Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis

Xitong Gao, John Wickerson, and George A. Constantinides
Circuits and Systems Research Group, Department of Electrical and Electronic Engineering
Imperial College London, United Kingdom
{xi.gao08, j.wickerson, g.constantinides}@imperial.ac.uk

ABSTRACT

Loops are pervasive in numerical programs, so high-level synthesis (HLS) tools use state-of-the-art scheduling techniques to pipeline them efficiently. Still, the run time performance of the resultant FPGA implementation is limited by data dependences between loop iterations. Some of these dependence constraints can be alleviated by rewriting the program according to arithmetic identities (*e.g.* associativity and distributivity), memory access reductions, and control flow optimizations (*e.g.* partial loop unrolling). HLS tools cannot safely enable such rewrites by default because they may impact the accuracy of floating-point computations and increase area usage. In this paper, we introduce the first open-source program optimizer for automatically rewriting a given program to optimize latency while controlling for accuracy and area. Our tool, SOAP3, 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, our tool has generated programs that enjoy up to a 12× speedup, with a simultaneous 7× increase in accuracy, at a cost of up to 4× more LUTs.

1. INTRODUCTION

There are many reasons why FPGA implementations of numerical algorithms are 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 [1]. Great advances have been made in this area recently, and the output from HLS tools is nowadays competitive with hand-crafted designs [2].

Numerical C programs typically spend most of their time in loops. For this reason, HLS tools adopt state-of-the-art scheduling algorithms to synthesize loops to run as fast as possible [3]. This is achieved by pipelining them to maximally exploit parallelism across loop iterations. However, their ability to perform pipelining 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 conventional rules (*e.g.* partial loop unrolling and array access pattern changes) to enable much more efficiently pipelined RTL designs. A simple example of this is the summation of all elements in an array:

```
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, and we will see in Sec. 8 that more efficient implementations are possible.

Unfortunately, in the presence of floating-point arithmetic, these program transformations could affect numerical accuracy. 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. The difference between the actual result in real arithmetic and the rounded result is known as the *round-off error*. Round-off errors, when accumulated, can have a devastating effect on numerical accuracy [4]. 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) * c`" and "`a * c + b * c`" in their programs.

Experienced engineers apply 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. 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. A straightforward process therefore does not exist 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, and thus make little of them when restructuring floating-point data-paths. For instance, *Vivado HLS* (VHLS) [5] has only a simple *expression balancing* feature that uses as-

FPGA'16, February 21-23, 2016, Monterey, CA, USA
 ISBN 978-1-4503-3856-1/16/02…$15.00
DOI: http://dx.doi.org/10.1145/2847263.2847282

sociativity 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, we have developed a tool, SOAP3—a fully automatic source-to-source optimizer—that augments VHLS by optimizing a given program using these transformations. Our 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 digital signal processing (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, SOAP3 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.

In contrast to the expression balancing optimization pass in VHLS, SOAP3 *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. SOAP3 also considers 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.

Generating candidate optimizations naïvely would produce a combinatorial explosion, even for small input programs. For instance, a simple summation of n variables could have $(2n-1)!!$[1] equivalent expressions [6, 7]. We therefore base our optimizer on the open-source SOAP2 framework [6, 8], which specifically tackles the efficient discovery of equivalent structures in numerical programs, by intelligently pruning the set of candidates as it progresses up the input program's abstract syntax tree. We also exploit SOAP2's ability to analyze the numerical accuracy of a given program. To analyze the run time and resource utilization of a given program, we use a variant of the *iterative modulo scheduling* algorithm [9] that computes fundamental lower bounds of these metrics.

We evaluated SOAP3 on a suite of 11 programs from the Livermore Loops [10] and PolyBench [11] benchmark suites. Our tool obtained a wide selection of Pareto-optimized programs. Programs with the best latency obtained speedups of up to 12× (7× on average across the suite), and increases in accuracy of up to 7× (2.7× on average), while using up to 4× (2.5× on average) more LUTs. We were unable to decrease the resource utilization in any of the benchmarks, as they have no redundant computations.

[1] $(2n-1)!! = 1 \times 3 \times 5 \times \cdots \times (2n-1)$.

Our contributions

- We described how standard program equivalences that do not affect program behavior (*e.g.* partial loop unrolling, and rules that remove extraneous array accesses) can enable non-standard transformation rules (*e.g.* arithmetic rules) to significantly impact latency, resource usage and accuracy in a loop (Sec. 5.2).
- We significantly improved the performance of the algorithm for discovering equivalent programs through improved accuracy analysis (Sec. 6.3), graph partitioning, and intelligent pruning of optimization candidates (Sec. 5.1).
- We designed a new scheduling analysis that estimates the latency and resource usage of a given optimization candidate (Sec. 6).
- Incorporating the above-mentioned techniques, we developed the first optimizer to *automatically* and *safely* produce optimized programs (and subsequent RTL implementations with Vivado HLS) on the four-dimensional Pareto frontier of options that trade off run time, accuracy, and area (LUTs and DSP elements). Our improvements in latency are significantly better than the only ones produced by Vivado HLS's *unsafe* optimizations. We have evaluated SOAP3 on a suite of Livermore Loops and PolyBench benchmarks (Sec. 8).

2. MOTIVATION

Figure 1 gives an implementation of the 5-point Seidel stencil computation, modified from PolyBench's 9-point version [11], where initially all values in the array `A` are single-precision floating-point values between 0 and 1. It resembles the code frequently used in fluid dynamic simulations for solving partial differential equations and systems of linear equations.

```
for (int t = 0; t < 20; t++)
  for (int i = 1; i < 1023; i++)
    for (int j = 1; j < 1023; j++)
      A[i][j] = 0.2 * (A[i-1][j] +
        A[i][j-1] + A[i][j] +
        A[i][j+1] + A[i+1][j]);
```

Figure 1: An excerpt from the Seidel stencil [11]. The inter-iteration data dependence of the innermost loop is underlined.

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 1. Hence, it serves as our example to demonstrate the power of SOAP3.

VHLS generates a schedule where each iteration requires 49 cycles (the *depth*, D, of the loop), and there are 46 cycles between the starts of consecutive loop iterations (the *initiation interval*, II), as enforced by the data dependences above. The innermost loop runs for 1022 iterations (the *trip count*, N), so the overall latency of the innermost loop is $((N-1) \times II) + D = 47015$ 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 D to 42 cycles, while $N = 1022$ remains the same, thus $L = 28630$ cycles. The overall resource usage remains roughly the same. However, as we mentioned in Sec. 1, VHLS is not aware of the inter-iteration data dependence. Although enabling EB did produce a faster implementation, there is still room for improvement. We further discovered that if we partially unroll the loop, VHLS's EB did not improve the total run time, despite using a lot more resources. As we have explained in Sec. 1, EB only makes use of associativity, and does not make use of other equivalence rules. These limitations pose great restrictions on VHLS's ability to produce a significantly faster implementation. Most importantly, VHLS does not guarantee that this optimization will not result in catastrophic numerical inaccuracies.

We then use SOAP3 to automatically discover equivalent programs from the program in Figure 1. Because SOAP3 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, SOAP3 specifically applies transformations to alleviate the constraints on the inter-iteration dependence, and discovers that the innermost loop can be rewritten to minimize latency in the following form:

```
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])));
```

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%.

SOAP3 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:

```
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;
}
```

Further increasing the optimization effort, which enables the loop to be further unrolled, leads to a program that is 7× as fast as the original, but uses 2.8× as many LUTs. To summarize, in Table 1, we compare VHLS with EB, against one of the many implementations that we have explored using SOAP3 with the increased optimization effort. For each implementation, the round-off errors are computed using static analysis, a part of our optimization procedure. Our tool estimates latency in clock cycles and the total counts of LUTs and DSP elements, but we performed place-and-route manually for exact statistics.

	VHLS	VHLS with SOAP3	VHLS with EB
Clock Period (ns)	2.60	2.66	2.65
Latency (cycles)	961 k	135 k	585 k
Program Run Time (ms)	2.50	0.358	1.56
LUTs / DSP Elements	620/5	1778/8	623/5
Round-off Error	10.68 µ	4.31 µ	unknown

Table 1: Comparison between the fastest implementations. The three columns respectively shows the original program with loop pipelining enabled, what VHLS can achieve alone, and the capability of SOAP3. 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.

3. HIGH-LEVEL OVERVIEW

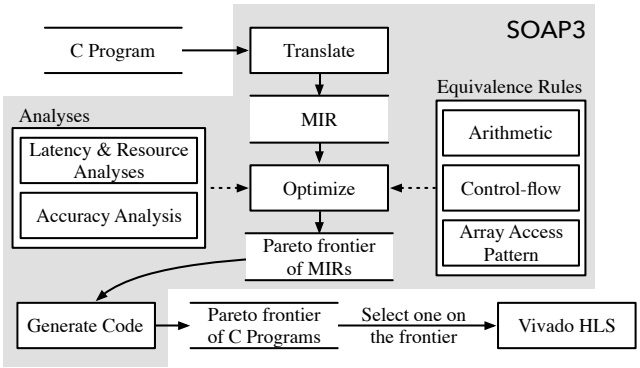


Figure 2: An overview of our automatic program optimization process. The shaded region shows our internal tool flow.

We start by introducing a high-level overview of our program optimization process (Figure 2). Our automatic optimization process starts by taking as an input, the original numerical program written in C, and translates it into a MIR (metasemantic intermediate representation graph). A MIR is a directed acyclic graph (DAG), and it serves as an abstract representation of the original program. It discards information about *how* a program is executed, which is dependent on how the program is structured in C, but retains the *effect* of program execution, keeping only the structure that leads to the final result. This procedure, explained in detail in Sec. 4, greatly reduces the number of program transformations we need to explore. We then discover equivalent MIRs using our efficient optimization procedure discussed in detail in Sec. 5. The optimized C programs can then be generated from the MIRs, using the SOAP2 framework's

code generation routines, to be synthesized in Vivado HLS to obtain RTL implementations.

Our optimization flow can be applied to nested **for** loops with constant loop bounds and step counts. It can be relatively easily extended to lift this restriction with polyhedral methods [12].

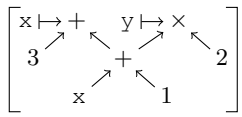
As we apply transformation rules to discover equivalent MIRs, we estimate latency, resource usage and analyze round-off errors for each MIR we have discovered. Non-Pareto-optimal MIRs—the ones with all three performance metrics (latency, resource usage and accuracy) worse than another MIR—are pruned immediately to keep the size of total MIRs discovered tractable. Sec. 6 explains in depth how we analyze latency, resource usage and accuracy.

4. INTERMEDIATE REPRESENTATION

There are infinitely many 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, the following two programs are equivalent, but syntactically different, as they carry out the same computations.

$$\begin{aligned} P_1 &: \texttt{x = x+1; y = 2*x; x = 3+x;} \\ P_2 &: \texttt{y = x+1; x = y; y = y*2; x = x+3;} \end{aligned} \tag{1}$$

In practice, it is desirable to eliminate as much as possible the need for these syntactic rewrites that do not affect our performance metrics. Following Gao *et al.*, we therefore perform transformations not on the program text directly, but on a DAG representation of the program called a MIR [8]. It expresses how each program variable is updated, 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, P_1 and P_2 can be automatically translated into an identical MIR:



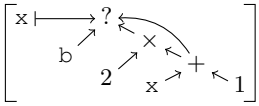
(2)

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.

MIRs also abstract the control structure 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:

```
x = x + 1; if (b) x = 2 * x;
```

can be represented in MIR form as follows:



(3)

MIRs are also capable of representing loops [8], but we do not exploit that in this paper, despite the centrality of loops to our work. When we optimize loop nests, we are specifically applying transformations to the kernels of the flattened loop nests. Therefore, we find that when analyzing the latency and resource usage of a loop, we need only have the *body* of the loop as a MIR.

4.1 Representing arrays

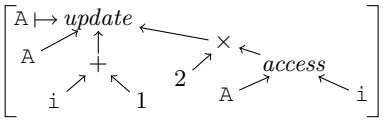
Gao *et al.* [8] did not include support for arrays in their original description of the MIR format. However, the examples that motivate our work all include arrays, so in this paper, we extend MIRs to be able to represent programs that use single- or multi-dimensional arrays.

In many imperative languages such as C, arrays are stateful objects, *i.e.* they 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*. Such behavior is not present in arithmetic expressions, many functional programming languages, SSA, as well as MIRs. This proves to be a challenge to us, because our efficient program optimization relies on recursively dividing 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 $update(A, \bar{x}, e)$ to return a new array that is the same as A but with (multi-dimensional) index $\bar{x}$ now containing e. Similarly, the function $access\,(A, \bar{x})$ returns the element of A at index $\bar{x}$. As a simple example, a loop body:

```
A[i + 1] = 2 * A[i];
```

can be translated into the following MIR:



(4)

The consequences of making arrays immutable are twofold. Firstly, we disallow pointer aliasing to keep the translation simple, *i.e.* "**float** *b = a;" is not allowed in the C code. 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 naïvely optimize MIRs. For instance, if A is an input array, consider the two expressions in a MIR, $update\,(A, \bar{x}, e)$ and $update\,(A, \bar{x}, e')$, where e, e' are equivalent. They respectively update the x-th element of the *same* immutable 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 "*update*" nodes using the method described in Sec. 5.

5. STRUCTURAL OPTIMIZATION

From a numerical program, we can generate a MIR using the translation in Sec. 4. 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.

5.1 Algorithm

As discussed in Sec. 1, even a small expression could have a huge number of equivalent ones. Exhaustively discovering all equivalent MIRs would result in combinatorial explosion of

```
function OPTIMIZE(op(e1, e2))
    s1 ← OPTIMIZE(e1),  s2 ← OPTIMIZE(e2)
    s' ← ∅,  s ← {op(e1', e2') | e1' ∈ s1 ∧ e2' ∈ s2}
    while s ≠ s' do
        s' ← s,  s'' ← ∅
        for r ∈ transformation_rules, e ∈ s do
            for e' where e ⇝r e' do
                s'' ← s'' ∪ {e'}
            end for
        end for
        s ← PRUNE(s'')
    end while
    return s
end function
```

Figure 3: The algorithm we used for the efficient discovery of equivalent structures in MIRs.

the number of equivalent MIRs in the search space. For this reason, we base ourselves on an algorithm from SOAP2 that searches efficiently, by discovering equivalences in a bottom-up hierarchy. In this section we discuss the improvements we have made to the algorithm which further increases the performance of this algorithm.

Our first contribution is that instead of optimizing the MIR immediately, we start by partitioning the MIR into multiple smaller sub-MIRs. In turn, each is optimized separately and generate a set of equivalent sub-MIRs. We then select combinations from these sub-MIRs 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.

Figure 3 shows the pseudocode of the optimization algorithm. It takes as an input a MIR graph, and produces a set of equivalent graphs that are estimated to be Pareto-optimal when converted into C programs and synthesized into circuits. Although this algorithm deals with a special case, *i.e.* a root node *op* with two child subtrees e_1, e_2, it can easily be generalized to an arbitrary number of child subtrees. Here, $e \overset{r}{\rightsquigarrow} e'$ means e' can be obtained by transforming part of the graph e in accordance with the transformation rule r. The next section discusses the transformation rules we used.

The algorithm starts by discovering equivalences in the leaves of a MIR, and progresses upwards for equivalent structures of the individual components that make up the graph, until the roots of the graph, where we have a set of MIRs equivalent to the original MIR. As it traverses through the MIR, the algorithm calculates the performance metrics at each node, using the analyses presented in the next section. Transformations that are not Pareto-optimal are immediately pruned from the search space, thus reducing the average complexity of the algorithm.

Our second contribution is the PRUNE function. We rely on this function to efficiently steer the direction of our Pareto frontier as we discover new candidates. It takes as an input the set of equivalent MIRs that we have discovered, and prunes MIRs in this set to reduce its size, keeping the number of MIRs discovered tractable. The SOAP2 framework prunes the MIRs that are Pareto-suboptimal, leaving only those that are on the Pareto frontier. However, because our Pareto frontier is 4D, there is a large increase in the number of Pareto-optimal MIRs. This Pareto pruning approach is no longer feasible for our benchmark examples. To tackle this, we introduce another step in PRUNE to further decrease the number of MIRs in the set by sampling. We developed a new sampling algorithm, inspired by Poisson-disk sampling algorithm [13], which samples the Pareto frontier by first randomly selecting one point, then iteratively growing the set of points by adding the neighbours from the point that are separated by at least a certain distance. We search by bisection for the distance that keeps 20% of all points in the Pareto frontier. 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 our improvements, the algorithm is significantly faster than the original optimization algorithm in SOAP2, with a 5-fold increase in speed, at a cost of fewer points on the Pareto frontier.

Arithmetic Rules	
Associativity	`(a + b) + c ⇝ a + (b + c)`
Commutativity	`a + b ⇝ b + a`
Distributivity	`(a + b) * c ⇝ a*c + b*c`
Negation	`a - b ⇝ a + (-b)`
Subtraction	`(a + b) - (a + b) ⇝ 0`
Const. prop.	`(a * b + c / d) * 0 ⇝ 0`
Division	`a / (5 / b) ⇝ a * b * 0.2`
Control Flow Restructuring Rules	
Partial loop unrolling	`for(i=0;i<1000;i++){Ci;} ⇝ for(i=0;i<1000;i+=2){Ci; Ci+1;}`
Access Reduction Rules	
Multiple reads	`x=A[i--]; y=A[i+1]; ⇝ x=A[i--]; y=x;`
Multiple writes	`A[i++]=x; A[i-1]=y; ⇝ A[i++]=y;`
Read after write	`A[i++]=x; y=A[i-1]; ⇝ A[i++]=x; y=x;`
Indep. accesses (where `i` ≢ `j`)	`A[i]=x; y=A[j]; ⇝ y=A[j]; A[i]=x;`

Table 2: Before-and-after examples to demonstrate the transformation rules we used. The arithmetic and control flow rules are inherited from Gao *et al.* [8]; the access reduction rules are introduced in this work.

5.2 Transformation Rules

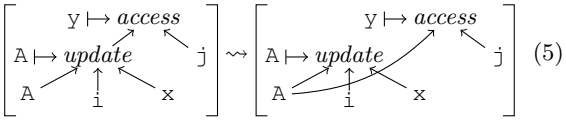
This section details the transformation rules used in our structural optimization algorithm in Figure 3. Each transformation rule on its own is not revolutionary, but for the first time, we bring them together to show a much better automatic structural optimization on the latency, resource usage and accuracy of numerical programs, than is possible using only a subset of them.

SOAP2 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 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 2, remove extraneous data dependences that arise after partial unrolling.

These rules, along with partial loop unrolling, mostly do not impact latency, because they are well studied in polyhedral loop dependence analysis, and tools such as LegUp 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, $\bar{i}, \bar{j}$ 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: $update\,(update\,(A, \bar{i}, e), \bar{i}, e') \rightsquigarrow update\,(A, \bar{i}, e')$.
- *Read after write*, eliminates a read from a location that has just been written: $access\,(update\,(A, \bar{i}, e), \bar{i}) \rightsquigarrow e$.
- *Independent accesses*, allows two array operations to be reordered if it can be proved that they never access the same location: $access\,(update\,(A, \bar{i}, e), \bar{j}) \rightsquigarrow access\,(A, \bar{j})$, if $\bar{i} \not\equiv \bar{j}$. We visualize this rule also in the following sample MIR transformation:



(5)

These access reduction rules may not seem powerful on their own, but when combined with other structural rules, they enable SOAP3 to detect dependences that can be removed in the MIR. This in turn allows more opportunities for the rules to further reduce loop latency. 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 could really make a difference in latency. Therefore all the rules in Table 2 are essential to the optimization of latency in numerical programs.

6. PERFORMANCE ANALYSIS

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

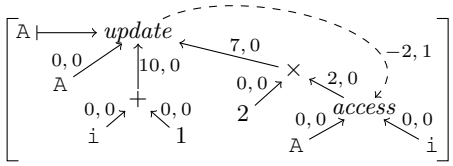
6.1 Latency Analysis

We measure the latency of a numerical program by estimating the total number of cycles required to execute it to completion. The most accurate estimate can be calculated with a complete scheduling of a numerical program. However, this would be computationally expensive, and would need to be repeated for tens of thousands of equivalent programs. Instead, our latency analysis computes the minimum initiation interval ($II_{\min}$) that must not be violated by any scheduling algorithm. (Recall from Sec. 1 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 [14], we compute $II_{\min}$ values using *iterative modulo scheduling* [9]. For our work, we have adapted this analysis to apply directly to MIRs. The structure of MIRs already captures intra-iteration data dependences; to this, we add extra latency information as attributes on the edges of MIRs, plus 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 $s \to t$, 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 $\langle l, d \rangle$ 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 (4) 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 integer linear programming problem for the dependence distance, and solves it using the Integer Set Library [12]. 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 following graph, and we call it a MIR with dependences (MIR^{dep}):



(6)

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 because the multiplier can reuse its output from the previous iteration as the input for the current 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 Vivado HLS, each array is usually translated into a dual-port RAM, which allows only two accesses per clock cycle [5], and thus constrains $II_{\min}$. For instance, for a loop to perform 3 accesses to a single array in each iteration, II must be greater than 1. This lower bound on II is known as *resource-based minimum initiation interval*, $II^{\text{res}}_{\min}$ [9]. It is defined as $\max_A \lceil n_A / r_A \rceil$, where A ranges over all arrays in the loop body, n_A is the number of accesses to the array A, and r_A is the maximum number of accesses allowed per cycle, which is 2 in our case.

The final step is to calculate an integer $II^{\text{rec}}_{\min}$ which is defined as $\max_c \lceil l(c)/d(c) \rceil$, where c ranges over all cycles in the MIR^{dep} graph, and we use $l(c)$ and $d(c)$ to respectively denote the sums of all latencies and dependence distances

of the edges in the path c. This value is known as the *recurrence-based minimum initiation interval* [9]. Because a typical MIR with array accesses could have a very large number of cycles, we efficiently search for an $II_{\min}$ using a modified Floyd–Warshall algorithm [9]. Finally, we estimate the total latency L_{est}, an approximation of the actual L, of the loop with:

$$L_{\text{est}} = (N-1)II_{\min} + D, \text{ where } II_{\min} = \max\left(II^{\text{rec}}_{\min}, II^{\text{res}}_{\min}\right)$$

where, recalling from Sec. 1, N is the maximum *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 programs in a bottom-up hierarchy, as described in Sec. 5, when an expression in a loop is optimized, its latency is estimated by scheduling its operations by using an As-Late-As-Possible (ALAP) [15] scheduling algorithm, where each operation is scheduled to the latest opportunity, while respecting the order of data dependences. 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 that is a fragment of a loop, our algorithm automatically shortens any paths between any pair 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 weights on dependent accesses with smaller dependence distances, because these impact the resulting loop II more than larger distances. We use the following formula as the analyzed latency value to guide the optimization for II for subexpressions in a loop, where Deps is a set of paths in the MIR, where each path is a sub-path of a cycle in the loop's MIR^{dep}:

$$L_{\text{est}} = \max_{p \in \text{Deps}} \frac{l(p)}{d(p)} \tag{7}$$

6.2 Resource Utilization Analysis

The hardware resource usage analysis of Gao *et al.* [8] 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 paper, we develop a new resource usage analysis that fully understands how resources are shared in an FPGA implementation of numerical programs.

We rely on the foundation of SOAP2, which counts the number $n_{\otimes}$ of each type of operation $\otimes$, 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 equation $a_{\otimes} = \lceil n_{\otimes}/II_{\min} \rceil$. For instance, if we know that a pipelined loop has $II_{\min} = 3$, and each iteration uses 6 multiplications, then we can compute that we need to synthesize at least 2 multipliers. Integer operators are typically not shared [16], so the number of operations is the number of allocated instances.

For straight-line code, non-pipelined loops, and consecutive loops, we use a simple ALAP scheduling [15] to estimate resource utilization.

Finally, we accumulate the number of LUTs and DSP elements for all allocated operators, which is the estimated resource utilization for the full program.

6.3 Accuracy Analysis

We build on the accuracy analysis of Gao *et al.* [8], which analyzes an upper bound of the absolute difference between the actual output of a numerical program and the expected output as if it is executed in real arithmetic. Because our benchmark suite consists of programs with large arrays, we further extend their work to support arrays, and 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 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. Additionally, because most of the loops in our benchmark programs consist of nested loops and have large iterations, we modified the analysis routine in SOAP2 to analyze only a small fraction of loop execution, and use our dependence analysis to detect whether errors are accumulated across iterations, in order to extrapolate the total round-off errors from the results.

7. TOOL USAGE

SOAP3 is a source-to-source optimizer that specifically targets numerical program statements written in a subset of standard C99. It introduces 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. SOAP3 supports arithmetic and Boolean expressions, assignment statements, **if** statements, **while** loops and **for** loops. The numerical data types we allow are **int** and **float**, as well as single- and multi-dimensional array types.

Figure 4 shows an example usage of SOAP3 in a C program. Note that it specifies the input values are respectively a two-dimensional array `A`, where its elements are single-precision floating point values between 0 and 1, and an integer `T` equal to 20. It also indicates the only output that we care about from this code is the resultant `A`.

```
#define N 1024
#pragma soap begin
#pragma soap in \
    float A[N][N] = [0, 1], int T = 20
#pragma soap out A
for (int t = 0; t < T; t++)
  for (int i = 1; i < N-1; i++)
    for (int j = 1; j < N-1; j++)
      A[i][j] = 0.2 * (A[i-1][j] +
        A[i][j-1] + A[i][j] +
        A[i][j+1] + A[i+1][j]);
#pragma soap end
```

Figure 4: An example C program that can be optimized with SOAP3.

Our tool is an open-source command-line utility, which only requires the user to provide a program written in C extended with the above **#pragma** statements. The Pareto optimal programs are all automatically generated by our tool, each is accompanied with our estimations of its latency and resource usage, and an analyzed bound on round-off errors. These programs can then be given to Vivado HLS to be synthesized into circuits.

Name	DSPs	LUTs	ratio	Error	ratio	Clock (ns)	Latency (cycles)	(s)	ratio
sum	2	303	0.257	914 µ	7.93	2.54	41.0 k	104 µ	12.8
	4	1181		1.15 µ		2.54	3.21 k	8.17 µ	
dotprod	5	411	0.231	926 µ	7.29	2.54	41.0 k	104 µ	12.4
	10	1781		127 µ		2.62	3.23 k	8.44 µ	
tridiag	5	470	0.288	63.1 µ	1.06	2.54	17.8 M	45.3 m	3.41
	8	1631		59.4 µ		2.69	4.93 M	13.3 m	
2mm	5	781	0.385	209	3.40	2.79	20.4 G	57.0	7.46
	8	2029		61.4		2.92	2.62 G	7.64	
3mm	5	760	0.207	114	6.76	2.55	32.3 G	82.3	9.13
	10	3677		16.9		2.82	3.19 G	9.01	
atax	5	627	0.507	353 m	1.54	2.60	176 M	457 m	5.42
	5	1237		230 m		2.61	32.4 M	84.3 m	
bicg	5	427	0.304	887 µ	6.72	2.54	160 M	407 m	8.98
	5	1406		132 µ		2.78	16.3 M	45.3 m	
gemm	5	524	0.234	1.99	2.97	2.54	10.8 G	27.4	9.13
	10	2240		0.67		2.69	1.12 G	3.00	
seidel	5	620	0.349	10.7 µ	2.46	2.60	960 M	2.50	7.16
	8	1778		4.31 µ		2.66	131 M	0.349	
gemver	5	809	0.382	7.28 M	4.46	2.87	23.1 M	66.2 m	3.15
	5	2120		1.63 M		2.77	7.60 M	2.10 m	(8.29)
mvt	5	701	0.251	91.0 µ	3.32	2.56	23.1 M	59.1 m	7.49
	10	2793		27.4 µ		2.80	2.82 M	7.89 m	(9.30)
syr2k	5	709	0.259	250 µ	4.07	2.89	14.0 G	40.3	6.95
	10	2740		61.4 µ		2.71	2.14 G	5.80	(7.62)
Geomean			0.289		3.69				7.19 (8.01)

Table 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 Sec. 8.3. We performed place-and-route for exact statistics.

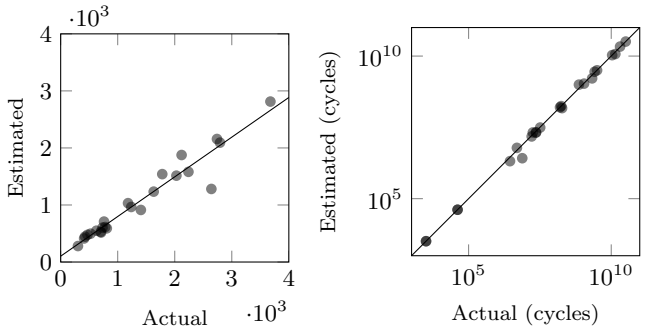
8. EVALUATION

8.1 Method

We have evaluated SOAP3 on a suite of benchmark examples which consists of several applications that have recurring inter-iteration dependences:

- A simple loop, `sum`, that sums all elements in an array;
- Two kernels from Livermore Loops [10]: `dotprod`, which calculates the dot product of two vectors, and `tridiag`, which solves a tridiagonal system of linear equations; and
- Nine kernels from PolyBench [11], which calculate matrix/vector transpositions, additions and multiplications (`2mm`, `3mm`, `atax`, `gemm`, `gemver`, `mvt`), the biconjugate gradient stabilized method (`bicg`), the Seidel stencil computation (`seidel`), 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 SOAP3, specifically targeting the Xilinx Virtex7 device running at 333 MHz, for the three objectives of accuracy, resource utilization and latency simultaneously. We then used Vivado HLS 2015.2 [5] to synthesize the resulting optimized programs into RTL implementations for exact latency information, and performed place-and-route using Vivado Design Suite 2015.2 [17], to obtain exact resource utilization statistics. Our tool produces a 4D Pareto frontier for each program; to better present our results, in the following section we only consider three dimensions, namely, accuracy, latency and LUTs.



(a) LUT count estimates (b) Latency estimates (log-log)

Figure 5: Comparisons of our estimated resource and latency statistics against Vivado HLS.

8.2 Results

Table 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 SOAP3. We synthesized each program to a circuit to obtain exact statistics, which are shown in Table 3.

Figure 5a compares our estimated LUT counts (vertical axis) against the exact LUT counts (horizontal axis) obtained by synthesizing RTL implementations of each program in Table 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 5b compares our estimated latency (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 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 Sec. 2, Figure 6 demonstrates the range of optimized programs discovered by SOAP3 when applied to the Seidel stencil loop kernel. All optimized programs are discovered in 876 seconds with SOAP3. In the figure, ×-points indicate the original program. By using only the rules of real arithmetic, our tool finds a more efficient program that can improve run time by 2.5×, as shown by the ○-points. However, by enabling partial loop unrolling and our dependence elimination rules, the performance is further improved, resulting in a 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 SOAP3 a greater freedom in terms of discovering more accurate expressions. In this example, the most efficient program is also the most accurate one: it minimizes round-off errors by approximately 2.5×. It is worth noting that our tool 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 Vivado HLS synthesizes dual port RAMs for arrays, and in one clock cycle we can only read from the memory allocating array twice.

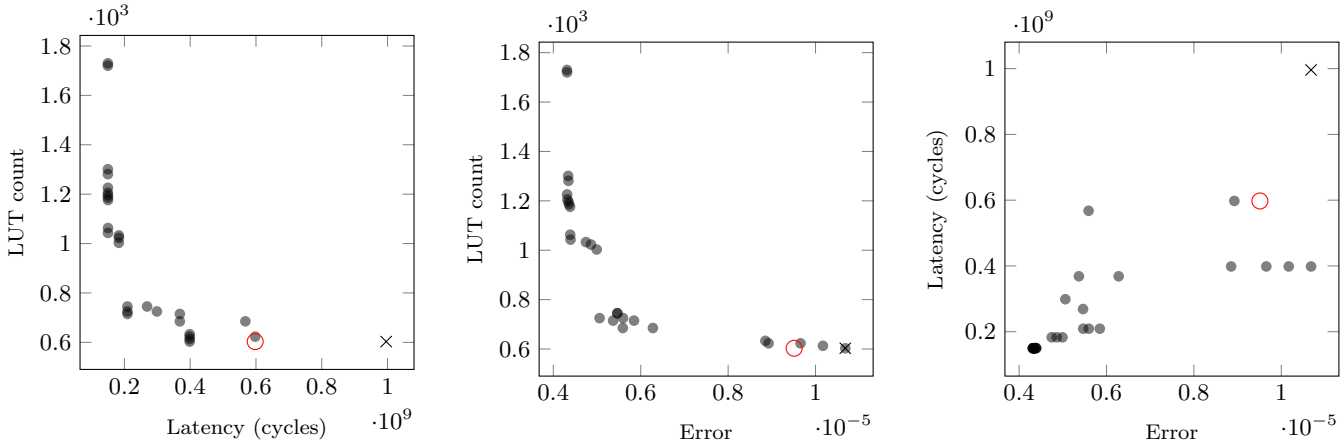


Figure 6: Pareto-optimal variants of the Seidel stencil program from Figure 1. Each graph shows a 2D projection of the Pareto frontier. In each graph, the original program is marked ×, and the lowest-latency variant obtained by arithmetic transformations alone is marked ○.

Our optimization flow discovers this bottleneck and stops exploring further loop unrolling.

Similar graphs for the other benchmarks can be viewed online,[2] each showing three projections from different axes of the Pareto frontier. Our web page can be used to interactively explore the positions of each data point on the three projections simultaneously, and view the corresponding generated C programs.

8.3 Discussion

As demonstrated by Figure 5b, SOAP3 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:

```
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    x[i] += ...;
```

We found that Vivado HLS occasionally fails to find the optimal schedule, predicted by SOAP3, that could pipeline this loop as tightly as possible. We fix this problem by rewriting the above code into:

```
for (int i=0; i<N; i++) {
  float sum = x[i];
  for (int j=0; j<N; j++)
    sum += ...;
  x[i] = sum;
}
```

This enables Vivado HLS to generate a hardware implementation with the expected II. The ratios in parentheses in Table 3 reflect the speedup by performing this simple fix.

9. RELATED WORK

Our work conducts program transformations on the MIR intermediate representation of Gao *et al.* [8]. Alternative program representations include *static* and *dynamic single assignment forms* (SSA, DSA) [18, 19], and *control and data flow graphs* (CDFG) [20]. These representations are less suitable for our work because they are all statement-based and do not identify as many equivalent programs as MIRs do. Dependence graphs [9], on the other hand, are designed for the purpose of capturing data flow dependences in scheduling techniques, but they generally do not preserve enough information for us to reconstruct a program from the graph itself.

Several HLS tools exploit dependence graph restructuring to improve loop parallelism, which allows for a smaller initiation interval, and in turn faster programs. Tree height reduction [21] aims to balance an arithmetic expression tree using associativity and distributivity. Xilinx's Vivado HLS has a similar feature called *expression balancing* [5]. Neither of these methods produce optimal loop pipelining, as they do not examine the implications of loop-carried dependences. Canis *et al.* [3] propose a similar approach called *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 [14], an open-source academic HLS tool. However, both LegUp and Vivado HLS only apply associativity in their restructuring.

Most importantly, none of the above mentioned techniques and tools aim to minimize, or even analyze, the impact of their transformations on resource usage and accuracy. In many numerically sensitive programs, small round-off errors would result in catastrophic inaccurate results. Therefore, HLS tools generally disable this feature by default for floating-point computations. For this reason, we have developed SOAP3 to optimize not only program latencies, but also resource usage and accuracy.

Several authors have considered program transformations that improve accuracy or resource usage. Damouche *et al.* [22] and Panchekha *et al.* [23] propose methods for optimizing numerical accuracy in software using equivalences from real arithmetic, but they consider individual expressions only, and have no control structure manipulation, such as optimizing across basic blocks or partial loop unrolling. Hosangadi *et al.* [24] minimize resource usage by employing symbolic algebra to reduce the number of operations, and Peyman-

[2] https://admk.github.io/soap/plot.html

doust *et al.* [25] factorize polynomials using Gröbner bases; both only deal with polynomial arithmetic expressions. The SOAP2 tool of Gao *et al.* [8] simultaneously optimizes numerical programs for resource usage and accuracy, but is unable to analyze latency.

10. CONCLUSION

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 paper has presented 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 *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 SOAP3 can optimize commonly used code fragments from PolyBench [11] and Livermore Loops [10] to have up to a 12× increase in performance, and up to 7× reduction of round-off errors, at the cost of up to 4× more resource utilization. Our tool is open-source and can be downloaded here: `https://github.com/admk/soap`.

Currently, SOAP3 supports only single-precision floating-point data types; we intend to extend this to multiple-precision floating-point and fixed-point types, and explore the impact on latency, resource utilization and numerical accuracy. This could, for instance, allow us to automate the manual design space exploration of matrix/vector multiplication architectures in [26].

11. ACKNOWLEDGMENTS

This work is supported by EPSRC (Grants EP/I020357/1, EP/K015168/1, and EP/I01236/1), Royal Academy of Engineering, and Imagination Technologies.

12. REFERENCES

[1] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An Overview of Today's High-Level Synthesis Tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[2] Berkeley Design Technology, Inc., "An Independent Evaluation of: High-Level Synthesis Tools for Xilinx FPGAs," 2010. [Online]. Available: http://www.xilinx.com/technology/dsp/BDTI_techpaper.pdf

[3] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *FPL*, 2014.

[4] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[5] Xilinx, Inc., "Vivado Design Suite User Guide—High-Level Synthesis," 2015.

[6] X. Gao, S. Bayliss, and G. A. Constantinides, "SOAP: Structural optimization of arithmetic expressions for high-level synthesis," in *FPT, 2013*.

[7] C. Mouilleron, "Efficient computation with structured matrices and arithmetic expressions," Ph.D. dissertation, ENS LYON, 2011.

[8] X. Gao and G. A. Constantinides, "Numerical program optimization for high-level synthesis," in *FPGA*, 2015.

[9] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in *MICRO*, 1994.

[10] J. Dongarra and P. Luszczek, "Livermore Loops," in *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 1041–1043.

[11] L.-N. Pouchet, "PolyBench/C—the Polyhedral Benchmark suite," http://web.cse.ohio-state.edu/~pouchet/software/polybench/.

[12] S. Verdoolaege, "isl: An Integer Set Library for the Polyhedral Model," in *ICMS*, 2010.

[13] R. Bridson, "Fast poisson disk sampling in arbitrary dimensions," in *SIGGRAPH Sketches*, 2007.

[14] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *FPGA*, 2011.

[15] G. Wang, W. Gong, and R. Kastner, "Operation Scheduling: Algorithms and Applications," in *High-Level Synthesis*. Springer, 2008.

[16] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *FPGA*, 2015.

[17] Xilinx, Inc., "Vivado Design Suite User Guide—Synthesis," 2015.

[18] B. Rau, "Data Flow and Dependence Analysis for Instruction Level Parallelism," in *Languages and Compilers for Parallel Computing*, ser. LNCS. Springer, 1992, vol. 589, pp. 236–250.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TOPLAS*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[20] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 11, no. 4, pp. 44–54, Oct. 1994.

[21] A. Nicolau and R. Potasmann, "Incremental tree height reduction for high level synthesis," in *DAC*, 1991.

[22] N. Damouche, M. Martel, and A. Chapoutot, "Intra-procedural optimization of the numerical accuracy of programs," in *Formal Methods for Industrial Critical Systems*, ser. LNCS. Springer, 2015, vol. 9128, pp. 31–46.

[23] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *PLDI*, 2015.

[24] A. Hosangadi, F. Fallah, and R. Kastner, "Factoring and eliminating common subexpressions in polynomial expressions," in *ICCAD*, 2004.

[25] A. Peymandoust and G. De Micheli, "Using symbolic algebra in algorithmic level DSP synthesis," in *DAC*, 2001.

[26] D. Boland and G. Constantinides, "Revisiting the reduction circuit: A case study for simultaneous architecture and precision optimisation," in *FPT*, 2013.