The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware

QIJING HUANG, RUOLONG LIAN, ANDREW CANIS, JONGSOK CHOI, RYAN XI, NAZANIN CALAGAR, STEPHEN BROWN, and JASON ANDERSON, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada

We consider the impact of compiler optimizations on the quality of high-level synthesis (HLS)-generated field-programmable gate array (FPGA) hardware. Using an HLS tool implemented within the state-of-the-art LLVM 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 some optimization parameter values, as well as 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.

Categories and Subject Descriptors: B.7 [**Integrated Circuits**]: Design Aids

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: High-level synthesis, FPGAs, performance, optimization

ACM Reference Format:
Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. 2015. The effect of compiler optimizations on high-level synthesis-generated hardware. ACM Trans. Reconfig. Technol. Syst. 8, 3, Article 14 (May 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2629547

1. 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). Although FPGA circuit design historically has been the realm of hardware engineers, HLS offers a path toward 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.

This work is supported by the National Sciences and Engineering Research Council of Canada, the University of Toronto, and Altera Corporation.
Authors' addresses: Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, N. Calagar, S. Brown, and J. Anderson, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 3G4, Canada; emails: qijing.huang@utoronto.ca, lanny.lian@mail.utoronto.ca, acanis@eecg.toronto.edu, jongsok.choi@mail.utoronto.ca, ryan.xi@utoronto.ca, nazanin.calagar@gmail.com, {brown, janders}@ece.toronto.edu.

DOI: http://dx.doi.org/10.1145/2629547

We believe, in fact, that FPGAs (rather than ASICs) will be the vehicle through which HLS enters the mainstream of IC design, owing to 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, Altera's OpenCL compiler for FPGAs [Altera 2012b], Xilinx's Vivado HLS tool [Xilinx 2013], ROCCC HLS from the University of California at Riverside [Villarreal et al. 2010], and LegUp from the University of Toronto [Canis et al. 2013] are implemented within the LLVM framework [LLVM 2010a]. Similarly, GAUT [Coussy et al. 2010] from the Université de Bretange Sud is implemented within GCC. Compilers perform their optimizations in *passes*, where each pass is responsible for a specific code transformation. Examples of passes include dead-code elimination, constant propagation, loop unrolling, and loop rotation. LLVM contains 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. These issues are explored in this article.

We study the impact of compiler passes using the open-source LegUp HLS tool. We target the Altera Cyclone II FPGA [Altera 2012a] 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, (3) the impact of pass parameters, and (4) 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 greater than $\pm 10\%$ is common. We also show that a given pass may improve some circuits and not others; 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 dependent, we propose an 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 that our optimization strategy consistently outperforms the standard `-O3` level in terms of hardware speed performance.

A preliminary version of this work appeared in Huang et al. [2013]. In this extended journal version, we investigate not only which passes impact HLS results but also study the effect of parameters used by certain passes, namely inlining and loop unrolling. The specific parameter values chosen and the interdependency between passes are demonstrated to significantly impact generated hardware circuits. Furthermore, we assess the effect on circuit quality of sequences of passes generated by our proposed HLS-directed approach, which we refer to as *pass recipes*. We show that many passes do not provide any benefit for generated circuits and therefore should not be included in the recipes. Finally, we examine whether HLS-oriented pass recipes are *also* beneficial for optimizing software intended to run on a standard processor (an x86 processor).

In other words, we attempt to answer the following question: for a given pass recipe shown to improve a generated hardware circuit using HLS, does that same recipe also provide better results for software running on a processor?

The rest of this article is organized as follows. Section 2 provides relevant background and describes related work. Section 3 presents results that illustrate the impact of LLVM optimization passes on generated hardware quality. In Section 4, we introduce our HLS-directed compiler optimization approach. Experimental results are presented and discussed in Section 5. Section 6 offers conclusions and suggests future work.

2. BACKGROUND

Compilers such as LLVM and GCC provide standard *optimization levels* that can be selected by the user. Higher optimization levels typically cause the compiler to perform more passes in an attempt to better optimize the generated result. The level is normally set by a compiler parameter, as in `-O1`, `-O2`, and `-O3`. The particular optimizations applied at each level are chosen to benefit the *average* results for a collection of benchmark programs. However, it is not guaranteed that a higher optimization level will give a better result for a specific program. 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 Triantafyllis et al. [2003], Almagor et al. [2004], and Pan and Eigenmann [2006]. Broadly speaking, research in the area involves devising heuristic methods to prune the large optimization space of selecting passes, thereby reducing the number of different passes that need to be applied/attempted. Milepost [Fursin et al. 2011] 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 is related to such efforts in the software domain and represents a step toward adaptive compiler optimization in the HLS *hardware* domain. A very recent related work [Cong et al. 2012], done concurrently with our own, considered source-code transformations and their impact on HLS, as well as a limited number of LLVM compiler passes. Our work, on the other hand, considers a broader set of passes and more hardware metrics (such as circuit wall-clock time), as well as proposes strategies to determine recipes of passes that work well for HLS.

2.1. The LegUp HLS and the LLVM Framework

The LegUp open-source HLS tool is implemented within the LLVM compiler framework, which is 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, composed 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.

A high-level diagram of the LegUp flow is shown in Figure 1. The LegUp HLS tool is implemented as back-end passes of LLVM that are invoked after the standard compiler passes. LegUp accepts a program's optimized IR as input and goes through the four stages shown in Figure 1 (① Allocation, ② Scheduling, ③ Binding, and ④ RTL generation) to produce a circuit in the form of synthesizable Verilog HDL code.

[1]http://www.eecs.umich.edu/mibench.

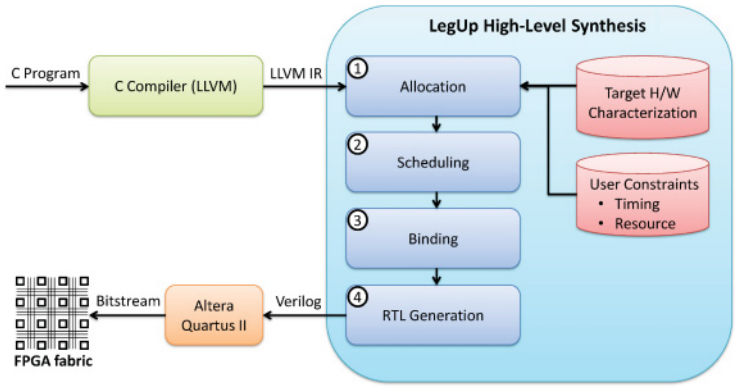

Fig. 1. LegUp flow.

The allocation stage allows the user to provide constraints to the HLS algorithms, as well as data that characterizes the target hardware. Examples of constraints are limits on the number of hardware units of a given type that may be used, the target circuit critical path delay, and directives pertaining to loop pipelining and resource sharing. The hardware characterization data specifies the speed (critical path delay) and area estimates (number of FPGA logic elements) for each hardware operator (e.g., adder, multiplier) for each supported bitwidth (typically 8, 16, 32, and 64 bit). The characterization data is collected only once for each FPGA target family using automated scripts. The scripts synthesize each operation in isolation for the target FPGA family to determine their speed and area.

At the scheduling stage, each operation in the program (in the LLVM IR) is assigned to a particular clock cycle (state) and an FSM is generated. The LegUp scheduler, based on the SDC formulation [Cong and Zhang 2006], operates at the basic block level, exploiting the available parallelism between instructions in a basic block. A *basic block* is a sequence of instructions that has a single entry and exit point. The scheduler performs operation *chaining* between dependent combinational operations when the combined path delay does not exceed the clock period constraint—chaining refers to the scheduling of dependent operations into a single clock cycle. Chaining can reduce hardware latency (number of cycles for execution) and save registers without impacting the final clock period.

The binding stage assigns operations in the program to specific hardware units. When multiple operators are assigned to the same hardware unit, multiplexers are generated to facilitate the sharing. Multiplexers require a significant amount of area when implemented in an FPGA logic fabric. Consequently, there is no advantage to sharing all but the largest functional units, namely multipliers, dividers, and recurring patterns of smaller operators. Multiplexers also contribute to circuit delay, and thus they are used judiciously by the HLS algorithms. LegUp also recognizes cases where there are shared inputs between operations, which allows hardware units to be shared without creating multiplexers. Last, if two operations with nonoverlapping lifetime intervals are bound to the same functional unit, then a single output register is used for both operations. This optimization saves a register as well as a multiplexer.

The RTL generation stage produces synthesizable Verilog HDL register transfer level code. One Verilog module is generated for each function in the C source program. Results show that LegUp produces solutions of comparable quality to a commercial HLS tool [Y Explorations 2012], and the interested reader is referred to Canis et al. [2013] for more details.

Table I. CHStone Benchmark Characteristics

Benchmark	Lines of C	Basic Blocks	IR Instructions	Class
adpcm	550	28	991	Media
blowfish	1,255	30	722	Encryption
dfadd	441	22	361	Arithmetic
dfdiv	292	62	362	Arithmetic
dfmul	270	44	273	Arithmetic
dfsin	580	221	1,098	Arithmetic
gsm	388	133	955	Media
jpeg	1,073	281	1849	Media
mips	271	49	411	Processor
motion	602	35	214	Media
sha	1,969	37	318	Encryption
Geomean	**565**	**58**	**550**	

3. IMPACT OF COMPILER OPTIMIZATIONS ON HLS

3.1. Methodology

In this section, we present an analysis of the impact of compiler optimization passes on HLS-generated hardware. We use 11 C benchmarks from the CHStone HLS benchmark suite [Hara et al. 2009]. The programs in this suite represent a variety of domains, including multimedia, communications, and encryption. The size of each benchmark in terms of the number of lines of code, basic blocks and LLVM instructions, and an indication of the type of the benchmark are 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 code for implementation in an Altera Cyclone II FPGA [Altera 2012b] using Altera Quartus II CAD system version 11.1SP2, configured for timing optimization. We simulated each generated circuit by using ModelSim to extract the number of cycles needed for the execution of the circuit (cycle latency) with the built-in input stimuli. The maximum clock frequency of the circuit (*FMax*) and area results were extracted from the Quartus II timing analysis report files. Total execution (wall-clock) time is computed as the number of execution cycles divided by postrouted *FMax*. The CHStone circuits are composed 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. Area results are reported in terms of the number of logic elements (*LE*)[2] in the targeted FPGA chip. For some experiments, we also report the number of hard multipliers[3] and/or the number of memory elements. All experiments were conducted on a cloud computing system having tens of thousands of cores [Loken et al. 2010].

3.2. Analysis of Passes in Isolation

We begin by analyzing LLVM optimization passes in isolation relative to `-O0` (no optimization). Figure 2 shows how 13 different passes affect the number of hardware execution cycles. The horizontal axis lists the name of each pass. To assess the impact of an individual pass on hardware execution cycles, we first calculate the geometric mean of execution cycles across the 11 benchmarks when a particular pass is used; we then compare this geomean with that achieved using `-O0`. The vertical axis shows

[2]Each *LE* contains a four-input look-up table (LUT) and a flip-flop.
[3]Multiplier blocks in Cyclone II FPGAs are 9- × 9-bit hardware multipliers that are implemented in columns of the FPGA fabric, which can be combined together to realize wider multipliers.

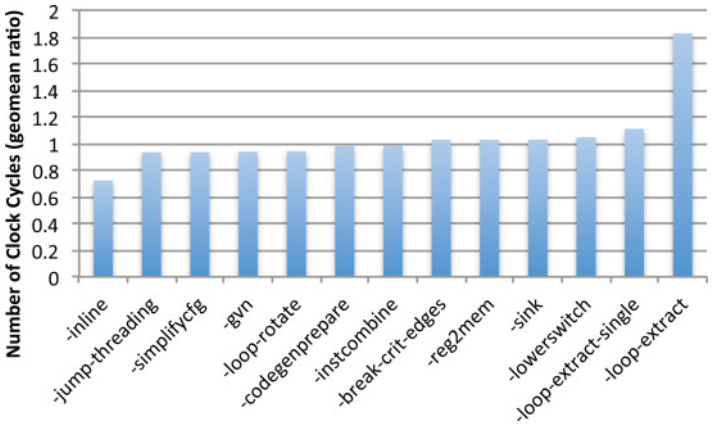

Fig. 2. Impact of compiler passes on geomean clock cycle latencies across 11 CHStone benchmarks.

Table II. Summary of the Individual Impact of 56 LLVM Different Optimization Passes on HLS Hardware

	Clock Cycles	*FMax*	Wall-Clock Time	*LE*s
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

the ratio of geomean of execution cycles relative to the `-O0` case. Values less than 1 represent reductions in cycle latency relative to the baseline case. Of the 56 different passes evaluated, only the 13 passes shown in Figure 2 impacted the geomean cycle latency by more than 1%.

Observe in Figure 2 that `-loop-extract` and `-loop-extract-single` cause 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 it implements each function as a separate Verilog module with handshaking between modules when one function calls another. Exlining loops as functions therefore 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 from a *while* loop into a *do-while* loop. This optimization 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 (Figure 2), we also analyzed *FMax*, wall-clock time, and area. Complete data for these metrics is omitted for brevity. Table II summarizes the impact of individual compiler passes on all hardware metrics. Four measurements are provided for each metric. The "Min" row gives the minimum geomean value of the metric across

[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 III. Summary of Four Impacting Passes on Clock Cycles for CHStone Benchmarks

	-inline	-jump-threading	-loop-extract-single	-loop-extract
Geomean	0.72	0.93	1.11	1.83
Min	0.31	0.82	1.00	1.07
Max	1.00	0.99	1.33	3.94
St. Dev.	0.27	0.06	0.12	1.07

11 CHStone circuits for any pass, relative to (−O0). 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 Figure 2). 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 *LE*s (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* constraint by potentially inserting more registers into the datapath to meet the specified target.

For completeness, to give a sense of the variation in a pass's impact across benchmarks, Table III provides detail for four specific passes and their impact on cycle latency: the two passes that had the most beneficial impact were `-inline` and `-jump-threading`, and the two that did the most damage were `-loop-extract-single` and `-loop-extract`. The data in the table is normalized to `-O0` (no optimization). The geomean and standard deviation are computed for a pass across all benchmarks. Observe that with the exception of `-jump-threading`, the deviations are fairly large (exceeding 10% of the mean), indicating significant variability in a pass's effect on any particular benchmark.

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

In this study, we have only considered compiler passes that are already included in the LLVM compiler. However, we can envision other hardware-specific compiler passes that would improve the hardware circuits produced by LegUp. For instance, HLS can exploit instruction-level parallelism (ILP) by synthesizing a datapath with the exact number of functional units needed to extract all of the available parallelism. Therefore, compiler passes that increase ILP will benefit HLS performance. Studies have found that basic blocks are generally short, 5 to 20 instructions on average, limiting the amount of ILP that can be found in one basic block. LegUp schedules operations in the control flow graph assuming that there is an implicit barrier between basic blocks, such that all operations from one basic block must finish before starting the next basic block (except for pipelined loops). Consequently, we would prefer to avoid control flow and execute operations from different basic blocks in parallel. Compiler passes such as speculative code motion have been studied in the SPARK compiler [Gupta et al. 2003] that move instructions across basic block boundaries and improve ILP. Furthermore, relevant research has been conducted in compilers targeting very long instruction word (VLIW) processors. VLIW processor architectures contain multiple functional units that can execute in parallel, leading to compiler optimizations that are also applicable to HLS. Compiler techniques in this area include trace scheduling

Table IV. Customized Recipe for the `dfmul` Benchmark

	Normalized Hardware Metric			
Recipe	Clock Cycles	*FMax*	Wall-Clock Time	*LE*s
-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
*LE*s	1.02	0.99	1.02	**0.91**

[Fisher 1981], which reduces the length of a sequence, or *trace*, of basic blocks that are frequently executed by the program. This trace is scheduled as if it were a single basic block, improving ILP along the trace. A similar technique proposed in Mahlke et al. [1992] begins by eliminating all branches internal to the control flow graph by using if-conversion, which converts control dependencies into data dependencies by using the result of branch conditions, called *predicates*, to conditionally execute instructions. The new basic block, called a *hyperblock*, can only be entered at the top but has multiple exit branches. This hyperblock can be scheduled as a single basic block with correspondingly improved ILP. Reverse if-condition optimization can be used to regenerate the control flow graph, or HLS can include predicate support in the final hardware. Loop iterations can also be executed in parallel using software pipelining algorithms such as iterative modulo scheduling [Rau 1996], which has been studied extensively in HLS. These speculative compiler techniques will usually improve performance, but at the cost of increasing area. Studying the impact and interaction of these hardware-specific compiler passes in HLS, or developing new passes that improve ILP, is an area for future work.

3.3. Customized Passes

To study 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 (*LE*s). We created four 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 later).

Results for the custom recipes for a representative benchmark, `dfmul`, are given in Table IV.[5] The left-most column of Table IV 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% versus `-O3`, and the *FMax* recipe improves *FMax* by 2%. The wall-clock time recipe improves wall-clock time by 9%—a significant improvement over `-O3`. Although 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 indeed is considerable potential to improve upon `-O3` results.

3.4. Impact of Pass Parameters

In addition to the impact of passes themselves, some passes have parameters that significantly impact how the pass operates, and thereby impact the HLS-generated hardware. In this section, we study the impact of two parameters found to have a prominent role in HLS: `-unroll-threshold`, which controls the `-loop-unroll` pass,

[5]Results for other circuits are omitted for brevity.

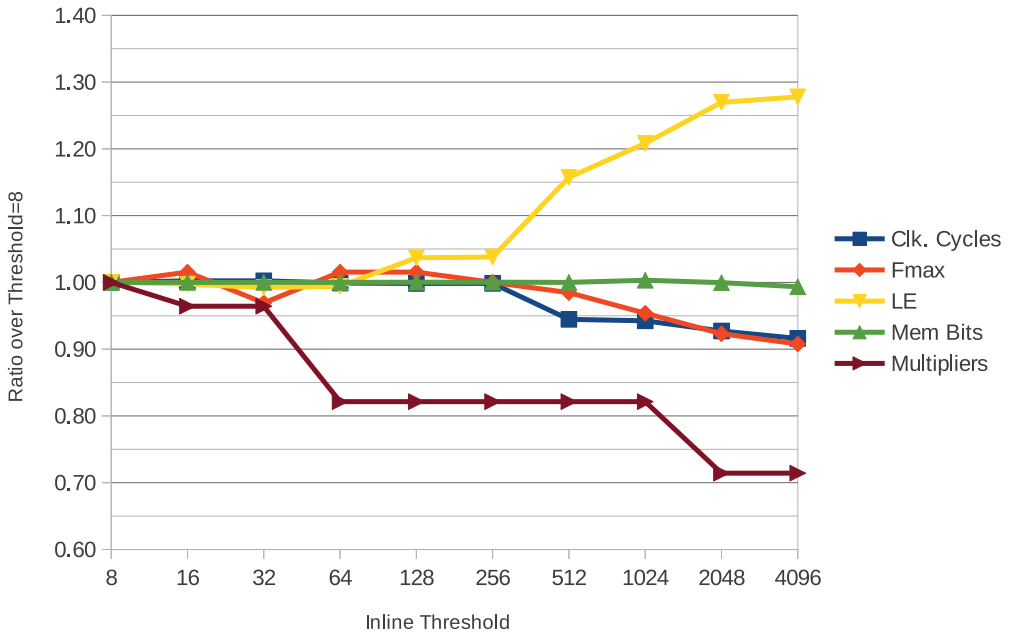

Fig. 3. Speed and area impact of `-inline-threshold` on generated hardware.

and `-inline-threshold`, which controls the `-inline` pass. As mentioned previously, the `-inline` pass can reduce cycle latency as it (1) allows optimization to happen *across* function boundaries and (2) eliminates the handshaking protocol between modules (functions) in the LegUp-generated hardware. In LLVM's inlining pass, a function is inlined if the resulting "cost" is less than the `-inline-threshold` parameter value, where the inlining cost is determined according to several properties of the function: the number of instructions in the function, the number of calls it makes to other functions, and the types and the sizes of the function arguments. If not specified, the default value of `-inline-threshold` is 225. This value is set empirically to encourage the inlining of infrequently called small functions.

Figure 3 shows the delay and area results of the hardware circuit generated when the `-O3` optimization level is used with different `-inline-threshold` values. For this experiment, it is important to also look at the usage of multipliers and memory elements, as they can also be significantly affected by the thresholds. The horizontal axis shows the `-inline-threshold` values; the vertical axis represents the geometric mean ratio (over the 11 benchmarks) relative to that achieved when `-inline-threshold=8`. Note that for multiplier usage, arithmetic mean was used instead of geometric mean, because some benchmarks used 0 multipliers.

Examining Figure 3, we observe that the geomean cycle latency starts to decrease when `-inline-threshold=512` and reduces to 0.92 when `-inline-threshold=4096`. However, we also see that *FMax* decreases when more inlining is performed. As more functions are inlined, the number of states needed in the generated finite state machines (FSMs) increases. When the number of states increases excessively, the FSM logic becomes the critical path and causes lower *FMax*. The `motion` benchmark is a representative example of this behavior. As shown in Table V, *FMax* drops from 98.57MHz to 50.53MHz when `-inline-threshold` is increased from 256 to 4,096. Another consequence of increasing the `-inline-threshold` is that the number of logic elements may also dramatically increase. If a function is called multiple times in a program, then

Table V. Largest State Machine Size and *FMax* with Different `-inline-threshold` Values in the `motion` Benchmark

`-inline-threshold`	256	512	1,024	2,048	4,096
Number of States	66	192	192	192	369
FMax (MHz)	98.57	62.78	66.68	65.74	50.53

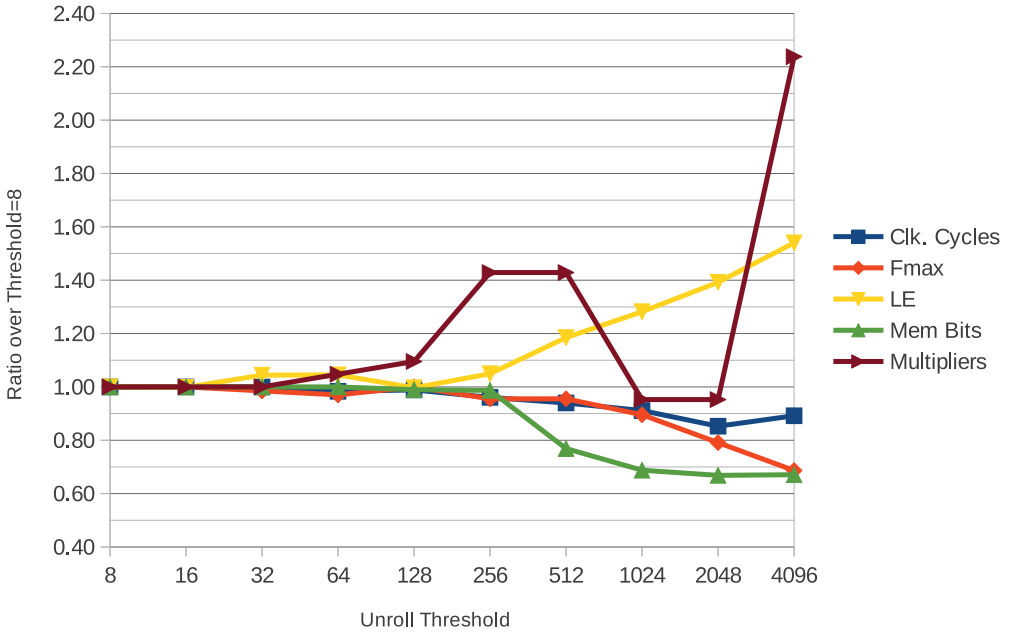

Fig. 4. Speed and area impact of `-unroll-threshold` on generated hardware.

inlining this function essentially creates multiple "copies" of the function, increasing total circuit area in hardware. The usage of memory bits stays mostly constant, as inlining has no direct impact on memory usage. The usage of multipliers tends to decrease as more functions are inlined. Note that the LegUp HLS tool does not share resources across separate functions. However, it is able to recognize recurring operation patterns within a single function and share resources [Hadjis et al. 2012]—inlining "creates" larger functions, thereby exposing more sharing opportunities.

We also investigated the `-loop-unroll` pass, as it can typically reduce cycle latency as follows. When there are few (or no) data dependencies across loop iterations (loop-carried dependencies), unrolling the loop exposes ILP. In addition, a more subtle effect of loop unrolling is that memory accesses in unrolled loops can be converted by LLVM into register accesses. This is also beneficial to hardware performance, as register accesses are faster than memory accesses. Loop unrolling happens when the size of the unrolled loop is less than the `-unroll-threshold`. The `-loop-unroll` pass normally requires the `-indvars` pass to canonicalize the loops so that trip counts of loops can be determined easily [LLVM 2010b] (`indvars` adjusts the induction variables of loops in ways that permit further optimizations to succeed). Thus, the `-loop-unroll` pass does not show a great impact in isolation. In LLVM, the default `-unroll-threshold` is 150.

We synthesized the circuits using the `-O3` optimization level with different values of `-unroll-threshold`, ranging from 8 to 4,096. In Figure 4, the results for the generated hardware are presented as the geometric mean ratio (arithmetic mean for multipliers) relative to the baseline case with `-unroll-threshold=8`. As illustrated, the clock

Table VI. Impact of -unroll-threshold on the Generated Hardware for the adpcm Benchmark

-unroll-threshold	Number of Multipliers	Un-Inlined Functions Name	Calls (#)	Multi. (#)	Execution Cycles
32	34	main	1	34	37,706
64	46	main	1	46	31,581
128	62	main upzero	1 200	62 0	34,465
256	140	main upzero	1 200	140 0	25,732
512	140	main upzero	1 200	140 0	25,634
1,024	48	main upzero encode decode	1 200 50 50	0 0 24 24	27,650

cycle latencies decrease progressively with more unrolling, with a slight increase when -unroll-threshold=4096. This increase is mainly caused by the gsm benchmark, whose total execution cycles increased from 5,227 to 8,576 when -unroll-threshold increased from 2,048 to 4,096. In this benchmark, when -unroll-threshold=4096, one of the functions is no longer inlined, which prevents optimizations across function boundaries and causes more handshaking operations.

Similar to the -inline-threshold case, *FMax* drops as the number of states increases with more unrolling. For example, in the gsm benchmark, when -unroll-threshold is set to 1,024, 2,048 and 496, the numbers of states in the largest FSMs are 325, 948, and 1,809, respectively. The corresponding *FMax* results are 45.73MHz, 25.71MHz, and 6.39MHz, respectively. As the loops are unrolled, some variables that would have been represented as arrays are represented as individual variables. Such variables are then implemented as registers instead of memory blocks, causing *LE* usage to increase, and the number of memory bits to drop. Unlike the previous result for -inline-threshold, the average usage of multipliers fluctuates when -unroll-threshold is varied. We explain this using one of the benchmarks, adpcm.

Table VI shows hardware characteristics for the adpcm benchmark when -unroll-threshold is varied from 32 to 1,024. The left-most column shows the values of -unroll-threshold and the second column shows the total number of multipliers used in the generated circuits. The left side of the third column lists the functions that are not inlined. For example, when -unroll-threshold=32,64, only the main function exists in the program—all other functions are inlined into the main function. Following that, the number of calls and the number of multipliers in the corresponding function are provided. The last column shows the cycle latency of the hardware. When -unroll-threshold increases from 32 to 512, we see that the multiplier usage progressively increases. Generally, multiplier usage increases when the loops that contain multiplications are unrolled. However, multiplier usage drops from 140 to 48 when -unroll-threshold=1024. This is because the encode and decode functions are no longer inlined into the main function—*unrolling* the loops causes the function's *inlining* costs to exceed the -inline-threshold. This result shows that inlining and unrolling interact with one another. Since both the encode and decode functions are called 50 times and use 24 multipliers each, not inlining the functions prevents them from being duplicated 50 times, which reduces the number of multipliers. Besides multiplier usage, we see a slight increase in cycle latency for -unroll-threshold=128 and 1,024, since there are functions not being inlined. Thus, there is an interdependency between

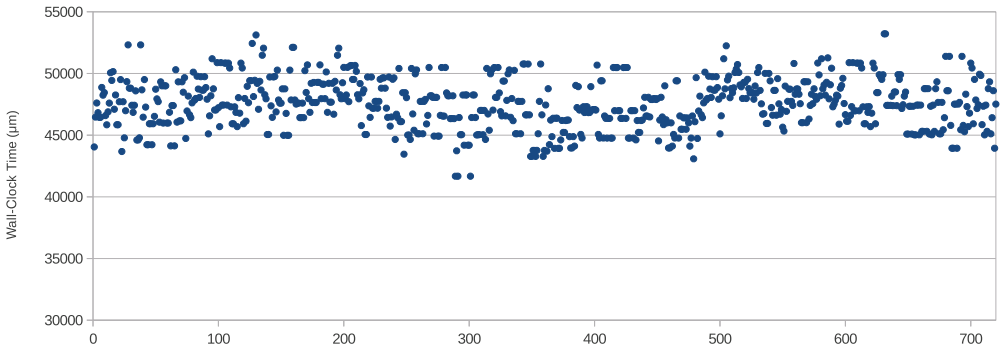

Fig. 5. Wall-clock time for the `jpeg` benchmark for all permutations of six optimization passes.

the two parameters: `-inline-threshold` and `-unroll-threshold`. Looking back at multiplier usage in Figure 4, the steep increase at `-unroll-threshold=4096` is caused by unrolling of a multiply-intense loop in the `gsm` benchmark.

In summary, beyond the chosen passes, the parameters provided to such passes can have a considerable impact on the quality of HLS-generated hardware, further underscoring the immense size of the optimization space.

3.5. Impact of Pass Ordering

We also consider the order in which passes are applied and have found it to have a significant impact on the hardware quality. Figure 5 shows the wall-clock time for the `jpeg` benchmark for all 6! (=720) orderings of the *same* six 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, with a minimum of 41.7ms and a maximum of 53.2ms (nearly 28% higher than the minimum). The results in Figure 5 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 it also is crucial to determine the order in which they should be applied.

To further study the impact of pass ordering, we selected 33 passes, comprised of all of 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 pairs of passes from this group and evaluated the pairs in both orders, performing synthesis, placement, routing, and ModelSim simulation for all $2 \times \binom{33}{2} = 2 \times 528 = 1{,}056$ 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 Figure 6. Of the 528 pass pairs, 117 had no impact in either order, and for 55 pairs the orders were tied. 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. The results clearly demonstrate the importance of pass ordering on HLS quality of results for the majority of pass pairs. Although it is tractable to evaluate all combinations of pairs of passes, it is computationally intractable to investigate all orderings of larger numbers of passes.

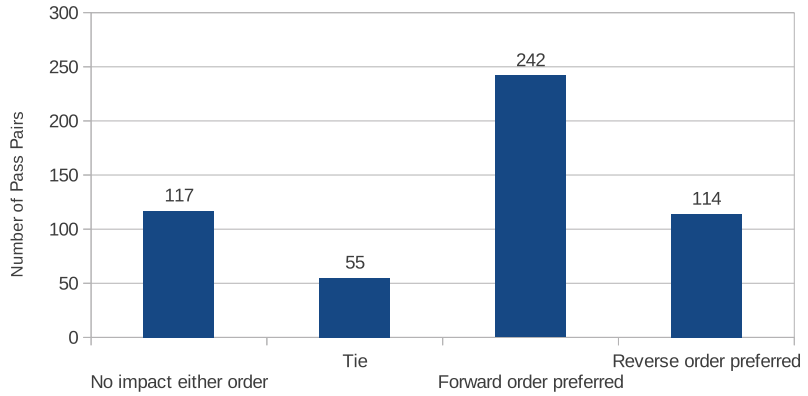

Fig. 6. Impact of pass pair forward/reverse ordering on clock cycle latency. Results are shown for all $\binom{33}{2} = 528$ combinations of 33 passes.

From the analysis of passes in isolation, we also generated a general benchmark-agnostic recipe containing only those passes that showed a benefit for a *majority* of benchmarks when applied in isolation. On average, the recipe performed worse than `-O3`, because some passes depend on other passes to show any impact. For example, there are passes that showed no benefit for a benchmark when applied in isolation, yet they showed a benefit when applied after certain other passes. Clearly, the `-O3` recipe includes some such passes that 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 that it would be difficult to devise a single recipe of passes that would benefit *all* circuits. We therefore opted to explore 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.

4. 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 later discussion). If p is deemed beneficial (line 8), then it is accepted and its effect on the IR is left intact (lines 9 through 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 reapply passes. The process of selecting passes from the pool and judiciously applying them continues until a stopping criteria is met (also discussed later). Note that although 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. Likewise, although the CHStone benchmarks do not exhibit significant opportunities for loop pipelining, it would be useful in the future to consider the impact of pass recipes on loop pipelining using a different benchmark set.

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

Table VII. Runtime Comparison between Proposed Profiler and ModelSim

	Simulation Time (s)	
Benchmark	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

PR, Profiler; MS, ModelSim.

determined if two criteria are known for each basic block:[6] (1) the number of times that it is executed and (2) the number of clock cycles that 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 $SchedLen(b)$ 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 although the profiling step may be deemed as costly from the runtime angle in a software compilation flow, the time consumed is very small compared to ModelSim simulation of the Verilog code. Table VII compares the runtime 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 later), the *Apply* function that executes the selected pass p on the best IR seen so far, and the composition of the pass pool P. For P, we use 41 of the 56 LLVM passes, including (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). Default values are used for pass parameters (e.g., the inlining threshold).

We implemented and evaluated three variants of Algorithm 1 offering different runtime/quality trade-offs, which we call 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 recipe

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

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

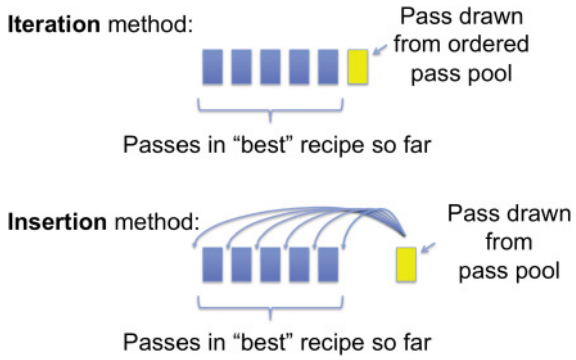

Fig. 7. Illustration of the iteration versus insertion methods.

ALGORITHM 1: Algorithm for applying optimization passes.

Input: IR_{Orig}
Input: Pass pool P
Output: IR_{Best}, $Recipe_{Best}$

```
1: Cycles_Best = CycleCount(IR_Orig);
2: IR_Best = IR_Orig;
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 ≤ Cycles_Best then
9:            Cycles_Best = Cycles_New;
10:           Recipe_Best = Recipe_New;
11:           IR_Best = IR_New;
12:      end if
13: end while
```

found so far and generates a new IR with the new recipe. In the iteration method, we first sort all passes based on the pairs analysis results (see Section 3) 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 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 (e.g., like that illustrated in Figure 5). Sorting the passes is thus unnecessary for the insertion method. Figure 7 pictorally illustrates the difference between the iteration method and the insertion method: with the iteration method, the

ALGORITHM 2: Apply function for the 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 the 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**

selected pass is added to the end of the "best pass so far" recipe; with the insertion method, all possible insertion points are explored.

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 three IRs and recipes instead of storing the single best IR and recipe. In insertion-3, the chosen pass p is applied to all three 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, e.g., if some passes had no impact). We ensure that the top three IRs stored are different from one another, thereby creating diversity in the recipes/solutions considered.

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 three 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. Figure 8 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.

5. EXPERIMENTAL STUDY

5.1. Speed Performance Analysis

Table VIII 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

Table VIII. Speed Performance Results

	Clock Cycles					*FMax* (MHz)					Wall Time (μs)				
Benchmark	−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**

(IT: Iteration Method, IN: Insertion Method, IN3: Insertion-3 Method).

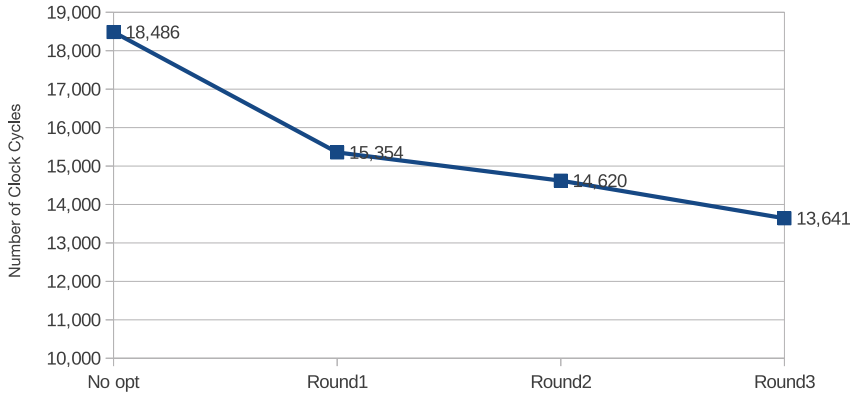

Fig. 8. Geomean clock cycle latency after each walk through the pass pool for the insertion-3 method.

column lists the names of each benchmark. The next-to-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 through 6 give the clock cycle latencies for each of the five different flows. First, observe that `-O3` provides a clear advantage over `-O0`: clock cycle latencies without any optimization are 12% higher, on average, versus those 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. Although the largest improvements in cycle latency were seen for the `adpcm` benchmark (due to a greater amount of loop unrolling and the subsequent 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 through 11 of Table VIII show the postrouting *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 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 versus `-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 the HLS stage of the design flow.

5.2. Area Analysis

Table IX gives the area results and reports the number of Cyclone II logic elements (*LE*s), memory bits, and multipliers used for each circuit for each of the four flows. Observe that on average, the number of *LE*s 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 nearly 3×. 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 allows a major function to be inlined three times, increasing area.

Table IX. Area Results

	LEs					Memory (bits)					Multipliers				
Benchmark	−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**

(IT: Iteration Method, IN: Insertion Method, IN3: Insertion-3 Method).

Table X. LLVM/HLS Runtime

	Runtime (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**

IT, iteration method; IN, insertion method; IN3, insertion-3 method.

The "Multipliers" columns show the multiplier block usage (for these columns, the geometric mean across the 11 benchmarks is used to present the results). Although a significant variation in the number of multipliers is shown in several benchmarks for different flows, a detailed look at the average results for each flow shows multiplier usage to be fairly even across all flows (within the range of ±4 multipliers of the -O3 result). We have observed that Quartus II synthesis incorporates sophisticated techniques for optimizing multiplier usage, replacing them with shifts/adds based on constant propagation.

5.3. Runtime Analysis

We now turn to the runtime required for the various flows, shown in Table X. 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 runtime in seconds for all LLVM optimizations and HLS for each circuit in each of the flows (not to be confused with the wall-clock times for actual circuit execution in Table VIII). As with the prior tables, the bottom row of Table X gives the ratio of the geomeans versus the `-O3` flow. The geomean runtime for the iteration method is 98 seconds, about 125× higher than the `-O3` flow runtime. For the insertion method, the geomean runtime is about 22 minutes, nearly 1,700× higher than the `-O3` flow. The geomean runtimes of the insertion-3 flow are significantly higher: 99 minutes, more than 7,500× higher than the `-O3` flow. Although the runtime of the insertion-3 method may be prohibitively large, we believe that the absolute runtimes are manageable for the iteration and insertion methods. The iteration method particularly provides an 11% wall-clock time reduction, on average, and its runtime is considerably less than the runtime of the 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 reused 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 runtime. We believe that considerable runtime 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. We explore this direction in the next section.

Table XI. Average Number of Passes in Recipes

Methods	Original Recipes	"Shrunk" Recipes
Iteration	67.6	5.9
Insertion	80.1	10.6
Insertion-3	80.0	12.9

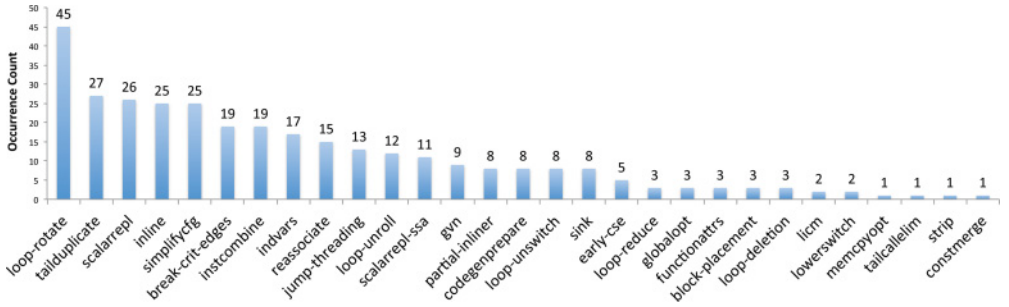

Fig. 9. Number of times that each pass is included in the reduced recipes.

5.4. Quality of Recipes

As shown in line 8 of Algorithm 1, a pass is added to the recipe as long as it does not increase cycle latency. The rationale for this is that some passes have no impact in isolation, but they may have positive impact when applied in conjunction with other passes. A question that arises then is this: do all passes included in the recipes provide a positive effect in terms of cycle latency? To answer this question, we developed a simple approach that iteratively removes each pass from a recipe one at a time and reassesses the cycle latency of the hardware. If the pass did not affect cycle latency, it is permanently removed; otherwise, it is reinserted in its original location. Through this straightforward approach to gauging pass utility, we found that it was possible to dramatically reduce the size of the recipes.

Table XI lists the average number of passes included in the original recipe, as well as the new reduced recipe. Clearly, the original recipes include many redundant passes that do not positively impact circuits in terms of cycle latency. We also noticed that only 29 passes (with 41 passes in the pass pool) remained in the 33 (3 methods $\times$ 11 benchmarks) reduced recipes. With fewer passes in the pass pool, the runtime of the different recipe-generation approaches can be reduced considerably. Therefore, we changed the pass pool to include only these 29 passes and reran the three optimization flows. By doing this, the geomean runtime of the iteration, insertion, and insertion-3 methods was reduced by 57.4%, 31.0%, and 66.1%, respectively. The cycle latencies of the generated circuits were improved slightly, by 0.29%, 0.41%, and 1.09%, respectively, versus when the 41-pass pool was used. Therefore, some of the removed passes had a slightly negative impact on cycle latency.

We also noticed that in the 33 reduced recipes, some passes were much more frequently used than others. As shown in Figure 9, the `-loop-rotate` pass was included 45 times, whereas 17 other passes were included fewer than 10 times. Next, we categorize the commonly used passes into three groups based on their impact.

5.4.1. Passes That Increase ILP. As mentioned in Section 3.2, compiler passes that increase ILP can benefit HLS-generated hardware. Many of the frequently used passes permit more ILP by merging basic blocks. For example, the `-tailduplicate` pass duplicates the basic blocks ending in unconditional branches into the tails of their predecessor basic blocks. Similarly, the `-simplifycfg` pass merges pairs of basic blocks

that are connected with an unconditional branch edge. The `-jump-threading` pass is another commonly used pass that duplicates basic blocks within other basic blocks and enables more ILP. In the case where a basic block has multiple predecessors and multiple successors, if some of its predecessors always cause a jump to one of its successors, `-jump-threading` forwards the branches from these predecessors to the corresponding successors and duplicates the basic block into the successors. Last, the `-loop-unroll` and `-inline` passes mentioned in Section 3.4 also have a significant impact on ILP.

5.4.2. Passes That Eliminate Operations. Other beneficial compiler passes eliminate operations by moving or eliminating instructions. For instance, all preceding passes that duplicate/merge basic blocks have a secondary effect, which is the removal of branch operations. Moreover, the `-loop-rotate` transforms *while* loops into *do-while* loops and eliminates the branch instructions at the beginning of the loops. The `-loop-unswitch` pass transforms the loops that contain loop-invariant branches into multiple different loops and moves the loop-invariant branches in front of the duplicated loops as predecessors. Both `-loop-rotate` and `-loop-unswitch` eliminate branch operations from loop bodies and thereby reduce overall latency. The `-sink` pass can also reduce redundant operations by pushing instructions into successor blocks so that they will not be executed if their results are not needed. The `-instcombine` pass also reduces operation count by combining algebraic instructions into fewer instructions, where possible. Two other frequently used passes are `-scalarrepl` and `-scalarrepl-ssa`. These can potentially replace structures and arrays with registers, reducing load/store (memory) operations.

5.4.3. Passes That Permit More Optimization. Some passes do not directly improve the circuit performance, but they permit other compiler passes to perform more optimizations. The `-break-crit-edge` pass breaks the critical edges[8] by adding a dummy basic block on the edge. With this pass, many passes that cannot operate with critical edges can then be applied. The `-indvars` adjusts the induction variables of loops to an analyzable form and enables further optimizations. The `-reassociate` pass reassociates commutative algebraic expressions in an order that can promote better constant propagation. These three passes are included in the 33 reduced recipes 19, 17, and 15 times, respectively.

5.5. Fine-Grained Optimization

In the preceding study, the compiler passes were applied "globally" to a benchmark, meaning that the entire benchmark was subjected to the same set of passes. Since the impact of compiler passes is program dependent, it is conceivable that the performance of the generated circuits could be improved if the compiler passes were selectively applied at a finer-grained function level. To investigate this, we created a flow that does the following: (1) iteratively applies the iteration method (without inlining) on each function's IR to obtain a new IR with the minimum cycle latency, (2) replaces the functions in the original IR of the entire benchmark with the optimized function IRs, and (3) reruns the iteration method on the updated benchmark IR with inlining enabled.

We found that this new flow impacted cycle latency by less than 0.5%, on average, versus the global approach. We believe that this is because most of the CHStone benchmarks contain one function that dominates overall execution time, making the

[8]A critical edge is an edge connecting a predecessor block that has multiple successors to a successor block that has multiple predecessors. Many optimizations cannot be applied on such a control-flow graph structure. For example, the `-tailduplicate` pass cannot be applied on a critical edge since the predecessor(s) of the duplicated basic block could branch to other basic blocks.

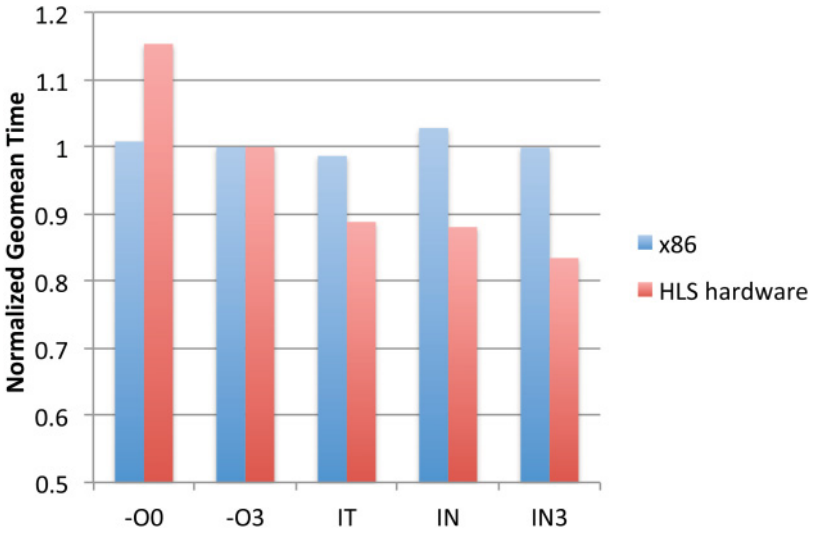

Fig. 10. Wall-clock time time of program execution on x86 and HLS-generated hardware normalized to -O3.

impact of optimizing the dominating function quite similiar to that of optimizng the entire program. It is an interesting direction for future work to consider fine-grained optimizations on a different benchmark set.

5.6. Recipes for x86 Processors

We now consider whether the recipes determined to be "good" for HLS-generated hardware performance are also beneficial to software running on a standard ×86 processor. We first optimized software code using the pass recipes produced by the three recipe-generation approaches for HLS. We then targeted the optimized code to an ×86 processor and profiled execution time using a Linux profiler called `Perf`.[9] We compared the ×86 results with the wall-clock time of the HLS-generated hardware. For the ×86 results, we profiled each program 3,000 times and took the average value to improve data accuracy. We found this to be necessary, as the runtime data fluctuated as the ×86 machine state varied.

Figure 10 shows the execution time results for programs executed on the ×86 processor, as well as the wall-clock time for HLS-generated hardware, normalized to `-O3`. The y-axis represents the geomean time across 11 benchmarks. The x-axis represents the recipes from the different flows. The figure shows that execution time is inconsistent across the two different compute platforms. For the ×86 runtimes, little variation is observed for the different optimization schemes—all approaches are within 3% of the `-O3` results. For HLS-generated hardware, the insertion method provides better wall-clock time than the iteration method, and the insertion-3 method provides further wall-clock time reductions versus the insertion method. However, such trends are not evident in the ×86 results. The ×86 results therefore are not a good predictor of circuit wall-clock time. In other words, the passes that enable more parallelism in HLS do not have an analogous impact on the x86, probably due to the more sequential nature of processor architecture.

[9]https://perf.wiki.kernel.org.

5.7. Relevance to Commercial HLS Tools

Although the preceding results demonstrate that careful selection of compiler pass recipes has a considerable impact on the speed of LegUp-generated hardware, a natural question is whether such results will also hold for commercial HLS tools. Unfortunately, to the authors' knowledge, commercial HLS vendors do not permit users to change the set of compiler passes applied to a program before HLS commences, and consequently, replicating the preceding results would best be done by the commercial vendors themselves. Note, however, that both Xilinx's VivadoHLS and Altera's OpenCL SDK are themselves built with the same LLVM compiler as LegUp. These commercial HLS tools are therefore able to apply the same sets of passes to programs as used in this work. We expect that the preceding results are not unique to the LegUp context and that performance gains are also be possible with commercial tools.

The use of custom pass recipes is not possible with commercial tools; however, such tools do provide a variety of directives that a user may specify. Taking Xilinx's VivadoHLS as an example, the tool has directives to inline functions, optimize specific instances of functions, pipeline and perform dataflow optimizations for increased concurrency, and set a target latency constraint for a function in cycles. Of these, only inlining resembles one of the LLVM compiler passes. At the loop level, VivadoHLS offers constraints for unrolling, merging consecutive loops, flattening nested loops, dataflow-style hardware generation for consecutive loops, loop pipelining, specifying loop trip counts, and a target latency in cycles. Of these, only loop unrolling resembles an LLVM compiler pass. VivadoHLS also supports array optimization directives that likewise bear little similarity to the existing LLVM compiler passes. Hence, we believe that there is little overlap between the 50+ compiler optimization passes available in LLVM and the constraints currently offered by commercial HLS tools.

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.

6. 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 among the first comprehensive studies 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 toward using compiler-based techniques to improve HLS results. In particular, we believe that it will be possible to prune the solution space of the insertion-3 method to reduce its runtime. We also would like to explore writing new custom optimization passes specifically intended for hardware.

REFERENCES

Altera. 2012a. *Implementing FPGA Design with the OpenCL Standard*. White Paper WP-01173-2.0. Altera Corporation. Available at http://www.altera.com/literature/wp/wp-01173-opencl.pdf.

Altera. 2012b. *Cyclone-II FPGA Family Datasheet*. Altera Corporation.

Lelac Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*. 231–239.

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systms* 13, 2, Article No. 24.

Jason Cong, Bin Liu, Raghu Prabhakar, and Peng Zhang. 2012. A study on the impact of compiler optimizations on high-level synthesis. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. 143–157.

Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 2006 43rd ACM/IEEE Design Automation Conference (DAC'06)*. 433–438.

Phillipe Coussy, Ghizlane Lhairech-Lebreton, Dominique Heller, and Eric Martin. 2010. GAUT—a free and open source high-level synthesis tool. In *Proceedings of IEEE Design Automation and Test in Europe (DATE'10)*.

Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming* 39, 296–327. Issue 3.

Joseph A. Fisher. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* 100, 7, 478–490.

Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. 2003. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the 16th International Conference on VLSI Design*. 461–466.

Stefan Hadjis, Andrew Canis, Jason Anderson, Jongsok Choi, Kevin Nam, Tomasz Czajkowski, and Stephen Brown. 2012. Impact of FPGA architecture on resource sharing in high-level synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*. 111–114.

Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing* 17, 242–254.

Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, and Jason Anderson. 2013. The effect of compiler optimizations on high-level synthesis for FPGAs. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'13)*. 89–96.

LLVM. 2010a. The LLVM Compiler. Infrastructure. Retrieved April 9, 2015, from http://www.llvm.org.

LLVM. 2010b. LLVM Loop Unroll Pass. Retrieved April 9, 2015, from http://www.llvm.org/docs/Passes.html#loop-unroll-unroll-loops.

Chris Loken, Daniel Gruner, Leslie Groer, Richard Peltier, Neil Bunn, Michael Craig, Teresa Henriques, Jillian Dempsey, Ching-Hsing Yu, Joseph Chen, L. Jonathan Dursi, Jason Chong, Scott Northrup, Jaime Pinto, Neil Knecht, and Ramses Van Zon. 2010. SciNet: Lessons learned from building a power-efficient top-20 system and data centre. *Journal of Physics: Conference Series* 256, 1, 012026.

Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. In *ACM SIGMICRO Newsletter* 23, 45–54.

Zhelong Pan and Rudolf Eigenmann. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. 319–332.

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. 2003. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. 204–215.

Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. 2010. Designing modular hardware accelerators in C with ROCCC 2.0. In *Proceedings of the 2010 IEEE 18th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'10)*. 127–134.

B. Ramakrishna Rau. 1996. Iterative modulo scheduling. *International Journal of Parallel Processing* 24, 13–64.

Xilinx. 2013. *C-Based Design: High-Level Synthesis with the Vivado HLS Tool*. Technical Report. Xilinx Incorporated. Available at http://www.xilinx.com/training/dsp/high-level-synthesis-with-vivado-hls.htm.

Y. Explorations. 2012. Y Explorations—C to RTL Behavioral Synthesis. Retrieved April 9, 2015, from http://www.yxi.com.

Received September 2013; revised February 2014; accepted April 2014