Bitwidth-Optimized Hardware Accelerators with Software Fallback

Ana Klimovic and Jason H. Anderson
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
Email: ana.klimovic@alum.utoronto.ca, janders@eecg.toronto.edu

***Abstract*—We propose the high-level synthesis of an FPGA-based hybrid computing system, where the implementations of compute-intensive functions are available in both software, and as hardware accelerators. The accelerators are optimized to handle *common-case* inputs, as opposed to worst-case inputs, allowing accelerator area to be reduced by 28%, on average, while retaining the majority of performance advantages associated with a hardware versus software implementation. When inputs exceed the range that the hardware accelerators can handle, a software fallback is automatically triggered. Optimization of the accelerator area is achieved by reducing datapath widths based on application profiling of variable ranges in software (under typical datasets). The selected widths are passed to a high-level synthesis tool which generates the accelerator for a given function. The optimized accelerators with software fallback capability are generated automatically by our framework, with minimal user intervention. Our study explores the trade-offs of delay and area for benchmarks implemented on an Altera Cyclone II FPGA.**

I. Introduction

Improving computing performance by increasing clock frequency is a trend of the past. The power and heat dissipation limitations of today's chips demand a more scalable approach to raise computational throughput. A recent approach to dealing with this challenge is that of heterogeneous computing. In a heterogeneous computing system, performance and energy efficiency are improved by implementing computations on different types of computing cores and tailoring each core to the specific type of computation for which it is used. Field-programmable gate arrays (FPGAs) are an attractive platform to implement such systems, as they can be configured to realize *any* digital circuit. FPGAs can therefore be used to implement accelerators that work in tandem with standard processors to improve performance. In such a hybrid processor/accelerator system, the processor executes sequential operations in software, while the FPGA-based hardware accelerators efficiently perform compute-intensive work in parallel. High-level synthesis (HLS) [9] raises the level of abstraction for hardware design by enabling a user to describe computations in software and automatically obtain a hardware implementation. This makes HLS a natural choice for the design and implementation of FPGA-based accelerators in hybrid systems. This paper centers on the high-level synthesis of FPGA-based accelerators in hybrid systems, and in particular, presents a new approach for optimizing accelerator area.

Traditionally, hardware design is constrained by worst-case inputs. This means that even if significant hardware area or power optimizations are possible for handling solely the common-case inputs, such optimizations are not incorporated into the design unless the resulting hardware can still correctly handle *every* valid input case. Consider the simple example of a ripple-carry adder, and assume that in most cases, the adder inputs lie in the range of 0 to 100, yet in rare cases, inputs as large as 1,000,000 may occur. The traditional approach calls for a 20-bit adder, capable of handling the worst-case inputs. However, if the adder need only handle the common cases, 7 bits is sufficient – a significant area reduction vs. the 20-bit case. We propose to extend the concept illustrated by the adder example to the design of entire accelerators in FPGA-based processor/accelerator systems. That is, we propose to optimize hardware accelerators for computations involving common-case inputs and provide automatic detection and software fallback (executed on the processor) for any cases that the over-optimized hardware cannot handle.

Our overall approach is as follows: Given a program with one or more functions intended for implementation as an FPGA-based accelerator, we profile the program in software using a common-case set of inputs to determine the maximum and minimum values of program variables. The profiled variable ranges are passed to a compiler-based range analysis tool [13], which propagates values forward and backward through the program's dataflow graph, potentially reducing variable ranges further. Next, we modify the program's intermediate representation (assembly code) to add input-checking functionality to the accelerator, consisting of comparing the run-time values of variables with their common-case ranges determined from profiling. For variables corresponding to accelerator inputs (i.e. values loaded from memory), when a run-time input value lies outside of its common-case range, the accelerator returns an error to the processor. The processor subsequently re-executes the computations in software for the out-of-range inputs. The modified intermediate representation of the program is passed to a high-level synthesis (HLS) tool [7], which produces a bitwidth-optimized accelerator capable of handling the common-case inputs and incorporating the desired software fallback detection behavior. The result is a hybrid system that delivers high performance for common-case inputs by way of the area-optimized accelerators, and yet uses considerably less area relative to a system with accelerators that must support all inputs.

In an experimental study, we apply the proposed approach to 7 benchmarks and assess the area and performance impact. We demonstrate that, on average, area-reductions of 28% are achieved vs. a baseline hybrid processor/accelerator system

978-1-4799-2198-0/13/$31.00 ©2013 IEEE

(generated by the HLS tool without bitwidth optimization). For the common-case inputs, overall cycle latency (i.e. the # of cycles needed for benchmark execution) is increased by 1.2× vs. the baseline, owing to the extra clock cycles needed for checking input legality. Overall, the approach thus provides a significant win in area-delay product, and we believe it will be especially useful in area-constrained settings. As our approach hinges on the early identification and profiling of common-case inputs, a natural question that arises concerns the impact of out-of-range inputs on cycle latency. Out-of-range inputs trigger software fallbacks and re-execution on the processor, thereby decreasing performance vs. deploying accelerators that can handle all inputs. To analyze this, we present results showing the increase in cycle latency across the benchmarks vs. the fraction of out-of-range inputs. To the authors' knowledge, the proposed use of bitwidth-optimized accelerators for common-case inputs in conjunction with software fallbacks represents a new approach to the optimization of hybrid systems.

The remainder of this paper is organized as follows: Section II presents background on hybrid processor/accelerator systems, high-level synthesis and range analysis. Section III describes our proposed framework, incorporating profiling, range analysis, and the automatic insertion of software fallbacks for hardware-accelerated functions in the targeted system. An experimental study is presented in Section IV. Section V offers conclusions and suggestions for future work.

II. Background

A. High-Level Synthesis (HLS) for Hybrid Systems

We implemented the proposed approach within the LegUp open-source high-level synthesis framework, developed at the University of Toronto [7]. LegUp automatically synthesizes a `C` benchmark program to a hybrid FPGA-based processor/accelerator system. The user specifies which `C` functions (and their descendants) should be realized as hardware accelerators, with the remaining functions executing in software on a MIPS soft processor [21]. For the hardware-designated functions, LegUp automatically deletes the original version of the functions from the software and replaces them with wrapper functions that invoke the accelerators, pass arguments and receive results via a memory-mapped bus interface – Altera's Avalon interface [1].

Fig. 1 shows the system architecture produced by LegUp. The processor and accelerators share a memory hierarchy, comprising a L1 cache (implemented on the FPGA), as well as off-chip memory. The architecture permits the processor and accelerators to share data with one another in two ways: 1) across the on-chip bus interface, or 2) through shared memory. Two Altera boards are supported by LegUp: Altera's Cyclone II-based DE2 board, and the Stratix IV-based DE4 board.

High-level synthesis in LegUp is implemented as back-end compiler passes within the open-source LLVM (low-level virtual machine) compiler framework [18]. LLVM parses the input `C` program, then transforms it into an intermediate representation (IR). The IR resembles machine-independent assembly code, with primitive instructions for computations and control. LLVM then performs a set of standard compiler optimizations on the IR, e.g. dead code elimination, constant

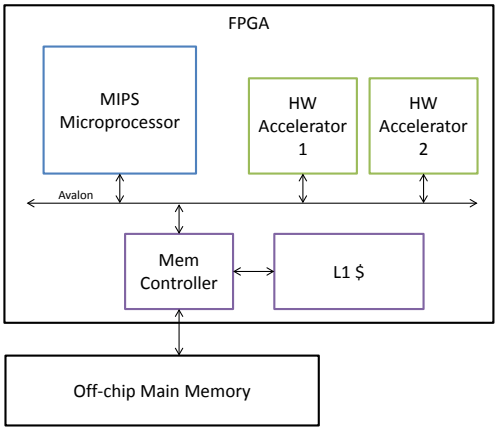


Fig. 1: Hybrid processor/accelerator architecture on FPGA.

folding, etc. LegUp accepts the optimized IR as input, performs high-level synthesis, and produces Verilog RTL for the selected accelerators. The Verilog can then be synthesized by vendor tools, such as Altera's Quartus II. Altera's SOPC Builder tool is used to connect the accelerators to the MIPS processor.

B. Range Analysis

The `C` language defines a set of standard datatypes which are 8, 16, 32, or 64 bits in length. This coarse quantization inevitably leads to the use of more bits than are actually required to represent program variables. There is little advantage for a programmer to optimize the bitwidths of variables, as processor datapaths are of fixed width. However, when a program is synthesized to hardware using HLS, optimizing the bit-level representation of variables leads directly to circuit area and power reductions [11].

Range analysis determines the maximum and minimum values that a program's variables take on during execution, thereby permitting a reduction in the number of bits needed to represent the variables. There are two forms of range analysis: *static* and *dynamic*. Static analysis infers ranges solely using information available at compile-time, such as constants in the code. For example, a loop index `i` that ranges from 0 to 100 (specified as constant in the code) can be represented using 7 bits. Such ranges are then propagated forward and backward through the program's dataflow graph [13], [19], allowing ranges for other variables to be inferred. Bitwidth reductions made via static analysis preserve program correctness; the program will execute correctly for *all* input datasets. Dynamic range analysis, on the other hand, infers ranges based on run-time information under a particular input dataset, permitting greater bitwidth reductions, with the caveat that the bitwidth-reduced program is no longer guaranteed to work for *all* inputs.

A number of approaches to bitwidth minimization have been proposed in recent literature, including [15] which minimizes bitwidths subject to user-specified accuracy requirements, and [17] which employs SAT-modulo theory. MiniBit [16] uses a static analysis technique via affine arithmetic to optimize bitwidths of fixed-point designs. Range analysis is

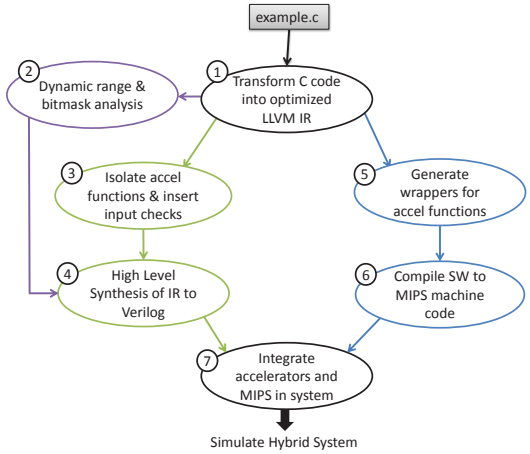


Fig. 2: Hybrid design flow for HLS of hardware accelerators with software fallback.

used to target integer bits, while precision analysis is applied to fraction bits. [22] introduces a static bitwidth minimization algorithm capable of reducing slice usage by almost 30%. The work in [5] uses dynamic analysis to determine bitwidths, subject to an allowable error constraint. The recent release of LegUp HLS (ver. 3.0) includes both static and dynamic range analysis [13], which we leverage in our research. Note that while we choose to use the static and dynamic analysis algorithms implemented by [13], our work is orthogonal to the particular bitwidth minimization approach used.

The work in [13] demonstrated an average area reduction of 9% using static range analysis, and a 34% area reduction with dynamic analysis. However, the dynamic analysis results in [13] are lower bound circuit areas, as the circuits are no longer correct for all inputs. Our work aims to provide most of the area-reduction benefits of dynamic range analysis, while retaining functional correctness.

III. Bitwidth-Optimized Accelerators with Software Fallback

We now delve into our proposed approach for optimizing the area of accelerators to handle common-case inputs, with software fallback to handle the worst-case inputs.

Fig. 2 shows the flow of our optimization framework, which surrounds and makes use of the LegUp high-level synthesis tool. The left side of the diagram shows profiling (in purple) and the hardware flow (in green), which result in a Verilog hardware description of the area-optimized accelerator in the hybrid system. The right side of the diagram shows the software flow (in blue) which produces an executable to be run on the MIPS microprocessor.

The flow begins with a `C` program *example.c* provided by the user. The user also provides a `Tcl` file (not shown in the diagram) to identify which function(s) to accelerate in hardware. In step 1, the `C` program *example.c* is transformed into the LLVM intermediate representation (IR) and standard compiler optimizations are applied (LegUp's default optimization level is `-O3`).

```
  1:  %28 = load i32* %ptr, align 4
+ 2:  %check1 = icmp sgt i32 %28, 511
+ 3:  br i1 %check1, label %Invalid,label %L1
+ 4: L1:
+ 5:   %check2 = icmp slt i32 %28, -512
+ 6:   br i1 %check2, label %Invalid,label %L2
+ 7: L2:
  8:   %29 = add nsw i32 %28, %27
  9:   ret i32 %29
+10: Invalid:
+11:   volatile store i32 1,i32* @global_status
+12:   ret i32 -1
```

Fig. 3: Intermediate Representation (IR) of accelerated function with inserted comparison and branch instructions (marked (+) for HLS of comparators in the hardware accelerator.

In step 2, the program is dynamically profiled by execution with a common-case input dataset (provided by user) to characterize the ranges of program variables. Dynamic profiling is done using the LLVM interpreter, which can directly execute the program's optimized intermediate representation, as produced in step 1. We modified the interpreter to track and report the minimum and maximum value each variable in the IR takes on during its execution. The ranges of variables from dynamic profiling are provided to the compiler-based range analysis tool described in [13], which propagates them through the dataflow graph, tightening the ranges further. At the end of this step, we determine the number of bits needed for each variable to handle the common-case inputs. Note that each variable can have a different number of bits; that is, we permit non-uniform bitwidths. The computed bitwidths will be used in the high-level synthesis of the accelerators (in step 4 discussed below).

In step 3, we modify the IR of the functions designated for implementation as hardware accelerators. We add input-checking functionality to ensure that inputs lie within the common-case ranges determined in step 2. When inputs fall outside the range, the accelerator will exit early and indicate to the processor that software fallback is needed. Fig. 3 shows a portion of the IR of a function selected for implementation as a hardware accelerator. The lines without a + are the original IR, prior to our modifications; the lines with a + were inserted automatically by our tool. Though a detailed explanation of the LLVM IR is outside the scope of this paper, the example nevertheless serves to illustrate our approach. Observe that the original IR consists of a `load` instruction followed by an addition (`add`) instruction whose result is returned to the processor (`ret`) (see lines 1, 8 and 9). Assuming that the `load` instruction corresponds to an input to the accelerator, our tool inserts compare/branch instructions to check whether the input value is larger than the maximum of the common-case range, or smaller than the minimum of the common-case range, and if so, branch accordingly to the exit-early case. Indeed, line 2 contains an integer compare (`icmp`) instruction that checks whether the value loaded is higher than 511. Line 3 is a branch instruction (`br`) that branches to the code labelled `Invalid` (exit-early code) in lines 10-12 if the input value is higher than 511. The exit-early code stores a 1 in a special memory address called `global_status` (line 11), and returns early to the processor (line 12). The processor will

```
  1: int AccelFunc_wrapper(int arg1, int* arg2) {
  2:   *AccelFunc_ARG1=(volatile int) arg1;
  3:   *AccelFunc_ARG2=(volatile int) arg2;
  4:   *AccelFunc_STATUS=1;

+ 5:     //check if accel returned prematurely
+ 6:     if (global_status){
+ 7:         printf("Software fallback ...");
+ 8:         return AccelFunc(arg1, arg2);
+ 9:     }
+10:     //otherwise return accelerator result
+11:     else
 12:         return *AccelFunc_DATA;
 13: }
```

Fig. 4: Software wrapper C code to call hardware accelerator. Lines marked "+" are inserted into original wrapper to check for and execute software fallback if necessary.

check the memory address to determine if software fallback is needed. Lines 5 and 6 check the analogous case of the input being smaller than the minimum of the common-case range. It is important to realize that the additional compare/branch instructions are only inserted to check data received from load instructions that correspond to accelerator inputs; that is, additional instructions are not inserted to check intermediate program values computed by math or logical operations.

In step 4 of Fig. 2, we use the LegUp HLS tool to generate a Verilog description of a bitwidth-optimized accelerator whose functionality is defined in step 3. The compare/branch instructions inserted in the IR result in the high-level synthesis of additional comparators in the accelerator circuit, and additional states in its finite state machine. In our experimental study, we assess area and performance overhead of the additional hardware complexity.

On the software side, in step 5, we generate a software wrapper for each accelerated function, as shown in Fig. 4. As above, the lines with a + are the result of modifications we make to the LegUp flow, while the lines of the original LegUp-generated wrapper have no +. Beginning with the original LegUp wrapper code, lines 2 and 3 are memory-mapped writes that pass function arguments to the accelerator over the Avalon interface[1]. Line 4 is a memory-mapped write that invokes the accelerator, and line 12 is a memory-mapped read that retrieves the value returned by the accelerator. While the accelerator is operating, the processor is stalled. Our tool inserts line 6, which checks the special memory address discussed above to determine if software fallback is necessary, and if so, invokes the software version of the function (line 8), executed on the MIPS.

In step 6, the software IR is compiled by LLVM to MIPS machine code. Finally, in step 7, we use Altera's SOPC Builder to integrate the hardware accelerators with the microprocessor hardware [3].

One important condition for successful software fallback execution in our hybrid system is that an accelerated function must not overwrite its inputs. Hardware accelerators that operate on data "in place" in memory will corrupt inputs with intermediate results. If the accelerator detects out-of-range data half-way through execution, when the system re-executes the function in software, memory accesses will retrieve corrupted values stored by the accelerator. Thus, we limit our study to functions that do not overwrite their inputs and do not write to shared global memory in a manner that would prevent successful software fallback if required.

Finally, it is worth pointing out that our flow (Fig. 2) is completely automated, requiring minimal user intervention.

[1] `AccelFunc_ARG1` and `AccelFunc_ARG2` are pointers to memory addresses defined automatically by LegUp in the hybrid system. Writes to these addresses correspond to data transfers across the Avalon interface.

IV. Experimental Study

In this section, we present our experimental study of the proposed bitwidth-optimized accelerators with software fallback. Among the benchmarks included with the LegUp tool [7] distribution, we select the 7 benchmarks that satisfy the following two criteria: 1) the benchmark does not overwrite its input values in memory, and 2) the area reduction of the benchmark's accelerator is significantly greater using dynamic range analysis compared to static range analysis. The first criterion is necessary for correct execution of a software fallback, as described in Section III. The second criterion is desired for applications of our proposed method of bitwidth optimization in order to obtain notable area reduction beyond the reductions achieved with static range analysis. For each of the benchmarks below, we select the most compute-intensive function (and its descendants) to implement as a hardware accelerator. Each benchmark calls the accelerator multiple times with different inputs. The 7 benchmarks used are:

1) Dhrystone: The well-known classic integer benchmark. The accelerator is called 20 times, once for each run of the entire benchmark.
2) LOS: The line of sight (LOS) benchmark uses the Bresenhams line algorithm [6] to determine whether each pixel in a 2-dimensional grid is visible from the source [8]. The hardware accelerator is called 512 times (once for each horizontal line).
3) Histogram: The histogram benchmark accumulates 36,000 integers into 5 equally-sized bins. The hardware accelerator is called 100 times on different portions of the input to complete the task.
4) ADPCM: The adaptive differential pulse code modulation (ADPCM) decoder and encoder benchmark is part of the CHStone suite [14]. We accelerate the encoder function and call it 100 times.
5) GSM: This benchmark is adopted from the CHStone benchmark suite [14]. We accelerate the autocorrelation function (part of linear predictive coding) and call the accelerator 100 times.
6) FIR: This benchmark implements a finite impulse response (FIR) filter, calling a hardware accelerator 512 times.
7) Black-Scholes: A quantitative finance benchmark to value options using a Monte Carlo approach. We call the hardware accelerator with 10 different random generator seeds.

The HLS-generated hybrid systems considered in this study are targeted to the Altera Cyclone II 90nm FPGA. Performance is evaluated by simulating each circuit in ModelSim to determine the number of cycles required to execute the

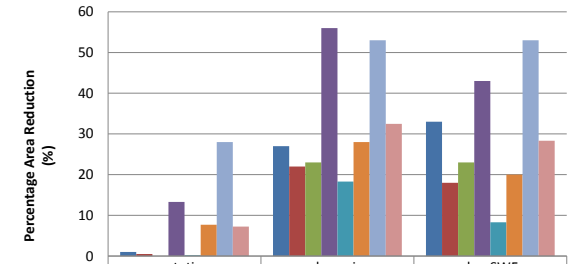
Accelerator Configuration	Input checking functionality in HW?	Bitwidth Reduction Method
Orig	No	None
Static	No	Static analysis
Dyn-orig	No	Dynamic analysis
Dyn-SWF	Yes	Dynamic analysis

TABLE I: Hardware configurations for test scenarios.

benchmark and return the correct result; we refer to this as the benchmark's *cycle latency*. We also assess and report each circuit's clock frequency (F_{max}) after placement and routing by Altera's Quartus II tool [4]. Circuit area is measured by the number of Cyclone II Logic Elements (LEs), where each LE is composed of a 4-input look-up-table (LUT) and a flip-flop.

Table I lists the flows and hardware configurations tested, and defines the notation used throughout the rest of this paper. The *orig* configuration involves no bitwidth reductions or modifications to the accelerator in high-level synthesis (i.e. default LegUp), thus serving as the baseline test for area comparisons. Note that although the *orig* configuration does not make use of static or dynamic range analysis to reduce bitwidths during HLS, RTL synthesis within Altera's Quartus II includes static bitwidth reduction as part of its standard set of optimizations. The two configurations *static* and *dyn-orig* represent static and dynamic bitwidth reduction, respectively, in HLS using the techniques in [13]. Recall that, as mentioned above, the *dyn-orig* systems only work for the profiled inputs – they are *not* guaranteed to work for all inputs. Finally, the *dyn-SWF* configuration represents the systems generated by the proposed techniques: bitwidth-optimized hardware with software fallback. By comparing the performance and area of *dyn-orig* and *dyn-SWF*, we can determine the overhead associated with incorporating out-of-range input detection and software fallback capability into the accelerators.

Fig. 5 shows the percentage reduction in circuit area achieved by reducing the bitwidth of variables for the *static, dyn-orig* and *dyn-SWF* configurations. On average, we find that accelerator circuit area is reduced by 7.2% with static range analysis (*static* configuration) and 33% with dynamic range analysis (*dyn-orig* configuration). The static and dynamic results align very closely to those reported in [13]. Due to the extra area overhead associated with comparators for input checking, required to detect inputs that fall outside of the profiled range, the effective circuit area reduction with dynamic range analysis is 28% (*dyn-SWF* configuration). Thus, we observe that *most* of the area-reduction benefits offered by dynamic range analysis are retained in *dyn-SWF*, even with the added hardware complexity needed to incorporate software fallback. The Dhrystone benchmark actually benefits from a 6% *decrease* in circuit area when comparators are added to the circuit. The decrease in circuit area is attributed to the ability of the Quartus II synthesis tool to simplify the existing circuit based on the constants fed into the new comparators (perhaps through the tool's own internal constant propagation pass). To verify this, we manually increased the comparison values in the Verilog hardware description of the accelerator (generated automatically by our flow) to large 32-bit constants



	static	dyn-orig	dyn-SWF
Dhrystone	1	27	33
LOS	0.5	22	18
Histogram	0	23	23
ADPCM	13.3	56	43
GSM	0.2	18.3	8.3
FIR	7.7	28	20
Black-Scholes	28	53	53
Average	7.2	32.5	28.3

Fig. 5: Percentage area reduction of hardware accelerator synthesized with static and dynamic analysis.

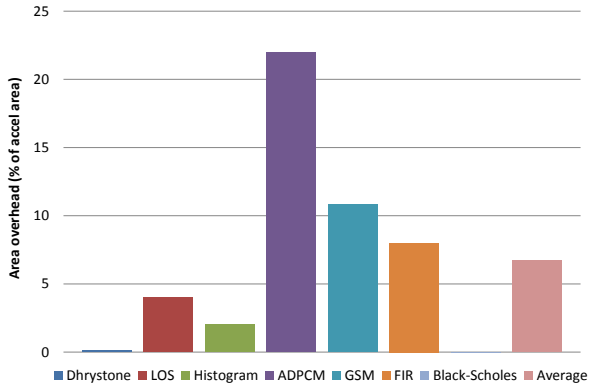
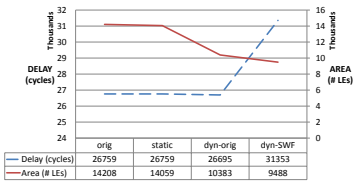


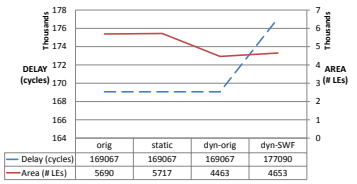
Fig. 6: Area overhead for implementation of input-checking functionality on hardware accelerators with reduced bitwidths.

and re-synthesized the circuit. As a result of our comparator modifications, the area of the *dyn-SWF* configuration increased with the introduction of a 1.3% comparator area overhead. The modified *dyn-SWF* accelerator's area reduction was 26% instead of 33% compared to *orig*.

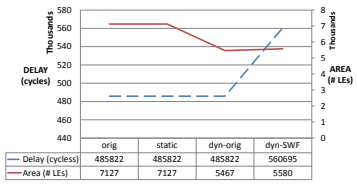
Fig. 6 shows the area overhead associated with adding comparators into the hardware accelerators. The overhead is determined by the area difference between the *dyn-SWF* and *dyn-orig* hardware configurations as a percentage of the total *dyn-SWF* hardware accelerator area. The average area overhead is 6.7% – an acceptable overhead considering the accelerator circuit area reduction of nearly 30% in the *dyn-SWF* configuration vs. the baseline case. The Black-Scholes benchmark shows the highest *dyn-SWF* area reduction (53%) and almost no comparator area, as there are only two load instructions in the accelerated function, thus only 2 comparators are synthesized. The ADPCM benchmark shows the highest comparator overhead (22%) but also a high overall circuit area reduction of 43% compared to the *orig* configuration.



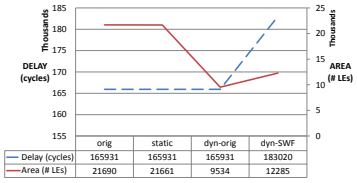
(a) Dhyrstone



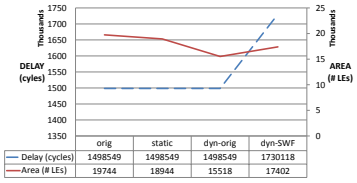
(b) LOS



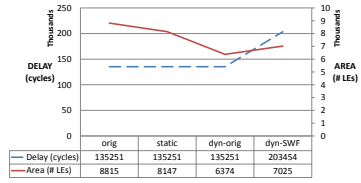
(c) Histogram



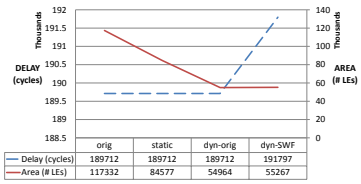
(d) ADPCM



(e) GSM



(f) FIR



(g) Black-Scholes

Fig. 7: Area and delay results for each benchmark.

Fig. 7 shows the area and cycle latency results for each benchmark in detail. In each graph, the red curve represents accelerator area (# of LEs) and the dotted blue curve represents the delay (in cycles). For each benchmark, we observe a noticeable area reduction between the static and dynamic analysis, with a slight increase when comparators are added in the *dyn-SWF* configuration. We also observe that the *static* and *dyn-orig* configurations have the same cycle latency as the original program in the hybrid system. However, cycle latency increases by 1.2× on average for the *dyn-SWF* configuration due to the extra instructions inserted to check each input against the minimum and maximum values supported by the hardware; these instructions increase the lengths of the schedules produced by HLS.

The delay results for the *dyn-SWF* configuration in Fig. 7 assume that the profiled input dataset *exactly* characterizes the run-time input range and thus, no software fallback is necessary. However, there may naturally exist cases for which it is impossible to exactly characterize an accelerator's input dataset prior to field deployment. As such, the accelerator may experience some inputs that fall outside of the profiled range, triggering fallback to software and higher cycle latencies. Fig. 8 gives results showing how cycle latency for each benchmark is affected as the number of out-of-range inputs increases. The horizontal axis represents the percentage of out-of-range inputs; the vertical axis shows the delay penalty associated with falling back to software, normalized to the case of "perfect" characterization (i.e. no accelerator inputs triggering software fallback). We produced the data in the figure by deliberately invoking the accelerators with varying numbers of out-of-range inputs, and performing ModelSim simulation to determine the cycle latency impact of running those cases on the MIPS processor.

Fig. 8 shows that the delay penalty for execution in software vs. hardware increases linearly with the percentage of out-of-range accelerator inputs. The delay penalty is under 2×, on average, when less than 15% of run-time datasets exceed the range of the profiled (common-case) dataset. If the profiled

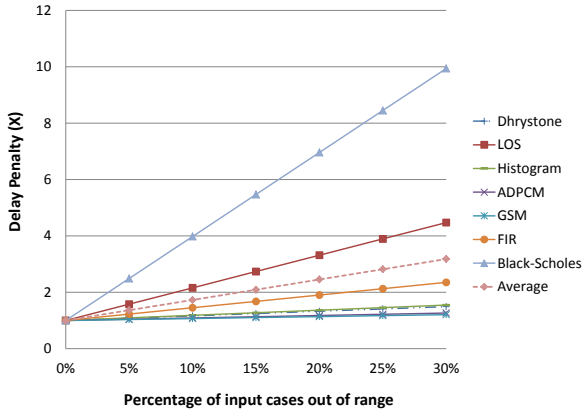


Fig. 8: Software fallback delay increase penalty in relation to the percentage of out of range input datasets.

dataset poorly characterizes the ranges of run-time variables such that only half the run-time datasets fit within the range of the profiled dataset, the delay penalty is over 4×, on average. Thus, we emphasize that our method is most beneficial for applications whose input datasets are well-characterized at the design stage, without significant variation in the field. Among the benchmarks in our study, Dhrystone, ADPCM, and Histogram show the least delay increase as the number of cases handled in software increases. The Black-Scholes benchmark shows the most drastic trade-off between delay and area. The benchmark shows the highest accelerator area reduction but also the highest delay penalty for execution in software. The high delay penalty is particularly a drawback for the Black-Scholes benchmark because the application is based on random number generation, making the run-time input dataset difficult to characterize with a single dataset. Thus, although we have shown that the Black-Scholes hardware accelerator area can be reduced by more than half using dynamic analysis, due to the high software fallback delay penalty, Black-Scholes is not a benchmark for which we would recommend applying our method.

In Table II we show the speedup of our hybrid system compared to a software-only implementation executed on the Tiger MIPS microprocessor [21]. We observe that executing on *dyn-orig* configured hardware accelerators provides a 8.3× speedup compared to software, on average. The 7.7× average speedup reported in the *dyn-SWF 0%* column of Table II shows that most of the performance benefits of a hardware vs. software implementation are preserved after the addition of comparators for out-of-range input checking. When 5% of calls to the accelerator are handled in software, the speedup of the hybrid system vs. a software-only implementation is 4.3× on average. When 10% of cases fallback to software, the speedup of the hybrid system decreases to 3.3× on average.

Table III summarizes the area reduction, area overhead and delay increase results for the cases of 0 software fallbacks and 10% of cases handled in software. On average, our method of area optimization for common-case inputs provides a 28% accelerator area reduction vs. the baseline with no bitwidth

Benchmark	Dyn-orig	Dyn-SWF 0%	Dyn-SWF 5%	Dyn-SWF 10%
Dhrystone	2.9x	2.5x	2.3x	2.1x
LOS	9.5x	9.1x	5.8x	4.2x
Histogram	3.1x	2.7x	2.5x	2.3x
ADPCM	2.0x	1.8x	1.7x	1.6x
GSM	1.8x	1.6x	1.5x	1.5x
FIR	7.9x	5.2x	4.2x	3.6x
Black-Scholes	31.1x	30.8x	12.4x	7.7x
AVERAGE	**8.3x**	**7.7x**	**4.3x**	**3.3x**

TABLE II: Speedup of our hybrid system compared to a software-only implementation on Tiger MIPS microprocessor.

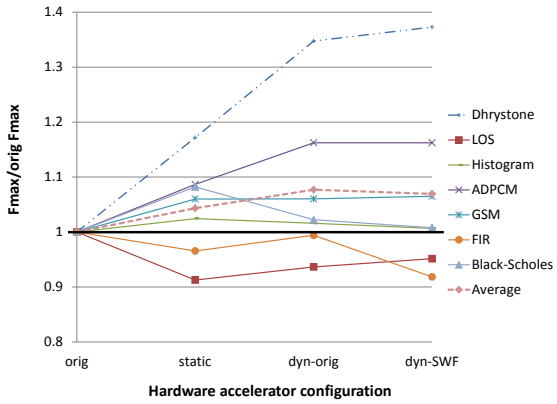


Fig. 9: Increase in maximum accelerator operating frequency for *static, dyn-orig* and *dyn-SWF* configurations compared to *orig*.

minimization in accelerators, where ∼7% of area is comparator overhead. When no inputs require software fallback, cycle latency is 1.2× that of the *dyn-orig/static* configuration, whereas falling back on software for 10% of cases results in a delay increase of 1.9×, on average. Overall, the proposed approach provides significant area reductions, at a modest performance hit, which we believe will be particularly useful in area-constrained embedded applications.

We also examined the effect of bitwidth reduction on circuit timing by measuring the maximum frequency, F_{max}, reported by the Quartus II tool after placement and routing. Fig. 9 shows the ratio of F_{max} for each hardware configuration in relation to F_{max} of the *orig* hardware accelerator configuration. For 5 out of 7 benchmarks, the maximum operating frequency of accelerators increases after the application of static or dynamic range analysis for bitwidth optimization (the ratio plotted in Fig. 9 is greater than 1 for these benchmarks). On average, the *dyn-orig* configuration allows for the highest operating frequency (9% greater than the original F_{max}) and the *dyn-SWF* configuration closely follows, allowing for an 8% increase to the original F_{max}. Static analysis alone allows for a 5% increase in frequency compared to the baseline. F_{max} measurements for the LOS and FIR benchmarks were dominated by noise introduced by the stochastic nature of the FPGA place and

Benchmark	Accelerator Area Reduction	Comparator Area Overhead	Delay Increase for 0% in SW	Delay Increase for 10% in SW
Dhrystone	33%	0%	1.2X	1.4X
LOS	23%	4%	1.1X	2.2X
Histogram	23%	2%	1.2X	1.4X
ADPCM	43%	22%	1.1X	1.2X
GSM	22%	11%	1.3X	1.2X
FIR	20%	8%	1.5X	2.2X
Black-Scholes	53%	0%	1.0X	4.0X
AVERAGE	**28%**	**6.7%**	**1.2X**	**1.9X**

TABLE III: Area and delay summary for hybrid system with *dyn-SWF* configured hardware accelerator.

route process. In general, we found that F_{max} measurements varied depending on the fitter seed provided to the Quartus II tool. The results presented in Fig. 9 are the average frequencies measured across 3 different fitter seeds. Fig. 9 shows that for the majority of benchmarks in this study, our common-case bitwidth optimization method allows for not only circuit area reduction, but also circuit simplifications that decrease critical path delays.

V. Conclusion

Hybrid computing architectures are becoming increasingly common as system architects seek to distribute computation across multiple resources to improve performance. High-level synthesis is a powerful tool which can play an important role in developing hybrid processor/accelerator systems. However, in order to be broadly embraced by industry, HLS tools must first overcome the primary obstacles associated with automated hardware design: area and delay overhead. Our work aims to bridge the gap between high-level synthesis and human-customized hardware design by allowing HLS tools to synthesize circuits tailored for common-case inputs, thus optimizing hardware bitwidths beyond the traditional limits.

In this work, we have shown that optimizing accelerator area for common-case inputs using dynamic analysis results in accelerator circuit area reduction of 28%, on average. We have shown that the synthesis of comparators for software fallback checking required by our common-case area optimization method presents a 7% overhead and a 1.2× cycle latency increase. Cycle latency in our hybrid system increases depending on how often the accelerator is called with input datasets containing values outside of the common-case range. If 10% of function calls must be executed in software, the cycle latency is 1.9× greater than if all cases were executed by the hardware accelerator. We conclude that optimizing accelerators for common-case inputs is a useful technique for reducing circuit area in applications where run-time inputs do not contain many outliers and thus a software fallback is rarely necessary. Future work involves studying the impact of common-case bitwidth reduction on the power and energy-efficiency of a hybrid processor/accelerator system.

Acknowledgment

The authors would like to thank Marcel Gort for his work and much-appreciated guidance on dynamic range analysis in LegUp which served as the starting point of this paper. We are also grateful to Jongsok (James) Choi and Andrew Canis for their foundational work on LegUp and for answering our questions along the way.

References

[1] Altera, Corp., San Jose, CA. *Avalon Interface Specification,* 2010.

[2] Altera, Corp., San Jose, CA. *Cyclone-II FPGA family datasheet*, 2012.

[3] Altera, Corp., San Jose, CA. *SOPC Builder User Guide,* 2010.

[4] Altera, Corp., San Jose, CA. *Quartus II version 10.1*, 2010.

[5] M. Arindam, D. Sinha, P. Banerjee, H. Zhou, "Low-power optimization by smart bit-width allocation in a SystemC-Based ASIC design environment", *IEEE Trans. on CAD*, vol. 26, no. 3, pp. 447-455, 2007.

[6] J.Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal,* 4, 1965.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. Brown and J. Anderson. "LegUp: An open source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. on Embedded Computing Systems*, vol. 13, no. 2, article 24, 2013.

[8] J. Choi, K. Nam, A. Canis, and J. Anderson. "Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems," *IEEE FCCM*, pp. 17-24, 2012.

[9] P. Coussy, D. Gajski, M. Meredith, A. Takach, "An introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 25, no. 4, pp. 8 – 17, 2009.

[10] DE2, Altera Corp, San Jose, CA. *DE2 Development and Education Board,* 2012.

[11] A. Gaffar, J. Clarke and G. Constantinides, "PowerBit - Power aware arithmetic bitwidth optimization," *IEEE ICFPT*, pp. 289-292, 2006.

[12] Global Standard for Mobile Association, *http://www.gsmworld.com*.

[13] M. Gort and J. Anderson, "Range and Bitmask Analysis for Hardware Optimization in High-Level Synthesis," *IEEE/ACM ASP-DAC*, pp. 773-779, 2013.

[14] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242-254, 2009.

[15] D. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk and G. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Trans. on CAD*, vol. 25, no. 10, pp. 1990-2000, 2006.

[16] D. Lee, A. Gaffar, R. Cheung, O. Mencer and W. Luk, "MiniBit: Bit-width optimization via affine arithmetic," *ACM DAC*, pp. 837-840, 2006.

[17] A. Kinsman, N. Nicolici, "Finite precision bit-width allocation using SAT-modulo theory," *IEEE/ACM DATE*, pp. 1106-1111, 2009.

[18] C. Lattner, and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *IEEE Int'l Symp. on Code Generation and Optimization*, pp. 75–86, 2004.

[19] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," *ACM PLDI*, 2000, pp. 108-120.

[20] United States Bureau of Labor Statistics, 2010.

[21] University of Cambridge. "The Tiger "MIPS" processor," *http://www.clcam.ac.uk/teaching/0910/ECAD+Arch/mips.html*, 2010.

[22] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng and J. Cong, "Bit-level optimization for high-level synthesis and FPGA-based acceleration," *ACM FPGA*, pp. 59-68, 2010.