207
C.2 taylor ................................................................. 207
C.3 filter ................................................................. 208
C.4 euler ................................................................. 208
C.5 pid ................................................................. 209
C.6 sum ................................................................. 209
C.7 dotprod ........................................................... 209
C.8 tridiag ............................................................ 210
C.9 2mm ............................................................... 210
C.10 3mm ............................................................... 211
C.11 atax .............................................................. 211
C.12 bicg .............................................................. 212
C.13 gemm ............................................................ 212
## List of Tables

4.1 Table of optimization results. ................................................. 145

5.1 Comparison among the optimized implementations generated by *Vivado high-level synthesis* (VHLS)’s expression balancing and our optimizer. The row “Total run time (s)” indicates the wall-clock time in seconds of running the synthesized circuits. ................................................. 155

5.2 Before-and-after examples to demonstrate the access reduction rules. .... 162

5.3 Comparisons of the original (non-shaded rows) and the optimized program with lowest latency (shaded rows), for each benchmark. Values in parentheses are obtained after slightly tweaking our experimental set-up; see Section 5.6.2. We performed place-and-route for exact statistics. ............ 171
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isl</td>
<td>integer set library</td>
</tr>
<tr>
<td>AA</td>
<td>affine arithmetic</td>
</tr>
<tr>
<td>AI</td>
<td>abstract interpretation</td>
</tr>
<tr>
<td>ALAP</td>
<td>as-late-as-possible</td>
</tr>
<tr>
<td>ALM</td>
<td>adaptive logic module</td>
</tr>
<tr>
<td>ASIC</td>
<td>application-specific integrated circuit</td>
</tr>
<tr>
<td>BB</td>
<td>basic block</td>
</tr>
<tr>
<td>CDFG</td>
<td>control/data-flow graph</td>
</tr>
<tr>
<td>CFG</td>
<td>control-flow graph</td>
</tr>
<tr>
<td>CNN</td>
<td>convolutional neural network</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit</td>
</tr>
<tr>
<td>DAG</td>
<td>directed acyclic graph</td>
</tr>
<tr>
<td>DFA</td>
<td>data-flow analysis</td>
</tr>
<tr>
<td>DFG</td>
<td>data-flow graph</td>
</tr>
<tr>
<td>DSA</td>
<td>dynamic single assignment</td>
</tr>
<tr>
<td>DSP</td>
<td>digital signal processing</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>E-PEG</td>
<td>program expression graph with equivalent structures</td>
</tr>
<tr>
<td>EB</td>
<td>expression balancing</td>
</tr>
<tr>
<td>EDA</td>
<td>electronic design automation</td>
</tr>
<tr>
<td>EEG</td>
<td>equivalent expression generator</td>
</tr>
<tr>
<td>FPGA</td>
<td>field-programmable gate array</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GMP</td>
<td>GNU Multiple Precision Arithmetic Library</td>
</tr>
<tr>
<td>GPU</td>
<td>general-purpose graphics processing unit</td>
</tr>
<tr>
<td>HDL</td>
<td>hardware description language</td>
</tr>
<tr>
<td>HLL</td>
<td>high-level language</td>
</tr>
<tr>
<td>HLS</td>
<td>high-level synthesis</td>
</tr>
<tr>
<td>IA</td>
<td>interval arithmetic</td>
</tr>
<tr>
<td>II</td>
<td>initiation interval</td>
</tr>
<tr>
<td>IIR</td>
<td>infinite impulse response</td>
</tr>
<tr>
<td>ILP</td>
<td>integer linear programming</td>
</tr>
<tr>
<td>IMS</td>
<td>iterative modulo scheduling</td>
</tr>
<tr>
<td>IR</td>
<td>intermediate representation</td>
</tr>
<tr>
<td>LAB</td>
<td>logic array block</td>
</tr>
<tr>
<td>LFP</td>
<td>least fixpoint</td>
</tr>
<tr>
<td>LI</td>
<td>loop invariant</td>
</tr>
<tr>
<td>LLIR</td>
<td>low-level virtual machine intermediate representation</td>
</tr>
<tr>
<td>LLVM</td>
<td>low-level virtual machine</td>
</tr>
<tr>
<td>LUT</td>
<td>look-up table</td>
</tr>
</tbody>
</table>
MA metasemantic analysis
MII minimum initiation interval
MIR metasemantic intermediate representation
MLAB memory logic array block
MPFR GNU Multiple Precision Floating-Point Reliable Library
MRT modulo reservation table
PEG program expression graph
PID proportional–integral–derivative
RAM random access memory
RAW read after write
RecMII recurrence-constrained minimum initiation interval
ResMII resource-constrained minimum initiation interval
RTL register-transfer level
SDC systems of difference constraints
SMT satisfiability modulo theory
SSA static single assignment
TFLOPS tera floating-point operations per second
VHLS Vivado high-level synthesis
VLIW very long instruction word
XST Xilinx Synthesis Technology
Introduction

As the computational capability of field-programmable gate array (FPGA) devices grows at an exponential rate, numerical applications running on them become increasingly more complex. Traditional design methodologies, working at the register-transfer level (RTL), have become increasingly costly, forcing us to work at a higher abstraction level [Gaj+92; BDT10a; Mee+12], e.g. to implement our designs using high-level languages (HLLs) such as C.

There are many reasons why FPGA implementations of numerical algorithms are now best obtained via high-level synthesis (HLS) from C: less development effort, the abundance of software engineers compared to hardware designers, the relative ease of testing C code on an ordinary microprocessor, the opportunities for rapid design space exploration, and so on [GR94; Mee+12; Nan+16]. Great advances have been made in this area recently, and the output from HLS tools is nowadays competitive with hand-crafted designs for certain types of applications [BDT10a].

Numerical C programs are typically written with floating-point arithmetic, following the IEEE 754 standard for floating-point computation [ANS08]. Floating-point numbers can represent a wide range of real values, and most programming languages support the standard seamlessly. The standard has become ubiquitous, and is used in most of our software and hardware implementations of numerical programs. Recently, Altera has introduced Arria 10 and Stratix 10 devices to incorporate hardened floating-point digital signal processing (DSP) blocks in the FPGA fabric [Alt15c]. As a result, we expect to see floating-point arithmetic continuing to dominate in the FPGA implementation of numerical applications.

Although we make use of floating-point arithmetic to implement algorithms, generally specified in real arithmetic, in practice, it is often neglected that floating-point com-
putations almost always have round-off errors, i.e. the discrepancy between the actual result in real arithmetic and the rounded result computed with floating-point arithmetic. Round-off errors, when accumulated, can have a devastating effect on numerical accuracy [Hig02].

In fact, properties such as associativity \((a + b) + c \equiv a + (b + c)\) and distributivity \(a \times (b + c) \equiv a \times b + a \times c\) which we consider to be fundamental laws of real numbers no longer hold under floating-point arithmetic [Gol91]. For instance, under single-precision floating-point arithmetic with rounding to the nearest, the result of \((2^{-24} + 2^{-24}) + 1 = 1.00000012\ldots\) is exact, but \((1 + 2^{-24}) + 2^{-24}\) is rounded to 1. Round-off errors in a numerical program are dependent on every arithmetic operation and every input value, and with the impact on floating-point accuracy being so esoteric, it is challenging for engineers to understand the repercussions of switching between “\((a + b) \times c\)” and “\(a \times c + b \times c\)” in their programs.

These numerical properties nevertheless open the possibility of using these rules to generate a program equivalent to the original program in real arithmetic, but which could have better quality than the original when evaluated in floating-point computation. Experienced engineers often apply such expression rewriting intuitions in numerical programs. For instance, when summing a sequence of floating-point values, one can sometimes reduce round-off error in the result by summing the inputs in ascending order of magnitude. On the other hand, one can often reduce latency by applying expression balancing, i.e. rearranging operators in an expression to construct a balanced tree, so that more operators can work in parallel. These heuristics cover a very limited number of possible transformations and may not always improve the original code. There does not exist a trivial process to apply steps of transformations using equivalence rules to optimally trade off latency, resources and numerical accuracy.

Existing HLS tools consider these rewrites to be unsafe [Xil12], and thus make very limited use of them when restructuring floating-point data-paths. For instance, Vivado high-level synthesis (VHLS) [Xil12] has only a very simple expression balancing feature that uses associativity to improve latency, and only expressions with either additions or
multiplications are optimized. Moreover, it does not produce optimal loop pipelining, because it does not take into account the implications of these transformations on inter-iteration dependences and does not explore partial loop unrolling. In addition, VHLS cannot reason about how this feature affects numerical accuracy; there is no guarantee that this transformation will not result in a catastrophically inaccurate implementation.

In response, this thesis proposes new methodologies and an associated tool—SOAP, a fully automatic source-to-source optimizer that augments VHLS—to optimize a given numerical C program using these transformations in tandem with conventional program transformations. The optimizer discovers not only one, but a wide spectrum of program candidates. When synthesized in VHLS, these candidates trade off three performance metrics of great importance to engineers: run time, resource usage and round-off error. Here, run time refers to the latency in clock cycles, resource usage refers to the number of look-up tables (LUTs) and DSP elements. Some of these performance metrics could be in conflict. For example, higher performance tends to require more circuitry, and how to resolve this trade-off depends on the user’s requirements. As a result, the tool produces a set of optimized programs, known as the Pareto frontier: those programs $P$ for which the tool has found no $P'$ that improves on $P$ in all three metrics. We thus go beyond existing literature by not only optimizing the precision requirements of an implementation, but changing the structure of the implementation itself.

The program optimization flow is safe and semantics-directed. Safety means that because we base our method on formal mathematics to optimize programs, our approach can be proved correct, in the sense that when executed using exact real arithmetic, the transformed version produces exactly the same output values as the original program. Semantics-directed transformation means that not only do we use program syntax, but also the semantics, i.e. the underlying meaning of programs such as inferred numerical accuracy, to guide optimization and guarantee safety properties of the optimized program. Our technique obtains when necessary, by analyzing the program, a bound and a round-off error bound on each variable in every program location. This information is then used to guide program optimization, by analyzing and manipulating not only the syntax, but also the semantics of programs.
Generating candidate optimizations naively would however produce a combinatorial explosion, even for small input programs. For instance, in the worst case, the parsing of a simple summation of \( n \) variables could result in \( (2n - 3)!! = 1 \times 3 \times 5 \times \cdots \times (2n - 3) \) distinct expressions modulo commutativity, i.e. we make use of associativity but ignore any distinction caused by commutativity [IM12; Mou11]. This is further complicated by distributivity as the expression \((a + b)^k\) could expand into an expression with a summation of \(2^k\) terms each with \(k - 1\) multiplications. Usually, for this reason, it would be infeasible to generate a complete set of equivalent expressions using the rules of equivalence, since an expression with a moderate number of terms will have a very large number of equivalent expressions, and this number grows faster than exponential rate with the increase of the number of terms. New approaches were therefore developed specifically to tackle the efficient discovery of equivalent structures in numerical programs. First, we invent a new intermediate representation, called metasemantic intermediate representation (MIR) to reduce the size of our search space without affecting its optimality. Second, we further reduce the effort of exploring the new search space by intelligently pruning the set of candidates as it progresses up the input program's abstract syntax tree.

All of the techniques above are designed with compositionality in mind. This means that we recursively break down each component we work on—such as programs and MIRs—into smaller components and use our analyses to calculate the results of each, such as accuracy, area and latency, and subsequently equivalent candidates; the final results are then constructed from those of the subcomponents. When compared with a global approach, there are two major advantages. First, because components are considered independent of each other, once analyzed the results can be reused in the analysis in the larger enclosing components. Another advantage of this is that although we will not formally prove the correctness of the methodologies in this thesis, they are designed to be easily expressible in a formal language, and by proving small lemmas, we can deductively prove the correctness of, for instance, a program-to-MIR translation.
1.1 Thesis Organization

Chapter 2 serves to explain various essential concepts used throughout this thesis, and discuss others’ work in depth to better motivate the thesis contributions. The techniques discussed in this thesis naturally extend HLS, a process to compile program written in HLLs such as C into circuits. We therefore start by introducing the advantages of HLS, compared against manual RTL implementations. We then bring rigorous approaches from software static analysis, specifically program semantics and abstract interpretation, to source-to-source transformation for HLS. We introduce existing intermediate representation (IR) designed for program optimizations, and highlight the advantages and disadvantages of them relevant to our requirements. We further explain existing program rewriting techniques for numerical accuracy, which inspired our efficient discovery of equivalent programs. Finally, we introduce the concept of loop pipelining and how restructuring numerical programs can optimize run time.

In Chapter 3 we propose new methods to automatically optimize the structure of arithmetic expressions for FPGA implementation as part of a HLS flow. This chapter introduces the basis of our approach to perform structural optimization on numerical programs, taking into account axiomatic rules derived from real arithmetic, such as distributivity, associativity and others. A new efficient method is proposed to generate a computable optimized subset of equivalent expressions from an original expression. Our approach explicitly target an optimized area/accuracy trade-off, by automatically rewriting arithmetic expressions, and analyzing each expression rewritten for its accuracy and area usage. This gives the synthesis tool the flexibility to choose an implementation satisfying constraints on both accuracy and resource usage. Using our technique to optimize the structure of a variety of real world and artificially generated examples in single-precision, we improve either their accuracy or the resource utilization by up to 60%.

Chapter 4 presents a similar source-to-source optimization targeting the trade-off between numerical accuracy and resource usage, but extends it to optimize general numerical programs, including if statements and while loops. Because there are infinite number
of ways to rewrite numerical C programs, and many of these rewrites produce programs that have the same resource usage, accuracy and latency properties, we introduce a novel expression-based IR called MIR to reduce the number of rewrites to explore. In Chapter 4 we explain in detail the structure of MIRs, and the back-and-forth translation between numerical C programs and MIRs. We efficiently discover equivalent structures in MIRs by exploiting not only the rules of real arithmetic, such as associativity and distributivity, but also rules that enable control-flow restructuring. Our numerical accuracy and resource usage analyses are further extended to analyze MIRs. Additionally, we broaden the Pareto frontier in our optimization flow to automatically explore the numerical implications of partial loop unrolling and loop splitting. In real applications, the tool discovers a wide range of Pareto optimal options, and the most accurate one improves the accuracy of numerical programs by up to 65%.

The optimization techniques we have discussed so far have only been limited to minimizing area and round-off errors of numerical programs. Often such optimizations result in programs with longer run time, yet there is potential for these transformations to significantly reduce the run time of numerical programs while improving resources and accuracy. In Chapter 5, we thus introduce a new analysis procedure to minimize yet another objective: the total run time of the optimized program. Together with accuracy and resource utilization, these three form the simultaneous goals we use to produce the Pareto frontier. In this chapter, MIRs are extended with new operators to allow for arrays and matrices in the source program, so that a wider range of practical numerical applications can be optimized. Numerical programs typically spend most of their run time in loops, hence state-of-the-art HLS tools use pipelining to schedule them efficiently. Still, the run time performance of the resultant FPGA implementation is limited by data-dependences between loop iterations. We thus present additional rewriting rules—memory access reductions—along with arithmetic identities and control-flow transformations to alleviate some of these dependence constraints. HLS tools cannot safely enable such rewrites by default because they may impact the accuracy of floating-point computations and increase area usage, whereas we optimize run time while controlling the implications on accuracy and area. Again, the tool reports a multi-dimensional Pareto frontier that the
programmer can use to resolve the trade-off according to their needs. When applied to a suite of PolyBench and Livermore Loops benchmarks, the tool generated programs that enjoy up to a $12 \times$ speedup, with a simultaneous $7 \times$ increase in accuracy, at a cost of up to $4 \times$ more LUTs.

Finally, Chapter 6 concludes this thesis by summarizing our research, and we discuss the potential for how future research into the structural optimization of numerical programs could further benefit the HLS community.

1.2 Contributions

In this section, we summarize the following original major contributions in this thesis:

- a new expression-based IR, *metasemantic intermediate representation* (MIR), developed based on the formal semantics of programs, to safely and considerably reduce the size of the search space of equivalent programs [GC15];

- an efficient algorithm to discover equivalent expressions and MIRs through bottom-up hierarchy, graph partitioning, and intelligent pruning of optimization candidates by performing accuracy, area estimation and latency analysis [Gao+13];

- we bring together standard program equivalences that do not affect program behavior (e.g. loop/branch splitting/merging, partial loop unrolling, rules that removes extraneous array accesses), and non-standard transformation rules (e.g. arithmetic rules), to be used in a novel way to significantly impact latency, resource usage and accuracy of a numerical program [Gao+16];

- an accuracy analysis based on abstract interpretation to calculate bounds on the output and their corresponding round-off errors of a given optimized expression [Gao+13] or program candidate [GC15];

- a scheduling analysis that estimates the latency and resource usage of a given optimized candidate [Gao+16]; and
• incorporating the above-mentioned techniques, the first optimizer to automatically and safely produce optimized programs (and subsequent RTL implementations with VHLS) on the three-dimensional Pareto frontier of options that trade off run time, accuracy, and area [Gao+16].

1.3 Publications

The original contributions in this thesis have lead to the following peer-reviewed publications:


Background

The main focus of this thesis is to devise optimization methods to efficiently rewrite numerical programs by automatically varying the structure of their control- and data-paths for an improved trade-off between numerical accuracy, throughput and resource utilization for HLS. This chapter therefore provides a review of recent developments in related work and various concepts that closely relate to the foundation of this central problem. This section serves to highlight the main research topic to be discussed in depth in each section.

We introduce various computing architectures, and explain the pros and cons of each one in Section 2.1. We then explain in detail how FPGAs can be used as an alternative to general-purpose processors such as central processing units (CPUs) and general-purpose graphics processing units (GPUs) to accelerate computational-intensive applications, by drawing comparisons of architectural differences to processors. This is then followed by an overview of how a traditional synthesis tool flow compiles RTL implementations into circuits by taking Altera Quartus II [Alt10] as an example.

Because of the ever-increasing complexity of applications designed into circuits, low-level approaches such as RTL design become intractable for humans [Gaj+92], and a gradual adoption of HLS tools has been observed in the FPGA community in recent years. For this reason, in Section 2.2, we review the current state and advantages of HLS, and describe the stages employed by these tools to compile a program in a high-level language into digital circuits. Because run time is one of our optimization objectives, we use modulo systems of difference constraints (SDC) scheduling to further examine in greater detail how it can make loops to run as efficiently as possible by pipelining them. This scheduling algorithm is used by HLS tools such as LegUp [LU].
The methods discussed in this thesis rely heavily on the static analysis of numerical programs based on abstract interpretation [CC77]. In Section 2.3, we therefore review abstract interpretation, a theory to create an abstraction for mathematical objects that are not directly computable—because they are undecidable, intractable or both—in order to compute an approximation of them efficiently. We informally introduce this theory, by way of reviewing it with a progressive program analysis example.

IRs are often used to facilitate program optimization and program analysis. Section 2.4 we therefore study in detail the existing IRs designed for program optimization. We take a closer look at how the static single assignment (SSA) and control-flow graph (CFG) approach of low-level virtual machine (LLVM) [LLIR] encodes a simple program example. As this technique limits itself to optimizing a single IR of a program, a novel approach known as equality saturation [Tat+] which uses a data structure to simultaneously represent multiple optimization candidates, is further examined.

In Section 2.5 we review prior work on restructuring arithmetic expressions and numerical programs. We examine existing optimization passes in software compilers and HLS tools. Since HLS tools are LLVM-based and compile C or C++ programs, they borrow the concept from software compiler frameworks, such as LLVM, of including optimization passes that may improve performance of floating-point computations, while potentially sacrificing numerical accuracies. On the other hand, we look at methods that could improve accuracy by simultaneously exploring multiple rewrites of expression candidates. Due to the calculation of the full set of equivalent programs often being computationally intractable, these methodologies either make extensive use of heuristics, or abstract interpretation techniques introduced in Section 2.3, in order to compute an under-approximation (a potentially smaller set) of the set of all equivalent programs.

### 2.1 Field-Programmable Gate Arrays

When it comes to implementing computations, we often choose from a spectrum of computing machines. These choices range from those with fixed architectures that
compute by executing *software* designs such as CPUs and GPUs, those that implement custom *hardware* architectures, such as FPGAs, to those with custom integrated circuits to carry out computations, *i.e.* application-specific integrated circuits (ASICs).

There has been a great amount of effort in recent decades to make fixed-architecture machines run as fast as possible; many novel and intricate ideas were proposed and we now have a variety of general circuits in CPUs to improve their performance [HP11]. For instance, CPUs could have a pipeline that spans several clock cycles to fetch and decode instructions, and access data from memory or registers to carry out computations. At the same time, they could make predictions about the branches taken in control-flows, as the pipeline must be flushed if an incorrect instruction is fetched, incurring a penalty in speed. Superscalar architecture and out-of-order execution are used to increase instruction-level parallelism. They may also exploit data-level and thread-level parallelism, in order to maximize opportunities to parallelize computations. These are just the tip of the iceberg, as many other architectural advancements exist.

The great majority of fixed-architecture computing machines are based on the *von Neumann architecture*, which consists of three parts: a computation unit, a memory and a bus between them to move data back and forth. Often applications running on these machines could spend a majority of their time and energy to move data and instructions in the memory from/to the right location in the processor as fast as possible, in order to carry out arithmetic computations. Because computational tasks frequently reuse input data and intermediate results, a hierarchy of caches, in tandem with cache-aware compiler optimizations [KW03], are often used to mitigate the costs of exchanging data between the processor and the memory. Despite these optimization efforts to run software code as fast as possible, the processor-memory bus, which is often referred to as the *von Neumann bottleneck* [Bac78], inherently exists and remains the limiting factor of performance in the architecture. This phenomenon is known to many as *hitting the memory wall* [Bac+13; WM94].

Custom architectures in general achieve much higher performance, thanks to their ability to implement arbitrary digital circuits specifically designed for the application under
consideration, and spatially distribute memory bandwidth and computations. This is in stark contrast to microprocessors such as CPUs and GPUs, which utilize general-purpose circuits to cope with a wide-range of applications. ASICs provide the best power efficiency and performance among all above architectures, however they are often associated with long development cycles and high costs; any updates to the design would require a complete and expensive re-spinning of the circuits [Bac+13], as they are inherently non-programmable. FPGAs provide a good trade-off between processors and ASICs. Not only do FPGAs have better performance and power characteristics than fixed architectures, they also offer high programmability which makes FPGAs cost-effective low-volume ASIC replacements [PB04; Bac+13]. At the same time, with a much shorter development period than ASICs, a hardware design on FPGAs can be implemented with a much lower cost. With a shorter time to market, it further enables a substantially larger profit by a competitively early market entry [Sem].

For the above reasons, being able to leverage parallelism from bit-level all the way to the loop- and task-level, FPGAs have been increasingly used as high-performance and low-power alternatives to CPUs and GPUs for many classes of applications [Bac+13; Bro+10; SF08]. For example, Thomas et al. [Tho+09] reported a FPGA-based random number generator can obtain a \(260\times\) speed-up, while costing less than 1\% of energy to produce each random sample, when compared to its software counterpart running on a CPU. Microsoft initiated a mid-scale deployment of Stratix V FPGAs in their data center, improving the throughput of their Bing web search engine by a factor of 95\% [Put+14].

### 2.1.1 FPGA Architecture

FPGAs owe their high performance and power efficiency to the design of the architecture, we thus use Altera Stratix V [Alt15d] as an example to explain the architecture. The Stratix V fabric contains a two-dimensional array of logic array blocks (LABs). Each LAB in turn consists of an array of 10 adaptive logic modules (ALMs). Figure 2.1 shows a high-level block diagram of an ALM in Stratix V. In an ALM, multiplexers can be configured to choose whether full adders and registers are used. Dedicated full adders enable more complex Boolean functions to be implemented in a single ALM, whereas
the use of registers, which store intermediate values, determines whether the circuit is combinational or sequential. The two LUTs in an ALM can be configured to compute a combination of two arbitrary Boolean functions, each with up to 5 inputs from 8 inputs in total. Stratix 10, slated to be released in the next couple of years, has up to 1.87 million ALMs and 7.47 registers in total for the most demanding applications [Alt15b]. Interconnects, another class of key configurable resources on FPGAs, enable the inputs and outputs of ALMs to be wired, in order to form larger and complete circuits from the two-dimensional array of ALMs.

**Figure 2.1.** A high-level block diagram of an ALM in Stratix V, from Stratix V Device Handbook [Alt15d].

FPGAs with enough ALMs and interconnects can implement arbitrary digital designs. This versatile architecture therefore overcomes the memory wall problem by not restricting itself to the von Neumann architecture. As we have mentioned earlier, FPGAs can implement a circuit that is individually tailored for the application, in contrast, CPUs have general-purpose circuits designed for a wide range of applications, which may therefore have lower power-efficiency and performance. Moreover, unlike the CPU which only has a small set of registers, the FPGA with its flexibility and abundant registers, allows designs to distribute memory blocks and computation units and place them in close proximity.

Traditionally, multipliers, when implemented as soft-logic in FPGAs, cost a large number of ALMs. Stratix devices thus further include an array of hardened components to carry
out arithmetic operations distributed on the FPGA fabric, known as DSPs blocks, or simply DSPs. Because of the dedicated hardened circuits, DSPs compute faster than arithmetic operators formed by ALMs only, meanwhile they free ALM resources to perform more non-arithmetic computations. In Stratix V, each DSP is paired with a LAB. These DSPs are fracturable, i.e. they can be configured in combinations to perform a wide variety of arithmetic operations, ranging from those using a single DSP element to synthesize three multipliers each with two 9-bit inputs, up to those combining four DSPs to form a complex-number multiplier with two 27-bit inputs [Alt15d]. Computations with larger inputs can also be implemented by using ALMs and DSPs to form larger arithmetic circuits. Finally, Stratix 10 will introduce hardened floating-point DSPs, enabling IEEE 754 [ANS08] single-precision floating-point additions and multiplications, achieving a performance of up to 10 tera floating-point operations per second (TFLOPS) [Alt15c]. These DSP blocks can also be adapted to multiply fixed-point inputs.

DSPs accelerate arithmetic computations, however they need to be supplied with inputs as fast as they can process to fully utilize them. In general, in most applications, data are frequently reused by the same computation unit. Stratix V therefore includes dedicated embedded memory called M20K blocks (20 Kb storage) to be arranged and combined into dual-port random access memories (RAMs). Half of the LABs on the device, called memory logic array blocks (MLABs) can also be configured to become a 640-bit RAMs. These memory blocks are distributed across the FPGA fabric, so that DSPs may find them in proximity.

2.1.2 RTL Design Flow

Modern FPGAs—with up to several million LUTs, and thousands of embedded memory and DSP blocks, wired through a programmable fabric of interconnects—are humanly intractable to program at the granularity of these individual components [KD08]. FPGA applications are thus commonly written in RTL hardware description language (HDL), such as Verilog [IEE06] and VHDL [IEE09]. These HDL source programs implement the desired hardware by describing the logic between registers. Electronic design automation
(EDA) tools can then automatically translate these descriptions into hardware circuits in FPGAs.

EDA tools, go through several stages to synthesize HDL source code into circuits. To explain these stages in depth, we take Altera Quartus II [Alt10] as an example design flow, shown in Figure 2.2.

![Quartus II design flow](image)

**Figure 2.2.** Quartus II design flow.

Quartus II starts its compilation of the RTL program by verifying source code for syntax and semantic errors and design specification for inconsistencies, then applies a methodology, called *technology mapping*, which maps a graph of device-independent logic gates in logic expressions onto a network of functional blocks (such as LUTs, DSPs and memory blocks) in the target FPGA device [CP08]; this generated network is known as a *technology-mapped netlist*. In this process, synthesis tools may optimize the circuit by performing additional transformations such as redundant logic removal [Alt10].

The following stage, *place & route*, utilizes a heuristic placement algorithm, which takes as it inputs the netlist, together with a device map showing the location of each of its functional units, in order to select a legal location on the FPGA for each functional block in the netlist, such that the routing of these blocks is optimized [Bet08]. In general, synthesis tools allow some freedom in the user’s preference of the placement of circuit. Additional automated optimizations may be applied to improve performance. For example, Quartus II has the option to enable *register retiming* [Alt10], which allows registers to move across combinational logic to reduce *critical path* delay, i.e. the longest delay required for an output of any source register to propagate to the input of any target register in the circuit. The end result of this step is a circuit fully mapped on the target FPGA.

In the following step, *timing analysis*, the tool computes the longest delay of all critical paths, which determines the maximum frequency at which the application can run. Users
can also inspect the list of critical paths and their delay statistics, so that one may focus their effort on optimizing the timing of these critical paths by, for instance, splitting them up by adding registers.

In the simulation stage, the resulting design is simulated using EDA simulation tools.

The final step, programming, is to translate the circuit generated by the tool into a bitstream, which is a binary data file used to program the FPGA. Similar to processors, which can be programmed by incrementally reading and executing instructions from an executable program file, a bitstream is used to program the individual components such as LUTs, DSP blocks, dedicated memory blocks and interconnects on an FPGA, so the circuit is formed on the device. The difference between them is that while processors continuously read instructions from memory, FPGAs are typically programmed only once during the initial setup, and the bitstream data are used spatially to infer a circuit rather than sequentially as instructions [Guc08].

2.2 High-Level Synthesis

High-level synthesis (HLS) is the process of compiling a high-level representation of an application (usually in C, C++ or MATLAB) into a RTL implementation [CM08; Gaj+92]. In turn, this RTL design can be synthesized into a circuit and programmed onto the FPGA device. With modern HLS tools, some applications are synthesized to have similar performance when compared with hand-crafted RTL implementations [BDT10b].

The major advantage of HLS tools is that they enable us to work in a HLL, as opposed to facing labour-intensive tasks such as optimizing timing and designing control logic in the RTL design process. This allows application designers to focus instead on the algorithmic and functional aspects of their implementation [CM08], without concerning themselves with the intricate details of manual RTL designs.

Another advantage of using HLS tools is that they are in general more productive and less error-prone to work with, when compared with traditional RTL tools. The reasons
are two-fold. Firstly, a C description is smaller than a traditional RTL description by a factor of 10 [CM08; BDT10b]. Secondly, RTL design can be notoriously difficult to debug, whereas C code can be easily tested on an ordinary microprocessor, and mature debug and analysis tools for C are freely accessible [Can+13].

HLS tools further benefit us in their ability to automatically search the design space with a reasonable design cost [BDT10b], potentially exploring a large number of trade-offs between performance, cost and power [McF+90]. Traditionally, this is generally much more difficult to achieve in RTL designs because of their low-level nature.

With recent advancements in this area, HLS tools have received a resurgence of interest. Many commercial tools have been released, such as Catapult High-Level Synthesis [MG], Impulse C [Imp] and PICO [Sch+02], to meet the burgeoning demand from the FPGA community. Xilinx incorporates a sophisticated HLS flow for C/C++ named VHLS into its Vivado design suite [Xil15], and their SDAccel Development Environment [Wir14] for C/C++/OpenCL allows data centers to leverage FPGAs. Altera’s HLS solution is their Altera SDC for OpenCL [Alt15a] which accelerates OpenCL applications on their FPGA devices. Besides commercial tools, many open-source HLS tools have also been released in recent years, such as ROCCC [Naj07], Trident [Tri+05] and LegUp [LU]. LegUp is now gaining significant traction in the research community.

## 2.2.1 HLS Design Flow

This section provides an overview of the stages taken by the HLS tool to compile a C program into RTL implementation, by using LegUp [LU; Can+13] as our example. LegUp is an HLS tool which compiles programs to run on a hybrid software/hardware architecture, and its design flow is shown in Figure 2.3, which consists of three major stages to be explained below.

The first stage is to determine which parts of an application on the function-level are suitable candidates to be synthesized into hardware circuits, while the rest can be run on a soft-processor. This stage starts by compiling a C source program into a software
executable targeting an FPGA-based MIPS processor. This processor has additional circuits designed to profile the software implementation of the original application. By running the compiled application on this processor, this profiling ability allows the processor to use statistics such as number of clock cycles, power and cache misses to identify problematic parts of the program at the function level that will benefit from a hardware redesign, so that the power efficiency and run time could be improved [Can+13].

After identifying functions of the application to be implemented as part of a hardware architecture, the next stage is then to synthesize hardware designs from these functions. LegUp’s synthesis toolchain is based on the LLVM compiler infrastructure [LA], and it synthesizes C functions into circuits in a series of steps.
It starts by using the LLVM front-end to compile a C function into *low-level virtual machine intermediate representation* (LLIR), a platform-independent IR that is capable of cleanly representing HLLs [LLIR], conventional and HLS-focused compiler optimization passes are used to transform the IR program, such that the result when synthesized will have better performance when running on the FPGA.

This is then followed by the HLS tool flow, which consists of four logical steps: allocation, scheduling, binding and RTL generation. The first step, *allocation*, extracts information from the application and user requirements to be used in subsequent stages, *e.g.* modules and RAM blocks to be synthesized on the target device. This is then followed by *scheduling*, which assigns the start and end states to each LLVM instruction in a finite state machine [LU], using a scheduling algorithm based on the formulation of SDC [LU; Can+13; CZ06]. Many applications spend most of their time in loops; a scheduling technique known as *loop pipelining*, is therefore used in HLS tools to make them run efficiently. This technique admits greater parallelism by allowing instructions in consecutive loop iterations to overlap as much as possible. LegUp uses *modulo SDC scheduling* [Can+14], which we will cover in Section 2.2.3, to minimize the wall-clock time of a pipelined loop.

The third logical step, *binding*, assigns each operator in the program to functional units to be synthesized into hardware, and maps program variables to registers. The rationale behind this step is that operators such as multipliers and dividers that tend to use a lot of LUTs and DSP blocks can be shared temporally. Sharing these functional units requires multiplexers, which is relatively expensive to implement in FPGA. Each assignment of an operator to a functional unit is thus associated with a cost. The problem of minimizing this cost is called the assignment problem, which is efficiently solved in LegUp with a polynomial time complexity using the Hungarian algorithm [Can+13; Kuh10]. Finally, the *RTL generation* step gathers information produced from the previous three steps, to generate Verilog source code corresponding to the C function being compiled.

The third, and also the final stage, is to integrate software and hardware components of the application into the FPGA device. Following [Can+13], we explain it as follows. Firstly, custom accelerator circuits generated by HLS, a MIPS processor, and communication interfaces between them are synthesized and programmed into the FPGA device. Because
some of the functions in the original C source code were implemented as hardware accelerators in the HLS compilation flow, LegUp replaces them with wrapper functions which can invoke the hardware accelerators at runtime. Finally, this modified source code can then be compiled into a MIPS binary to be executed on the FPGA.

2.2.2 Loop Pipelining

Loop parallelism, and consequently, program run time is one of the main optimization objectives we optimize later in Chapter 5. Hence in this subsection we first introduce the concept of loop pipelining. We consider our example program `dotprod` in Figure 2.4, which computes the dot-product, \( d \), of two arrays \( A \) and \( B \) of floating-point values. We assume both arrays are stored in the same RAM, which has one read port, and accessing this RAM has a one cycle latency. We further assume no limits on the number of arithmetic operators that can be allocated, floating-point multipliers and adders are both fully pipelined, and use 7 and 10 cycles respectively to produce outputs.

```c
#define N 1024
float dotprod(float A[N], float B[N]) {
    float d = 0.0f;
    for (int i = 0; i < N; i++) {
        d = d + A[i] * B[i];
    }
    return d;
}
```

Figure 2.4. A simple dot-product example which calculates the dot-product of two arrays \( A \) and \( B \), each with 1024 elements.

A trivial way to schedule the loop in `dotprod` is to allow each iteration to complete before starting the next iteration; this is however not very efficient. As we can see in Figure 2.5, with a good schedule, operations across loop iterations can often temporally overlap, giving way to parallelism and improve performance of the loop execution. In Figure 2.5, iterations are laid out in rows, each clock cycle is a column, \texttt{mul} and \texttt{add} are multiplication and addition respectively, \( A[0] \) and \( B[0] \) are read accesses from the two arrays, and the arrows indicate the data-flow of \( d \) across iterations. This schedule
allows consecutive iterations to start every 10 cycles; and this number of clock cycles between the start of consecutive iterations is known as the initiation interval (II). Loop iterations in this schedule repeat for 1024 times (the trip count, $N$), and each iteration requires 20 cycles (the depth, $D$, of the loop), as a result the overall latency $L$ of this loop is $(N - 1) \times II + D = (1024 - 1) \times 10 + 20 = 10,250$ cycles.

Any valid schedule of this loop must satisfy the constraints imposed by data-dependences. For instance in our example, it is clear that in a single iteration, in the loop body, multiplication of $A[i]$ and $B[i]$ must precede addition of $d$ and the multiplied result. Furthermore, in the $(i + 1)$-th iteration, access to the variable $d$ must wait until $d$ is updated with a new value in the $i$-th iteration, data-dependences therefore also exist on $d$ across loop iterations. We call the former kind of dependences intra-iteration dependences and the latter inter-iteration dependences.

Besides data-dependence constraints, the number of resources available also affects loop scheduling. For instance, under our assumption in dotprod the RAM can only be read once per cycle, our schedule thus should avoid reading from the same RAM in the same clock cycle. We say a schedule is optimal, in the sense that the overall latency $L = (N - 1) \times II + D$, is minimized, while none of the constraints are violated. However with a much more complex program, finding the optimal schedule is often an intractable task. Limits on resource availability, along with dependence constraints, make scheduling an NP-hard problem which is difficult to solve optimally and efficiently [Hwa+91]. In the following part of this section, we discuss modulo SDC scheduling [ZL13; Can+14] used in HLS tools, such as LegUp, to efficiently attack the scheduling problem.
2.2.3 Modulo SDC Scheduling

Many algorithms exist to schedule pipelined loops. For example, Fan et al. [Fan+08] proposed that a schedule can be found by formulating the constraints into *asatisfiability modulo theory* (SMT) problem and use an SMT solver to modulo schedule operations. An alternative technique, *iterative modulo scheduling* (IMS) [Rau94], has been a widely adopted by compilers that use software pipelining to schedule instructions for *very long instruction word* (VLIW) processors [MS03]. This method has also been widely adopted in HLS tools such as PICO-NPA [Sch+02], Trident [Tri+05] and LegUp [Can+13; Can+14]. IMS for software pipelining [Rau94], however did not consider operator chaining, *i.e.* allowing operations with combinational logic in a data-flow sequence to be carried out in the same clock cycle. Schreiber et al. [Sch+02] in their adoption of IMS in HLS, found operator chaining to be non-trivial in IMS and requires static timing analysis of combinational components [Can+14]. A new method, the modulo SDC scheduling algorithm, has thus recently gained traction and has been used by Canis et al. [Can+14] in their LegUp and by Zhang et al. in [ZL13], because an SDC formulation is more suited to model the effect of chaining operators. For this reason, an overview of the modulo SDC scheduling approach is provided in this subsection.

**Constructing the Data-Dependence Graph**

In the first stage of modulo SDC scheduling, dependence relations are extracted from the body of this loop. These dependence relations form a dependence graph, where vertices are operations, and edges between pairs of vertices indicate dependence relations. This dependence graph can subsequently be used to derive data-dependence constraints. Figure 2.6 shows the complete dependence graph of *dotprod*’s loop body.

Intra-iteration dependence edges in the graph are each labelled with \( (l, 0) \), a pair of attributes of integers accordingly. The first integer \( l \) signifies the number of clock cycles that must elapse between the start of the predecessor and the successor operations. The latter value 0 indicates that the dependence occurs in the same iteration. To illustrate,
Figure 2.6. The dependence graph formed by the data-dependences in the loop body of the dot-product example in Figure 2.4. The dashed edge highlights the inter-iteration dependence.

the edge between the multiplier “×” and adder “+” has an attribute ⟨7, 0⟩, because “×” takes 7 cycles to generate an output.

Additionally, inter-iteration dependences create elementary cycles in the dependence graph. For example, in each iteration the initial value of \(d\) depends on the final value of \(d\) from the previous iteration. In this graph we thus add an edge from the output of the addition to the variable \(d\). We further describe that this dependence has a *dependence distance* of 1, as 1 iteration must elapse between the start of each pair of value updates and its corresponding use. This edge is then assigned an attribute ⟨10, 1⟩ which signifies that the adder has a latency of 10 cycles, and this dependence has a distance 1.

**Finding the Minimum Initiation Interval**

Modulo SDC scheduling owes its efficiency to assuming an initial constant II and attempting to search for a schedule that satisfies all constraints. This search stops if a feasible schedule is found, otherwise II can be incremented by 1 and the search is repeated until we discover a valid schedule. To begin, we can find a lower bound on II, which we call the *minimum initiation interval* (MII), as our initial constant II. The MII is computed such that all schedules with an II less than MII violate some constraints. For each of the inter-iteration dependences and resource constraints, an MII can be found, respectively known as *recurrence-constrained minimum initiation interval* (RecMII) and *resource-constrained minimum initiation interval* (ResMII) [Rau94; Can+14; ZL13].
We first introduce methods to compute both values, then the overall MII can then be computed using the following equation:

\[ MII = \max (\text{RecMII}, \text{ResMII}) \tag{2.1} \]

Firstly, we compute ResMII, by finding the most constrained resources in the loop, as these limits impact the ResMII value. For example, \text{dotprod} does not impose limits on the number of floating-point operators that can be allocated, but assumes there is a constraint which restricts the rate of memory accesses, \textit{i.e.} only one read is allowed in each clock cycle. Because each iteration requires two accesses to the same memory, a ResMII = 2 will thus fully occupy the RAM throughput. To generalize this to all loops, we consider for each type of operation \( \otimes \) being used in the loop, the number of available resources \( r_{\otimes} \) for \( \otimes \) and the number of occurrences \( n_{\otimes}(G) \) of \( \otimes \) in the loop dependence graph \( G \), to evaluate \( \lceil n_{\otimes}(G)/r_{\otimes} \rceil \), the maximum ratio constitutes the final ResMII, or equivalently:

\[ \text{ResMII} = \max_{\otimes \in \text{OpTypes}} \left\lceil \frac{n_{\otimes}(G)}{r_{\otimes}} \right\rceil, \tag{2.2} \]

where OpTypes is the set of all types of operations, \textit{e.g.} array accesses, arithmetic units, and others, used in the loop.

The second step is to evaluate a minimal RecMII by ensuring for all cycles \( c \) in the graph \( G \) the following inequalities hold:

\[ \forall c \in \text{Cycles}(G) \colon \sum_{e \in c} \text{lat}(e) - \text{RecMII} \times \sum_{e \in c} \text{dist}(e) \leq 0, \tag{2.3} \]

where \( \text{Cycles}(G) \) computes the set of all cycles in the graph \( G \); \( e \in c \) enumerates all edges in the cycle \( c \); and for an edge \( e \) between two vertices \( v_1 \) and \( v_2 \) of the form \( v_1 \xrightarrow{(l,d)} v_2 \), \( \text{dist}(e) \) and \( \text{lat}(e) \) respectively evaluate to the latency \( l \) and dependence distance \( d \). Hence, \( \sum_{e \in c} \text{lat}(e) \) and \( \sum_{e \in c} \text{dist}(e) \) respectively sum the latencies and dependence distances along all edges in the cycle \( c \). Equivalently, we can derive an equation for RecMII for the graph \( G \):

\[ \text{RecMII} = \max_{c \in \text{Cycles}(G)} \left[ \frac{\sum_{e \in c} \text{lat}(e)}{\sum_{e \in c} \text{dist}(e)} \right]. \tag{2.4} \]
For example, in the dependence graph (Figure 2.6) of our simple program (Figure 2.4), one cycle, \( \rightarrow+ \), exists and the sums of latencies and dependence distances along this cycle are 10 and 1 respectively, thus the RecMII of this graph is \([10/1] = 10\).

The simplest possible method of finding RecMII is therefore to enumerate all cycles in the graph and compute the ratio between sums of latencies and dependence distances. Unfortunately in the worst case, the number of cycles is exponential in the number of edges in a graph, this approach could become intractable for large loops. An alternative method based on the Floyd-Warshall shortest path algorithm [Flo62] which runs in polynomial time, is thus proposed in [Rau94] to efficiently find RecMII.

**Scheduling Operations**

After assuming a tentative constant II, in modulo SDC scheduling, we try to construct an SDC problem in order to solve for the schedule. We aim to assign each operation, corresponding to each vertex \( v \) in the graph, to a time slot \( s_v \) when it begins its operation. While in this process, the scheduling must ensure that no assignment violates data-dependence and resource constraints. For instance if a multiply operation is allocated with a time slot in the second clock cycle, in each iteration it will start computation in the second clock cycle of that iteration.

To begin, we ignore the resource limits for now and formulate an SDC problem for the dependence constraints. For each dependence edge \( u \rightarrow (d,l) v \) from vertex \( u \) to vertex \( v \) in the graph, it is possible to write down the following inequality, where \( s_u \) and \( s_v \) are respectively the time slots for vertices \( u \) and \( v \):

\[
s_u - s_v \leq \text{II} \times d - l. \tag{2.5}
\]

For instance, the edge \( \times \rightarrow (7,0) + \) is an intra-iteration dependence, hence II does not constrain the scheduling relation between these two operations and we substitute \( d \)
with 0 and l with 7, and the II term vanishes, to derive (2.6) below, and the back-edge 
\[ \langle 10,1 \rangle \rightarrow d \] produces the following inequality in (2.7):

\[
\begin{align*}
    s_x - s_+ & \leq -l, \\
    s_+ - s_d & \leq l - 10.
\end{align*}
\]

Besides dependence constraints, additional constraints are used to limit the length of critical paths in combinational logic, such that for instance, a long chain of additions can be broken down into multiple cycles to guarantee the frequency requirement. For all paths \( u \rightarrow \cdots \rightarrow v \) between inter-iteration dependent vertices \( u \) and \( v \) consisting of only combinational logic, it is possible to estimate a critical path delay \( \text{delay}(u, v) \). This critical path delay is defined as the largest sum of propagation delays of each intermediate operation along any combinational path from \( u \) to \( v \). For a pair of dependent vertices \( u \) and \( v \) such that \( \text{delay}(u, v) > T_{\text{clk}} \), where \( T_{\text{clk}} = \frac{1}{f_{\text{clk}}} \) and \( f_{\text{clk}} \) is the target clock frequency, we can create the following constraint [CZ06]:

\[
s_u - s_v \leq -\left\lceil \frac{\text{delay}(u, v)}{T_{\text{clk}}} \right\rceil + 1.
\]

This inequality ensures that for any the critical path delay between \( u \) and \( v \) greater than \( T_{\text{clk}} \), this path cannot be scheduled within one clock cycle and it should be split into at least \( \left\lceil \frac{\text{delay}(u, v)}{T_{\text{clk}}} \right\rceil \) cycles.

Besides dependence and frequency constraints, Zhang et al. [ZL13] further introduces optional ones, such as lifetime constraints, which aim to minimize the register requirements, and relative timing constraints, which can be used to satisfy the timing requirement of user-specified I/O protocols.

The rationale of using the SDC formulation to model constraints is that the feasibility of an SDC system and the corresponding solution, if it exists, can be found efficiently. More precisely, using the Bellman-Ford algorithm [Sch05], it can run in \( \Theta(ln) \) time, where \( l \) is the number of constraint inequalities and \( n \) is the number of variables [ZL13]. Additionally, an SDC problem can be incrementally solved, i.e. a new solution, if exists,
can be updated in $O(m + n \log n)$ time when a constraint is added or removed, by using the algorithm presented by Ramalingam et al. [Ram+99]. In contrast, traditional *integer linear programming* (ILP) scheduling techniques make use of $O(mn)$ variables to represent a scheduling problem, where $m$ is the number of time slots and $n$ is the number of operations [Hwa+91], and solving this ILP problem often demands expensive branch-and-bound procedures [ZL13] as ILP is NP-complete [Kar10].

Unfortunately, resources constraints, because of their non-linearity, cannot be easily expressed as SDC constraints. Therefore, a data structure called the *modulo reservation table* (MRT) is used to keep track of resource constraints as the loop is incrementally scheduled [Can+14]. The MRT has $II$ columns and each row tracks an available resource. When a certain resource is used in the time slot $s_u$, the MRT records an entry for this resource in column $s_u \mod II$ and the corresponding row of this resource. To illustrate, consider our example in Figure 2.4, which assumes a single read to the memory in one cycle for both arrays. The MRT thus has $II = 10$ columns, and 1 row for accessing the memory. If $A[i]$ is assigned with a time slot 0, then a schedule assigning $B[i]$ to the same time slot must be invalid, because a record exists for $A[i]$ in row 1 column 0, and thus $B[i]$, which is competing for the same resource, hence must be scheduled in a later time slot.

A typical modulo SDC scheduling algorithm begins with a schedule without resource constraints. A priority function is then used to sort all resource-constrained operations by perturbation, i.e. we place greater importance to operations that have a larger impact on the schedule when they are moved [Can+14]. For each operator $u$ in this sorted list, if $u$ is currently scheduled at time slot $t_u$ and does not have a resource conflict in the MRT, a new constraint $s_u = t_u$ is constructed, otherwise a different constraint $s_u \geq t_u + 1$ is used to ensure the operation $u$ is scheduled at least one cycle later, so that it does not compete for the resource in time slot $t_u$. This newly created constraint is then tentatively added to the SDC problem. For a feasible resulting SDC problem, a new solution can be found incrementally, otherwise the algorithm backtracks to a latest feasible SDC formulation and tries to schedule other operators before $u$. A time budget can also be used to limit the number of attempts to schedule resource-constrained operators; if a valid schedule cannot
be found under the given budget, the II can be incremented by 1 to relax the dependence and resource constraints, and the above mentioned procedure can be repeated until a valid schedule is found.

2.2.4 Obstacles in Adoption

High-level synthesis, with its development-cost advantage over traditional RTL design paradigm, is gaining traction in the circuit design community. It is, however, in its early phase, and these tools still pose challenges in terms of using them as early adopters.

HLS tools may have limited support for HLL constructs. For instance, VHLS requires pointer arrays to reference values or arrays of values, platform-specific functions such as `memcpy()` and `memset()` are supported but `const` values must be used, and finally, while tail-recursive functions written with C++ template constructs can be transformed into loops in compile time, recursion in general cannot be implemented [Xil12]. Additionally, software programs often rely on libraries, many of which are platform-specific, whereas in HLS, these libraries may not be appropriate and may likely be unavailable. These above limitations make migrating existing software source code to a functional HLS design a demanding task.

Optimizing C code for HLS could be a laborious process and requires expertise in hardware design. Although software compilers face similar challenges to make programs run faster, experienced programmers can often manually fine-tune software implementations for performance. Because of the flexibility of HLLs, currently it is difficult for designers to apply common intuitions, and the quality of the synthesis result may be difficult to predict [Gup+04]. Winterstein et al. [Win+13] implemented K-means clustering algorithms in HLS, and discovered that with extensive manual code transformations and `#pragma` statements that are specific to VHLS, the tool can be persuaded to produce an efficient circuit. When compared to the RTL counterparts, their HLS designs achieved up to 40% of the performance in terms of area-time product. Zhang et al. [Zha+15] implemented a convolutional neural network (CNN) accelerator in VHLS, they optimized their design by program transformations techniques such as loop interchange, tiling,
pipelining and unrolling, and noticed that by enumerating combinations of loop tile sizes, loop nest ordering and unroll factors, they were able to select the best implementation by analytically estimating the throughput of each. Similarly, Suda et al. [Sud+16] explored the design space of their CNN accelerator by solving a resource-constrained throughput optimization problem, in order to generate a high-performance CNN accelerator to be synthesized in the Altera OpenCL compiler [Alt15a]. HLS tools can provide some syntactic constructs to automate lower-level code transformations such as instruction parallelism, loop pipelining and unrolling. It is, however, up to the engineer to decide how to utilize these transformations, and to determine whether they will improve the design. Higher-level optimizations such as the manual design space exploration explained in earlier examples, may allow tools to vastly improve the performance of HLS applications. Automating these techniques in HLS, however, remains a significant challenge.

HLS tools can often make use techniques such as loop pipelining discussed in Section 2.2.3 to detect and exploit parallelism at the instruction-level. Coarser-grain parallelism, which is tightly associated with the algorithmic details, is however much more complex [Nan+16], and many optimization opportunities are not yet exploited by HLS tools. For instance, unlike the CPU, which has a monolithic storage and thus their performance is often limited by the memory wall, the FPGA has dedicated RAM blocks in the logic fabric to distribute memory bandwidth via data reuse. HLLs such as C were designed with a mindset of the von Neumann architecture, and the source code in C typically does not specify how the memory hardware is utilized. For this reason, HLS tools must be able to intelligently partition a monolithic memory into smaller chunks that can be accessed in parallel, in order to maximize performance; how to attain this is still a challenging research area [Con+11a; Con+12; Wan+13; Win+15].

Circuits produced by HLS tools are expected to be semantics-preserving. This means that they should be functionally equivalent to the original C programs; in other words, for any given program inputs, the tools should guarantee that the computed outputs from the original C code and the synthesized circuit should be identical. A different class of optimizations, which we call lossy optimization, break this promise by optimizing data-paths in a way that may impact numerical accuracy. HLS tools could benefit from
these optimizations in the future, which bring further performance improvements that cannot be attained by traditional optimizations alone. Because these approaches could affect numerical accuracy, performance optimization and round-off error analysis may be carried out simultaneously. We will further discuss lossy optimization methods such as expression balancing enjoyed by VHLS and LegUp in Section 2.5.

2.3 Program Analysis and Abstract Interpretation

As our way of living is becoming increasingly dependent on programs, errors in safety-critical system can incur huge expenses, and even cost lives. For example, the maiden flight of Ariane 5 resulted in a failure, because of a software instruction failed to convert a 64-bit floating-point number into 16-bit signed integer, as the result was too large to be represented [Dow97]. The Patriot defense system failed to intercept an incoming missile because of an accumulated round-off error in the system’s internal clock, which resulted in the deaths of 28 people in 1991 [Off92]. Static analysis, a process of analyzing a piece of program written in an HLL without executing it, is therefore a research topic of great importance to prevent similar catastrophic errors and mitigate the cost of failure in the future.

It is unfortunate that because of the halting problem [Tur37] and a direct consequence of it, Rice’s theorem [Ric53], any nontrivial property on the outcome of a program is in general undecidable. This means that an interesting property, a yes or no question which is never always true or always false for all programs, is in general undecidable; or in other words, it cannot be answered algorithmically. Even a question as seemingly innocuous as “does this program return zero” falls into this category. A static analyzer, when faced with such a question, therefore does not attempt to produce a definite yes or no, instead it answers with either a definite yes or I don’t know [Min04]; and producing a meaningful yes in an efficient manner poses a challenging task to static analyzers. Additionally, they often rely heavily on formal techniques to perform well. Typical techniques employed include symbolic execution, model checking [Kro+03], satisfiability modulo theories [DMB08], data-flow analysis based on lattices [Nie+99], abstract interpretation [CC77], etc.
There are static analyzers specifically tailored to prove the absence of run-time errors in computing systems. Astrée [Ast], based on the theory of abstract interpretation, has been successful in proving the safety of the flight control systems of the Airbus A340 and A380 series, and the automatic docking software of the Jules Vernes Automated Transfer Vehicle [Bou+09]. Other static analysis tools which employ abstract interpretation include MathWorks’s Polyspace Bug Finder [PBF], Fluctuat [Flu], and ECLAIR [ECL].

This section starts by introducing the data-flow analysis framework to analyze a simple program, abstract interpretation is then applied to this example, and the properties of the resulting analysis are further discussed.

### 2.3.1 Data-Flow Analysis Framework

In this section, we use the data-flow analysis (DFA) framework [Nie+99] to deduce the semantics of a program named `simple` in Figure 2.7, which consists of only one variable `x`. We assume an initial set `ι ⊆ ℝ` of values of `x`, and the property pertaining to us is whether a particular value `x_{invalid}` is unreachable. Computations are performed in real arithmetic for simplicity. By computing an `X`, the set of all reachable final values of `x`, it suffices to check `x_{invalid} \notin X`. A sensible definition for the set of values that can be reached by `x` is a subset of all real numbers `ℝ`, i.e., an element of `℘(ℝ)`, where `℘(ℝ)` denotes the power set of `ℝ`, also known as the set of all subsets of `ℝ`.

```plaintext
real simple(real x) {
    while (x > 1.0)
        x *= 0.9;
    return x;
}
```

**Figure 2.7.** A simple program example to be statically analyzed.

The first step of DFA is to translate the body of `simple` into a control/data-flow graph (CDFG), as shown in Figure 2.8 where each block consists of a single statement or conditional, and the edges in the graph model the data- and control-flows. The `tt` and
respectively highlight the control-flow branches taken when the conditional “$x < 1$” evaluates to either true or false.

![Diagram of CDFG](image)

**Figure 2.8.** The CDFG of `simple` in Figure 2.7.

The individual blocks in the CDFG can therefore be defined as functions $f : \wp(\mathbb{R}) \to \wp(\mathbb{R})$, where both its input and output are elements of $\wp(\mathbb{R})$. For instance, for the statement “$x *= 0.9;$” a function $f_1$ can be defined as follows:

$$f_1(S) = \{0.9v \mid v \in S\}. \quad (2.9)$$

Here, the definition of $f_1$ indicates that for all possible input values $v$ of $x$ in the set $S$, we multiply it by 0.9 and collect the multiplied results into a new set as the output of $f_1$.

Similarly, because “$x > 1$” has two conditional branches, two functions, $f_{2,tt}$ and $f_{2,ff}$, respectively for both true- and false-branches of it can be defined:

$$f_{2,tt}(S) = S \cap \{v \in \mathbb{R} \mid v > 1\},$$
$$f_{2,ff}(S) = S \cap \{v \in \mathbb{R} \mid v \leq 1\}. \quad (2.10)$$

where $X \cap Y$ computes the intersection of the two sets $X$ and $Y$.

In the next step, the edges of the CDFG are labelled with numbers 0, 1, 2 and 3 to signify different locations of the program. For each edge labelled $i$, it is now possible to compute an $A(i)$, a set of values that could be reached by $x$ in a program execution at each location.
by wiring up the functions $f_1$, $f_{2,tt}$ and $f_{2,ff}$ that correspond to program statements. This gives rise to the following system of data-flow equations:

\[
\begin{align*}
A(0) &= \iota, & \text{(2.11)} \\
A(1) &= f_{2,tt}(A(0) \cup A(2)), & \text{(2.12)} \\
A(2) &= f_1(A(1)), & \text{(2.13)} \\
A(3) &= f_{2,ff}(A(0) \cup A(2)), & \text{(2.14)}
\end{align*}
\]

where $A(0) \cup A(1)$ is the union of $A(0)$ and $A(1)$.

Unfortunately, computationally solving this system of equations is not an easy task. In the rest of this section, the two significant impediments are explained, and subsequently, theories are introduced to address them. Section 2.3.2 discusses how the system of data-flow equations can be solved analytically for the most desired solution, using a minimal set of reachable values for $A(1)$ as an example. However, this analytical solution cannot be computed by a machine. By building on top of this foundation, Section 2.3.3 therefore provides a practical solution to compute a safe approximation of this set by an algorithm.

### 2.3.2 Least Fixpoint Solution to a Data-Flow Analysis Problem

There are multiple solutions to this system. For example, we can solve it manually by substituting $A(0)$ and $A(2)$ in (2.12) with (2.11) and (2.13). We arrive at:

\[
A(1) = (\iota \cup \{0.9v \mid v \in A(1)\}) \cap \{v \in \mathbb{R} \mid v > 1\}. & \text{(2.15)}
\]

It turns out that the set of all real numbers greater than 1:

\[
A(1) = \{v \in \mathbb{R} \mid v > 1\}, & \text{(2.16)}
\]
is a solution to (2.15). Substituting $A(1)$ in the right-hand side of (2.15) with this value proves that it is indeed the solution for this equation, assuming all sets below are subsets of $\mathbb{R}$ to simplify the derivation:

$$A(1) = \left( \iota \cup \left\{ 0.9v \mid v \in \{ v' \mid v' > 1 \} \right\} \right) \cap \{ v \mid v > 1 \}$$

$$= \left( \iota \cup \left\{ 0.9v \mid v > 1 \right\} \right) \cap \{ v \mid v > 1 \} = \left( \iota \cup \left\{ v \mid v > 0.9 \right\} \right) \cap \{ v \mid v > 1 \}$$

$$= \left( \iota \cap \{ v \mid v > 1 \} \right) \cup \left( \left\{ v \mid v > 0.9 \right\} \cap \{ v \mid v > 1 \} \right)$$

$$= \left( \iota \cap \{ v \mid v > 1 \} \right) \cup \{ v \mid v > 1 \} = \{ v \mid v > 1 \}. \quad (2.17)$$

Intuitively, a manual inspection of simple finds that $x$ can reach values $v, 0.9v, 0.9^2v,$ and so on, such that all values in this sequence are greater than 1, for each $v \in \iota$; or more succinctly, an alternative solution to $A(1)$ should be:

$$A(1) = \{ v' \mid v' > 1 \land v' = 0.9^k v \land v \in \iota \land k \in \mathbb{N} \}, \quad (2.18)$$

where $k \in \mathbb{N}$ denotes $k$ is one of 0, 1, 2, ..., i.e. a natural number.

It is evident to us the latter solution (2.18) is more precise, hence more desirable, than the former (2.16). Not only does it contain information the former has, i.e. all values reachable by $A(1)$ are greater than 1, it also expresses the fact that it only consists of values of the form $0.9^k v$ that are greater than 1, where $v \in \iota$ and $k \in \mathbb{N}$. A useful definition of preciseness is therefore the subset relation “$\subseteq$”. If it is known that $X \subseteq X'$, and $X$ and $X'$ are both solution to a system of data-flow equations, then $X$ is clearly more appealing than $X'$.

The set $\wp(\mathbb{R})$, with a preciseness ordering “$\subseteq$”, is a partially ordered set. It has three following properties for any $X, Y, Z \in \wp(\mathbb{R})$: it is reflexive, $X \subseteq X$; it has the antisymmetry property, i.e. if $X \subseteq Y$ and $Y \subseteq X$, then $X = Y$; and finally it is transitive, if $X \subseteq Y$ and $Y \subseteq Z$, then $X \subseteq Y$. In contrast to a total order such as the set of reals $\mathbb{R}$, not every two elements in $\wp(\mathbb{R})$ can be compared, e.g. neither of the sets $\{1, 2, 3\}$ and $\{2, 3, 4\}$ is a subset of one another.
For the purpose of computing the solution to $A(1)$’s equation (2.15), a function $f : \wp(\mathbb{R}) \to \wp(\mathbb{R})$ can be defined:

$$f(X) = (\iota \cup \{0.9v \mid v \in X\}) \cap \{v \in \mathbb{R} \mid v > 1\}, \quad (2.19)$$

so that all solutions of the original equation (2.15) are now in this following set, which are known as the fixpoints* of $f$:

$$\text{Fix}(f) = \{X \in \wp(\mathbb{R}) \mid f(X) = X\}. \quad (2.20)$$

By using this particular definition of preciseness, it is clear that we wish to find the fixpoint of $f$ with the smallest number of elements in it. With this in mind, two important questions arise:

1. Is the most precise solution unique? A unique most precise solution is defined as the only one which is the most precise among all possible solutions to the systems of data-flow equations. In other words, if it exists, then it is defined as the least fixpoint (least fixpoint (LFP)) of $f$ which is a subset of all other fixpoints, i.e. $\text{lfp}(f) \subseteq Y$ for any $Y \in \text{Fix}(f)$. As we have discussed earlier, multiple fixpoints exist, and it is possible that these fixpoint solutions are not comparable.

2. If a unique solution exists and it is unique, how do we find it? This is equivalent to finding a way to compute the LFP $\text{lfp}(f)$ using $f$.

As it turns out, to answer the first question, Tarski’s fixpoint theorem [Tar55; Nie+99] can be used to prove that $\text{lfp}(f)$ is indeed unique.

*Another common name for fixpoint is fixed point. To avoid being mistaken for the fixed point representation, a binary number representation, the term fixpoint is used instead.
For the second question, Kleene’s fixpoint theorem shows that in our example analysis, the most precise solution of $A(1)$ can be computed using:

\[
\text{lfp}(f) = \bigcup_{k \in \mathbb{N}} f^k(\varnothing). \tag{2.21}
\]

Here, $\varnothing$ is the empty set, and a function of the form $h^n(x)$, where $h : M \to M$ for any domain $M$ and $n \in \mathbb{N}$, is recursively defined as:

\[
h^n(x) = \begin{cases} 
    h(h^{n-1}(x)) & \text{if } n > 0, \\
    x & \text{if } n = 0.
\end{cases} \tag{2.22}
\]

The functions $f^k(\varnothing)$ for the first $k + 1$ iterations can be evaluated as follows:

\[
\begin{align*}
    f^0(\varnothing) &= \varnothing, \\
    f^1(\varnothing) &= \iota \cap \{v \mid v > 1\}, \\
    f^2(\varnothing) &= f(f^1(\varnothing)) = (\iota \cup \{0.9v \mid v \in \iota\}) \cap \{v \mid v > 1\}, \\
    f^3(\varnothing) &= (\iota \cup \{0.9v \mid v \in \iota\} \cup \{0.9^2v \mid v \in \iota\}) \cap \{v \mid v > 1\}, \ldots, \\
    f^k(\varnothing) &= (\iota \cup \{0.9v \mid v \in \iota\} \cup \ldots \cup \{0.9^{k-1}v \mid v \in \iota\}) \cap \{v \mid v > 1\}.
\end{align*} \tag{2.23}
\]

Finally, the most precise solution to (2.15) can be computed using the LFP formula for $f$, which is exactly the same as the alternative solution that was manually computed in (2.18):

\[
\text{lfp}(f) = \bigcup_{k \in \mathbb{N}} f^k(\varnothing) = \{v \mid v > 1\} \cap \bigcup_{k \in \mathbb{N}} \{0.9^k v \mid v \in \iota\} \tag{2.24}
\]

\[
= \{v' \mid v' > 1 \land v' = 0.9^k v \land v \in \iota \land k \in \mathbb{N}\}.
\]

Even though we have derived a method to statically analyze a program, significant obstacles stymie the efficient usage of it. Firstly, in the case study of simple, because the LFP is evaluated as the union of $f^k(\varnothing)$ in a sequence, this sequence is likely to be infinite, and thus cannot be computed fully. Secondly, the set of input values, $\iota$, not only determines the number of iterations necessary in order to calculate the LFP, but also impacts the amount of computation required in each iteration. For instance if $\iota = \{4\}$ then it is only necessary to track the computation for a single input value 4, whereas when $\iota = \{v \mid 0 \leq v \leq 1000\}$, there are infinitely many values in the set. As a result, in
general, the LFP of an arbitrary self-map function \( f : \mathbb{L} \to \mathbb{L} \) is thus not computable in finite amount of time. In Section 2.3.3, a method known as abstract interpretation is introduced to overcome the computability problem.

### 2.3.3 Abstract Interpretation with Intervals

A framework of methods, known as abstract interpretation (AI), is proposed by Cousot \textit{et al.} [CC77] to formally mitigate the problem of computability in program analysis. Instead of finding the LFP, which may not be computable, it is much more efficient to work out an \textit{approximation} of the LFP. Despite the outcome of an AI-based static analysis not being as precise as the LFP, the significant benefits of AI is two-fold. Firstly, the program analysis framework can now produce a “yes” or “I don’t know” answer to a query of a program property in a finite amount of time. Secondly, it provides the means to prove the correctness of an answer produced by the static analyzer using AI in formal mathematics.

Here, a simple formulation of interval arithmetic is first introduced, which is then exercised in the AI framework, again by using the simple example in Figure 2.7.

#### Interval Arithmetic

\textit{Interval arithmetic} (IA), is a method to enclose a set of solutions that may arise in computation problems [Moo+09]. The standard method is to use a \textit{pair} of values \([a, b]\), to represent \(\{v \mid a \leq v \leq b\}\), a potentially \textit{infinite} set of real numbers between \(a\) and \(b\).

As it is closely related to sets of reals and real arithmetic, IA gets the benefits of both worlds.

Firstly, operations, similar to the set operations such as the subset relation “\(\subseteq\)”, union “\(\cup\)” and intersection “\(\cap\)” from \(\mathbb{R}\), can also be defined for real intervals \textit{Interval}. The corresponding operations are known as the partial ordering “\(\sqsubseteq\)”, \textit{join} “\(\sqcup\)” and \textit{meet} “\(\sqcap\)”, a partially ordered set with these operations defined for all elements within it is a complete
The definitions of these operators on two intervals \([a, b]\) and \([c, d]\) are as follows:

\[
[a, b] \sqsubseteq [c, d] := a \geq c \land b \leq d,
\]

\[
[a, b] \sqcup [c, d] := [\min(a, c), \max(b, d)],
\]

\[
[a, b] \sqcap [c, d] :=
\begin{cases}
\max(a, c), \min(b, d) & \text{if } \max(a, c) \leq \min(b, d), \\
\bot & \text{otherwise}.
\end{cases}
\]

(2.25)

where \(s := t\) indicates \(s\) is defined as \(t\), \(\min(x, y)\) and \(\max(x, y)\) respectively compute the minimum and maximum of \(x\) and \(y\), and \(\bot, \top\) respectively denote intervals with no elements, i.e. an empty interval, and the entire set of reals. For completeness, the above relations can be further extended for \(\top\) and \(\bot\), where \(X^\sharp \in \text{Interval}\) is an interval:

\[
X^\sharp \sqcup \top := \top, \quad \text{and} \quad X^\sharp \sqcap \bot := \bot.
\]

(2.26)

As a consequence, \(\bot\) and \(\top\) are respectively the least and greatest elements of \(\text{Interval}\). This means that \(\bot \sqsubseteq X^\sharp\) and \(X^\sharp \sqsubseteq \top\) for any interval \(X^\sharp\).

Secondly, in a similar fashion to real arithmetic, arithmetic operations can also be defined for real intervals:

\[
[a, b] + [c, d] = [a + c, b + d],
\]

\[
[a, b] - [c, d] = [a - d, b - c],
\]

\[
[a, b] \times [c, d] = [\min(s), \max(s)],
\]

where \(s = \{a \times c, a \times d, b \times c, b \times d\}\),

\[
-[a, b] = [-b, -a].
\]

(2.27)

A scalar value \(x\), which can be abbreviated by \(x\) itself, represents an interval \([x, x]\) in IA.

**An Informal Approach to Approximation**

AI is a theoretical framework to approximate mathematical objects that are not directly computable. We start by explaining what it means to *approximate*, then show how an approximation can be proved to be safe.
In the DFA of simple in Section 2.3.1, we arrived at a function $f : \wp(\mathbb{R}) \to \wp(\mathbb{R})$ defined in (2.19), which is not directly computable. The IA introduced earlier could serve as an inspiration to produce a different function $\tilde{f} : \text{Interval} \to \text{Interval}$, which is very similar to the original $f$:

$$\tilde{f}(X^\sharp) = (\iota^\sharp \sqcup (0.9X^\sharp)) \cap [1, \infty], \quad (2.28)$$

where $\iota^\sharp$ is an interval that bounds the set of initial values $\iota$. There are two noticeable differences between $f$ and $\tilde{f}$. Firstly, the domain, where $\tilde{f}$ carries out its computation, is $\text{Interval}$ instead of $\wp(\mathbb{R})$. Secondly, because the domain used is different from $f$, the function definition is therefore updated accordingly for $\tilde{f}$.

When given an input $X$, $f$ must enumerate on all $X$ to compute the result $f(X)$ precisely. As we have discussed earlier this is infeasible as $X$ could contain infinite number of values. Conversely, $\tilde{f}$ does not suffer from this problem, since (2.27) dictates any IA operations can be performed by a finite number of real arithmetic computations.

As it is possible to prove that both fixpoint theorems in Section 2.3.2 hold for $\text{Interval}$ and $\tilde{f}$, its LFP, an approximation to the LFP to $f$, can therefore be computed as follows:

$$\text{Ifp}(\tilde{f}) = \bigsqcup_{k \in \mathbb{N}} \tilde{f}^k(\bot). \quad (2.29)$$

For example, consider the case when $\iota = [0, 10]$,

$$\tilde{f}^0(\bot) = \bot,$$

$$\tilde{f}^1(\bot) = ([0, 10] \sqcup 0.9\bot) \cap [1, \infty] = [1, 10], \quad (2.30)$$

$$\tilde{f}^2(\bot) = ([0, 10] \sqcup (0.9 \times [1, 10])) \cap [1, \infty] = [1, 10], \ldots$$

As all other values in the sequence evaluates to $[1, 10]$, $\text{Ifp}(\tilde{f})$ is hence $[1, 10]$, which was computed in 3 iterations. It is easy to see in Figure 2.7 that the reachable values of $x$ before executing the statement “$x *= 0.9$;” is indeed $[1, 10]$. 

2.3 Program Analysis and Abstract Interpretation
In many cases, an algorithm to compute $\tilde{f}^k(⊥)$ can terminate. It is clear that if $\tilde{f}^k(⊥) = \tilde{f}^{k+1}(⊥)$ for some $k \in \mathbb{N}$, then for all integers $j$ that are greater than $k$, $\tilde{f}^j(⊥) = \tilde{f}^k(⊥)$ and there is no point in computing $\tilde{f}^j(⊥)$. Widening and narrowing operators [CC77; Nie+99] can be used to reduce the number of iterations required in the iterative computation steps, hence accelerating, or even ensuring, termination by sacrificing precision of the computed fixpoint, i.e. the result is no longer the LFP, but is a fixpoint $X \sqsubseteq \text{lfp}(\tilde{f})$ that can be computed more easily.

**Galois Connection**

Although our informal derivation gives us empirical evidence of the usefulness and correctness of intervals in place of real sets, a series of questions pertaining the theory behind it remain. The questions are of the format “is $X$ a correct abstraction of $Y$”, where $X$ and $Y$ refer to each of the following pairs: $(\text{Interval}, \wp(\mathbb{R}))$, $(\tilde{f}, f)$, and $(\text{lfp}(\tilde{f}), \text{lfp}(f))$.

Fortunately, Cousot et al. [CC77] show that if a Galois connection can be formed between $\wp(\mathbb{R})$ and Interval, then all of the above questions can be now answered with a resounding “yes”. We can define [Nie+99]:

**Definition 2.1.** [Galois connection] A Galois connection is a relation between two complete lattices $\langle L, \subseteq \rangle$ and $\langle M, \sqsubseteq \rangle$, given by a pair of functions $\alpha : L \to M$ and $\gamma : M \to L$, such that for all $l \in L$ and $m \in M$:

\[
\alpha(l) \sqsubseteq m \quad \text{if and only if} \quad l \subseteq \gamma(m), \quad (2.31)
\]

alternatively the following property may sometimes be easier to work with:

\[
l \subseteq \gamma(\alpha(l)) \quad \text{and} \quad \alpha(\gamma(m)) \subseteq m. \quad (2.32)
\]

The Galois connection above can often be concisely written as:

\[
\langle L, \subseteq \rangle \xrightarrow{\alpha, \gamma} \langle M, \sqsubseteq \rangle. \quad (2.33)
\]
Here, the functions $\alpha$ and $\gamma$ are often called the abstraction function and concretization function respectively.

Bearing on the informal approach earlier, the following Galois connection can be established to formalize it:

$$\langle \wp(\mathbb{R}), \subseteq \rangle \rightleftharpoons \langle \text{Interval}, \subseteq \rangle,$$

(2.34)

The functions $\alpha : \wp(\mathbb{R}) \to \text{Interval}$ and $\gamma : \text{Interval} \to \wp(\mathbb{R})$ that satisfy (2.31) can be defined as follows:

$$\alpha(X) = [\inf(X), \sup(X)],$$

$$\gamma([a, b]) = \{v \mid a \leq v \leq b\},$$

(2.35)

where $\inf(X)$ and $\sup(X)$ are respectively the infimum and supremum of $X$. Here $\alpha$ takes a potentially infinite set of reals and translate it into a pair of values representing an interval bounding the reals, whereas $\gamma$ performs the backwards translation from an interval to a set of reals. For instance:

$$\alpha(\{1, 1.2, 3\}) = [1, 3],$$

$$\gamma([1, 3]) = \{v \mid 1 \leq v \leq 3\}.$$

(2.36)

It is clear that information could be lost when the abstraction function $\alpha$ is applied. As shown in the example above, a set of three real values $1, 1.2$ and $3$ can be approximated by an interval $[1, 3]$, which represents a set of real values ranging from $1$ to $3$.

Furthermore, from a function $g : L \to L$, the AI framework allows an approximated function $g^\sharp : M \to M$ to be inductively abstracted by the Galois connection [Nie+99]. For example, consider (2.19) which is not computable, an approximated candidate $f^\sharp$ can be computed by $\alpha \circ f \circ \gamma$. Since we are computing in $\text{Interval}$, we assume that $\iota = \gamma(\iota^\sharp)$ and use $a^\sharp$ and $\overline{a}^\sharp$ respectively denote the lower- and upper-bounds of an interval $a^\sharp$, i.e.

$$a^\sharp = [a^\sharp, \overline{a}^\sharp]:$$

$$f^\sharp(X^\sharp) = \alpha\left(f\left(\gamma\left(X^\sharp\right)\right)\right) = \alpha\left(f\left(\{v \mid X^\sharp \leq v \leq \overline{X}^\sharp\}\right)\right)
= \alpha\left(\iota \cup \{0.9v \mid X^\sharp \leq v \leq \overline{X}^\sharp\} \cap \{v \mid v > 1\}\right)
= \alpha\left(\iota \cap \{v \mid v > 1\}\right) \cup \left(\{v \mid 0.9X^\sharp \leq v \leq 0.9\overline{X}^\sharp\} \cap \{v \mid v > 1\}\right).$$

(2.37)
We introduce $\epsilon$ to denote an infinitesimal positive value. Because $\alpha(A \cup B) = \alpha(A) \sqcup \alpha(B)$:

$$f^\sharp(X^\sharp) = \alpha \left( \left\{ v \mid v^\sharp \leq v \leq \overline{v}^\sharp \right\} \cap \left\{ v \mid v > 1 \right\} \right) \sqcup \alpha \left( \left\{ v \mid \min(0.9X^\sharp, 1 + \epsilon) \leq v \leq 0.9X^\sharp \right\} \right),$$

which is identical to $\tilde{f}$ defined in (2.28), therefore the correctness of $\tilde{f}$ is analytically proven. Although we derived a special case using the approximate function induction technique, in a more general fashion, this method can be applied to the functions and operators in the DFA of reachable sets of reals. Consequently, the DFA of a general program based on intervals can be induced.

**Further Generalization**

For simplicity, $\text{simple}$ is used as an example of DFA. However unlike $\text{simple}$, general programs can consist of more than one variable. A DFA for a program should therefore compute a reachable set of values for each variable. A typical way to do so is to use a mapping which associates each program variable with a value, which is defined as an element $\sigma \in \Sigma$, where $\Sigma = \{	ext{Var} \rightarrow \mathbb{L}\}$ and $\mathbb{L}$ is a set of values. For example, a single program state of $\text{simple}$ could be $\sigma_0 \in \{	ext{Var} \rightarrow \mathbb{R}\}$, and $\sigma_0 = [x \mapsto 0]$, which denotes that the state $\sigma_0$ has only a variable $x$, and $x$ is assigned a value 0. We could also have a state that captures multiple program states, e.g. $\sigma^\sharp \in \{	ext{Var} \rightarrow \text{Interval}\}$, where $\sigma^\sharp(x)$ could be an interval.

Galois connections can be compositionally constructed [Nie+99]. If we know that $\wp(\mathbb{R}) \xrightarrow{\gamma} \text{Interval}$, then the following Galois connection between mappings can also be constructed:

$$[\text{Var} \rightarrow \wp(\mathbb{R})] \xrightarrow{\gamma'} [\text{Var} \rightarrow \text{Interval}],$$

(2.40)
where $\alpha'(f) = \alpha \circ f$, $\gamma'(g) = \gamma \circ g$, and a term of the form $s \circ t$, where $s : B \to C$ and $t : A \to B$ denotes a function which accepts an input from $A$ and produces an intermediate output in $B$ using $t$, then in turn $s$ is used on the intermediate value to compute a final output in $C$.

### 2.3.4 Abstract Domains

The interval domain offers us an efficient way to enclose reachable values of variables. However, it is unable to capture the correlation among these variables. For instance, if we know that $x$ and $y$ are reals between $0$ and $1$, and $x \leq y$, intervals cannot express the relation $x \leq y$ and using bounds $x \in [0, 1]$ and $y \in [0, 1]$, we evaluate the expression $(1 - y)x$ is bounded by $0$ and $1$. In contrast, if we were able to make use of the inequality $x \leq y$, then it is possible to deduce $(1 - y)x \leq (1 - x)x \leq \frac{1}{2}$, which yields a much tighter bound than using intervals alone.

In general, the design of abstract domains is a trade-off between how good a approximation it can obtain, and how efficiently it can be computed. For example, we could imagine another abstract domain, $\text{Sign}$, which only captures the signedness of variables, to also enclose a set of reachable values. Although it is even faster than intervals to compute, it sacrifices the precision of the value bounds. Ranging from the fastest to the most expensive, a hierarchy of abstract domains for reals and floating-point computations are proposed by various authors as discussed below.

A new abstract domain, $\text{Octagon}$, is proposed by Miné [Min07a], to enclose values in a system of difference-bound inequalities, which are of the form:

\[
x - y \leq c, \quad \text{or} \quad \pm x \leq c,
\]

where $x$ and $y$ are variables and $c$ is a constant. This domain is very efficient as a Floyd-Warshall algorithm [Flo62], which runs in $\mathcal{O}(n^3)$, where $n$ is the number of variables, can be used to compute it [Min04]. Although not as efficient as intervals, it is much more expressive than intervals. This is because intervals are equivalent to a set of inequalities
of the latter form, $\pm x \leq c$, but they cannot represent the former which captures the correlations between variables, $x - y \leq c$.

Based on affine arithmetic (AA), Ghorbal et al. [Gho+09] propose an abstract domain, Taylor1+. This domain uses an equation of the form to capture the bound on a variable $x$:

$$\tilde{x} = \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \varepsilon_i,$$

where $\alpha_i^x \in \mathbb{R}$ for each $i$, and each $\varepsilon_i$ known as the noise symbol, which has an unknown quantity that lies in $[-1, 1]$, introduces a perturbation weighted by $\alpha_i^x$ on the constant $\alpha_0^x$. If two variables $x$ and $y$ are correlated, $\tilde{x}$ and $\tilde{y}$ can consist of the same $\varepsilon_i$ to encode the correlation. They found that for a 2nd order infinite impulse response (IIR) filter, Taylor1+ can give an exact bound on the filter output, whereas both Interval and Octagon failed as the bound size increases exponentially in each iteration, albeit slower than both methods.

Cousot et al. [CH78] introduce the polyhedra approach to abstract domains. This method makes use of arbitrary polyhedra, which could be expressed by a set of linear inequalities such as the following:

$$\begin{cases} 2x + 7y \leq 32, \\ 8x - 3y + z \geq 0. \end{cases}$$

Libraries that implement the polyhedra domain include APRON [Apr] and Parma Polyhedra Library [PPL]. Although the polyhedra domain is very accurate for linear computations, Ghorbal et al. [Gho+09] find both libraries to be much slower than their AA based approach.

In addition to abstract domains on reals and floating-point values, integer-based domains are further presented by several authors. Granger [Gra89] introduce the simple congruences on integers, which consists of equations of the form $x = a \mod b$, where $x$ is a variable and $a$ and $b$ are integers. They further propose a linear congruences formulation which is more expressive, for instance, an equation $3x + 4y = 5 \mod 6$ can be used to capture the relation between integer variables $x$ and $y$. 
Round-off Error Analysis

In Section 2.5, we will discuss techniques which optimize numerical accuracies in a program by restructuring it. Their methods utilize a common analysis of floating-point round-off errors, which is based on a formulation of AI. In this section, this numerical accuracy analysis approach is thus explained in detail.

Because of the finite characteristic of IEEE 754 floating-point format [ANS08], it is not always possible to represent exact values with it. Computations in floating-point arithmetic often induce round-off errors. Therefore, Martel [Mar07] bound the ranges of the values in floating-point calculations, as well as the round-off errors introduced in these computations. This accuracy analysis determines the bounds of all possible outputs and their associated range of round-off errors for expressions.

Martel [Mar07] introduces an abstract error semantics for the calculation of round-off errors in the evaluation of floating-point expressions. It consists of three components: an abstract domain to enclose computed floating-point values and round-off errors, a suite of partial order operations on the abstract domain, and finally, a set of arithmetic operations to evaluate arithmetic expressions using the abstract domain.

First we define the domain $E^\sharp = \text{Interval}_F \times \text{Interval}$, where $\text{Interval}$ and $\text{Interval}_F$ respectively represent the set of real intervals, and the set of floating-point intervals (intervals that each enclose a range of floating-point values). The value $(x^\sharp, \mu^\sharp) \in E^\sharp$ represents a safe bound on floating-point values and the accumulated error represented as a range of real values.

Secondly, for the Cartesian product of partial orders, a partial order relation inherently arises by allowing element-wise partial order operations [AJ94]. In other words, the
following operations for the partial order relations can be induced for \( E^\# \), by using the respective definitions of the operators in \( \text{Interval}_F \) and \( \text{Interval} \):

\[
\begin{align*}
(x_1^\#, \mu_1^\#) \subseteq (x_2^\#, \mu_2^\#) &:= x_1^\# \subseteq x_2^\# \land \mu_1^\# \subseteq \mu_2^\#, \\
(x_1^\#, \mu_1^\#) \cup (x_2^\#, \mu_2^\#) &:= (x_1^\# \sqcup x_2^\#, \mu_1^\# \sqcup \mu_2^\#), \\
(x_1^\#, \mu_1^\#) \cap (x_2^\#, \mu_2^\#) &:= (x_1^\# \sqcap x_2^\#, \mu_1^\# \sqcap \mu_2^\#),
\end{align*}
\]

for \((x_1^\#, \mu_1^\#) \in E^\#\) and \((x_2^\#, \mu_2^\#) \in E^\#\).

Furthermore, the error domain forms the following Galois connection:

\[
\varphi (\mathbb{F} \times \mathbb{R}) \iff \text{Interval}_F \times \text{Interval},
\]

Finally, arithmetic operations can also be defined for the values in \( E^\# \). To start we introduce the functions \( \uparrow^\# \) and \( \downarrow^\# \), which respectively compute the interval of rounded floating-point results and the range of round-off error from arithmetic operations under a given rounding mode. The function \( \uparrow^\# : \text{Interval} \to \text{Interval}_F \) computes the floating-point bound from a real bound, by rounding the infimum \( a \) and supremum \( b \) of the input interval \([a, b]\):

\[
\uparrow^\# ([a, b]) := [\uparrow_o (a), \uparrow_o (b)]_F.
\]

where the subscript \( F \) indicates the interval is a floating-point interval, and \( \uparrow_o (v) \) indicates rounding a value \( v \) to a floating-point value. The function \( \downarrow^\# : \text{Interval} \to \text{Interval} \) determines the range of round-off error due to the floating-point computation:

\[
\downarrow^\# ([a, b]) := \left[ -\frac{z}{2}, \frac{z}{2} \right], \text{ where } z = \max(ulp(a), ulp(b)).
\]
We can now define the arithmetic operations on values in $E^\#$. For addition, subtraction and unary subtraction, we have:

\[
\begin{align*}
(x_1, \mu_1) + (x_2, \mu_2) := & \ \left[ \uparrow_0 (x_1 + x_2), \downarrow_0 (x_1 + x_2) \right], \\
(x_1, \mu_1) - (x_2, \mu_2) := & \ \left[ \uparrow_0 (x_1 - x_2), \downarrow_0 (x_1 - x_2) \right], \quad (2.48) \\
- (x_1, \mu_1) := & \ (-x_1, -\mu_1).
\end{align*}
\]

Multiplication on $E^\#$ can be formulated, by expanding $(x_1 + \mu_1)^\#(x_2 + \mu_2)^\#$ and divide the terms into the multiplied value $x_1^\# x_2^\#$ and error terms:

\[
\begin{align*}
(x_1^\#, \mu_1^\#) \times (x_2^\#, \mu_2^\#) := & \ \left[ \uparrow_0 (x_1^\# x_2^\#), \downarrow_0 (x_1^\# x_2^\#) \right], \\
& \ \left[ x_1^\# \times \mu_2 + x_2^\# \times \mu_1 + \mu_1^\# \times \mu_2 + \downarrow_0 (x_1^\# x_2^\#) \right]. \quad (2.49)
\end{align*}
\]

The addition, subtraction and multiplication of intervals follow the standard rules of IA defined earlier in (2.27).

Expressions can be evaluated for their accuracy by the method as follows. Initially, for real-valued variables, the following function can be used to convert an interval of real values into a value in the error domain:

\[
\text{cast } (x^\#) := \left( \uparrow_0 (x^\#), \downarrow_0 (x^\#) \right). \quad (2.50)
\]

For example, for the real variable $a \in [0.2, 0.3]$ under single precision with rounding to nearest:

\[
\text{cast } ([0.2, 0.3]) = ([0.200000003, 0.300000012], [-1/67108864, 1/67108864]). \quad (2.51)
\]

After this, similar to the way we use intervals to analyze a program in Section 2.3.3, floating-point arithmetic expressions can be analyzed. For instance, the expression $(a + b)^2$ can be evaluated just as we would expect in real arithmetic, instead, the arithmetic operators are overridden to operate on values in the error domain $E^\#$. 

2.3 Program Analysis and Abstract Interpretation
For example, assume that real variables \( a \in [0.2, 0.3], \ b \in [2.3, 2.4] \), it is possible to derive that in single-precision floating-point computation with rounding to the nearest, \((a + b)^2 \in [6.24999857, 7.29000187] \) and the error caused by this computation is bounded by \([-1.60634534 \times 10^{-6}, 1.60634534 \times 10^{-6}] \).

### 2.4 Intermediate Representations

IRs are data structures designed to be independent of the machine architecture and source language. They are often invented with the intention to ease program analysis and optimization in mind, by abstracting information from the original program that are irrelevant to our objectives. In this section, we introduce several categories of existing IRs, and delve deeper into the advantages and disadvantages of each.

#### 2.4.1 Static Single Assignment Form and Control-Flow Graph

Traditionally, static single assignment form [Alp+88; Rau92] together with the control-flow graph are used to represent data- and control-flow of a program [Cyt+91], because they are more favorable program representations on which optimization passes can be implemented, when compared to the original HLL or the output language. SSA can be advantageous in implementing conventional optimization techniques, e.g. code motion [Cyt+86], removing redundant computations [Ros+88], and constant propagation [Cyt+91]. Because the LLIR [LLIR] is based on SSA and CFGs, and is commonly used in many HLS tools such as LegUp [LU], we introduce SSA and CFGs by compiling the dot-product example in Figure 2.4 into LLIR as shown in Figure 2.9.

The LLIR of our example function consists of parts that are known as basic blocks (BBs). Each BB in turn often contains a label that uniquely identifies the BB, a list of LLIR statements in SSA form without any branches, *i.e.* the statements are executed sequentially, and a terminator instruction, which is typically a branch instruction that leads the control-flow to a different BB, by referencing a BB label or a function return.
```c
define float @dotprod(
    float* nocapture readonly %A,
    float* nocapture readonly %B) #0
{
    ; <label>:0
    br label %2

    ; <label>:1 ; preds = %2
    ret float %8

    ; <label>:2 ; preds = %2, %0
    %i.02 = phi i32 [ 0, %0 ], [ %9, %2 ]
    %d.01 = phi float [ 0.000000e+00, %0 ], [ %8, %2 ]
    %3 = getelementptr inbounds float, float* %A, i32 %i.02
    %4 = load float, float* %3, align 4, !tbaa !2
    %5 = getelementptr inbounds float, float* %B, i32 %i.02
    %6 = load float, float* %5, align 4, !tbaa !2
    %7 = fmul float %4, %6
    %8 = fadd float %d.01, %7
    %9 = add nuw nsw i32 %i.02, 1
    %exitcond = icmp eq i32 %9, 1024
    br i1 %exitcond, label %1, label %2
}
```

**Figure 2.9.** The compiled and optimized LLIR output from the dot-product example in Figure 2.4.

The LLVM framework implicitly constructs a CFG from the IR code, which is a directed graph representing the control-flow of a program. The vertices in the CFG constitute BBs, while the edges indicate the control-flow directions (i.e. branches to other BBs), often with predicate attributes to determine whether the branch is taken. For instance, we consider the first line of the third BB in Figure 2.9:

```
; <label>:2 ; preds = %2, %0
```

which indicates it has a label value 2 and the control-flow coming to this BB is from either BB2 or BB0, here we use BBn as a shorthand denoting a BB labelled n. Additionally, this BB ends with the branch terminator instruction:

```
br il %exitcond, label %1, label %2
```

This instruction directs the control-flow to BB1 or BB2, and the variable %exitcond in the terminator instruction decides which branch is taken. Finally, the complete CFG is
shown in Figure 2.10. It is noteworthy that BB2 has two edges that leads to either BB1 or BB2 itself. If $\%exit\text{cond}$ evaluates to false (ff), then another iteration of BB2 will commence, otherwise (tt) the exit condition is satisfied and will lead the control-flow to BB1.

![CFG Diagram](image)

**Figure 2.10.** The CFG of the LLIR code in Figure 2.9.

Each BB contains sequential computations that are represented by SSA instructions. The SSA form describes the operations in the original program, such that each variable in it is assigned exactly one value.

The sequence of instructions that assigns to $\%3$-$\%9$ in Figure 2.9 carries out most of the computations in the program. It starts by reading $A[i]$ and $B[i]$, multiplying them together, then adding the result with $d$ to form a new variable, and finally, the iteration value is incremented by 1. It may seem unusual that the accumulated sum of products and the iteration value are not assigned to $d$ and $i$ respectively. We can imagine two BBs, one initializes $d$ and $i$ to zeros, the other updates these two variables in a loop. As all variables in an SSA form must be assigned once only, one of the BBs should use different names for these two variables. When the control-flows of these two BBs join, we must introduce a way to read from the variables that are assigned in the two BBs in the succeeding BB. A new instruction, called the $\phi$-function is therefore defined for our purpose. The $\phi$-function accepts two variable names as its inputs, and produces the value
of either variable as its output, determined by which preceding BB the control-flow came from. For example, in LLIR, the instruction:

```ml
%d.01 = phi float [ 0.000000e+00, %0 ], [ %8, %2 ]
```

shows that if the control-flow originated from BB0, then a constant zero is returned, otherwise the control-flow had to come from BB2 and the value of %8 is used instead.

The rationale of SSA is that we can abstract away anti- and output dependences by never assigning to the same variable twice, while only true data-flow dependences remain. An anti-dependence is a dependence relation when a read operation must precede a write to the same variable, and an output dependence is when two writes refer to the same location. By removing these dependences and deferring the analysis of them, certain program optimization analyses can run much more efficiently. Analyses that may benefit from this include scheduling [Rau94], liveness analysis (estimating the lifetime of variables to reduce register requirements) [Cyt+91], detecting opportunities for parallelism [CF87], and finding equivalent parts in the program [Alp+88].

In a cyclic CFG, the control-flow could potentially revisit a BB, and instructions in this BB will inevitably assign a different value to the same variable, which forms anti- and output dependences, which could have a detrimental effect on efficient loop pipelining in some computing machines. An alternative IR, the dynamic single assignment (DSA) form [Rau92] can therefore be used in place of the SSA to address this issue. The DSA defines a linear sequence of virtual registers for each variable, such that every time the variable is assigned in a dynamic execution path, a new virtual register is used.

### Alternatives to SSA and CFG

There are a number of alternative IRs that are similar in construction to the SSA and CFG approach in LLIR. For instance, the data-dependence graph [Rau94] introduced in Section 2.2.3 is designed for the purpose of capturing data-flow dependences in polyhedral methods. Data-flow graph (DFG) is a popular alternative to SSA, which is often a directed acyclic graph (DAG). In general, a DFG’s vertices are input, output and
operation nodes, and the edges capture the dependences between these nodes. A DFGs however generally does not preserve enough information for us to reconstruct a program from the graph itself. A group of data structures, known as CDFG [OG86], is commonly used to represent programs in HLS tools, e.g. SPARK [Gup+04]. A CDFG resembles a CFG such as the one used in LLIR, but in lieu of using sequential instructions in SSA form in graph vertices, each vertex contains a DFG, where no SSA temporary variables are used and data-flow dependences can by explicitly identified by edges.

2.4.2 Equality Saturation

The IRs we discussed above are all used to analyze and transform the underlying program structure, so as to produce a new representation of the optimized program. In a conventional optimizing compiler, program optimization is often carried out in a sequence of transformation passes, where each pass accepts a program, often written in a certain IR, and produces an optimized program in the IR. The traditional practice is to always apply these optimization phases in a fixed order, but a good ordering of these phases is crucial to achieve a good optimized result, and the optimal ordering varies across applications being compiled [Alm+04]. The process of finding the optimal ordering is known as the phase-ordering problem, which is in general undecidable [TB06]. Moreover, programs running on CPUs or GPUs are usually quantified by their throughput or latency, in contrast, designs on FPGAs concern us with additional objectives besides run time, such as power consumption and resource utilization that impact the quality of synthesized circuit. Multiple designs which trade-off these objectives could exist, and which design to choose relies on the specifics of the use case. It is therefore sensible to explore the design space by optimizing multiple objectives simultaneously. For the above reasons, it is desirable to have an IR and the associated optimization procedures to efficiently discover equivalent structures that lead to different implementations of the original program.

In software, a novel approach called equality saturation is proposed in [Tat+] to find multiple possible optimized variants of the original program, and subsequently deal with the phase-ordering problem. It creates a new graph-based IR, program expression graph (PEG), to encode the effects of executing the program.
To begin, we review the structure of the PEG, by considering a simple loop example in Figure 2.11. By understanding how the PEG can be evaluated for the output values, we can interpret how the PEG captures the control- and data-flow information of the program. PEGs, similar to arithmetic expressions expressed in a tree structure, can be evaluated in a bottom-up fashion, by recursively propagating computed values from the leaf nodes to the root of the tree. However, unlike arithmetic expressions which are acyclic, edges in PEGs may form cycles to express loops in the original program.

```c
int x = 1;
int y = 1;
while (y <= 10) {
    x = y * x;
    y = y + 1;
}
```

(a) The original program.

Figure 2.11. A simple loop which computes the factorial of 10, and the resulting PEG. This example and its PEG, showing computations that lead to the final x and y, is taken from [Tat+].

Data-Flow Nodes

All loops in the PEG are formed by \( \theta \) nodes, which is used in the following form:

\[
\begin{align*}
\theta & \leftarrow i \leftarrow \text{func}(i)
\end{align*}
\]

where it accepts two child graphs, \( i \) and \( \text{func} \), and \( \text{func} \) further takes the \( \theta \) node as one of its inputs to form a complete cycle. Evaluating a \( \theta \) node produces a list of values computed iteratively by the node’s subgraph. The first value in the list, is the computed result of \( i \), which we name \( i \), and values in the rest of the sequence are iteratively computed by \( \text{func} \). In functional programming, this is similar to iteratively computing the fixpoint \( \text{fix} \) of an initial empty list \([[]]\), which is defined as:

\[
\text{fix} F([[\ ]]) = \lim_{n \to \infty} F^n([[]]), \quad \text{where } F(v) = \text{prepend}(i, \text{map(func}, v)) .
\]
Here, \( \text{map}(\text{func}, v) \) applies the subgraph computation \( \text{func} \) to all elements in the list \( v \), and \( \text{prepend}(i, v') \) prepends the element \( i \) to the list \( v' \).

For example, the following subgraph extracted from Figure 2.11b evaluates to the sequence \([1, 2, 3, 4, \ldots]\):

\[
\theta_1 \\
1 \\
+ \\
1
\]

It is noteworthy that \( \theta \) nodes may have subscripts. For instance, in Figure 2.11b, both nodes \( \theta_1 \) share the same subscript 1. This is used to indicate that the two sequences produced by both \( \theta \) nodes iterate simultaneously, i.e. they share the same iteration count so that a new value for \( x \) can be computed as we update \( y \). Therefore, the \( \theta_1 \) node in the left of this figure produces a sequence of the factorials of \([1, 2, 3, \ldots]\), i.e. \([1, 2, 6, 24, \ldots]\).

Computation nodes, such as arithmetic + and Boolean operators \( \leq \) and \( \neg \) in Figure 2.11b, operates on a list of values, by performing the computation on each value in the list.

For instance, the \( \leq \) node accepts two inputs, the sequence \([1, 2, 3, \ldots]\) derived earlier, and a scalar 10, computes the result of \( x \leq 10 \) for each element \( x \) in the sequence, and finally produces the list, where \( \text{tt} \) and \( \text{ff} \) respectively denote true and false Boolean values:

\[
[\text{tt}, \text{tt}, \text{tt}, \text{tt}, \text{tt}, \text{tt}, \text{tt}, \text{ff}, \text{ff}, \text{ff}, \ldots]. \quad (2.53)
\]

The subsequent \( \neg \) node then negates all elements in the list:

\[
[\text{ff}, \text{ff}, \text{ff}, \text{ff}, \text{ff}, \text{ff}, \text{ff}, \text{tt}, \text{tt}, \text{tt}, \ldots]. \quad (2.54)
\]

**Control-Flow Nodes**

The \( \theta \) node encodes an infinite sequence of computed values, whereas the output value of the program is a scalar. By further representing control-flow information in a PEG, it
becomes possible to refer to a single value in this sequence, as the output of the program. To do so, Tate et al. [Tat+] further introduce pass and eval nodes. The pass node finds the first true (tt) value in a sequence of Boolean values, and returns the index of this value, and eval takes two child nodes, where the first node evaluates to a list $v$ of values, and the second is an scalar $n$ used to select a scalar value from $l$, as the output of this node.

To illustrate, the $\text{pass}_1$ node finds the first $\text{tt}$ in the list $\{2.54, 11\}$. The $\text{eval}$ node of the output variable $y$ subsequently fetches the 11-th item from the list $[1, 2, 3, 4, \ldots]$ we produced earlier, which is 11. Similarly we can apply the same process to find that the output $x$ is $10!$, i.e. the factorial of 10.

By using pass and eval nodes to represent the control-flow in an algebraic fashion, and mixing data- and control-flows together in the graphical representation, PEGs provide us with greater opportunities to optimize data-flow across control-flow boundaries and vice versa. Simple equivalence rules can be defined for these nodes algebraically. For instance, arithmetic operators can be distributed over $\theta$ and eval, e.g. $\text{eval}(j, k) + i \equiv \text{eval}(j + i, k)$. Complex transformations can therefore be deductively constructed from these simple equivalence rules.

**Equivalence Finding**

By applying transformation passes to the PEG, their approach detects incremental modifications, and appends these changes to the original PEG. The new changes, represented as extra structures in the PEG, are linked to their corresponding equivalent nodes by equivalence edges. These edges indicate a pair of subgraphs are equivalent, forming a program expression graph with equivalent structures (E-PEG), that could capture multiple PEGs in the same structure. The resulting E-PEG is similar to the one in Figure 2.12, where dashed edges indicate equivalences. It is notable that each edge allows a binary choice, therefore the number of PEGs that can be represented in an E-PEG could be exponential in the number of equivalence edges.
Figure 2.12. An simple E-PEG example, taken from [Tat+].

By repeatedly applying transformation rules to append all possible equivalent structures to the E-PEG, this graph will eventually saturate, i.e. no more equivalent structure can be added to the graph because all possible equivalences are now discovered. This process and the resulting E-PEG is more space-time efficient than enumerating all possible PEGs along the path, because E-PEG encourages sharing common subgraphs, even across equivalent edges. This saturated graph can always be produced regardless of in what order we apply the passes, hence preventing the phase-ordering problem. Furthermore, E-PEG defers the decision of whether an optimization should be committed until we have reached full saturation, allowing the global optima to be discovered. In contrast, because each optimization pass in a conventional compiler is performed consecutively, the compilers must make the decision to commit changes immediately after applying the optimization, which consequently often results in local optima.

2.5 Discovering Equivalent Programs

In this section, we explain existing optimization methods to restructure numerical programs with arithmetic equivalences. Because general numerical programs—consisting of program statements, conditional branches and loops—generalize arithmetic expressions, we start by introducing optimization methods for expressions, followed by those for general numerical programs.
2.5.1 Improving Performance by Rewriting Arithmetic Expressions

**Software Compilers**

It is common knowledge to software programmers that a typical optimizing compiler, such as *GNU Compiler Collection* (GCC) [St16] or Clang [Cla16], has some traditional static analysis-based optimization passes such as dead code elimination, loop strength reduction and constant propagation. These optimization passes, however, limit themselves by producing implementations that do not impact numerical accuracy, *i.e.* they compute the same output given identical inputs.

A less well-known fact about these compilers is that they also support a variety of optimization passes that are not enabled by default. These options, when enabled, allow the compiler to yield faster software implementations for programs with a large proportion of floating-point arithmetic computations. These optimization passes rewrite arithmetic expressions into more efficient alternative forms which are equivalent to the originals in real arithmetic, but when executed on a machine they compute different results. The reason for the differences is that arithmetic in machines has finite precision, computed results must be rounded to the nearest representable values. These discrepancies, when accumulated, could potentially result in wildly inaccurate outputs.

There are a number of compiler options in GCC [St16] that specifically perform the above optimizations. For example, the following options exist to enable expression-rewriting heuristics:

- `-fassociative-math` enables arithmetic expression rewriting by associativity, one of the heuristics applied is to perform exponentiation by squaring, *e.g.* an expression $x \times x \times x \times x$ which requires 3 multiplications, can be optimized as $(x \times x) \times (x \times x)$, reducing the number of multiplications to 2 by sharing the value of the subexpression $x \times x$;
• `-freciprocal-math` can be used to rewrite \( x/y \) into \( x \times (1/y) \), if \( 1/y \) can be commonly shared among subexpressions; and

• `-fno-signed-zeros` ignores the signedness of floating point zeros, for example, \( 0.0f \) and \( -0.0f \) are identical, so that \( 0.0f \times x \) can be simplified to \( 0.0 \) without concerning us with the signedness of the result.

An extra optimization option, which encompasses the above options, can be used to enable them all together; it further highlights the unsafe nature of these transformations in its name, i.e. `-unsafe-math-optimizations`. Besides GCC, Clang, which uses the LLVM framework, provides a similar option, `-enable-unsafe-fp-math` to use arithmetic equivalences to reduce the number of floating-point operations in a program, by possibly sacrificing numerical accuracy.

**High-Level Synthesis Tools**

In addition to the unsafe arithmetic expression rewriting heuristics inherited from the LLVM framework, VHLS [Xil12] and LegUp [LU], which are both LLVM-based, can make use of hardware-specific expression rewriting optimization passes to allow greater parallelism in synthesized circuit, thus improving throughput. By way of illustration, Xilinx’s VHLS has a similar feature called expression balancing [Xil12], which aims to balance an arithmetic expression tree using associativity. A technique known as tree height reduction [NP91] further incorporates distributivity and control-flow rewriting. However, neither of these methods produces optimal loop pipelining, as they do not examine the implications of loop-carried dependences. For example, a loop body:

```
sum = ((sum + A[i]) + B[i]) + C[i];
```

when synthesized in VHLS with expression balancing, will produce a schedule which corresponds to the following:

```
sum = (sum + A[i]) + (B[i] + C[i]);
```
This loop is more efficient than the original, because it has a delay of 2 adders between consecutive iterations, instead of the original 3-adder delay, in the inter-iteration dependences of \( \text{sum} \). However, as we will see below, there is still room for improvement.

Canis et al. [Can+14] propose a similar approach called \textit{recurrence minimization}. They specifically tackle loop pipelining by incrementally restructuring dependence graphs to minimize longest paths of recurrences. Their method is subsequently incorporated in LegUp [LU]. For instance, by synthesizing the same original loop in LegUp, it detects that there are inter-iteration dependences between each pair of \( \text{sum} \) from consecutive iterations. The tool will therefore minimize the latency between these dependences by using associativity to restructure the expression. This optimization produces a schedule which corresponds to the following loop body:

\[
\text{sum} = \text{sum} + ((A[i] + B[i]) + C[i]);
\]

It is notable that, by further delaying the addition of \( \text{sum} \), the loop now has only a delay of 1 adder between consecutive iterations. This technique can greatly reduce the run time of pipelined loops, especially if the inter-iteration dependences has a long chain of additions. However, similar to VHLS, they only apply associativity in their restructuring without regard for accuracy.

**Polynomial Factorization**

The above mentioned tools restrict themselves to simple arithmetic equivalences, as they are intended for fast and simple optimizations that can easily be applied by a compiler. Several techniques take one step further, by focusing on multivariate polynomials, and factorizing them to minimize the number of arithmetic operations in an expression. These approaches are applicable to both software and hardware designs, as they minimize the number of operations in an expression, the throughput of the optimized design can be reduced. In addition, in an FPGA circuit, this will also translate to a reduction of resources utilized.
It is well known that the Horner scheme is the optimal way to factorize a univariate polynomial so that its operator count is minimized [Neu01], e.g. a polynomial $x^3 + 2x^2 + 3x + 4$, which uses 4 multipliers and 3 adders if common subexpressions are eliminated, can be factorized into $x(x(x+2)+3)+4$, which uses 2 fewer multipliers. However, a multivariate polynomial could be expressed in multiple ways using the Horner scheme, and finding the optimal one is a difficult problem. Ceberio et al. [CK04] therefore propose a greedy algorithm to efficiently factorize a multivariate polynomial to overcome this. Hosangadi et al. [Hos+04] propose an algorithm for the factorization of polynomials, in order to eliminate common subexpressions, and subsequently reduce addition and multiplication counts for a faster software implementation. However it is not possible to choose different optimization levels with their method. Peymandoust et al. [PDM01] present an approach that only deals with the factorization of polynomials in HLS using Gröbner bases. The weaknesses of this are its dependence on a set of library expressions [Hos+04] and the high computational complexity of Gröbner bases.

**Shortcomings**

The above approaches utilize a number of heuristics to rewrite expressions, and do not explore all possible rewrites in full by taking into account additional equivalence relations. This could limit their applicability to a small number of special cases. The implementation obtained therefore is often likely to be suboptimal, as further improvements are possible. More importantly, none of the above mentioned techniques and tools aim to minimize, or even analyze, the impact of their transformations on numerical accuracy. HLS tools therefore generally disable these unsafe features by default for floating-point computations. It is in general a good practice to avoid them, if numerical accuracy is a critical factor to ensure the correctness of the application being compiled.

### 2.5.2 Rewriting Arithmetic Expressions for Accuracy

In many numerically sensitive programs, small round-off errors, when accumulated, would result in catastrophically inaccurate results. In particular, Panchekha et al. [Pan+15]
show that inaccurate computations, due to round-off errors, resulted in the retraction of scientific articles [AM99], and even wild discrepancies in stock market indices [BDM99].

In response, the software community has seen an emergence of techniques that rewrite expressions in numerical programs, to ameliorate round-off errors in numerically sensitive applications. However, round-off errors in numerical programs are well known to be perplexing to debug [TM14], and it is in general difficult to apply intuition manually and rewrite programs to optimize numerical accuracy [TM14]. The numerical accuracy optimization techniques therefore often explore the search space of equivalent expressions, using equivalence relations in real arithmetic, ranging from the simplest possible, e.g. associativity, commutativity and distributivity, to more sophisticated ones, such as trigonometry facts, equivalence rules that are known to sometimes improve accuracy

\[ x - y = \frac{x^2 - y^2}{x + y} \] [Pan+15], and many more.

As the vastness of the search space prohibits us to explore it fully, existing approaches confronting this problem resort to heuristics.

Darulova et al. [Dar+13] use genetic programming to evolve the structure of arithmetic expressions into more accurate forms. However there are several disadvantages with metaheuristics, such as convergence can only be proved empirically and scalability is difficult to control because there is no definitive method to decide how long the algorithm must run until it reaches a satisfactory goal.

The method proposed by Martel [Mar07] is based on operational semantics with abstract interpretation, but even their depth limited strategy is, in practice, at least exponentially complex.

Ioualalen et al. [IM12], create a polynomial-size structure, APEG, to represent an exponential number of equivalent expressions related by rules of equivalence. However it restricts itself to only a handful of these rules to avoid combinatorial explosion of the structure and offers no options for tuning its optimization level.

Panchekha et al. [Pan+15] present a tool, Herbie, which employs a greedy hill-climbing heuristic to iteratively rewrite expressions in locations that introduce the largest round-off
errors. Here, detailed overviews are provided for two distinct approaches, APEG and Herbie.

**APEG**

The APEG proposed by Ioualalen et al. [IM12] was inspired by E-PEG, originally introduced by Tate et al. [Tat+]. E-PEGs, originally intended to discover equivalent structure in programs but not arithmetic expressions, nevertheless include equivalence rules that are similar to the ones that exist in real arithmetic. For instance, distributivity can be applied over nodes such as $\theta$ and $\text{eval}$ nodes.

The APEG is similar in construction to the E-PEG, which is also a graph-based structure, and promotes the maximal sharing of common subtrees. The major difference is that because of the lack of control-flow in the program, APEGs are acyclic, and do not consist of nodes that correspond to control-flows. APEGs subsume arithmetic expressions by allowing all possible nodes in an arithmetic expression, i.e. it can contain leaf nodes such as a constant value or a variable identifier, unary or binary arithmetic operators that respectively have one or two child nodes. In addition, to allow an APEG to encode an exponential number of equivalent expressions within a polynomial space, it further introduces two different kinds of nodes that efficiently model equivalences.

Firstly, the APEG defines a node containing an abstraction box, $\otimes, (p_1, p_2, \ldots, p_n)$, where $\otimes$ is a commutative associative operator such as addition $+$ or multiplication $\times$, and $p_1, p_2, \ldots, p_n$ are the children of this node. The abstraction box can be used to represent equivalent expressions generated by different associative parsings of the expression $p_1 \otimes p_2 \otimes \ldots \otimes p_n$. To illustrate, an expression $a + b + c$ can be encoded by the abstraction box $[+, (a, b, c)]$, which can represent all equivalent parsings of the original expression, i.e. $(a + b) + c$, $a + (b + c)$ and $(a + c) + b$. Each abstraction box with $n$ children can represent up to $(2n - 3)!!$ equivalent expressions without commutativity [IM12; Mou11], as commutativity does not impact numerical accuracy for commutative operators.

---

*E-PEG and equality saturation is discussed in Section 2.4.2.*
Secondly, in an analogous fashion to the E-PEG, the APEG additionally admits a new kind of node which can be used to enclose a set of equivalent subexpressions. Defined as \((p_1, p_2, \ldots, p_n)\), the equivalent class is a node which signifies all children subtrees with root nodes \(p_1, p_2, \ldots, p_n\) represent expressions that are mathematically equivalent. This is analogous to forming equivalence edges in E-PEGs.

![Figure 2.13](image.png)

**Figure 2.13.** An example APEG for the expression \(((a + a) + b) \times c\), from [Mar12].

Take Figure 2.13 as our example, which not only contains standard tree nodes which we would expect from an arithmetic expression tree, but also an abstraction box and equivalent classes. Each equivalent class is represented by an ellipse with a dashed border. It is evident that each equivalent class indeed reflects a set of equivalent subtrees. For instance, consider the bottom equivalent class which has two equivalent subtrees, the left and right ones represent \(2 \times a\) and \(a + a\) respectively. In addition, as the subexpression \((a + a) + b\) is a summation of multiple elements, an abstraction box \([+, (a, a, b)]\) can therefore model the equivalent parsings of it.

Despite the compactness of APEGs, searching for the most accurate expression within an APEG is still exponentially complex, where the search utilizes the static analysis of floating-point errors introduced in Section 2.3. For instance, consider an APEG with \(n\) equivalent classes, where each contains 2 different choices. In the worst case, this may amount to a search of \(2^n\) distinct equivalent expressions to determine the optimal solution. To resolve this problem, Ioualalen et al. [IM12] employs a depth-limited search heuristic to reduce the size of the search space. Additionally, as explained earlier, the time complexity to search for the optimal parsing of an abstraction box is double factorial. They hence make use of another heuristic, which greedily pairs up terms in an abstraction.
box $B = +, (p_1, p_2, \ldots, p_n)$. To begin, the method searches for a pair of child subtrees $p_i$ and $p_j$ in $B$, such that the expression $p_i \otimes p_j$ computes with the smallest round-off error. The nodes $p_i$ and $p_j$ are subsequently removed from $B$, and a new subtree $\hat{p_i \otimes p_j}$ is then added to $B$. The above greedy pairing procedure is then iteratively repeated until finally one node is left in $B$; this last node is therefore the expression with an improved numerical accuracy obtained in the process.

Ioualalen et al. [IM12] found their APEG approach can reduce round-off errors in expressions by up to 50%. They further proved the correctness of their APEG approach by forming a Galois connection:

$$\psi \left( \langle \mathcal{E} \rangle_{\mathcal{A}} \right) \xrightarrow{\gamma} \Pi_{>},$$

(2.55)

where $\psi \left( \langle \mathcal{E} \rangle_{\mathcal{A}} \right)$ is all possible traces of rewrites of an initial expression $e$ using associativity, distributivity and commutativity, and $\Pi_{>}$ is the final saturated APEG.

**Herbie**

Unlike the APEG approach which focuses on building an IR that can represent exponentially many equivalent expressions, Herbie, introduced by Panchekha et al. [Pan+15], on the other hand, place emphasis on building an algorithm to efficiently improve numerical accuracy by arithmetic rewrite rules.

Initially, for a given expression $e$, Herbie samples a randomized set of input values used to evaluate it, and we use $I$ to denote this set. In the optimization process, Herbie analyzes the discrepancy between evaluating an expression exactly, and in a given floating-point format with precision $\hat{p}$, due to round-off errors, for a common input $i \in I$.

However, because expressions contain irrational computations such as square roots, and transcendental functions, e.g. trigonometry, logarithm, etc., exact computation is generally unattainable as it could require infinite precision in floating-point. Instead, Herbie resorts to GNU Multiple Precision Floating-Point Reliable Library (MPFR), and iteratively increases the precision used to compute the approximate results for $e$, until the leading 64 bits in the result remain identical across iteration for all randomly sampled inputs $i \in I$. 
Evaluating \( e \) in the final precision \( p \), therefore produces an almost exact (AE) result for \( e \). It is notable that in formulae, the term \( \hat{x} \) over a \( x \) is used to denote an approximation of \( x \), whereas the absence of it indicates an AE result.

An iterative accuracy refinement procedure is then carried out to optimize an initial expression \( e_0 \). This process identifies operators in the expression \( e_0 \) which contribute the largest round-off errors, and specifically applies arithmetic rewriting to these operators, and repeats the process for the resulting expressions. This procedure is carried out for a predetermined number \( N \) of repeats (Herbie uses \( N = 3 \)) as follows.

To begin, for each operator in an original expression \( e \), we accumulate all round-off errors for all sample inputs \( i \in I \). For instance, consider a binary operator \( + \) which accepts two input arguments \( e_1 \) and \( e_2 \), where each argument is a subexpression. To guarantee the correctness of the round-off error analysis of \( + \), its subexpressions are evaluated into respective AE values \( v_1 \) and \( v_2 \) for a given input \( i \). Using \( v_1 \) and \( v_2 \) as inputs, the operator \( + \) is then evaluated twice, in precision \( \hat{p} \) and in AE precision \( p \) to produce \( \hat{v}_i \) and \( v_i \) respectively, where \( \hat{v}_i \) and \( v_i \) are respectively the approximate result of evaluating the addition in floating-point with inputs \( v_1 \) and \( v_2 \). The error between \( x = \hat{v}_i \) and \( y = v_i \) is defined as follows [Pan+15; Sch+14], which counts the number of floating-point values between the approximate and exact results:

\[
\mathcal{E}(x, y) = \log_2 |\{z \in \mathbb{F} | \min(x, y) \leq z \leq \max(x, y)\}|, \tag{2.56}
\]

The \( \mathcal{E}(\hat{v}_i, v_i) \) is then accumulated for all sample inputs \( i \in I \), and the final sum is the total error measure for the operator \( + \), or equivalently:

\[
\text{Error} = \sum_{i \in I} \mathcal{E}(\hat{v}_i, v_i). \tag{2.57}
\]

The top \( M \) operators with the largest errors are then selected to be rewritten in the next stage. Panchekha et al. use \( M = 4 \) in [Pan+15].

---

\(^{†}\)The original paper [Pan+15] uses “exact” to describe what is in fact a floating-point value with a precision \( p \), which approximates the exact value. Because this could potentially mislead, “almost exact” is used here instead.

\( ^{‡} \mathbb{F} \) is the set of floating-point values in a precision which was not mentioned by the original author [Pan+15].
The second step is to rewrite the arithmetic expression, using the information gathered in the previous step about the largest local errors in the expression. The authors of Herbie identify that if subexpression of the selected operators are rewritten, transforming the selected operators' expression tree can provide greater chances of cancelling terms, hence significantly reduce the round-off errors. The rewriting process therefore performs a recursively bottom-up transformation, which first rewrites the smaller subexpressions, then applies transformation rules to the selected operators. In contrast to APEGs, Herbie does not restrict its database of rewrites to the basic arithmetic rules such as associativity and distributivity, as they also admit expressions with fractions, square roots, exponentiation, logarithm and trigonometric functions; additional rules thus must be provided for their corresponding algebraic identities. In addition, Herbie has rules for special polynomial patterns. For instance, it may rewrite $a^3 + b^3$ into $(a + b)(a^2 - ab + b^2)$.

The third step is to further simplify the expression generated from the rewriting step. In this step, a set of simplification rules are applied to the expression for a small number of iterations, which transforms the original into a new expression. However, unlike step 2, as new expressions are discovered, an equivalence graph is gradually constructed. An expression with the smallest tree structure can then be extracted from the graph as the simplified result.

Herbie further identifies that some of these expressions, when evaluated, could produce highly inaccurate outcomes near zero or infinity. In the fourth step, it will then substitute the expression with a series expansion to approximate the original, which may no longer suffer from this problem. This series expansion can be further restructured in future iterations of refinement.

As the above process produce multiple candidate expressions, in the next iteration of the accuracy refinement procedure, the four steps above are applied to all candidate expressions, and these steps will be repeated $N$ times in total, generating a set of accurate expressions that are mathematically equivalent to the original. Finally, because different candidate expressions may excel on different input conditions, a dynamic programming algorithm, is therefore applied to split the input space into multiple parts, where each
part can be computed by different optimized expressions. As an example, consider the following formula for finding a root of the quadratic equation $ax^2 + bx + c = 0$ [Pan+15]:

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$  \hfill (2.58)

Their technique can optimize this expression to generate a scheme of expressions to be evaluated under different conditions, to improve the numerical accuracy of $x_1$, when evaluated with double-precision:

$$x_1 = \begin{cases} 
\frac{4ac}{-b + \sqrt{b^2 - 4ac}} / 2a & \text{if } b < 0, \\
\left(-b - \sqrt{b^2 - 4ac}\right) \frac{1}{2a} & \text{if } 0 \leq b \leq 10^{127}, \\
\frac{b}{a} + \frac{c}{b} & \text{if } 10^{127} < b.
\end{cases} \hfill (2.59)$$

Note that the third alternative, $-\frac{b}{a} + \frac{c}{b}$, is not equivalent to the other two in real arithmetic, because series expansion is applied to approximate the square root.

Panchekha et al. [Pan+15] discovered that their tool can improve the numerical accuracy of a suite of benchmark examples from [Ham86], by recovering up to 60 bits of floating-point precision, with a performance overhead of 40% for the generated expressions.

### 2.5.3 Numerical Programs

The techniques we have explored so far have only been limited to individual arithmetic expressions; for a complete numerical program transformation, not only is it necessary to support sequential execution of straight-line code, but also control-flow structures such as conditional branches and loops. However, the techniques in this research area to optimize numerical accuracy are currently not mature enough to be used in compilers and HLS tools.

Tate et al. [Tat+] with their E-PEGs, enable equivalent programs to be discovered efficiently. However, their method does not consider the implications of rewriting arithmetic expressions in the data-flow on numerical accuracy.
Damouche et al. [Dam+15] developed a new semantics-based method, built on top of the foundation of APEGs from Ioualalen et al. [IM12], for the purpose of accuracy optimization in general numerical programs. Their method formally defines a set of inference rules for rewriting a sequent, a formula used encode the original program, into another one representing an optimized program. To summarize from an informal standpoint, the inference rules analyze the basic blocks in the numerical program, which consist of arithmetic operations and variable assignments, into standard arithmetic expressions. These arithmetic expressions can then be analyzed and optimized by constructing APEGs and extracting an optimized expression from the APEGs. Because they only focus on the basic blocks of the program, containing only data-flows, their approach is not able to optimize across control-flow boundaries such as if statements and while loops. Their method can improve numerical accuracy by approximately 20% for simple loop examples.

To summarize, the lack of the ability to trade-off multiple performance objectives in program optimization provides a strong motivation for the work in this thesis. We believe that rewriting a numerical program could not only improve its numerical accuracy, but also at the same time increase its throughput while optimizing for resource utilization. In addition, as mentioned earlier, all above program optimization methods have their shortcomings, they thus form the basis for this thesis to develop an optimization framework which avoids these disadvantages. Similar to these program optimization techniques, this thesis presents a method which builds on top of the formal semantics of programs. This ensures the correctness of program optimization.
Structural Optimization of Arithmetic Expressions

By exploiting rules of equivalence in arithmetic, such as associativity \((a + b) + c \equiv a + (b + c)\) and distributivity \((a + b) \times c \equiv a \times c + b \times c\), it is possible to automatically generate different implementations of the same arithmetic expression. We optimize the structures of arithmetic expressions in terms of the following two quality metrics relevant to FPGA implementation: the resource usage when synthesized into circuits, and a bound on roundoff errors when evaluated. Our goal is the joint minimization of these two quality metrics. This optimization process provides a Pareto optimal set of implementations. For example, the tool discovers that with single-precision floating-point representation, if \(a \in [0.1, 0.2]\), then the expression \((a + 1)^2\) uses fewest resources when implemented in the form \((a + 1) \times (a + 1)\) but is most accurate when expanded into \(((a \times a) + a) + a + 1\).

However it turns out that a third alternative, \(((1 + a) + a) + (a \times a)\), is never desirable because it is neither more accurate nor uses fewer resources than the other two possible structures. Our aim is to automatically detect and utilize such information to optimize the structure of expressions.

A naïve implementation of equivalent expression finding would be to explore all possible equivalent expressions to find optimal choices. However, this would result in combinatorial explosion [IM12]. Since none of the techniques explained in Section 2.5 of Chapter 2 capture the optimization of both accuracy and performance by restructuring arithmetic expressions, we base ourselves on the software work of Martel [Mar07], but extend their work in the following ways. Firstly, we develop new hardware-appropriate semantics to analyze not only accuracy but also resource usage, seamlessly taking into account common subexpression elimination. Secondly, because we consider both resource usage and accuracy, we develop a novel multi-objective optimization approach to scalably...
construct the Pareto frontier in a hierarchical manner, allowing fast design exploration. Thirdly, equivalence finding is guided by prior knowledge on the bounds of the expression variables, as well as local Pareto frontiers of subexpressions while it is optimizing expression trees in a bottom-up approach, which allows us to reduce the complexity of finding equivalent expressions without sacrificing our ability to optimize expressions. Following Martel [Mar07] and Ioualalen et al. [IM12], the methodology explained in this chapter makes use of formal semantics as well as abstract interpretation [CC77] to significantly reduce the space and time requirements and produce a subset of the Pareto frontier.

In order to further increase the options available in the Pareto frontier, we introduce freedom in choosing mantissa widths for the evaluation of the expressions. Generally as the precision of the evaluation increases, the utilization of resources increases for the same expression. This gives flexibility in the trade-off between resource usage and precision. Our approach and its associated tool, SOAP, allow high-level synthesis flows to automatically determine whether it is a better choice to rewrite an expression, or change its precision in order to meet optimization goals.

The three contributions of this chapter are:

1. Efficient methods for discovering equivalent structures of arithmetic expressions.

2. A semantics-based program analysis that allows joint reasoning about the resource usage and safe ranges of values and errors in floating-point computation of arithmetic expressions.

3. A tool which produces RTL implementations on the area-accuracy trade-off curve derived from structural optimization.

This chapter is structured as follows. Sections 3.1 and 3.2 extend the basic concepts of semantics with abstract interpretation explained in Section 2.3 to respectively analyze accuracy and resources. Section 3.3 discusses various abstract semantics for finding equivalent structure in arithmetic expressions, as well as the analysis of their resource usage estimates and bounds of errors. Section 3.4 gives an overview of the implementa-
tion details in SOAP. Finally, we discuss the results of optimized example expressions in Section 3.5 and end with concluding remarks for this chapter in Section 3.6.

3.1 Accuracy Analysis

The accuracy analysis used by SOAP follows the method based on abstract error domain introduced by Martel [Mar07] to analyze the round-off error of restructured floating-point expressions. As it was mentioned in Section 2.3.4, they did not have a preference for the choice of definition of $\text{ulp}$. Hence, here we propose to use a definition of $\text{ulp}$ derived from the standard IEEE 754 floating-point representation.

To begin, the concepts of the floating-point representation [ANS08] should be introduced. Any values $v$ representable in floating-point with standard exponent offset can be expressed with the format given by the following equation:

$$v = s \times 2^{e-(2^{k-1}-1)} \times 1.m_1m_2m_3\ldots m_p.$$  \hspace{1cm} (3.1)

In (3.1), the bit $s$ is the sign bit, the $k$-bit unsigned integer $e$ is known as the exponent bits, and the $p$-bits $m_1m_2m_3\ldots m_p$ are the mantissa bits, here we use $1.m_1m_2m_3\ldots m_p$ to indicate a fixed-point number represented in unsigned binary format.

Note that a non-zero floating-point with the smallest magnitude can be found by setting $e = 0$ and $m_1, m_2, \ldots, m_p$ to 0, which may still be far away from 0 when compared to the next non-zero floating-point value, where $e = 0$, $m_1, m_2, \ldots, m_{p-1}$ are all 0, and $m_p = 1$. To solve this discontinuous behaviour, extra logic can be used to allow a different floating-point representation when $e = 0$:

$$v = s \times 2^{-(2^{k-1}-1)} \times 0.m_1m_2m_3\ldots m_p.$$  \hspace{1cm} (3.2)

This alternative behaviour is known as gradual underflow, whereas the original is abrupt underflow.
In SOAP, the distance between two adjacent floating-point values $f_1$ and $f_2$ satisfying $f_1 \leq x \leq f_2$ for a value $x$ [Gol91] is known as the unit of the last place function $\text{ulp}(x)$. To characterize this function, we further restrict that $f_1$ and $f_2$ must not equal and no other floating-point values exists between them. We can now provide the definition for $\text{ulp}(x)$, where GU and AU respectively indicate gradual and abrupt underflow modes:

**Definition 3.1.** In our analysis, the function $\text{ulp}(x)$ is defined as:

$$\text{ulp}(x) = \begin{cases} 
\infty, & \text{if } x = -\infty \text{ or } \infty, \\
2^{e(x)-(2^{k-1}-1)} \times 2^{-p}, & \text{if GU and } x \text{ is not } -\infty \text{ or } \infty, \\
\max \left(2^{-(2^{k-1}-1)}, 2^{e(x)-(2^{k-1}-1)} \times 2^{-p}\right) & \text{if AU and } x \text{ is not } -\infty \text{ or } \infty.
\end{cases}$$

(3.3)

where $e(x)$ is the exponent of $x$, $k$ and $p$ are the parameters of the floating-point format as defined in (3.1).

Since Martel [Mar07] does not define the arithmetic operator for division. The following equations for division are therefore introduced in this thesis. Firstly, divisions on intervals can be implemented as follows:

$$\left[\begin{array}{c} a, \ b \\ c, \ d \end{array}\right] := \left[\begin{array}{c} \min(s), \ \max(s) \end{array}\right],$$

(3.4)

where $[a, b], [c, d] \in \text{Interval}$ and:

$$s = \begin{cases} 
\{-\infty, \infty\} & \text{if } c \leq 0 \leq d, \\
\left\{\frac{a}{d}, \frac{a}{d}, \frac{b}{c}, \frac{b}{c}\right\} & \text{otherwise}.
\end{cases}$$

(3.5)

By evaluating the sum of error propagated $\frac{x_1 \mu_1}{x_2 + \mu_2} - \frac{x_1}{x_2}$ and the round-off error introduced by division, the division on values in the abstract error domain can be derived as follows:

$$\left(\begin{array}{c} x_1 \mu_1 \\ x_2 \mu_2 \end{array}\right) := \left(\begin{array}{c} x_1 \mu_1 \\ x_2 \mu_2 \end{array}\right) - \left(\begin{array}{c} x_2 \mu_1 - x_1 \mu_2 \\ x_2 \left(\frac{x_1}{x_2} + \mu_2\right) + 4t_0 \left(\frac{x_1}{x_2}\right) \end{array}\right).$$

(3.6)
Recall from Section 2.3.4 of Chapter 2, \( \uparrow^{\xi} \left( \frac{x_1}{x_2} \right) \) computes the range of floating-point values by rounding values in \( \frac{x_1}{x_2} \), and \( \downarrow^{\xi} \left( \frac{x_1}{x_2} \right) \) returns the range of errors introduced in the rounding process.

We use the function \( \text{Error} : \text{AExpr} \rightarrow \mathbb{E}^{\xi} \) to represent the analysis of round-off error in an expression tree, as described in Section 2.3.4 of Chapter 2, using the above ulp equation in Definition 3.1, where \( \text{AExpr} \) denotes the set of all arithmetic expressions.

For each expression in a set of equivalent expressions discovered, \( e \in \epsilon \), each expression \( e \) evaluates to a distinct value in the abstract error domain. Section 2.3.4 of Chapter 2 presents a method to compare against each other with a partial ordering. However, a total ordering is much more preferable, as all expressions can be easily compared against one another. In SOAP, the following function \( \text{AbsError} \) is used to convert an evaluated outcome \( v \in \mathbb{E}^{\xi} \) into a scalar to denote the magnitude of round-off error:

\[
\text{AbsError}(e) = \max \left( |\mu^{\xi}_{\min}|, |\mu^{\xi}_{\max}| \right), \quad \text{where} \quad \left( x^t, [\mu^{\xi}_{\min}, \mu^{\xi}_{\max}] \right) = \text{Error}(e). \quad (3.7)
\]

### 3.2 Resource Usage Analysis

Here we define similar formal semantics which calculate an approximation to the FPGA resource usage of an expression, taking into account common subexpression elimination. This is important as, for example, rewriting \( a \times b + a \times c \) as \( a \times (b + c) \) in the larger expression \( (a \times b + a \times c) + (a \times b)^2 \) causes the common subexpression \( a \times b \) to be no longer present in both terms. Our analysis must capture this.

The analysis proceeds by labelling subexpressions. Intuitively, the set of labels \( \text{Label} \), is used to assign unique labels to unique expressions, so it is possible to easily identify and reuse them. For convenience, let the function \( \text{fresh} : \text{AExpr} \rightarrow \text{Label} \) assign a distinct label to each expression or variable, where \( \text{AExpr} \) is the set of all expressions. It is noteworthy that \( \text{fresh} \) is a bijection. Before we introduce the labeling semantics, we define the environment \( \lambda : \text{Label} \rightarrow \text{AExpr} \cup \{ \perp \} \), which is a function that maps labels

3.2 Resource Usage Analysis
to expressions, and \( \text{Env} \) denotes the set of such environments. A label \( l \) in the domain of \( \lambda \in \text{Env} \) that maps to \( \perp \) indicates that \( l \) does not map to an expression. An element \( (l, \lambda) \in \text{Label} \times \text{Env} \) stands for the labeling scheme of an expression. Initially, we map all labels to \( \perp \), then in the mapping \( \lambda \), each leaf of an expression is assigned a unique label, and the unique label \( l \) is used to identify the leaf. That is for the leaf variable or constant \( x \):

\[
(l, \lambda) = (\text{fresh}(x), [\text{fresh}(x) \mapsto x]).
\]

This equation uses \([\text{fresh}(x) \mapsto x]\) to indicate an environment that maps the label \( \text{fresh}(x) \) to the expression \( x \) and all other labels map to \( \perp \), in other words, if \( l = \text{fresh}(x) \) and \( l' \neq l \), then \( \lambda(l) = x \) and \( \lambda(l') = \perp \).

For example, consider the expression \((a + b)^2 = (a + b) \times (a + b)\), we initially have for the variables \( a \) and \( b \):

\[
(l_a, \lambda_a) = (\text{fresh}(a), [\text{fresh}(a) \mapsto a]) = (l_1, [l_1 \mapsto a]),
\]

\[
(l_b, \lambda_b) = (l_2, [l_2 \mapsto b]).
\]

Then the environments are propagated in the flow direction of the DFG, using the following formulation of the labeling semantics:

\[
(l_x, \lambda_x) \otimes (l_y, \lambda_y) = (l, (\lambda_x \odot \lambda_y)[l \mapsto l_x \otimes l_y]),
\]

where \( l = \text{fresh}(l_x \otimes l_y), \odot \in \{+, -, \times\} \).

Specifically, \( \lambda = \lambda_x \odot \lambda_y \) signifies that \( \lambda_y \) is used to update the mapping in \( \lambda_x \), if the mapping does not exist in \( \lambda_x \), and results in a new environment \( \lambda \); and \( \lambda[l \mapsto x] \) is a
shorthand for \( \lambda \odot [l \mapsto x] \). As an example, using (3.9), recall to mind that \( l_1 = l_a, l_2 = l_b \), we derive for the subexpression \( a + b \):

\[
(l_{a+b}, \lambda_{a+b}) = (l_a, \lambda_a) + (l_b, \lambda_b)
\]

where \( l_3 = fresh(l_a + l_b) \)

\[
= (l_3, (\lambda_a \odot \lambda_b) [l_3 \mapsto l_a + l_b])
\]

\[
= (l_3, [l_1 \mapsto a] \odot [l_2 \mapsto b] \odot [l_3 \mapsto l_1 + l_2])
\]

\[
= (l_3, [l_1 \mapsto a, l_2 \mapsto b, l_3 \mapsto l_1 + l_2]),
\]

where \( l_a + l_b \) is a syntactic construct to signify that the subexpressions with labels \( l_a \) and \( l_b \) are added to form an expression. Finally, for the full expression \( (a + b) \times (a + b) \):

\[
(l, \lambda) = (l_{a+b}, \lambda_{a+b}) \times (l_{a+b}, \lambda_{a+b})
\]

\[
= (l_4, [l_1 \mapsto a, l_2 \mapsto b, l_3 \mapsto l_1 + l_2, l_4 \mapsto l_3 \times l_3]).
\]

From the above derivation, it is clear that the semantics capture the reuse of subexpressions. The estimation of area is performed by counting, for an expression, the numbers of additions, subtractions and multiplications in the final labeling environment, then calculating an approximation to the number of LUTs used to synthesize the expression by adding the LUTs requirement for each operator. If the number of operators is \( n_\odot \) where \( \odot \in AOp \), and \( AOp \) denotes the set of arithmetic operators, then the number of LUTs in total for the expressions is estimated as:

\[
R_{LUT} = \sum_{\odot \in AOp} n_\odot R^{LUT}_\odot,
\]

where the value \( R^{LUT}_\odot \) denotes the number of LUTs per \( \odot \) operator, which is dependent on the type of the operator and the floating-point format used to generate the operator.

Despite the simplicity of the resource estimation model, it is observed that this model can accurately predict the LUT count of the synthesized circuits, as the resource reduction from cross-module optimizations are negligible for floating-point data-paths. In later chapters, we will further demonstrate how our approach can be extended to provide estimate DSP block counts additionally.
In the following sections, we use the function \( \text{Area} : \text{AExpr} \rightarrow \mathbb{N} \) to denote our resource usage analysis.

### 3.3 Equivalent Expressions Analysis

In Section 2.3.4 of Chapter 2, we introduce semantics that define additions and multiplications on intervals, then transition to error semantics that compute bounds of values and errors. In Section 3.2, an alternative semantics that eliminate common subexpressions are introduced, which we call the labelling environments. The environments define the meaning of arithmetic operations on the environments. In this section, we now take the leap from not only analyzing an expression for its quality, to defining arithmetic operations on sets of equivalent expressions, and use these rules to discover equivalent expressions. Before this, it is necessary to formally define equivalent expressions and the functions used to discover them.

#### 3.3.1 Discovering Equivalent Expressions

From an expression, a set of equivalent expressions can be discovered by our equivalence relation \( \equiv \) on the set of all arithmetic expressions \( \text{AExpr} \), and \( \equiv \) is a strict subset of \( \text{AExpr} \times \text{AExpr} \). It is noteworthy that a relation is said to be an equivalence relation when it is reflexive, symmetric and transitive, i.e. for all \( \epsilon_1, \epsilon_2, \epsilon_3 \in \text{AExpr} \), we have the following rules in our inference system:

\[
\begin{align*}
\text{[Reflexivity]} & \quad \epsilon_1 \equiv \epsilon_1, \\
\text{[Symmetry]} & \quad \frac{\epsilon_1 \equiv \epsilon_2}{\epsilon_2 \equiv \epsilon_1}, \\
\text{[Transitivity]} & \quad \frac{\epsilon_1 \equiv \epsilon_2 \land \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3}.
\end{align*}
\] (3.14)

Here, a rule of the form \( \frac{a}{b} \) means that if the formula \( a \) is true, then \( b \) also holds. We extend our inference system with additional arithmetic rules that relate equivalent
expressions. Let's define $e_1, e_2, e_3 \in \text{AExpr}$, $v_1, v_2, v_3 \in \mathbb{R}$. Firstly, we start with a set of associativity rules:

\[ [\text{Assoc}_1] \quad \forall \otimes \in \{+, \times\} : (e_1 \otimes e_2) \otimes e_3 \equiv e_1 \otimes (e_2 \otimes e_3), \]
\[ [\text{Assoc}_2] \quad \forall \otimes \in \{+, \times, /\} : (e_1 \otimes e_2) \otimes e_3 \equiv (e_1 \otimes e_3) \otimes e_2, \]
\[ [\text{Assoc}_3(\div)] \quad (e_1/e_2)/e_3 \equiv e_1/(e_2 \times e_3), \]
\[ [\text{Assoc}_4(\div)] \quad e_1/(e_2/e_3) \equiv e_1 \times e_3/e_2. \]

Secondly, we define commutativity rules for addition and multiplication:

\[ [\text{Commut}] \quad \forall \otimes \in \{+, \times\} : e_1 \otimes e_2 \equiv e_2 \otimes e_1. \]

Thirdly, distributivity rules enable further equivalent expressions to be explored with our relation:

\[ [\text{Distrib}_1] \quad \forall \otimes \in \{\times, /\} : e_1 \otimes e_2 \times e_3 \equiv (e_1 + e_2) \otimes e_3, \]
\[ [\text{Distrib}_2(\times)] \quad e_1 + e_1 \times e_2 \equiv (1 + e_2) \times e_1, \]
\[ [\text{Distrib}_3(\times)] \quad e_1 + e_1 \equiv 2 \times e_1, \]
\[ [\text{Distrib}_4(\div)] \quad e_1 + e_2/e_3 \equiv (e_1 \times e_3 + e_2)/e_3, \]
\[ [\text{Distrib}_5(\div)] \quad e_1/e_3 + e_2/e_4 \equiv (e_1 \times e_4 + e_2 \times e_3)/(e_3 \times e_4), \]
\[ [\text{Distrib}_6(\div)] \quad \forall \otimes \in \{+, -\} : -(e_1 \otimes e_2) \equiv (-e_1) \otimes (-e_2), \]
\[ [\text{Distrib}_7(\div)] \quad \forall \otimes \in \{\times, /\} : -(e_1 \otimes e_2) \equiv -(e_1 \otimes e_2) \equiv e_1 \otimes (-e_2). \]

The following rule rewrites an expression with subtraction into an addition:

\[ [\text{Subtract}] \quad e_1 - e_2 \equiv e_1 + (-e_2). \]

The following reduction rules propagates constant values in expression trees. The $\text{ConstProp}_1$ rule states that if an expression is an arithmetic operation of two constant
values, then it can be simply evaluated to produce the result; and ConstProp2 rewrites an expression that has a negation on a constant value into a single constant.

\[
\text{[ConstProp1]} \quad \frac{(v_2 = v_1 \otimes v_2) \wedge (\otimes \in \text{BinAOp})}{v_1 \otimes v_2 \equiv v_3}, \quad (3.19)
\]

\[
\text{[ConstProp2(−)]} \quad \frac{-v_2 = -v_1}{-v_1 \equiv v_2}.
\]

Here we define BinAOp = \{+,-,\times,\div\} to be the set of binary arithmetic operators. We also define another set of reduction rules looking for common patterns that can be simplified:

\[
\begin{align*}
\text{[Identity}_1] & \quad \forall \otimes \in \{\times,\div\} : e_1 \otimes 1 \equiv e_1, & \text{[Elim}_1(-)] & \quad e_1 - e_1 \equiv 0, \\
\text{[Identity}_2] & \quad \forall \otimes \in \{+,\neg\} : e_1 \otimes 0 \equiv e_1, & \text{[Elim}_2(/)] & \quad e_1/e_1 \equiv 1, \\
\text{[ZeroProp(\times)]} & \quad 0 \times e_1 \equiv 0, & \text{[DoubleNeg(−)]} & \quad -(−e_1) \equiv e_1.
\end{align*}
\]

(3.20)

Finally, the following two allow structural induction on expression trees, e.g. it is possible to derive that \(a + (b + c) \equiv a + (c + b)\) from \(b + c \equiv c + b\):

\[
\begin{align*}
\text{[Tree}_1] & \quad \frac{(e_1 \equiv e_2) \wedge (e_3 \equiv e_4) \wedge (\otimes \in \text{BinAOp})}{e_1 \otimes e_3 \equiv e_2 \otimes e_4}, \\
\text{[Tree}_2] & \quad \frac{e_1 \equiv e_2}{-e_1 \equiv -e_2}.
\end{align*}
\]

(3.21)

We say that \(e_1\) is equivalent to \(e_2\) if and only if \(e_1 \equiv e_2\). For some expressions \(e_1\) and \(e_2\). Many rules are considered redundant and thus excluded from our equivalence relation, as these can be derived by combining rules from our inference system. For instance, \(e_1 \times (e_2 + e_3) \equiv e_1 \times e_2 + e_1 \times e_3\) can be derived by using Distrib1, in tandem with the Commut rule.

### 3.3.2 Scalable Methods for Rewriting

The above rules of equivalence relate an expression with all of its equivalent expressions. In general because of combinatorial explosion, the set of all equivalent expressions is so large to be derived, which motivates us to develop scalable methods that execute fast enough even with large expressions.
Instead of deriving the full set of equivalent expressions, we can define a new relation \( \rightsquigarrow \), a subset of \( \equiv \), which is identical to our equivalent relation \( \equiv \) except that we place a few restrictions on the relation. This new relation can be generated by removing the equivalence relation rules in (3.14). Firstly, reflexivity can be removed, because it is not necessary to rediscover expressions. Secondly, we disable transitivity from \( \equiv \), as we can have the flexibility to apply \( \rightsquigarrow \) in a series of steps to generate equivalent expressions. Finally, we further disallow symmetry in (3.14) for the reduction rules in (3.19) and (3.20) to reduce the space-time complexity of the search space, because often performance metrics, such as accuracy and resource usage, improve when the number of terms in the expression is reduced.

To make use of the new relation we define the following category of functions:

**Definition 3.2.** We call a function an equivalent expression generator (EEG) function if and only if the function takes as an input an initial set of equivalent expressions, and generates another set of expressions equivalent to those in the input set.

For instance, an EEG function \( \triangleright : \wp(AExpr_\equiv) \to \wp(AExpr_\equiv) \), where \( \wp(AExpr_\equiv) \) denotes the power set of all equivalent expressions \( AExpr_\equiv \), can be defined as follows:

\[
\triangleright(\epsilon) = \{ e' \in AExpr \mid e \rightsquigarrow e' \land e \in \epsilon \},
\]

where \( \epsilon \) is a set of equivalent expressions.

We define a functional:

\[
cl_N : (\wp(AExpr_\equiv) \to \wp(AExpr_\equiv)) \to (\wp(AExpr_\equiv) \to \wp(AExpr_\equiv)),
\]

which takes as an input a EEG function and produces another EEG function:

\[
f'(\epsilon) := \bigcup_{i=0}^{N} f^i(\epsilon), \text{ where } f' = cl_N(f),
\]
here, $f$ and $f'$, respectively the input and output of $cl_N$, are both EEG functions. In the rest of the section, we omit the brackets surrounding the input of $cl_N$ for simplicity, e.g. $cl_N(f)$ can be written as $cl_N f$.

As an example use of the functional $cl_N$, we may note that we can substitute $f$ with $\triangleright$ in $cl_N f(\epsilon)$ to generate a set of equivalent expressions, by taking the union of $N$ steps of repeated application of $\triangleright$ to $\epsilon$. By further allowing $N$ to approach $\infty$, we obtain the full set of equivalent expressions of $\epsilon$ that can be discovered using our inference system, i.e. the transitive closure of equivalent expressions related by $f$, from an initial set of equivalent expressions $\epsilon$:

$$cl_\infty \triangleright (\epsilon) = \bigcup_{i=0}^{\infty} \triangleright^i(\epsilon).$$

(3.25)

Alternatively, we can view $cl_\infty f$ as computing the least fixpoint of $g$:

$$\text{lfp } g = \bigcup_{i=0}^{\infty} g^i(\emptyset), \quad \text{where } g(\epsilon) := f(\epsilon) \cup \epsilon. \quad (3.26)$$

We may further omit the $\infty$ from $cl_\infty$ to denote the transitive closure, e.g. the above example in (3.25) can be simplified to be $cl \triangleright (\epsilon)$.

In practice, it is often infeasible to generate the full transitive closure of a given expression, we therefore impose further constraints on how we discover equivalent expressions.

Firstly, instead of exploring the full transitive closure, that is, by allowing the number of steps $N$ in (3.24) to be infinite, we may restrict $N$ to be a small finite value to allow a smaller set of equivalent expressions to be computed. In later experiments, we have chosen $N = 10$.

Secondly, the complexity of equivalent expression finding is reduced by fixing the structure of subexpressions at a certain depth $k$ in the original expression. The definition of depth is given as follows: first the root of the parse tree of an expression is assigned depth $d = 1$; then we recursively define the depth of a node as one more than the depth of its greatest-depth parent. If the depth of the node is greater than $k$, then we fix the structure of its child nodes by disallowing any equivalence transformation beyond
this node. We let ◁ₖ denote this “depth-limited” equivalence finding function, where
ₖ is the depth limit used. We can then use clₙ ◁ₖ and cl ◁ₖ to denote the functions to
respectively compute the union of N steps of ◁ₖ and the transitive closure. This approach
is similar to Martel’s depth-limited equivalent expression transform [Mar07], however
Martel’s method eventually allows transformation of subexpressions beyond the depth
limit, because rules of equivalence would transform these to have a smaller depth. This
contributes to a time complexity at least exponential in terms of the expression size.
In contrast, our technique has a time complexity that does not depend on the size of
the input expression, but grows with respect to the depth limit ₖ. Note that the full
equivalence closure using the inference system we defined earlier in (3.25) is at least
O((2ⁿ − 3)!!) where n is the number of terms in an expression, as we discussed earlier.
As the maximum number of terms in a binary tree with a depth ₖ grows at a rate O(2ᵏ),
the number of equivalent expressions that can be discovered is at least O((2 × 2ᵏ − 3)!!)
with respect to ₖ. In the production of experimental results, ₖ is chosen to be either 2 or
3.

Finally, we use an iterative algorithm to accelerate the computation of clₙ f(ε), where f
is a ∪-distributive EEG (see Definition 3.3) such as ◁ₖ. In each iteration, we keep track of
the equivalent expressions that are newly discovered in the current iteration, so that in
the next iteration we apply f only to those expressions, to avoid redundant computation.
This algorithm is shown in Figure 3.1 to efficiently compute clₙ f(ε), where f can be ◁ₖ.
The correctness of this algorithm is discussed in greater depth in Appendix A.

**Definition 3.3.** **We say an EEG function f is ∪-distributive if and only if the function
satisfies** f(εₐ ∪ ε₉) = f(εₐ) ∪ f(ε₉).

The algorithms we have described so far do not incorporate analyses detailed in Sec-
tions 3.1 and 3.2, hence, they do not guide the optimization process with objectives
to minimize. The following section explains how the analyses can be used to steer the
algorithms to optimize the trade-off between accuracy and area in synthesized circuits of
transformed expressions.

3.3 Equivalent Expressions Analysis
function CLOSURE\((f, N, \epsilon)\)
\[ s_0 \leftarrow \epsilon \]
\[ s'_0 \leftarrow \epsilon \]
for \( i \leftarrow 1, \ldots, N \) do
\[ s'_i \leftarrow f (s'_{i-1}) \setminus s_{i-1} \]
\[ s_i \leftarrow s_{i-1} \cup s'_i \]
if \( s'_i = \emptyset \) then
\[ \text{return } s_i \]
end if
end for
return \( s_i \)
end function

Figure 3.1. Our algorithm to compute \( cl_N f(\epsilon) \), which discovers a set of equivalent expressions with a \( \cup \)-distributive EEG \( f \) from an initial set of equivalent expressions \( \epsilon \).

3.3.3 Pareto Frontier

Because we optimize expressions in two quality metrics, i.e. the accuracy of computation and the estimate of FPGA resource utilization, there is a trade-off between them. We desire the largest subset of all equivalent expressions \( E \) discovered such that in this subset, no expression dominates any other expression, in terms of having both better area and better accuracy. This subset is known as the Pareto frontier [Leg+10].

Figure 3.2 shows a simplified algorithm for calculating the Pareto frontier for a set of equivalent expressions \( \epsilon \), faster algorithms exist. For instance, we could sort these expressions by accuracy, as a fast sorting algorithm takes \( O(n \log(n)) \) for \( n \) elements, and then prune suboptimal ones in one pass.

Here, \( \text{frontier}/\{e\} \) is a set identical to \( \text{frontier} \), except that the element \( e \) is removed.
We use the function \( \text{AbsError} \) to analyze the magnitudes of error bounds as defined in (3.7).

3.3.4 Equivalent Expressions Semantics

Similar to the analysis of accuracy and resource usage, a set of equivalent expressions can be computed with semantics. That is, we define structures, i.e. sets of equivalent expressions, that can be manipulated with arithmetic operators. In our equivalent
function FRONTIER(\(\epsilon\))
    frontier ← \(\epsilon\)
    for \(e \in \epsilon\) do
        for \(e' \in \epsilon\) do
            if Area(\(e'\)) < Area(\(e\)) and AbsError(\(e'\)) < AbsError(\(e\)) then
                frontier ← frontier/\{e\}
            end if
        end for
    end for
    return frontier
end function

Figure 3.2. The algorithm used to compute \(fr(\epsilon)\), i.e. the Pareto frontier from a set of equivalent expressions \(\epsilon\).

expressions semantics, an element of \(\wp(\text{AExpr}_=)\) is used to assign a set of expressions to each node in an expression parse tree. To begin with, at each leaf of the tree, the variable or constant is assigned a set containing itself, as for \(x\), the set \(\epsilon_x\) of equivalent expressions is \(\epsilon_x = \{x\}\). After this, we propagate the equivalence expressions in the parse tree’s direction of flow, using (3.27) defined below, where \(fr\) is the algorithm shown in Figure 3.2:

\[
\epsilon_x \otimes \epsilon_y := fr\left(cl_{\downarrow_k} (E_{\otimes} (\epsilon_x, \epsilon_y))\right),
\]

where \(E_{\otimes}(\epsilon_x, \epsilon_y) = \{e_x \otimes e_y \mid e_x \in \epsilon_x \land e_y \in \epsilon_y\}\), \(\otimes \in \{+, -, \times, /\}\). \hfill (3.27)

It is noteworthy that we override the meaning of \(\otimes\), from arithmetic computations originally, to denote the construction of equivalent expressions. The equation implies that in the propagation procedure, it recursively constructs a set of equivalent subexpressions for the parent node from two child expressions, and uses the depth-limited equivalence function \(cl_{\downarrow_k}\) to work out a larger set of equivalent expressions. Similarly, we can define another equation that propagates equivalent subexpressions in an expression with a unary subtraction:

\[
-\epsilon := fr\left(cl_{\downarrow_k} (E_{\text{unary}} (\epsilon))\right),
\]

where \(E_{\text{unary}} (\epsilon) = \{-e \mid e \in \epsilon\}\). \hfill (3.28)
To reduce computation effort, we select only those expressions on the Pareto frontier for the propagation in the DFG. Although in worst case the complexity of this process is exponential, the selection by Pareto optimality accelerates the algorithm significantly. For example, consider the sample DFG in Figure 3.3, for the subexpression \( a + b \), we have:

\[
\epsilon_a + \epsilon_b = \text{fr}(\text{cl} \uparrow_k (E \otimes (\epsilon_a, \epsilon_b)))
\]

\[
= \text{fr}(\text{cl} \uparrow_k (E \otimes \{a\}, \{b\}))
\]

\[
= \text{fr}(\{a + b, b + a\}).
\]

(3.29)

Figure 3.3. The DFG for the sample expression \((a + b) \times (a + b)\).

Alternatively, we could view the semantics in terms of DFGs representing the algorithm for finding equivalent expressions. The parsing of an expression directly determines the structure of its DFG. For instance, consider the tree structure of the expression \( e_0 = (a + b) \times (a + b) \), as shown in Figure 3.3. This tree structure can be used to generate a DFG illustrated in Figure 3.4, which when data-flow analysis is applied, discovers a set of equivalent expressions to \( e_0 \). The circles labeled 3 and 7 in this diagram are shorthands for the operations \( E_+ \) and \( E_x \) respectively, where \( E_+ \) and \( E_x \) are defined in (3.27).

Figure 3.4. The DFG for finding equivalent expressions of \((a + b) \times (a + b)\).
For our example in Figure 3.4, similar to the construction of data-flow equations in Section 2.3 of Chapter 2, we can produce a set of equations from the data-flow of the DFG, which now produces equivalent expressions:

\[
\begin{align*}
A(1) &= A(1) \cup \{a\}, & A(2) &= A(2) \cup \{b\}, \\
A(3) &= E_+(A(1), A(2)), & A(4) &= A(3) \cup A(5), \\
A(5) &= \triangleright_k(A(4)), & A(6) &= fr(A(5)), \\
A(7) &= E_x(A(6), A(6)), & A(8) &= A(7) \cup A(9), \\
A(9) &= \triangleright_k(A(8)), & A(10) &= fr(A(9)).
\end{align*}
\]  

(3.30)

By solving this system of equations for the value \(A(10)\), we find a set of expressions that are equivalent to the original that produce an optimized trade-off between area and accuracy. Because of loops in the DFG, it is no longer trivial to find the solution. In general, the analysis equations are solved iteratively, using the DFA approach discussed in Section 2.3.1 of Chapter 2. We can regard the set of equations as a single transfer function \(F\) as in (3.31), where the function \(F\) takes as input the variables \(A(1), \ldots, A(10)\) appearing in the right-hand sides of (3.30) and outputs the values \(A(1), \ldots, A(10)\) appearing in the left-hand sides. Our aim is then to find an input \(\vec{x}\) to \(F\) such that \(F(\vec{x}) = \vec{x}\), i.e. a fixpoint of \(F\).

\[
F((A(1), \ldots, A(10))) = (A(1) \cup \{a\}, \ldots, fr(A(9))).
\]  

(3.31)

Initially we assign \(A(i) = \emptyset\) for \(i \in \{1, 2, \ldots, 10\}\), and we denote \(\emptyset = (\emptyset, \ldots, \emptyset)\). Then we compute the least fixpoint of \(F\):

\[
\text{lfp } F := \bigcup_{n \in \mathbb{R}} F^n(\emptyset).
\]  

(3.32)

This expression can be computed iteratively by first evaluating \(F(\emptyset)\), \(F^2(\emptyset) = F(F(\emptyset))\), and so forth, until the fixpoint is reached for some iteration \(n\), i.e. \(F(F^n(\emptyset)) = F^{n+1}(\emptyset)\). Hence, we know that for any iterations \(m > n + 1\), \(F^m(\emptyset) = F^n(\emptyset)\). The value \(n\) should be a finite constant, because the relation \(\sim\) can only reach a finite number of expressions.
In cases when $lfp F$ is computational intensive, we could limit the number of iterations $n$, to compute an under-approximation (a subset) of $lfp F$.

The fixpoint solution $lfp F$ gives a set of equivalent expressions derived using our method, which is found at $A(10)$. In essence, the depth limit acts as a sliding window. The semantics allow hierarchical transformation of subexpressions using a depth-limited search and the propagation of a set of subexpressions that are locally Pareto optimal to the parent expressions in a bottom-up hierarchy.

The problem with the semantics above is that the time complexity of $cl \triangleright_k$ scales poorly, since the worst case number of subexpressions needed to explore increases exponentially with $k$. Therefore an alternative method is to optimize it by changing the structure of the DFG slightly, as shown in Figure 3.5. The difference is that at each iteration, the Pareto frontier filters the results to decrease the number of expressions to process for the next iteration, whereas the former approach filters the Pareto-suboptimal candidates only at the end of the iterative procedure. The latter method is therefore pruning the set of discovered candidates more frequently than the former. Equivalently, this approach yields the following semantics for arithmetic operations on equivalent expressions as an alternative to (3.27):

$$
\epsilon_x \otimes \epsilon_y := cl (fr \circ \triangleright_k) (E_\otimes (\epsilon_x, \epsilon_y)), \quad \text{where } \otimes \in \{+, -, \times, /\},
$$

$$(3.33)
$$

$$
-\epsilon := cl (fr \circ \triangleright_k) (E_{\text{unary}} (\epsilon)).
$$

**Figure 3.5.** The alternative DFG for $(a + b) \times (a + b)$.

In the rest of this chapter, we use `frontier_trace` to indicate our equivalent expression finding semantics, and `greedy_trace` to represent the alternative method.
In addition to the above approaches, another possibility is to view the optimization in a perspective from denotational semantics. We can define a recursively-defined function $O[e] \sigma^\sharp$ which accepts an expression $e$, and $\sigma^\sharp$, an input condition on the variables. This function produces a set of optimized expressions equivalent to $e$. The set of input conditions will be formally defined in Chapter 4. Therefore for $e_1, e_2 \in \text{AExpr}$ and a variable $x$:

$$O[e_1 \otimes e_2] \sigma^\sharp = f_{\sigma^\sharp} \left( \left\{ e'_1 \otimes e'_2 \mid e'_1 \in O[e_1] \sigma^\sharp, e'_2 \in O[e_2] \sigma^\sharp \right\} \right),$$

$$O[x] \sigma^\sharp = \{x\},$$

(3.34)

where $f_{\sigma^\sharp}$ is a function that transforms and optimizes a set of equivalent expressions based on the initial condition $\sigma^\sharp$, e.g. $fr \circ cl \uparrow k$ or $cl(fr \circ k)$ used in $\text{frontier\_trace}$ or $\text{greedy\_trace}$.

### 3.3.5 Simultaneous Optimization of Multiple Expressions

To simultaneously optimize multiple expressions, a new operator, the barrier operator “|”, is introduced to concatenate expressions. Multiple expressions, when concatenated, allow common subexpressions to be shared. For instance, the expressions $a + (b + c)$ and $a \times (b + c)$ can concatenate to form a new expression $e$:

$$e = a + (b + c) \mid a \times (b + c),$$

(3.35)

and the subexpression $b + c$ is shared within $e$, as this behaviour naturally arises from the resource usage analysis discussed in Section 3.2.

The barrier operator has no rules of equivalence to discourage any transforms across the expression boundaries. For a single expression, the quality of accuracy is determined by its round-off error. However, when evaluating multiple expressions there are choices to make in terms of determining the overall accuracy of the system of multiple expressions. The reasons are two-fold [Mar09]. Firstly, perhaps only a user-defined subset of the expressions that compute the final results are influential. Secondly, we may wish to minimize either the $L^1$-, $L^2$-, $L^\infty$-norm, or a geometric mean of the errors, depending on
which one is more relevant. For this reason, SOAP provides the choice to minimize any of these above norms, and is designed to minimize the $L^{\infty}$-norm of all expressions by default, by computing the least upper bound of the following accuracy semantics of two expressions joined by the barrier operator:

$$(x^1, \mu_1^1) \mid (x^2, \mu_2^1) := (x^1, \mu_1^1) \sqcup (x^2, \mu_2^2),$$

where the $\sqcup$ operator on the abstract semantics is introduced in Section 2.3.4. This approach computes the worst-case bound on errors encountered in the evaluation of all individual expressions in the system of multiple expressions.

With the extended accuracy analysis in place, the automatic multi-objective optimization techniques for single expression can be adapted to multiple expressions, by concatenating them into a single expression using the barrier operator.

### 3.4 Implementation

The majority of SOAP, is implemented in Python. For computing errors in real arithmetic, we use exact arithmetic based on rational numbers within the GNU Multiple Precision Arithmetic Library (GMP) library [Gra+91]. In case when exact arithmetic is not possible because of high computational costs, floating-point arithmetic can be used to efficiently and safely bound round-off error values. We also use the MPFR library [Fou+07] for access to floating-point rounding modes and arbitrary precision floating-point computation.

Because of the workload of equivalent expression finding, the underlying algorithm is optimized in many ways. Firstly, for each iteration, the relation finding function $\blacktriangleleft_{k}$ is only applied to newly discovered expressions in the previous iteration, using the algorithm in Figure 3.1. The second optimization is to cache results of function calls such as $\blacktriangleleft_{k}$, Area and Error, since there is a large chance that these results from subexpressions are reused several times, subexpressions are also maximally shared to eliminate duplication in memory. Thirdly, the computation of $\blacktriangleleft_{k}$ is fully multi-threaded.
The resource statistics of operators are provided using FloPoCo [DP11] and Xilinx Synthesis Technology (XST) [Xil13]. Initially, for each combination of an operator, an exponent width between 5 and 16, and a mantissa width ranging from 10 to 113, a total of 2496 distinct implementations are generated using FloPoCo. All of them are optimized to use DSP blocks. They are then synthesized using XST, targeting a Virtex-6 FPGA device (XC6VLX760). Because LUTs are generally more constrained resources than DSP blocks in floating-point computations, only synthesis statistics in LUTs is provided. Finally, an RTL code generation backend can produce synthesizable code from an optimized candidate expression.

3.5 Results

Because Martel’s approach defers selecting optimal options until the end of equivalent expression discovery, we developed a method that could produce exactly the same set of equivalent expressions from the traces computed by Martel, and has the same time complexity. The difference is that we adopted it to generate a Pareto frontier from the discovered expressions, instead of only error bounds. This allows us to compare martel_trace, i.e. our implementation of Martel’s method, against our methods frontier_trace and greedy_trace discussed in Section 3.3. Figure 3.6 optimizes the expression \((a + b)^2\) using the three methods above, all using depth limit 3, and the input ranges are \(a \in [5, 10]\) and \(b \in [0, 0.001]\) [Mar07]. The IEEE 754 single-precision floating-point format with rounding to nearest was used for the evaluation of accuracy and area estimation. The scatter points represent different implementations of the original expression that have been explored and analyzed, and the (overlapping) lines denote the Pareto frontiers. In this example, our methods produce the same Pareto frontier that Martel’s method could discover, while having up to 50% shorter run time. Because we consider an accuracy/area trade-off, we find that we can not only have the most accurate implementation discovered by Martel, but also an option that is only 0.0005% less accurate, but uses 7% fewer LUTs.
We go beyond the optimization of a small expression, by generating results in Figure 3.7 to show that the same technique is applicable to simultaneous optimization of multiple large expressions. The expressions $e_1$ and $e_2$, with input ranges $a \in [1, 2]$, $b \in [10, 20]$, $c \in [10, 200]$ are used as our example:

$$e_1 = (a + a + b) \times (a + b + b) \times (b + b + c) \times (b + c + c) \times (c + c + a) \times (c + a + a),$$

$$e_2 = (1 + b + c) \times (a + 1 + c) \times (a + b + 1).$$

We generated and optimized RTL implementations of $e_1$ and $e_2$ simultaneously using frontier_trace and greedy_trace with the depth limits indicated by the numbers in the legend of Figure 3.7. Note that because the expressions evaluate to large values, the errors are also relatively large. We set the depth limit to 2 and found that greedy_trace executes up to $10 \times$ faster than frontier_trace, while discovering a sizable subset of the Pareto frontier of frontier_trace. Also our methods are significantly faster and more scalable than martel_trace, because of its poor scalability discussed earlier, our computer ran out of 8 GB of memory before we could produce any results. If we normalize the time allowed for each method and compare the performance, we found that greedy_trace with a depth limit 3 takes slightly less time than frontier_trace with a depth limit 2, but produces a generally better Pareto frontier.

The alternative implementations of the original expression provided by the Pareto frontier of greedy_trace can either reduce the LUTs used by approximately 10% when accuracy is not crucial, or can be about 10% more accurate if resource is not our concern. It also enables the ability to choose different trade-off options, such as an implementation that is 7% more accurate and uses 7% fewer LUTs than the original expression.

Furthermore, Figure 3.8 varies the mantissa width of the floating-point format, and presents the Pareto frontier of both $e_1$ and $e_2$ together under optimization. Floating-point formats with mantissa widths ranging from 10 to 112 bits were used to optimize and evaluate the expressions for both accuracy and area usage. It turns out that some implementations originally on the Pareto frontier of Figure 3.7 are no longer desirable, as
by varying the mantissa width, new implementations are both more accurate and less resource demanding.

Besides the large example expressions above, Figure 3.9 and Figure 3.10 are produced by optimizing expressions with real applications under single precision. Figure 3.9 shows the optimization of the Taylor expansion of $\sin(x + y)$, where $x \in [-0.1, 0.1]$ and $y \in [0, 1]$, using greedy_trace with a depth limit 3. The function taylor($f, d$) indicates the Taylor expansion of function $f(x, y)$ at $x = y = 0$ with a maximum degree of $d$. For order 5 we reduced error by more than 60%. Figure 3.10 illustrates the results obtained using the depth limit 3 with the Motzkin polynomial [Dem11] $x^6 + y^4 z^2 + y^2 z^4 - 3x^2 y^2 z^2$, which is known to be difficult to evaluate accurately, especially using inputs $x \in [-0.99, 1]$, $y \in [1, 1.01]$, $z \in [-0.01, 0.01]$.

All these above results are generated with the same type of floating-point operators in each expression. Although in this chapter we do not analyze the number of DSPs used in synthesized circuits, the DSP count increases linearly with the estimated LUT count. In the next chapter we further introduce the estimation of DSP elements used as another objective to optimize.

Because of the scalability problem of the depth limit $k$ mentioned in Section 3.3, $k \leq 3$ for all of our experiments. By setting $k = 4$, the tool does not terminate in reasonable amount of time and saturates the memory (16 GB) of our system. In the following chapters, we propose methods to limit the number of iterations and the number of equivalent expressions discovered to mitigate the lack of scalability of $k$.

Finally, Figure 3.11 demonstrates the accuracy of the area estimation used in our analysis. It compares the actual LUTs necessary with the estimated number of LUTs using our semantics, by synthesizing more than 6000 equivalent expressions derived from $a + b + c$, $(a + 1) \times (b + 1) \times (c + 1)$, $e_1$, and $e_2$ using varying mantissa widths. The dotted line indicates exact area estimation, a scatter points that is close to the line means the area estimation for that particular implementation is accurate. The solid black line represents the linear regression line of all scatter points. On average, our area estimation is a 6.1%
over-approximation of the actual number of LUTs, and the worst case over-approximation is 7.7%.

Figure 3.6. Optimization of $(a + b)^2$.

Figure 3.7. Simultaneous optimization of both $e_1$ and $e_2$.

3.6 Summary

This chapter provides a formal approach to the optimization of arithmetic expressions for both accuracy and resource usage in high-level synthesis. The method proposed
in this chapter and the associated tool, SOAP, encompass three kind of semantics that describe the accumulated roundoff errors, count operators in expressions considering common subexpression elimination, and derive equivalent expressions. For a set of input expressions, the proposed approach works out the respective sets of equivalent expressions in a hierarchical bottom-up fashion, with a windowing depth limit and Pareto selection to help reduce the complexity of equivalent expression discovery. Using SOAP, we improve either the accuracy of our sample expressions or the resource utilization by up to 60%, over the originals under single precision. SOAP enables a high-level synthesis tool to optimize the structure as well as the precision of arithmetic expressions, then to automatically choose an implementation that satisfies accuracy and resource usage constraints.

Because we underpin our approach in formal semantics, it provides the necessary foundation which permits us to extend the method for general numerical program transformation in high-level synthesis. Therefore in Chapter 4, we base ourselves on the methodologies developed in this chapter, and propose a structural approach to program optimization by safely rewriting equivalent structures in numerical programs.
Figure 3.9. The Taylor expansion of $\sin(x + y)$.

Figure 3.10. The Motzkin polynomial $e_m$. 
Figure 3.11. Accuracy of Area Estimation.
Numerical Program Optimization

The previous chapter introduced a new methodology to efficiently restructure arithmetic expressions for the optimized trade-off between two performance metrics, i.e. numerical accuracy when evaluated and area usage in synthesized FPGA implementations. This method however has a substantial limitation when applied to general numerical programs, that is, it can only be applied to straight-line codes without control structures such as branches and loops.

In this chapter, a new general program optimization technique for numerical algorithms is therefore proposed, which analyzes and optimizes if statements as well as while loops in a numerical program. This enables the joint optimization of accuracy and resource usage, as well as the trade-off between both performance metrics. A tool is thus developed to perform source-to-source optimization of numerical programs targeting FPGAs, and generate implementations that trade off resource usage and numerical accuracy.

Similar to the approach proposed in Chapter 3, equivalence rules such as associativity \((a + b) + c \equiv a + (b + c)\), and distributivity \((a + b) \times c \equiv a \times c + b \times c\) are exploited to automatically optimize implementations for the optimal trade-off between resource usage, i.e. the number of LUTs and DSP elements utilized, and accuracy when evaluated using floating-point computations. This process generates a Pareto frontier of optimized numerical programs. For example, with single-precision floating-point format, the tool finds that given an input \(x \in [0, 100]\) and \(y \in [0, 2]\), then the program:

```
if (x < 1) {
  x = (x + y) + 0.1f;
} else {
  x = x + (y + 0.1f);
}
```

is most accurate when the subexpression “\((x+y)+0.1f\)” is rewritten as “\((x+0.1f)+y\)”;
on the other hand, the original program uses fewest resources when subexpressions are shared and the `if` statement is eliminated:

\[ x = x + (y + 0.1f); \]

The structural optimization of general numerical programs is much more complex than that of arithmetic expressions. The reasons are three-fold. First, during program execution, variables are often updated with new values. Our optimization should therefore perform static analysis on the values of variables, and use the result to optimize specifically for the trade-off between accuracy and resource usage. Second, the combinatorial explosion of expression equivalence, is further exacerbated by the expressiveness of a general numerical program. Finally, it is much more difficult to formally define program equivalence, and subsequently, to search efficiently for optimized equivalent programs.

The difficulty in defining program equivalence is due to the fact that two programs can be identical in function, but have distinct syntactic structure because of the expressiveness of a HLL. In fact, one can easily imagine infinitely many ways to rewrite numerical programs, and often these equivalent programs have identical resource usage, latency and accuracy characteristics. In practice, it is desirable to eliminate as much as possible the need for these syntactic rewrites that do not affect our performance metrics, so that the search space of equivalent programs is greatly reduced.

We explored in Section 2.4 of Chapter 2 various intermediate representations (IRs) designed for program transformation, and specifically examined program expression graph (PEG) in closer details, because it fits our requirement to abstract away as much irrelevant syntactic information as possible. However, PEG is not suited for the optimization of numerical accuracy because cycles in graphs are used. Evaluating the numerical accuracy of an IR with a cyclic structure requires analyzing a large proportion of the IR, whereas a tree structure can be reasoned compositionally in a bottom-up hierarchy. By introducing equivalence edges in the graph, the number of elementary cycles in the graph could increase exponentially in the number of equivalences that have been discovered, which further exacerbates the already high computational demand. This has significant implications for program transformation. First, for the above reason, their approach
does not use static analysis to optimize for numerical accuracy, while we wish to reason about accuracy and utilize them to steer optimization. Moreover, the tree structure allows us to easily support partial loop unrolling by simply extending the equivalence relations while avoiding re-evaluation. In contrast, like [Mar09] and [Dam+15], equality saturation is unable to perform partial loop unrolling. In this chapter, a solution to the above limitations of PEGs is therefore proposed, by introducing a new tree-based IR with fixpoint constructs to specifically tackle the problem of program equivalences.

Additionally, as none of the methods in Section 2.5 of Chapter 2 looks at the multiple-objective optimization of numerical programs, this chapter is the first to propose a tool that performs a semantics-directed program transformation, which optimizes not only arithmetic expressions, but also numerical programs. The tool optimizes for the trade-off between numerical accuracy and resource usage when synthesized to FPGAs.

The optimization flow is designed to be flexible, and its implications are three-fold. Firstly, arithmetic computations can be optimized across assignments, if statements and while loops. Secondly, we automatically explore the numerical implications of partial loop unrolling and loop splitting, which can create more opportunity for minimizing round-off errors, hence further increases range of options in the Pareto frontier of trade-offs. Finally, our method naturally subsumes constant propagation, redundant code elimination, and also branch and loop fusions.

The main contributions in this chapter are as follows:

1. Metasemantic intermediate representation (MIR), A new IR of the behaviour of numerical programs. Its structure is designed to be manipulated and analyzed with ease.

2. Semantics-based analyses that reason about not only the resource utilization (number of LUTs and DSP elements), and safe ranges of values and errors for programs, but also potential errors such as overflows and non-termination. This provides the cost functions that we wish to minimize by rewriting programs.
3. A new framework of numerical program transformations is developed based on the methods in Section 3.3 of Chapter 3. It enables the back and forth translation between the program and MIR, which preserves the semantics of the original program in real arithmetic.

4. The updated tool, SOAP, which trades off resource usage and accuracy by providing a safe, semantics-directed and flexible optimization targeting numerical programs for high-level synthesis. Experimental results are presented in Section 4.7.

This chapter is organized as follows. We start by defining our program syntax in Section 4.1. Using the syntax definition, this section gives a detailed formal explanation of the numerical program transformation, which consists of three stages. Sections 4.2 describe how numerical programs can be translated into MIRs. Sections 4.3 and 4.4 respectively discuss how we infer bounds and error bounds on variables and analyze resource usage estimates. Section 4.5 explains how these analyses can be used to efficiently discover equivalent structures in the analyzed MIR. Section 4.6 explains how a chosen MIR can be translated into an optimized numerical program. Then we present the optimization results in Section 4.7 and finally Section 4.8 concludes this chapter.

4.1 Syntax Definition

Before we discuss program transform, we first look at the syntax definition used to write numerical programs. Our program transformation optimizes programs written in a subset of C99. In this section, we formally introduce the syntax that constitutes a subset of C that supports arithmetic and Boolean expressions, conditional branches, as well as while and for loops. Our language allows numerical data types int and float, respectively standing for integer and floating-point types.

We define AExpr, BExpr as the set of arithmetic and Boolean expressions respectively, and Stmt denotes the set of program statements. We then have following simplified
syntax definition for expressions and numerical programs, written in the Backus-Naur form [Knu64]:

\[
\begin{align*}
& a ::= n \mid x \mid -a_1 \mid a_1 \otimes a_2, \\
& b ::= !b_1 \mid a_1 \oplus a_2 \mid b_1 \oslash b_2, \\
& s ::= [t] \mid x = a \mid s_1 s_2 \mid \text{if} (b) \{s_1\} \mid \text{else} \{s_2\} \mid \text{while} (b) \{s\}, \\
& x ::= v, \quad v ::= x \mid y \mid z \mid \ldots, \\
& t ::= \text{int} \mid \text{float}.
\end{align*}
\]

We define \(\otimes\) \(\in\) \{+, -, *, /\}, \(\oplus\) \(\in\) \{<,\leq,>,\geq,=,!=\}, and \(\oslash\) \(\in\) \{\|,&&\} respectively to be the arithmetic, comparison and Boolean operators, \(n\) is a numerical constant of type either \text{int} or \text{float}. The \(v\) terms \(x, y, z \in \text{Var}\) are variables. In the next chapter, the definition of \(x\) is further extended to arrays. Additionally, \(a, a_1, a_2 \in \text{AExpr}\) are arithmetic expressions, and \(b, b_1, b_2\) ranges over Boolean expressions, \(\text{BExpr}\). Program statements \(s, s_1, s_2 \in \text{Stmt}\) comprise assignment statements, sequential statements, \text{if} branches and \text{while} loops. Although \text{for} loop is not explicitly defined in the above syntax definition, it can be trivially expressed using a \text{while} loop. Finally, \(t\) is an optional keyword which can be either \text{int} or \text{float} to specify the type of \(x\) if \(x\) is not previously declared. Note that a term of the form \([d]\) indicates \(d\) is optional.

Furthermore, we introduce the “\#pragma soap begin” and “\#pragma soap end” directives to delimit the code fragment to be optimized. We can also use “\#pragma soap in” and “\#pragma soap out” to provide input ranges and to declare output variables, respectively.

As a simple example, the program in Figure 4.1 computes an approximate value of \(\pi^2a/6\). It has two inputs \(a\), a floating point value between 0 and 1, and \(n\), an integer value between 10 and 20, which determines the number of iterations for the loop, and a return variable \(y\).

Despite the simplicity of our syntax, it includes all the features of a full programming language rather than an arithmetic expression language used in Chapter 3. We will add
#pragma soap begin
#pragma soap in \n      float a = [0.0, 1.0], int n = [10, 20]
#pragma soap out float y
x = 0;
y = 0.0f;
while (x < n) {
x = x + 1;
y = y + a / (x * x);
}
#pragma soap end

Figure 4.1. A simple program, basel, written with our syntax definition.

support for arrays and matrices in Chapter 5, and show that this can be added with few
changes to our method.

4.2 Metasemantic Intermediate Representation

There are infinite number of ways to rewrite numerical C programs, and many of these
rewrites produce programs that have the same resource usage, accuracy and latency
characteristics. For instance, consider the following two pairs of programs, where each	pair are equivalent, but syntactically different, as they carry out the same (and potentially
redundant) computations.

\[
\begin{align*}
\text{(a) } & \quad \text{if } (b) \\
& \quad x = x + 1; \quad x = x + 1; \quad x = 2 \times (x + 1); \\
& \quad y = 2 \times x; \quad y = y \times 2; \quad \text{if } (b) \quad \text{else} \\
& \quad x = x + 3; \quad x = x + 3; \quad x = 2 \times x; \quad x++; \\
\end{align*}
\]

Figure 4.2. Two pairs of programs that are equivalent but syntactically different.

In practice, it is desirable to eliminate as much as possible the need for these syntactic
rewrites that do not affect our performance metrics, e.g. the numerical accuracy and
resource usage of synthesized circuits. It is therefore desirable to perform transformations
on a DAG representation of the program, rather than on the program text directly. This
section introduces a new DAG-based IR, which we call the *metasemantic intermediate representation* (MIR). It expresses how each program variable is updated while preserving the control- and data-flow of the original program, but abstracts away the order in which the updates occur, and ignores any temporary variables that are not marked as program outputs.

As an example, the two equivalent programs $P_1$ and $P'_1$ in Figure 4.2 can be automatically translated into an identical MIR:

$$
\begin{align*}
&\begin{array}{c}
x + 3 + y \times 2 \\
x \leftarrow 1
\end{array}.
\end{align*}
$$

(4.2)

MIRs also abstract the control structure (i.e., *if* branches and *while* loops) of a program, preserving only the computations that lead to the outputs. For instance, by using the ternary conditional operator “?” from C, programs with conditionals such as $P_2$ and $P'_2$ in Figure 4.2 has the following MIR form:

$$
\begin{align*}
&\begin{array}{c}
x \leftarrow b \times 2 + x \leftarrow 1
\end{array}.
\end{align*}
$$

(4.3)

This representation is useful to us, because a single MIR is able to capture a class of syntactically-distinct programs, all of which have the same resource usage, accuracy, and latency characteristics. By searching for transformations on MIRs, we drastically reduce the size of our search space. Note that expressions in the MIR can share common structures; this is useful for modeling the sharing of common subexpressions and makes the search for optimizations much more efficient.

The first step of our approach is to analyze the program return value into a MIR. This procedure is called *metasemantic analysis* (MA). The MA abstracts away irrelevant information, and preserves the essence of program execution. Details such as temporary variables and the ordering of program statements are discarded, whereas the abstraction
still retains data-flow dependencies and keeps only computations that contribute to the final results.

We work with the MIR as an abstraction of the program because the discovery of equivalent structures can be much simplified. For instance, the program “\(x = 1; \ y = 2;\)” is the same as “\(y = 2; \ x = 1;\)” because interleaving of non-dependent statements does not change program semantics. If we were to base our transformations on the program syntax, we will need to enable this kind of equivalence relation even though it has zero impact on our optimization with respect to resource usage and accuracy. A simpler IR means that we can explore a much smaller search space.

Our method analyzes a program by recursively dividing the program into smaller parts, where each part can be separately analyzed into a MIR and composed together to form a single MIR. A MIR is a mathematical object that associates each program variable with a semantic expression. A semantic expression is an arithmetic expression, but with additional syntactic features to support if statements and while loops. We represent semantic expressions with DAGs that share common structures and define SemExpr as the set of semantic expressions, and MIR as the set of MIRs. Because a MIR pairs a variable with an expression, we can view it as a function Var \(\rightarrow\) SemExpr that maps a variable into a semantic expression. For instance, \(\mu(x)\) returns the associated expression of the variable \(x \in \text{Var}\) for the MIR \(\mu \in \text{MIR}\). For each variable, its semantic expression in itself provides a complete picture of how computations can lead to the resulting value of the variable. In the rest of this section, we progressively explain how each type of program statement defined in (4.1) is analyzed into a MIR.

Similar to arithmetic expressions, which can be written in a linear form (e.g. \(a + b\)), or in a tree structure (e.g. \(\frac{a}{b}\)), MIRs can also be expressed in both. The former is more concise, whereas the latter explicitly shares common structures.
4.2.1 Assignment statements

An assignment statement is in the form of “\(x = e_i\)”, where \(x \in \text{Var}\) is a program variable and \(e \in \text{AExpr}\) is an arithmetic expression. The metasemantic analysis of it produces a MIR as follows:

\[
\begin{cases}
    e & \text{if } y = x \\
    y & \text{otherwise}
\end{cases}
\] \(y \in \text{Var}\). 

(4.4)

The MIR in (4.4) signifies for a variable \(y \in \text{Var}\), if \(y = x\), then we assign the expression \(e\) to the variable \(x\). In graph form, \(e\) is a semantic expression represented with a DAG. The DAG shares all common subexpressions in \(e\). For instance, an expression written as \((x + 1) \times (x + 1)\) shares the subexpression node \(x + 1\) by reusing the node in the DAG. For each other program variable \(y \in \text{Var}\), where \(y \neq x\), \(y\) is associated with a semantic expression \(y \in \text{SemExpr}\), representing that the MIR does not alter the value of all program variables except \(x\), because only \(x\) is updated in the statement.

For example, we consider a program with two variables \(x\) and \(y\). Analyzing the statement “\(y = x \times 2;\)” produces the following MIR graph, it is notable that the variable \(x\) is shared between two semantic expressions:

\[
\begin{array}{c}
\xrightarrow{x} \times \\
\xrightarrow{y} \times \\
\end{array}
\] \(\times 2\). 

(4.5)

4.2.2 Sequential statements

A sequential statement, “\(s_1s_2\)” is formed by joining together \(s_1\) and \(s_2\), where \(s_1, s_2 \in \text{Stmt}\) are statements. It signifies that \(s_1\) and \(s_2\) are executed in sequence. Therefore, it is necessary to append the effect of executing \(s_2\) to that of \(s_1\), to arrive at the full MIR of “\(s_1s_2\)” This concept can be realized by defining a new operator \(\ast\), the composition operator, such that the MIR of “\(s_1s_2\)” is equal to \(\mu_2 \ast \mu_1\), where \(\mu_1\) and \(\mu_2\) are the MIRs of \(s_1\) and \(s_2\) respectively. The resulting MIR of \(\mu_2 \ast \mu_1\) is constructed by substituting, for every expression \(e \in \text{SemExpr}\) in \(\mu_2\), each variable \(x\) in \(e\) with \(\mu_1(x)\), which is the
associated expression of $x$ in $\mu_1$. It is noteworthy that $\mu_1$ is always computed before $\mu_2$ when evaluated, as there is a data-dependence from $\mu_2$ to $\mu_1$. Furthermore, the operator allows the format $e * \mu$, where $e \in \text{SemExpr}$ is called the target expression and $\mu \in \text{MIR}$ is the source MIR, to mean the variables in $e$ is substituted with $\mu$ using the composition strategy above.

We illustrate this by finding the MIR of a simple example program, “$x=x+1; y=x*2;$”. Using the MIR of assignments, the MIRs of the respective assignment statements can be derived, as shown in (4.6).

\[
\begin{array}{c}
\text{x} \mapsto x \\
\text{y} \mapsto x \\
\text{1} \mapsto 2
\end{array} 
\begin{array}{c}
\text{x} \mapsto x \\
\text{+} \mapsto x \\
\text{1} \mapsto y
\end{array} 
\begin{array}{c}
\text{x} \mapsto +
\end{array} 
\begin{array}{c}
\text{2} \mapsto x
\end{array}
\]

(4.6)

By substituting the variables with corresponding expressions, we arrive at the MIRs as shown in (4.7) below. This MIR simplification step is always carried out when possible.

\[
\begin{array}{c}
\text{x} \mapsto +
\end{array} 
\begin{array}{c}
\text{2} \mapsto x
\end{array}
\]

(4.7)

### 4.2.3 Conditional Branches

Conditional branches, or if statements, are represented with “if ($b$) { $s_1$ } else { $s_2$ }”. Here $b \in \text{BExpr}$ is a Boolean expression, and $s_1, s_2 \in \text{Stmt}$ are respectively the true- and false-branches. Our analysis of if statements is slightly more complex, as we start to consider control flows. The analysis is carried out in two steps. The first step is to compute recursively, the MIRs $\mu_1, \mu_2 \in \text{MIR}$ of the respective true- and false-branches, namely, $s_1$ and $s_2$. We introduce the conditional node “?” which is derived from C syntax, to signify conditional branches in expressions. The left-most, middle and right-most children of this node are respectively the Boolean expression, the true- and false-expressions. Then the second step is to compute a new MIR, where each program variable $x \in \text{Var}$ is associated
with a conditional node with three children, the Boolean expression \( b, \mu_1(x) \) and \( \mu_2(x) \).

The final MIR is therefore:

\[
[x \mapsto b ? \mu_1(x) : \mu_2(x)]_{x \in \text{Var}}.
\] (4.8)

As an example, we consider the program “\( \text{if } (x < 0) \ y = x \times 2; \)”, where the set of program variables is \( \{x, y\} \). Its MIR in graph form is therefore:

\[
\begin{bmatrix}
\frac{x \mapsto ?}{x} & \frac{y \mapsto ?}{y} \\
\frac{x}{+x} & \frac{0}{x} & \frac{2}{y}
\end{bmatrix}.
\] (4.9)

Because both true- and false-expressions of \( x \) are the same, regardless of the truth value of \( x < 0 \), the two expressions evaluate to the same value. In our analysis we can immediately simplify the expression of \( x \), and the resulting MIR is semantically equivalent to the original:

\[
\begin{bmatrix}
\frac{x \mapsto x}{x} & \frac{y \mapsto ?}{y} \\
\frac{x}{0} & \frac{2}{y}
\end{bmatrix}.
\] (4.10)

The traditional approach of program abstraction uses CDFGs [Nam+04]. CDFGs preserves the ordering of sequential statements, uses a one-to-one mapping from assignment statements to assignment nodes, uses storage nodes to store the result of assignments, i.e. it allows nodes to act as a memory to store values, and finally, contains cycles in graphs to represent program loops. In contrast, our MIRs, from our analysis point of view, use no local storage of temporary values, and discard unnecessary intermediate statements. Most importantly, we treat control structures as operators in expressions, in the same way as arithmetic computations, and we will show later in Section 4.5 this enables greater flexibility in program transformation based on MIRs. In comparison with CDFGs, these above facts make MIRs a more suitable candidate for exploring the search space of program transformations.
4.2.4 Loops

A possible way to represent a while loop, \texttt{while (b) \{ s \}}", where \( b \in \texttt{BExpr} \) and \( s \in \texttt{Stmt} \), is to effectively analyze it as a fully unrolled loop. Unfortunately the resulting semantic expressions have an infinite depth, which cannot be represented fully in a data structure. Because these expressions have recurring patterns, we therefore introduce a new operator for semantic expressions, \texttt{fix}", which we call the \texttt{fixpoint} operator, to capture this pattern. For each variable \( x \) updated in loop, we further use the fixpoint expression \texttt{fix}(f) to represent the while loop, where the function \( f : \texttt{SemExpr} \rightarrow \texttt{SemExpr} \), defined as \( f(e_x) = b ? e_x \ast \mu_s : x \), represents the computation of one iteration of the loop, and \( \mu_s \) is the MIR of the loop body. Finally, the fixpoint expression of a while loop can be written more succinctly using \texttt{lambda-expression} [Bar13], \( \texttt{fix} (\lambda e_x \cdot b ? e_x \ast \mu_s : x) \).

In graph form, for simplicity the fixpoint node admits three child nodes, namely, the Boolean expression \( b \), the loop body represented by a MIR, and the loop exit variable. The loop body MIR can be obtained with our MA of the loop body, and the loop exit variable denotes which variable we use on loop exit as the evaluated result of the fixpoint expression. We let \( \mu_s \) to be the MIR of the loop body \( s \), and derive the MIR of the while loop, by computing the fixpoint expression for each variable:

\[
\begin{align*}
\times & \mapsto \begin{cases} 
\text{fix} \quad \text{if } x \in \text{var} (\mu_s) \\
\times & \text{otherwise}
\end{cases} \quad \times \in \text{Var}
\end{align*}
\]

(4.11)

Here, \( \text{var} (\mu_s) \) computes the set of variables that is assigned in the loop body \( \mu_s \). If a program variable \( x \) is in the set \( \text{var} (\mu_s) \), it is paired with its fixpoint expression \( \text{fix} \); otherwise \( x \) is not updated in the loop, and the loop has no effect on its value, therefore it is paired with an expression \( x \). Finally, the constructed MIR maximally shares common expressions across nested MIRs.
As a simple example, consider the program in Figure 4.3a. The MIR of this example program allows a nested MIR to reference and reuse an expression from the outer MIR as demonstrated in Figure 4.3b, where:

\[ \mu = [x \mapsto x + 1], \text{ and } \quad b = x < y. \]  

\[ y = y + 1; \]
\[ \textbf{while} \ (x < y) \ { \{ \}
\]
\[ x = x + 1; \}
\]
\[ (a) \text{The program.} \]

\[ (b) \text{The MIR.} \]

\[ \text{Figure 4.3.} \quad \text{A simple program which exhibits common subexpressions reuse across nested MIRs.} \]

4.2.5 Example analysis

To illustrate all these above translation process in conjunction, we can perform the MA on the program \texttt{basel} in Figure 4.1. Here, we translate it into the following MIR, and the semantic expression for \( x \) is omitted for simplicity:

\[ \begin{bmatrix}
  x & \mapsto \ldots \\
  y & \mapsto \\
  \text{fix} & \\
  x & \mapsto \ldots \\
  y & \mapsto \\
  a & \mapsto a \\
  n & \mapsto n \\
\end{bmatrix}. \]  

\[ \begin{bmatrix}
  x & \mapsto \ldots \\
  y & \mapsto \\
  \text{fix} & \\
  x & \mapsto \ldots \\
  y & \mapsto \\
  a & \mapsto a \\
  n & \mapsto n \\
\end{bmatrix}. \]  

\[ y = y + 1; \]
\[ \textbf{while} \ (x < y) \ { \{ \}
\]
\[ x = x + 1; \}
\]
\[ (a) \text{The program.} \]

\[ (b) \text{The MIR.} \]

4.3 Accuracy Analysis

Because we make use of accuracy analysis to navigate the Pareto optimization of program candidates, we start by providing an overview of how the accuracy of simple arithmetic expressions are analyzed with the SOAP framework, and since it only allows arithmetic
expressions with simple operators \{+, -, \times\}, we explain how it can be extend fully to analyze MIRs and semantic expressions.

In a typical program execution, values of variables, typically integers \(\mathbb{Z}\) and floating-point values \(\mathbb{F}\), are modified according to the effect of the program statements, and they are propagated through arithmetic operators from the beginning to the end of the program. In Section 3.1 of Chapter 3, alternative semantics are proposed to instead propagate ranges of values together with the associated round-off error bounds, i.e. the value-error bound \((v^\pi, e^\pi) \in \mathbb{E}^\pi\), in order to analyze the accuracy of floating-point numerical programs. In this section, this technique is further generalized to numerical programs.

Initially, we formalize the analyzed program values of a program as an abstract program state using the domain \(\Sigma_{\mathbb{E}^\pi} = \text{Var} \to \mathbb{E}^\pi\), and a \(\sigma^\pi \in \Sigma_{\mathbb{E}^\pi}\) maps each variable \(x\) to their associated value-error bound \(\sigma^\pi(x) \in \mathbb{E}^\pi\). The purpose of this abstract program state is to provide information on the bounds on values and errors for all variables in any particular location of program execution. For instance, we assume an abstract state \(\sigma^\pi_0 \in \Sigma_{\mathbb{E}^\pi}\) which provides the input values of \(a\) and \(b\) to a program, where:

\[
\sigma^\pi_0 = \left[ a \mapsto ([0, 1], e^\circ_0), b \mapsto ([1, 2], e^\circ_0) \right]. \quad (4.14)
\]

This means that initially \(a\) and \(b\) are floating-point values bounded by \([0, 1]\) and \([1, 2]\) respectively, and the error interval \(e^\circ_0 = [0, 0]\) denotes the absence of round-off errors.

In Section 3.1 of Chapter 3 we introduced the function \(\text{Error} : A\text{Expr} \to \mathbb{E}^\pi\) to evaluate the bounds on the result and its round-off error of computing an expression, but we now extend it to explicitly make use of the input ranges of variables. Here we introduce a new function \(\text{Es} : \text{SemExpr} \to \Sigma_{\mathbb{E}^\pi} \to \mathbb{E}^\pi\) which further accepts initial bounds on the values and errors of variables. The formula \(\text{Es}[e] \sigma^\pi\), where \(e \in \text{SemExpr}\) and \(\sigma^\pi \in \Sigma_{\mathbb{E}^\pi}\), is used to denote the accuracy analysis of the expression \(e\) with the input state \(\sigma^\pi\).
Then in single-precision, the error analysis of \( a + b \), given the initial bounds \( \sigma^0 \) in (4.14), produces the following result:

\[
E_s [a + b] \sigma^0 = \left( [1, 3], \left[ -1.19209304 \times 10^{-7}, 1.19209304 \times 10^{-7} \right] \right). \tag{4.15}
\]

This means that the result of this computation is in the range of \( v^f = [1, 3] \), and the round-off error induced by this computation is bounded by the interval \( e^f \).

The method outlined in Chapter 3 to analyze the accuracy of arithmetic expressions supports only addition, subtraction, multiplication and division. In this section, we explain in detail how it is extended to support MIRs, and our additional operators in semantic expressions, i.e. the composition, ternary conditional and fixpoint operators.

### 4.3.1 MIR

In the same way that an expression can be analyzed for its accuracy, a MIR, which is a mapping of variables to semantic expressions, can be analyzed by performing the \( E_s \) analysis for each of its expressions. For instance, the accuracy of a MIR:

\[
\mu_0 = [a \mapsto a + b, b \mapsto a \times 0.5],
\]

with an input state:

\[
\sigma^0 = [a \mapsto ([0, 1], [0, 0]), b \mapsto ([1, 2], [0, 0])], \tag{4.17}
\]

can be analyzed as follows.

First we analyze the individual expressions \( \mu_0(a) = a + b \) and \( \mu_0(b) = a \times 0.5 \), which produce respectively the results below:

\[
\left( v^a, e^a \right) = \left( [1, 3], \left[ -1.19209304 \times 10^{-7}, 1.19209304 \times 10^{-7} \right] \right), \tag{4.18}
\]
\[
\left( v^b, e^b \right) = \left( [0, 0.5], \left[ -2.98023259 \times 10^{-8}, 2.98023259 \times 10^{-8} \right] \right). \tag{4.19}
\]
Then the analyzed results are collected into an abstract state assigning the value-error bounds to their corresponding variables, that is:

\[
\begin{align*}
\{a \mapsto (v_a^L, e_a^L), b \mapsto (v_b^L, e_b^L)\}.
\end{align*}
\]  

(4.20)

To generalize, we can formally define a function, \( E_m[\mu] \sigma^\sharp \), to perform the above analysis, which takes as inputs the MIR \( \mu \in \text{MIR} \) and an abstract input state \( \sigma^\sharp \in \Sigma_{\sharp \text{E}} \). It computes a new state \( \sigma' \), where for each variable \( x \in \text{Var} \), \( \sigma'(x) \) is the analyzed value-error range of the expression \( \mu(x) \). The error analysis of a MIR is therefore:

\[
E_m[\mu] \sigma^\sharp = \{x \mapsto E_s[\mu(x)] \sigma'^\sharp\}_{x \in \text{var}(\mu)}.
\]  

(4.21)

The notation \( \{x \mapsto E_s[\mu(x)] \sigma'^\sharp\}_{x \in \text{var}(\mu)} \) means that the mapping is constructed by collecting for each variable \( x \in \text{var}(\mu) \), the pairing of \( x \) with the analyzed value-error bound of the semantic expression \( \mu(x) \).

### 4.3.2 Composition Operator

The analysis of an expression \( e \ast \mu \), where \( e \in \text{SemExpr} \) and \( \mu \in \text{MIR} \) is carried out in two steps. Initially, given an input state \( \sigma^\sharp \), \( \mu \) is analyzed using (4.21), and we write \( \sigma'^\sharp \) as the analyzed state. Then the expression \( e \) is analyzed for its accuracy as usual, using \( \sigma'^\sharp \) as the input state. Equivalently, this procedure can be defined as:

\[
E_s[e \ast \mu] \sigma^\sharp = E_s[e] \left( E_m[\mu] \sigma^\sharp \right).
\]  

(4.22)

### 4.3.3 Ternary Conditional Operator

A conditional expression is written as \( b?e_1:e_2 \), where \( b \in \text{BExpr} \) and \( e_1, e_2 \in \text{SemExpr} \). The truth value of Boolean expression \( b \) determines whether \( e_1 \) or \( e_2 \) is evaluated to be the resulting value of the expression. Correspondingly, in our accuracy analysis, we impose a constraint defined by the Boolean expression \( b \) on the value ranges of input variables,
such that \( e_1 \) is evaluated with the ranges of values satisfying the constraint, while \( e_2 \) is computed with ranges that violate the constraint.

For example, we analyze an expression \((a < 0) \ ? \ (a - 0.1) : a\) in single-precision. Initially, we assume the program state consists of a variable \( a \), which is a floating-point value that has no associated round-off error and is bounded by \([-1, 10]\), that is:

\[
\sigma_0^\delta = [a \mapsto ([−1, 10], [0, 0])].
\] (4.23)

We consider two cases, when the condition \( a < 0 \) is respectively true and false. For \( a < 0 \) to be true, \( a \) must be in the range of \( v_0^\delta = [−1, 0^-] \), where \( 0^- \) is the greatest single-precision floating-point value less than 0, because \( a \) must be strictly smaller than 0. Then we restrict the range of \( a \) to \( v_0^\delta \), so \( a - 0.1 \) is analyzed to be \((v_1^\delta, e_1^\delta)\), where:

\[
(v_1^\delta, e_1^\delta) = ([-1.10000002, -0.100000001], [-5.81145372 \times 10^{-8}, 6.10947666 \times 10^{-8}]).
\] (4.24)

Similarly, when \( a < 0 \) is false, we restrict the bound on \( a \) with \( v_{¬b}^\delta = [0, 10] \) and analysis of the expression \( a \) simply gives \( v_2^\delta = [0, 10] \) and no round-off error, \( e_2^\delta = [0, 0] \). Finally, the analyzed value and error ranges for the expression can be obtained by joining these two cases together, by respectively evaluating \( v_1^\delta \sqcup v_2^\delta \), which produces \( v^\delta = [-1.10000002, 10] \), and the error bound \( e_1^\delta \sqcup e_2^\delta \). The final result is therefore:

\[
E_s \left[(a < 0) \ ? \ (a - 0.1) : a\right] \sigma_0^\delta
= ([-1.10000002, 10], [-5.81145372e - 08, 6.10947666e - 08]).
\] (4.25)

Our analysis is based on interval arithmetic which is very efficient but it sacrifices accuracy by computing an over-approximation of the exact results. For instance, the above analysis cannot capture the fact that all evaluated result \( v \) satisfies either \(-1.1 \leq v < -0.1\) or \( 0 \leq v \leq 10 \). The majority of such information losses occur because IA computes loose bounds on the analyzed floating-point results. In contrast, joining two error bounds generally produces a precise bound, because error bounds are often much less correlated, and they often overlap as there is a high chance that there exists an arithmetic
computation which produces an exact floating-point outcome. In Section 4.7, empirical results show that despite the accuracy analysis potentially produces loose bounds, it can still be used effectively as an indicator of the round-off errors in actual executions.

We now provide a formal definition of the above example analysis. We use the notation \( \sigma^\sharp|_b \) and \( \sigma^\sharp|_{\neg b} \), where \( \sigma^\sharp \in \Sigma_E^\sharp \) is the program state, and \( b \in \text{BExpr} \) is the Boolean expression, to respectively mean the program state \( \sigma^\sharp \) is constrained by either \( b \) being true or false. Therefore, the following formula is used to perform the accuracy analysis on a conditional expression:

\[
E_s \left[ \begin{array}{c} b \\ \sigma^\sharp \end{array} \right] \quad \sigma^\sharp = \left( E_s[e_1|\sigma^\sharp|_b] \sqcup \left( E_s[e_2|\sigma^\sharp|_{\neg b} \right) \right). \tag{4.26}
\]

### 4.3.4 Fixpoint Operator

An expression with a fixpoint operator, \( \text{fix}_b \), has three child nodes, the Boolean expression \( b \in \text{BExpr} \), the loop body represented with a MIR \( \mu_s \in \text{MIR} \), and the return variable \( x \in \text{Var} \). Similar to executing a \textbf{while} loop, evaluating the expression is to iteratively evaluate \( b \) for its truth value, if \( b \) is true, then the loop MIR \( \mu_s \) is used to update the program state for the next iteration and we repeat the process and iterate until \( b \) is evaluated to false.

Before we explain how a fixpoint expression can be analyzed for its accuracy, we introduce the concept of \textit{loop invariant} (LI). In our context, a LI of a \textbf{while} loop is a set of bounds on loop variables that holds invariantly on entry to each loop iteration. In Section 2.3 of Chapter 2, we explain how the LI of a simple program loop can be computed, here we further extend this concept to general programs expressed in MIRs.

For instance, we consider the \textit{basel} example in Figure 4.1. If our input is \( n = 10 \), then the LI on the variable \( x \) is that its value is an integer, and is bounded by \([0, 9]\) on loop entry, whereas on loop exit, \( x \) is always equal to 10. The reason for inferring the LI is as follows. Since we optimize the fixpoint expression’s child nodes in a bottom-up hierarchy,
the optimization of \( \mu_s \) precedes \( \mu_s \) itself. Hence, we use the LI as the input state to optimize \( \mu_s \), as the LI encompasses all possible program states the loop body \( \mu_s \) will encounter when executed.

Our accuracy analysis of \( \mu_s \) follows the above pattern. Initially, we start with an input state \( \sigma_0^\sharp \). In the first iteration \( k = 0 \), \( \sigma_0^\sharp = \sigma_0^\sharp \) is split into two disjoint parts, namely, \( \sigma_0^\sharp |_b \) and \( \sigma_0^\sharp |_{\neg b} \), they respectively satisfy and violate the Boolean constraint \( b \). The state \( \sigma_0^\sharp |_b \) represents all possible program states that enters the loop \( \mu_s \), so \( \sigma_1^\sharp = \mathcal{E}_m [\mu_s] \sigma_0^\sharp |_b \) captures all possible program states after the loop body. This procedure is repeated for iterations \( k = 1, 2, 3, \ldots \), until a certain iteration \( n \), where \( \sigma_n^\sharp = \sigma_{n-1}^\sharp \). Hence, we can obtain the LI by computing \( \sigma_0^\sharp |_b \sqcup \sigma_1^\sharp |_b \sqcup \cdots \sqcup \sigma_n^\sharp |_b \), and the loop exit states with its counterpart, i.e. \( \sigma_0^\sharp |_{\neg b} \sqcup \sigma_1^\sharp |_{\neg b} \sqcup \cdots \sqcup \sigma_n^\sharp |_{\neg b} \). Here, the loop exit state collects all possible program states on loop exit. The meaning of \( \sqcup \) operator on states is similar to joining intervals and value-error bounds, which is defined to join the two value-error bounds in respective states for each variable, which is defined as follows, where \( \sigma_a^\sharp, \sigma_b^\sharp \in \Sigma_{\mathcal{E}_g^\sharp} \):

\[
\sigma_a^\sharp \sqcup \sigma_b^\sharp = [x \mapsto \sigma_a^\sharp (x) \sqcup \sigma_b^\sharp (x)]_{x \in \text{Var}}.
\] (4.27)

Alternatively, we can compute the LI as the LFP of \( g : \Sigma_{\mathcal{E}_g^\sharp} \rightarrow \Sigma_{\mathcal{E}_g^\sharp} \), where:

\[
g(y) = \mathcal{E}_m [\mu_s] \left( y \sqcup \sigma^\sharp \right)_b. \] (4.28)

This LFP above can be computed with the algorithm in Figure 4.4. The value \( \bot \) indicates an empty or unreachable state, and for any state \( \sigma^\sharp \in \Sigma_{\mathcal{E}_g^\sharp} \), we have \( \bot \sqcup \sigma^\sharp = \sigma^\sharp \). The return values \( \sigma^\sharp_{LI} \) and \( \sigma^\sharp_{LE} \) are respectively the LI and the loop exit state \( \mathcal{E}_s \left[ b \mu_s \right] \sigma^\sharp \).

In the rest of this chapter, we use \( \mathcal{E}_s^{\mu_s} \left[ b \mu_s \right] \sigma^\sharp \) to signify the result \( \sigma^\sharp_{LI} \).

Our method extends the iterative method we have previously explained in Section 2.3 of Chapter 2, by not only evaluating the LI, but also the loop exit state. Because this iterative process may not terminate for non-terminating loops, we introduce a parameter \( \text{max}_\text{iter} \)
function \textsc{FixpointAccuracyAnalysis}(\sigma^\text{\text{fix}}_{b \xrightarrow{\mu_s \times} \sigma^s})

\sigma^s_0 \leftarrow \sigma^s; \sigma^s_{\text{LI}} \leftarrow \perp; \sigma^s_{\text{LE}} \leftarrow \perp; k \leftarrow 0

\text{loop}
\begin{align*}
\sigma^s_{tt} & \leftarrow \sigma^s_k |_b \\
\sigma^s_{ff} & \leftarrow \sigma^s_k |_{\neg b} \\
\sigma^s_{\text{LI}} & \leftarrow \sigma^s_{\text{LI}} \sqcup \sigma^s_{tt} \\
\sigma^s_{\text{LE}} & \leftarrow \sigma^s_{\text{LE}} \sqcup \sigma^s_{ff} \\
\sigma^s_{k+1} & \leftarrow \mathcal{E}_{m}[\mu_s] \sigma^s_k
\end{align*}
\text{if } \sigma^s_{k+1} = \sigma^s_k \lor k \geq \text{max}_{\text{iter}} \text{ then}
\begin{align*}
\text{return } \sigma^s_{\text{LI}}, \sigma^s_{\text{LE}}
\end{align*}
\text{end if}

k \leftarrow k + 1
\text{end loop}
\text{end function}

Figure 4.4. The accuracy analysis of a fixpoint expression.

to limit the number of iterations; if this limit is reached, the tool will produce a warning to indicate that the analysis may never terminate and thus it may be inaccurate. Widening operators [CC04] are additionally used to accelerate the fixpoint computation.

4.4 Resource Usage Analysis

In this section we give a detailed explanation of how resources in MIRs can be shared and how to analyze the resource utilization of MIRs.

Expressions can have common subexpressions, and eliminating them reduces resource usage. We identify and eliminate common subexpressions when we construct DAGs from programs. Resource statistics can be estimated by accumulating LUT and DSP counts of each operator in the DAG. However, we can further merge multiple nodes into one to reduce the estimated resource usage of generated code. Conventional compiler optimizations [Kuc+] such as branch and loop fusion, as well as redundant code elimination can be applied in an equivalent fashion to our MIRs to share conditional and fixpoint operators. This process further reintroduces control structures to MIRs, so that the code generation stage can make use of the result to synthesize code without redundant control structures.
4.4.1 Sharing conditional expressions

In its essence, the sharing of the conditional operator is equivalent to branch fusion. For example, we consider the MIR of the program in Figure 4.5a. The MA of it produces the MIR in Figure 4.5b.

```c
if (x < 0) {
    x = 1;
    y = 2;
} else {
    x = 3;
    y = 4;
}
```

(a) The program.

**Figure 4.5.** The sharing of conditional expressions in a simple program.

Because we compute an abstraction of the program, the MIR does not keep the structure of the *if* statement to allow them to be optimized separately, as doing this would allow our optimization to produce more accurate implementations. The resulting MIR of the program consists of two conditional expressions as shown in Figure 4.5b. Because of this, after optimization, the MIR may has duplicate control paths. To resolve this, we introduce new kinds of nodes, as shown in Figure 4.5c, to “bundle up” more than one conditional expressions, when they all have the same Boolean expression.

4.4.2 Resource sharing in composition expressions

Composition expressions can be fused in an analogous manner. Any MIRs $\mu_1$ and $\mu_2$ can be merged to form a new MIR if there are no conflicts in the variable-expression pairing, *i.e.* for any variable $x$ that is assigned an expression in both $\mu_1$ and $\mu_2$, $\mu_1(x)$ is equal to $\mu_2(x)$. A “bundle” of expressions can also be created to denote the sharing of the composition operator. For example, a simple MIR in Figure 4.6a can be restructured by fusing both variables $x$ and $y$. 

Resource sharing in fixpoint expressions

For fixpoint expressions that represent \texttt{while} loops, discovering resource sharing opportunities is a little more complex. We fuse fixpoint expressions in an analogous fashion as described above for conditional expressions, and it is related to the conventional loop fusion compiler optimization. For instance, the program in Figure 4.7a has the MIR in Figure 4.7b, where:

\[
\begin{align*}
    b &= x > y, \\
    \mu_1 &= [x \mapsto x/2], \text{ and } \\
    \mu_2 &= [x \mapsto x/2, z \mapsto z + x].
\end{align*}
\]  

During program optimization, the two fixpoint expressions are optimized individually, because loop splitting could enhance the accuracy of both \(x\) and \(z\). After this, in the resulting MIR, fixpoint expressions may or may not share computations. The fixpoint expressions can often be fused together to save resources in the conditional expressions and the control logic that are formerly duplicated, if they share common computations that can be merged without conflict. For instance, Figure 4.7c shows how two expressions can be fused into one fixpoint expression.

\begin{verbatim}
while (x > y) {
    z = z + x;
    x = x / 2;
}
\end{verbatim}
However in some cases the above transformations could create cyclic graphs. To illustrate this, consider the following conditional expression fusion example in Figure 4.8a. By fusing both conditionals $a$ and $b$, a cycle is created as in Figure 4.8b.

![Diagram showing before and after fusion of conditionals](https://example.com/diagram.png)

(a) Before fusion, the MIR is acyclic.

(b) Fusing the two conditional operators $a$ and $b$ creates a cycle.

**Figure 4.8.** Example branch fusion creates a cycle.

Not only does conditional expression fusion create dependency cycles, all of these above transformations when performed in conjunction also have the potential to generate cycles and there are potentially multiple ways of restructuring and combining fixpoint expressions, such cycles in MIRs cannot be directly translated back into the source syntax. In our example, only one of either $a$ or $b$ can be fused without generating a dependency cycle in the MIR. There is more than one way of restructuring an MIR, and each way could produce a different MIR that generates a different program. It is therefore necessary to be specific about the strategy to make this procedure deterministic. Our method explores fusion opportunities with a depth-first traversal of the MIR. While traversing nodes, we attempt to form the maximal sharing for the current node; when a cycle is created, the algorithm backtracks to the previous acyclic version. This process is deterministic and ensures the order of resource sharing.

**Resource counting**

Finally, after applying the resource sharing transformations outlined above, we count the number of LUTs and DSPs, by accumulating the statistics for each operator instance in the MIR, in the same way described in Section 3.2 of Chapter 3. For this resource estimation, we gather statistics of individual operators by synthesizing each arithmetic operator (addition, subtraction, multiplication, division and comparison), for single-
and double-precision floating-point, using Altera’s floating-point megafucntions [Alt13]. Similarly, the resource usage of integer arithmetic, conditional and fixpoint operators are synthesized and estimated from multiplexers, and the composition operator uses no resources because it does not perform computations.

4.5 Equivalent Structure Analysis

The next step is to use the analyses of accuracy and resource usage of equivalent structures in MIRs to efficiently discover optimized equivalent MIRs. In this section, equivalent relations from Section 3.3 of Chapter 3 are extended for control-flow structures, and we guide this process efficiently with our analyses of accuracy and resource usage.

We start by taking one step further from simple arithmetic equivalence relations such as associativity, commutativity and distributivity described in Section 3.3 of Chapter 3. We define additional equivalent relations for composition, conditional and fixpoint expressions to fully enable the equivalence transformations of MIRs. Then we go on to improve the methods described in SOAP to explore more equivalent structures, but still in an efficient and scalable way.

4.5.1 Equivalence Relations

In Chapter 3, the SOAP framework formally defines a set of equivalent relations:

\[ \equiv \subset AExpr \times AExpr, \]  \hspace{1cm} (4.30)

for discovering equivalent arithmetic expressions. This equivalent relation \( \equiv \) consists of arithmetic equivalence rules such as associativity, commutativity and distributivity, as well as reduction rules that propagate constants and simplify expressions. We now expand it by making \( \equiv \) a subset of \( M \times M \):

\[ \equiv \subset M \times M, \quad \text{where } M = \text{SemExpr} \cup \text{MIR}. \]  \hspace{1cm} (4.31)
by defining additional rules for \( \equiv \), which relates equivalent semantic expressions that
make use of the additional operators introduced in this chapter.

For the following rules, we assume \( b, b_1, b_2 \in \text{BExpr}, e, e_1, e_2 \in \text{SemExpr}, \mu \in \text{MIR}, \) and \( \otimes \in \{+, -, \times, /\} \), and each rule has its inversed version.

**Boolean Operators**

Because in this chapter boolean expressions are introduced, we also make use of standard
boolean equivalences, which include commutativity \( b_1 \otimes b_2 \equiv b_2 \otimes b_1 \), associativity
\((b_1 \otimes b_2) \otimes b_3 \equiv b_1 \otimes (b_2 \otimes b_3)\), distributivity \((b_1 \lor b_2) \land b_3 \equiv (b_1 \land b_3) \lor (b_2 \land b_3)\), de Morgan’s
laws, where \( b_1, b_2, b_3 \) are boolean expressions, \( \otimes \in \{\lor, \land\} \), and \( \lor \) and \( \land \) are boolean or
and and operators respectively. Finally, reduction rules are also used to simplify boolean
expressions, which include contradiction \( b \land \neg b \equiv \text{ff} \), tautology \( b \lor \neg b \equiv \text{tt} \), and constant
propagation.

**Ternary Conditional Operator**

The ternary conditional operator has the following distributivity rules. Firstly, for unary
and binary arithmetic operators:

\[
\begin{align*}
-(b \ ? e_1 : e_2) & \equiv b \ ? -e_1 : -e_2, \\
e \otimes (b \ ? e_1 : e_2) & \equiv b \ ? (e \otimes e_1) : (e \otimes e_2), \\
(b \ ? e_1 : e_2) \otimes e & \equiv b \ ? (e_1 \otimes e) : (e_2 \otimes e).
\end{align*}
\]

(4.32)

Secondly, we can distribute over the conditional operators:

\[
\begin{align*}
b_1 \ ? (b_2 \ ? e_1 : e_2) : e & \equiv b_2 \ ? (b_1 \ ? e_1 : e) : (b_1 \ ? e_2 : e), \\
b_1 \ ? e : (b_2 \ ? e_1 : e_2) & \equiv b_2 \ ? (b_1 \ ? e : e_1) : (b_1 \ ? e_2 : e).
\end{align*}
\]

(4.33)
Thirdly, we introduce additional reduction rules to simplify ternary conditional expressions if possible:

\[
\text{tt} ? e_1 : e_2 \equiv e_1, \quad \text{ff} ? e_1 : e_2 \equiv e_2. \tag{4.34}
\]

**Composition Operator**

Similarly, the composition operator can also be distributed across arithmetic and conditional operators:

\[
\begin{align*}
(-e) \ast \mu & \equiv -(e \ast \mu), \\
(e_1 \otimes e_2) \ast \mu & \equiv (e_1 \ast \mu) \otimes (e_2 \ast \mu), \tag{4.35} \\
(b ? e_1 : e_2) \ast \mu & \equiv (b \ast \mu) ? (e_1 \ast \mu) : (e_2 \ast \mu).
\end{align*}
\]

**Fixpoint Operator**

The fixpoint operator represents loops, and loops can be partially unrolled, similarly, we can define a set of equivalence relations to perform partial unrolling on fixpoint expressions:

\[
\text{fix} (b, \mu, x) \equiv \text{fix} \left( b, p^k_\mu(\mu) \right) \text{ for } k \in \mathbb{N}. \tag{4.36}
\]

Here \( x \in \text{Var} \) and \( p_\mu \) is a function that computes the unrolling of the loop \( \text{MIR} \, \mu \), where

\[
p_\mu(\mu') = [x \mapsto (b \ast \mu) ? (\mu'(x) \ast \mu) : \mu(x)]_{x \in \text{Var}}. \tag{4.37}
\]

This set of rules formally defines the partial unrolling of loops with a certain number of steps \( k \). Using the rules to unroll the loop “\textbf{while} \ (b) \ \{ s \}” where \( b \in \text{BExpr} \), \( s \in \text{Stmt} \) is equivalent to unrolling the loop syntactically as follows, as an example, we consider cases when \( k = 0, 1, 2 \) in Figure 4.9.

The next set of rules in (4.38) extends the structural induction rules for arithmetic expression in (3.21) of Chapter 3 to inductively discover equivalent structures in child
while \((b)\) {
  s
  if \((b)\) {
    s
    if \((b)\) {
      s
    }
  }
}

(a) \(k = 0\).

(b) \(k = 1\).

(c) \(k = 2\).

Figure 4.9. A transformed while loop with different partial loop unroll depths.

expressions. In these rules, recall that the formulae above the line are the premises, whereas the ones below the line are conclusions. If the premise is true, then the conclusion must be true. For instance, in a conditional expression \(x < y \, ? \, (x + y) + z \, : \, x\), because the subexpression \((x + y) + z\) is equivalent to \(x + (y + z)\), it also has an equivalent expression \(x < y \, ? \, x + (y + z) \, : \, x\). We define similar rules respectively for the conditional, fixpoint and composition operators, and finally we also have an analogous rule for MIRs, which formalizes that if any subexpressions \(\mu(x)\) of an MIR \(\mu\) has an equivalent expression \(e_x\), then the MIR also has an equivalent MIR by replacing the expression \(\mu(x)\) with \(e_x\).

\[
\begin{align*}
  b &\equiv b' \\
e_1 &\equiv e'_1 \\
e_2 &\equiv e'_2 \\
\hline
  b &\equiv b' \\
e &\equiv e' \\
\hline
  e &\equiv e' \\
\hline
  b ? e_1 : e_2 &\equiv b' ? e'_1 : e'_2 \\
\hline
  \mu &\equiv \mu' \\
\hline
  f &\equiv f' \\
\hline
  \forall x \in \text{Var} : \exists e_x \in \text{SemExpr} : (\mu(x) \equiv e_x) &\equiv \mu = [x \mapsto e_x]_{x \in \text{Var}} \\
\end{align*}
\]

(4.38)

4.5.2 Discovering Equivalent Structures Efficiently

As we discussed earlier, discovering the full set of equivalent expressions by finding the transitive closure of the relations is infeasible because of combinatorial explosion. Chapter 3 proposed a method to drastically reduce the space and time complexity of discovering equivalent expressions, while achieving high quality optimizations.

We base our equivalent expression discovery on this method, but extend it to support not only simple arithmetic expressions but also additional program transform features.
proposed in this chapter, i.e. conditional, composition and fixpoint expressions, and MIRs. This enables full program transformations.

This section provides an informal overview of the equivalent discovery procedure. In Appendix B, we discuss the formal definition of this procedure for all semantic expressions and MIRs by extending the optimization function $O[e] \sigma^\sharp$ proposed in Section 3.3 of Chapter 3.

**Ternary Conditional Operator**

For conditional expressions of the form $b ? e_1 : e_2$, we first optimize the Boolean expression $b$. Then we use each equivalent expression of $b$ to constraint the program state for the optimization of $e_1$ and $e_2$. After optimizing child nodes, we combine the equivalent expressions of these three child nodes to form a set of equivalent conditional expressions. We further discover additional equivalent expressions from this set, and finally keep only those that are Pareto-optimal.

**Composition Operator**

Expressions of the form $e * \mu$ are optimized by first finding equivalences of $\mu$, and then for each of them discovered, we compute a new program state $\sigma_0^\sharp = E_m[\mu] \sigma^\sharp$, and use it to optimize $e$. Finally, all optimized combinations of $e$ and $\mu$ form a complete set of expressions equivalent to $e * \mu$, and Pareto-suboptimal candidates are pruned from this set.

**Fixpoint Operator**

Fixpoint expressions are optimized in three steps. Initially, partially unrolled versions are discovered using the rules defined in (4.36), the expressions are partially unrolled by a factor up to 3. Then for each unrolled version, we obtain its loop invariant using the algorithm illustrated in Figure 4.4, and the loop invariant is then used to optimize its child
nodes, which are the Boolean expression and the loop MIR. Finally, a set of equivalent fixpoint expressions can be derived by combining equivalent child nodes together, and we eliminate Pareto-suboptimal fixpoint expressions from this set.

**MIR**

MIRs can be optimized by first optimizing their expressions individually, then constructing a set of MIRs by enumerating all combinations of optimized expressions for each variable. The size of the final set is then reduced by pruning Pareto-suboptimal candidates.

However, because a MIR consists of multiple expressions and each has its own accuracy, we need a strategy to quantify the accuracy of a MIR. In Section 3.3.5 of Chapter 3, we discuss the reasons for this requirement, and various strategies to optimize multiple expressions, which are readily extendible to MIRs.

### 4.6 Code Generation

The final stage is to translate the optimized MIR back to a program in its original syntax. As discussed earlier, the MA produces an abstraction of the program, which means there are generally many ways of generating different programs from the same MIR. For this reason, certain heuristic optimizations are performed before or during code generation, such as branch- and loop-fusion transformations explained in our resource usage analysis to produce a unique and deterministic translation from the MIR.

Our code generation is carried out in three stages. The first stage applies the transformations outlined in Section 4.4 to perform loop- and branch-fusion, and to allow sharing of expressions across nested MIRs.

After the first stage, we create a topological sort of all nodes, which produces a linear ordering of all nodes such that the control- and data-dependences are preserved. We then perform a simple one-to-one mapping from the list of nodes to program code. An arithmetic node is translated into an assignment statement which assigns a temporary
variable with the result of the arithmetic computation. A ternary conditional and a fixpoint node respectively generate an \texttt{if} statement and a \texttt{while} loop. Finally, a composition node $\mu \rightarrow e$ ensures that $\mu$ is generated before $e$.

The final and optional step of our code generation, is to perform code sinking, which moves parts of the code so that when their results are not needed, they are not executed \cite{LA}. For instance, the result of the statement “$y = x + 1$;” is only used in the true-branch of the program:

\begin{verbatim}
y = x + 1;
if (x < 1)
  y = y * 2;
else
  y = x;
\end{verbatim}

This statement can therefore be moved into the true-branch of the \texttt{if} statement, so it may be evaluated only when needed. This final step further allows HLS tools to apply \texttt{if}-conversion to revert the changes if the tool believe this improves the resulting circuit.

### 4.7 Evaluation

In this section we optimize a set of numerical programs using SOAP in a benchmark suite with six different examples. We use the IEEE 754 32-bit single precision format with rounding to nearest mode as the data types of the floating-point values used in these examples. The benchmark consists of two introductory numerical programs, and four real applications that are frequently encountered in numerical analyses, where round-off errors could have big impacts on the quality of their execution. Appendix C contains the source code of all benchmark examples below.

- \texttt{simple}: for an input $x \in [0, 20]$, we repeatedly multiply it with 0.9, until the result is less than or equal to 1, the number of iterations is dependent on $x$ and can only be determined by analyzing the program.
• **basel**: the example in Figure 4.1.

• **taylor**: the Taylor expansion of $\cos(x + y)$, with single precision inputs $x \in [-0.1, 0.1]$ and $y \in [0, 1]$, and an integer $n \in [10, 20]$, which determines the bound on the iteration count.

• **filter**: computes the unit step response of a 3rd-order IIR filter, where inputs are bounded by $[0, 1]$ and all coefficients are bounded by $[0, 0.2]$, and it has a fixed iteration count 20.

• **euler**: it uses Euler’s method to solve the differential equation of a harmonic oscillator $\ddot{x} + \omega^2 x = 0$, with both an initial stationary position $x$ and $\omega^2$ bounded by $[0.0, 1.0]$, a step size of 0.1, and an iteration count $n \in [0, 20]$. It returns the position $x$ and velocity $\dot{x}$.

• **pid**: the example proportional–integral–derivative (PID) controller that was used as a case study in [Dam+14] as their motivation of automated accuracy optimization of numerical programs, we make it more challenging by changing constant coefficients to be bounds to model not only one, but a large selection of PID controllers, with $kp \in [9.0, 10.0]$, $ki \in [0.5, 0.7]$ and $kd \in [0, 3]$, and an iteration count $n \in [0, 20]$.

Our configuration for the efficient discovery of equivalent expressions uses a depth limit $k = 2$, and a $D = 3$ as the maximum number of times of partial loop unrolling. For each program we optimize it with SOAP to discover a wide range of implementations, and select the most accurate and least resource demanding equivalent implementations for further analysis.

Our code generator produces C source codes from the optimized MIRs, which are then synthesized with LegUp [LU], with floating-point operator sharing turned off to achieve maximum frequency, then compiled and verified with Quartus [Alt10], targeting an Altera Stratix IV device (EP4SGX530) [Alt16] for the actual resource usage statistics and the frequency achieved.

4.7 Evaluation
In order to provide evidence that the minimization of the statically analyzed round-off errors strongly correlates with the reduction of actual errors encountered during computation in circuits, the selected optimization candidates are also simulated for numerical accuracy. This simulation is carried out 1000 times with 1000 uniformly distributed random inputs for each benchmark example, then we compare the maximum round-off errors encountered.

4.7.1 Results

Table 4.1 shows the results of optimizing the above benchmark examples. Here we explain the meaning of each row and column.

The rows labeled “FR” and “MA” respectively show the statistics for the most resource efficient, and the most accurate implementations.

The column “Time (s)” shows the time required for the optimization. The optimization runtime is longer than a typical compiler optimizer, because during equivalent structure discovery, a significant amount of equivalent structures are examined.

The column “PF” shows for each benchmark example the total number of trade-off options in the Pareto frontier. Larger number indicates more choices to trade-off accuracy and area.

The values shown in the “Error Bound” column are the maximum absolute errors found by SOAP, while “Simulation” shows the actual absolute value bound on round-off errors found during simulation, and the percentage shows the actual accuracy improvement found in simulation of “MA” over “FR”. For each of these benchmark problems, the optimization for accuracy of them correlates to the reduction of simulated round-off errors in actual executions. They all show improvements in accuracy over the original by up to 65% in actual execution. The analyzed round-off errors are larger than the simulated bounds, because our efficient accuracy analysis over-approximates the worst-case round-off errors.
The “Resources” columns show our estimation of the number of LUTs and DSPs required for each of these programs, and the numbers in brackets are the corresponding statistics obtained from Quartus synthesis.

Table 4.1. Table of optimization results.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Accuracy</th>
<th>Resources</th>
<th>Fmax MHz</th>
<th>Time (s)</th>
<th>PF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error Bound</td>
<td>Simulation</td>
<td>LUTs</td>
<td>DSPs</td>
<td>MHz</td>
</tr>
<tr>
<td>simple</td>
<td>FR 7.1786E-06 6.8728E-06</td>
<td>241 (415)</td>
<td>4 (4)</td>
<td>386.3</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>MA 5.8827E-06 5.8262E-06</td>
<td>932 (1264)</td>
<td>16 (16)</td>
<td>318.9</td>
<td>168</td>
</tr>
<tr>
<td>basal</td>
<td>FR 2.9340E-06 3.2456E-07</td>
<td>3988 (4081)</td>
<td>0 (0)</td>
<td>306.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MA 1.1733E-06 1.4871E-07</td>
<td>28648 (29136)</td>
<td>0 (0)</td>
<td>279.0</td>
<td></td>
</tr>
<tr>
<td>taylor</td>
<td>FR 2.0482E-07 3.5672E-08</td>
<td>1596 (2116)</td>
<td>24 (24)</td>
<td>308.7</td>
<td>2008</td>
</tr>
<tr>
<td></td>
<td>MA 1.5756E-07 2.077E-08</td>
<td>6661 (7489)</td>
<td>112 (112)</td>
<td>174.7</td>
<td></td>
</tr>
<tr>
<td>filter</td>
<td>FR 7.5396E-07 1.0960E-07</td>
<td>3470 (4529)</td>
<td>24 (24)</td>
<td>289.4</td>
<td>1749</td>
</tr>
<tr>
<td></td>
<td>MA 5.2858E-07 6.7255E-08</td>
<td>21041 (26190)</td>
<td>104 (104)</td>
<td>172.0</td>
<td></td>
</tr>
<tr>
<td>euler</td>
<td>FR 9.0607E-05 2.4342E-07</td>
<td>1532 (2362)</td>
<td>12 (12)</td>
<td>327.7</td>
<td>1392</td>
</tr>
<tr>
<td></td>
<td>MA 4.8777E-05 8.549E-08</td>
<td>24314 (32637)</td>
<td>128 (128)</td>
<td>187.1</td>
<td></td>
</tr>
<tr>
<td>pid</td>
<td>FR 5.9869E-04 7.4389E-07</td>
<td>3975 (5576)</td>
<td>24 (24)</td>
<td>245.4</td>
<td>3607</td>
</tr>
<tr>
<td></td>
<td>MA 4.7977E-04 6.13867E-07</td>
<td>18047 (21927)</td>
<td>88 (88)</td>
<td>100.7</td>
<td></td>
</tr>
</tbody>
</table>

Because our benchmark examples are designed to be resource efficient, there is no room for resource usage optimization of the original program. However we are able to consistently reduce the resource usage of a plain partial loop unrolling by more than 25%, because our optimization can discover subexpression sharing opportunities, propagate constants values, and also aggressively reduce the size of expressions by powerful reduction rules such as $e - e = 0$ and $0 \times e = 0$.

Besides the choices of implementations that are either most accurate or most resource efficient, each optimization also offers a wide selection of optimized programs on the Pareto frontier. For instance, Figure 4.10a shows the Pareto frontier of euler, which has 26 different trade-off options. Furthermore, in the optimization of euler, our optimization not only identifies that it is resource efficient when the two return variables are computed by the same loop, but also by individually optimizing the accuracy of the two variables, we produce a program with two loops, each with a different goal, that is to compute their respective return variables as accurately as possible, this generated a program that consists of two loops that have completely different structures. With this, we further widen the trade-off curve with the most accurate option improving the
accuracy by 65%. In Figure 4.10b, because the loop kernel of filter has the expression:

\[ \sum_{i=0}^{2} (a_i y_{i+1} + b_i x_i), \]  

which has a large number of equivalent expressions, even without increasing the resource usage, our optimization improves its accuracy by 14.5%.

Moreover, points within the shaded region are Pareto-optimal as they optimize DSP count, since our Pareto frontier has three dimensions, which are respectively accuracy, LUT utilization and the number of DSPs.

### 4.7.2 Quality of Resource Estimation

The Pareto frontier is sensitive to how the quality metrics of each point (e.g. LUTs) compare against each other (i.e. the rank), while the actual number of the metrics is irrelevant. The Pareto frontier therefore is unaffected if the quality metrics are perturbed by a small error such that the rank is unchanged. To ensure that the resource estimation method used in our optimization can identify accurately whether an actual implementations is on the Pareto frontier, we gathered 150 implementations discovered across the benchmark examples, and for each one we rank its number of estimated LUTs among them, and do the same for actual LUTs. Figure 4.11 plots for each implementation, the rank of estimated LUTs against the rank of actual LUTs, which shows ranking of the resource estimation we produce is very close to the Quartus synthesized circuits.

### 4.8 Summary

A new method is proposed and carried out in SOAP that performs general numerical program transformation for the trade-off optimization among accuracy, and two resource related metrics, LUT and DSP utilizations. To optimize a numerical program, it starts by abstracting the program into a MIR, which we designed to extract the essence of executing the program, and removes unnecessary informations such as temporary variables, interleaving of non-dependent statement, etc. The MIR is then optimized efficiently by
discovering a wide range of equivalent trade-off implementations of the original MIR. An optimized MIR can then be chosen to be translated into a numerical program. By using SOAP, we optimize the accuracy of our sample applications by up to 65% in actual program executions.

In Section 4.7, our experiments show that accurate implementations of numerical algorithms often increase their number of arithmetic operations for reduced round-off errors. In general, this could result in circuits that have longer wall-clock time. However with greater area budget, we could have a greater freedom in rewriting programs to have greater throughputs. It is also observed that even a small program requires a long optimization time, which prevented us from exploring partial loop unrolling depths greater than 3, and hindered better accuracy improvements. In addition, the optimized programs may require a substantial increase in resource budget, because they do not share resources among different clock cycles. This will be addressed in the following chapter by introducing a new resource estimation analysis to allow operations to be shared temporally. In the next chapter, we therefore propose refinements for a faster optimization to explore deeper loop unrolling, and explore how to optimize numerical programs consisting of pipelined loops to trade off accuracy, resources and latency.
Figure 4.10. The Pareto frontier.

Figure 4.11. The quality of resource estimation.
Accurate and Resource Efficient
Pipelining of Numerical Programs

Numerical C programs typically spend most of their time in loops. For this reason, HLS tools adopt state-of-the-art polyhedral compilation techniques [Can+14] to synthesize loops to run as fast as possible. This is achieved by pipelining them to maximally exploit parallelism across loop iterations. Certain program transformations, such as conventional program equivalences (e.g. partial loop unrolling and array access pattern changes) are highly ubiquitous in their compilation process.

However, their ability to perform pipelining, even with any combinations of these equivalences, is fundamentally constrained by data-dependences that are carried across iterations, i.e. inter-iteration dependences. To relax these constraints, we must use equivalence rules in real arithmetic (e.g. associativity and distributivity), in tandem with the conventional rules above to enable much more efficiently pipelined RTL designs. A simple example of this is the summation of all elements in an array:

```c
float sum = 0;
for (int i = 0; i < N; i++)
    sum += a[i];
```

This code can be partially unrolled and the sequence of additions can be rewritten using tree adders to reduce its latency, but we will see later in Section 5.6 that more efficient implementations are possible.

In contrast to the expression balancing optimization pass in VHLS, the new SOAP in this chapter automatically produces results that are significantly better than manually tuning partial unrolling factors and expression balancing #pragmas in VHLS, because it is fully aware of how data-dependences are carried across iterations, and uses this to steer the
optimization process. SOAP is also conscious of the impact these transformations could have on round-off errors, and minimizes them in the optimization process, as we treat numerical accuracy as one of the three simultaneous objectives. Furthermore, VHLS only generates one result which does not necessarily improve over the original code.

The technical work presented in Chapters 3 and 4 lays the necessary foundation for the new methods proposed in this chapter. Firstly, we exploit the SOAP framework’s ability to analyze the numerical accuracy of a given program. Secondly, the framework provides the basis for resource analysis, as common subexpression sharing can be detected by the method detailed in Section 4.4 of Chapter 3. Thirdly, we can make use of MIRs to explore program rewrites. Finally, the efficient algorithms for equivalent program discovery in SOAP can be readily used.

In previous chapters, we only analyze structural reuse in the form of common subexpressions. HLS tools further allows certain arithmetic operations to be shared temporally. In this chapter, we therefore additionally analyze resource utilization by considering the implications of sharing the same resources among different clock cycles. For this purpose, we use a variant of the first few steps shared by modulo SDC scheduling [Can+14] and iterative modulo scheduling (IMS) algorithm [Rau94], which efficiently analyze the run time and resource utilization of a given program, by computing fundamental lower bounds of these metrics.

SOAP is evaluated on a suite of 11 programs from the Livermore Loops [DL11] and PolyBench [Pou] benchmark suites. Our tool obtained a wide selection of Pareto-optimized programs. Programs with the best latency obtained speedups of up to $12 \times (7 \times \text{on average across the suite})$, and increases in accuracy of up to $7 \times (2.7 \times \text{on average})$, while using up to $4 \times (2.5 \times \text{on average})$ more LUTs. We were unable to decrease the resource utilization in any of the benchmarks, as they have no redundant computations.

The contributions of this chapter include:

- An extended suite of program equivalence rules, which introduces access reduction rules that removes extraneous array accesses. This chapter provides evidence that
standard program equivalence techniques that do not affect program behavior, e.g. partial loop unrolling and access reduction rules, can give rise to the freedom for non-standard transformation rules, e.g. arithmetic rules, to significantly impact latency, resource usage and accuracy in a loop (Section 5.4.2).

- A new scheduling analysis that estimates the latency and resource usage of a given optimized candidate (Section 5.5). The resource usage analysis not only identifies common subexpressions, but also opportunities to share arithmetic operations temporally.

- A significantly faster efficient discovery of equivalent programs through a faster accuracy analysis that analyze a fraction of loop nest executions, speed up analysis of loop nests without inter-iteration dependences (Section 5.5.3), graph partitioning, and intelligent pruning of optimization candidates (Section 5.4.1).

- Incorporating the above-mentioned techniques, SOAP is now capable of automatically and safely producing optimized programs (and subsequent RTL implementations with VHLS) on the three-dimensional Pareto frontier of options that trade off run time, accuracy, and area. Its improvements in latency are notably better than the only ones produced by VHLS’s unsafe optimizations. SOAP is further evaluated on a suite of Livermore Loops and PolyBench benchmarks (Section 5.6).

This chapter, which is a natural extension to previous chapters, is organized as follows. Section 5.1 details how a simple numerical program can be optimized to run efficiently as our motivating example. Our automatic optimization process consists of three major steps. It starts by the process of metasemantic analysis, which takes as an input the original numerical program written in C, and translates it into a MIR. Section 5.3 explains how this process is extended to multi-dimensional arrays. We then discover equivalent MIRs using our efficient equivalent program discovery procedure, which produces a Pareto frontier of optimized MIRs. Section 5.4 discusses the improvements made to this procedure to further increase its performance. This process in turn makes use of the two new performance analyses in Section 5.5, which respectively estimate the latency and
resource utilization. This section further explains how round-off errors can be bounded in a given program with arrays. The optimized C programs can then be generated from the MIRs, using the code generation routines from Chapter 4, to be synthesized in VHLS to obtain RTL implementations. In Section 5.6 we evaluate the results of optimizing a suite of benchmark examples extracted from PolyBench [Pou] and Livermore Loops [DL11].

Figure 5.1 illustrates the high-level overview of the internal tool flow of the final iteration of SOAP.

![Figure 5.1](image_url)

**Figure 5.1.** An overview of our automatic program optimization process. The shaded region shows our internal tool flow.

### 5.1 Motivation

Figure 5.2 gives an implementation of the Seidel stencil computation, extracted from PolyBench [Pou], where initially all values in the array $A$ are single-precision floating-point values between 0 and 1. It resembles the typical code frequently used in fluid dynamic simulations for solving partial differential equations and systems of linear equations.
#define N 1024

for (int t = 0; t < 20; t++)
    for (int i = 1; i < N-1; i++)
        for (int j = 1; j < N-1; j++)

Figure 5.2. An excerpt from the Seidel stencil [Pou]. The inter-iteration data-dependence of the innermost loop is underlined (A[i][j] and A[i][j-1]).

We start by synthesizing this program in VHLS. We enable loop pipelining in VHLS, which asks it to optimize the loop by overlapping its iterations. However, we can observe that this program has very limited opportunity for pipelining, because each iteration \( j \) of the innermost loop ends by writing to \( A[i][j] \), and the next iteration \( j+1 \) begins by reading from \( A[i][j] \); this inter-iteration dependence is highlighted in Figure 5.2. Hence, it serves as our example to motivate a better SOAP to efficiently pipeline loops.

VHLS generates a schedule where the depth of the loop \( D \) is 49, and \( II \) as enforced by the data-dependences above is 46. The trip count of the innermost loop is \( N = 1022 \). The overall latency of the innermost loop is therefore \( ((N−1) \times II) + D = 47,015 \) cycles.

We then enable VHLS’s expression balancing (EB) optimization. When synthesized, this optimization pass tries to reorder the sequence of additions in the loop body into a tree structure, thus reducing the \( II \) to 28 cycles, and the depth \( D \) to 42 cycles, while the trip count \( N = 1022 \) remains the same. The overall latency is now \( ((N−1) \times II) + D = 28,630 \) cycles. The overall resource usage remains roughly the same.

However, as mentioned in Section 2.5 of Chapter 2, VHLS’s EB has two shortcomings. Firstly, it is not aware of the inter-iteration data-dependence and misses the opportunity to further pipeline this loop. Secondly, and most importantly, VHLS does not guarantee that this optimization will not result in catastrophic numerical inaccuracies.

We further discover that if the loop is partially unrolled, VHLS’s EB did not improve the total run time, despite using a lot more resources. Additionally, EB only makes use of
associativity, but not other equivalence rules. These limitations pose great restrictions on VHLS’s ability to produce a significantly faster implementation.

We then use the enhanced SOAP of this chapter to automatically discover equivalent programs from the program in Figure 5.2. Because SOAP explores a large number of paths that lead to a Pareto frontier of implementations, here we illustrate one of the many paths that could be taken by minimizing latency, while trying to optimize accuracy and resource usage. By using just arithmetic equivalences, SOAP specifically applies transformations to alleviate the constraints on the inter-iteration dependence, and discovers that the innermost loop can be rewritten to minimize latency as shown in Figure 5.3.

```c
for (int j = 1; j < 1023; j++)
A[i][j] = 0.2 * (A[i][j-1] +
((A[i][j] + A[i][j+1]) +
(A[i+1][j] + A[i-1][j])));
```

**Figure 5.3.** The optimized program using only arithmetic equivalences.

Although this loop still has a data-dependence between consecutive iterations, this transformation greatly reduces latency because most of the loop iterations can now be overlapped. We find that this simple transformation can reduce II to 19, which speeds up the original program by 2.3×, using almost the same number of LUTs and DSP elements as the original program. At the same time, the sequence of additions are now reordered to minimize round-off errors, improving the accuracy by 18%.

SOAP also supports more complex control-flow restructuring transformations, such as partial loop unrolling, in tandem with rules that optimize memory accesses and arithmetic calculations. This can further reduce the loop’s latency. In this example, unrolling the loop by a factor of two (i.e. updating two matrix elements on every iteration and halving the trip count) and applying other rules, results in a program with II = 19, D = 152, N = 511. When implemented on a device it is 4.8× faster than the original, and almost twice as accurate, at a cost of 17% more LUTs, as shown in Figure 5.4.

Further increasing the optimization effort, which enables the loop to be more deeply partially unrolled, leads to a program that is 7× faster than the original, but uses 2.8×
for (int j = 1; j < 1023; j += 2) {
    float t0 = A[i][j-1], t1 = A[i][j+1];
    float t2 = (A[i][j] + t1) + (A[i+1][j] + A[i-1][j]);
    float t3 = 0.04f * t2 + 0.2f *
        ((t1 + A[i][j+2]) + (A[i+1][j+1] + A[i-1][j+1]));
    A[i][j] = 0.2f * (t0 + t2);
    A[i][j+1] = 0.04f * t0 + t3;
}

Figure 5.4. The optimized program using arithmetic equivalences in tandem with control-flow restructuring and memory access optimization.

LUTs. To summarize, in Table 5.1, we compare VHLS with EB, against one of the many implementations that we have explored using SOAP with the increased optimization effort. The three columns respectively shows the original program with loop pipelining enabled, what VHLS can achieve alone, and the capability of SOAP. It is important to note that the round-off error is unknown for VHLS with EB, because it cannot predict the impact of its unsafe optimizations on accuracy. We performed place-and-route for exact statistics.

Table 5.1. Comparison among the optimized implementations generated by VHLS’s expression balancing and our optimizer. The row “Total run time (s)” indicates the wall-clock time in seconds of running the synthesized circuits.

<table>
<thead>
<tr>
<th></th>
<th>VHLS</th>
<th>VHLS with EB</th>
<th>VHLS with SOAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period (ns)</td>
<td>2.60</td>
<td>2.65</td>
<td>2.66</td>
</tr>
<tr>
<td>Inner latency (cycles)</td>
<td>47.0 k</td>
<td>28.6 k</td>
<td>6.59 k</td>
</tr>
<tr>
<td>Total run time (s)</td>
<td>2.50 m</td>
<td>1.56 m</td>
<td>0.358 m</td>
</tr>
<tr>
<td>LUTs</td>
<td>620</td>
<td>623</td>
<td>1778</td>
</tr>
<tr>
<td>DSP elements</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Round-off error</td>
<td>10.68 µ</td>
<td>unknown</td>
<td>4.31 µ</td>
</tr>
</tbody>
</table>

5.1 Motivation
5.2 Syntax Definition

Because we incorporate multi-dimensional arrays in our program optimization, this section further extends the syntax definition in Section 4.1 of Chapter 4 to support array data types. This can be achieved by modifying the definition of $x$ in (4.1):

$$x ::= v([a])^*.$$  \hspace{1cm} (5.1)

Here, $([a])^*$ indicates that the term $[a]$ can be repeated zero or more than zero times, and $v$ is a variable and $a$ is an arithmetic expression. The number of repetitions ($N$) indicates the variable $v$ refers to an $N$-dimensional array, and the absence ($N = 0$) denotes a scalar. When $x$ is used to declare an array, $a$ must be a positive integer to conform to the C syntax. To reflect this restriction and all above changes, the new syntax definition for numerical programs is therefore:

$$a ::= n \mid v([a])^* \mid -a_1 \mid a_1 \otimes a_2, \quad b ::= !b_1 \mid a_1 \oplus a_2 \mid b_1 \odot b_2,$$

$$s ::= [t]v([c])^* \mid [a] \mid s_1 s_2 \mid \text{if} (b) \{s_1\} \mid \text{else} \{s_2\} \mid \text{while} (b) \{s\},$$

$$v ::= x \mid y \mid z \ldots, \quad t ::= \text{int} \mid \text{float}.$$ \hspace{1cm} (5.2)

Here, $c \in \mathbb{N}$ is the set of positive integers. Recall that $a,a_1,a_2 \in \text{AExpr}$, $b,b_1,b_2 \in \text{BExpr}$, $s,s_1,s_2 \in \text{Stmt}$, $v \in \text{Var}$ are respectively arithmetic expressions, Boolean expressions, program statements, and variables of type $t$ which is either $\text{int}$ or $\text{float}$.

5.3 Extending MIRs with Arrays

In Section 4.2 of Chapter 4 we discussed a new IR for program optimization. However, it did not include support for arrays in the original description of the MIR format. All examples that motivate us in extending SOAP include arrays, so in this chapter, we extend MIRs to be able to represent programs that not only use scalar values, but also single- or multi-dimensional arrays.
In many imperative languages such as C, arrays are stateful objects, i.e. they have side-effects and are used to store information, and changes to them are reflected to concurrent parts of the program that may be oblivious to the changes. This characteristic is known as the lack of referential transparency [SS90]. Such behavior is not present in arithmetic expressions, many functional programming languages, SSA forms, as well as MIRs. This proves to be a challenge to us, because our efficient program optimization requires referential transparency to reliably and recursively divide the program into smaller subprograms that can be optimized independently, without affecting other subprograms.

To remedy this, we treat arrays as immutable. We use a function $\text{update}(A, \bar{x}, e)$ to return a new array that is the same as $A$ but with (multi-dimensional) index $\bar{x}$ now containing $e$. Similarly, the function $\text{access}(A, \bar{x})$ returns the element of $A$ at index $\bar{x}$. As a simple example, a loop body “$A[i + 1] = 2 \times A[i]$;” can be translated into the following MIR:

$$
\begin{align*}
A & \xrightarrow{\text{update}} A' \\
\bar{i} & \xrightarrow{\text{update}} \bar{i} + 1 \\
1 & \xrightarrow{\times} 2 \\
A & \xrightarrow{\text{access}} i
\end{align*}
$$

The implication of making arrays immutable is two-fold. Firstly, we disallow pointer aliasing, i.e. $\text{float } *b = a;$ is not allowed in the C code, to keep the translation simple. However this is not a problem for us because the programs that can benefit from our optimizations usually do not manipulate pointers. This issue can also be addressed in the future by performing pointer analysis. Secondly, diverged paths in array updates could occur if we naively optimize MIRs. For instance, if $A$ is an input array, consider the two expressions in a MIR, $\text{update}(A, \bar{x}, e)$ and $\text{update}(A, \bar{x}, e')$, where $e, e'$ are equivalent. They respectively update the $x$-th element of $A$ with $e$ and $e'$ and return different arrays. A C program cannot be generated from this MIR without duplicating $A$. We solve this problem by partitioning the MIR at “$\text{update}$” nodes using the method described in Section 5.4.

To give an example, consider the program which computes the Fibonacci sequence in Figure 5.5a, and it can be represented by the MIR shown in Figure 5.5b.
for (int i=2; i<1023; i++)
{
}

(a) The program.

(b) The MIR.

Figure 5.5. A simple example program and its corresponding MIR.

5.4 Structural Optimization

From a numerical program, we can generate a MIR using the translation process in Section 5.3. The next step is to transform the MIR, and discover MIRs that are equivalent to the original MIR in real arithmetic, but may execute differently in finite-precision arithmetic because of round-off errors.

5.4.1 Improved Algorithm

As discussed in previous chapters, even a small expression could have a huge number of equivalent ones. Exhaustively discovering all equivalent MIRs would result in combinatorial explosion of the number of equivalent MIRs in the search space. For this reason, we base ourselves on an algorithm from Section 4.5 of Chapter 4 that searches efficiently by discovering equivalences in a bottom-up hierarchy. In this section, we discuss two major improvements to the algorithm which further increase its performance.

Partitioning

Instead of optimizing the MIR immediately, we start by partitioning the MIR into multiple smaller sub-MIRs. The partition boundaries are determined by update operators. For instance, we consider the partially unrolled Fibonacci example in Figure 5.6a. The MIR
of the loop body is shown in Figure 5.6b. The partition boundaries are indicated by the region surrounded by the red dotted curve. A multiply shared subexpression, such as \( i - 1 \), also determines the partition boundary by merging its partition with one of its parents with the smallest partition by node count. If all parents contain the same number of nodes then a choice is made randomly.

```c
for (int i=3; i<1023; i+=2) {
}
```

(a) The partially unrolled loop.

Figure 5.6. An example to illustrate how MIRs are partitioned.

Because transformation rules can only be applied to each partition but not across them, the size of the search space can be reduced further. In turn, each are optimized separately and generate a set of partitions equivalent to the original. We then select combinations from these partitions to be merged, this generates a set of MIRs that are equivalent to the original. Finally, we preserve those MIRs merged on the Pareto frontier.

### Optimization

Previously in SOAP, as we optimize parts of MIRs, the Pareto frontier is used to filter discovered equivalent candidates (MIRs or semantic expressions), which keeps the size of the set relatively small and manageable. As we optimize larger programs, the run time of the tool increases significantly. Currently, the SOAP framework prunes the MIRs that are Pareto-suboptimal, leaving only those that are on the Pareto frontier. However, because our Pareto frontier is three-dimensional, there is a large increase in the number of Pareto-optimal MIRs. This Pareto pruning approach is no longer feasible for our
benchmark examples. Therefore in this chapter, not only do we use the Pareto frontier to filter candidates, we also introduce a PRUNE function to further reduce the size of Pareto frontier.

We rely on the PRUNE function to efficiently steer the direction of our Pareto frontier as we discover new candidates. It takes as an input the set of Pareto-optimal equivalent candidates that we have discovered, and prunes elements in this set to reduce its size by sampling, keeping the number of discovered MIRs tractable. The pruning algorithm is inspired by Poisson-disk sampling algorithm [Bri07]. Our algorithm in Figure 5.7 starts by first randomly selecting one point from the Pareto frontier $\epsilon_0$ (denoted by RANDOMSAMPLE($\epsilon_0$)). It then grows the set of points by adding the neighbours from the point that are separated by at least a certain distance $\delta$, where the distance $\delta$ is decreased iteratively by a factor $\zeta = 0.8$, until $\epsilon$ contains at least $\eta = 20\%$ of all points in the Pareto frontier, or a maximum number of attempts (attempt count) is reached. The distance between two options $\epsilon$ and $\epsilon'$ is computed as follows:

$$\text{dist}(\epsilon, \epsilon') = \sqrt{\sum_{f \in F} \left( \frac{f(\epsilon) - f(\epsilon')}{\max(f(\epsilon), f(\epsilon'))} \right)^2},$$

(5.4)

where $f \in F$ enumerates each function that evaluates the performance of a candidate $\epsilon$, i.e. the function $f \in F$ computes either the accuracy, area or latency of $\epsilon$.

This method is superior to random sampling, because random sampling often samples points that are close together, which usually are very similar implementations.

We found that with all improvements above and a faster accuracy analysis in Section 5.5.3, the algorithm is significantly faster than the original optimization algorithm in Section 4.5.2 of Chapter 4. Even though this algorithm may discover potentially fewer candidates on the Pareto frontier, we can now explore greater partial loop unrolling depths to widen the swing of the Pareto frontier in the same amount of time.
function SAMPLE($\epsilon_0$)
  $\delta = 1.0$
  $\epsilon = \{\text{RANDOMSAMPLE}(\epsilon_0)\}$
  for $i = 1, 2, \ldots, \text{attempt\_count}$ do
    for $e \in \epsilon_0$ do
      in\_range = \text{ff}
      for $e' \in \epsilon$ do
        if $\text{dist}(e, e') < \delta$ then
          in\_range = \text{tt}
          break
        end if
      end for
      if $\neg \text{in\_range}$ then
        $\epsilon = \epsilon \cup \{e\}$
      end if
    end for
  end for
  if $|\epsilon| \geq \eta |\epsilon_0|$ then
    break
  end if
  $\delta = \zeta \delta$
end function

Figure 5.7. The algorithm used to sample the Pareto frontier.

5.4.2 Transformation Rules

This section details the new transformation rules in the equivalence relation $\equiv$ and consequently in the structural optimization relation $\Rightarrow$. Each transformation rule on its own is not revolutionary, but for the first time, they are used in tandem with arithmetic rules and control-flow restructuring rules introduced respectively in Chapters 3 and 4. This enables a much better automatic structural optimization on the latency, resource usage and accuracy of numerical programs, than is possible using only a subset of them.

In Chapters 3 and 4, SOAP provides a range of equivalence rules that are used in the optimization, such as associativity, distributivity, commutativity, constant propagation, and partial loop unrolling. In Table 5.2, we list those rules that proved effective when minimizing loop latencies. Although these rules are used to transform MIRs, we present before-and-after examples written in C to allow the effect of each rule to be readily understood.
Our new rules, the access reduction rules, with formal definitions below and examples in Table 5.2, remove extraneous data-dependences that arise after partial unrolling. These rules, along with partial loop unrolling, mostly do not really impact latency, because they are very well studied in polyhedral loop dependence analysis, and tools such as VHLS can make use of them automatically. However, they give the necessary freedom to arithmetic rules to affect latency. The rules are as follows, where $A$ is an array, $i, j$ are subscripts, and $e, e'$ are expressions:

- **Multiple reads**, eliminates the second of two reads of the same location. This arises naturally from the MIR, as common subexpressions are shared.

- **Multiple writes**, eliminates a write that is overwritten:

$$\text{update} \ (\text{update} \ (A, i, e), i, e') \leadsto \text{update} \ (A, i, e').$$  \hspace{1cm} (5.5)

- **Read after write**, eliminates a read from a location that has just been written:

$$\text{access} \ (\text{update} \ (A, i, e), i) \leadsto e.$$  \hspace{1cm} (5.6)

- **Independent accesses**, allows two array operations to be reordered if it can be proved that they never access the same location:

$$\text{access} \ (\text{update} \ (A, i, e), j) \leadsto \text{access} \ (A, j), \text{if} \ i \neq j.$$  \hspace{1cm} (5.7)

We also visualize this rule in Figure 5.8, which shows a sample MIR transformation.
These rules may not seem powerful on their own, but when combined with other structural rules, they enable SOAP to detect dependences that can be removed in the MIR. This in turn allows more opportunities for the rules to further reduce loop latency. By way of illustration, we optimize the Fibonacci series example program in Figure 5.5a for latency. By partially unrolling the loop with a factor 2, we obtain the program in Figure 5.6a. We can see that because of the rigid array access pattern, associativity cannot be applied easily to the loop kernel. However, by applying the above access reduction rules first, we give associativity the freedom to reduce latency by half and improve accuracy by 50%, as shown in Figure 5.9.

```
for (int i = 2; i < 1023; i += 2) {
    float t2 = A[i - 2], t3 = A[i - 3];
    A[i - 1] = t2 + t3;
    A[i] = 2 * t2 + t3;
}
```

**Figure 5.9.** The optimized program that computes the Fibonacci sequence. It reduces latency of the original in Figure 5.5a by half and improves accuracy by 50%.

Without the above access reduction rules, it is therefore not possible to reach this optimized implementation. Conversely, it is not possible to relax scheduling constraints due to inter-iteration dependences without arithmetic equivalence rules, as these reduction rules are there to assist transformation rules that make a difference in latency. Therefore the rules in Table 5.2 must be used in conjunction with arithmetic and control-flow rules to optimize latency in numerical programs.
5.5 Performance Analysis

This section explains how we analyze MIRs for our three performance metrics: latency, resource usage, and accuracy.

5.5.1 Latency Analysis

The purpose of our latency analysis is not to create a complete scheduling of numerical programs, as this would be computationally expensive, and would need to be repeated for tens of thousands of equivalent programs. Instead, it computes a lower bound of the loop’s II, the minimum initiation interval (MII). (Recall that the initiation interval is the number of clock cycles that must elapse between the starts of two consecutive loop iterations, and is determined by data dependences and resource constraints.) We then compute the overall latency of the loop, and subsequently, the total latency of the program.

Following LegUp [LU], we compute MII values using the first few steps of modulo SDC scheduling [Can+14] introduced in depth in Section 2.2.3 of Chapter 2, by viewing MIRs as dependence graphs, as the structure of a MIR already captures intra-iteration data-dependences. In addition to this, we add extra latency information as attributes on the edges of MIRs, and new edges to form cycles that capture inter-iteration data-dependences. The analysis is carried out in three stages.

The analysis starts with the MIR of the loop under analysis. Each edge in the MIR, say $s \rightarrow t$, represents a data-dependence: the operation at node $s$ must be evaluated fully before the operation at $t$ can begin. The first step is to add a pair $(l, d)$ for each edge of the MIR. Here, $l$ is the latency of the edge (the number of clock cycles that must elapse between the start of $s$ and the start of $t$) and $d$ is the dependence distance (the number of loop iterations that must elapse between the start of $s$ and the start of $t$). Because all operations in the MIR are performed in a single iteration, all edges have $d = 0$. The value of $l$ is given by the latency of the operation at node $s$; if $s$ corresponds to an input variable or a numerical constant, then $l = 0$. 
The second stage is to add edges to form a cyclic dependence graph that captures read after write (RAW) dependences across loop iterations. This step involves checking whether each pair of “access” and “update” nodes has a dependence, and if so, adding a new edge between them with latency and dependence distance attributes. As an example, consider the MIR in (5.3) and assume each iteration increments $i$ by 1. Because in the original program, $A[i]$ and $A[i+1]$ are respectively reading from and writing to the same array $A$, we need to check if these accesses could touch the same memory location in different iterations. For this, our analysis formulates an ILP problem for the dependence distance, and solves it using the integer set library (isl) [Ver10]. In this example, the dependence distance is 1 because the value written to $A[i+1]$ in the current iteration $i$ is immediately used in the next iteration $i+1$. Similarly, we also add new edges for reads and writes to the same variable, which can be treated as a special array with only one element. Our analysis yields the MIR in Figure 5.10.

![Figure 5.10. The MIR with edges labelled with latency attributes.](image)

Note the new dashed edge from the update node to the access node, which is labeled $\langle -2, 1 \rangle$. The first value, $-2$, signifies that the latency of the edge between $\times$ and access, which is 2 cycles, is canceled out. This is because the multiplier can reuse its output from the previous iteration as the input for the current iteration without requiring a 2-cycle delay to read from $A$ for the value that was updated in the previous iteration. The second value, 1, indicates that there is a data flow dependence from iteration $i$ to iteration $i + 1$.

We assume no limit on the number of operators we can allocate, so operators do not constraint II. However, in VHLS, each array is usually translated into a dual-port RAM,
which allows only two accesses per clock cycle [Xil12], and thus constraints MII. Following Section 2.2.3, we evaluate:

\[
\text{ResMII} = \max_{a \in A} \left\lceil \frac{n_a}{r_a} \right\rceil, \tag{5.8}
\]

where \( a \in A \) ranges over all arrays in the loop body, \( n_a \) is the number of accesses to the array \( a \), i.e. the number of shared access and update nodes accessing the array \( a \) in the dependence graph, and \( r_a = 2 \) is the maximum number of accesses allowed per cycle per array.

The final step is to calculate \textit{recurrence-constrained minimum initiation interval} (RecMII) which is defined in Section 2.2.3 as:

\[
\text{RecMII} = \max_{c \in C} \left\lceil \frac{l_c}{d_c} \right\rceil, \tag{5.9}
\]

where \( c \in C \) ranges over all cycles in the graph, and \( l_c \) and \( d_c \) are respectively the sums of all latencies and dependence distances of the edges in the cycle. Because a typical MIR with array accesses could have a very large number of cycles, we efficiently search for an MII using a modified Floyd–Warshall algorithm [Flo62], following [Rau94].

Finally, we estimate the total latency \( L \) of the loop with:

\[
L = (N - 1) \text{MII} + D, \quad \text{where } \text{MII} = \max (\text{RecMII}, \text{ResMII}).
\]

Recalling from Section 2.2.3 in Chapter 2, \( N \) is the maximum \textit{trip count}, i.e. the loop’s total number of iterations, and \( D \) is the loop’s depth, i.e. the total number of cycles per iteration.

Because we optimize MIRs in a bottom-up hierarchy, when an expression is optimized that does not constitute an inter-iteration dependence is optimized, its latency is estimated by scheduling its operations by using an \textit{as-late-as-possible} (ALAP) [Wan+08] scheduling algorithm, where each operation is scheduled to the latest opportunity, while respecting the order of data dependences. This fast scheduling algorithm is also used to estimate the depth \( D \) of a pipelined loop.
Because the expression is eventually used in a loop, and the II of the loop is critical to how fast the loop can execute, it is necessary to start optimizing for II as soon as possible. Therefore, in our latency analysis of a MIR or an expression that is a fragment of a inter-iteration dependence cycle, our algorithm automatically shortens any paths between any pairs of dependent accesses in the MIR, as we use the latency analysis as a component to manoeuvre our optimization on the Pareto frontier. Moreover, we place greater weight on dependent accesses with smaller dependence distances, because these impact the resulting loop II more significantly than larger distances. For instance, consider a loop body that has two dependent accesses across iterations, i.e. the graph contains two cycles:

\[ A[i] = f(A[i-1], A[i-2]); \]

Here as we optimize this program in a bottom-up hierarchy, \( f \) is optimized before the loop body. For MII considerations, the subexpression tree \( f \) is thus rewritten such that the following latency cost is minimized in the latency analysis of \( f \):

\[ L = \max \left( \frac{l_1}{d_1}, \frac{l_2}{d_2} \right). \] (5.10)

Here, \( l_1 \) and \( l_2 \) are respectively the lengths of the longest latency-weighted paths from the nodes \( \text{access}(A, i-1) \) and \( \text{access}(A, i-2) \) to the root of \( f \), and \( d_1 \) and \( d_2 \) are respectively the dependence distances of the two nodes to the write of \( A \), where \( d_1 = 1 \) and \( d_2 = 2 \).

### 5.5.2 Resource Utilization Analysis

The hardware resource usage analysis of Chapter 3 captures the sharing of common subexpressions, but cannot analyze resource binding, which allows common operations to be shared across clock cycles. For instance, in the floating-point expression \( a + (b + c) \), the two additions can be computed using one addition operator only. In this section, we develop a new resource usage analysis that fully understands how resources are shared temporally in an FPGA implementation of numerical programs.
We rely on the foundation of resource usage analysis from Chapters 3 and 4, which counts the number $n_{\otimes}$ of each type of operation $\otimes \in \mathbf{Op}$, while maximally sharing common subexpressions. In a pipelined loop, we compute a lower bound $a_{\otimes}$ on the number of instances of $\otimes$ that must be allocated, using the following formula:

$$a_{\otimes} = \begin{cases} \left\lceil \frac{n_{\otimes}}{\text{MII}} \right\rceil & \text{if } \otimes \text{ is shared,} \\ n_{\otimes} & \text{otherwise.} \end{cases}$$ \hspace{1cm} (5.11)

Here, integer operators are typically not shared [Li+15], so the number of operations is the number of allocated instances.

For instance, if we know that a pipelined loop has MII = 3, and each iteration uses 6 multiplications, then we can compute that we need to synthesize at least 2 multipliers.

For straight-line code, non-pipelined loops, and different loops, we use a simple ALAP scheduling [Wan+08] to estimate resource utilization.

Finally, we accumulate the number of LUTs and DSP elements for all allocated operators. In addition, we estimate the number of LUTs required by multiplexers generated for sharing operators, where $R_{\text{LUTs}}^{\text{mux}}$ approximates $1/n$ of the number of LUTs required by an $n$-to-1 multiplexer. The final result is the estimated resource utilization for the full program:

$$r_{\text{LUT}} = \sum_{\otimes \in \mathbf{Op}} \left( a_{\otimes} R_{\otimes}^{\text{LUTs}} + (n_{\otimes} - a_{\otimes}) R_{\text{LUTs}}^{\text{mux}} \right),$$

$$r_{\text{DSP}} = \sum_{\otimes \in \mathbf{Op}} a_{\otimes} R_{\otimes}^{\text{DSPs}},$$

where $R_{\otimes}^{\text{LUTs}}$ and $R_{\otimes}^{\text{DSPs}}$ denote the number of LUTs and DSPs required by one operator $\otimes$ respectively.

### 5.5.3 Accuracy Analysis

We extend the accuracy analysis of Chapter 4 to support arrays. Because our benchmark suite consists of programs with large arrays, we keep the analysis efficient by treating an entire array as a pair of a floating-point interval and an interval of accumulated round-off
errors. These intervals, representing the worst case bounds of all elements in the array, accumulate all values that are assigned to the array, and never shrink the range bounded by these intervals when we assign new values to an array location. We therefore define the read and write accesses to an array as follows in the accuracy analysis:

\[
E_s [\text{access} (A, \bar{i})] \sigma^\sharp = E_s [A] \sigma^\sharp,
\]
\[
E_s [\text{update} (A, \bar{i}, e)] \sigma^\sharp = E_s [A] \sigma^\sharp \sqcup E_s [e] \sigma^\sharp,
\]

(5.13)

where \( A \in \text{Var} \) is an array variable, \( \bar{i} \) is a subscript to index an element in \( A \), \( e \in \text{SemExpr} \) is an expression, and \( \sigma^\sharp \in \Sigma^E \) is the input abstract program state (recall from Section 4.3 of Chapter 4).

Alternatively, we can view each element \( A[i] \), where \( i \) is a tuple with \( N \) non-negative integers, in an \( N \)-dimensional array \( A \) as a variable. Updating \( A \) thus produces an abstract state which collects all elements in \( A \):

\[
E_s [\text{access} (A, \bar{i})] \sigma^\sharp = \bigsqcup_{i \in E_s [\bar{i}] \sigma^\sharp} \sigma^\sharp (A[i]),
\]
\[
E_s [\text{update} (A, \bar{i}, e)] \sigma^\sharp = \left[ A[i] \mapsto \sigma^\sharp (A[i]) \sqcup E_s [e] \sigma^\sharp \right]_{i \in E_s [\bar{i}] \sigma^\sharp},
\]

(5.14)

where \( i \in E_s [\bar{i}] \sigma^\sharp \) ranges over all possible indices bounded by \( E_s [\bar{i}] \sigma^\sharp \) that are tuples with \( N \) non-negative integers.

Additionally, because most of the loops in our benchmark programs consist of nested loops and have large iteration counts, the fixpoint analysis routine in Section 4.3.4 is modified to analyze only a small fraction of the innermost loop execution of a loop nest. By ensuring the innermost loop iterator increments by the same amount, we can guarantee the accuracy analysis to be fair for each equivalent implementation for fixpoint expressions with different unroll factors. For the experimental outcomes in Section 5.6, we analyze 10\% of the total executions of an innermost loop for the purpose of optimization.

Finally, we further use the dependence analysis in the MIR graph explained in Section 5.5.1 to detect whether errors are accumulated across iterations. This process is
carried out by first analyzing the MIR of the loop body for intra-iteration dependences. The absence of such dependences indicates the round-off errors do not accumulate across iterations, and it suffices to analyze the fixpoint expression for one loop iteration.

5.6 Evaluation

We have evaluated SOAP on a suite of benchmark examples which consists of several applications that have recurring inter-iteration dependences:

- A simple loop, `sum`, that sums the elements in an array.

- Two kernels from Livermore Loops [DL11]: `dotprod`, which computes the dot product of two vectors, and `tridiag`, which solves a tridiagonal linear system of equations.

- Nine kernels from PolyBench [Pou], which calculate matrix/vector transpositions, additions and multiplications (`2mm`, `3mm`, `atax`, `gemm`, `gemver`, `mvt`), the bi-conjugate gradient stabilized method (`bicg`), the Seidel stencil computation (`seidel`, modified to compute with 5 points), and symmetric rank-2k operations (`syr2k`).

All elements of input arrays and matrices are set to be single-precision floating-point values between 0 and 1. We optimized all of these benchmark examples using SOAP, specifically targeting the Xilinx Virtex7 device running at 333 MHz, for the three objectives of accuracy, resource utilization and latency simultaneously. We then used VHLS 2015.2 [Xil12] to synthesize the resulting optimized programs into RTL implementations for exact latency information, and performed place-and-route using Vivado Design Suite 2015.2 [Xil15], to obtain exact resource utilization statistics. Finally, SOAP produces a four-dimensional Pareto frontier for each program, where the dimensions are accuracy, latency, the number of LUTs and DSP element count, for clarity we visualize the results in three dimensions only.
Table 5.3. Comparisons of the original (non-shaded rows) and the optimized program with lowest latency (shaded rows), for each benchmark. Values in parentheses are obtained after slightly tweaking our experimental set-up; see Section 5.6.2. We performed place-and-route for exact statistics.

<table>
<thead>
<tr>
<th>Name</th>
<th>DSPs</th>
<th>LUTs</th>
<th>Error</th>
<th>Clock</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ratio</td>
<td></td>
<td>(ns)</td>
<td>(cycles)</td>
</tr>
<tr>
<td>sum</td>
<td>2</td>
<td>303</td>
<td>0.257</td>
<td>914μ</td>
<td>7.93</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1181</td>
<td></td>
<td>1.15μ</td>
<td></td>
</tr>
<tr>
<td>dotprod</td>
<td>5</td>
<td>411</td>
<td>0.231</td>
<td>926μ</td>
<td>7.29</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1781</td>
<td></td>
<td>127μ</td>
<td></td>
</tr>
<tr>
<td>tridiag</td>
<td>5</td>
<td>470</td>
<td>0.288</td>
<td>63.1μ</td>
<td>1.06</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1631</td>
<td></td>
<td>59.4μ</td>
<td></td>
</tr>
<tr>
<td>2mm</td>
<td>5</td>
<td>781</td>
<td>0.385</td>
<td>209</td>
<td>3.40</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2029</td>
<td></td>
<td>61.4</td>
<td></td>
</tr>
<tr>
<td>3mm</td>
<td>5</td>
<td>760</td>
<td>0.207</td>
<td>114</td>
<td>6.76</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3677</td>
<td></td>
<td>16.9</td>
<td></td>
</tr>
<tr>
<td>atax</td>
<td>5</td>
<td>627</td>
<td>0.507</td>
<td>353m</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1237</td>
<td></td>
<td>230m</td>
<td></td>
</tr>
<tr>
<td>bicg</td>
<td>5</td>
<td>427</td>
<td>0.304</td>
<td>887μ</td>
<td>6.72</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1406</td>
<td></td>
<td>132μ</td>
<td></td>
</tr>
<tr>
<td>gemm</td>
<td>5</td>
<td>524</td>
<td>0.234</td>
<td>1.99</td>
<td>2.97</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2240</td>
<td></td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>seidel</td>
<td>5</td>
<td>620</td>
<td>0.349</td>
<td>10.7μ</td>
<td>2.46</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1778</td>
<td></td>
<td>4.31μ</td>
<td></td>
</tr>
<tr>
<td>gemver</td>
<td>5</td>
<td>809</td>
<td>0.382</td>
<td>7.28M</td>
<td>4.46</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2120</td>
<td></td>
<td>1.63M</td>
<td></td>
</tr>
<tr>
<td>mvt</td>
<td>5</td>
<td>701</td>
<td>0.251</td>
<td>91.0μ</td>
<td>3.32</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2793</td>
<td></td>
<td>27.4μ</td>
<td></td>
</tr>
<tr>
<td>syr2k</td>
<td>5</td>
<td>709</td>
<td>0.259</td>
<td>250μ</td>
<td>4.07</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2740</td>
<td></td>
<td>61.4μ</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.289</td>
<td>3.69</td>
<td></td>
</tr>
</tbody>
</table>

5.6.1 Results

Table 5.3 compares, for each benchmark in our evaluation set, the performance metrics of the original program against those of the program with the smallest latency discovered by SOAP. We synthesized each program to a circuit to obtain exact statistics, which are shown in Table 5.3.

Figure 5.11 compares our estimated LUT counts (vertical axis) against the exact LUT counts (horizontal axis) obtained by synthesizing RTL implementations of each program.
Figure 5.11. Comparisons of our estimated LUT counts against actual LUT counts from VHLS.

in Table 5.3. Although our estimates deviate from the exact values, because we compute lower bounds on resource utilizations, and finite state machines synthesized and address calculation are not taken into account, our estimate can still accurately predict the general trend—a linear regression of all scatter points finds $R^2 = 0.9344$.

Figure 5.12 compares our latency estimates (vertical axis) against the actual latency values (horizontal axis). The solid line represents the linear regression of data points that we have gathered in Table 5.3. This line is a tight fit with our data, with $R^2 = 0.9959$, which indicates that our latency estimation can accurately predict the exact latency of synthesized implementations.

Returning to our motivating example from Section 5.1, Figure 5.13 demonstrates the range of optimized programs discovered by SOAP when applied to the Seidel stencil loop kernel. In the figure, ×-points indicate the original program. By using only the rules of real arithmetic, SOAP finds a more efficient program that can improve run time by $2.5\times$, as shown by the ○-points. However, by enabling partial loop unrolling and our dependence elimination rules, the performance is further improved, resulting in a
Figure 5.12. Comparisons of our estimated latency statistics against actual latency from VHLS.

6.7× reduction of total run time. Furthermore, we have found that numerical accuracy can often be optimized at the same time as we optimize the initiation intervals of loops. Because by partially unrolling loops, the sizes of the expressions in loop grow, which provides SOAP a greater freedom in terms of restructuring expressions and discovering more accurate variants. In this example, the fastest program is also the most accurate one: it minimizes round-off errors by approximately 2.5×. It is worth noting that SOAP can detect that as it explores deep levels of partial loop unrolling, we start to see a diminishing return in performance as it hits a bottleneck in memory bandwidth. This is due to the fact that VHLS synthesizes dual port RAMs for arrays, and in one clock cycle we can only read from the memory allocating array twice. Our optimization flow detects this bottleneck as it prunes them from the Pareto frontier, and stops exploring further loop unrolling.

Similar graphs for the other benchmarks can be viewed online, each showing three projections from different axes of the 3D Pareto frontier. Our web page can be used to in-

*https://admk.github.io/soap/plot.html
teractively explore the positions of each data point on the three projections simultaneously, and view the corresponding generated C programs.

5.6.2 Discussion

As demonstrated by Figure 5.12, SOAP generally produces accurate latency estimates. However, we have discovered a few notable discrepancies. For instance, gemver, mvt and syr2k all have significant differences between our estimated latency and the actual latency from synthesized RTL implementations. An inspection of these programs reveals that they all share a common programming idiom:

```c
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        x[i] += ...;
```

We found that VHLS occasionally fails to find the optimal schedule, predicted by SOAP, that could pipeline this loop as tightly as possible. Rewriting the above code into the following:

```c
for (int i=0; i<N; i++)
    float sum = x[i];
    for (int j=0; j<N; j++)
        sum += ...;
    x[i] += sum;
```

fixes this problem, and enables VHLS to generate a hardware implementation with the expected II. The ratios in parentheses in Table 5.3 reflect the speedup by performing this simple fix.

After this modification, we discovered that for all of the benchmark examples, VHLS generated circuits with the same II values predicted by SOAP, despite in practice, the MII may be unachievable for certain applications that are resource-constrained. Because floating-point operations are often associated with long latencies, the MII of floating-point numerical programs could require multiple cycles, reducing the constrain imposed by resources on the actual II.
5.7 Summary

Minimizing the latency of loops is a central task for HLS tools that obtain FPGA implementations from numerical C programs. Loop latency can often be reduced by performing simple rewrites to minimize inter-iteration data-dependences, but HLS tools cannot enable such rewrites by default because they may impact the accuracy of floating-point computations. This chapter presents the first tool that is able to automatically rewrite a given program to optimize latency, while controlling for accuracy and resource usage. Our experimental results suggest that, in fact, latency and accuracy are often \textit{not} in conflict: that programs aggressively optimized for latency can also have minimal round-off errors, albeit with greater resource usage. We have demonstrated that SOAP can optimize commonly used code fragments from PolyBench [Pou] and Livermore Loops [DL11] to have up to a $12 \times$ increase in performance, and up to $7 \times$ reduction of round-off errors, at the cost of up to $4 \times$ more resource utilization.
Figure 5.13. Pareto-optimal variants of the Seidel stencil program from Figure 5.2. Each graph shows a 2D projection of the 3D Pareto frontier. In each graph, the original program is marked ×, and the lowest-latency variant obtained by arithmetic transformations alone is marked by the red circle.
HLS tools are typically designed to adhere to a rigid specification which outlines their behaviour. It is a traditional practice to design this specification and the subsequent tool to ensure that the synthesized circuits perform functionally identically to the original source program written in the HLL. This is also viewed as good practice because it has predictable outcomes. Guided by the rules of the language, programmers translate mathematical objects such as algorithms and physical information respectively into source code and numerical data, in a way similar to tools adhering to their specifications. This manual process of translation is unfortunately an approximate one. Computations as simple as $\sqrt{3}$ must be approximated, e.g. they are carried out in floating-point arithmetic, because of the finite nature of computing machines. Therefore, HLS tools cannot be relied upon for an exact interpretation of the mathematical objects we wish to implement, even if they guarantee the functional equivalence between the source code and the synthesized result.

Despite the awareness of the approximate nature of numerical software/hardware implementations using floating-point operations, engineers often take the risk of neglecting this fact, and anticipate their designs to behave identically to the mathematical algorithms visioned in real arithmetic within a reasonable but not well-defined error margin. As it was shown in Section 2.5.2 in Chapter 2, round-off errors when accumulated could have detrimental effects on our daily life. The aforementioned functional equivalence between source and circuit guaranteed by HLS tools is therefore unable to regain any lost accuracies due to approximation.

Traditional IR-level HLS program optimization consists of a series of transformation passes. Most of these passes do not predict whether they have negative impact on the resulting circuit, and they limit their capabilities by preserving functional equivalence. Varying the
order of these passes could have significant impact on the quality, as these passes interact with one another in a complicated manner, it is difficult to predict the overall impact on performance [Hua+15]. For \( n \) passes, there are up to \( n! \) distinct ways to order, it is thus a considerable challenge to decide the optimal pass ordering, which is exacerbated by the fact that it could be highly dependent on the input program [Con+13].

The above shortcomings of traditional HLS tools and optimizing compilers provide a strong motivation for the work proposed in this thesis.

Firstly, we can apply the philosophy of relaxing the functional equivalence required by HLS tools. In the mean time, we infer equivalences of the underlying mathematical objects in real arithmetic which hardware designs are approximating. One can often improve the numerical accuracy by choosing a better alternative among these equivalences.

Secondly, by the same paradigm shift, a wide range of optimization opportunities can be explored to minimize throughput and resource utilization. These opportunities were previously lost out to the necessity of ensuring consistent behaviour.

Finally, optimization can be carried out by applying steps of equivalence rewrites driven by a prediction model. Traditional optimization passes can be broken up into much smaller common parts made of equivalence rules which can easily be proved mathematically correct. By using models to predict run time, resources and accuracy to guide the optimization process, it is possible to explore multiple designs that trade-off the three performance metrics while removing concerns about the ordering problem. Many optimization passes, such as constant propagation, dead code removal, common subexpression elimination, etc., are naturally subsumed by the new approach. As the computational power of machines increases exponentially, we can foresee an increase in the scale of the vast search space to be explored in the future.

This thesis therefore broadens the horizon of HLS tools, and equips them with the new program optimization paradigm by leveraging the above observations. Specifically, the trade-off among numerical accuracy, resource utilization and throughput is optimized in
floating-point numerical programs for HLS. Here we summarize the contributions of this thesis.

To the best of our knowledge, this thesis is the first to introduce multiple-objective performance optimization in a unified framework for discovering equivalence in programs. Chapter 3 implements this framework and optimizes a suite of expressions that are difficult to optimize by hand, and improves numerical accuracy and area automatically. In the experimental results, it turns out that the two central goals, i.e. improving accuracy and minimizing area, are often not in conflict, as optimized expressions can enjoy enhancements that can be achieved in both metrics. Guided by the concept of abstract interpretation, we further introduce the semantics-based program analyses to jointly reason about safe ranges of round-off errors and resource utilization, and subsequently, discovery of equivalent expressions. This technique lays the necessary foundation for program equivalence beyond simple arithmetic expressions.

The infinite size of the equivalent program space, coupled with undecidability of program properties, makes the program optimization an even more challenging task than the one of arithmetic expressions. For this, Chapter 4 introduces a new graph-based intermediate representation, MIR, for capturing the semantics of numerical programs. This approach reduces the size of the search space, and the IR itself is derived from the formal semantics of programs to ensure the correctness of equivalent MIRs and the back-and-forth translation between C and MIR. This further eliminates the problem of optimization pass ordering, because by using the equivalence discovery framework, the Pareto frontier can be extended incrementally with small steps of rewrites to multiple candidates. Traditional compiler optimizations are naturally subsumed and further enhanced by the MIRs, as many optimization techniques such as loop splitting and loop fusion that previously must be profiled to justify enabling them, can emerge automatically from the optimization process. By optimizing a suite of resource-efficient benchmark examples, the tool improves the numerical accuracy by up to 65%.

Formerly, HLS tools’ ability to pipeline loops is fundamentally constrained by intra-iteration dependencies. Traditional optimization techniques such as partial loop unrolling
may have minimal effects on the initiation interval of pipelined loops, as these do not impact the data-path structure, which ensures that the functional equivalence is preserved. Encouraged by the promising effects of Nicolau et al.’s tree height reduction technique [NP91] and LegUp’s recurrence minimization [Can+14], Chapter 5 further incorporates latency analysis into the unified program optimization framework. It was found that traditional optimization techniques when used in tandem with the arithmetic equivalence rules and memory access reduction rules can significantly improve the latency and accuracy of a numerical program. In Chapter 4, the experimental results identify that the static analysis of round-off errors for each candidate explored is the key factor for the speed of optimization. This problem is addressed in this chapter by graph partitioning and candidate pruning algorithms. This further enables deeper partial loop unrolling factors not explored in Chapter 4. Often as we optimize numerical programs by allowing greater resource budgets, latency and round-off error can be simultaneously minimized, as more resources would allow greater flexibility to discover equivalent programs that often perform well in terms of run time and accuracy. Additionally, this process is simultaneously driven by resource minimization, allowing the area-latency product to decrease as we explore increasingly deeper partial unrolling. By optimizing a suite of benchmark examples from PolyBench and Livermore loops, the tool improves the latency and accuracy of each by up to $12\times$ and $7\times$ respectively, at a cost of $4\times$ more resource utilization.

6.1 Future Prospects

In its current form, the new approach to program optimization explained in this thesis forms the underlying basis for a much larger set of future work. Even though it is precursory on its own, the promising experimental results showcase the powerful optimization it can bring to optimizing compilers and HLS tools. Here, a list of potential directions of future research is discussed that could further widen the scope of our technique for a broader range of applications.
LLIR-Level Program Optimization. We could envision a back-and-forth translator from LLIR [LA; LLIR] to MIR graphs. This could enable a much wider applicability of the techniques presented in the thesis to both LLVM-based HLS tools and software compilers. Additionally, it could benefit from existing LLVM optimizations passes by using the optimized LLIR code as inputs. There are, however, obstacles in migrating to LLIR as the source language. Firstly, LLIR is SSA-based. Since it uses temporary variables for intermediate results in computation, a full liveness analysis [CT11; Nie+99; Boi+08] may be necessary to eliminate temporary variables from the resulting MIR. Secondly, control-flows in LLIR are more freely structured. Unlike C, which defines if statements and while loops and discourages the use of goto statements, control-flow in LLIR are composed by basic blocks and branches between pairs of them. This requires the MIR to be further extended to cope with complex control-flow patterns. Conventionally, programs written with branches are often analyzed using continuation style semantics [Fel+88]. It is not evident how this semantics can be embedded within MIRs.

Tighter bounds on round-off errors. As an alternative to interval analysis, the accuracy analysis could enjoy more sophisticated abstract domains that capture the correlations between variables, and produce tighter bounds for results. Currently, the analysis cannot produce meaningful, i.e. finite, bounds on the round-off errors of certain numerical programs. If the analysis fails to bound errors, then currently the optimization cannot be directed to a more accurate implementation. By using abstract relational domains, it is possible to produce a much tighter bound on the values of program variables, and the associated errors. There are a few relational domains-based static analysis techniques of floating-point errors [Min07b; Put+04; GP11; Ast], however making use of them still poses challenges. Each floating-point operation introduces an independent error term as a new variable in the formulation of these relational domains, and it may be difficult to determine how to collapse these error terms into a smaller set of variables, as the optimization in this thesis can introduce a large number of error variables.

Special and fused operators. There could be a lot of interest in the HLS community on how SOAP can be incorporated with existing work on fused floating-point data-path synthesis. Langhammer et al. [LV09] propose that the normalization and denormalization
stages could be regarded as redundant between operators in a floating-point data-path. By removing these stages, subsets of the data-path become fixed-point data-paths, in the meanwhile saving resources and improving throughput at a cost of accuracy. It could be compelling to isolate the normalization/denormalization stages into operators in the SOAP framework, so that a mixed floating-point/fixed-point program can more efficiently trade-off resources, accuracy and latency.

Multiple word-lengths. In this thesis, experiments have been carried out on floating-point operations with a fixed mantissa width only. It would be beneficial to further integrate fixed-point support. Additionally, by further supporting multiple precisions in the data-path, i.e. allowing each operator to compute with different precisions, the trade-off relationship among our three primary performance measures can be even more effective. Techniques, known as multiple word-length optimization [Con+11b; Lee+06; Can+02], exist to apply a heuristic approach to perturb the precisions in a data-path, so that a performance metric can be optimized while the round-off errors of outputs satisfy an error budget. Instituting such techniques in the SOAP framework is rewarding as it can further reduce the area and latency requirement of a synthesized circuit for a given accuracy. All of these approaches optimize a fixed data-flow graph, whereas in SOAP the structure of the data- and control-paths vary as we optimize them. Analyzing each of the candidates for an optimal precision assignment to each operator is very inefficient because of the number of candidates explored. Moreover, current techniques work with a predetermined error budget, and yet in fact a Pareto frontier exists for each data-path to trade-off accuracy, resources and latency.

Numerical analysis and linear algebra. There are two distinct approaches to the analysis of round-off errors. One focuses on the round-off errors by statically analyzing numerical programs, and applies this in a way which is as general as possible, similar to the method presented in this thesis. On the other hand, there are techniques employed by numerical analysts to evaluate and improve the numerical accuracy and stability of particular algorithms analytically. Many creative solutions to challenges are invented in this process. For instance, Kahan’s compensated summation algorithm is an accurate way to compute a sum of \( n \) values, \( \sum_{i=0}^{n-1} x_i \) [Kah65], which is shown in Figure 6.1. This
algorithm cannot be discovered easily using the method outlined in this thesis, and a way
to extend the framework to optimize programs as creatively as humans still eludes us
at the moment. Higham et al. [Hig02] discuss in great depth many existing numerical
accuracy problems encountered in finite-precision computation of polynomials and linear
algebra subprograms and how to analyze and overcome inaccuracies, often in terms of
relative errors. Bridging the gap between computational and mathematical approaches
for numerical analysis will allow us to automate many accuracy optimizations that were
previously unexplored by the tool.

```c
float compensated_summation(float X[N])
{
    float sum = 0.0f;
    float e = 0.0f;
    for (i = 0; i < n; i++)
    {
        float tmp = sum;
        float y = X[i] + e;
        sum = tmp + y;
        e = (tmp - sum) + y;
    }
    return sum;
}
```

Figure 6.1. Kahan’s compensated summation algorithm to accurately compute the sum of \( n \) elements \( \sum_{i=0}^{n-1} x_i \).

**Continuity analysis and optimization.** The robustness of programs is very important
to us. In many cases, we wish our algorithms to be free from discontinuity, i.e. a small
change in the initial condition would not result in an undesirably large jump in the
outputs. For this, Chaudhuri et al. [Cha+11] and Goubault et al. [GP13] respectively
propose methods to analyze the robustness of programs. The former approach formally
proves whether an algorithm is ill-conditioned in terms of the existence of discontinuity,
whereas the latter statically analyzes programs to determine whether round-off errors
introduce significant discontinuous behaviour. To illustrate, consider an \texttt{if} branch,
```
if (e > 0) c_1 else c_2
```
where \( e \) is a floating-point expression. If \( e \) is positive and
very close to 0 when evaluated in real arithmetic, the floating-point result of \( e \) could be
non-positive, due to the effect of the round-off errors. In these extraordinary cases, the \( c_2 \)
branch may be executed instead of the intended $c_1$. These above new techniques could inspire us to implement the optimization of discontinuous behaviour, such as the one shown in the example, as another objective.

**Memory partitioning.** The experimental results in this work see a diminishing performance return when loops are deeply unrolled, because of a memory bottleneck. As memory accesses saturate in loop execution, i.e. all memory ports are working in 100% utilization, it is impossible to gain further performance improvements. Currently, the tool stops exploring further loop unrolling when this happens. By automatically partitioning arrays upon hitting such a memory bottleneck, further throughput improvements can be achieved.

**Integer programs.** SOAP can optimize programs written with integer operations, but with notable limitations. Firstly, accuracy analysis does not provide useful information, because integer operations do not have round-off errors. Secondly, the scheduling analysis in Chapter 5 can no longer accurately predict the run time performance of the resulting circuit, because HLS tools employ operator chaining to schedule multiple fast dependent operations within one clock cycle, whereas currently our analysis cannot estimate the impact of this technique. Finally, resource estimation could be much less accurate, because optimizations performed by RTL allow a single array of LUTs to be used to implement multiple arithmetic/logic operations. It is notable that because floating-point operations require multiple cycles to complete and their circuits are compactly designed, the latter two limitations apply specifically to integer programs, and have negligible impact on floating-point programs.

**Other practical considerations.** Finally, we may consider limiting perspectives if overcome could make the resulting tool much more usable. Firstly, SOAP does not scale well for programs larger than a few loops, which still requires the user to manually partition the program into smaller snippets to be optimized individually and tune parameters to trade quality for optimization speed. Another limitation is that programs cannot be optimized without knowledge about the input variables. Herbie [Pan+15] makes no assumption about the input space, and can nevertheless optimize arithmetic expressions,
by splitting the input space into multiple parts to be evaluated by expressions that are optimized for different regions.

6.2 Final Remarks

This thesis adapts existing techniques such as accuracy, latency and resource usage analysis, and further introduces novel approaches, e.g. MIR and efficient equivalence discovery, and delivers them in a unified framework. The functional equivalence relaxation paradigm is relatively under-explored, because these optimizations are often highlighted as unsafe by the HLS tools, as they cannot analyze the numerical implications of these optimizations. HLS tools therefore have very limited optimization options based on this particular concept. With the constructive results produced by this thesis, optimizations based on our concept can not only raise performance measures, but also result in even safer implementations as we improve numerical accuracies. The equivalence discovery algorithm in tandem with MIRs could have great potential in compiler optimization based on our concept. Furthermore, since machine learning algorithms are error-resilient [Les+11; Kim+09; HB91; ZS03], the methods demonstrated in this thesis have promising capabilities to improve their resource usage, latency and accuracy.
Bibliography


Sound Acceleration of Equivalent Expression Discovery

The functions used in Section 3.3.2 can sometimes be slow to compute using a naïve implementation. By using abstract interpretation and the properties of certain equivalent expression generator (EEG) functions, we can further accelerate the computation. We start by defining a property of the EEG functions used in Section 3.3.2 known as \( \cup \)-distributive, then propose a new algorithm to accelerate the computation of \( \text{cl}_N f(\epsilon) \), where \( f \) is a \( \cup \)-distributive EEG.

**Corollary A.1.** By the definition of \( \Rightarrow \) in (3.22), it is clear that \( \Rightarrow \) is \( \cup \)-distributive.

We then continue to prove that the algorithm \( \text{CLOSE}(f, N, \epsilon) \) in Figure 3.1 indeed computes \( \text{cl}_N f(\epsilon) \). Firstly, we prove the following lemma:

**Lemma A.1.** \( \text{cl}_N f(\epsilon) = \epsilon \cup f (\text{cl}_{N-1} f(\epsilon)) \) for any \( \cup \)-distributive EEG function \( f \).

**Proof.** Following (3.24), \( \text{cl}_N f(\epsilon) = f^0(\epsilon) \cup f^1(\epsilon) \cup \cdots \cup f^N(\epsilon) \). Because \( f \) is \( \cup \)-distributive, we apply distributivity to the right-hand side to derive:

\[
\text{cl}_N f(\epsilon) = \epsilon \cup f \left( f^0(\epsilon) \cup f^1(\epsilon) \cup \cdots \cup f^{N-1}(\epsilon) \right), \tag{A.1}
\]

which equals to \( \epsilon \cup f (\text{cl}_{N-1} f(\epsilon)) \) by definition. \( \square \)

Then this allows us to deduce that the algorithm indeed computes \( \text{cl}_N f(\epsilon) \):

**Theorem A.1.** In the algorithm in Figure 3.1, at iteration \( n \), the set of equivalent expressions \( s_n \) computes exactly \( \text{cl}_n f(\epsilon) \), if \( f \) is a \( \cup \)-distributive EEG.
Proof. We start by assuming that at iteration $m > 0$, $s_m = \text{cl}_m f(\epsilon)$, and we prove this equality still holds if substitute $m$ with $m + 1$. From the algorithm, we can deduce:

$$s_{m+1} = s_m \cup s_{m+1}' = s_m \cup (f(s'_m) - s_m) = s_m \cup f(s'_m) = s_m \cup f(f(s'_{m-1}) - s_m) .$$

We substitute $s_m$ using Lemma A.1 to get:

$$s_{m+1} = \epsilon \cup f(s_{m-1}) \cup f(f(s'_{m-1}) - s_{m-1}) .$$

Using distributivity of $f$ over $\cup$ and the iteration $m$ of the algorithm, we can derive:

$$s_{m+1} = \epsilon \cup f(s_{m-1} \cup (f(s'_{m-1}) - s_{m-1})) = \epsilon \cup f(s_m) .$$

Finally, we make use of the assumption $s_m = \text{cl}_m f(\epsilon)$, followed by Lemma A.1 to show:

$$s_{m+1} = \epsilon \cup f(\text{cl}_m f(\epsilon)) = \text{cl}_{m+1} f(\epsilon) .$$

It is trivial that $s_0 = \epsilon = \text{cl}_0 f(\epsilon)$, by induction, $s_n = \text{cl}_n f(\epsilon)$ thus holds for all $n \in \mathbb{N}$. □

Unfortunately, in the alternative method `greedy_trace`, we cannot make use of the efficient algorithm in Figure 3.1 to compute $\text{cl}_N (fr \circ \triangleright_k)$ directly. The reason is that in general, $fr \circ \triangleright_k$ is not $\cup$-distributive. Imagine two equivalent expressions $e_1$ and $e_2$, and $e_1$ strictly dominates $e_2$, i.e. $e_1$ is better than $e_2$ in terms of accuracy and resources, then we can observe that:

$$fr(\{e_1\} \cup \{e_2\}) = fr(\{e_1, e_2\}) = \{e_1\} , \quad \text{(A.2)}$$

which is not equal to:

$$fr(\{e_1\}) \cup fr(\{e_2\}) = \{e_1\} \cup \{e_2\} . \quad \text{(A.3)}$$

To resolve this, following abstract interpretation discussed in Section 2.3 of Chapter 2, we can introduce a new abstract domain to the power set of equivalent expressions. This
new domain reduces the computation effort of equivalent expression discovery, because it is a simpler domain with fewer elements than the power set.

The abstract domain, represented by $\langle AExpr^\sharp, \sqsubseteq \rangle$, can be inductively defined using the following Galois connection:

$$
\langle \wp (AExpr_{\sqsubseteq}), \sqsubseteq \rangle \not\overset{\gamma}{\underset{\alpha}{\leftrightarrow}} \langle AExpr^\sharp_{\sqsubseteq}, \sqsubseteq \rangle,
$$

where its abstraction and concretization functions are defined as follows:

$$
\alpha(\epsilon) = \fr(\epsilon), \quad \gamma(\epsilon^\sharp) = \epsilon^\sharp.
$$

Specifically, $\alpha$ is identical to $\fr$, and $\gamma$ is simply the identity function, i.e. it returns the input as its output.

The partial ordering on them ($\sqsubseteq$) and the join operator ($\sqcup$) can then be defined inductively as follows using the Galois connection:

$$
\epsilon^\sharp_1 \sqsubseteq \epsilon^\sharp_2 := (\epsilon^\sharp_1 \sqcup \epsilon^\sharp_2) = \epsilon^\sharp_2,
$$

$$
\epsilon^\sharp_1 \sqcup \epsilon^\sharp_2 := \alpha \left( \gamma \left( \epsilon^\sharp_1 \right) \sqcup \gamma \left( \epsilon^\sharp_2 \right) \right).
$$

Additionally, an abstract variant of an arbitrary EEG $f$ can also be inductively defined:

$$
f^\sharp(\epsilon^\sharp) = \alpha \circ f \circ \gamma \left( \epsilon^\sharp \right) = \fr \left( f \left( \epsilon^\sharp \right) \right),
$$

Finally, the abstract variant of $\cl_N f$ can be proposed with a simple modification to the algorithm in Figure 3.1, by replacing $\cup$ with $\sqsubseteq$ in $s_i \leftarrow s_{i-1} \cup s'_i$, we now have an abstract closure function:

$$
\cl_N^\sharp f^\sharp(\epsilon) \sqsubseteq \bigcup_{n \in \mathbb{N}} f^{\sharp n}(\epsilon),
$$

that operates with an abstract EEG $f^\sharp$. The correctness of this formulation can be proved by adapting Theorem A.1 to the modified algorithm with the property:

$$
f^\sharp \left( \epsilon^\sharp_1 \sqcup \epsilon^\sharp_2 \right) \sqsubseteq f^\sharp \left( \epsilon^\sharp_1 \right) \sqcup f^\sharp \left( \epsilon^\sharp_2 \right).
$$
The alternative method, `greedy_trace` which computes $cI_N \triangleright V_k$, where $V_k = f \circ \triangleright k$ can finally be accelerated using the modified algorithm.
Formal Definitions of Equivalent MIR Discovery

In Section 4.5 of Chapter 4 informally explained how equivalent semantic expressions and MIRs can be discovered. In this appendix we formally define the equivalent discovery procedure for all newly introduced operators in Section 4.2, i.e. the ternary conditional, composition, and fixpoint operators, and also extend this definition to MIRs in a similar fashion.

Formally, we extend the optimization function \( O[:]: M \rightarrow \Sigma_{E^t} \rightarrow M \), where \( M = \text{SemExpr} \cup \text{MIR} \), proposed in Section 3.3 of Chapter 3 to these above structures. For ternary conditional operators, we have:

\[
O\left[ b \overset{?}{\rightarrow} e_1 \overset{?}{\rightarrow} e_2 \right]^{\sigma^t} = f_{\sigma^t} \left( \left\{ \frac{\{ b' \in O[b] \sigma^t, e'_1 \in O[e_1] \sigma^t |_{b'}, e'_2 \in O[e_2] \sigma^t |_{b'} \} }{b'} \right\} \right), \quad \text{(B.1)}
\]

where the function \( f_{\sigma^t} \) is defined in Section 3.3.4 of Chapter 3.

Similarly, the function can also be defined for the composition operator:

\[
O\left[ e \overset{*}{\rightarrow} \mu \right]^{\sigma^t} = fr \left( \left\{ \frac{\{ e' \in O[e] \left( E_m [\mu'] \sigma^t \right), \} }{e'} \right\}^{\sigma^t} \right), \quad \text{(B.2)}
\]

where \( fr(e, \sigma^t) \) computes the Pareto-optimal set of equivalent semantic expressions from the initial set \( e \), using the program state \( \sigma^t \) to evaluate the quality metrics (i.e. round-off errors and resource utilization).
In addition, a formal mathematical definition of the process of discovering equivalent fixpoint expressions is:

\[
O \left[ b \ \xrightarrow{\text{fix}} \ \mu \ x \right] \sigma^\sharp = \text{fr} \left\{ \left\{ \left\{ \left\{ \left\{ \sigma^\sharp_k = E^\sharp \left[ b \ \xrightarrow{\text{fix}} \ \mu \ x \right] \sigma^\sharp, \ \right\} \right\} \right\} \right\} \right\}
\]

where \( K \) is the partial unroll factor limit. We set \( K = 3 \) in the experimental results discussed in Section 4.7.

Finally, MIRs can also be optimized in a similar fashion, by recursively discovering equivalent expressions within it:

\[
O \left[ \mu \right] \sigma^\sharp = \text{fr} \left\{ \left\{ \left\{ \left\{ \left\{ \left\{ \sigma^\sharp, \ \right\} \right\} \right\} \right\} \right\} \right\}
\]

Appendix B Formal Definitions of Equivalent MIR Discovery
Benchmark Source Code

In Chapters 4 and 5 we explore the experimental results of several numerical programs as our benchmark examples. This appendix contains the source code of the benchmark suite used.

```c
#pragma soap in float x=[0, 20]
#pragma soap out x

while (x > 1.0) {
    x = 0.9f * x;
}

Figure C.1. simple

#pragma soap in \n    int n=[10, 20], float x=[-0.1, 0.1], float y=[0, 1]
#pragma soap out z

float a = 1;
int b = 1;
float p = 1;
float z = 0.0f;
for (int i = 0; i < n; i++) {
    a = -a;
    b *= (2 * i + 1) * (2 * i);
    p *= (x + y) * (x + y);
    z += (a / b) * p;
}

Figure C.2. taylor
```
#pragma soap in \
    float a0=[0, 0.2], float a1=[0, 0.2], \
    float a2=[0.0, 0.2], float b0=[0, 0.2], \
    float b1=[0, 0.2], float b2=[0.0, 0.2], \
    float x=[0, 1], int n=20
#pragma soap out y

float x1 = 0.0f, x2 = 0.0f;
float y1 = 0.0f, y2 = 0.0f;
float y = x;
for (int i = 0; i < n; i++) {
    float yt = y;
    y = b0 * x + b1 * x1 + b2 * x2 +
      a0 * y + a1 * y1 + a2 * y2;
    x2 = x1;
    x1 = x;
    y2 = y1;
    y1 = yt;
}

Figure C.3. filter

#pragma soap in \
    float u=[0.0, 1.0], float w=[0.0, 1.0], \
    int n=[0, 20], float dt=[0.1, 0.1]
#pragma soap out u, v

float u;
float v = 0.0f;
for (int i = 0; i < n; i++) {
    float u0 = u + v * dt;
    v -= w * u * dt;
    u = u0;
}

Figure C.4. euler
```c
#define n 20
#pragma soap in
    float kp=[9, 10], float ki=[0.5, 0.7], float kd=[0, 3],
    float dt=[0.2, 0.2], float m=8.0, float c=5.0,
#pragma soap out m

float i = 0.0f, e0 = 0.0f;
float m, e, d, r;
for (int j = 0; j < n; j++) {
    e = c - m;
    i += ki * dt * e;
    d = kd * (e - e0) / dt;
    r = kp * e + i + d;
    e0 = e;
    m += 0.01f * r;
}

Figure C.5. pid

#define N 4096
#pragma soap in float x[N] = [0.0, 1.0]
#pragma soap out sum

float sum = 0;
for (int i = 0; i < N; i = i + 1) {
    sum = sum + x[i];
}

Figure C.6. sum

#define N 4096
#pragma soap in
    float x[100] = [0.0, 1.0], float z[100] = [0.0, 1.0]
#pragma soap out q

float q = 0.0f;
for (int k = 0; k < N; k++) {
    q = q + z[k] * x[k];
}

Figure C.7. dotprod
```c
#define loop 1024
#define n 1024
#pragma soap in 
    float x[n] = [0.0, 1.0], float y[n] = [0.0, 1.0], 
    float z[n] = [0.0, 1.0]
#pragma soap out x

int l; int i;
for ( l=1 ; l<=loop ; l++ ) {
    for ( i=1 ; i<n ; i++ ) {
        x[i] = z[i]*( y[i] - x[i-1] );
    }
}

Figure C.8. tridiag

// D := alpha*A*B*C + beta*D

#define N 1024
#pragma soap in 
    float A[N][N] = [0.0, 1.0], float B[N][N] = [0.0, 1.0], 
    float C[N][N] = [0.0, 1.0], float D[N][N] = [0.0, 1.0], 
    float tmp[N][N] = [0.0, 1.0]
#pragma soap out D

int i; int j; int k;
float alpha = 32412;
float beta = 2123;
for ( i = 0; i < N; i++ )
    for ( j = 0; j < N; j++ ) {
        tmp[i][j] = 0;
        for ( k = 0; k < N; ++k )
            tmp[i][j] += alpha * A[i][k] * B[k][j];
    }
for ( i = 0; i < N; i++ )
    for ( j = 0; j < N; j++ ) {
        D[i][j] *= beta;
        for ( k = 0; k < N; ++k )
            D[i][j] += tmp[i][k] * C[k][j];
    }

Figure C.9. 2mm
```
// G = (A * B) * (C * D)
#define N 1024
#pragma soap in
    float A[N][N] = [0.0, 1.0], float B[N][N] = [0.0, 1.0],
    float C[N][N] = [0.0, 1.0], float D[N][N] = [0.0, 1.0],
    float E[N][N] = [0.0, 1.0], float F[N][N] = [0.0, 1.0],
    float G[N][N] = [0.0, 1.0]
#pragma soap out G
int i; int j; int k;
for (i = 0; i < N; i++) /* E := A*B */
    for (j = 0; j < N; j++) {
        E[i][j] = 0;
        for (k = 0; k < N; ++k)
            E[i][j] += A[i][k] * B[k][j];
    }
for (i = 0; i < N; i++) /* F := C*D */
    for (j = 0; j < N; j++) {
        F[i][j] = 0;
        for (k = 0; k < N; ++k)
            F[i][j] += C[i][k] * D[k][j];
    }
for (i = 0; i < N; i++) /* G := E*F */
    for (j = 0; j < N; j++) {
        G[i][j] = 0;
        for (k = 0; k < N; ++k)
            G[i][j] += E[i][k] * F[k][j];
    }

Figure C.10. 3mm

#define N 4000
#pragma soap in
    float A[N][N] = [0.0, 1.0], float x[N] = [0.0, 1.0],
    float y[N] = [0.0, 1.0], float tmp[N] = 0
#pragma soap out y
int i; int j;
for (i = 0; i < N; i++) {
    tmp[i] = 0;
    for (j = 0; j < N; j++)
        tmp[i] = tmp[i] + A[i][j] * x[j];
    for (j = 0; j < N; j++)
        y[j] = y[j] + A[i][j] * tmp[i];
}

Figure C.11. atax
#define N 4000
#pragma soap in \
    float A[N][N] = [0.0, 1.0], float s[N] = [0.0, 1.0], \
    float q[N] = 0, \
    float p[N] = [0.0, 1.0], float r[N] = [0.0, 1.0]
#pragma soap out q

int i; int j;
for (i = 0; i < N; i++)
    s[i] = 0;
for (i = 0; i < N; i++)
{
    q[i] = 0;
    for (j = 0; j < N; j++)
    {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}

Figure C.12. bicg

// C := alpha*A*B + beta*C
#define N 1024
#pragma soap in \
    float C[N][N] = [0.0, 1.0], float A[N][N] = [0.0, 1.0], \
    float B[N][N] = [0.0, 1.0]
#pragma soap out C

int i; int j; int k;
float alpha = 32412;
float beta = 2123;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
    {
        C[i][j] *= beta;
        for (k = 0; k < N; ++k)
            C[i][j] += alpha * A[i][k] * B[k][j];
    }

Figure C.13. gemm
// Vector Multiplication and Matrix Addition
#define N 1024
#pragma soap in
  float A[N][N] = [0.0, 1.0],
  float u1[N] = [0.0, 1.0], float v1[N] = [0.0, 1.0],
  float u2[N] = [0.0, 1.0], float v2[N] = [0.0, 1.0],
  float w[N] = 0, float x[N] = 0,
  float y[N] = [0.0, 1.0], float z[N] = [0.0, 1.0]
#pragma soap out w
int i; int j;
float alpha = 43532;
float beta = 12313;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    x[i] = x[i] + beta * A[j][i] * y[j];
for (i = 0; i < N; i++)
  x[i] = x[i] + z[i];
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    w[i] = w[i] + alpha * A[i][j] * x[j];

Figure C.14. gemver

// Matrix Vector Product and Transpose
#define N 1024
#pragma soap in
  float x1[N] = [0.0, 1.0], float x2[N] = [0.0, 1.0],
  float y_1[N] = [0.0, 1.0], float y_2[N] = [0.0, 1.0],
  float A[N][N] = [0.0, 1.0]
#pragma soap out x1, x2
int i; int j;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    x1[i] = x1[i] + A[i][j] * y_1[j];
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    x2[i] = x2[i] + A[j][i] * y_2[j];

Figure C.15. mvt


```c
#define N 1000
#define TSTEPS 20
#pragma soap in float A[N][N] = [0.0, 1.0]
#pragma soap out A

int t; int i; int j;
for (t = 0; t < TSTEPS; t++)
    for (i = 1; i < N - 1; i++)
        for (j = 1; j < N - 1; j++)

Figure C.16. seidel

// Symmetric rank-2k operations
#define N 1024
#pragma soap in \ 
    float alpha = [0.0, 1.0], float beta = [0.0, 1.0], \ 
    float A[N][N] = [0.0, 1.0], float B[N][N] = [0.0, 1.0], \ 
    float C[N][N] = [0.0, 1.0]
#pragma soap out C

int i; int j; int k;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++) {
            C[i][j] += alpha * A[i][k] * B[j][k];
            C[i][j] += alpha * B[i][k] * A[j][k];
        }

Figure C.17. syr2k
```