The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs

Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, Jason Anderson
Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada
Email: legup@eecg.toronto.edu

Abstract—We consider the impact of compiler optimizations on the quality of high-level synthesis (HLS)-generated FPGA hardware. Using a HLS tool implemented within the state-of-the-art LLVM [1] compiler, we study the effect of compiler optimizations on the hardware metrics of circuit area, execution cycles, Fmax, and wall-clock time. We evaluate 56 different compiler optimizations implemented within LLVM and show that some optimizations significantly affect hardware quality. Moreover, we show that hardware quality is also affected by the order in which optimizations are applied. We then present a new HLS-directed approach to compiler optimizations, wherein we execute *partial* HLS and profiling at intermittent points in the optimization process and use the results to judiciously undo the impact of optimization passes predicted to be damaging to the generated hardware quality. Results show that our approach produces circuits with 16% better speed performance, on average, versus using the standard `-O3` optimization level.

I. Introduction

High-level synthesis (HLS) raises the level of abstraction for hardware design by allowing software programs written in a standard language to be automatically compiled to hardware. First proposed in the 1980s, HLS has received renewed interest in recent years, notably as a design methodology for field-programmable gate arrays (FPGAs). While FPGA circuit design has historically been the realm of hardware engineers, HLS offers a path towards making FPGA technology accessible to software engineers, where the focus is on using FPGAs to implement accelerators that perform computations with higher throughput and energy efficiency relative to standard processors. We believe, in fact, that FPGAs (rather than ASICs) will be the vehicle through which HLS enters the mainstream of IC design, owing their reconfigurable nature. With custom ASICs, the silicon area gap between human-designed and HLS-generated RTL leads directly to (potentially) unacceptably higher IC manufacturing costs, whereas with FPGAs, this is not the case, as long as the generated hardware fits within the available target device.

Modern HLS tools are implemented within software compiler frameworks. For example, among widely-used frameworks, Xilinx's AutoPilot tool [2], ROCCC from UC Riverside [3], and LegUp from the University of Toronto [4] are implemented within the LLVM compiler [1], and GAUT [5] from the Université de Bretange Sud is implemented within GCC. Consequently, the programs that are input to such tools are subjected to standard compiler optimizations applied before HLS commences. Compilers perform their optimizations in *passes*, where each pass is responsible for a specific code transformation, for example, dead-code elimination, constant propagation, loop unrolling, or loop rotation. LLVM contains implementations for 56 such optimization (transform) passes

that may alter the program, as well as many other passes that analyze the code to provide decision-making data for transform passes (see: `http://llvm.org/docs/Passes.html`). The familiar command-line optimization levels (e.g. `-O3`) correspond to a particular set and sequence of compiler passes. The compiler passes within LLVM were intended to optimize software programs that run on a microprocessor. Their impact on HLS-generated hardware is not well-studied nor is the manner in which they should be applied to best optimize hardware quality. It is precisely these issues that we explore in this paper.

We study the impact of compiler passes using the open-source LegUp HLS tool. We target the Altera Cyclone II FPGA [6] and assess hardware quality using several metrics: area, *FMax*, execution cycles, and wall-clock time. We conduct a wide range of experiments to explore: 1) the impact of each LLVM pass in isolation, 2) the interdependency between different passes, and 3) the impact of pass ordering. We present a detailed analysis for several passes demonstrated to have a significant hardware impact. We show that the particular set of passes applied can have a significant impact on hardware quality – variance in the range of over $\pm 10\%$ is common. We also show that a given pass may improve some circuits and not others; and likewise, a pass may improve hardware along one axis (e.g. area), while at the same time degrade hardware along a second axis (e.g. speed).

Given that the impact of a particular pass or set of passes is program dependant, we propose a HLS-directed approach to the application of compiler optimization passes. At a high level, our approach works as follows: we iteratively apply one or more passes and then "score" the result by invoking partial HLS coupled with rapid profiling (in software). Transformations made by passes deemed to positively impact hardware are accepted. Conversely, we *undo* the transformations of passes that we predict to be damaging to hardware quality. Results show our optimization strategy consistently outperforms the standard `-O3` level in terms of hardware speed performance. While a prior work examined a limited number of code transformations in the HLS context [7] and their integration into HLS algorithms, to our knowledge, this is the first published study of a broad collection of different optimizations in a state-of-the-art HLS compiler.

The rest of this paper is organized as follows: Section II provides relevant background and describes related work. Section III presents results that illustrate the impact of LLVM optimization passes on generated hardware quality. In Section IV, we introduce our HLS-directed compiler optimization approach. Experimental results are presented in Section V.

Section VI offers conclusions and suggests future work.

II. BACKGROUND

While the standard compiler optimization levels offer a simple set of choices for a developer, the particular optimizations applied at each level are generally chosen to benefit the run-time of a basket of programs. It is not guaranteed, for example, that for a specific program the -O3 level produces superior results to the -O2 level. This has led the (software) compiler community to consider selecting a particular set of compiler optimization passes on a per-program (or even per-code-segment) basis. Such "adaptive" compiler optimization has been the subject of active research in recent years, with a few examples of highly-cited works being [8], [9], [10]. Broadly speaking, research in the area involves devising heuristics to prune the large optimization space of selecting passes, thereby reducing the number of different passes that need to be applied/attempted. Milepost [11] is a GCC-based optimization approach that uses machine learning to determine the set of passes to apply to a given program, based on a static analysis of its features. It achieved 11% execution time improvement, on average, for the ARC reconfigurable processor on the MiBench program suite[1].

Our work bears similarity to such efforts in the software domain, and represents a step towards adaptive compiler optimization in the HLS *hardware* domain.

The LLVM Framework and LegUp High-Level Synthesis

LLVM is an open-source compiler framework used in both industry and academia. LLVM's front-end, clang, parses the input C source and translates it into LLVM's *intermediate representation* (IR). The IR is essentially machine-independent assembly code in static-single assignment (SSA) form, comprised of simple computational instructions (e.g. add, shift, multiply) and control-flow instructions (e.g. branch). LLVM's opt tool performs a sequence of compiler optimization passes on the program's IR – each such pass directly manipulates the IR, accepting an IR as input and producing a new/optimized IR as output.

The LegUp HLS tool is implemented as back-end passes of LLVM that are invoked after the compiler optimizations. LegUp accepts a program's optimized IR as input and schedules the IR instructions into clock cycles, binds the scheduled instructions to functional units, constructs the corresponding FSM, and writes out RTL Verilog code synthesizable by commercial FPGA RTL synthesis tools. The LegUp scheduler, based on the SDC formulation [12], operates at the basic block level, exploiting the available parallelism between instructions in a basic block. One Verilog module is generated for each function in the program. Results show that LegUp produces solutions of comparable quality to a commercial HLS tool (eXCite [13]) and the interested reader is referred to [4] for more details.

TABLE I
CHSTONE BENCHMARK CHARACTERISTICS.

Benchmark	Lines of C	Class
adpcm	550	media
blowfish	1,255	encyrption
dfadd	441	arithmetic
dfdiv	292	arithmetic
dfmul	270	arithmetic
dfsin	580	arithmetic
gsm	388	media
jpeg	1,073	media
mips	271	processor
motion	602	media
sha	1,969	encryption
Geomean	**565**	

III. IMPACT OF COMPILER OPTIMIZATIONS ON HLS

A. Methodology

In this section, we present an analysis of the impact of compiler optimization passes on HLS-generated hardware. We use C benchmarks from the CHStone high-level synthesis benchmark suite [14], which are chosen from a variety of domains (e.g. multimedia, communications, encryption). Eleven of the 12 CHStone benchmarks were used, as errors were encountered for one benchmark, aes, with certain combinations of optimization passes[2]. The number of lines and category of each benchmark is listed in Table I. Note that each CHStone benchmark has built-in input stimuli and golden outputs, allowing us to execute the benchmark's hardware implementation and verify functional correctness. We synthesize the LegUp-generated Verilog to the Altera Cyclone II FPGA [6] using Quartus II ver. 11.1SP2, applying a 1 GHz clock constraint to each circuit[3]. We used ModelSim to extract the number of cycles needed for the execution of each circuit (cycle latency) with the built-in input stimuli. *FMax*, and area were extracted from Altera post-routing report files. Total execution (wall-clock) time is computed as the # of execution cycles divided by post-routed *FMax*. The CHStone circuits are comprised of compute kernels that are executed several times with different inputs; the wall-clock time is the total time needed for each CHStone benchmark to complete its execution of all inputs. All experiments were conducted on cloud computing system having tens of thousands of cores [15].

B. Analysis of Passes in Isolation

We begin by analyzing LLVM optimization passes in isolation relative to -O0 (no optimization). Fig. 1 shows how each pass affects the number of hardware execution cycles. The horizontal axis lists the names of each pass. The vertical axis represents the geometric mean ratio (over the 11 benchmarks) of cycle latency when a particular pass is used, relative to the -O0 case. Values less than 1 represent reductions in cycle latency relative to the baseline case. Observe that many passes
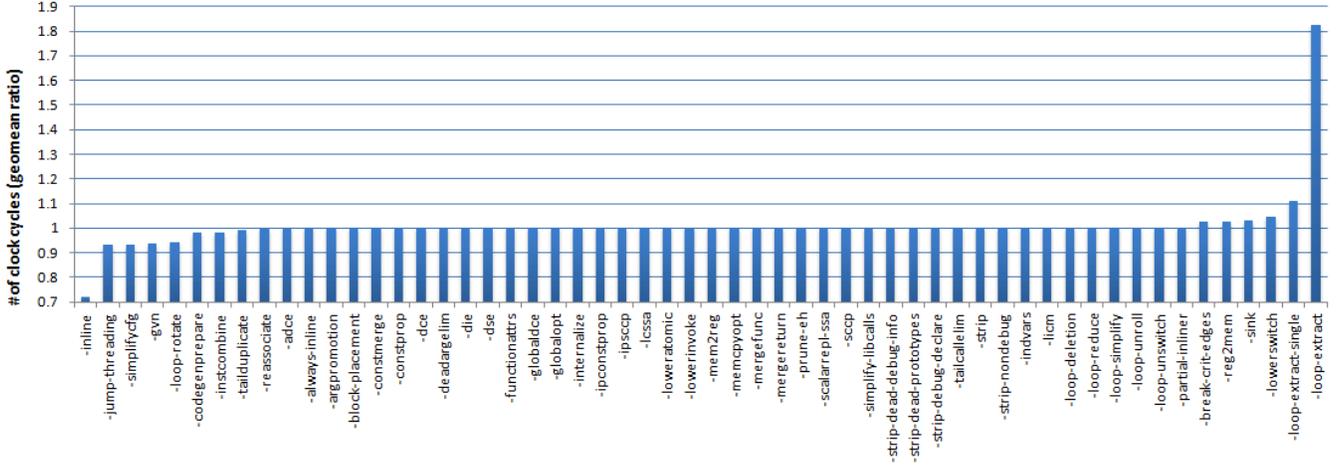
Fig. 1. Impact of individual compiler passes on geomean clock cycle latencies across 11 CHStone benchmarks.

have no impact on cycle latency, at least when applied in isolation. Of the 56 different passes evaluated, only 13 of the passes impacted the geomean cycle latency by more than 1%. Note that some of the passes in Fig. 1 are already in the `-O3` optimization level recipe – our intent here is to assess the impact of each pass in isolation.

While it is outside the scope of this paper to discuss each pass in detail, we did perform an in-depth analysis for passes having a considerable hardware impact. Observe in Fig. 1 that `-loop-extract` and `-loop-extract-single` caused a large increase in the geomean number of execution cycles (values > 1). Both of these optimizations extract loops into separate functions. The LegUp HLS tool does not optimize across function boundaries, and moreover, implements each function as a separate Verilog module, with handshaking between modules occurring when one function calls another. Exlining loops as functions therefore naturally leads to higher numbers of execution cycles. The `-inline` pass has precisely the opposite effect: a large decrease in cycle latency is observed when callees are collapsed (inlined) into callers.

Other passes that improve the hardware include `-loop-rotate` and `-simplifycfg`. The `-loop-rotate` pass changes the position of the loop header within the IR, effectively transforming a loop from a *while* loop into a *do while* loop. We observed this can reduce the number of FSM states for each loop iteration in hardware by eliminating one branch instruction per iteration[4]. The `-simplifycfg` pass simplifies the program's control flow graph by merging basic blocks connected through unconditional branches and by eliminating empty basic blocks, both of which reduce the total number of states in the schedule.

Besides cycle latency (Fig. 1), we also analyzed *FMax*, wall-clock time and area (# of Cyclone II logic elements (LEs)). Complete data for these metrics is omitted for space considerations. Instead, Table II summarizes the impact of individual compiler passes on all hardware metrics. Four mea-

[4]Rotated loops contain a single conditional backward branch at the end of each iteration, rather than one conditional forward branch at the beginning and one unconditional backward branch at the end.

TABLE II
SUMMARY OF THE INDIVIDUAL IMPACT OF 56 LLVM DIFFERENT OPTIMIZATION PASSES ON HLS HARDWARE.

	Clock cycles	FMax	Wall-clock time	LEs
Min	0.72	0.76	0.92	0.93
Max	1.83	1.05	2.24	1.34
St. Dev	0.12	0.04	0.17	0.05
# Impactful Passes	13	16	20	10

surements are provided for each metric. The "min" row gives the minimum geomean value of the metric (across 11 CHStone circuits) for any pass, relative to (-O0) (no optimization). For example, the 0.72 value for the "Clock cycles" metric indicates that one pass caused a 28% decrease in cycle latency, on average, across the benchmarks (see `-inline` in Fig. 1). The "max" row gives the maximum change caused by any pass. The "St. Dev" row gives the standard deviation of change in the geomean, across all 56 passes. The last row of the table shows the number of passes (out of 56) that caused a more than 1% swing in the metric (on average). Table II indicates that *FMax* and the number of LEs (area) are less sensitive to individual compiler passes than cycle latency and wall-clock time (see the standard deviation row). The relative stability in *FMax* is not surprising, as the LegUp HLS tool attempts to meet a user-provided *FMax* target, by potentially inserting more registers in the datapath to meet the specified target (LegUp's default *FMax* target for Cyclone II is 66MHz).

We observed the set of beneficial passes to be highly benchmark dependant. For example, on the metric of wall-clock time, the following 5 passes were found to be individually beneficial for the `adpcm` benchmark: `-block-placement`, `-break-crit-edges`, `-reg2mem`, `-scalarrepl-ssa`, and `-simplify-libcalls`. Whereas, for the `jpeg` benchmark, there were 6 beneficial passes: `-sink`, `-loop-extract-single`, `-block-placement`, `-simplifycfg`, and `-loop-rotate`. Observe that there is little overlap between the two beneficial pass sets.

Recipe	Normalized Hardware Metric			
	Clock cycles	FMax	Wall-clock time	LEs
-O3	1.00	1.00	1.00	1.00
Clock cycle	**0.92**	1.00	0.92	0.92
FMax	1.42	**1.02**	1.39	1.29
Wall-clock time	0.92	1.01	**0.91**	0.93
LEs	1.02	0.99	1.02	**0.91**

C. Customized Passes

To understand the potential for compiler optimization passes to "beat" a standard compiler optimization strategy, -O3, we used the pass analysis data above to create custom "recipes" of passes tailored to each benchmark for each of the four metrics: clock cycle latency, *FMax*, wall-clock time and area (LEs). We created 4 customized recipes for each benchmark, one for each metric, containing only those passes that positively benefited the benchmark on the particular metric in the individual pass analysis. In each custom recipe, we ordered the passes alphabetically (alternative orders are discussed below).

Results for the custom recipes for a representative example of the benchmarks, dfmul, are given in Table III[5]. The left-most column of Table III lists the recipes, beginning with -O3. The remaining columns show the results for each recipe on each hardware metric, normalized to the -O3 results. For example, the "Clock cycles" recipe improves clock cycle latency by 8% vs. -O3, and the *FMax* recipe improved *FMax* by 2%. The wall-clock time recipe improved wall-clock time by 9% – a significant improvement over -O3. Note that for the data in Table III, LLVM's link-time optimization passes were applied after the custom recipes, as well as after -O3. While it is impractical for an end-user to be expected to conduct a similar analysis for each program being compiled, the results serve to illustrate that there is indeed considerable potential to improve upon -O3 results.

D. Impact of Pass Ordering

We also considered the order in which passes are applied and found it to have a significant impact on the hardware quality. Fig. 3 shows the wall-clock time for the jpeg benchmark for all 6! (= 720) orderings of the *same* 6 passes shown to be beneficial in isolation for this benchmark's wall-clock time. A wide range of wall-clock times was observed. The average wall-clock time was 47.6ms (standard deviation 2ms). The minimum time achieved was 41.7ms, whereas the maximum was 53.2ms (nearly 28% higher than the minimum!). The results in Fig. 3 clearly demonstrate that optimization passes are highly interdependent on one another. Thus, to optimize HLS-generated hardware, is it not simply a matter of determining which optimization passes are helpful, but also crucial to determine the order in which they should be applied.

To further underscore the impact of pass ordering, we selected 33 passes, comprised of all those passes that had an impact in isolation (on top of -O0) and also those passes that had an impact when removed from -O3. We considered all

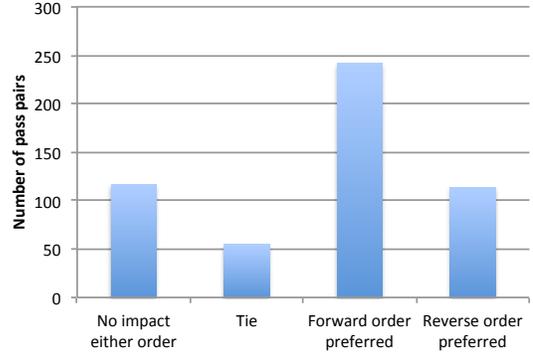[5]Results for other circuits could not be included for space reasons.

Fig. 2. Impact of pass pair forward/reverse ordering on clock cycle latency. Results shown for all $\binom{33}{2} = 528$ combinations of 33 passes.

pairs of passes from this group and evaluated the pairs in both orders, performing synthesis, placement, routing and Model-Sim simulation for all $2 \times \binom{33}{2} = 2 \times 528 = 1056$ combinations for the 11 CHStone benchmarks. Then, looking at the impact of each pass pair on the clock cycle latency of each benchmark, we counted: 1) the number of pass pairs that had no affect in either order on any benchmark; 2) the number of pass pairs for which "forward" (alphabetical) order improved an *equal* number of benchmarks as "reverse" order (a tie); 3) the number of pass pairs for which the forward order improved more benchmarks than the reverse order; and, 4) the number of pass pairs for which the reverse order improved more benchmarks than the forward order. The results of this analysis are shown in Fig. 2. Of the 528 pass pairs, 117 had no impact in either order, and for 55 pairs the orders were tied – forward order helped an equal number of benchmarks as reverse order. For the remaining 356 pairs, one order was better than the other in reducing cycle latency. For 242 pairs, forward order was preferred over reverse order, whereas, for 114 pairs, the reverse order was preferred. We had expected roughly equal numbers of pairs to prefer forward vs. reverse order, nevertheless, the results clearly demonstrate the importance of pass ordering on HLS quality of results for the majority of pass pairs. While it is tractable to evaluate all combinations of pairs of passes, it is computationally intractable to investigate all orderings of larger numbers of passes.

From the analysis of passes in isolation, we also generated a general benchmark-agnostic recipe containing only those passes which showed a benefit for a *majority* of benchmarks when applied in isolation. On the average, the recipe performed worse than -O3, owing to notion that some passes depend on other passes to show any impact. For example, there are passes which showed no benefit whatsoever for a benchmark when applied in isolation, yet showed a benefit when applied after certain other passes. Clearly, the -O3 recipe includes some of such passes which do not affect results in isolation.

Given our experience with customized recipes and the observation that the compiler passes beneficial to each benchmark are both benchmark dependent and order dependent, we felt it would be difficult to devise a single recipe of passes that would benefit *all* circuits. We therefore opted to explore
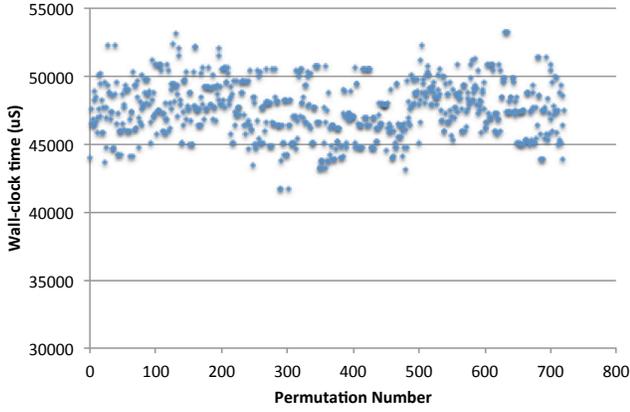
Fig. 3. Wall-clock time for `jpeg` benchmark for all permutations of six optimization passes.

Benchmark	Simulation Time (s)	
	PR	MS
adpcm	1.8	37
blowfish	1.4	99
dfadd	0.4	2
dfdiv	0.5	2
dfmul	0.3	2
dfsin	1.3	27
gsm	1.2	5
jpeg	5.1	3,425
mips	0.4	2
motion	0.3	3
sha	0.7	84
Geomean	0.8	15
Ratio	1.0	20

a more adaptive feedback-based pass recipe approach that automatically determines a good recipe of passes for a given benchmark without any user intervention, as described in the next section.

IV. HLS-DIRECTED COMPILER OPTIMIZATION

Algorithm 1 shows the top-level flow of our scheme. The input to the algorithm is the program's unoptimized IR, as well as an ordered list of candidate optimization passes, P, which we refer to as the *pass pool*. Within a while loop (line 4), we iteratively choose a pass p from the pass pool (line 5), execute it (apply it to the IR) (line 6), and then estimate whether p will be beneficial or detrimental to the HLS-generated hardware (line 7). We use total hardware execution cycles as the cost metric, as it is correlated with wall-clock time and can be determined rapidly (see below). If p is deemed beneficial (line 8), it is accepted, and its effect on the IR is left intact (lines 9-11). Otherwise, p is rejected and the IR is rolled back to the state prior to p being applied. Once we come to the end of the pass pool, we start again from the beginning, and attempt to re-apply passes. The process of selecting passes from the pool and judiciously applying them continues until a stopping criteria is met (also discussed below). Note that while we focus on circuit speed performance in this work, future work may consider the automatic generation of pass recipes that optimize circuit area or power.

We devised an approach to determine the number of hardware execution cycles for a given IR without requiring time-consuming logic simulation with ModelSim. Our approach is based on the observation that the total number of hardware cycles can be determined if two criteria are known for each basic block[6]: 1) the number of times it is executed, and 2) the number of clock cycles it needs to execute. Specifically,

$$CycleCount(IR) = \sum_{b \,\in\, BB(IR)} Execs(b) \cdot SchedLen(b) \quad (1)$$

where $BB(IR)$ is the set of basic blocks in the IR, $Execs(b)$ is the number of times basic block b is executed, and

---

[6]A basic block has a single entry and exit point.

$SchedLen(b)$ is the schedule length of b. $Execs(b)$ can be determined by profiling the execution of the IR in *software*[7] – hardware simulation is not required. $SchedLen(b)$ can be determined by executing HLS up to the scheduling step. Thus, both criteria can be computed rapidly for each basic block, providing an accurate picture of the post-HLS cycle latency for an IR. Note that while the profiling step may be deemed as costly from the run-time angle in a software compilation flow, the time consumed is very small compared to ModelSim simulation of the Verilog RTL. Table IV compares the run-time required by our approach and ModelSim for each of the CHStone benchmarks. On average, our approach extracts cycle latencies 20× faster than ModelSim.

The other tunable aspects of Algorithm 1 include the stopping criteria of the while loop (discussed below), the *Apply* function that executes the selected pass p on the best IR seen so far (also discussed below), and the composition of the pass pool P. For P, we use 41 of the 56 LLVM passes, namely, we include 1) all passes that showed any impact when applied in isolation, 2) passes that showed any impact when we removed them from -O3, and 3) passes not in -O3 and that showed no impact in isolation (as these might show an impact when combined with other passes).

We implemented and evaluated three variants of Algorithm 1 offering different run-time/quality trade-offs, which we refer to as the *iteration method*, the *insertion method*, and the *insertion-3 method*. The first two variants differ from one another in their implementation of the *Apply* function, which applies the chosen pass p to the best IR found so far. In the iteration method, we first sort all passes based on the pairs analysis results (see Section III) so that the pairwise pass ordering favors reductions in clock cycle latency. Passes that showed no impact in isolation (or through the pairs analysis) were added to the end of the list. We apply the passes in order, in particular, we apply the selected pass, p, at the *end* of the recipe that produces the best IR so far. Hence, the iteration

---

[7]This is possible because the CHStone benchmarks contain inputs within the programs themselves, and can therefore be executed without user intervention. For general programs, one would need to execute them with representative inputs.

TABLE V
SPEED PERFORMANCE RESULTS (IT: ITERATION METHOD, IN: INSERTION METHOD, IN3: INSERTION-3 METHOD).

Benchmark	Clock Cycles					Fmax (MHz)					Wall Time (μs)				
	-O0	-O3	IT	IN	IN3	-O0	-O3	IT	IN	IN3	-O0	-O3	IT	IN	IN3
adpcm	41,561	41,131	22,130	22,130	10,585	47	47	49	51	53	886	866	452	438	199
blowfish	214,140	214,400	196,943	200,972	196,774	57	63	62	64	60	3,747	3,409	3,181	3,151	3,303
dfadd	870	797	796	781	788	87	91	90	92	102	10	9	9	8	8
dfdiv	2,542	2,265	2,242	2,231	2,231	65	78	75	81	71	39	29	30	28	32
dfmul	305	292	275	266	266	92	91	93	91	93	3	3	3	3	3
dfsin	71,123	64,611	63,888	63,560	63,560	48	58	50	48	46	1,480	1,110	1,284	1,312	1,389
gsm	11,051	5,897	5,428	5,186	5,412	59	49	67	57	61	187	120	81	90	89
jpeg	1,555,336	1,410,002	1,397,580	1,391,902	1,362,751	31	28	30	31	37	50,043	50,958	46,539	44,785	36,732
mips	5,276	5,244	5,225	5,184	5,184	80	79	78	79	78	66	66	67	65	66
motion	8,505	8,430	6,409	6,361	6,375	71	98	66	78	62	121	86	97	82	104
sha	249,111	206,392	202,004	201,746	201,746	66	54	73	61	58	3,756	3,854	2,764	3,291	3,472
Geomean	18,404	16,381	14,717	14,572	13,641	61	63	64	64	63	300	260	231	229	217
Ratio	1.12	1.00	0.90	0.89	0.83	0.97	1.00	1.01	1.01	1.00	1.16	1.00	0.89	0.88	0.84

Algorithm 1 Algorithm for applying optimization passes.

Input: IR_{Orig}
Input: Pass pool P
Output: $IR_{Best}, Recipe_{Best}$
 1: $Cycles_{Best} = CycleCount(IR)$;
 2: $IR_{Best} = IR$;
 3: $Recipe_{Best}$ = empty;
 4: **while** Stopping Criteria Is Not Met **do**
 5:     Choose next pass p from pass pool P;
 6:     $IR_{New}, Recipe_{New} = Apply(p, IR_{Orig}, Recipe_{Best})$;
 7:     $Cycles_{New} = CycleCount(IR_{New})$
 8:     **if** $Cycles_{New} \leq Cycles_{Best}$ **then**
 9:         $Cycles_{Best} = Cycles_{New}$;
10:         $Recipe_{Best} = Recipe_{New}$;
11:         $IR_{Best} = IR_{New}$;
12:     **end if**
13: **end while**

method is highly pass-order dependent, which is not true for the other two methods.

In the insertion method, we consider all possible insertion positions for p in the recipe that produced the best IR so far, and keep the recipe and IR corresponding to the insertion position that produced the IR with the lowest number of clock cycles. Our insertion method is thus somewhat analogous to the classic *insertion sort* algorithm which, given an element to insert into a sorted list, walks the list from beginning to end to find the correct insertion position. The advantage of the insertion method is that it reduces the dependence on the order in which the passes are applied because it attempts all possible insertion positions for each pass, selecting the position that yields the best results. Thus, its overarching intent is to find the "good" points in the ordering solution space (such as that illustrated in Fig. 3). Sorting the passes is thus unnecessary for the insertion method.

Clearly, the insertion method requires significantly more computation than the iteration method: after drawing M passes from the pool P, the iteration method will have considered M possible IRs, whereas the insertion method will have considered $M \cdot (M+1)/2$ possible IRs. The iteration and insertion method's $Apply$ functions are shown formally in Algorithms 2 and 3, respectively.

Our last variant, insertion-3, is similar to the insertion method except that it stores the top 3 IRs and recipes, instead

of storing the single best IR and recipe. In insertion-3, the chosen pass p is applied to all 3 of the top IRs/recipes. By storing three IRs/recipes instead of just one, we permit a broader exploration of the solution space. Note that different sequences of passes may produce the *same* IR (say, for example, if some passes had no impact). We require that the top 3 IRs stored be different from one another (by diff'ing the IRs), thereby ensuring diversity in the recipes/solutions considered.

Algorithm 2 Apply function for iteration method.

Input: p, IR_{Orig}, $Recipe_{Best}$
Output: IR_{New}, $Recipe_{New}$
 1: $Recipe_{New} = Recipe_{Best}$ with p added to its end;
 2: IR_{New} = IR produced by applying $Recipe_{New}$ to IR_{Orig};

Algorithm 3 Apply function for insertion method.

Input: p, IR_{Orig}, $Recipe_{Best}$
Output: IR_{New}, $Recipe_{New}$
 1: N = the # of passes in $Recipe_{Best}$;
 2: $Cycles_{Curr} = \infty$;
 3: **for** $i = 0$ to N **do**
 4:     $Recipe_{temp}$ = first i passes in $Recipe_{Best}$, followed by p, followed by the next $N - i$ passes in $Recipe_{Best}$;
 5:     IR_{temp} = IR produced by applying $Recipe_{temp}$ to IR_{Orig};
 6:     $Cycles_{temp} = CycleCount(IR_{temp})$
 7:     **if** $Cycles_{temp} \leq Cycles_{Curr}$ **then**
 8:         $Cycles_{Curr} = Cycles_{temp}$;
 9:         $Recipe_{New} = Recipe_{temp}$;
10:         $IR_{New} = IR_{temp}$;
11:     **end if**
12: **end for**

For the stopping criteria, we terminate when one of the following two conditions is true: 1) we have "walked" through all passes in the pass pool 3 times (determined empirically), or 2) no benefit was realized during the most-recently-completed "walk" through the pass pool, in which case we terminate early. Fig. 4 shows how the geomean cycle latency (across all CHStone circuits) changes across three walks through the pass pool for the *insertion-3 method*. Observe that most of the improvement in cycle latency happens in the first walk.

| Benchmark	LEs					Memory (bit)					Multipliers				
	-O0	-O3	IT	IN	IN3	-O0	-O3	IT	IN	IN3	-O0	-O3	IT	IN	IN3
adpcm	19,229	16,937	15,250	15,551	17,569	27,646	27,646	26,110	26,110	23,870	30	40	68	52	70
blowfish	6,687	6,118	6,464	6,537	6,901	150,784	150,720	150,720	150,720	150,144	0	0	0	0	0
dfadd	6,161	6,076	6,057	5,958	5,990	17,056	17,056	17,056	17,056	17,056	0	0	0	0	0
dfdiv	12,390	12,842	12,491	12,148	13,293	13,495	13,495	13,495	13,495	13,495	32	32	32	32	32
dfmul	3,559	3,884	3,617	3,436	3,481	12,032	12,032	12,032	12,032	12,032	32	32	32	32	32
dfsin	24,264	24,702	26,384	24,629	24,839	13,911	13,911	13,911	13,911	13,911	70	70	70	70	70
gsm	10,372	12,228	10,740	12,014	10,788	10,704	10,288	10,576	10,144	10,656	16	22	22	16	22
jpeg	31,870	34,351	33,215	37,473	43,594	470,427	470,054	470,427	470,150	470,523	52	50	56	46	42
mips	3,659	3,659	3,987	3,228	3,224	4,992	4,736	4,992	4,480	4,480	8	8	8	8	8
motion	16,899	4,670	18,245	5,630	16,841	34,464	33,312	34,656	33,344	34,528	0	8	0	0	8
sha	7,842	13,149	8,126	12,564	12,539	135,160	135,056	135,160	135,208	135,208	0	4	0	4	4
Geomean	10,283	9,720	10,376	9,602	10,935	30,163	29,814	29,988	29,478	29,455	8	12	9	10	13
Ratio	1.06	1.00	1.07	0.99	1.12	1.01	1.00	1.01	0.99	0.99	0.70	1.00	0.75	0.83	1.04
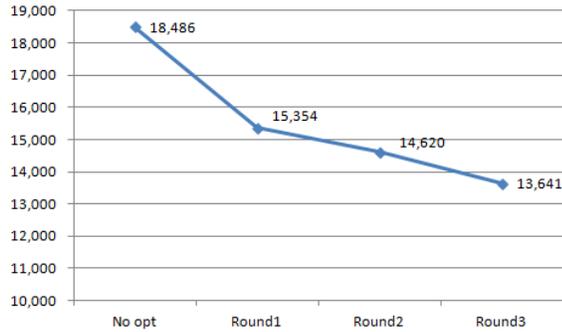
Fig. 4. Geomean clock cycle latency after each walk through the pass pool for the insertion-3 method.

V. EXPERIMENTAL RESULTS

Table V shows the speed-performance results for circuits optimized using five different compiler optimization flows: no optimization (-O0), standard -O3 optimization, the iteration method, insertion method, and the insertion-3 method. The left-most column lists the names of each benchmark. The second-last row of the table gives geometric mean results across all circuits; the last row of the table shows the ratios of the geomeans relative to -O3, which is LegUp's default optimization. Columns 2-6 give the clock cycle latencies for each of the 5 different flows. First, observe that -O3 provides a clear advantage over -O0: clock cycle latencies without any optimization are 12% higher, on average, vs. with -O3. All of the proposed flows produce significantly better results than -O3, on average. The iteration method provides 10% improvement; the insertion method offers 11% improvement; and, the insertion-3 method provides 17% improvement in cycle latency. While the largest improvements in cycle latency were seen for the adpcm benchmark (due to a significant reduction in loads/stores via their translation into register accesses), the iteration, insertion and insertion-3 methods were able to improve upon -O3 for *all* circuits.

Columns 7-11 of Table V show the post-routing *FMax* of the circuits for their Cyclone II implementation, as reported by the Altera TimeQuest static timing analysis tool. Observe that *FMax* was relatively flat across all flows, with the exception of there being a 3% degradation in *FMax* without any optimization (-O0). The five right-most columns of the table show the wall-clock execution time of the circuits for the different flows. Without any compiler optimizations, wall-clock times are 16% higher than -O3, on average. As the *FMax* changes were modest with the proposed flows, the cycle latency improvements seen with the proposed flows yield wall-clock time improvements vs. -O3. The average reductions in wall-clock time are 11%, 12%, and 16%, for the iteration, insertion, and iteration-3 methods, respectively. The results demonstrate that considerable performance gains can be had at a high-level of the design flow, prior to detailed logic synthesis, mapping and physical implementation.

Table VI gives the area results and reports the number of Cyclone II logic elements (LEs), memory bits, and multipliers used for each circuit for each of the four flows. LEs contain a 4-input look-up-table (LUT) and a flip-flop. Multiplier blocks in Cyclone II are hard ASIC-like 9-by-9-bit multipliers that are implemented in columns of the FPGA fabric, and that can be combined together to realize wider multiplications. Observe that, on average, the number of LEs and memory bits is not significantly affected by the compiler optimization flow. An exception is the LE count in the iteration and insertion-3 flows, which increased slightly due to a single benchmark, motion, whose area grew by ∼4×. This exception is due to the lack of successful application of the pass -indvars, which prevents the pass -loop-unroll from unrolling the loops and subsequently prevents other passes from significantly affecting LE count (indvars adjusts the induction variables of loops in ways that permit further optimizations to succeed). The right-most group of columns shows the multiplier block usage (for these columns, circuits that used 0 multipliers were modeled as having used 1 multiplier in the geometric mean computation). Although the ratio data appears to show a significant reduction in multiplier usage for two of three proposed flows, a detailed look at the numbers in the tables shows multiplier usage to be fairly even across all flows. We have observed that Quartus II synthesis incorporates sophisticated techniques for optimizing multiplier usage, replacing them with shifts/adds based on constant propagation (which is affected by the earlier compiler optimization passes).

We now turn to the run-time required for the various flows,

	Run-time (s)			
Benchmark	**-O3**	**IT**	**IN**	**IN3**
adpcm	1.8	127	1,872	6,578
blowfish	1.4	115	604	5,600
dfadd	0.4	163	831	1,112
dfdiv	0.5	32	625	2,881
dfmul	0.3	79	319	926
dfsin	1.3	26	2,077	3,332
gsm	1.2	250	4,931	9,079
jpeg	5.1	208	13,963	132,252
mips	0.4	448	282	2,590
motion	0.3	27	1,772	16,951
sha	0.7	82	1,487	17,755
Geomean	**0.8**	**98**	**1,312**	**5,966**
Ratio	**1**	**125**	**1,668**	**7,584**

shown in Table VII. We ran all flows and benchmarks on a single machine containing an Intel Core i5-2410M @2.30GHz processor with 2GB of RAM. The values in the table represent the run-time in seconds for all LLVM optimizations and high-level synthesis for each circuit in each of the flows (not to be confused with the wall-clock times for actual circuit execution in Table V). As with the prior tables, the bottom row of Table VII gives the ratio of the geomeans vs. the -O3 flow. The geomean run-time for the iteration method is 98 seconds, about $125\times$ higher than the -O3 flow run-time. For the insertion method, the geomean run-time is about 22 minutes, $\sim 1,700\times$ higher than the -O3 flow. The geomean run-times of the insertion-3 flow are significantly higher: 99 minutes, over $7,500\times$ higher than the -O3 flow. While the run-time of the insertion-3 method may be prohibitively large, we believe the absolute run-times are manageable for the iteration and insertion methods. The iteration method in particular provides an 11% wall-clock time reduction, on average, and its run-time is considerably less than the run-time of Altera back-end FPGA synthesis, placement and routing tools. Moreover, the approaches can be run once for a benchmark and then the recipe produced can be re-used in future compilations. The focus of our work was on understanding the potential for compiler optimizations to impact hardware quality – we did not focus on run-time. We believe that considerable run-time reductions can be achieved through a more careful analysis of when certain passes may potentially provide a benefit, allowing us to "skip" passes under certain circumstances.

In summary, we believe the proposed automated approaches to selecting compiler optimizations on a per-program basis are practical, and will be of keen interest to FPGA users seeking high design performance. Such approaches also appear to be a useful mechanism for narrowing the gap between HLS-generated hardware and manually-designed RTL. The specific recipes of optimizations selected for each benchmark for each flow could not included for space reasons, however, they are available online at `http://legup.eecg.toronto.edu` in the publications section of the website.

VI. CONCLUSIONS AND FUTURE WORK

We considered the impact of compiler optimization passes on HLS-generated hardware and proposed approaches for the automated generation of *recipes* of passes to benefit hardware speed performance. The proposed techniques work by selecting and applying a particular optimization pass, performing a fast estimation of its impact on the resulting hardware, and then potentially undoing its impact based on the predicted outcome. Results show that the automatically-generated pass recipes produce circuits with 16% better wall-clock time, on average, versus those produced using standard -O3 optimization. To the authors' knowledge, ours is the first comprehensive study of methods for applying an extensive set of compiler optimization passes in the HLS context.

Directions for future work include compiler optimizations for circuit area and power consumption. Additionally, we believe that the proposed iteration and insertion methods are just a first step towards using compiler-based techniques to improve HLS results. In particular, we believe it will be possible to prune the solution space of the insertion-3 method through memoization techniques to recognize and discard already-explored portions of the solution space, reducing run-time. We also would like to explore writing new custom optimization passes specifically intended for hardware.

REFERENCES

[1] *LLVM Compiler Project (http://www.llvm.org)*, 2010.
[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. on CAD*, vol. 30, no. 4, pp. 473 –491, 2011.
[3] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *IEEE FCCM*, 2010, pp. 127–134.
[4] A. Canis, J. Choi, and et al., "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *ACM/SIGDA FPGA*, 2011, pp. 33–36.
[5] P. Coussy, G. Lhairech-Lebreton, D. Heller, and E. Martin, "GAUT – a free and open source high-level synthesis tool," in *IEEE DATE*, 2010.
[6] *Cyclone-II FPGA family datasheet*, Altera, Corp., 2012.
[7] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Trans. on CAD*, vol. 23, no. 2, pp. 302 – 312, 2004.
[8] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *ACM/IEEE CGO*, 2003, pp. 204–215.
[9] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," in *ACM LCTES*, 2004, pp. 231–239.
[10] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *ACM/IEEE CGO*, 2006, pp. 319–332.
[11] G. Fursin, et al., "Milepost GCC: Machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, pp. 296–327, 2011.
[12] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *IEEE/ACM DAC*, 2006, pp. 433–438.
[13] *Y Explorations – C to RTL behavioral synthesis (http://www.yxi.com)*, 2012.
[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," *Jour. of Information Processing*, vol. 17, pp. 242 – 254, 2009.
[15] C. Loken, et al., "SciNet: Lessons learned from building a power-efficient top-20 system and data centre," *J. of Physics: Conference Series*, vol. 256, no. 1, 2010.