Range and Bitmask Analysis for Hardware Optimization in High-Level Synthesis

Marcel Gort and Jason H. Anderson
Dept. of Electrical and Computer Engineering, University of Toronto
Toronto, Ontario, Canada
{gortmarc|janders}@eecg.utoronto.ca

Abstract—We consider the extent to which the bit-level representation of variables can be used to optimize hardware generated by high-level synthesis (HLS). Two approaches to bit-level optimization are considered (individually and together): 1) range analysis, and 2) bitmask analysis. Range analysis aims to predetermine min/max ranges for variables to reduce the bitwidth required to represent variables in hardware. Bitmask analysis characterizes individual bits within a word as either constants (`1` or `0`), sign bits, or unknowns, where constants/don't-cares permit hardware to be eliminated under certain conditions. Static compiler-based analysis is contrasted with dynamic profiling-based analysis in terms of their potential to impact area and speed of HLS-generated hardware. For a set of benchmarks implemented in the Altera Cyclone II FPGA, results show bit-level optimizations in HLS based on static analysis reduce circuit area by 9%, on average, while additional optimizations based on dynamic analysis provide 34% area reduction.

I. Introduction

Programs today use standard datatypes that are 8, 16, 32, or 64-bits in length. As such, programs are *over engineered* in the sense that variables are frequently represented using more bits than are actually required. For example, it is standard practice for a programmer to use a 32-bit `int` datatype for a loop index that is known to have a limited range, say, from `0` to `100`. The reason for this is straightforward: processor datapaths are necessarily generic and of fixed width. Consequently, there is little to be gained in terms of a program's performance by optimizing variable bitwidths. Conversely, when high-level synthesis (HLS) is applied to generate customized hardware for a program, hardware quality (area, speed and power) is impacted considerably by the bit-level representation of program variables. Eliminating bits leads directly to reductions in circuit area and power, and potential improvements to circuit speed. This work considers techniques for statically (i.e. at compile time) and dynamically (i.e. using run-time profiling) determining representations of variables in HLS, with the objective of optimizing hardware area.

We employ two strategies for optimizing a variable's bit-level representation: 1) range analysis and 2) bitmask analysis. Range analysis seeks to determine the maximum and minimum values that variables take on in a program's execution and in so doing, bound the number of bits required to represent the variable. As an example, consider an `int` variable in a program that is used to represent a Boolean type – the 32-bit `int` can be "shrunk" to a single bit when the program is synthesized to hardware. Likewise, 7 bits suffice to represent an `int` loop index that ranges from `0` to `100`. Variable ranges can be deduced from constants in the source code, and then propagated through a program's control-dataflow graph (CDFG) (static analysis) to infer ranges for other variables (i.e. variables indirectly tied to constants). Ranges can also be inferred from instrumented program execution (dynamic analysis). Range analysis is a well-studied topic in the compiler domain, and, for static analysis, our work leverages the recent implementation of Campos et al. [1].

Bitmask analysis, on the other hand, seeks to characterize the individual bits in a variable. For example, assume that A and B are unknown 16-bit values and consider the `C`-language statement: `Z = A & (B << 2)`. In this case, the two right-most bits of Z are guaranteed to be logic-`0` and this property can be applied to minimize the size of hardware that uses Z as an operand (e.g. if Z feeds into a multiplier, the two right-most bits of the product are guaranteed to be logic-`0`). Note that while *bitmask* analysis guarantees that Z's two LSBs are `0`, *range* analysis can infer nothing regarding Z's min and max values. The two forms of analysis thus offer complementary information. To our knowledge, ours is the first work to combine *both* range and bitmask analysis to optimize HLS-generated hardware. Our bitmask analysis is implemented within the LLVM compiler framework [2].

We evaluate bit-level optimization techniques using an open-source high-level synthesis tool [3] and target the Altera Cyclone II $90n$m commercial FPGA [4]. Results show that, based on static analysis alone that analyzes both ranges and bitmasks, circuit area can be reduced by 9%, on average, for a suite of benchmark programs [5], compared with Altera's Quartus II RTL synthesis tool, which itself significantly prunes the circuit based on constants in the RTL code. With additional dynamic profile-driven analysis, area reductions increase to 34%, on average. The remainder of this paper is organized as follows: Section II introduces related work on bitwidth minimization and gives background on high-level synthesis. Section III describes our bitwidth minimization approach. An experimental study is presented in Section IV. Section V offers conclusions and suggestions for future work.

II. Related Work

A. Range Analysis

In this work, we use the range analysis implementation described in [1], combined with our own implementation of bitmask analysis (c.f. Section III). The work in [1] is modeled after a constraint-based framework proposed by Su and Wagner [6], wherein the program is analyzed to yield a set of con-

straints defining the range of each variable. For example, consider the following code fragment:

```
1:  int j = 2;
2:  while (j < n) {
3:      ...
4:      j = j + 2;
5:  }
```

and let $J = [l, u]$ represent the range of variable `j` ($l = min, u = max$), and $N = [c, d]$ represent the range of variable `n`. Su and Wager's approach formulates a set of constraints for each variable. For variable `j`, from line 1, we know that $[2, 2] \sqsubseteq J$ and from lines 2 and 4, $(J+2) \sqcap [-\infty, d] \sqsubseteq J$, where d is the upper bound on n. The constraints for each variable are annotated onto a dependency graph, wherein each node represents a program variable and edges represent dependencies between variable ranges. The graph is constructed such that solving for the ranges of each variable is achieved by traversing the graph, and range analysis of loops is handled by applying a Bellman-Ford-like analysis. This gives a rough picture of the approach. The interested reader is referred to [1] and [6] for details.

B. Bitwidth-Aware High-Level Synthesis

Several prior works have looked at bit-level optimization in HLS. Stephenson et al. described a tool called *Bitwise* that applied an iterative approach to the static range analysis of variables [7]. Bitwise-optimized programs were synthesized to FPGA hardware, yielding area reductions ranging from 15% to 86%, though the benchmarks used in the study had hard-coded or randomly-generated inputs of specific widths (rather than unknown inputs from the user or a file) – possibly easing range analysis. Our work builds upon [7] by integrating bitmask analysis with range analysis to yield further hardware reductions.

Bitwidth-aware HLS was also discussed in [8], where the authors used Stephenson's analysis (above) to determine bitwidths and altered HLS algorithms to map operations of similar bitwidths to shared functional units in FPGA hardware. The idea was to reduce the size of a functional unit to the maximum width needed by any operation bound to the unit. Later work, however, has shown there to be little value in functional unit sharing in FPGA technology [9], owing to the high costs of implementing multiplexers in FPGAs. Other works include [10], which applied range analysis for bitwidth minimization in DSP circuits. [12] optimized bitwidth such that a limited amount of error is permitted in the generated hardware. Our work, however, is focussed on bit-level optimizations that maintain functional correctness. Constantinides et al. [11] examined HLS optimizations for the case of pre-specified (known) wordlengths. A recent work [14] explored how to efficiently recognize and implement bitwise operations in HLS, reducing area and improving speed. Our work is orthogonal to [14], in that we focus on eliminating unnecessary bits within variables. It would be interesting in the future to evaluate the combined benefits of both approaches.

Some HLS efforts, such as IBM's Liquid Metal project, use customized languages that permit the explicit specification of variable bitwidths [13], e.g. to declare a 10-bit integer variable. While our work is concerned with the use of a standard language, C, the proposed techniques are appliable to such custom languages, though may potentially yield smaller benefits.

C. High-Level Synthesis

We use LegUp [3], an open high-level synthesis tool that targets FPGAs [3]. The tool is implemented as back-end passes within the LLVM compiler framework [2]. In LLVM, the program being compiled is represented in an *intermediate representation* (IR), which is, essentially, machine-independent assembly code. The IR comprises simple operations such as add, multiply, shift, logical, branch, etc. After standard compiler optimization passes are run within LLVM (e.g. dead-code elimination, common sub-expression elimination), the optimized IR is passed to the HLS tool, which performs resource allocation, scheduling, binding, RTL generation. The Verilog RTL produced by HLS can be compiled by standard FPGA synthesis tools.

Our bitmask analysis pass (described below) is implemented within LLVM and executes on the IR just prior to HLS.

III. Bitwidth Reduction Approach

Our bitwidth reduction approach uses range and bitmask-based techniques in tandem. We first run a range-based bitwidth reduction pass using the implementation described in [1][1]. The resulting ranges are used to set *initial* bitmasks for each instruction in the control-dataflow graph (CDFG), which is accessible within the LLVM compiler. Our bitmask-based bitwidth reduction approach generates a bitmask representation for each instruction, where each bit of the bitmask can be an unknown (`?`), a sign bit (`S`), or a known value (logic-`0` or logic-`1`). For example, an 8-bit bitmask that represents a range of values from -2 to 2 is represented as `SSSSS???`. However, if it is known that the value can only be -2 or 2, the bitmask is `SSSSS?10`. As this example shows, using a bitmask-based approach rather than a range-based approach allows for the elimination of arbitrary bits, rather than only the elimination of most-significant bits.

Our approach repeatedly traverses the CDFG, first in the forward direction, then in the backward direction, and updates instruction bitmasks throughout these traversals. The forward and backward traversals of the CDFG continue until no bitmask updates are made (no further bitwidth reductions are possible). The forward and backward traversals use forward and backward *transit functions*, which we developed for each instruction type in the LLVM IR. The forward transit functions dictate how the output bitmask of an instruction can be constrained based on the bitmasks of the operands (inputs) of the instruction. The backward transit functions (used in the backward CDFG traversals) define the opposite.

Propagating information in the forward direction is a matter of applying the forward transit function of an instruction to the bitmasks of its operands. Propagating bitmask information in the backward direction involves visiting each instruction and,

[1]Note that we modified the handling of several instruction types from [1] so that they provide a tighter bound on min and max variable values.

for each of its users (instructions that use it as an operand), applying the backward transit function. In this case, the bitmask of the instruction is set to the intersection of its current bitmask and the union of each one of its users (passed through the appropriate backward transit function). In other words, the bitmask of the instruction is set to the most-restrictive of: 1) its current bitmask, or 2) a bitmask computed to accommodate all of its users. The forward and backward transit functions for arithmetic and shift instruction types are described in Tables I and II.

Examples in the tables and examples found in the rest of this paper use notation similar to LLVM's intermediate representation. For example, line 4 in Fig. 2: `%3 = and i4 %2, 1; f0:000?` indicates that a variable called `%3` is assigned the result of a 4-bit `AND` between a variable called `%2` and the constant logic-`1`. The text to the right of the semicolon in each line is a comment, which is used to indicate the bitmask value of the instruction output. In this case, this instruction has the bitmask value `000?` on the first forward traversal pass (`f0`).

In Tables I and II, LSB (least significant bit) refers to the index of the right-most non-zero bit in a variable, MSB (most-significant bit) refers to the index of the left-most non-zero bit, and *rSigned* refers to the index of the right-most sign bit, which is identical to all bits to its left. In all cases, *rSigned* is less than or equal to MSB. For example, the number -2 (`SSSSS110`) has $MSB = 7$, $LSB = 1$, and $rSigned = 2$. Additionally, in the tables, $op0$ refers to the first operand of an instruction, $op1$ refers to the second operand of an instruction, and out refers to the output of the instruction. For a given variable, bits left of the MSB are *guaranteed* to be `0`, as are bits to the right of the LSB, and it is precisely this *width reduction* in the variable that permits reductions in synthesized hardware.

Logical operations (`AND`, `OR`, and `XOR`) are not included in the table since they are relatively straightforward to handle: they perform their respective bitwise logical functions in the forward direction and in the backward direction, the input bitwidths are truncated to the output bitwidth. Any bitwise logical operation between a sign bit and an unknown will result in an unknown (e.g. $S \vee ? = ?$). In all other cases, bitwise operations behave as expected (e.g. $0 \wedge ? = 0, S \vee 0 = S, ? \oplus 0 = ?$). In the backward direction, the operands of an instruction are truncated to the bitwidth of the output. An exception is the case of `XOR`, where an input is only truncated to the size of the output if *both* inputs can be truncated (neither has a different user that requires higher order bits). This is because truncating only one of the inputs to an `XOR` may change the values of bits left of $MSB(out)$. In the example below, the MSB of the output of the `XOR` instruction (%4) is bit 2 because bit 3 (logic-`0`) is the `XOR` of two logic-`1` values (bits 3 of %2 and %3). When backward propagating through this `XOR` instruction, we would need to truncate both or none of the inputs in order to avoid altering the value of bit 3 in the output (logic-`0`). Because one of the inputs has another user (%5 uses %2) and cannot be truncated, %3 also cannot be truncated.

The LLVM IR also has special instructions for zero extension, sign extension and truncation of a single operand. Zero extension is handled as expected, by padding the input bitmask with logic-`0` values in the forward direction (e.g. zero extending `????` to 8 bits = `0000????`), and setting the input bitmask to a truncated version of the output bitmask in the backward direction. Sign extension is handled similarly, by padding the bitmask with `S` values in the forward direction (e.g. sign extending `????` to 8 bits = `SSSS????`), and truncating in the backward direction for the majority of cases[2]. The forward transit function for LLVM's truncation instruction is the same as the backward transit function of zero extension (except with $op0$ and out swapped), and the backward transit function for truncation is the same as the forward transit function for zero extension (except with $op0$ and out swapped).

```
%0 = load i4 %address1 ; fo:???? ; b0:????
%1 = load i4 %address2 ; fo:???? ; b0:????
%2 = or i4 %0, 4 ; f0:1??? ; b0:1???
%3 = or i4 %1, 4 ; f1:1??? ; bo:1???
%4 = xor i4 %2, %3; fo:0???
%5 = and i4 %2, 15; f0:1???
```

Fig. 1. Example of backward propagation through an `XOR`.

```
1: %0 = load i4 %address0; f0:???? b0: 00??
2: %1 = load i4 %address1; f0:???? b0: 00??
3: %2 = or i4 %0, %1; f0:???? b0: 00??
4: %3 = and i4 %2, 1; f0:000?
5: %4 = and i4 %2, 2; f0:00?0
```

Fig. 2. Example of forward and backward bitmask propagation.

An example of forward and backward CDFG propagation is provided for the code snippet in Fig. 2. The bitmasks of the instruction outputs are updated as follows:

1. Lines 1 and 2: Variables `%0` and `%1` are loaded with 4-bit values from memory, so their bitmasks are unknown (`????`).
2. Line 3: The forward transit function for `OR` is assigned to `%2` for operand variables `%0` and `%1`, with the result being `????`.
3. Line 4: The forward transit function for `AND` is assigned to `%3` for the operands `%2` and a constant `1` (i.e. `???? & 0001`) with the result being `000?`.
4. Line 5: The forward transit function for `AND` is assigned to `%4` for the operands `%2` and the constant `2` (i.e. `???? & 0010`) with the result being `00?0`.
5. Backward propagation begins with the instruction on line 3, which has users `%3` (line 4) and `%4` (line 5). Applying the backward transit function for the logical operation `OR` to `%3` yields `000?` and to `%4` yields `00?0`. Setting the bitmask of `%2` to the intersection of its current value (`????`) and the union of its users (`000?` $\cup$ `00?0`) yields `00??`.
6. Instructions on lines 1 and 2 each have one user (%2) and are assigned its bitmask passed through the backward transit function for `OR`, resulting in `00??`.

Through this forward and backward analysis, we are able to prove that for this example, when the instructions are synthesized in hardware, only 2 bits are necessary to represent the

[2]The one exception is in a situation such as the following: given a sign extension from 4 bits to 8 bits, backward propagating `SSS?0000` would yield `?000`, not `0000`. This is because all bits left of $MSB(op0)$ in the output derive their value from the bit at $MSB(op0)$, so this bit must not be eliminated.

TABLE I
FORWARD AND BACKWARD TRANSIT FUNCTIONS FOR SHIFT INSTRUCTIONS.

	Forward	**Backward**

Logical shift right
$out = op0 >> op1$
	If $op1$ (the number of bits to shift) is a constant value, then the bitmask of $op0$ can just be shifted by that amount ($bitmask(out)$ = $bitmask(op0)$ $>>$ $op1$). The bits left of $MSB(op0)$ in out are filled with logic-`0`. If shifting by a bitmask with some unknowns, then the bitmask of $op0$ can not simply be shifted right by a certain number of bits. Instead, the minimum and maximum shift values ($op1_{min}$ and $op1_{max}$) are used to conservatively determine the bitmask of out. The MSB and rSigned of $op0$ are shifted right by the minimum possible shift value ($rSigned(out) = rSigned(op0) - op1_{min}$, $MSB(out) = MSB(op0) - op1_{min}$) and the LSB of $op0$ is shifted right by the maximum possible shift value ($LSB(out) = LSB(op0) - op1_{max}$). $Bitmask(out)$ is set to `?` from $LSB(out)$ to $rSigned(out)$ and `S` from $rSigned(out)$ to $MSB(out)$. In the following example, the max value of $op1$ (`%1`) is 1, and the min value is 0. $Bitmask(out)$ of `%3` is thus set to `??00` since the LSB of `?000` is shifted to the right by the max (1), and the MSB of `?000` is shifted by min (0).
	
	`%0 = load i4 %address ; fo:????`
	`%1 = and i4 %0, 1; fo:000?`
	`%2 = and i4 %0, 8; f0:?000`
	`%3 = lshr i4 %2, %1; fo:??00`
		Similar to the backward transit function though to obtain the value of $op0$, the bitmask of out is essentially shifted left by the value in $op1$. When $op1$ is not a constant, $MSB(op0) = MSB(out) + op1_{max}$, $rSigned(op0) = rSigned(out) + op1_{max}$ and $LSB(op0) = LSB(out) + op1_{min}$.

Arithmetic shift right $out = op0 >> op1$	If $MSB(op0) < bitwidth(op0) - 1$, then all values shifted in will necessarily be logic-`0`, so the shift can be treated as a logical shift, as described above. Otherwise, it isn't known what will be shifted in so $MSB(out) = MSB(op0)$. In all other ways, the forward transit function for arithmetic shift right is identical to the forward transit function for logical shift right.	Identical to the backward transit function for logical shift right.
Shift left $out = op0 << op1$	Similar to the backward transit function for shift right, except with $op0$ and out swapped.	Similar to the forward transit function for logical shift right, except with $op0$ and out swapped.

TABLE II
FORWARD AND BACKWARD TRANSIT FUNCTIONS FOR ARITHMETIC INSTRUCTIONS.

	Forward	**Backward**
Add $out = op0 + op1$	To compute the bitmask, addition is performed bit-by-bit from right to left. Each bit is assigned a value based on value of same-index bits in $op0$ and $op1$, and the value of carry in from previous bits to the right. By computing the addition in this way, the output bitmask does not always need to use one extra bit to represent the result. For example, `00?? + 10?? = 1???`.	Input is truncated to the size of the output (e.g if the bitmask of out is `????` and the bitmask of $op0$ was `??10?`, then bitmask of $op0$ would be truncated to `?10?` after the application of the backward transit function).
Subtract $out = op0 - op1$	Very similar to addition, except with $op1$ set to its two's-complement.	Input is truncated to the size of the output.
Multiply $out = op0 \times op1$	If both operands are unsigned, $MSB(out) = MSB(op0) + MSB(op1) + 1$, otherwise, if one or more of the operands are signed, $MSB(out) = bitwidth(out) - 1$. In all cases, $rSigned(out) = rSigned(op0) + rSigned(op1) + 1$, and $LSB(out) = max[LSB(op0), LSB(op1)]$. Output bitmask is set to `?` from $LSB(out)$ to $rSigned(out)$ and `S` from $rSigned(out)$ to $MSB(out)$. The only case in which some bits between $LSB(out)$ and $rSigned(out)$ may be set to known values is if both operands have least-significant values that are known. For example: `?101` $\times$ `?011 = ?????111`.	When one of the inputs of the multiplication has $LSB > 0$, the left-most bits in the other input may never impact the final result. For example, given the multiplication of two `char` datatypes with the bitmask of $op0$ = `????????` and bitmask of $op1$ = `??????00`, with an 8-bit output bitmask of `??????00`, the two left-most bits of $op0$ will never affect the final product. When backward propagating, we can use this property to reduce the bitmask of $op0$ from `????????` to `??????`. Keeping track of least-significant bits can help eliminate most-significant bits.
Division $out = op0 \div op1$	Let T be the largest power of 2 less than or equal to $\lvert op1\rvert_{min}$. If both operands are unsigned, then $MSB(out) = MSB(op0) - T$, based on the relationship between division and right shift. Otherwise, if the quotient may be negative, $MSB(out) = bitwidth(out) - 1$. In all cases, $rSigned(out) = rSigned(op0) - T$ and $LSB(out)$ is set to 0. The output bitmask is set to `?` from $LSB(out)$ to $rSigned(out)$ and is set to `S` from $rSigned(out)$ to $MSB(out)$.	Input is unchanged.
Remainder $out = op0 \% op1$	$MSB(out) = MSB(op1)$, $rSigned(out) = rSigned(op1)$, $LSB(out) = 0$.	Input is unchanged.

variables, rather than 4 bits which would otherwise be necessary. We observed that 3-4 forward/backward traversals of the CDFG are normally sufficient for bitmasks to converge for all instructions.

IV. Experimental Study

We evaluate the proposed optimizations in two ways: First, we gauge their effectiveness in reducing variable bitwidths at the pre-FPGA-synthesis stage. For this, we sum the bitwidths of all arithmetic, logical, shift, select and `phi` instructions in the optimized LLVM IR of the program being synthesized by the HLS tool. Second, we synthesize the HLS-generated bit-level-optimized circuits to the Altera Cyclone II FPGA and report the area (look-up tables (LUTs) and registers) as well as speed ($FMax$).

We consider six different flows, allowing us to evaluate the relative merits of range analysis, bitmask analysis, and the use of dynamic (run-time) data vs. static (compile-time) data. The flows are as follows: 1) baseline (no bit-level optimization in HLS), 2) bitmask analysis only (static), 3) range analysis (static), 4) range analysis and bitmask analysis (static), 5) range analysis (dynamic), 6) range analysis and bitmask analysis (dynamic). The "static" flows rely solely on compile-time information – ranges/bitmasks are inferred from propagation of constants in the code. The "dynamic" flows rely on run-time-extracted variable ranges, meaning that circuits optimized with such data are *only* guaranteed to work for the particular program inputs used to produce the ranges. The use of dynamic data for bitwidth reduction is useful for applications with predictable input data. Additionally, the dynamic flows represent a lower bound on the variable bitwidths, which is useful when analyzing the quality of static analysis. To gather ranges dynamically, we altered the LLVM interpreter, `lli`, to track variable ranges during its execution of a program's IR.

For benchmarks, we use eight of the 12 benchmarks in the CHStone high-level synthesis suite [5] (for the remaining four, the range analysis code [1] produced errors related to the handling of 64-bit arithmetic operations). We also use two benchmarks included with the HLS tool distribution: `fft` (fast-Fourier transform) and `dhrystone` (an integer benchmark). For all experiments, the `-O3` optimization level was used with LLVM. All of the benchmarks include input data and *golden* output data. For all flows, we used the golden output to verify the correct functionality of the bit-level optimized circuits using ModelSim functional simulation. The run-time of each flow was under one minute, which is significantly less than the run-time of Altera's Quartus II tool that follows high-level synthesis, so we do not include run-time information in this paper.

Table III gives the bitwidth reduction results for the six experimental flows described above. Numbers in the table represent the summed bitwidth across all operators in each benchmark, for a given flow. The geometric mean bitwidth across all benchmarks is given at the bottom of each column. The last row of the table gives ratios comparing each flow to the baseline case (no bit-level optimization in HLS).

Walking the columns of Table III from left to right, we begin with the "Bitmask" column and observe that the techniques described in Section III provide a 25% reduction in total bitwidth, on average, compared with the baseline of no bit-level optimization. The "Range" column shows that the static range analysis in [1] provides a 28% reduction in bitwidth. Combined, the bitmask and range analysis ("Bitmask+Range") approaches reduce total bitwidth by 35% – a significant reduction considering that these analyses rely only on static compile-time data. The combined approach provides larger reductions than either approach in isolation, indicating that, at least to some extent, bitmask analysis and range analysis offer complementary benefits. We discussed in Sec. III how bitmask analysis can reduce bitwidths over and above what is possible with only range analysis, in some cases. It is also true that range analysis reduces bitwidths over and above what is possible only with bitmask analysis. For example, the range analysis implementation that we applied does loop analysis to calculate tight bounds on loop counters, which is not done in our bitmask analysis.

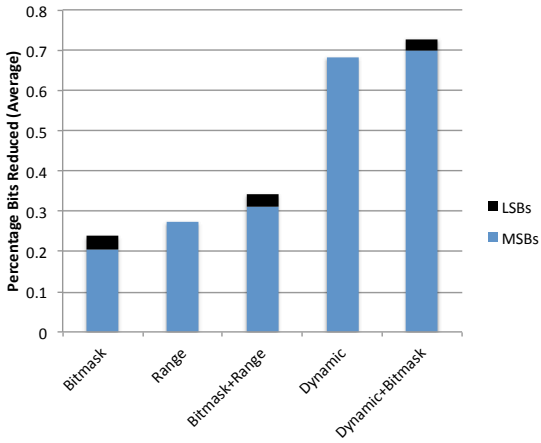

Fig. 3. Most-significant and least-significant bits reduced (average).

The last two columns of Table III use ranges extracted from run-time profiling rather than static analysis. A 71% average reduction in bitwidth is achieved using dynamically-gathered ranges. The right-most column shows that propagating such ranges through bitmask analysis provides additional bitwidth reductions: 75% on average vs. the baseline. There is thus tremendous value to be gained in letting an HLS tool know about the ranges exercised in program execution. We believe that a software engineer frequently has some knowledge about variable ranges (e.g. he/she may know a particular variable never exceeds 1,000). HLS tools that accept standard programming languages as input should permit such ranges to be supplied to the tool, using pragmas specified in comments.

As range analysis determines the max and min values of variables, it can only affect the most-significant bits (MSBs) of a variable's bit-level representation. Bitmask analysis, on the other hand, can eliminate both most-significant and least-significant bits (LSBs). Fig. 3 shows percentage reductions (vs. the baseline) in MSBs and LSBs, averaged across all benchmarks for the bit-level optimization flows. We used arithmetic average for the data in Fig. 3, as for some circuits, no LSBs were eliminated, so we could not compute geometric mean reductions. Fig. 3 shows that the majority of bitwidth reduction is due to elimination of MSBs. LSB reductions are in the single-digit percentage range and are only present in the

TABLE III
BITWIDTH REDUCTION RESULTS.

Benchmark	Baseline	Bitmask	Range	Bitmask+Range	Dynamic	Dynamic+Bitmask
dhrystone	2934	2426	1801	1712	389	385
fft	1850	1189	1367	1171	764	590
adpcm	15020	11730	11324	9906	4462	3645
aes	10704	9253	9340	8909	1929	1698
blowfish	10074	6223	6169	5549	5130	4568
gsm	14462	10354	10399	9444	6229	5301
jpeg	22640	20869	18527	17944	4986	4512
mips	2868	1745	2115	1646	768	560
motion	3576	2862	2043	1900	655	567
sha	5752	4805	4679	4365	3053	2812
Geomean:	**6653**	**5017**	**4782**	**4321**	**1904**	**1636**
Ratio:	**1.00**	**0.75**	**0.72**	**0.65**	**0.29**	**0.25**

TABLE IV
CYCLONE II IMPLEMENTATION RESULTS

	LUTs			Registers			FMax (MHz)		
Benchmark	**Baseline**	**Bitmask+ Range**	**Dynamic+ Bitmask**	**Baseline**	**Bitmask+ Range**	**Dynamic+ Bitmask**	**Baseline**	**Bitmask+ Range**	**Dynamic+ Bitmask**
dhrystone	5244	4120	3738	3575	3131	2438	117.94	114.09	115.96
fft	2046	2043	1880	1048	1028	746	92.89	91.3	91.3
adpcm	21695	18631	7036	11039	10020	4291	55.46	56.04	56.16
aes	19784	15792	8871	11470	9162	4066	49.38	49.82	46.47
blowfish	10621	10590	10296	7412	7353	7040	75.41	73.61	71.62
gsm	9787	9645	7807	6612	6487	5029	33.2	32.39	32.98
jpeg	33618	31083	22057	20688	19388	11885	18.02	17.53	19.15
mips	3384	3358	2116	1620	1590	999	98.8	95.56	110.22
motion	4054	4020	2946	2526	2526	1656	112.18	111.83	125.85
sha	10686	8243	7612	7779	5838	5371	99.42	106.68	109.42
Geomean:	**8655**	**7838**	**5711**	**5230**	**4794**	**3217**	**65.7**	**65.2**	**67.3**
Ratio:	**1.00**	**0.91**	**0.66**	**1.00**	**0.92**	**0.62**	**1.00**	**0.99**	**1.02**

flows that include bitmask analysis.

Turning to the FPGA implementation results, the left side of Table IV shows the number LUTs in the Cyclone II circuit implementations for three bit-level optimization flows: baseline (no bit-level optimizations in HLS), combined static bitmask and range analysis-based optimization, and combined dynamic bitmask and range analysis-based optimization. Observe that the combined static approach reduces LUT usage by 9%, on average, relative to the baseline. This is a significant result, as Quartus II is a commercial tool that incorporates its own static bit-level optimization techniques applied during RTL synthesis. The 9% area reductions are on top of Quartus II's optimizations (the impact of Quartus II's optimizations are reflected in the baseline numbers in the table). In some circuits, for example `sha` and `dhrystone`, area was reduced by more than 20%. While the techniques implemented within Quartus are proprietary and are not disclosed publicly, the results demonstrate that some of the proposed bit-level analyses/optimizations implemented in LLVM at the HLS stage are complementary to those implemented within Quartus.

With dynamic ranges, LUT counts are reducted by 34%, on average, echoing the significant bitwidth reductions presented in Table III for the dynamic flows. The middle columns of the table give analogous data for register usage. The area reductions align closely with those observed for LUTs.

We also analyzed the post-routing speed performance ($FMax$) of the circuits across the three flows in Table IV. The $FMax$ data is presented in the three right-most columns of Table IV. For all runs of Quartus, the target $FMax$ was set to 1Ghz. Observe that with static analysis, the speed performance of circuits is roughly flat vs. the baseline (a 0.8% reduction in speed is seen, which we believe lie in the noise range). With dynamic analysis, circuit speed is improved modestly, by 2.4%, on average. Though it is encouraging that the area reductions do not come at the cost of degraded speed performance, we had, in fact, expected more significant speed improvements from bit-level optimization. However, it is possible that many of the bitwidth reductions were not on the critical paths of some circuits.

V. CONCLUSIONS

With the advent of high-level synthesis, creating efficient hardware from a software-like generic description is paramount. Although modern RTL synthesis tools eliminate redundant/unused logic, this work shows that there is an opportunity to further reduce area by reducing instruction bitwidths. Specifically, this work shows that through the use of both range and bitmask-based bitwidth-reduction techniques, circuit area can be reduced by an average of 9% above the savings already provided by using Altera's Quartus II synthesis framework. Furthermore, we showed that by coupling our bitmask-

based bitwidth reduction technique with dynamic run-time profiling data used to generate ranges for variables, area can be reduced by an average of ∼34%. Regardless of the type of range-analysis used (dynamic or static), we showed that using our bitmask-based approach further reduces variable bitwidths – it is complementary to range analysis (at least to some extent).

Future work includes further improvements to the range-based bitwidth reduction techniques, which should be extended to infer more information about the values loaded from memory, potentially by using LLVM's built-in alias analysis abilities. Additionally, more careful consideration of loop counter ranges needs to be implemented, in a manner similar to [7]. Finally, there is an interesting opportunity to use the hardware minimized using profile-based ranges within a processor/accelerator system. In such a system, the accelerator would only be guaranteed to work if variables ranges remained within those observed during dynamic profiling. If ranges fell outside of those profiled, the application would fall back to a software version of the code running on the processor.

REFERENCES

[1] V. Campos, R. Rodrigues, I. Costa, and F. Pereira, "Speed and precision in range analysis," in *Brazilian Symposium on Programming Languages*. SBC, 2012.

[2] *The LLVM Compiler Infrastructure Project (http://www.llvm.org)*, LLVM, 2010.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *ACM/SIGDA FPGA*, 2011, pp. 33–36.

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

[5] 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, no. 0, pp. 242 – 254, 2009.

[6] Z. Su and D. Wagner, "A class of polynomially solvable range constraints for interval analysis without widenings," *Theoretical Computer Science*, vol. 345, no. 1, pp. 122–138, Nov. 2005.

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

[8] J. Cong, Y. Fan, G. Han, Y. Lin, J. Xu, Z. Zhang, and X. Cheng, "Bitwidth-aware scheduling and binding in high-level synthesis," in *IEEE/ACM ASP-DAC*, 2005, pp. 856–861.

[9] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," in *ACM FPGA*, 2012, pp. 111–114.

[10] B. Le Gal, C. Andriamisainat, and E. Casseaut, "Bit-width aware high-level synthesis for digital signal processing systems," in *IEEE Int'l SOC Conference*, 2006, pp. 175–178.

[11] G. Constantinides, P. Cheung, and W. Luk, "Heuristic datapath allocation for multiple wordlength systems," in *IEEE/ACM DATE*, 2001, 791–796.

[12] A. Ahmadi and M. Zwolinski, "Symbolic noise analysis approach to computational hardware optimization," in *IEEE/ACM DAC*, 2008, pp. 391–396.

[13] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-compatible and synthesizable language for heterogeneous architectures," in *OOPSLA '10*, 2010, pp. 89–108.

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