Numerical Program Optimization for High-Level Synthesis

Xitong Gao, George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
London SW7 2AZ, United Kingdom
{xi.gao08, g.constantinides}@imperial.ac.uk

ABSTRACT

This paper introduces a new technique, and its associated open source tool, SOAP2, to automatically perform source-to-source optimization of numerical programs, specifically targeting the trade-off between numerical accuracy and resource usage as a high-level synthesis flow for FPGA implementations. We introduce a new intermediate representation, which we call metasemantic intermediate representation (MIR), to enable the abstraction and optimization of numerical programs. We efficiently discover equivalent structures in MIRs by exploiting the rules of real arithmetic, such as associativity and distributivity, and rules that allow control flow restructuring, and produce Pareto frontiers of equivalent programs that trades off LUTs, DSPs and accuracy. Additionally, we further broaden the Pareto frontier in our optimization flow to automatically explore the numerical implications of partial loop unrolling and loop splitting. In real applications, our tool discovers a wide range of Pareto optimal options, and the most accurate one improves the accuracy of numerical programs by up to 65%.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Design Aids—Optimization

General Terms

High-Level Synthesis, Numerical Accuracy, Round-off Error

1. INTRODUCTION

Floating-point numerical algorithms are essential to many applications. It is often desirable to compute their results as accurately as possible. Typically, computations are performed following the IEEE 754 standard [1]. Due to a finite number of values that can be represented in floating-point arithmetic, numerical algorithms generally always have round-off errors. Therefore, equivalence rules such as *associativity* $(a + b) + c \equiv a + (b + c)$, and *distributivity* $(a + b) \times c \equiv a \times c + b \times c$ for real arithmetic no longer hold under floating-point arithmetic. This allows us to exploit these equivalence relations, to automatically generate different equivalent expressions from the same arithmetic expression. For the optimization of FPGA implementations, these equivalent expressions can then be selected for the optimal trade-off between resource usage when synthesized into circuits, that is, the number of look-up tables (LUTs) and digital signal processing (DSP) elements utilized, and accuracy when evaluated using floating-point computations. As an example, we optimize the program "`if` $(x < 1)$ `then` $(x := (x + y) + 0.1)$ `else` $(x{:=}x + (y + 0.1))$" using our tool with single-precision floating-point format, given an input $x \in [0, 100]$ and $y \in [0, 2]$. On the one hand, we find that it is most accurate when the subexpression $(x + y) + 0.1$ is written as $(x + 0.1) + y$, because the subexpression is only evaluated when $x < 1$, our tool infers a tighter bound on x, $[0, 1]$, to optimize it. On the other hand, the original program uses fewest resources when subexpressions are shared and the `if` statement is eliminated, *i.e.* "$x := x + (y + 0.1)$". This kind of optimization generates a Pareto optimal set of implementations. A naïve strategy to search for the Pareto optimal implementations is to discover all possible equivalent expressions. However, this would result in combinatorial explosion and become intractable even for very small expressions [4, 6, 10]. To remedy this, Gao *et al.* [4] proposed a novel approach, known as SOAP, to significantly reduce the space and time complexity to produce a subset of the Pareto frontier.

In this paper, we propose a new general *program* optimization technique for numerical algorithms, which allows `if` statements as well as `while` loops, and developed its accompanied tool, SOAP2, to enable the joint optimization of accuracy and resource usage, as well as the trade-off between these performance metrics. The tool performs source-to-source optimization of numerical programs targeting FPGAs, and generate implementations that trade off resource usage and numerical accuracy.

Our main contributions in this paper are as follows:

1. A new intermediate representation (IR) of the behaviour of numerical programs, designed to be manipulated and analyzed with ease; a new framework of numerical program transformations to enable the back and forth translation between the program and the IR, which preserves the semantics of the original program.
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.
3. A new tool, SOAP2, which trades off resource usage and accuracy by providing a safe, semantics-directed

and flexible optimization targeting numerical programs for high-level synthesis.

2. RELATED WORK

There are many existing techniques that trade off resource usage and numerical accuracy in circuits. Wordlength optimization is a classical method for trading off precision and performance by minimizing data path wordlengths [3]. In the high-level synthesis (HLS) community, a technique is developed in [2] to automatically trade-off the data path wordlengths of algorithms that contain loops. However, these methods changes datapath size by varying *precisions* of arithmetic operators, whereas there is currently little work on performing structural improvements to datapaths in HLS, except for SOAP's arithmetic expression optimization [4].

With regard to the structural optimization of only arithmetic expressions without control structures, currently there are only a handful of tools that could optimize by *truly restructuring*, *i.e.* they exploit any of the three equivalence relations in real arithmetic, namely associativity, commutativity and distributivity. Many target either numerical accuracy [6], or performance metrics such as resources, latency, or throughput [8, 11, 10, 5]. Currently, SOAP is the only tool that could trade off area and accuracy in this category.

For true numerical software transformation with control structures, a method has been developed in [9] to utilize abstract interpretation and natural semantics [7]. However, this technique can neither unroll loops partially, nor optimize across loop boundaries. Furthermore, they specifically optimize numerical accuracy, and found that frequently this technique produces much slower implementations, while we also consider performance, by improving both accuracy and the resource usage of programs.

As none of the above-mentioned looks at the multiple-objective optimization of numerical programs, we are the first to propose a tool that performs a semantics-directed and truly restructuring program transformation, which optimizes not only arithmetic expressions, but also numerical programs, for the trade-off between numerical accuracy and resource usage when synthesized to FPGAs.

3. LANGUAGE DEFINITION

Before we discuss program transform, we first introduce NumImp, a simple imperative language that supports arithmetic and Boolean expressions, `if` statements, as well as `while` loops. Our program transformation optimizes NumImp programs. Our language allows numerical data types `int` and `float`, respectively integer and floating-point types.

As a simple example, the program in (1) computes an approximate value of $\pi^2 a/6$. 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.

$$\begin{aligned}
&\texttt{input}\ (a : [0.0, 1.0], n : [10, 20]); \texttt{output}\ (y);\\
&x := 0; y := 0.0;\\
&\texttt{while}\ (x < n)\ \texttt{do}\ (\\
&\quad x := x + 1;\\
&\quad y := y + a/(x \times x);\\
&);
\end{aligned} \tag{1}$$

Despite the simplicity of NumImp, it includes all the features of a full programming language rather than an expression language used in prior work [4]. Additional language features, for example, array and matrix types, and also power, exponentiation and logarithm operators, can be added with few changes to our method.

4. PROGRAM TO MIR

The first step of our approach is to analyze each program return value into a metasemantic intermediate representation (MIR), which is a mapping from program variables to expressions, with additional operators to represent control structures. We call these expressions *semantic expressions*. 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 dataflow dependencies and keeps only computations that contribute to the final results.

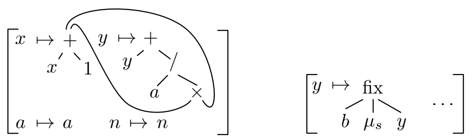
We work with the MIR as an abstraction of the program because the discovery of equivalent structures can be greatly simplified. For instance, the program "$x := 1; y := 2$" is the same as "$y{:=}2; x{:=}1$" because interleaving of non-dependent statements does not change program semantics. If we were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

In SOAP2, we have automated the above analysis. In this section we explain how it is performed in detail, by manually analyzing the simple example in (1) into an MIR.

Our analysis is compositional, which means that it starts by analyzing the individual statements in the program into MIRs, and these MIRs are then combined to form the program's MIR. We begin by analyzing the first statement, $s_1 =$ "$x{:=}0$". This assigns the value 0 to the variable x, and does not affect other program variables. The MIR of s_1 is trivially analyzed into $\mu_1 = [x \mapsto 0, y \mapsto y, a \mapsto a, n \mapsto n]$. This means that executing the statement changes the value of x to 0, but not the values of other variables. We use the notation $x \mapsto e$, where x is a variable and e is an expression, to indicate x is paired with e. Hence the variable x is paired with the expression 0, while y is paired with an expression that contains only the variable itself, y. Similarly $s_2 =$ "$y := 0.0$" can be analyzed into $\mu_2 = [x \mapsto x, y \mapsto 0.0, a \mapsto a, n \mapsto n]$. These two MIRs μ_1 and μ_2 can then combined to form the MIR of "$s_1; s_2$", which is the MIR of the two statements executed in sequence. It is constructed by substituting the variables in the expressions of μ_2, with their corresponding expressions in μ_1. For instance, the expression x of μ_2 has the variable x, which is substituted to become 0, the corresponding expression of x in μ_1. Therefore the MIR of "$s_1; s_2$" is $\mu_0 = [x \mapsto 0, y \mapsto 0.0, a \mapsto a, n \mapsto n]$.

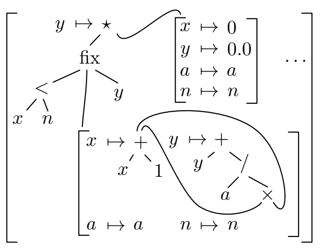
The two statements in the loop body "$x := x + 1; y := y + a/(x \times x)$" can be analyzed in a similar fashion, which produces the MIR in Figure 1a. Because the resulting MIR shares common structures, for example, the subexpression $(x + 1) \times (x + 1)$ and the expression of x share the same subexpression $x + 1$, a directed acyclic graph (DAG) is used to allow all common subexpressions to be shared among expressions in an MIR.

The next step is to analyze the `while` loop into an MIR. Because the output we care about is the value of y after program termination, for simplicity, we derive the MIR for the variable y only, and the expressions for other variables can be abbreviated as $\cdots$. The MIR for the loop is shown in Figure 1b. Syntactically, the fixpoint node "fix" is used to encode the necessary information about the `while` loop, where $b = x < n$ is the Boolean expression of the loop, μ_s is the MIR of the loop body (Figure 1a), and y signifies that y is the value at loop exit that we use as the evaluated value of the fixpoint expression. Mathematically, it is analogous to a recursive call in a functional programming language.



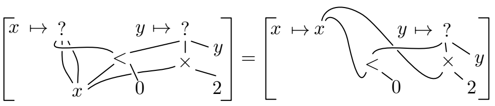
(a) The loop body (b) The `while` loop
Figure 1: The MIRs.

Finally, we combine Figure 1b and the MIR of the first two statements, μ_0, to arrive at the full MIR of the program. Here, we use $e \star \mu$, where e is an expression and μ is an MIR, to indicate that each variable x in the expression e are substituted with its corresponding expression in μ.



(2)

Although the example program (1) has no `if` statements, our analysis is capable of translating `if` statements into MIRs. As an example we consider the program "`if` $(x < 0)$ `then` $(y := x \times 2)$ `else skip`", where the set of program variables is $\{x, y\}$. The MIR of the program is shown in the left-hand side of (3). We introduce the conditional expression with the 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. Because both true- and false-expressions of x are the same, regardless of the truth value of $x < 0$, the two expressions will evaluated to the same value, we then further simplify it to become the one shown in the right-hand side.



(3)

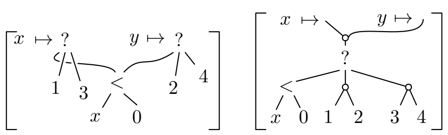
5. TRANSFORMATIONS

The next step is to use the analyses of accuracy and resource usage of equivalent structures in MIRs to efficiently discover optimized equivalent MIRs. We start by explaining how the accuracy and the resource usage of programs encoded in MIRs are analyzed. After this, we explain how our analyses of accuracy and resource usage can guide the efficient discovery of equivalent structures in MIRs.

5.1 Performance Analyses

In a typical program execution, values of variables, typically integers and floating-point values, 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. By comparison, not only do we propagate the values, but we use the static analysis approach of [4] to propagate the round-off errors associated with the numerical computations. For instance, if we know that floating point values a and b are respectively bounded by $[0, 1]$ and $[1, 2]$, and they have no round-off errors associated with them, then evaluating $a + b$ would result in a floating-point value that is bounded by $[1, 3]$, and we can compute the operation would result in a round-off error in the range of $[-1.19209304 \times 10^{-7}, 1.19209304 \times 10^{-7}]$. We use this method to propagate values and round-off errors starting from the leaves of a semantic expression, until we reach the root node, where we end up with the round-off error associated with the expression.

Expressions can have common subexpressions, we eliminate them when we construct DAGs from programs, which reduces resource usage. However, we can further merge multiple nodes into one to reduce resource usage. For example, the metasemantic analysis of the program "`if` $(x < 0)$ `then` $(x := 1; y := 2)$ `else` $(x := 3; y := 4)$" produces the following MIR in Figure 2a. 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 2a. 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 2b, to "bundle up" more than one conditional expressions, when they share the same Boolean expression. Finally, resource statistics can be estimated by accumulating LUT and DSP counts of each operator in the DAG.



(a) Before fusion (b) After fusion
Figure 2: The sharing of conditional expressions.

By combining both analyses, we can compare the accuracy and resource usage of each equivalent semantic expressions, and produce a Pareto frontier that trades off performance metrics including accuracy, and LUT and DSP counts.

5.2 Equivalent Structure Analysis

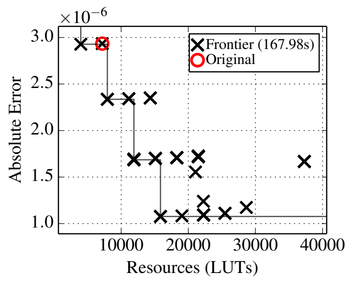
As we discussed earlier, discovering the full set of equivalent expressions by finding the transitive closure of the relations is infeasible because of combinatorial explosion. SOAP introduces a new method to drastically reduce the space and time complexity of discovering equivalent expressions, while achieving high quality optimizations [4]. We base our equivalent expression discovery on this method, but extend it to support additional program transform features. Our analysis starts by finding the set of equivalent expressions of the leaves of the DAG, which are the nodes themselves, for instance, a variable x has a set of equivalent expressions $\{x\}$. After this, these equivalent expressions from the child nodes are propagated to the parent node, to form a set of equivalent parent expressions. We then discover equivalences of expressions in this set, using not only rules such as associativity, distributivity, constant propagation and many others that are derived from SOAP [4], but also additional rules to flexibly transform control structures. These rules enables partial loop unrolling and extends arithmetic rules to conditional expressions. Using these rules would often create a large number of equivalent structures, which requires us to analyze the accuracy and resource usage of each, and propagate only those that are Pareto optimal, to significantly reduce the amount of time required for our equivalent structure analysis. We keep propagating until the root of the DAG is reached, and we arrive at a set of equivalent expressions of the original expression under optimization.

6. CODE GENERATION

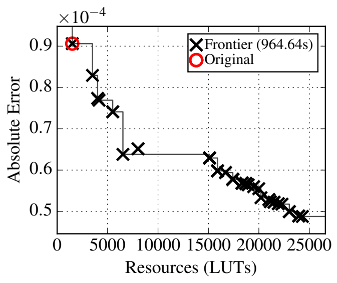
The final stage is to translate the optimized MIR back to a program in its original syntax. As discussed earlier, the MA produces an abstraction of the program, which means there are generally many ways of generating different programs from the same MIR. For this reason, certain heuristic optimizations 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. After this step, we perform a simple one-to-one mapping from MIRs to program code, using a breadth-first traversal of the MIR.

7. CONCLUSION

We use our tool, SOAP, to optimize the example program in (1), and produce the Pareto trade-off between the number of LUTs and the accuracy of the program in Figure 3a targeting an Altera Stratix IV device (EP4SGX530). Our tool automatically discovers that by partially unrolling the loop three times, although the program uses more resources, but it improves the accuracy by approximately 60%.



(a) Example program



(b) `euler`

Figure 3: The Pareto frontiers.

Figure 3b shows the optimization of a program that 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 ω^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 $\dot{x}$. 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%.

With the foundation and framework that we developed, our tool can be extended in the following ways. First, it can be trivially extended to support additional numerical data structure such as arrays and matrices. Secondly, the Pareto optimization can be extended to optimize the latencies of equivalent programs, as restructuring programs and partially unrolling loops could have a notable impact on the ability to pipeline program loops, especially when arrays are incorporated. Finally, fixed point representations, along with the interaction between our structural optimization and multiple wordlength optimization [3] could also generate a lot of interest from the HLS community.

Our tool is open source and can be downloaded freely at: `https://github.com/admk/soap`.

8. REFERENCES

[1] ANSI/IEEE. IEEE standard for floating-point arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society.

[2] D. P. Boland and G. A. Constantinides. Word-length Optimization Beyond Straight Line Code. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 105–114, New York, NY, USA, 2013. ACM.

[3] G. Constantinides, P. Cheung, and W. Luk. The multiple wordlength paradigm. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001. FCCM '01.*, pages 51–60.

[4] X. Gao, S. Bayliss, and G. Constantinides. SOAP: Structural optimization of arithmetic expressions for high-level synthesis. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 112–119, Dec 2013.

[5] A. Hosangadi, F. Fallah, and R. Kastner. Factoring and eliminating common subexpressions in polynomial expressions. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design, ICCAD '04*, pages 169–174, 2004.

[6] A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *Proceedings of the 19th International Conference on Static Analysis, SAS '12*, pages 75–93.

[7] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, 1987.

[8] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, page 75.

[9] M. Martel. Program transformation for numerical precision. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, PEPM '09*, pages 101–110. ACM, 2009.

[10] C. Mouilleron. *Efficient Computation with Structured Matrices and Arithmetic Expressions.* PhD thesis, Ecole Normale Supérieure de Lyon-ENS LYON, 2011.

[11] Xilinx. Vivado Design Suite User Guide—High-Level Synthesis, 2014.