Making HLS A Bit Wiser: From Standard High-Level Datatypes to Arbitrary Low-Level Bitwidths

by

Hsuan Hsiao

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2017 by Hsuan Hsiao

Abstract

Making HLS A Bit Wiser:

From Standard High-Level Datatypes to Arbitrary Low-Level Bitwidths

Hsuan Hsiao

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2017

High-level synthesis provides an easy-to-use abstraction for designing hardware circuits. However, standard datatypes available in high-level languages are over provisioned for typical applications, incurring unnecessary area overhead since the underlying FPGA hardware can support arbitrary bitwidths. We provide a bitwidth minimization (BWM) framework that analyzes and eliminates unused bits in the circuit's datapath and allows for the use of arbitrary width datatypes in the high-level source code. We then propose *Sensei*, an advisor that predicts the post-synthesis area savings brought about by reducing bitwidth and presents users with a ranking of program variables based on area impact. Together, these two contributions aim to bridge the gap between the over-provisioned datapaths of high-level languages and the arbitrary-width datapaths of low-level circuits.

Acknowledgements

First, I would like to thank my advisor, Professor Jason Anderson, for his constant guidance and support. I would not have reached this point if not for his encouragement, inspiration and endless advices. He has been a great motivator for my academic and personal development, and will always be a role model.

Second, I would like to thank my parents and my brother for their unconditional love and support. Their patience and trust in me has helped immensely in this journey. It would not be the same without them constantly humouring me whenever I decide to do random things.

A big thank you to Josh, for always being there for me, supporting and inspiring me to strive for higher. All the thoughtful rants and encouraging outlooks on life have helped me in shaping my own perspective on many things.

Last but not least, I am fortunate to have been surrounded by lots of wonderful people during these years. The current and past graduates in our research team: Joy, Jin Hee, Xander, Steven, Brett, Blair, Andrew, James, Lanny, Bain and Safeen; as well as fellow labmates of PT477: Karthik, Fernando, Mario, Nariman, Naif, Charles and Shehab. Thank you all for the invaluable feedback and suggestions on both research and many aspects of personal development. Above all, thank you all for creating such a fun and supportive environment for me to learn and grow.

Contents

1 Introduction **1**

1.1 Contributions 4

1.2 Thesis Outline 4

2 Background **5**

2.1 The LegUp High-Level Synthesis Tool 5

2.2 The LLVM Intermediate Representation 7

2.3 Quartus II CAD Tool 10

2.4 Cyclone V Resources 11

2.5 Summary 12

3 Bitwidth Minimization Framework **13**

3.1 Background 14

3.2 Related Work 15

3.3 Bitwidth Minimization Framework in LegUp 16

3.3.1 Updated Implementation of Core Bitmask Analysis 16

3.3.2 Bitmask Propagation for Functions 20

3.3.3 Bitmask Propagation for Memory Operations 23

3.3.4 User-Declared Arbitrary Bitwidths 26

3.4 Evaluation 29

3.4.1 Methodology 30

3.4.2 Bit Savings . . . 31
3.4.3 Resource Utilization . . . 33
3.4.4 Bit and Resource Savings with Default HLS Settings . . . 36
3.4.5 Case Study of User-Declared Bitwidths . . . 37
3.5 Conclusion . . . 38
4 Sensei: Area Reduction Advisor **40**
4.1 Overview . . . 41
4.2 Related Work . . . 44
4.3 Convolutional Neural Networks . . . 44
4.3.1 Architectural Components . . . 45
4.3.2 Runtime Parameters of a CNN . . . 47
4.4 Sensei Predictor . . . 48
4.4.1 Analytical Predictor . . . 49
4.4.2 CNN-Based Predictor . . . 50
4.5 Methodology . . . 57
4.5.1 Experimental Setup . . . 57
4.5.2 Evaluation Criteria . . . 58
4.6 Results . . . 60
4.6.1 CNN Model Selection . . . 60
4.6.2 Impact of Input Dataset Configuration . . . 64
4.6.3 Impact of CNN Hyperparameters . . . 65
4.6.4 Comparison of Analytical to CNN-based Predictor . . . 72
4.7 Conclusion . . . 75
5 Conclusion **76**
5.1 Future Work . . . 77
Appendices **79**

A Bitwidth Minimization Framework Data **80**

A.1 Raw Data for different Components of Bitwidth Minimization Framework 80

B Advisor Framework Data **83**

Bibliography **84**

Chapter 1

Introduction

As a result of the continued technology scaling arising from Moore's Law, circuit designers are able to incorporate more and more functionality onto a single chip. However, a larger system with more functionality comes with a trade-off of increased design complexity and increased time and effort required for circuit verification and modification. High-level synthesis (HLS) seeks to reduce these burdens by enabling the automated synthesis of a hardware circuit from a software program. Hardware design and verification can proceed at a higher level of abstraction and with greater ease, thereby reducing design time and cost. HLS has become a mainstream design methodology for field-programmable gate arrays (FPGAs), with both Xilinx and Intel offering commercial HLS solutions. The importance of HLS is underscored by the fact that the vendors themselves now use their own HLS solutions to produce compute accelerators for specific application domains, such as machine learning [7, 40].

As with any design methodology, whether it be for hardware or software, the introduction of an abstraction layer trades off optimality and customizability for increased usability. With HLS, the ability to allow designers to use a familiar software language to specify hardware designs trades off *some* of the designer's ability to fine-tune circuits for higher performance and area efficiency. While much easier to design, verify, and modify,

the area footprint and circuit performance of HLS-generated circuits may be far from those produced by manual RTL design [34, 43], particularly in specific circumstances where there exists a "known good" hardware implementation (e.g. an FFT circuit implementation). Ongoing research efforts in both academia and industry have sought to improve various aspects of the HLS design methodology [17,29,42]. For the scope of this thesis, we focus on reducing the area footprint of HLS-generated circuits and providing new design methodologies for fine-tuning the area of circuits produced by HLS. In particular, we explore a unique source of area inefficiency encountered only in HLS due to its high-level abstraction.

One attractive feature of HLS is that it allows designers to use a familiar software language to specify a hardware design. We make the observation that software languages designed for typical processor architectures may incorporate processor-specific features into the language. However, such processor-specific features are not necessarily transferable to custom hardware circuitry. Thus, these "features" in the software language can limit (or even negatively impact) the performance and efficiency of the generated circuit when the same language is used for specifying a hardware design. In this thesis, we take an in-depth look at one such language feature: standard datatype width.

In a standard processor, the pipeline and execution units are all fixed at widths of 8, 16, 32 or 64 bits in order to simplify memory accesses and alignment. Datatypes with arbitrary widths would not improve performance, so software languages typically only provide primitive datatypes quantized to 8, 16, 32, and 64 bits. On the other hand, a key advantage of using custom hardware versus a microprocessor is that datapath widths can be tailored in hardware to match the application at hand for better performance and area efficiency. The use of software languages to specify hardware in HLS presents a unique dilemma faced by the HLS design methodology: it is challenging for HLS users to produce circuits with custom-width datapaths.

To address this shortcoming, we implement a compiler pass that analyzes the in-

put program's dataflow graphs and, through static analysis, automatically drops unused bits and replaces known constant (non-toggling) bits with their respective constant signals. This provides a convenient, automatic reduction of datapath widths for the HLS user. However, compiler passes are conservative, primarily because they execute without knowledge of program input data. Consequently, there often exist further opportunities that compilers are not able to leverage. To compensate for this, we introduce the notion of an arbitrary-width datatype in the software language to allow the HLS user to specify a reduced width datapath in the source code itself. This is done by attaching special attributes to the standard software datatypes, such that they are captured only by HLS tools and ignored by standard compilers. These two contributions together constitute our bitwidth minimization (BWM) framework.

In order to make it easier for HLS users to construct circuits with reduced-width datapaths, we further enhance the BWM framework by introducing *Sensei*, an advisor framework that guides HLS users on when and where to reduce datapath widths for maximum benefit. We observe that a key research challenge in HLS is tied to its "black box" nature and the consequent disconnect between the software program (input) and the generated circuit (output). This disconnect significantly impedes HLS usability and leads to the question of how should a software developer *change* their program to produce a better-quality HLS-generated circuit? Sensei takes a step to address this question for the user. It accurately estimates the area impact of reducing the bitwidth of each program variable and provides the programmer with a list of key variables to focus on. The reductions in the widths of these variables are likely to have the most impact on final circuit area. Area-reduction predictions are made using two predictor approaches: one analytical and one based on a convolutional neural network (CNN). Both predictors only require information available at the HLS compiler stage, ensuring portability across different hardware platforms.

1.1 Contributions

The contributions of this thesis are as follows:

- We present the BWM framework, a compiler pass implemented within the LegUp HLS tool [10] that automatically infers reduced datapaths in the generated circuit. We also introduce arbitrary-precision datatypes to the software language to enable the specification of reduced-width datapaths in the HLS-synthesized circuits.
- We present *Sensei*, an area-reduction advisor for HLS users that uncovers opportunities for area savings via bitwidth reduction of program variables.
- We explore the use of CNNs in Sensei to predict the area impact of source-code bitwidth changes. We introduce a novel transformation of the compiler's dataflow graph (DFG) into a spatial representation well-suited to CNNs.
- We implement Sensei on top of the LegUp HLS tool and perform an experimental study of a range of CNN models compared against a traditional analytical approach. Our promising results showcase the strong potential for CNNs in HLS.

1.2 Thesis Outline

In Chapter 2, we provide background information on what HLS is and the key steps involved in the HLS tool flow. We also provide a high-level overview of the LLVM compiler framework that LegUp is built on top of, as well as a brief overview of the steps in the Quartus II CAD flow. Chapter 3 describes the BWM framework, discussing how our compiler analysis infers narrower widths and how the addition of arbitrary-width datatypes is made possible. Chapter 4 dives into the detailed implementation of Sensei, our area-reduction advisor. We conclude with Chapter 5 and touch on some interesting extensions and modifications for future work.

Chapter 2

Background

In this chapter, we review background material necessary for understanding the main contributions of this thesis. We first give a high-level overview of the LegUp HLS tool, with emphasis on aspects affecting area and bitwidth. We then move on to discuss the LLVM compiler (within which LegUp is built), and its internal representation of programs, on which all compiler optimizations are applied. The final two sections of the chapter briefly describe the Intel Cyclone V FPGA architecture, and the associated Quartus II CAD tools, as the tools and FPGA family are used for all experiments in this research.

2.1 The LegUp High-Level Synthesis Tool

LegUp [10] is an open-source HLS tool developed at the University of Toronto. It takes a software program written in *C* as input and automatically generates an RTL circuit description in *Verilog HDL*. HLS tools typically implement three major phases of transformation in order to generate a hardware design from a software description: allocation, scheduling, and binding [12, 14–16]. Allocation refers to the process of determining how many functional units of each type to create for the program's hardware implementation (e.g. how many multiplier units are permitted in the hardware?). In LegUp, allocation is

performed first and defines the resource constraints that are used in the scheduling and binding steps. For example, the HLS tool can choose to create only one multiplier unit and force all multiply operations in the program to be serialized so that only one multiply can proceed at a particular point in time. Alternatively, it can choose to create as many multiplier units as there are multiply operations in the program so that all multiplies can proceed at the same time in the absence of data dependencies. Scheduling refers to the step at which operations are assigned to specific clock cycles. This step defines *when* an operation will be executed, which impacts the number of cycles the program will take to finish and the maximum operating frequency of the circuit. For example, if the operands of a multiplication are ready but the product is not used until much later, the HLS tool can choose to schedule the multiplication anytime from now until then. The binding step examines the scheduled operations to determine which functional unit in the hardware should be used to implement the operation. There are typically less functional units than the number of operations that require them; the binding algorithm therefore determines how functional units are shared. As the number of operations bound to a functional unit increases, the area incurred by the multiplexer on each of the inputs of the functional unit also increases. For example, if there are two adder units where adder A has 2 operations bound to it and adder B has 6 operations bound to it, the binding algorithm will assign the next addition to adder A since the increase in area of the multiplexer is less.

LegUp's design tool flow is shown on Figure 2.1. The LegUp HLS tool is built on top of the LLVM (low-level virtual machine) compiler framework [26], discussed further in Section 2.2. The input `C` program is parsed and transformed into a platform-independent *intermediate representation* (IR) by Clang [3], LLVM's `C` frontend. Various standard compiler optimizations and HLS-specific optimizations are applied to the IR; the optimizations improve the quality of the generated hardware circuit. Using the final optimized IR, the three main steps of HLS (as overviewed above) are implemented as a backend pass of the LLVM framework, producing a Verilog HDL file as output. The Verilog can

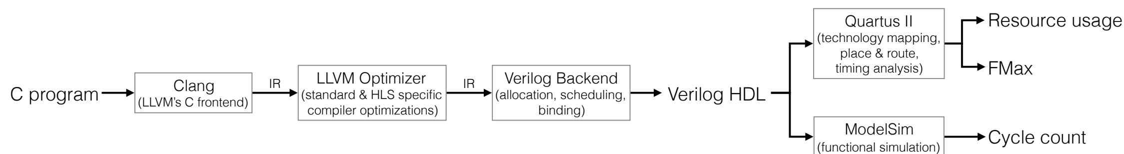


Figure 2.1: Design flow using LegUp

then be pushed through Intel's Quartus II CAD tool to map the circuit onto an FPGA and obtain area and performance numbers. The functional correctness of the design can be verified through simulation with Mentor Graphics's ModelSim RTL simulator.

2.2 The LLVM Intermediate Representation

In the LLVM compiler framework, all optimizations, code transformations and information transfer are done through the platform-independent IR. In this section, we highlight a few characteristics of the IR that are important to our framework, some that we leverage and some that we overcome. Details of how we leverage and overcome each characteristic is discussed further in Chapter 3.

LLVM's IR is based on a static single assignment (SSA) paradigm, meaning that every variable in the IR is assigned exactly once and is defined before it is used. For example, in the simple `C` code shown in Listing 1, `a` is assigned two values: `A1[1]` on line 4 and `A1[4]` on line 8. The LLVM IR representation of this code is shown in Listing 2, where the first assignment to `a` is on line 5 (assigned to `%1`) and the second assignment on line 8 (assigned to `%4`). In a similar fashion to this, if a `C` variable is assigned a value n times, there will be n LLVM IR variables that correspond to the single `C` variable. For our BWM framework that allows the HLS user to specify a variable as a reduced-width datatype, this characteristic of the IR requires us to maintain the mapping between a program variable and its corresponding IR variables in order to reduce the width of all instances of it.

To aid in the many-to-one mapping between IR variables and `C` program variables,

LLVM provides debug information that can be enabled by compiling the source code with the `-g` option. In the IR annotated with debug information, extra calls to intrinsic functions (i.e. special functions provided and implemented by the compiler) `llvm.dbg.declare` and `llvm.dbg.value` are added whenever a value is assigned to a variable. `llvm.dbg.declare` is called when a variable is declared, and `llvm.dbg.value` is called when a variable is assigned a new value. *Metadata* in LLVM refers to optional information that is contained within the IR specification of the program, but can be

```
1  volatile int A1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3  int main(void) {
4      int a = A1[1];  // 1st assignment to a
5      int b = A1[3];
6      int c = a + b;
7
8      a = A1[4];      // 2nd assignment to a
9      b = A1[5];
10
11     return a + b + c;
12 }
```

Listing 1: Example `C` source code for demonstrating SSA form.

```
1  @A1 = internal global [10 x i32] [i32 0, i32 1, i32 2, i32 3, i32 4, i32 5, i32 6,
    ↪  i32 7, i32 8, i32 9], align 4
2
3  ; Function Attrs: nounwind
4  define i32 @main() #0 {
5    %1 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 1),
      ↪  align 4, !dbg !22, !tbaa !23      ; 1st assignment to a
6    %2 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 3),
      ↪  align 4, !dbg !28, !tbaa !23
7    %3 = add nsw i32 %2, %1, !dbg !30
8    %4 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 4),
      ↪  align 4, !dbg !32, !tbaa !23      ; 2nd assignment to a
9    %5 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 5),
      ↪  align 4, !dbg !33, !tbaa !23
10   %6 = add i32 %3, %4, !dbg !34
11   %7 = add i32 %6, %5, !dbg !34
12   ret i32 %7, !dbg !34
13 }
```

Listing 2: LLVM IR for Listing 1 demonstrating SSA form.

```
1  @A1 = internal global [10 x i32] [i32 0, i32 1, i32 2, i32 3, i32 4, i32 5, i32 6,
   ↪  i32 7, i32 8, i32 9], align 4
2
3  ; Function Attrs: nounwind
4  define i32 @main() #0 {
5    %1 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 1),
     ↪  align 4, !dbg !22, !tbaa !23      ; 1st assignment to a
6    tail call void @llvm.dbg.value(metadata !{i32 %1}, i64 0, metadata !10), !dbg !27
     ↪  ; debug info for %1
7    %2 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 3),
     ↪  align 4, !dbg !28, !tbaa !23
8    tail call void @llvm.dbg.value(metadata !{i32 %2}, i64 0, metadata !11), !dbg !29
9    %3 = add nsw i32 %2, %1, !dbg !30
10   tail call void @llvm.dbg.value(metadata !{i32 %3}, i64 0, metadata !12), !dbg !31
11   %4 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 4),
     ↪  align 4, !dbg !32, !tbaa !23      ; 2nd assignment to a
12   tail call void @llvm.dbg.value(metadata !{i32 %4}, i64 0, metadata !10), !dbg !27
     ↪  ; debug info for %4
13   %5 = load volatile i32* getelementptr inbounds ([10 x i32]* @A1, i32 0, i32 5),
     ↪  align 4, !dbg !33, !tbaa !23
14   tail call void @llvm.dbg.value(metadata !{i32 %5}, i64 0, metadata !11), !dbg !29
15   %6 = add i32 %3, %4, !dbg !34
16   %7 = add i32 %6, %5, !dbg !34
17   ret i32 %7, !dbg !34
18 }
19
20 ...
21
22 !10 = metadata !{i32 786688, metadata !4, metadata !"a", metadata !5, i32 4,
   ↪  metadata !8, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [a] [line 4]
23 !11 = metadata !{i32 786688, metadata !4, metadata !"b", metadata !5, i32 5,
   ↪  metadata !8, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [b] [line 5]
24 !12 = metadata !{i32 786688, metadata !4, metadata !"c", metadata !5, i32 6,
   ↪  metadata !8, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [c] [line 6]
```

Listing 3: LLVM IR for Listing 1 with debug information.

discarded without affecting the program's correctness. In both of the `llvm.dbg` functions, the first argument specifies metadata that wraps the IR variable this function call pertains to, and the last argument specifies metadata that contains information about the source code variable. For example, in Listing 3, we see that lines 6 and 12 are added to provide extra information about the value assignments that happen on lines 5 and 11, respectively. Looking at line 6, we see that the call to `llvm.dbg.value` maps IR variable `%1` to metadata `!10`, shown on line 22. We leverage this existing infrastructure in LLVM

to simplify the effort required in constructing our BWM framework.

Another characteristic of the LLVM IR is that it requires all operands to have the same data type and width as the result. In the case of the simple example in Listing 2, all of %1 through %7 are 32-bit integers. However, if there are any instances where two variables of different width are used in an operation, an extra sign extend or zero extend instruction is needed before the operation to conform to this LLVM IR requirement. This poses a challenge to the flexibility of our framework, imposing a wider datapath and functional unit than what is really necessary; we discuss how we overcome this in Section 3.3.4.

LLVM provides a hierarchical structure for the different components of the IR representation of a program. At the highest level, a *module* encapsulates the entire program. Within a module, several components can exist, including global variables, functions and metadata. In addition to variables declared as globals in the source code, all arrays are considered global variables in the LLVM IR.

2.3 Quartus II CAD Tool

Quartus II is software for programmable logic design provided by Intel [5]. To synthesize a Verilog HDL design into a bitstream for configuring an FPGA, Quartus II performs two major steps: 1) analysis and synthesis, and 2) place and route. In the first step, the compiler checks the Verilog HDL design for syntax errors and performs technology mapping of the design. This step infers structures such as flip-flops and state machines from the Verilog HDL and performs analysis and optimization to remove redundant logic, minimize resource usage and map logic onto the device architecture as efficiently as possible. In the place and route step, logic functions are assigned to cell locations on the device. Various optimizations are performed to improve resource usage and timing performance, and the user can additionally set options such as enabling register packing

and limiting length of carry chains. In either step of the CAD flow, Quartus II performs a variety of optimizations that can influence the effectiveness of our BWM framework.

2.4 Cyclone V Resources

FPGAs have a variety of building block resources on board, with adaptive logic modules (ALMs), registers, memory blocks and DSPs being the resources used for implementing logic. We briefly describe each resource type that is relevant to the evaluation of our framework and highlight how they may trade off with each other. An ALM in the Cyclone V family of FPGAs is composed of an 8-input fracturable look-up table (LUT), as shown in Figure 2.2 [4]. Each ALM is able to implement either one 6-input function, two independent 4-input functions, or a 5-input and a 3-input function with independent inputs. Within the ALM, there are also four registers and two full adders to enable higher density and arithmetic capability. For a design implemented on the FPGA, an ALM may be used either for its LUTs, or registers, or both.

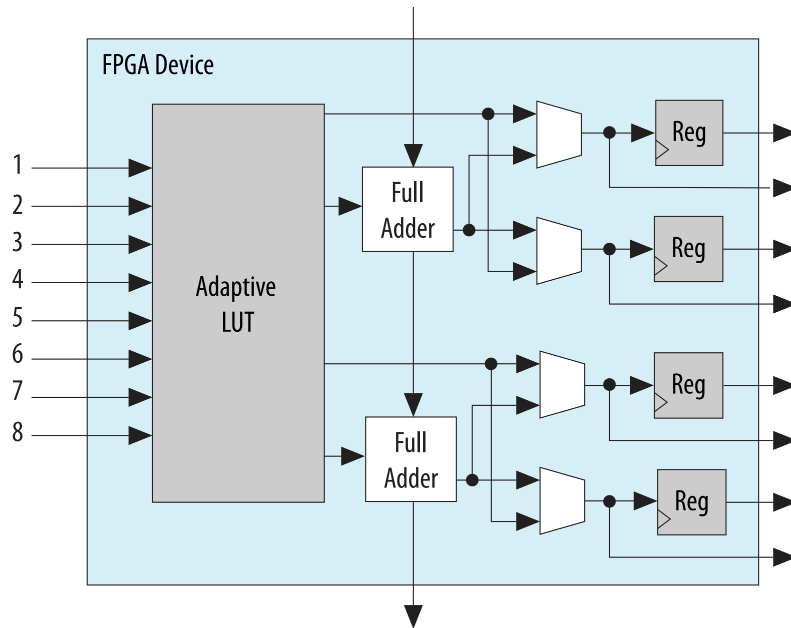


Figure 2.2: Components inside a Cyclone V ALM [4].

Each DSP block in a Cyclone V device is capable of being configured as three 9×9 multipliers, two 18×18 multipliers or one 27×27 multiplier. The Quartus compiler may choose to use DSPs to implement certain logic instead of ALMs, and vice versa, to enable higher performance and area efficiency.

Cyclone V devices contain two types of memory blocks: 10 kb dedicated memory blocks (M10K) and 640 b memory logic array blocks (MLAB). Each MLAB is made up of ten ALMs and are ideal for wide and shallow memory arrays.

2.5 Summary

This chapter provides background information on the tools and frameworks that we build upon in this thesis. We first describe the typical design flow of the LegUp HLS tool. We then discuss the relevant features and limitations of the LLVM IR, the Quartus II CAD tool and the Cyclone V FPGA architecture that we use in our framework.

Chapter 3

Bitwidth Minimization Framework

Applications written in high-level programming languages, like `C`, typically use standard datatypes, which are 8, 16, 32 or 64 bits. However, this set of standard bitwidths is often too coarse-grained for optimal representation of most data values. While this inefficiency in bit usage is inherent to software programming, it can be alleviated when the application is synthesized in hardware, where the datapaths can be specified to arbitrary widths. Several prior works have explored the idea of synthesizing a reduced-area circuit based on the analysis of the widths of software variables, demonstrating promising results [13, 19, 39, 44]. Some HLS tools available today also allow users to specify the exact width of a software variable using special arbitrary-precision datatypes [1, 2, 6, 41].

In this chapter, we describe our bitwidth minimization (BWM) framework in the LegUp HLS tool, which is aimed at eliminating any static bits (i.e. constant 1 or 0) or unused bits from the circuit's datapath. The framework consists of two components: 1) a custom compiler optimization pass that identifies static and unused bits in the program's dataflow graph; and 2) an infrastructure that allows users of LegUp to declare non-standard widths for `C` program variables and propagates this custom bitwidth information to the Verilog code. The compiler pass is an updated implementation of the bitmask analysis first proposed by Gort and Anderson [19]. We also introduce optimizations

for function calls and memory operations to enable further circuit area savings. The infrastructure that allows users to declare `C` program variables as non-standard datatypes provides a means to overcome the overly conservative nature of compiler analysis. We leverage the fact that designers often have knowledge of the range of values that certain variables can take; feeding such information to the HLS tool can aid in generating a more optimized circuit.

In Section 3.1, we describe the main concept behind the core bitmask analysis by Gort and Anderson, which lays the foundation for our BWM framework. Section 3.2 provides a brief overview of other related techniques in literature. In Section 3.3, we describe our BWM framework in detail, which includes the updated implementation of the core bitmask analysis, the addition of two new optimizations, and the support for user-defined non-standard widths in the `C` source code. In Section 3.4, we present area numbers to quantify the effectiveness of the different parts of our framework. Finally, Section 3.5 summarizes the contributions described in this chapter.

3.1 Background

In the work by Gort and Anderson [18, 19], range analysis and bitmask analysis are combined to obtain the minimal width required for the datapaths in the HLS-generated circuit. In this technique, every variable and its intermediate state in the datapath is associated with a *bitmask*. Each value's bitmask determines which bits of that value can and cannot be safely removed without affecting the circuit's functionality. To set the initial bitmask, the range analysis from Campos et al. [9] is first applied to establish the upper and lower bound of every LLVM IR variable. Gort and Anderson devise a set of bitmask propagation rules to propagate static bits throughout the dataflow graph of the programs. These bitmask propagation rules are based on instruction type and account for all bitwise and integer arithmetic instructions. The algorithm iteratively

traverses through the program's instruction stream, applying the propagation rules until all bitmasks converge. The authors implement this framework in LegUp and observe a reduction of 18% in post-synthesis LUT usage when evaluated on an Intel Cyclone II device. The benchmarks used in their evaluation includes the CHStone benchmark suite [22] (arithmetic circuits were omitted), `fft` and `dhrystone`.

3.2 Related Work

Several earlier works have evaluated the effectiveness of using bitwidth information to reduce circuit area. Stephenson et al. [39] propose the *Bitwise Compiler*, which propagates value ranges through the program's control flow graph. They use program constructs to establish constraints on a variable's value range and define a set of forward and backward transit functions to propagate the ranges. When integrated with the *DeepC Silicon compiler* [8], silicon area savings ranging from 15% to 86% are achieved on Xilinx's 4000 series FPGAs. The focus of this work is on propagating value ranges, which is able to capture reduction of the most significant bits of a variable. Our bitmask based approach described in this chapter complements the range analysis and is able to capture reduction in any bit within the variable.

Cong et al. [13] apply Stephenson's algorithm to annotate the dataflow graph with bitwidth information. The scheduling and binding algorithms of the HLS tool utilize this information to minimize the total number of bits required in the functional units of the circuit. Area reduction of 36% is achieved when synthesized with Intel's Quartus II CAD tool onto Stratix FPGA boards. While this work does not propose new ways to reduce the width of a variable, it brings the range information into the different steps involved in HLS, showcasing the potential area reduction of a bitwidth aware HLS flow. Similar to this work, we also bring bitwidth information into the HLS flow, affecting mainly the allocation and binding of functional units. However, we do not modify any of the

HLS algorithms to specifically optimize for area, thus evaluating the impact of bitwidth reduction on overall circuit area under a performance and area balanced situation.

Mahlke et al. [32] create a framework for *PICO* (an early HLS tool) to iteratively propagate bitwidth constraints throughout the program's instruction stream. They leverage range analysis and focus on tight loop nests. This approach yields a mean reduction of 49% in total gate count. Unlike our work in this chapter, which targets FPGAs with a fixed architecture, they target the synthesis of nonprogrammable hardware accelerators that are pipelined units more similar to ASICs.

3.3 Bitwidth Minimization Framework in LegUp

The following sections present the four major components that make up the BWM framework in the LegUp HLS tool:

1. Updated implementation of the core bitmask analysis
2. Bitmask propagation for functions
3. Bitmask propagation for memory operations
4. Support for user-defined arbitrary bitwidth

We describe the details of each component in this section, and evaluate their effectiveness in Section 3.4.

3.3.1 Updated Implementation of Core Bitmask Analysis

The core compiler pass that determines which bits in a variable are safe to remove is based on the algorithm initially proposed by Gort and Anderson [19]. In this pass, every LLVM value is associated with a bitmask. Each bit in the bitmask can take on one of three states in the following precedence order: a known bit (static-0 or static-1), an

extension bit (E), or an unknown bit (?). For example, the bitmask of an 8-bit variable that is assigned only even numbers from -4 to 3 would be `EEEEE??0`. The known bits and the extension bits are both opportunities for reduction in the datapath width since they can be replaced by constants or easily inferred from the sign bit; whereas, an unknown bit represents true datapath hardware. The goal of the compiler pass is to statically assign as many of the bits in a variable's bitmask to be a known bit or an extension bit as possible.

In Gort and Anderson's algorithm, constants are propagated through the program's dataflow graph at the bit level via a series of forward and backward passes on the program's variables. Their algorithm alternates between the forward propagation phase, which performs a forward pass on all variables, and the backward propagation phase, which performs a backward pass on all variables, until the bitmasks for all variables converge.

The forward propagation phase of the algorithm performs *assignment analysis*, which looks at how a variable is defined and generates its bitmask based on that. In this phase, each instruction's output value is visited once and put through a forward pass. A forward pass of a value involves using the *forward transit function* pertaining to the specific instruction type that produces that value and using the bitmasks of the operands to generate a reduced bitmask of that value. Like the values themselves, bitmasks traverse through the program's dataflow graph. Along the way, each instruction type dictates how the bitmasks are transformed. Once a reduced-bitmask is generated, it is put through the *forward aggregation function* along with the variable's bitmask from previous passes to generate the new bitmask resulting from the forward pass of the instruction.

For example, the forward pass of the variable `c` is depicted in Figure 3.1a. Since `c` is the result of an `and` instruction, it utilizes the forward transit function of the `and` instruction, shown in Table 3.1, and the bitmasks of the operands to generate a bitmask for `c`. In this example, if we know that `a` has a mask of `????????` and `b` has a mask of

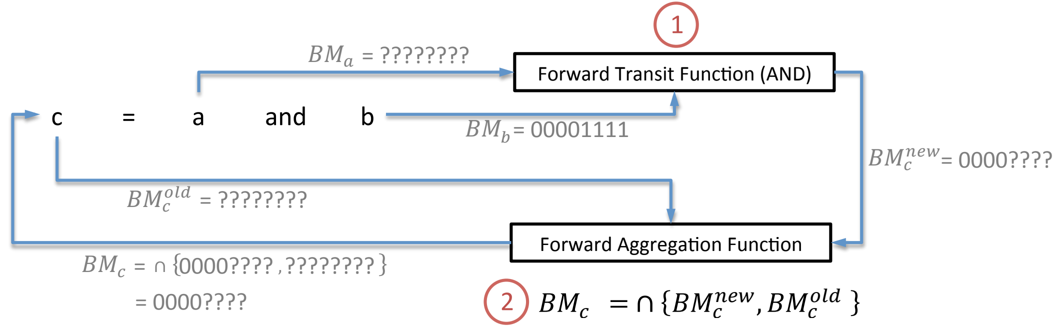
`00001111`, applying the forward transit function of `and` states that `c` will have a bitmask of `0000????`. Using the bitmask generated through the forward transit function and `c`'s original (i.e. from previous passes) bitmask, the forward aggregation function produces the final bitmask by taking the intersection of the two masks (Figure 3.1a equation). This final bitmask is the result of applying a forward pass on this instruction and reflects the actual useful toggling bits of this variable.

Table 3.2 and Table 3.3 show the definition of the *intersection* and the *union* of bitmask values. Taking the intersection of two bitmasks corresponds to finding the most optimized mask for each bit. A known bit will dominate both an extension bit and an unknown bit; an extension bit will dominate an unknown bit; but, two different-valued known bits will result in an unknown bit. Opposite to the intersection is the union of two bitmasks, which produces the least optimized mask for each bit since it has to accommodate both masks. An unknown bit will dominate both an extension bit and a known bit; an extension bit will dominate a known bit; and, two different valued known bits will result in an unknown bit. The intersection is used wherever we update a variable's old bitmask to a new reduced-width bitmask, while the union is used wherever we aggregate bitmasks from multiple variables, since we need to be conservative.

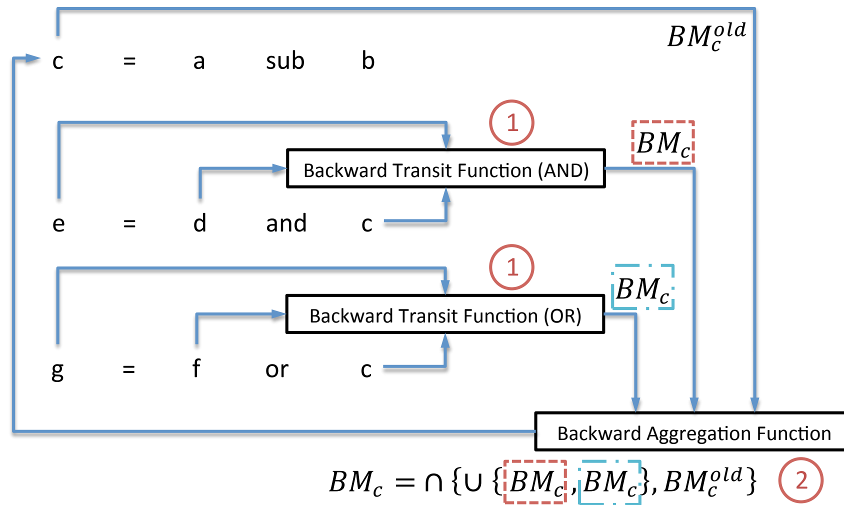
`and`	1	0	?	E
1	1	0	?	E
0	0	0	0	0
?	?	0	?	?
E	E	0	?	E

Table 3.1: Forward transit function for the `and` instruction. The row and column headings represent the bitmask of the first and second operand respectively. The entries represent the resultant bitmask given the specific operand bitmask values.

The backward phase of the core algorithm performs *use analysis*, which looks at how a variable is used and generates a bitmask based on the use requirements. In this phase, each variable (i.e. an instruction's output value) is visited once and goes through a backward pass. The backward pass on a variable first looks for all instructions that



(a) Forward pass of variable c



(b) Backward pass of variable c

Figure 3.1: Forward pass and backward pass on a value. (a) A bitmask is generated by using the operands' bitmasks and the forward transfer function of the instruction (1), and then the final value is obtained by taking the intersection of the generated mask and the old mask (2). An example with bitmask values going through the transit function shown in grey.

∩	1	0	?	E
1	1	?	1	1
0	?	0	0	0
?	1	0	?	E
E	1	0	E	E

Table 3.2: Intersection of bitmask values.

∪	1	0	?	E
1	1	?	?	E
0	?	0	?	E
?	?	?	?	?
E	E	E	?	E

Table 3.3: Union of bitmask values.

use this variable as an operand. For each time the variable is used, the *backward transit function* of the user instruction is invoked to generate a reduced bitmask of the variable. Once a reduced mask for the variable is generated from all instances where it is used, the *backward aggregation function* takes the union of all the reduced bitmasks and intersects this result with the variable's original bitmask. Note that in the backward pass of a variable, the instruction type that defines the variable is not used anywhere.

For example, Figure 3.1b shows how the backward pass updates the bitmask of variable `c`. `c` has two user instructions, each of which applies their backward transit function. Then to satisfy the bit requirements of both of `c`'s user instructions, the backward aggregation function takes the union of their masks. The result is intersected with `c`'s original bitmask to produce `c`'s updated bitmask from the backward pass.

All forward transit functions and backward transit functions for bitwise and arithmetic operations are defined in Gort and Anderson's original formulation. Our updated implementation of the core bitmask analysis consists of: 1) updating the codebase from LLVM 2.9 to LLVM 3.5, which saw several significant changes in the accessor functions for IR variables; 2) adding more aggressive analysis for loop induction variables; and 3) adding and fixing several bitmask propagation rules to address certain corner cases and account for the sign of values.

3.3.2 Bitmask Propagation for Functions

One optimization that we introduce in the BWM framework is the ability to propagate bitmasks across functions. The core bitmask analysis only handles bitwise and arithmetic operations; propagation halts whenever an unsupported instruction is encountered. Function calls, which were previously unsupported, can be inlined by the compiler to allow for propagation. However, in many cases, it is beneficial to not inline functions so that they can be shared and reused without incurring extra area.

To propagate bitmasks across function boundaries, the core algorithm was re-written

from being an LLVM function pass to a module pass, which examines all functions together in the bitmask propagation algorithm. Upgrading to a module pass not only allows us to propagate the bitmask through function arguments and return values, but also allows us to generate and propagate bitmasks for global variables, as discussed further in Section 3.3.3. Similar to the core algorithm, the mechanism to propagate bitmasks across functions lies in the new bitmask propagation rules introduced for two new instruction types: the `call` instruction and the `ret` (return) instruction. Bitmask support for function arguments are also introduced since they are treated as distinct entities in the LLVM IR. Figures 3.2, 3.3 and 3.4 show the additional rules and their interactions that enable propagation across function boundaries.

Recall that in all cases, a forward transit function of an instruction will update the mask associated with the output value of the instruction (forward flow of data, i.e. how the value is defined), and a backward transit function of an instruction will update the mask associated with an operand of the instruction.

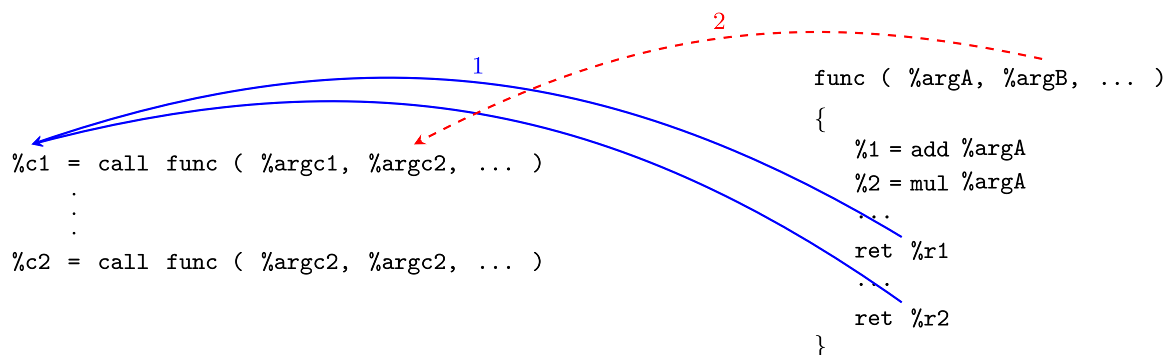


Figure 3.2: Forward (1) and backward (2) transit functions of call instruction.

In the forward pass of the `call` instruction, we update the bitmask of the `call` instruction's output using the operand of the `ret` instruction inside the function itself (arrow 1 in Figure 3.2). This is analogous to updating the output of an arithmetic operation based on the operand of that instruction. It is possible for a function to have multiple `ret` instructions. To address this, we introduce the concept of using the masks

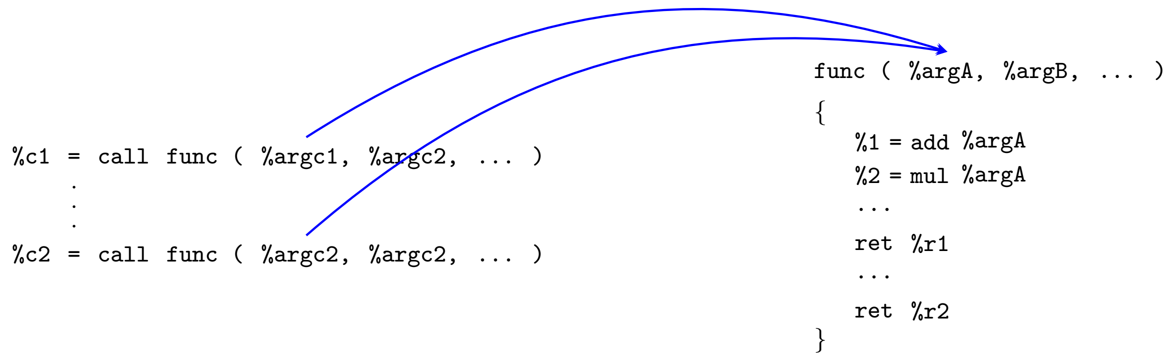


Figure 3.3: Forward transit function of function argument.

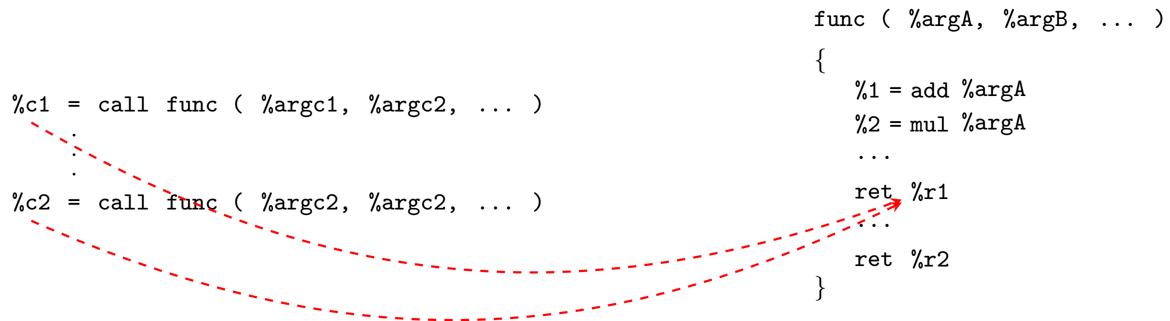


Figure 3.4: Backward transit function of return instruction.

generated by multiple instructions to produce a final mask in the forward transit function; previously, all forward transit functions look only at the instruction at hand. The union of each of the masks from the `ret` instructions of the function is used to determine the mask of the value returned from a call instruction. As with all other forward passes, the newly generated bitmask is intersected with the old bitmask to determine a more optimized bitmask.

In the forward pass of a function argument, we update the bitmask of the function argument based on the bitmask of the input values used to call that function (Figure 3.3). Since a function can be called at various locations in the program with different input values, in order to make the update, we look through all other callsites of this function and take the union of the bitmasks of all of their input values. Again, the newly generated bitmask is intersected with the old bitmask of the argument to produce a more optimized

bitmask.

A `ret` instruction is a control flow instruction which specifies a value to return to the calling function as its operand; it does not produce an output value. Since it does not define a variable, there is no forward transit function associate with it.

In the backward transit function of the `call` instruction, we update the bitmask of the input value used to call the function based on how it is used in the function. Since the function argument holds the bitmask of the input variable within the scope of the function, it is used to update the bitmask of the input value passed to the function as an argument (arrow 2 in Figure 3.2). This is analogous to updating the operand of an arithmetic operation based on the output of that instruction (i.e. how it is used in an instruction is a relation from the output to the operand, versus how it is used in a function is a relation from the function argument to the function parameter).

A function argument is not an instruction, so it is never a user of a value, hence there is no backward transit function for it.

In the backward transit function of the `ret` instruction, we take the union of all the output values from all call sites of the function to generate a reduced bitmask (Figure 3.4). This bitmask is then used to determine what the minimum acceptable bitmask would be for the value being returned.

3.3.3 Bitmask Propagation for Memory Operations

The main goal of propagating bitmasks through memory operations is to open the possibility of reducing the sizes of arrays. In this optimization, we perform analysis on all elements of an array that are read and/or written to. The datapath width of the surrounding circuit that uses the value from a `load` instruction will determine the minimium width required to read from memory. Likewise, the datapath width surrounding the circuit that produces a value for a `store` instruction to write to memory will determine the minimum width required of that memory location. The final size of the array elements

will take the union of the bitmasks associated with all reads and writes to the same array.[1] A global variable can be considered as a special case of an array with a single element.

To propagate bitmasks through memory operations, we define forward and backward transit functions for `load` and `store` instructions. Generally, the forward transit function of an instruction uses the mask of the operand register to update the mask of the output register. For a `load` instruction, this translates to updating the width of the output register based on the width of elements inside the array. The `load` instruction has the following form, where `reg` denotes the output register and `mem` denotes the memory address:

$$reg = load(mem) \tag{3.1}$$

Since a `load` instruction retrieves a value from memory and insert it into a register, the forward transit function of `load` returns the bitmask associated with the elements stored in the memory. Here, the mask of the value in memory refers to the mask associated with all possible values that can be read from this specific `load` instruction. For each array, we use separate bitmasks for `load` and `store` instructions since there exists the possibility that all reads of elements in an array use a different subset of bits from all writes to the same elements in the array. The forward pass of `load` is therefore defined as follows, where M_{reg} is the bitmask of the output register, M_{mem}^{load} is the bitmask of the contents inside the array associated with a read operation:

$$M_{\text{reg}} = \bigcap \left(M_{\text{mem}}^{\text{load}},\ M_{\text{reg}}^{\text{old}} \right) \tag{3.2}$$

As before, the forward pass on the `load` instruction includes the intersection of the original bitmask and the reduced bitmask after the forward transit function.

[1] We analyze the minimum width necessary for the array but do not currently modify the memory instantiation algorithm. As a result, bitmask propagation of memory operations currently only saves area for the `load` and `store` instructions but not the storage location.

The forward pass of the `store` instruction determines the width of the array elements based on the values being written. The `store` instruction has the following form:

$$mem = store(reg) \tag{3.3}$$

Since there may be multiple `store` instructions used to write into the same elements of the same array, the forward transit function therefore visits all other `store` instructions to the same array and computes the union of the masks of each of their register values. We define the forward pass of a `store` instruction as follows, where M_{reg} is the bitmask of the variable being written to memory and M_{mem}^{store} is the bitmask of the contents in memory associated with a write operation:

$$M_{\text{mem}}^{\text{store}} = \bigcap \left(\bigcup \left(\ M_{\text{reg}} \text{ of all stores } \right), \ M_{\text{mem}}^{\text{store old}} \ \right) \tag{3.4}$$

Generally, the backward transit function of an instruction uses the mask of the output register to infer the mask needed by the operand register. For the `load` instruction, this requires us to use the bitmask of the output register to determine the bitmask of the array elements. Mirroring the forward transit function of the `store` instruction, determining the bitmask of the array elements involves visiting all other `load` instructions that access the same array. The union of the bitmasks of all their registers are used as the reduced mask. The backward transit function of the `load` instruction is defined as follows:

$$M_{\text{mem}}^{\text{load}} = \bigcap \left(\bigcup \left(\ M_{\text{reg}} \text{ of all loads } \right), \ M_{\text{mem}}^{\text{load old}} \ \right) \tag{3.5}$$

For the `store` instruction, we determine the bitmask of the operand register based on the bitmask of the array elements. The backward transit function is therefore the

intersection of the register's original mask and the array elements' mask, defined as:

$$M_{\text{reg}} = \bigcap \left(M_{\text{mem}}^{\text{store}},\ M_{\text{reg}}^{\text{old}} \right) \tag{3.6}$$

In this analysis, we determine the minimum width requirements of the array elements based on how they are accessed in the surrounding circuit. In the current *implementation*, we make two simplifications since we cannot currently take advantage of the benefits of array-width reduction without making drastic changes to the memory instantiation algorithm of LegUp. The first simplification involves combining the bitmask for the reads and writes to the same array element by taking the union of their masks:

$$M_{\text{mem}} = \bigcup \left(M_{\text{mem}}^{\text{load}},\ M_{\text{mem}}^{\text{store}} \right) \tag{3.7}$$

The second simplification requires that elements of an array have the same width. This means that a single bitmask is associated with each array. The only arrays that are eligible to be reduced are those that always `load` and `store` reduced-width values. These two simplifications are compatible with LegUp's default method for synthesizing arrays into FPGA storage. Since each array is stored in a logical RAM, which has a fixed word width and a fixed word depth, no BRAM usage savings would be achieved even if we were to analyze the width of each array element individually.

3.3.4 User-Declared Arbitrary Bitwidths

Static compiler analysis is conservative by design, and therefore, its ability to reduce circuit area without external aid is limited. At the other end of the spectrum is dynamic profiling, which gathers the exact range of values a variable can take on throughout the application life time; the datapath width can be minimized for the specific set of application inputs used in profiling. Gort and Anderson [18, 19] profile the exact values used for every variable in a circuit to evaluate the maximum possible area reduction

achievable. They observe an average of 54% reduction in LUTs, compared to the 18% reduction achieved by only applying static compiler analysis. However, to achieve this degree of area reduction, much of the circuit is optimized away, and the final circuit likely only works for specific inputs. Our goal is to achieve an HLS area reduction somewhere in the gap between aggressive dynamic profiling and conservative static analysis, while keeping the circuit general and operable for all possible inputs. A straightforward way to push the area reduction into this middle region is to have the HLS user specify a reduced width for the variables that they know for certain will not need the full width of the standard data type. From this perspective, we can consider dynamic value profiling as an extreme case where the user specifies the minimum width required for every variable in the program exercised with specific inputs. Conservative static analysis is the other extreme where the designer does not provide any information about the variables in the program.

Enabling user-declared arbitrary-width datatypes involves providing a means of declaring non-standard bitwidths in the input source code and modifying LegUp's back-end to be made aware of this extra information. To maximize the usefulness of user-declared arbitrary bitwidths, the framework should deliver the following key objectives:

1. Arbitrary width datatypes should be easy and intuitive to use in `C` code.

2. Programs using arbitrary width datatypes should be capable of being compiled as a valid software program to run on the CPU for portability and quick verification of functionality.

3. Arbitrary bitwidth information should be represented in such a way that it can be passed from the source code all the way to the Verilog backend without interfering with nor being destroyed by intermediate compiler optimizations.

4. Arbitrary bitwidth information for a `C` program variable should be propagated to all IR variables that correspond to it.

5. The user-declared width should not be overridden by the compiler to a wider width due to rules enforced by standard compilers (e.g. LLVM enforces that all operands of an instruction have the same width), as discussed in Section 2.2.

To address all of the above objectives, we leverage metadata and source debugging information from the LLVM framework to enable support for user-declared arbitrary bitwidths.

At the user end, LegUp provides a `types.h` header file that defines all the arbitrary-width datatypes. Each arbitrary-width datatype is defined as a standard-width datatype with an extra attribute attached to specify its width (snippet shown on Listing 4). Users simply include this `types.h` header file in their `C` program and declare arbitrary width types similar to standard types (example shown in Listing 5). To the user, the arbitrary datatypes behave no different than the standard datatypes, achieving objective 1 above. Since these attributes are custom, they are ignored by all compilers other than LegUp (i.e. the custom datatypes appear as standard datatypes), achieving objective 2 above.

When the `C` program goes through the Clang frontend, attributes are converted into calls to the LLVM intrinsic functions `llvm.var.annotation` and `llvm.global.annotations` for local variables and global variables respectively, at the location they are declared. Unfortunately, these annotation intrinsic functions operate on the IR variables directly, thus disabling any other LLVM optimizations on the variable. To alleviate this, we add an LLVM pass that extracts the width information from the annotation intrinsic function calls and attaches it onto the `llvm.dbg.declare` and `llvm.dbg.value` intrinsic function calls as metadata. Within the LLVM framework, extensive effort has been put into ensuring that the debug intrinsic functions do not interfere with other optimizations and are automatically optimized along with the rest of the program. By removing the annotation intrinsic function calls and using the debug intrinsic function calls, we are able to achieve objectives 3 and 4 on our list.

Note that since the bitwidth information is stored as metadata, it does not incur

```
#define uint1 __attribute__((annotate("bitwidth=1"))) unsigned char
...
#define uint8 __attribute__((annotate("bitwidth=8"))) unsigned char
#define uint9 __attribute__((annotate("bitwidth=9"))) unsigned short
...
#define uint16 __attribute__((annotate("bitwidth=16"))) unsigned short
#define uint17 __attribute__((annotate("bitwidth=17"))) unsigned int
...
#define uint32 __attribute__((annotate("bitwidth=32"))) unsigned int
#define uint33 __attribute__((annotate("bitwidth=33"))) unsigned long long
...
#define uint64 __attribute__((annotate("bitwidth=64"))) unsigned long long
#define int1 __attribute__((annotate("bitwidth=1"))) char
...
#define int8 __attribute__((annotate("bitwidth=8"))) char
#define int9 __attribute__((annotate("bitwidth=9"))) short
...
#define int16 __attribute__((annotate("bitwidth=16"))) short
#define int17 __attribute__((annotate("bitwidth=17"))) int
...
#define int32 __attribute__((annotate("bitwidth=32"))) int
#define int33 __attribute__((annotate("bitwidth=33"))) long long
...
#define int64 __attribute__((annotate("bitwidth=64"))) long long
```

Listing 4: Snippet of `types.h` provided by LegUp to define arbitrary width datatypes.

any logic overhead in terms of instructions. In addition, we do not need to modify the actual data type in the IR, bypassing any situation where unnecessary bits are wasted because of LLVM's requirement for all operand widths to be the same type and width as the output. This metadata is extracted and used to initialize the width requirements for the automatic bitwidth minimization analysis described in this chapter, and the final minimum width is conveyed to LegUp's Verilog backend. Since the Verilog backend does not have the same requirement as LLVM where operands must have the same width, objective 5 is addressed.

3.4 Evaluation

We evaluate the effectiveness in area reduction of each of the four components in the BWM framework outlined in Section 3.3. For the three compiler analysis components,

```
#include "../../legup-library/legup/types.h"
#define SIZE 20

volatile int11 A1[SIZE][SIZE] = {0, 1, 2, 3, 4, 5, 6, ... 396, 397, 398, 399};
volatile int11 B1[SIZE][SIZE] = {400, 401, 402, 403, ... 796, 797, 798, 799};

volatile int resultAB1[SIZE][SIZE];

int multiply(int i, int j) {
    int k, sum = 0;
        for(k = 0; k < SIZE; k++) {
        sum += A1[i][k] * B1[k][j];
    }
    resultAB1[i][j] = sum;
        return sum;
}

int main(void) {
    int i, j;

    unsigned long long count = 0;
    for(i = 0; i < SIZE; i++) {
        for(j = 0; j < SIZE; j++) {
            count += multiply(i, j);
        }
    }

    return count;
}
```

Listing 5: Matrix multiply source code using non-standard width types.

we first examine the total number of bits reduced in the program's instruction stream, then we compare the post-synthesis resource utilization reported by Intel's Quartus II CAD tool. To illustrate the potential benefit of allowing user-declared arbitrary bitwidth specification, we present the results of a case study.

3.4.1 Methodology

Each component of the compiler analysis is evaluated on the CHStone benchmark suite [22], which consists of 12 benchmarks that span a variety of application domains, including arithmetic, media processing, security and microprocessor simulation. The baseline for our comparison is the circuit generated by LegUp with all the default HLS

settings, function inlining disabled, and no BWM optimization applied. Function inlining is disabled to ensure that there are function calls present to observe any potential effects of propagating bitmasks across functions.

To obtain the total number of bits reduced, we sum the number of bits used for every variable in the program's dataflow graph and normalize this to the sum of every variable's full native width. To obtain the post-synthesis resource utilization numbers, the Verilog RTL generated by LegUp is pushed through Intel's Quartus II CAD tool chain, targeting the Cyclone V family of FPGAs, and we inspect the numbers reported after place and route. We also present each circuit's maximum operating frequency ($Fmax$), obtained from Intel's TimeQuest static timing analysis tool, since we can expect the circuit's critical path to shorten when circuits reduce in size. Finally, all circuits are simulated with Mentor Graphics' ModelSim to ensure that functional correctness is maintained.

3.4.2 Bit Savings

Table 3.4 and Figure 3.5 show the percentage of bits reduced in the program's instructions for the CHStone benchmarks. The first column represents the bitwidth analysis scheme using only the propagation rules for bitwise and arithmetic operations, as initially proposed by Gort and Anderson. The second column corresponds to applying the propagation rules for cross-function boundary optimization, in addition to the core BWM. The third column corresponds to applying the propagation rules for memory related instruction, in addition to the core BWM. The fourth column combines the second and third columns.

From these results, we see that the core BWM analysis saves 21% of bits on average in the bitwise and arithmetic instructions of the CHStone benchmarks. The cross function boundary bitmask propagation introduces an additional 2.1% of bit savings on top of the core BWM analysis, compared to the baseline, having the most significant effect on the df benchmarks. The df benchmarks are the only ones where the bitmask propagation is

bench	core BWM	func	mem	func+mem
adpcm	11.625	12.488	24.555	25.417
aes	23.366	23.366	28.069	28.069
blowfish	29.637	29.637	31.018	31.018
dfadd	14.845	20.995	14.909	21.173
dfdiv	18.365	21.653	18.421	22.044
dfmul	21.077	25.244	21.144	25.757
dfsin	12.953	18.051	12.986	18.531
gsm	24.648	24.796	25.414	25.568
jpeg	16.228	20.316	18.815	24.707
mips	34.808	35.087	37.935	38.214
motion	22.966	23.849	23.461	24.344
sha	21.651	21.940	23.290	23.580
AVERAGE	21.014	23.118	23.335	25.702

Table 3.4: Percentage of bits reduced by different bitmask propagation scheme.

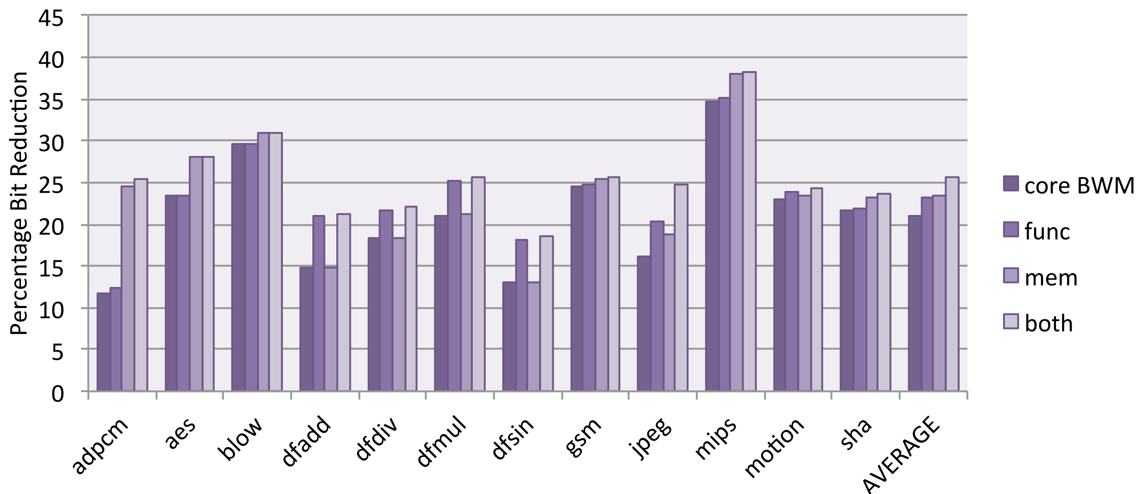


Figure 3.5: Percentage bit reduction for different bitwidth propagation scheme.

able to propagate through the function arguments and then all the way back to the return value. The bitmask propagation through memory operations introduces an average of 2.3% additional savings on top of the core BWM analysis, compared with the baseline, most of which comes from the `store` instruction. Since the two bitmask propagation schemes are orthogonal to each other, the number of bits saved when both optimizations are applied are added together, totalling an average of 4.7%. On average, we are able to reduce 25.7% of the bits used in a program's instructions overall, when the compiler analysis part of the BWM framework (Sections 3.3.1, 3.3.2 and 3.3.3) is active.

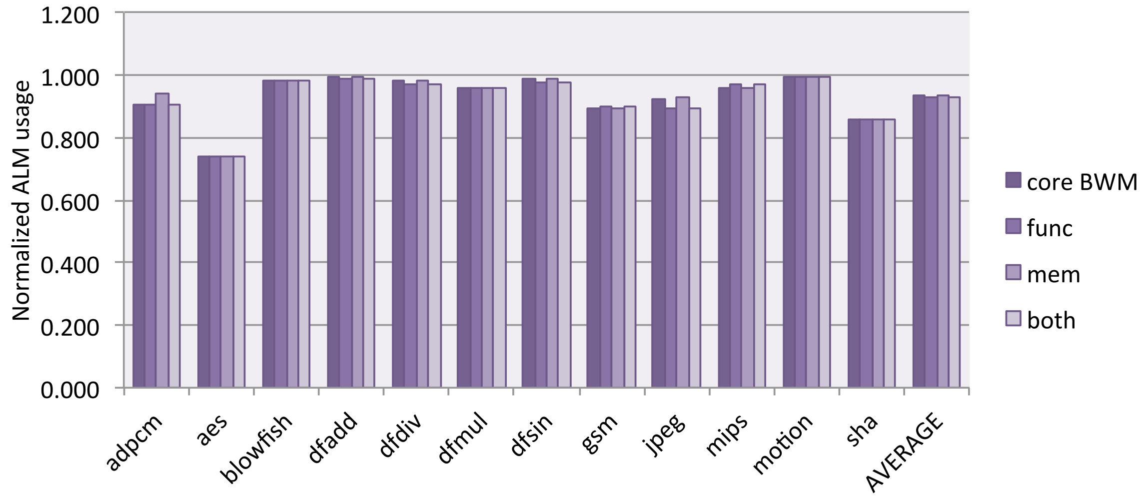


Figure 3.6: Normalized ALM usage for different bitwidth propagation scheme

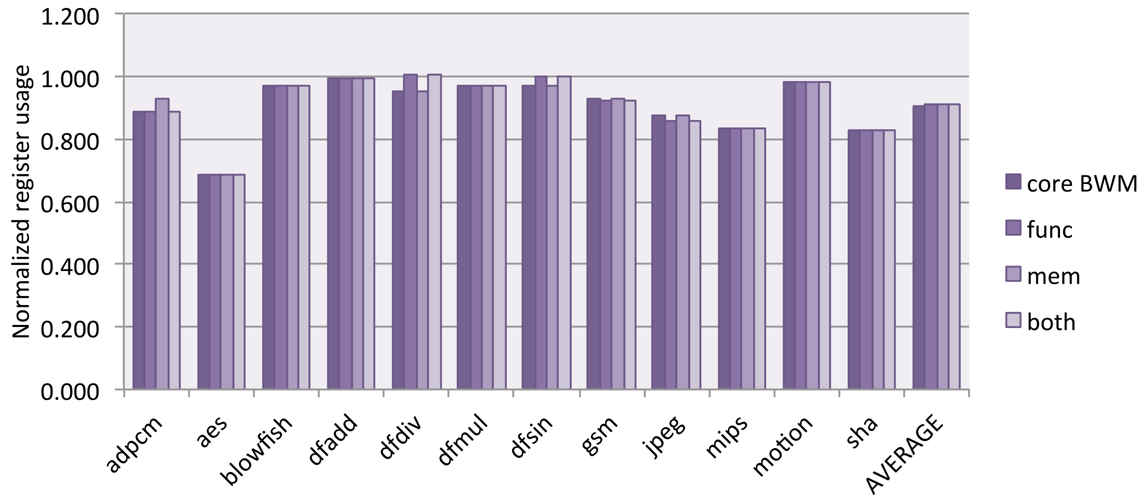


Figure 3.7: Normalized register usage for different bitwidth propagation scheme

3.4.3 Resource Utilization

Figures 3.6, 3.7, 3.8 and 3.9 show the post-synthesis resource usage per benchmark corresponding to each of the four columns in Table 3.4. The normalized Fmax for each benchmark is shown in Figure 3.10. The raw numbers, in table form, can be found in Appendix A.

Since the Quartus II CAD tool already performs several bit-level optimizations under the hood during technology mapping and place and route, we find that the bit reductions

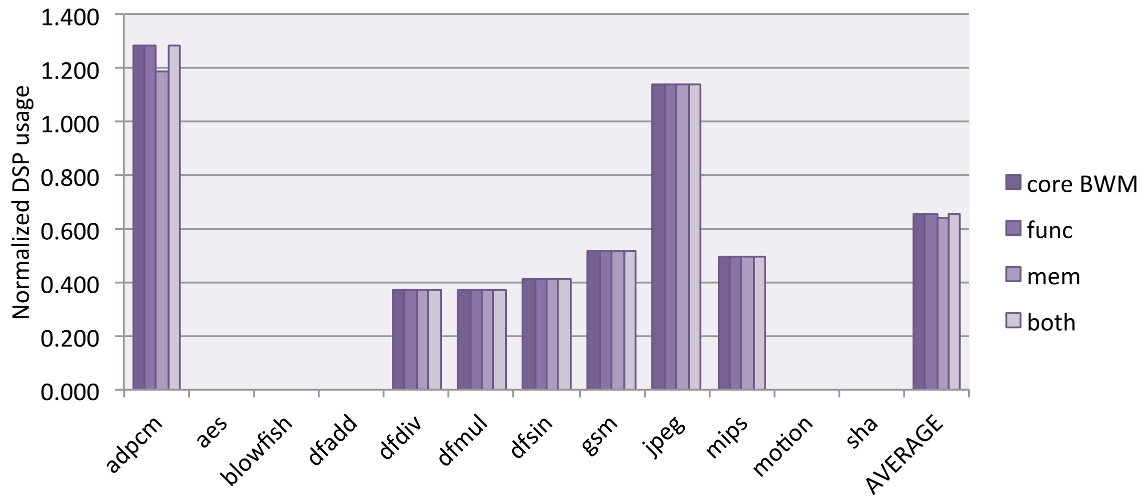


Figure 3.8: Normalized DSP usage for different bitwidth propagation scheme

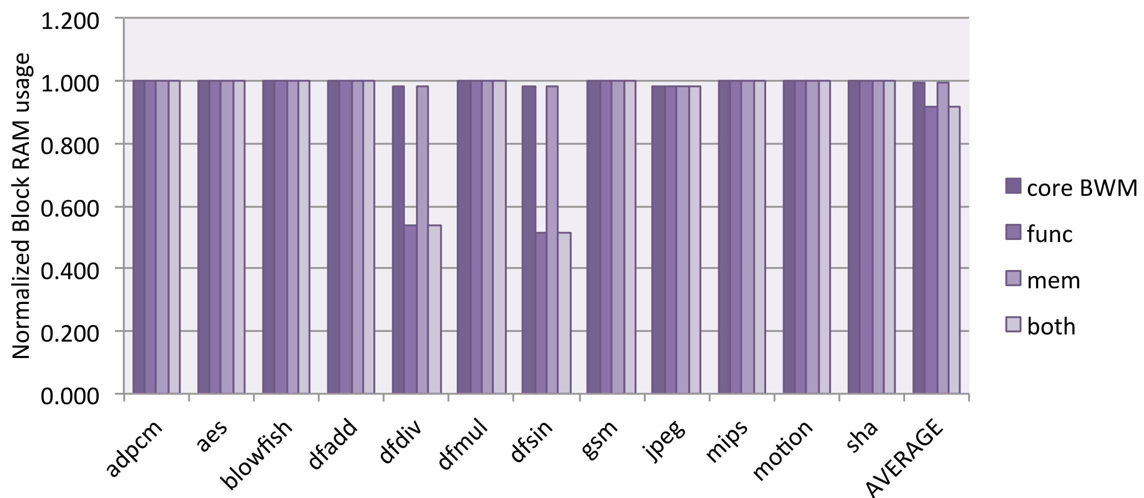


Figure 3.9: Normalized Block RAM usage for different bitwidth propagation scheme

in the instructions do not translate one-to-one to the reductions in resource usage. The 21% bit reduction in the core BWM propagation scheme yields an average of 6.7% reduction in ALM usage, 9.2% reduction in register usage, 34.2% reduction in DSP usage, and 0.4% reduction in block RAM usage. Much of the ALM, register, and DSP usage reductions come from the reduction in the size of the functional units needed to perform arithmetic computations in the circuit. In Figure 3.8, we see an increase in DSP usage for adpcm and jpeg compared with the baseline where BWM is not applied. This is

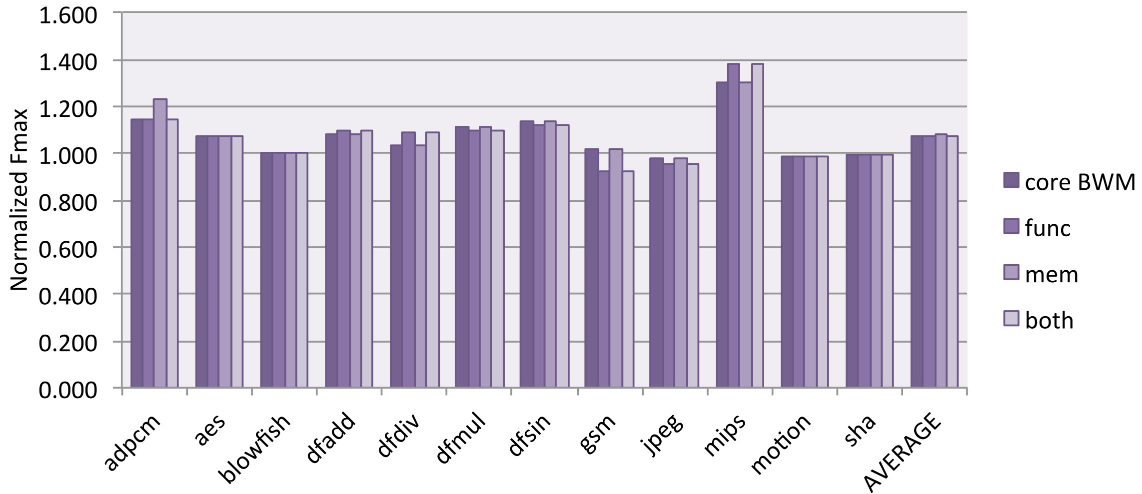


Figure 3.10: Normalized Fmax for different bitwidth propagation scheme

attributed to the binding algorithm in LegUp not being able to share functional units between different operations with different operand widths. In Figure 3.9, we see a reduction in block RAM usage for dfdiv, dfsin and jpeg. In this work, we do not modify LegUp's memory instantiation algorithm to take reduced widths into account, so all arrays are still instantiated with their full native width. The three cases where we observe a reduction in BRAM usage are attributed to optimizations in Quartus' mapper and fitter algorithms.

With function propagation enabled, we observe a decrease in ALM usage in dfadd, dfdiv, dfsin and jpeg, when compared with the core BWM analysis. This is justified by the observation that these four benchmarks in particular see a greater number of bits reduced with function propagation enabled. Of these four benchmarks, dfdiv and dfsin see an increase in register usage and a decrease in block RAM usage, which suggest that there is a shift in how logic is implemented and resources are traded off against each other. With a large decrease in BRAM usage, a slight increase in register usage, and a less than expected reduction in ALM usage, this indicates that some logic that used to be stored as memory bits are now computed and stored in registers. Jpeg sees a decrease in register usage in addition to the decrease in ALM usage, but takes a hit in the Fmax

of the overall circuit. The reduction in Fmax is due to a change in the critical path of the circuit, which traverses a different but longer carry chain of the circuit. On the other hand, dfadd exhibits only a reduction in ALM usage and no change in any other resources; with less logic needed to be implemented, Fmax is improved overall.

With bitmask propagation across memory operations enabled, resource usage stays mostly the same as the core BWM, with the exception of adpcm. For adpcm, we observe increases in ALM and register usage as well as a decrease in DSP usage, indicating a change in the resource used to implement logic. In this case, the change is likely due to extra bits being optimized away from adders around the store operation, and Quartus deciding that using the logic fabric is better than using DSP. Similar to our evaluation of function propagation, the benchmark with the largest change in resource usage corresponds to that with the greatest bit reduction.

With all parts of the BWM framework enabled, the 25.7% bit reduction in instructions translates to an average of 7.1% reduction in ALM usage, 8.8% reduction in register usage, 34.2% reduction in DSP usage, and 8% reduction in block RAM usage. Fmax is improved by 7.3% as a result of smaller circuitry.

3.4.4 Bit and Resource Savings with Default HLS Settings

In the results presented above, function inlining is disabled to ease in observing any potential effects of propagating bitmasks across function boundaries. In Table 3.5 and Figure 3.11, we show the impact of the BWM framework on the default push-button setting in LegUp where inlining is enabled. With the BWM framework applied, we see an average reduction of 12.1% in ALM usage, 10.6% in register usage, 8.9% in BRAM usage, and 34.8% in DSP usage. Because of the reduction in circuit size, we also see a 7.6% improvement in the average operating frequency of the circuit. The numbers presented here are more representative of the benefits that can be reaped if the HLS user were to simply present a `C` benchmark and toggle the Tcl parameter to turn on the BWM

optimization.

bench	% bit reduction	ALMs	Reg	RAM	DSPs	Fmax
adpcm	20.754	0.455	0.809	1.000	1.000	1.128
aes	29.021	0.715	0.700	1.000	-	1.017
blowfish	32.272	0.959	0.947	1.000	-	1.056
dfadd	31.831	1.000	0.969	1.000	-	1.008
dfdiv	31.466	0.933	1.016	0.484	0.375	1.046
dfmul	37.315	0.943	0.953	1.000	0.375	1.253
dfsin	25.436	0.948	1.003	0.458	0.400	1.080
gsm	32.164	0.898	0.923	1.000	0.598	1.038
jpeg	20.615	0.925	0.869	0.985	1.318	1.026
mips	39.548	0.953	0.788	1.000	0.500	1.256
motion	18.352	0.972	0.943	1.000	-	1.016
sha	26.470	0.851	0.808	1.000	-	0.989
AVERAGE	28.770	0.879	0.894	0.911	0.652	1.076

Table 3.5: Percentage of bit reduction and normalized resource usage and Fmax for push-button LegUp setting on CHStone Benchmarks

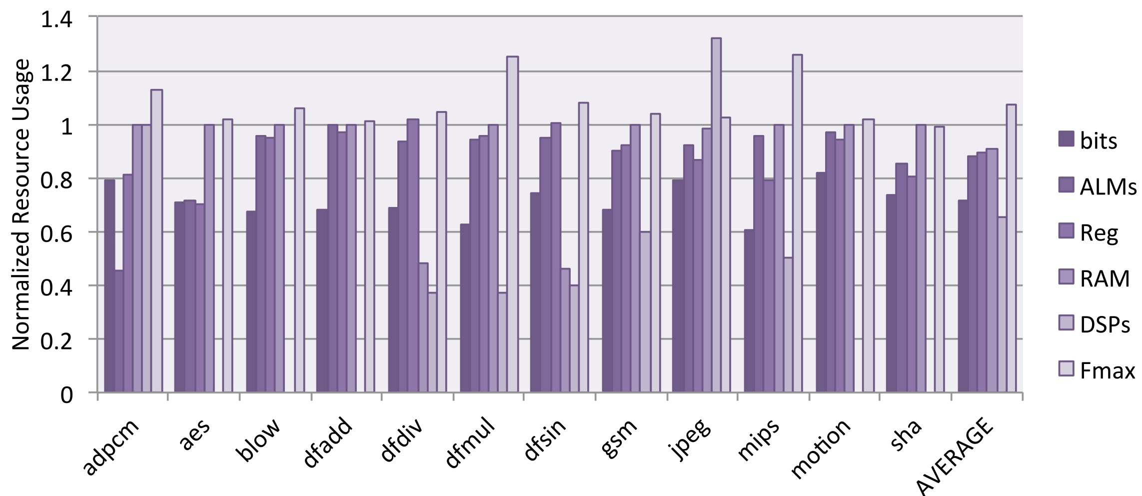


Figure 3.11: Normalized resource usage and Fmax for push-button LegUp setup with BWM framework

3.4.5 Case Study of User-Declared Bitwidths

To demonstrate the benefits of allowing users to specify the width of a variable in C, we evaluate area and performance of an example matrix multiply application. The `C` code using non-standard types is shown in Listing 5. In this example, we know ahead of

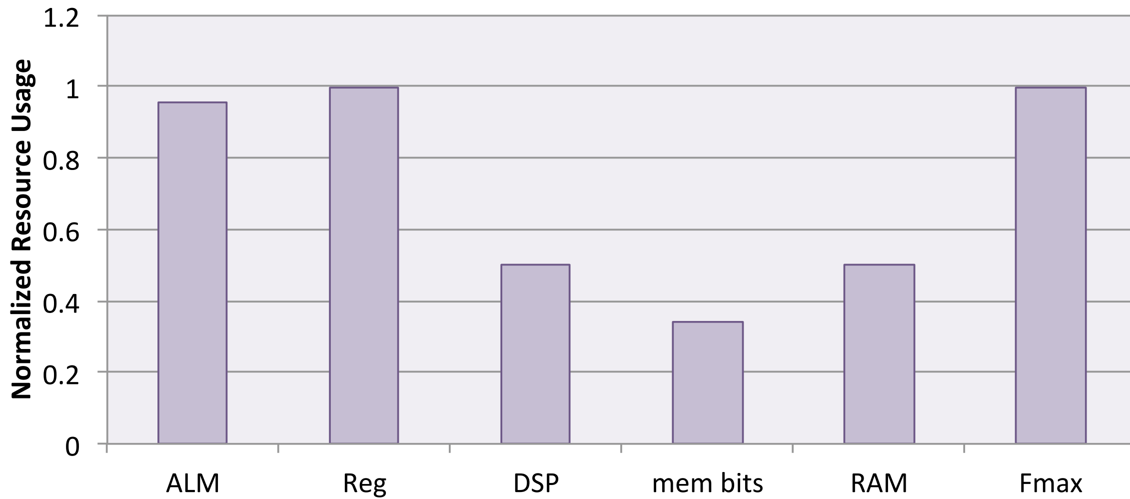


Figure 3.12: Normalized resource usage and Fmax for matrix multiply using non-standard width variable type

time that we are never going to assign the input vectors `A1` and `B1` a value greater than 800. This means that all elements within these two arrays only need 11 bits to correctly represent their value. The only change to the source code that is necessary is to include the LegUp provided header file (line 1) then use the desired non-standard width type in the variable declaration (lines 3 and 4). Figure 3.12 shows the normalized resource usages and Fmax. By informing LegUp that the input vectors are 11 bits instead of 32 bits, the BWM framework is able to reduce the size of the multipliers and the width of the datapath more aggressively.

3.5 Conclusion

In this chapter, we describe the bitwidth minimization framework that we implement in the LegUp HLS tool. The BWM framework provides a compiler pass that automatically infers reduced-width datapaths wherever possible in the program and a means for HLS users to use arbitrary-width datatypes in the source code. We show that with the CH-Stone benchmark suite, the BWM framework on average reduces ALM usage by 12.1%, register usage by 10.6%, BRAM usage by 8.9%, and DSP usage by 34.8%. The smaller

circuit size also results in an average improvement of 7.6% in the FMax of the circuits. Overall, we see that the ability to identify reduced-width datapaths in the circuit at a high level further improves the area reduction, on top of the numerous optimizations that Quartus II already performs under the hood.

Chapter 4

Sensei: Area Reduction Advisor

The BWM framework described in Chapter 3 enables HLS designs to utilize the underlying hardware's support for arbitrary-width datapaths. In this chapter, we shift the focus to *how* the users of HLS can take advantage of the arbitrary-width datapath support in their designs.

The typical coding style that programmers are accustomed to when developing with a high-level language such as `C` is targeted towards a standard processor pipeline with fixed-width datapaths. With the traditional design paradigm, relatively little attention is given on figuring out the exact minimum number of bits required by each variable since the width of datapaths inside the processor pipeline is fixed at coarse granularities (to take advantage of memory bandwidth). Programmers tend to think of types of different width as distinct quantized entities, as opposed to thinking of them on a continuous spectrum (e.g. adding two 4-bit values results in a 5-bit sum). Using a high-level language to specify hardware via HLS, the designer should bear in mind that they are no longer coding for a regular processor pipeline and should be more cautious of the widths of program variables.

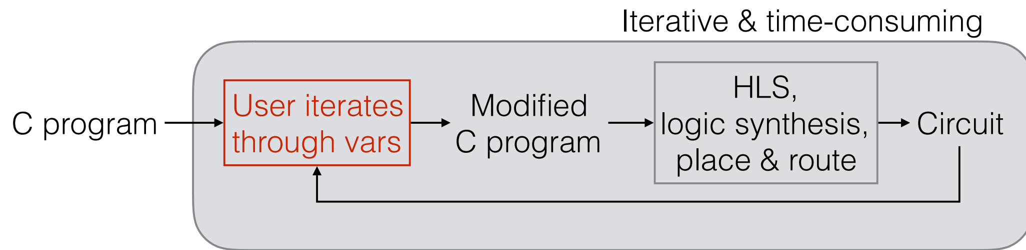
In this chapter, we propose *Sensei*, a framework that analyzes the input program and generates a ranked list of program variables and their impact on post-synthesis area.

By providing a ranked list of variables, we make it easier for designers to reduce the area of their circuit by constraining the widths of impactful program variables. For designers coming from a software background, where relatively little attention is paid to the width of datatypes, Sensei also serves as a reminder that the area of the circuit can be improved by constraining the width of program variables. Moreover, Sensei provides design space exploration (DSE) assistance where the user is relieved of the burden of having to manually iterate through all the program variables and guess the post-synthesis area impact of each of them. Sensei focuses the designers' effort on reasoning about *how many bits a variable really needs*, as opposed to figuring out which variable provides the most post-synthesis area impact.

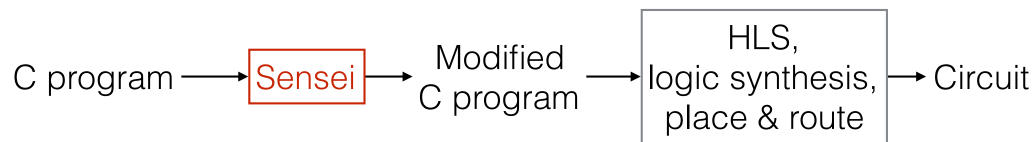
4.1 Overview

Figure 4.1a depicts the normal DSE steps for tuning an HLS-generated circuit using program variable bitwidths. The designer constrains the width of `C` program variables, synthesizes the circuit to obtain the area impact, decides which variable to change next and repeats the process. Not only is it time consuming to undergo full synthesis multiple times, but the sequential nature of the iterative approach also makes DSE time-consuming. Another downside of this iterative DSE approach is that it relies heavily on designer intuition and effort to come up with what to try next. For a less experienced designer, this can be an extremely time intensive trial-and-error process.

Contrast this with the proposed advisor-based approach, shown in Figure 4.1b, where Sensei takes in the original `C` source code as input and generates a list of variables in descending order of their impact on final area. Instead of having to manually iterate through all variables until the desired area reduction is achieved, and synthesizing each iteration along the way, Sensei guides the designer by reducing the problem from: *which variables can be bitwidth-constrained and yield high area savings?*, to a simpler problem:



(a) DSE with iterative approach



(b) DSE with Sensei

Figure 4.1: Design space exploration steps with (a) a naive iterative approach and (b) Sensei.

can this particular variable be constrained?. Though the user still needs to consider correctness, by reducing the scope of the problem, they no longer have to "guess" about how reducing the width of a variable may affect the final circuit area. They can focus their attention on the ranked list of variables, which Sensei predicts are impactful from the area angle. The additional guidance improves HLS ease-of-use, reducing design time and cost regardless of the experience level of the designer, allowing an area target to be met in less time.

Additionally, Sensei can be incorporated into an iterative approach if desired. The designer can present a modified program to the advisor to generate a new list of variables if they wish to squeeze out more area savings by examining whether any new variables will become an effective source of area reduction after a first batch of changes. This flow removes full synthesis from the design iterations, enabling the designer to explore a wide range of designs more quickly. Exploration can be done in the order of seconds, compared to the many hours (or longer) it would take to iteratively synthesize each design. Overall, Sensei increases productivity by focusing the designer's attention on

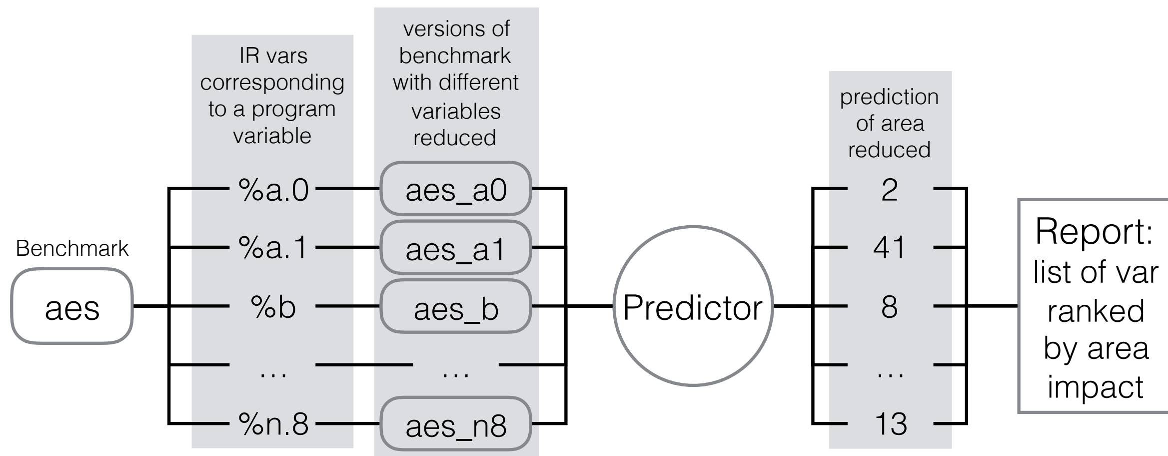


Figure 4.2: Example of Sensei ranking the area impact of variables in a benchmark named `aes`.

fruitful modifications.

Figure 4.2 depicts the setup of how Sensei ranks the area impact of a program's variables. For a given `C` program, Sensei automatically enumerates all IR variables that correspond to `C` program variables. For each IR variable, it generates a *hypothetical* version of the original program's DFG with the IR variable's width reduced by a fixed pre-defined amount. Then, LegUp's bitwidth minimization framework is executed to propagate the hypothetical bitwidth reduction throughout the DFG, thereby allowing bitwidths of other variables to also be reduced. The area reduction predictor in Sensei then generates a prediction of the post-synthesis area reduction afforded by the hypothetical bitwidth-reduced DFG. In the end, Sensei gathers its predictions for all IR variables and generates a report that ranks them in order of their impact on area.

The key technical challenge is in predicting the post-synthesis area reductions associated with a candidate variable bitwidth change. For this, we implement two versions of the area predictor: one *analytical* and one employing a *CNN*. Both methods rely solely on information available at the compiler stage during HLS.

4.2 Related Work

Sensei provides guidance on how to fine-tune the software description of circuits synthesized via HLS. Sensei's functionality thus facilitates and eases design space exploration in HLS, which prior work has tackled from several angles. So et al. [38] focus on loop nest computations and present a compiler-based DSE algorithm that applies loop transformations to obtain different variants of the design; the best design is selected by using synthesis estimation techniques that balance the computation intensity with the memory access rate. Kulkarni et al. [25] iteratively apply aggressive compiler tranformations on the dataflow graph of a program and use compile-time estimations to evaluate the area impact of varying design alternatives.

Many commercial HLS tools provide support for directives in the form of pragmas as a means for the user to fine-tune specific parts of their design. Carrion Schafer and Wakabayashi [35] use GA-ML, a genetic algorithm based on a predictive model generated from machine learning techniques, to find the Pareto-optimal set of directives for a given design. Liu and Carloni [28] employ the random forest learning model to iteratively refine the Pareto set of HLS directives. Instead of randomly sampling the design space, they construct the training set using transductive experimental design, an algorithm designed for active learning that selects representative yet hard to predict designs. Lo and Chow [30] consider the problem of automatically selecting an optimal set of HLS directives using sequential model-based optimization, which is a method for selecting hyperparameters in highly-parameterized optimization algorithms. To the authors' knowledge, this work is the first to apply CNNs in HLS for prediction.

4.3 Convolutional Neural Networks

A convolutional neural network is a machine learning technique that is widely used in image recognition and other tasks [24]. In this section, we provide a brief overview of

the different components of CNNs, with emphasis on the terminologies and parameters used in this thesis work. Complete details of all possible components of the CNN can be referred to in past literature [27, 36].

4.3.1 Architectural Components

A typical CNN consists of multiple *layers*, each of which performs a transformation on the data passing through it. The entry point of the CNN is the input data layer, which takes in the raw input data such as an image, optionally performs translation and scaling, and generates an output matrix with a fixed height, width, and depth. For example, in the VGG16 network [37], which classifies 224×224 RGB images into 1000 different categories of objects, the input layer accepts as input the red, green, and blue pixel intensities of a 224×224 image and stores them in a 3D matrix with width=224, height=224, and depth=3. We use the term *row* to refer to a specific position in height, *column* to refer to a specific position in width, and *channel* to refer to a specific position in depth.

In a CNN, the majority of computation is spent in the *convolutional layers*, each of which extracts spatial features from its input. To detect a feature centered at a specific location of the input matrix, the convolutional layer uses a *filter* that defines a specific feature. The computational work involves performing a dot product between the filter and the region of the input matrix centered around the location of interest. The values that make up the filter are referred to as the *weights* of a layer; Section 4.3.2 later describes how weight values are obtained in training. The result of the dot product is stored as a single value in the output matrix, where the value indicates the likelihood of the feature being at the location. To search for the presence of a feature throughout the entire input matrix, the same filter is applied to every region centered around every pixel of the input. The resulting output matrix is called a *feature map*, indicating where a feature is detected on an input image. To detect n features, n filters are used and the output matrix of the layer will have a depth of n; that is, there are n output feature

maps.

For example, consider a 3×3 filter designed to find a horizontal line, and a second 3×3 filter designed to find a vertical line. To search for the presence of a horizontal line everywhere on the input, the first filter is applied to every 3×3 region of the input matrix, and each of the dot product results are stored in the first channel of the output matrix. Likewise, to search for the presence of a vertical line everywhere on the input, the second filter is applied to every 3×3 region of the input matrix, and each of the dot product results are stored in the second channel of the output matrix. If the input image contains only has a horizontal line along the centre row of the image and a vertical line along the centre column of the image, the first channel of the output matrix will only have a high value along the centre row and low value everywhere else, and the second channel of the output matrix will only have a high value along the centre column and low value everywhere else.

In addition to the number of features to detect, another parameter often used in the convolutional layer is the *stride* at which the filters are applied. The stride specifies the step size to be used as a filter is swept across an input image, and can typically be specified as two separate parameters, one for the width and one for the height. For example, with a column stride of 2, a filter would be applied starting at every second pixel of an input image. Choosing a larger stride reduces the number of computations necessary by a factor of stride size.

Another computational layer that typically appears in CNNs is called a *fully connected layer*. Unlike the convolutional layer, which applies a filter to a spatially localized region of the input matrix, a fully connected layer uses every value in the input matrix to generate a single output value. To generate one output value, the fully connected layer uses a set of weights equal to the size of the layer's input, and performs a dot product between the weights and the input matrix. Similar to the convolutional layers, the fully connected layer can have multiple sets of weights to detect the presence of multiple

features, each appearing as a different channel in the layer's output vector. However, because the size of the weight filters in a fully connected layer is much larger than the size of the weight filters in a convolutional layer, it is much more memory and compute intensive.

As the CNN is repeatedly presented with different training inputs, the network adjusts the weights for each of the convolutional layers and fully connected layers so the final generated output is as close as possible to the correct output values, called *labels*. In order to learn from mispredicted output values, the output of the CNN is typically connected to a loss layer, which computes the distance between the current output values and the golden/correct output values. In this work, we aim to produce an area value: the number of logic elements (ALMs) reduced for a candidate bitwidth reduction. The typical loss function used for a problem such as ours is the Euclidean loss function, which computes the sum of squares of the difference between the predicted and actual value. We refer to this value as the *sum of square* (SOS) error, and is computed as follows, with N as the total number of samples, x^P as the predicted value, and x^A as the actual value:

$$SOS = \frac{1}{N} \sum_{i=1}^{N} \|x_i^P - x_i^A\|^2 \tag{4.1}$$

A perfect prediction would produce an SOS of 0, which is the objective of the CNN training.

4.3.2 Runtime Parameters of a CNN

There are two phases that a CNN can operate in: *training* and *testing*. The training phase evolves the weights in the network so that they produce predictions closer to the label value. The testing phase makes predictions for input data that are not used during training – i.e. on input data that the CNN has not seen before. In this thesis, we perform various evaluations on the prediction accuracy of the CNN. When the CNN

is in the training phase, every input being fed into the network has a label, and the mispredictions on this input data contribute to the adjustment of weights within the layers of the network. When the CNN is in the testing phase, every input being fed into the network also has a label, but mispredictions on this input do not result in the adjustment of weights in the network, but rather provide an indication on how well the network is able to generalize what it has learned from the training input data to the testing input data. This implies that the testing input data is kept separate from the training input data. Given a database of different inputs with labels, generally 10% to 20% of the entire set is designated as the testing set and the rest is designated as the training set [21].

During the training phase, the entire set of training data is repeatedly presented to the network to allow for gradual convergence of weights in order to generate good predictions. Since different prediction tasks can have different numbers of training samples, it is typical to refer to the length of training by the number of times the entire training set has been presented to the network. One iteration through the entire training set is referred to as one *epoch*. For the networks in this thesis, we train for 1000 epochs to allow us to confirm a saturation in the network's prediction accuracy.

4.4 Sensei Predictor

As mentioned in Section 4.1, the predictor in Sensei takes a bitwidth-reduced DFG as input and generates a prediction on the area savings captured by the DFG. In our implementation, we use ALM reduction as the metric for area savings (further details discussed in Section 4.5). In this section, we discuss what information in the DFG is used and how we use this information to generate the ALM reduction prediction. Section 4.4.1 describes how the prediction is obtained in the analytical predictor, and Section 4.4.2 describes how the prediction is obtained in the CNN-based predictor.

Operation	#ALMs reduced per bit
ADD	0.5
SUB	0.5
MUL	1.4375
SDIV	41.5
UDIV	39.1875
SMOD	41.5
UMOD	39.1875
AND	0.5
OR	0.5
XOR	0.5
SHL	2.875
LSHR	2.875
ASHR	2.875
SELECT	0.5
ICMP	0.4375

Table 4.1: Area Model for functional units[1]

4.4.1 Analytical Predictor

The high-level idea of the analytical approach is to generate an area reduction prediction for each node in the DFG then sum up the area reduction for all nodes to obtain the total amount of area reduced in the bitwidth-reduced DFG. This approach bears similarity with typical approaches used in HLS for area and delay estimation. To predict the amount of area reduced for a node, we multiply the number of bits reduced by the area-reduction factor associated with the operation type of the node. Listed in Table 4.1, the area reduction associated with each operation type is obtained by creating a microbenchmark for each operation type in isolation and profiling the ALM reduction when the bitwidth is reduced from 32 bits to 16 bits.

The analytical method assumes a smooth continuous linear relation between the number of bits reduced in each operation and the number of ALMs reduced in the post-synthesis area. This is, however, not always the case with FPGA design where the basic logic element has a fixed size and multiple outputs. As such, gradual changes in an

[1]See Appendix B for the detailed resource breakdown of each operation.

operation's bitwidth may or may not affect resource usage. There is a "ratchet effect", wherein gradual increases/decreases in bitwidth may not effect resource usage until a certain width is reached, at which time a change in resource usage is observed. To better model this behaviour and offer a framework that will perform relatively well across different FPGA architectures, we employ CNN-based prediction, as discussed in the next section.

4.4.2 CNN-Based Predictor

In contrast to the analytical predictor, where we explicitly define a linear relation between bitwidth reduction in the DFG and the post-synthesis area reduction, the CNN-based predictor *learns* the relation from the training data. CNNs are designed to automatically figure out the important distinguishing features from one raw image to another, without explicitly being told what these features are. This data-driven approach is desirable as it enables portability across different downstream CAD tools and target devices. A CNN-based predictor has the ability to capture nonlinear relations between bitwidth reduction and ALM reduction. This approach is also able to identify *patterns of nodes* in the DFG, as opposed to considering each DFG node in isolation, as is done in the analytical approach above. In other words, in the CNN-based approach, a node and its surrounding context in the DFG are passed as input to the predictor. With the ability to infer non-linear relationships and to examine operations not in an isolated manner, we expect the CNN-based predictor to produce better area predictions than the analytical predictor. We describe the transformation from a DFG to a CNN input in Section 4.4.2.1. Section 4.4.2.2 describes the CNN network architecture explored in this thesis.

4.4.2.1 CNN Input Formulation

In order to transform a program's DFG into an input accepted by the CNN, we need to overcome the problem of mapping a variable-sized graph onto a fixed-sized matrix. We

achieve this by capturing a subset of the DFG that is representative of the entire DFG. To capture the area impact of reducing a specific variable's bitwidth, we take a *window* of the DFG surrounding that variable, and represent the DFG window (a portion of the entire DFG) as an image. Starting with the bitwidth-reduced variable, our window includes a portion of the DFG in the transitive fanout and transitive fanin of the bitwidth-reduced variable. In the DFG, variables correspond to nodes and the data dependences correspond to edges. We refer to the variable being considered for bitwidth reduction as the *target variable*. The middle row of the image corresponds to the target variable. Rows above the middle correspond to DFG nodes in the transitive fanin of the target variable; conversely, rows below the middle correspond to DFG nodes in the transitive fanout of the target variable. Figure 4.6 shows a simple example of the mapping, explained in more detail later. The natural question that arises is: how big should the window (and associated spatial image representation) be? If the target variable's bitwidth were reduced, those reductions may propagate to reductions of variables in its transitive fanin and fanout, particularly to nearby nodes of the DFG in the local vicinity of the target variable. It is desirable if the window size is large enough to "capture" such bitwidth propagations.

To determine the window size, we first note that in the LLVM IR, most instructions are either unary or binary, with the exception being `select`, `phi` and `call` instructions having 3 or more operands. Thus, most nodes in the DFG have fanin ≤ 3. Regarding typical node fanout, we profile a suite of benchmarks (CHStone [22]) to obtain the fanout distribution of IR nodes, shown in Figure 4.3. Observe that 95% of all IR variables have 3 or fewer fan-outs. Given these observations, we opt to support a connectivity of 3 (i.e. up to 3 fanouts and 3 fanins for each node) in our spatial DFG representation.

We also study the typical depths of DFGs, again using the CHStone benchmark suite. We profile the maximum number of DFG nodes that need to be traversed before reaching the target variable, as well as the maximum number of nodes that need to

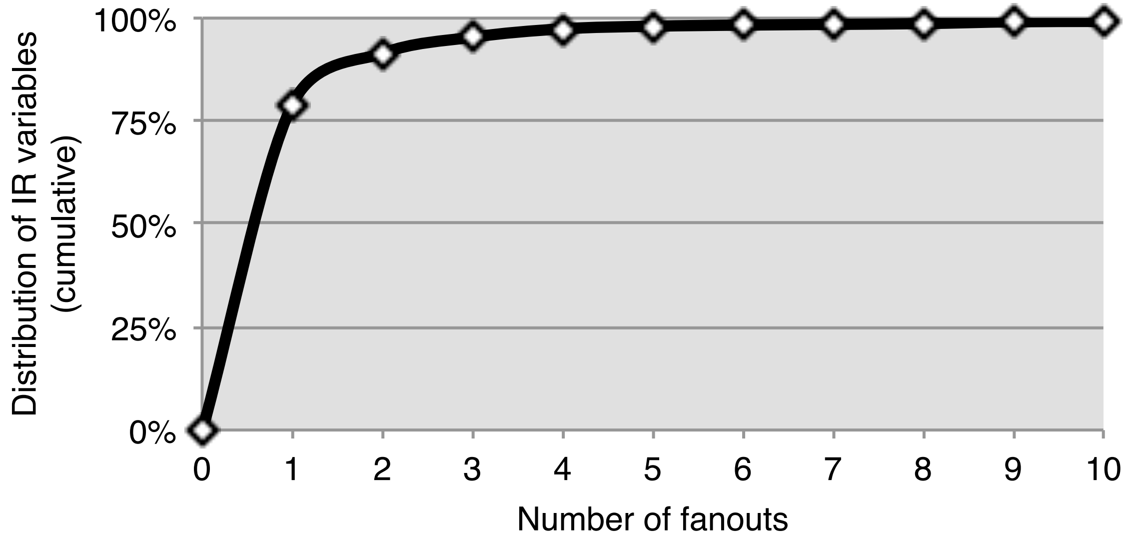


Figure 4.3: Average distribution of the number of fanouts for each IR variable.

be traversed from that variable to reach the final consumer of the value (i.e. a return instruction, a store instruction, or a function call such as printf). Figures 4.4 and 4.5 show the distribution of these numbers, respectively, for benchmarks in the CHStone suite. These two figures represent the upper bound on the number of DFG levels to examine before and after a target variable in order to capture all changes in the bitwidth for the CHStone benchmarks. If we fix the window fanout and fanin depths to *Depth* levels of nodes each in our spatial representation, the maximum size of the DFG window is $(2 \cdot \sum_{i=1}^{Depth} \text{connectivity}^i) + 1$ nodes, where the 1 is for the target variable itself. In our experiments, we set *connectivity* $= 3$ and *Depth* $= 5$, determined empirically to produce reasonable results. This configuration therefore captures a maximum of $(2 \cdot \sum_{i=1}^{5} 3^i) + 1 =$ 727 nodes in the DFG.

We now introduce our spatial DFG representation, which is best explained visually using an example. Figure 4.6 shows a DFG and its corresponding spatial representation (image) with *Depth* $= 2$. In this case, node d is the target variable (shown shaded). All pixels in the middle row of the image correspond to node d. d has a single fanout, f, that appears below d in the image, consuming three pixels. f has one fanout, g, which is

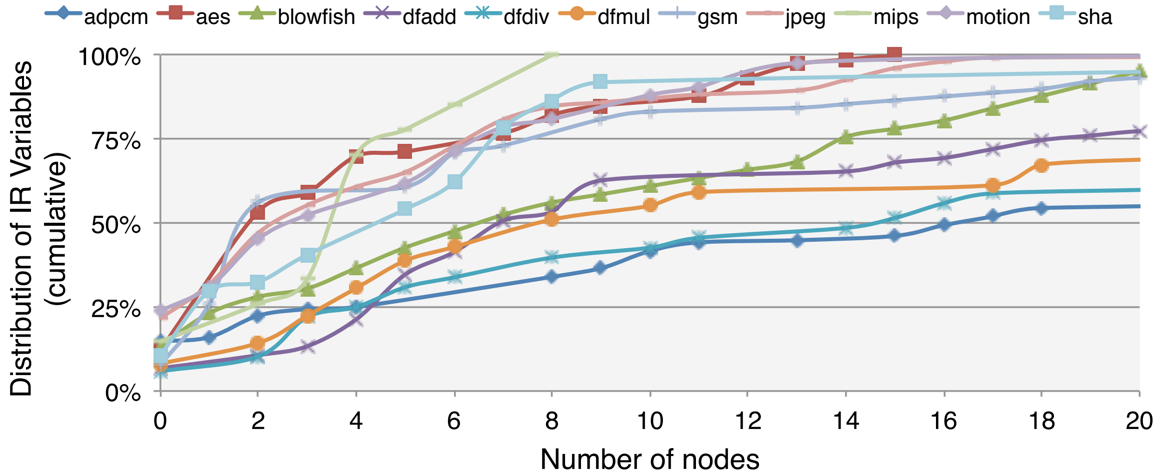


Figure 4.4: Cumulative distribution of the depth of nodes above a target variable.

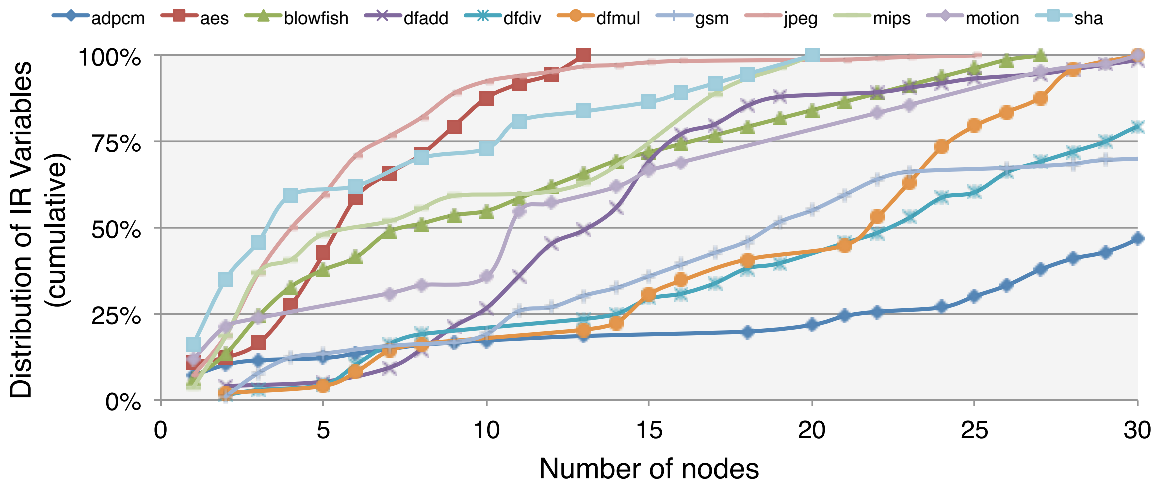
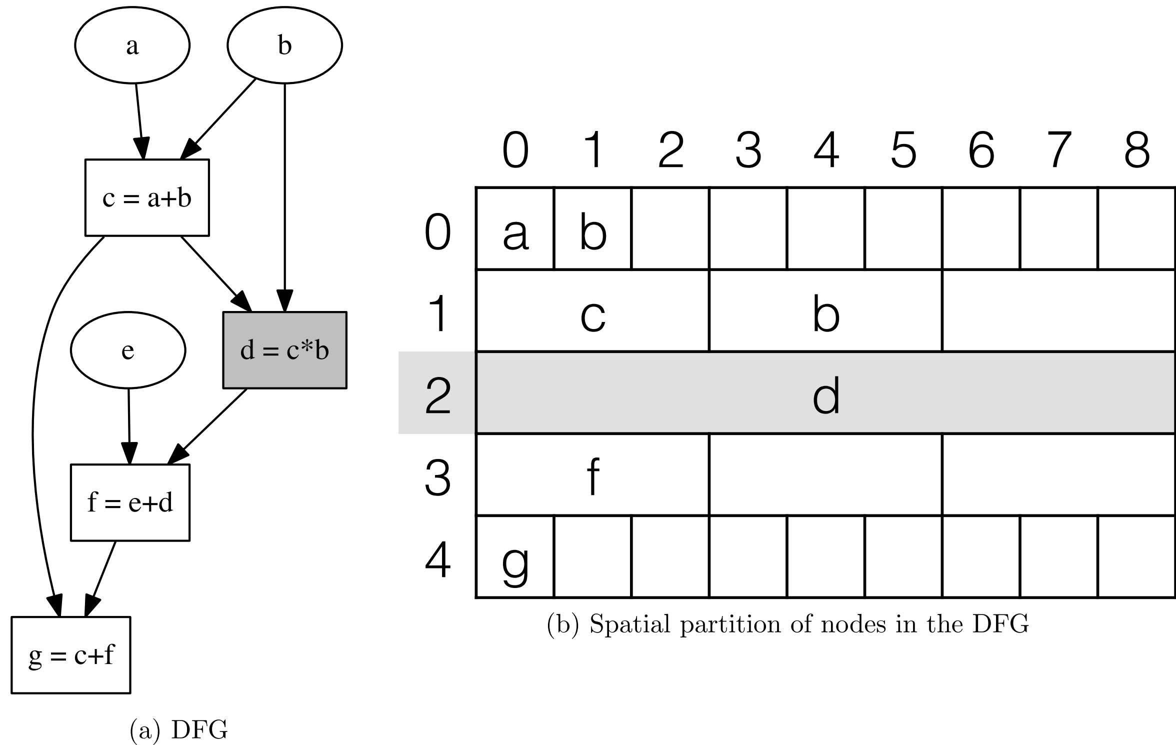


Figure 4.5: Cumulative distribution of the depth of nodes below a target variable.

represented by a pixel in the bottom row of the image. In the case of fanout greater than one, we look at the bitwidth amount reduced for each of the variables and place the top three variables in descending order from left to right. The top half of the image represents d's transitive fanin in a similar way. c and b are the fanins of d, and since a multiplication is commutative and if we assume that the bitwidth reduction on c is greater than or equal to b, we assign the leftmost 3 pixels to c and the next 3 pixels to b. Figure 4.7 shows the decision chart that determines the ordering of operands within the three partitions



(a) DFG

(b) Spatial partition of nodes in the DFG

Figure 4.6: An example of mapping a DFG to an image with $Depth = 2$.

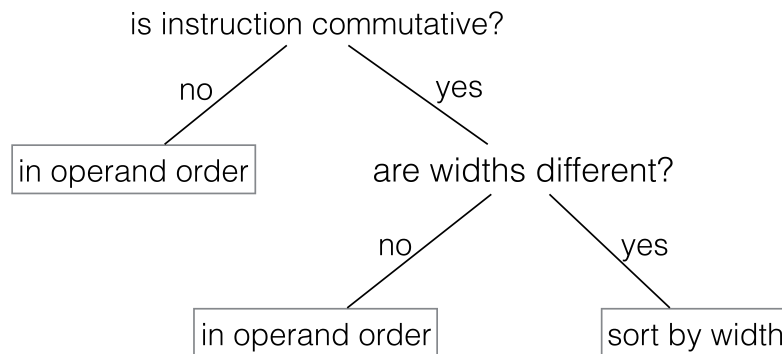


Figure 4.7: Order of operands in DFG image representation.

allocated for the operands. For example, if the operation is not commutative such as a divide, regardless of the bitwidth reduction on the two fanin variables, we assign the pixels left to right in the order that the operands appear. With the bottom half of the image where three partitions are allocated for the users of the variable, the users are sorted by their bitwidth reduction, then by their appearance in program order.

Once the nodes of the DFG are mapped to the pixels in the image representation, we

Channel	1	2	3	4
Config A	Original bitwidth	Δ bitwidth	**Operation type**	
Config B	Original bitwidth	Δ bitwidth	**Ordered op type**	
Config C	Original bitwidth	Δ bitwidth	Ordered op type	**Degree of FU sharing**
Config D	Original bitwidth	Δ bitwidth	Ordered op type	**Δ degree of FU sharing**

Table 4.2: Different configurations of information presented to the CNN-based predictor.

can experiment with predictors that make use of a variety of different pieces of information available during HLS. In a typical colour image, three channels are used, representing the R, G, B intensities of each pixel. Here, our "pixels" correspond to DFG nodes and we use up to four channels, containing information we deem to be useful to area prediction. If a pixel does not correspond to a variable in the DFG, it holds a value of 0. Table 4.2 summarizes the different combinations of information we experiment within this thesis.

Since we use 8 bits to represent each channel, we fully utilize the channel range by scaling all encoded values from 0 to 255. Such a scaling approach is to make the range of values roughly consistent across channels, which is a technique typically used in machine learning to help speed up the training process. In channel 1, we encode the original width of a variable in the DFG. The bitwidth of a variable ranges from 1 to 64; we scale this to the 8-bit channel via $(\text{bitwidth} \times 4 - 1)$. In channel 2, we encode the change in bitwidth of a variable after the initial hypothetical bitwidth reduction of the target variable and the invocation of the BWM framework to propagate the bit reduction. Again, since we do not expect the width of any variable to be reduced by more than 64 and there are 8 bits available in this channel, the actual data encoded in this channel is $(\Delta\text{bitwidth} \times 4 - 1)$. We designate the first two channels to represent the variable's bitwidth and the change in bitwidth for all four CNN configurations.

In configuration A, the third channel contains information regarding the operation type that produces that variable. The numerical value used for this configuration is the native enumerated-type value that exists in LLVM 3.5, which spans from 0 to 64 for instruction types and an extra four values to differentiate constant integers, constant expressions, function arguments, and global variables from instructions. For configurations

B to D, we refine the numerical values representing each operation type by profiling each operation type's typical area consumption. An operation that consumes a larger area is given a higher numerical value and vice versa; and operations that consume similar area are given values numerically close to one another.

In configuration C, we introduce information gathered from the HLS binding step in the fourth channel. For each variable, we look at the functional unit that produces it and determine the number of other variables that also use the same functional unit after synthesis. This information is introduced in order to provide more insight into how the different functional units in the circuit are shared, which is an important factor in determining the post-synthesis area reduction. We refine this information further in configuration D, which looks at the difference in the number of sharers for each functional unit (before and after bitwidth reduction) as opposed to the absolute number of sharers. This configuration involves performing the binding step of HLS twice, once with the original bitwidth and once with the reduced bitwidth. The different CNN configurations produce different prediction accuracies, which will be shown in Section 4.6.

4.4.2.2 CNN Model Design

Given that the inputs to the CNN are effectively pseudo-images, there are no well-known CNN models that can easily be employed for our purposes. We therefore explored different network architectures to obtain an acceptable model for the task of predicting post-synthesis area reduction based on a program's DFG. Taking inspiration from the success of the VGG16 CNN in image recognition [37], which showed the benefits of using small 3×3 convolutional filters and increasing the number of convolutional layers in the network, the models we explore follow the same principles: 1) we use small 3×3 filters in all the convolutional layers; and 2) we consider models of zero or more convolutional layers followed by one or more fully connected layers. In all cases, the network has one fully connected layer with an output feature map of 1×1024 at the end, used for

generating the final real-valued prediction. The prediction accuracy results for different architectures are presented in Section 4.6.1.

In addition to varying the number of convolutional layers, we also explore the effects of varying some of the hyperparameters in the CNN architecture. Table 4.3 summarizes the hyperparameters we explore and the values used in each experiment. We discuss the effects of varying these hyperparameters in Section 4.6.3.

Hyperparameter	Options
Number of filters in convolutional layer	32, 64, 128
Input pixels normalized by mean	yes, no
Stride of filters in convolutional layer	always 1, alternating between 1 and 3

Table 4.3: CNN model hyperparameters in our experiments.

4.5 Methodology

4.5.1 Experimental Setup

To evaluate the Sensei framework, we construct the evaluation dataset from the CHStone benchmark suite [22], which is a widely used benchmark suite for evaluating HLS frameworks. The CHStone benchmark suite is comprised of applications from a variety of domains, including media processing, security and microprocessor simulation. Ten out of twelve of the CHStone benchmarks are included in this study; `adpcm` and `jpeg` are omitted since they require more DSP blocks than available on the Intel Cyclone V device we target[2]. To gather datapoints for this study, we enumerate all IR variables that correspond to a `C` program variable for every benchmark in the CHStone benchmark suite. For each such IR variable, we generate variants of the benchmark by reducing the width of the IR variable to each of these values: 1, 2, 4, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, and

[2]Note also that in this work, we do not consider resource-constrained scenarios wherein the underlying FPGA lacks sufficient resources of a given type (e.g. DSP blocks), and other resource types (e.g. ALMs) are used in place of the original intended resource type. That direction is left to future work.

64 bits, up to the variable's original width. Variables corresponding to memory addresses are excluded since they do not affect the circuit's datapath width.

With the above method, we generate ∼7000 bitwidth-reduced circuits to be used as CNN training and test cases (i.e. separate benchmark circuits wherein one IR variable's bitwidth is reduced by a given amount), each of which are generated by LegUp HLS and pushed through Quartus II to obtain the detailed resource usage report. We use the GPC supercomputer at the SciNet HPC Consortium [31] to synthesize each of the circuits. Note that some of the datapoints obtained this way render the original circuit incorrect with respect to the original input vectors. However, the circuit would be correct if the input vectors and the functionality were to reflect the corresponding bitwidth reduction. The onus is on the designer to think about whether or not a reduction of a given width is legal for their design. We focus on estimating ALM area consumption; future work will consider the area of other block types, e.g. DSP blocks.

To evaluate the generality of our CNN setup, we partition the entire data set into 10% test set and 90% training set. For each benchmark, we randomly assign 10% of the data to the test set, and the rest are assigned to the training set. With the ten benchmarks in CHStone, this yields 6400 data points in the training set and 640 data points in the test set. We use Caffe [23] to train the network for 1000 epochs for each of the experiments. To eliminate any bias in the selection of test and training set, we perform three random partitions of test and training set and confirm that the results to be generally consistent across the three partitionings.

4.5.2 Evaluation Criteria

To quantify the efficacy of the Sensei predictors, we evaluate two aspects of the predictions generated: 1) the accuracy of the predicted ALM reduction; and 2) the ranking of variables in terms of their influence to post-synthesis area reduction. A good predictor that produces a high quality of result (QoR) will not only produce the correct ranking of

the variables, but also produce a prediction close to the actual number of ALMs reduced. Having an accurate prediction on the number of ALMs reduced allows the HLS user to easily gauge the extend at which their circuit can be reduce.

To evaluate the accuracy, we use a prediction-error metric we call the *percentage of acceptable predictions* (PAP). We compute the error per datapoint as the absolute difference between the predicted and the actual ALM reductions, normalized to the number of ALMs in the overall circuit. We define a threshold ϵ (percentage of the overall circuit area) that represents the maximum error for an acceptable prediction. We then compute PAP as the percentage of all datapoints having error lower than ϵ. In this metric, we define ϵ as a percentage of the overall circuit area in order to account for the significance of a variable's area reduction. A small ϵ value means either the absolute value of the error in prediction is small or the error in prediction is small relative to the size of the circuit. The smaller the overall circuit, the more crucial it is to get an accurate prediction. With the CHStone benchmarks used in this study, the total number of ALMs ranges from 1200 to 9000. The PAP numbers are used to evaluate experiments mentioned in Sections 4.4.2.1 and 4.4.2.2, as well as the results presented in Sections 4.6.1, 4.6.2 and 4.6.3. All prediction accuracy assessments are done using the test data set.

To quantify the predictors' ability to rank variables, we look at the top 5 variables predicted by each predictor and compute the average ALM reduction per bit achieved. This metric does not evaluate what happens when multiple variables are reduced at the same time since our predictors do not have that information. Instead, we look at each prediction in isolation and provide the average to get a sense of what the typical ALM reduction is. We measure this by identifying the top 5 variables predicted by each of the predictors, measure the actual ALM reduction for each prediction, and take the average across the 5 variables. We also show the lowest and highest ALM reductions among the predicted top 5 variables for each predictor. This criteria is used in Section 4.6.4 when we compare the analytical predictor to the CNN-based predictor.

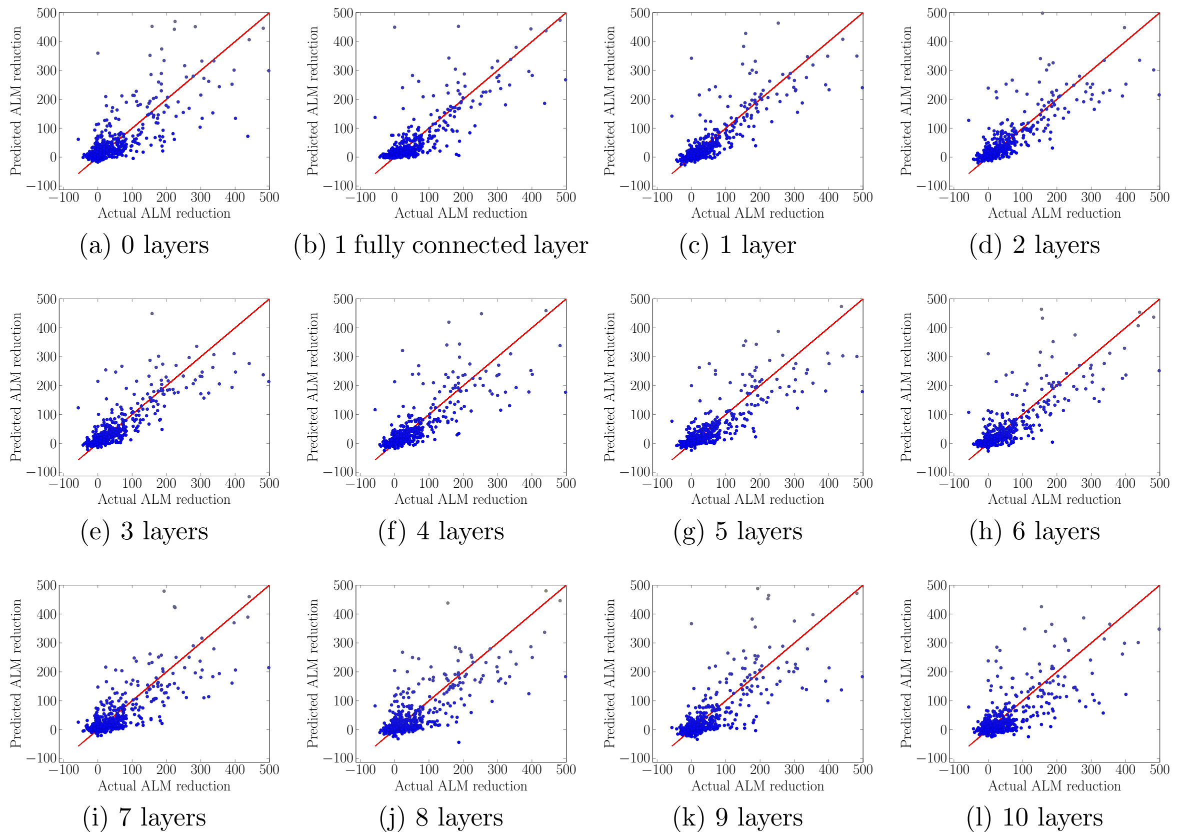
4.6 Results

In this section, we present the results for the different configurations of predictors described in Section 4.4.

4.6.1 CNN Model Selection

For each input configuration and each set of network hyperparameters, we study the correlation between the number of convolutional layers in the network, and the prediction accuracy results. Figures 4.8 and 4.9 show the prediction accuracy on the test set as we increase the number of convolutional layers in a network with one fully connected layer. Figure 4.9 shows the prediction on the entire test set and Figure 4.8 shows the predictions that falls between -100 and 500 ALM reduction. The y-axis shows the predicted number of ALMs reduced, and the x-axis shows the actual number of ALMs reduced after Quartus II synthesis, place and route. The closer the points are to the red line, which corresponds to the $y = x$ line, the higher the accuracy of the prediction. The ideal case is to have all points line up at the red line.

Figure 4.8a is the network where the input image is directly feeding into the default one fully connected layer of the network. By visual inspection of Figures 4.8b (adding 1 more fully connected layer) and 4.8c (adding 1 convolutional layer), both graphs display higher QoR than Figure 4.8a, which indicates that the network benefits from having more than 8 million weights. Furthermore, Figure 4.8c displays a higher QoR than Figure 4.8b, which shows that the addition of a convolutional layer improves the QoR over the addition of a fully connected layer. This confirms our initial hypothesis that the introduction of spatial awareness of values in the DFG improves prediction accuracy. Beyond one convolutional layer, the difference is harder to discern by visual inspection. We plot the PAP of each CNN model architecture when $\epsilon = 0.3$ alongside the average SOS error of the test set in Figure 4.10. As discussed in Section 4.3.1, the SOS error



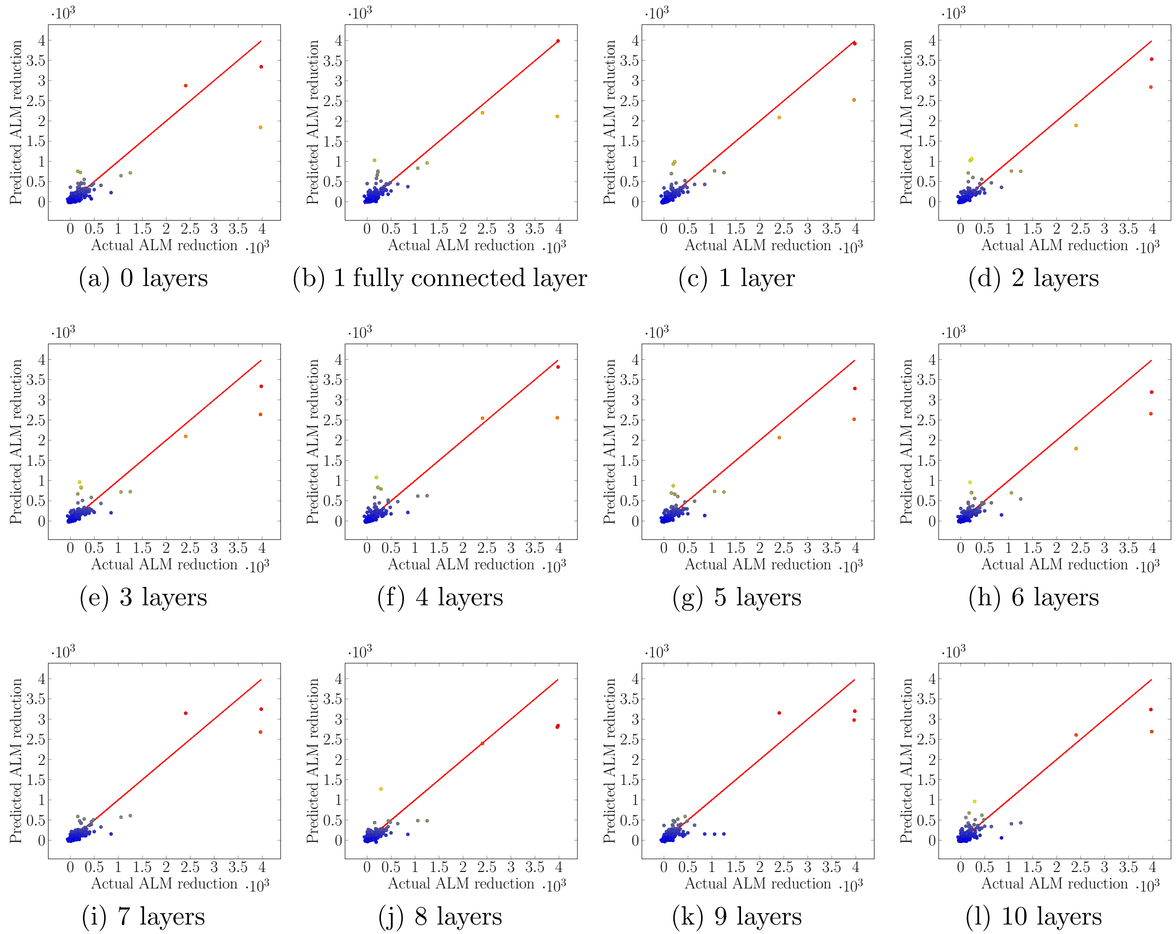
(a) 0 layers (b) 1 fully connected layer (c) 1 layer (d) 2 layers

(e) 3 layers (f) 4 layers (g) 5 layers (h) 6 layers

(i) 7 layers (j) 8 layers (k) 9 layers (l) 10 layers

Figure 4.8: CNN architecture exploration by varying the number of convolutional layers in the network. Each graph plots the predicted ALM reductions vs. the actual ALM reductions of the test set, within the -100 to 500 range.

is the cost function used in the neural network during training, and the purpose of the training phase is to minimize the cost function. $\epsilon = 0.3\%$ is chosen since it is the first instance where any network architecture is able to predict over 50% of the test set within ϵ. The full set of data where ϵ is swept from 0.1% to 1%, as well as the final SOS error value reported by the Caffe framework is presented in Table 4.4. We expect to see an inversely proportional relationship between the SOS error and the prediction accuracy (i.e. PAP). However, this is not always observed, such as in the case of moving from 1 to 2 convolutional layers, and 6 to 7 convolutional layers. For both of these cases, the network achieves a lower PAP even though the SOS error is reduced. Looking at Figures 4.8c, 4.8d, 4.9h, and 4.9i, we observe a trend where some points that are very far from the red line are moved closer to the line, yet some points that are close to the red line are



(a) 0 layers (b) 1 fully connected layer (c) 1 layer (d) 2 layers

(e) 3 layers (f) 4 layers (g) 5 layers (h) 6 layers

(i) 7 layers (j) 8 layers (k) 9 layers (l) 10 layers

Figure 4.9: CNN architecture exploration by varying the number of convolutional layers in the network. Each graph plots the predicted ALM reductions vs. the actual ALM reductions of the entire test set.

moved slightly further away. In order to reduce the error on the badly predicted points, the network trades off some of the accuracy of good predictions. Under a strict accuracy constraint, this trade off is not ideal.

By examining the graphs in Figure 4.8 and Figure 4.10, we can see that the network with 1 convolutional layer produces the highest accuracy, and that the best accuracy lies somewhere between 1 to 5 convolutional layers. Beyond 5 convolutional layers, the PAP drops, and we see an increase in the average SOS error reported by the Caffe framework. The saturation point at 5 convolutional layers for this study is likely attributed to the relatively small amount of training data that we have in this study. We expect that as

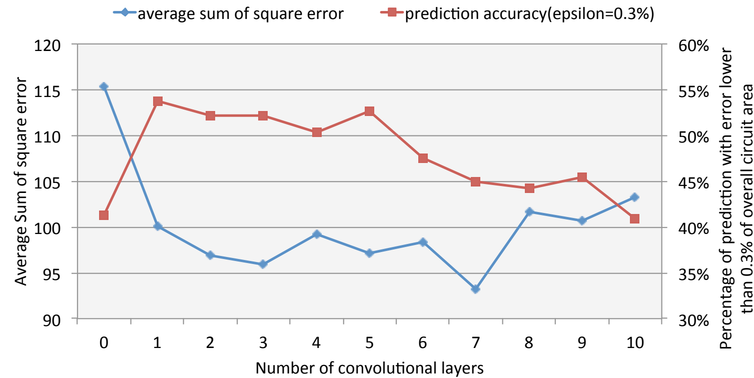


Figure 4.10: Average sum of square error reported by the Caffe framework and the PAP when $\epsilon = 0.3\%$ for networks with 0 to 10 convolutional layers.

Number of conv. layers	0	1	2	3	4	5	6	7	8	9	10
Average SOS error	115.35	100.06	96.86	95.87	99.23	97.10	98.41	93.19	101.72	100.65	103.23
ϵ											
0.1%	15%	22%	21%	21%	23%	25%	19%	18%	19%	21%	21%
0.2%	29%	42%	38%	38%	39%	40%	37%	34%	36%	33%	33%
0.3%	41%	54%	52%	52%	50%	53%	48%	45%	44%	45%	41%
0.4%	50%	64%	62%	61%	59%	62%	56%	55%	53%	56%	49%
0.5%	56%	70%	68%	68%	67%	69%	65%	60%	60%	63%	57%
0.6%	61%	77%	76%	74%	72%	74%	70%	66%	66%	69%	63%
0.7%	68%	79%	79%	77%	77%	79%	74%	71%	70%	74%	68%
0.8%	72%	83%	82%	82%	80%	81%	78%	74%	75%	76%	72%
0.9%	75%	86%	84%	83%	82%	83%	81%	77%	77%	79%	75%
1.0%	78%	87%	85%	85%	84%	84%	83%	79%	80%	81%	77%

Table 4.4: Average SOS error and PAP at different values of ϵ with different number of convolutional layers.

the number of datapoints used in training increases (which would be commonplace in an industrial context where many more benchmark designs are available), the network would require more convolutional layers before overfitting. Even when the saturation point shifts to a higher number of convolutional layers, we expect the observation that the PAP degrades as the number of convolutional layers becomes too large will still hold. As CNN network size grows, the potential for overfitting the training data also grows, thereby damaging results for the test dataset.

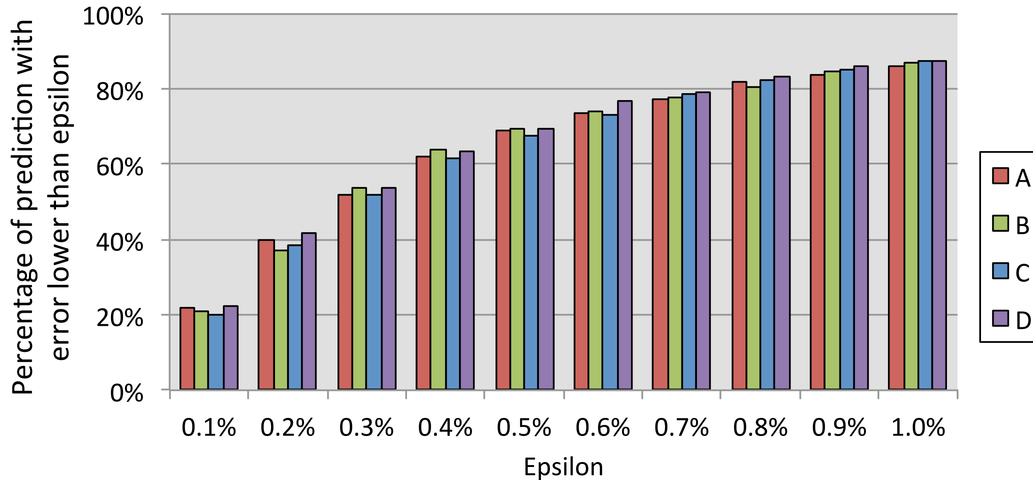


Figure 4.11: PAP results when different input configuration is fed into a CNN with 1 convolutional layer.

ϵ	A	B	C	D
0.1%	22.03%	20.94%	20.16%	**22.34%**
0.2%	40.00%	37.19%	38.28%	**41.88%**
0.3%	51.88%	**53.91%**	52.03%	53.75%
0.4%	61.88%	**64.06%**	61.56%	63.59%
0.5%	69.06%	69.22%	67.66%	**69.53%**
0.6%	73.59%	74.06%	73.28%	**76.56%**
0.7%	77.34%	77.66%	78.44%	**79.22%**
0.8%	81.72%	80.63%	82.34%	**83.13%**
0.9%	83.59%	84.69%	85.00%	**85.94%**
1.0%	85.94%	86.72%	**87.19%**	**87.19%**

Table 4.5: PAP with different input information configuration (best in bold).

4.6.2 Impact of Input Dataset Configuration

In Section 4.4.2.1 (Table 4.2), we described four different ways of representing the input dataset, each of which passes different information to the CNN-based predictor. We train CNN with a single convolutional layer on all four such input configurations. Figure 4.11 and Table 4.5 show the PAP of each of the four input configurations as we sweep the error threshold ϵ from 0.1% to 1%. Moving from input configuration A to B, the introduction of a proportionally ordered encoding of operation type provides an average improvement of 0.2% in accuracy. Intuitively, we expect to get better results when we provide a more

meaningful set of numbers as input. However, this comparison shows that contrary to intuition, the network is not as sensitive to the ordering of operations even though it does improve the results slightly. When comparing input configurations B and C, which saw the introduction of binding information, we see that there is a degradation in PAP when an extra channel of input is introduced. However, when we refine the values presented in this channel, as in configuration D, we have an average of 1.3% increase in PAP when compared to configuration B. From these two observations, we see the importance of including meaningful information – blindly presenting extra information to the network can be harmful to the overall quality of the predictions.

Of the four different configurations, A and B rely solely on information obtainable before performing any target-specific HLS steps. Configuration C can be obtained after performing the allocation, scheduling, and binding steps of HLS. Configuration D is obtained after performing the allocation, scheduling, and binding steps an additional time, with the bitwidth-reduced DFG. Depending on the prediction accuracy required, the user can trade off runtime against the quality of prediction.

4.6.3 Impact of CNN Hyperparameters

CNNs have a variety of network hyperparameters that influence the prediction results. In this work, we experiment with three such parameters: 1) the number of filters in the convolutional layers of the network, 2) the stride of the filters in the convolutional layers of the network, 3) whether or not input pixel values are normalized by the mean value of the pixels in the channel.

4.6.3.1 Number of Filters in Convolutional Layers

The number of filters in each convolutional layer influences the number of features each layer can pick up. Theoretically, the deeper each layer is, the more fine-grained features the layer is able to identify, since there are more weight filters stored in the network.

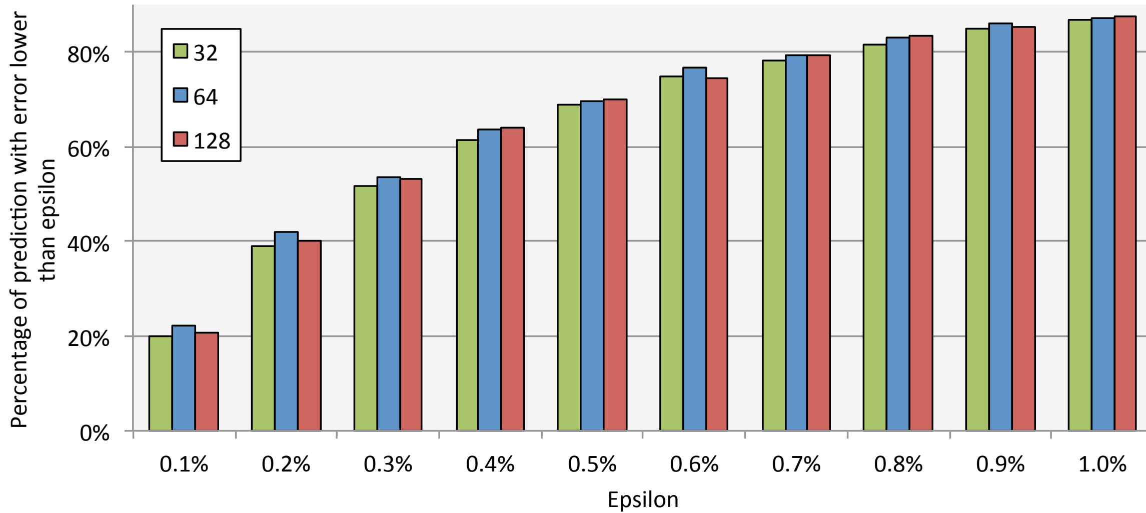
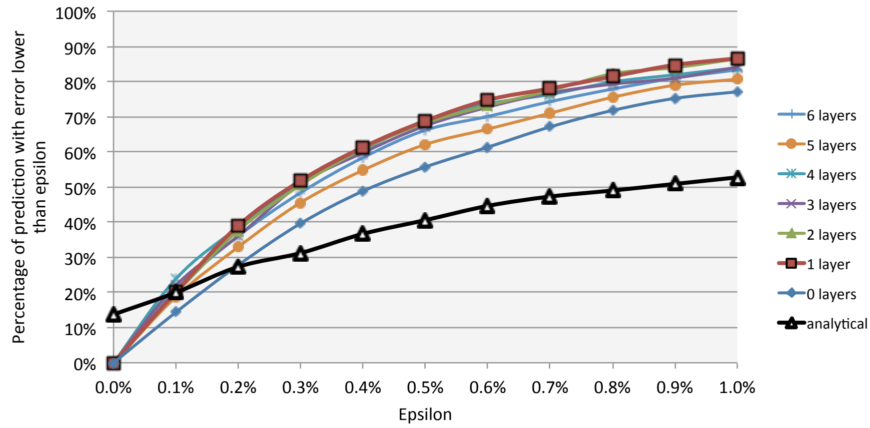


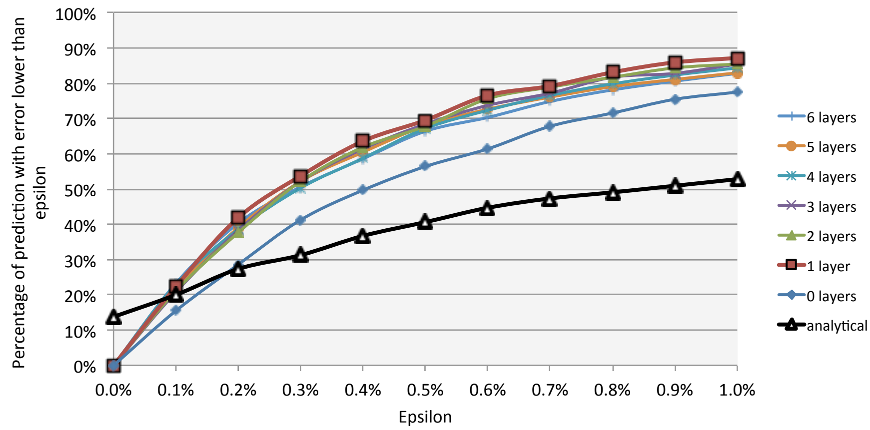
Figure 4.12: PAP comparison of different number of filters in the convolutional layers.

However, there exists an upper limit on the number of effective weight filters in the network; above a certain number of filters, some become redundant or pick up features that are specific to one training sample and not generalizable to other inputs. Figure 4.12 shows the PAP on the test set with one convolutional layer when the number of filters in the convolutional layer is 32, 64, and 128. A layer with 64 filters consistently outperforms a layer with 32 filters when ϵ is varied from 0.1% to 1%. When the number of filters is increased to 128, we observe a minimal increase in accuracy in half of the evaluation points and a degraded accuracy in the other half (when the ϵ constraint is more strict).

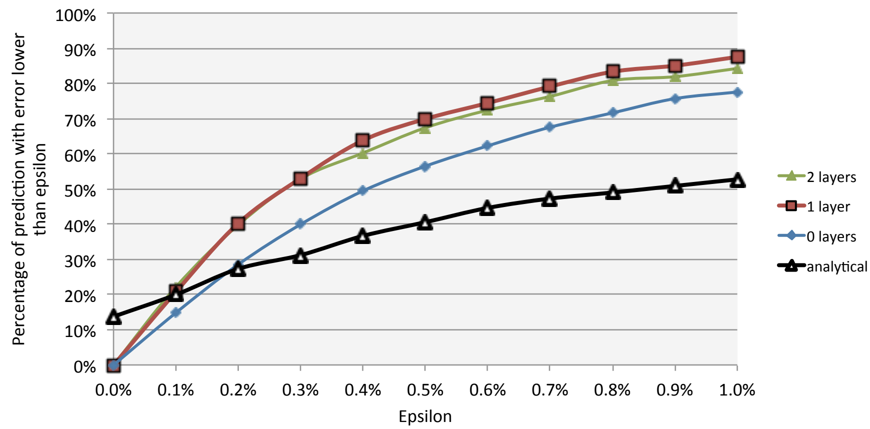
Figure 4.13 show the prediction accuracy results of networks with 0 to 6 convolutional layers when the number of filters for each convolutional layer is set to 32, 64 and 128. For a layer with 128 filters, the GPU used in this experiment does not have sufficient memory to handle more than 2 convolutional layers, so we only show the results for up to 2 convolutional layers. In all three configurations, the most accurate network is the one with one convolutional layer. From this result, we see that the location at which we add more weight filters in the neural network can impact its prediction accuracy. For our application of the CNN, adding more filters to a single convolutional layer provides more benefit than adding more convolutional layers. This reveals that instead of capturing



(a) 32 filters



(b) 64 filters



(c) 128 filters

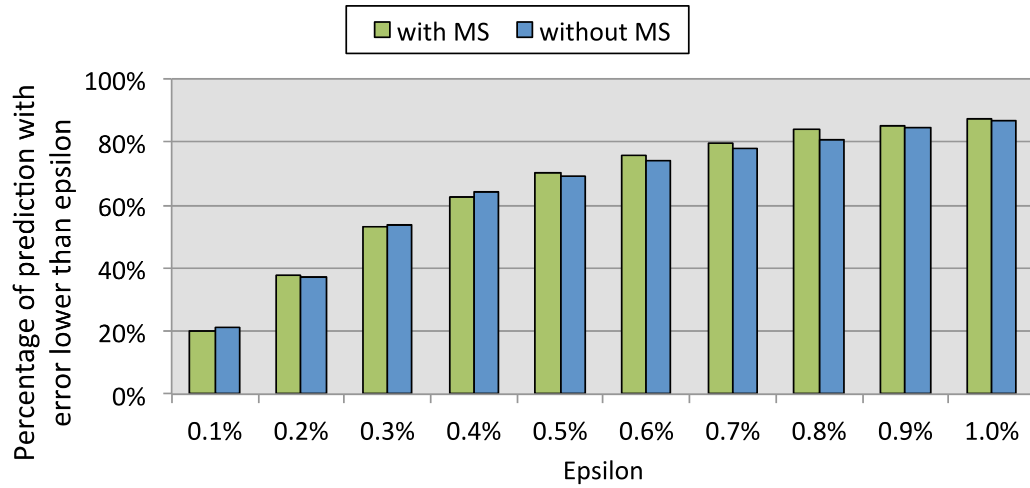
Figure 4.13: PAP of CNNs with 0 to 6 convolutional layers, each with a different number of filters in the convolutional layers.

a smaller set of fine-grained features first and then composing larger features out of the fine-grained features, the network performs much better when it is able to capture many fine-grained features right away. In our context, coarse-grain features look at how bitwidths propagate across large blocks of instructions in the DFG, whereas fine-grain features look at neighbouring instructions. Our results show that neighbouring instructions are more useful for predicting area savings than instructions that are further away.

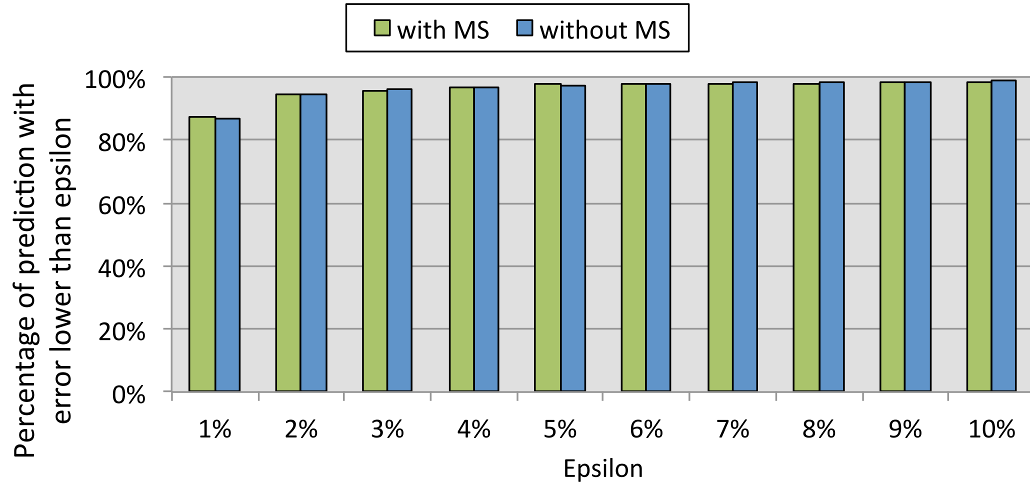
4.6.3.2 Impact of Mean Subtraction

When CNNs are applied for image recognition, two transformations to the input data are typically applied: feature scaling and mean subtraction (i.e., subtracting each value by the mean of all input values). These transformations are executed in order to facilitate easier weight sharing and faster convergence. Since the values in our input are already scaled to values from 0 to 255 by design, we only evaluate the effect of mean subtraction. Figure 4.14 shows the comparison between applying and omitting mean subtraction for a network with one convolutional layer. The two prediction results are fairly comparable to each other, with the greatest difference (3%) occuring at $epsilon = 0.8\%$, and an average difference of 0.6%.

Figure 4.15 shows the accuracy of prediction with and without mean subtraction when the number of convolutional layers is varied from 1 to 7. Most of the networks produce comparable prediction accuracy with or without mean subtraction. All but the networks with 2 and 7 convolutional layers observe a maximum difference of less than 3% when mean subtraction is toggled; a maximum difference of 5% is observed in the networks with 2 and 7 convolutional layers. While in very specific cases, mean subtraction can yield slighly better prediction results, there is not a clear trend. We conclude that with our particular application of CNNs, mean subtraction is not necessary.



(a) Epsilon ranging from 0.1% to 1%

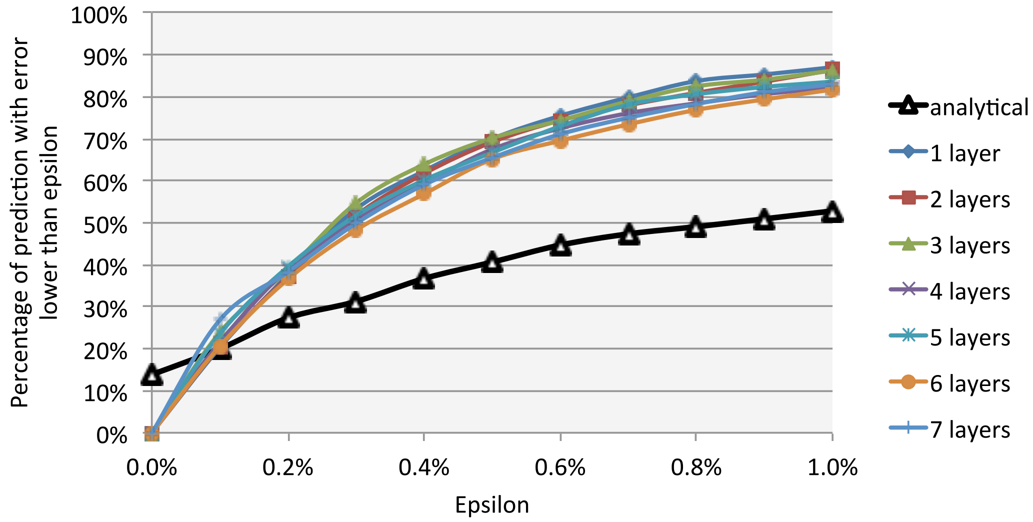


(b) Epsilon ranging from 1% to 10%

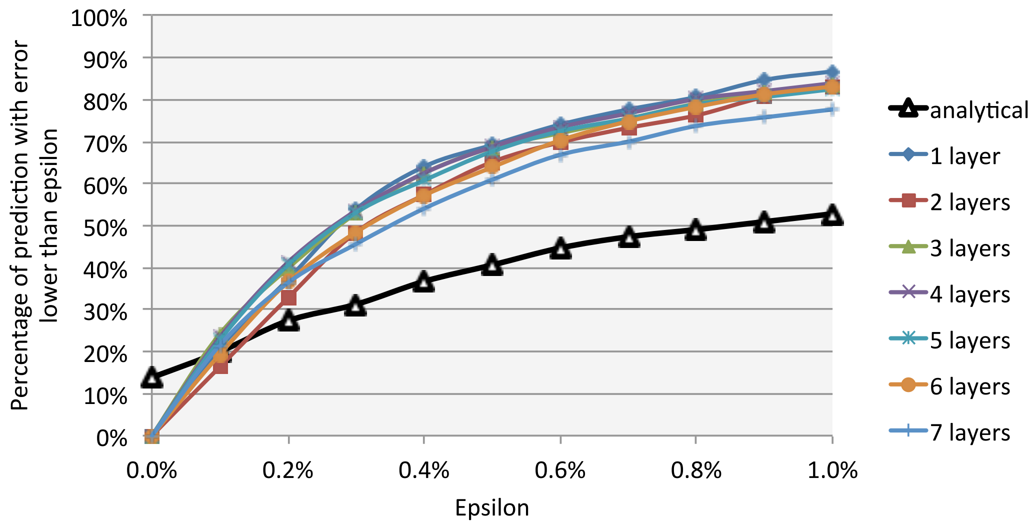
Figure 4.14: PAP of CNN with 1 convolutional layer with and without mean subtraction (MS).

4.6.3.3 Filter Stride

Figure 4.16 shows the PAP when the filter stride is set to 1 for all layers and when it is set to alternating strides of 1 and 3 (e.g. for a network of 3 convolutional layers, layer 1 and 3 use a stride of 1 and layer 2 uses a stride of 3). Since the filters in the convolutional layers are 3×3, a stride of 3 represents the largest stride possible without ignoring any pixels in the input to a convolutional layer,. Looking at the networks



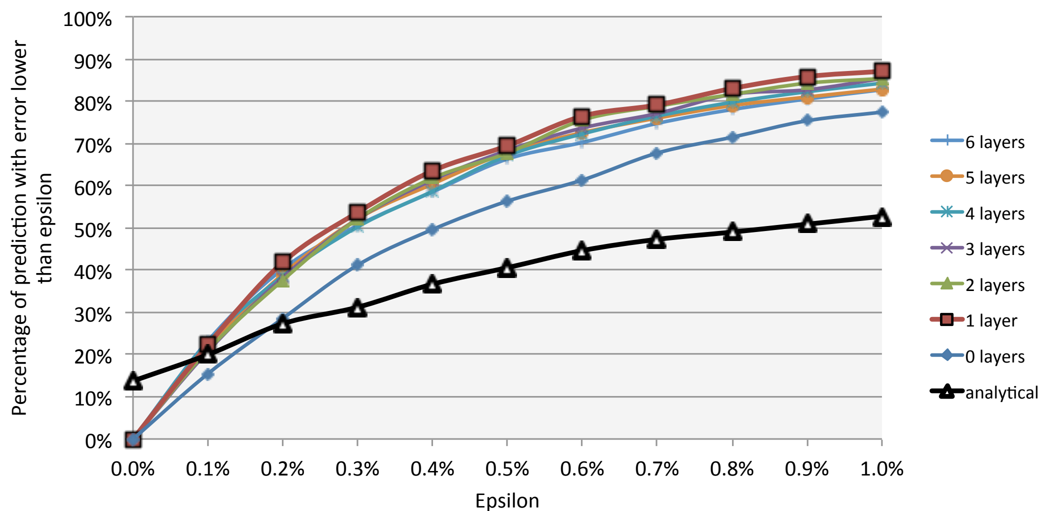
(a) With mean subtraction



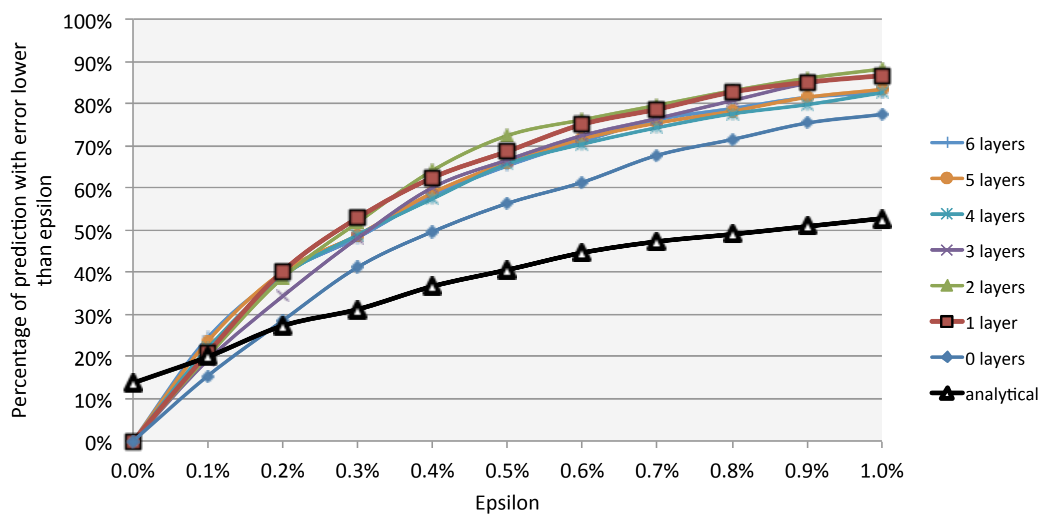
(b) Without mean subtraction

Figure 4.15: PAP of CNNs with 0 to 7 convolutional layers, with and without mean subtraction.

using alternating strides (Figure 4.16b), we see that the one convolutional layer network still has the highest PAP when $\epsilon \leq 0.3\%$, but the two convolutional layer network overtakes it when $\epsilon \geq 0.4\%$. When comparing the two convolutional layer network using alternating stride with the one convolutional network using constant stride, there is minimal difference in the PAP. Figure 4.17 shows the difference in PAP at different values of ϵ when compared with a constant stide of 1. We see that increasing the stride



(a) Constant stride of 1



(b) Alternating stride between 1 and 3

Figure 4.16: Accuracy of CNNs with 1 to 6 convolutional layers with filter stride configured to (a) all 1, and (b) alternating 1 and 3.

of convolutional layers generally results in a slight degradation to the PAP. If a more relaxed accuracy requirement is acceptable, it may be worth using a stride greater than one, since it reduces the dimensionality of the output feature map and increases the spatial coverage of the original image.

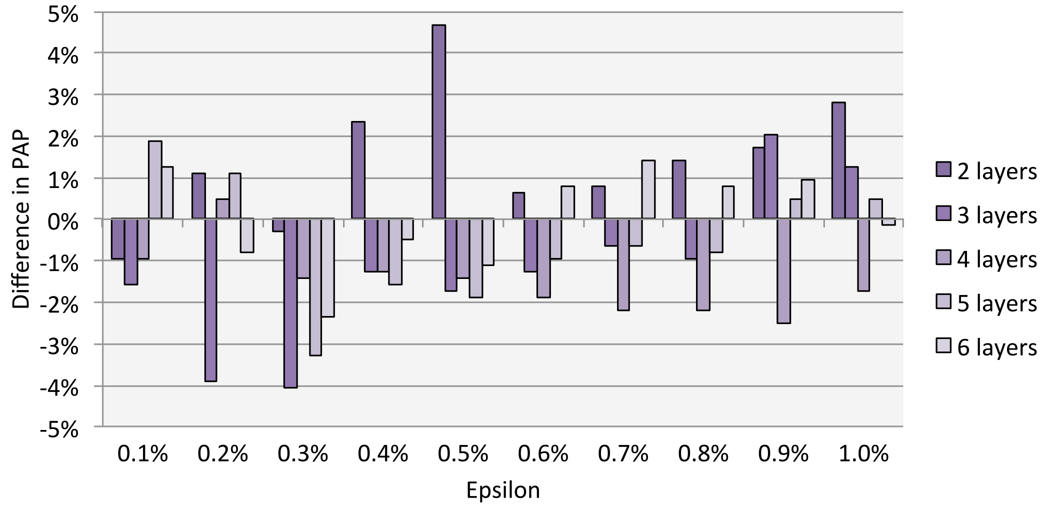
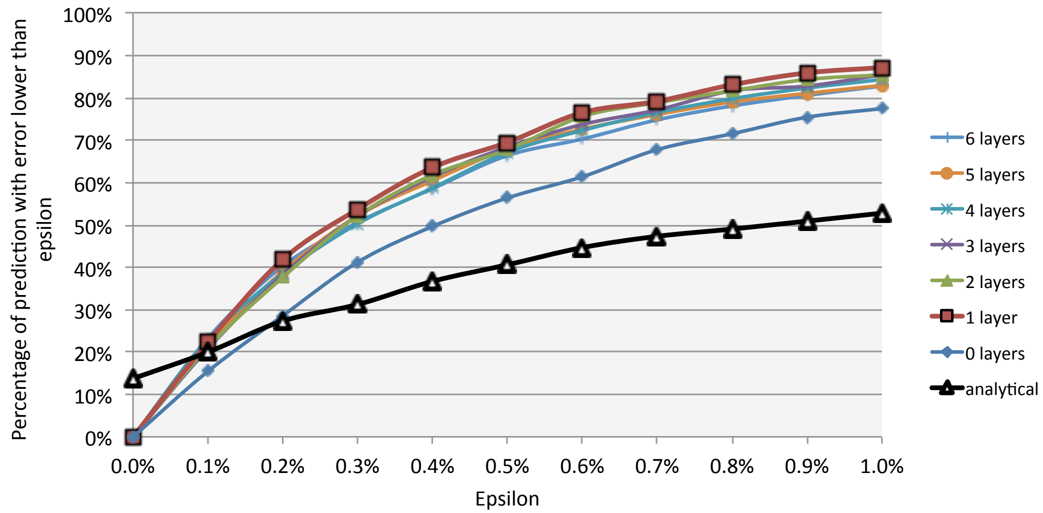


Figure 4.17: Difference in PAP of CNNs with alternating strides of 1 & 3, compared with constant stride of 1.

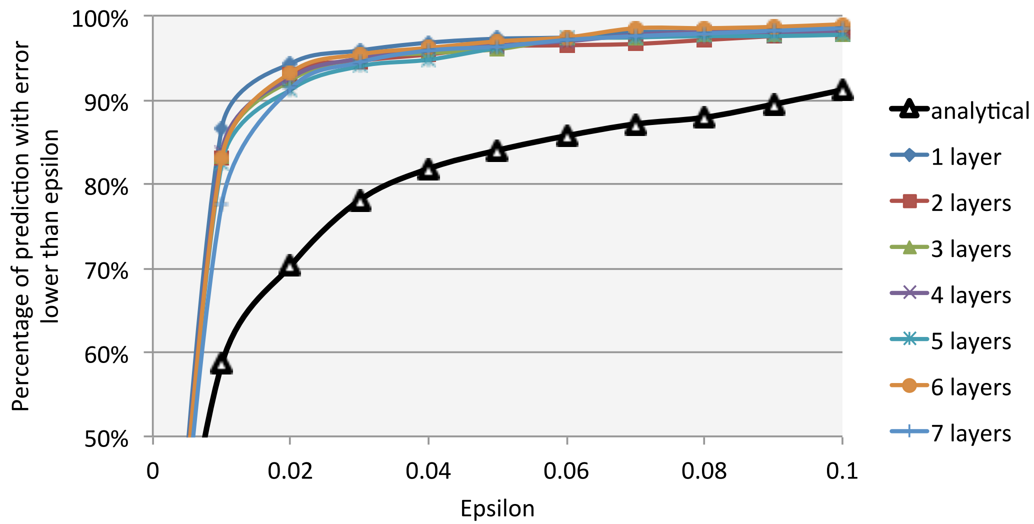
4.6.4 Comparison of Analytical to CNN-based Predictor

4.6.4.1 Accuracy

From the results in previous sections, the best CNN-based predictor for Sensei consists of one convolutional layer with input configuration D, 64 filters in the convolutional layer, and no mean subtraction on the input pixels. We compare this predictor to the analytical predictor, and we show the PAP as we sweep ϵ from 0.1% to 1% and 1% to 10% of the original circuit area in Figure 4.18. From the graph, we see that in order to have over half of the predictions to be acceptable, we have to relax ϵ to 0.6% of the overall circuit area for the analytical predictor, compared to 0.3% for the CNN-based predictor. At $\epsilon = 1\%$, only 53% of the predictions generated by the analytical predictor are acceptable, compared to 87% from our best CNN-based predictor (1 convolutional layer). With error tolerance of up to 10% of the original circuit area, the prediction accuracies of the analytical and our best CNN-based predictor increase to 91% and 99% respectively. Figure 4.19 shows the predicted vs. actual ALM reduction on the same test set for the analytical predictor. Unlike our CNN-based predictor, many of the points lie far above and below the ideal red line. The majority of these mispredictions are due



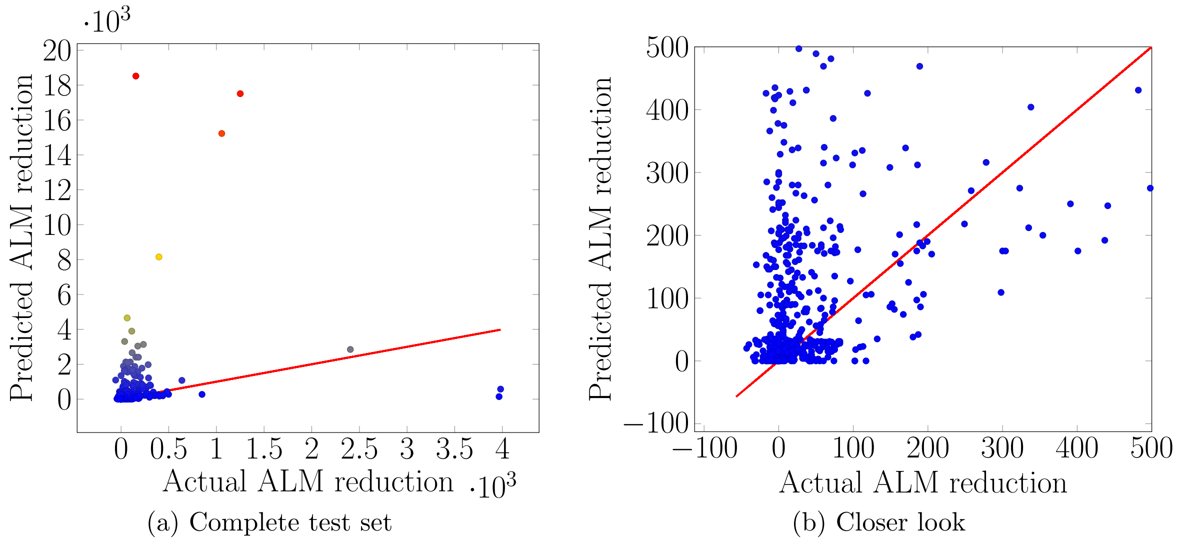
(a) Epsilon ranging from 0.1% to 1%



(b) Epsilon ranging from 1% to 10%

Figure 4.18: PAP of CNN-based predictor and the analytical predictor as we vary the error threshold ϵ.

to over-estimating the ALM reduction, since the formulation of the analytical predictor does not take into account the possibility that a DFG node and one of its immediate neighbours may be covered by the same LUTs in the FPGA mapping.



(a) Complete test set (b) Closer look

Figure 4.19: Analytical model prediction on the test set.

4.6.4.2 Ranking

To demonstrate the efficacy of Sensei in ranking the top variables, Figure 4.20 shows the potential for area savings using the top results from the predictions of our test set. For the CNN-based predictor, we use the architecture with one convolutional layer since it yields the highest PAP at $\epsilon = 1\%$. Our metric for the area savings potential is the number of ALMs reduced per bit reduced. We compute this by taking the top 5 variables selected by the CNN-based predictor and measure their actual area savings (averaged across the 5 variables). We also show the lowest and highest savings among the predicted top 5 variables. We compare this against the top 5 variables selected by the analytical predictor. For 6 of the 10 benchmarks, following the *advice* of the CNN-based predictor yields better area savings (up to 7.4× for `dfdiv`). For the remaining 4 circuits, the difference between the two predictors is minor. On average, the CNN-based predictor reports variables that yield higher area savings than the analytical predictor.

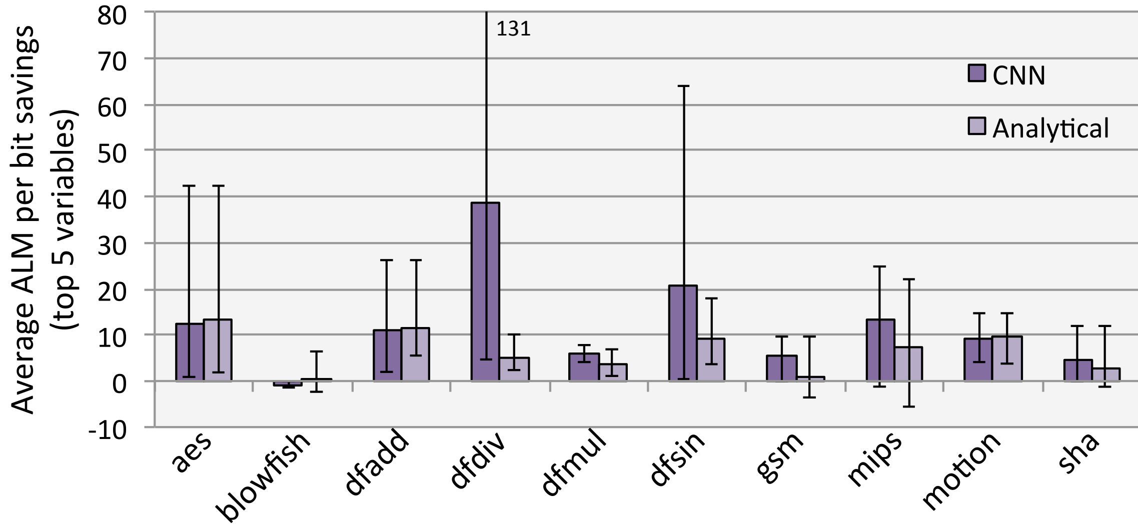


Figure 4.20: ALM per bit savings averaged across predicted top 5 variables. Each bar shows the average, and each accompanying line shows the lowest and highest savings of the 5 variables.

4.7 Conclusion

In this chapter, we present Sensei, an area reduction advisor that informs HLS users of the potential for ALM savings through the use of arbitrary bitwidth specification in the `C` source code. We evaluate two different predictor implementations within Sensei, one based on a typical analytical method often used in HLS, and one based on a CNN. For the CNN-based predictor, we explore the design space of input representations, different network parameters, and different CNN model architectures. We observe that in terms of the ability to rank the top variables by their impact on post-synthesis ALM reduction, both predictors are fairly strong, with the CNN-based predictor having an edge on most cases. However, when evaluating the ability to produce an estimate on the amount of ALM reduction, the CNN-based predictor outperforms the analytical predictor. Overall, the results are quite encouraging, as we believe this to be the first work to apply CNNs to predict post-synthesis area in HLS.

Chapter 5

Conclusion

In this thesis, we explore the extent to which the area footprint of an HLS-generated circuit can be reduced via information available at the source-code level. We provided a bitwidth minimization framework, which automatically identifies and removes any unused or non-toggling bits from the program's datapath. The BWM framework additionally introduces the ability for users to declare arbitrary-width datatypes in their software source code. The combination of: 1) the explicit declaration of variables requiring reduced width by the HLS user, and 2) the automatic width constraint propagation throughout the program by a compiler pass, provides a new way to improve the area efficiency of HLS-generated circuits.

To complement the BWM framework, we introduce Sensei: an advisor framework that aids the HLS user in choosing which variables to consider for the use of arbitrary-width datatypes. It does this by ranking all variables within a program based on their impact on post-synthesis area. Sensei provides a novel take on area prediction in HLS, employing a CNN to predict post-synthesis area reduction using the DFG of a program. This proof-of-concept evaluation demonstrates the potential of employing CNNs in CAD, particularly HLS. In summary, our work addresses the inefficiency that arises in HLS, when the high-level language imposes standard over-provisioned datatypes when the

underlying hardware can support arbitrary-width datapaths.

5.1 Future Work

Beyond the scope of this thesis, new research directions can be explored to further optimize the performance and area efficiency of HLS-generated circuits. For the BWM framework, the arbitrary-width constraints are reflected in the datapath width of the generated circuit, but we make no modifications to the allocation, scheduling, and binding algorithms. The introduction of arbitrary widths can change some assumptions used in the current algorithms and add an extra dimension, which opens the scope for novel ideas to be explored in these algorithms: On the allocation side, functional units, such as multipliers of different width, are typically considered as distinct resources. It will be interesting to explore whether it is beneficial to eliminate the instantiation of functional units at certain widths and enforce sharing with a functional unit of larger width (or multiple functional units of smaller widths). Corresponding changes in the scheduling and binding algorithms would need to be implemented to facilitate this change.

Closely related, we can explore whether there are any benefits in making the binding algorithm treat functional units as the same resource if they are closely related in size and are of the same type. By doing this, we can better distribute compute requirements to compute resources. This would reduce the size of multiplexers and thus improve area efficiency and circuit operating frequency.

Another direction for exploration is the efficiency of memory usage. More work can be done to explore the tradeoffs of reducing the width of arrays. On one hand, reducing the size of arrays can reduce the number of memory banks used. On the other hand, keeping arrays at the same size as other arrays can enable sharing of memory banks across different arrays, potentially reducing the number of unused bits in memory banks. Additionally, with the capability of automated array partitioning [11], it may be worth

investigating the merits of having arrays that include elements with different widths. The partitioning scheme can account for this variation and select partitions based on the width requirement of each element.

In terms of the Sensei framework, a natural extension is to extend the predictor to consider DSP and BRAM usage. It may also be possible to train CNNs to accurately estimate post-routing circuit delay or power consumption at the HLS stage, thereby enabling better decisions to be made both in the HLS tools themselves and by the programmers.

On top of the support for integer arithmetic operations, both the BWM framework and Sensei can be extended to support floating point operations. An important avenue for exploration when dealing with floating point numbers is the precision of the operations.

In the fields of machine learning [20] and approximate computing [33], many applications do not need full precision since they are inherently tolerant to errors. Both the BWM framework and Sensei could be applied to these cases. The BWM framework can be extended to propagate not only bitmasks, but also error tolerances as specified by the HLS user. This requires designing forward and backward transit functions for error tolerance, which poses new and interesting challenges. Sensei can also be extended to predict application error in the presence of reduced precision. It could rank and find the top variables that not only incur high area savings, but also minimize error. It could achieve area savings that were not possible before by exploiting the variables' error tolerance.

Appendices

Appendix A

Bitwidth Minimization Framework Data

A.1 Raw Data for different Components of Bitwidth Minimization Framework

bench	ALM	REG	DSP	RAM	Fmax
adpcm	0.906	0.891	1.286	1.000	1.143
aes	0.743	0.689	-	1.000	1.072
blowfish	0.982	0.970	-	1.000	1.006
dfadd	0.996	0.996	-	1.000	1.078
dfdiv	0.981	0.954	0.375	0.985	1.037
dfmul	0.960	0.974	0.375	1.000	1.114
dfsin	0.990	0.970	0.412	0.984	1.137
gsm	0.897	0.928	0.517	1.000	1.015
jpeg	0.923	0.874	1.141	0.985	0.983
mips	0.962	0.837	0.500	1.000	1.303
motion	0.993	0.981	-	1.000	0.984
sha	0.860	0.828	-	1.000	0.991
AVERAGE	0.933	0.908	0.658	0.996	1.072

Table A.1: Normalized circuit area and performance for base BM

bench	ALM	REG	DSP	RAM	Fmax
adpcm	0.906	0.891	1.286	1.000	1.143
aes	0.743	0.689	-	1.000	1.072
blowfish	0.982	0.970	-	1.000	1.006
dfadd	0.989	0.993	-	1.000	1.100
dfdiv	0.971	1.008	0.375	0.537	1.091
dfmul	0.960	0.971	0.375	1.000	1.101
dfsin	0.976	1.000	0.412	0.516	1.124
gsm	0.899	0.926	0.517	1.000	0.920
jpeg	0.897	0.858	1.141	0.985	0.957
mips	0.974	0.835	0.500	1.000	1.384
motion	0.993	0.982	-	1.000	0.984
sha	0.860	0.828	-	1.000	0.991
AVERAGE	0.929	0.913	0.658	0.920	1.073

Table A.2: Normalized circuit area and performance for function propagation

bench	ALM	REG	DSP	RAM	Fmax
adpcm	0.940	0.930	1.190	1.000	1.229
aes	0.743	0.689	-	1.000	1.072
blowfish	0.982	0.970	-	1.000	1.006
dfadd	0.996	0.996	-	1.000	1.078
dfdiv	0.981	0.954	0.375	0.985	1.037
dfmul	0.960	0.974	0.375	1.000	1.114
dfsin	0.990	0.970	0.412	0.984	1.137
gsm	0.897	0.928	0.517	1.000	1.015
jpeg	0.928	0.874	1.141	0.985	0.983
mips	0.962	0.837	0.500	1.000	1.303
motion	0.993	0.981	-	1.000	0.984
sha	0.860	0.828	-	1.000	0.991
AVERAGE	0.936	0.911	0.644	0.996	1.079

Table A.3: Normalized Circuit Area and Performance for Cross Memory Propagation

bench	ALM	REG	DSP	RAM	Fmax
adpcm	0.906	0.891	1.286	1.000	1.143
aes	0.743	0.689	-	1.000	1.072
blowfish	0.982	0.970	-	1.000	1.006
dfadd	0.989	0.993	-	1.000	1.100
dfdiv	0.971	1.008	0.375	0.537	1.091
dfmul	0.960	0.971	0.375	1.000	1.101
dfsin	0.976	1.000	0.412	0.516	1.124
gsm	0.899	0.926	0.517	1.000	0.920
jpeg	0.897	0.858	1.141	0.985	0.957
mips	0.974	0.835	0.500	1.000	1.384
motion	0.993	0.981	-	1.000	0.984
sha	0.860	0.828	-	1.000	0.991
AVERAGE	0.929	0.912	0.658	0.920	1.073

Table A.4: Normalized Circuit Area and Performance for All Bitmask Propagation

Appendix B

Advisor Framework Data

B.1 Raw Data for profiling results of area model

Operation	Width	ALMs	Reg	Mbits	DSP	Width	ALMs	Reg	Mbits	DSP	delALM
ADD	32	17	-	-	-	16	9	-	-	-	8
SUB	32	17	-	-	-	16	9	-	-	-	8
MUL	32	24	-	-	3	16	1	-	-	1	23
SDIV	32	909	2116	744	-	16	245	664	-	-	664
UDIV	32	840	2094	686	-	16	213	632	-	-	627
SMOD	32	909	2116	744	-	16	245	664	-	-	664
UMOD	32	840	2094	686	-	16	213	632	-	-	627
AND	32	17	-	-	-	16	9	-	-	-	8
OR	32	17	-	-	-	16	9	-	-	-	8
XOR	32	17	-	-	-	16	9	-	-	-	8
SHL	32	72	-	-	-	16	26	-	-	-	46
LSHR	32	72	-	-	-	16	26	-	-	-	46
ASHR	32	72	-	-	-	16	26	-	-	-	46
ICMP	32	14	-	-	-	16	7	-	-	-	7
SELECT	32	17	-	-	-	16	9	-	-	-	8

Table B.1: Area model for operations.

Bibliography

[1] Cadence. C-to-Silicon Compiler. http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx.

[2] Calypto Design Systems. Catapult: Product Family Overview. http://calypto.com/en/products/catapult/overview.

[3] clang: a C language family frontend for LLVM. https://clang.llvm.org/.

[4] Cyclone V Device Overview. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf.

[5] Introduction to the Quartus II Software. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/intro_to_quartus2.pdf.

[6] Xilinx Inc. Vivado Design Suite - VivadoHLS. http://www.xilinx.com/products/design-tools/vivado/index.htm.

[7] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An opencl deep learning accelerator on arria 10. In *ACM FPGA*, pages 55–64, 2017.

[8] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00375)*, pages 70–80, 1999.

[9] Victor Hugo Sperle Campos, Raphael Ernani Rodrigues, Igor Rafael de Assis Costa, and Fernando Magno Quintão Pereira. Speed and precision in range analysis. In *Proceedings of the 16th Brazilian Conference on Programming Languages*, SBLP'12, pages 42–56, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[11] Yu Ting Chen and Jason Anderson. Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software. In *FPL*, 2017.

[12] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[13] Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng. Bitwidth-aware scheduling and binding in high-level synthesis. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 856–861, New York, NY, USA, 2005. ACM.

[14] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, July 2009.

[15] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design Test of Computers*, 11(4):44–54, Winter 1994.

[16] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[17] Xitong Gao, John Wickerson, and George A. Constantinides. Automatically optimizing the latency, area, and accuracy of c programs for high-level synthesis. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 234–243, New York, NY, USA, 2016. ACM.

[18] Marcel Gort. *Fast CAD for FPGAs*. PhD thesis, University of Toronto, June 2014.

[19] Marcel Gort and Jason Helge Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, pages 773–779, 2013.

[20] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[21] Isabelle Guyon. A scaling law for the validation-set training-set size ratio. In *AT & T Bell Laboratories*, 1997.

[22] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou,

and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[25] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Fast area estimation to support compiler optimizations in fpga-based reconfigurable systems. In *IEEE FCCM*, page 239, 2002.

[26] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM CGO*, pages 75–88, 2004.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 05 2015.

[28] Hung-Yi Liu and Luca P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *ACM/IEEE DAC*, pages 50:1–50:7, 2013.

[29] Junyi Liu, John Wickerson, and George A. Constantinides. Loop splitting for efficient pipelining in high-level synthesis. *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 00:72–79, 2016.

[30] Charles Lo and Paul Chow. Model-based optimization of high level synthesis directives. In *FPL*, pages 1–10, 2016.

[31] Chris Loken, Daniel Gruner, Leslie Groer, W Peltier, Neil Bunn, Michael Craig, Teresa Henriques, Jillian Dempsey, Ching-Hsing Yu, Joseph Chen, L Jonathan Dursi, Jason Chong, Scott Northrup, Jaime Pinto, Neil Knecht, and Ramses Van Zon. Scinet: Lessons learned from building a power-efficient top-20 system and data centre. 256:012026, 12 2010.

[32] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transac-*

tions on Computer-Aided Design of Integrated Circuits and Systems, 20(11):1355–1371, Nov 2001.

[33] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, March 2016.

[34] S. Sarkar, S. Dabral, P. K. Tiwari, and R. S. Mitra. Lessons and experiences with high-level synthesis. *IEEE Design Test of Computers*, 26(4):34–45, July 2009.

[35] Benjamin Carrión Schäfer and Kazutoshi Wakabayashi. Machine learning predictive modelling high-level synthesis design space exploration. *IET Computers & Digital Techniques*, 6(3):153–159, 2012.

[36] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[37] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[38] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *ACM PLDI*, pages 165–176, 2002.

[39] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 108–120, New York, NY, USA, 2000. ACM.

[40] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *ACM FPGA*, pages 65–74, 2017.

[41] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 127–134, 2010.

[42] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 12:1–12:8, New York, NY, USA, 2013. ACM.

[43] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: A case study using vivado hls. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 362–365, Dec 2013.

[44] Jiyu Zhang, Zhiru Zhang, Sheng Zhou, Mingxing Tan, Xianhua Liu, Xu Cheng, and Jason Cong. Bit-level Optimization for High-level Synthesis and FPGA-based Acceleration. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 59–68, New York, NY, USA, 2010. ACM.