Profiling-Driven Multi-Cycling in FPGA High-Level Synthesis

Stefan Hadjis[1], Andrew Canis[1], Ryoya Sobue[2], Yuko Hara-Azumi[3], Hiroyuki Tomiyama[2], Jason Anderson[1]

[1]Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada
[2]Dept. of Electronic and Computer Engineering, Ritsumeikan University, Shiga, Japan
[3]Dept. of Communications and Compuer Engineering, Tokyo Institute of Technology, Tokyo, Japan
Email: legup@eecg.toronto.edu

Abstract—Multi-cycling is a well-known strategy to improve performance in digital design, wherein the required time for selected combinational paths is lengthened to multiple clock cycles (rather than just one). The approach can be applied to paths associated with computations whose results are not needed immediately – such paths are allowed multiple clock cycles to "complete", reducing the opportunity for them to form the critical path of the circuit. In this paper, we consider multi-cycling in the high-level synthesis context (HLS) and use software profiling to guide multi-cycling optimizations. Specifically, prior to HLS, we execute the program in software with typical datasets to gather data on the number of times each code segment executes. During HLS, we then extend the schedule for infrequently executed code segments and apply multi-cycling to the dilated schedules, which exhibit greater opportunities for multi-cycling. In essence, our approach ensures that non-frequently executed code segments will not form the critical path of the HLS-generated circuit. In an experimental study targeting the Altera Stratix IV FPGA, we evaluate the impact on speed performance and area for both traditional multi-cycling, as well as the proposed software profiling-driven multi-cycling, and show that profiling-driven multi-cycling leads to an average speedup of over 10% across 13 benchmark circuits, with some circuit speedups in excess of 30%. Circuit area is reduced by 11%, yielding a mean 20% improvement in area-delay product.

I. Introduction

High-level synthesis (HLS) methodologies automatically synthesize a hardware circuit from a software program, easing the burden of hardware design by allowing software methodologies to be used. HLS is gaining popularity as a methodology for field-programmable gate arrays (FPGAs) and commercial HLS tools are available from both of the main vendors [1, 2]. The strategy of the vendors with respect to HLS is to: 1) make their technology easier to use for hardware engineers, and 2) ultimately facilitate the use of FPGAs by software engineers to implement compute accelerators. There is a quality gap, however, between HLS-generated hardware and human-crafted hardware – the auto-generated hardware may be slower and consume more area and power than a manually designed circuit, especially for applications where there exists a specific spatial layout that delivers high performance. In this paper, we aim to boost the speed of HLS circuits through multi-cycling of combinational paths.

In traditional sequential circuit design, combinational path delays may not exceed the clock period. Multi-cycling is a well-known optimization technique, where selected combinational paths are permitted to have *longer* delays than the clock period, P, specifically, path delays as long as $n \times P$, where n is an integer greater than 1. Such relaxed constraints are possible for paths whose computed signal values are not required/used in the immediately subsequent clock cycle(s), and whose inputs can be held steady in registers for n cycles. Combinational paths with such relaxed constraints are said to be *multi-cycle paths*. As an example, consider a circuit with a longest path delay of $10ns$, and a second-longest path delay of $8ns$. Without multi-cycling, the circuit's minimum clock period is $10ns$ (100MHz). However, if the worst-case path can be multi-cycled and permitted 2 cycles to "complete", then the circuit's minimum clock period is reduced to $8ns$ (125MHz), as two clock cycles ($16ns$ total) is long enough to accommodate the original $10ns$ worst-case path.

HLS is a particularly opportune stage of the flow to both discover and create multi-cycling opportunities, as a key responsibility of HLS algorithms is to determine which computations to schedule in certain clock cycles, defining a finite-state machine (FSM). This means that, within HLS, one has complete visibility regarding the paths whose computed data is not needed in the subsequent cycle, and may therefore be multi-cycled. Conversely, consider a circuit described at the RTL or gate level, wherein it is difficult to automatically infer the paths that may be multi-cycled. In addition to HLS easing the discovery of multi-cycle paths, one can alter the HLS scheduling algorithms to deliberately create new multi-cycling opportunities – an approach we apply in this work.

While prior work has considered multi-cycling in HLS (e.g. [3]), a novel contribution of this work is software-profiling-driven scheduling for multi-cycling. The key concept is, before HLS commences, to profile the program-to-be-synthesized in software under a typical input dataset, gathering data about the number of times each program segment (basic block) executes. Then, during HLS scheduling, we dilate the hardware schedules corresponding to infrequently executed program segments. Lengthening their schedules has no appreciable impact on the overall circuit latency, due to their infrequent execution. However, the elongated schedules of such segments permit greater amounts of multi-cycling, assuring their associated hardware will not be on the critical path of the HLS-generated circuit. In an experimental study targeting the Altera Stratix IV FPGA [4], we show that the proposed methodology reduces circuit area-delay product by 20% on average and can provide circuit speedups in excess of 30%.

II. Background

Recent HLS research has shown that significant circuit speedups are possible using multi-cycle paths compared to fully pipelined paths [3]. For example, eliminating registers

978-3-9815370-4-8/DATE15/©2015 EDAA

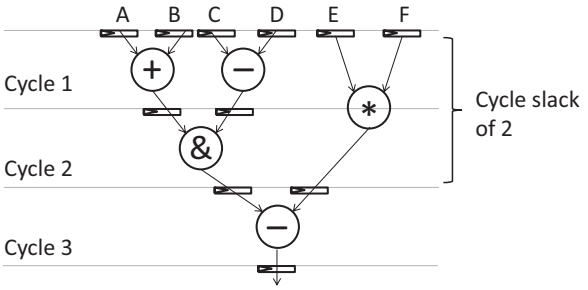

Fig. 1. A multi-cycle path. The multiplier can complete in two clock cycles and therefore has a multi-cycle slack of 2.

both saves area and removes several components of register-to-register delay, namely, clock-to-Q delay, setup time, clock skew. Multi-cycling also allows synthesis tools to optimize logic across what were previously register boundaries.

[3] takes advantage of these benefits by exploring circuit de-pipelining, replacing pipelined computations with multi-cycled computations. They develop an efficient algorithm to discover and constrain all multi-cycle paths in a circuit, while considering every reachable circuit state. However, [3] does not report the speedup offered by multi-cycling in isolation; rather, it presents results for multi-cycling combined with bit-level optimizations and control-flow graph restructuring.

More recently, [10] observed circuit speedups with an iterative HLS flow that uses place and route delay estimates. Multi-cycling was not the focus of that work however, with only simple constraints generated that resulted in marginal speedups.

Our contribution is to go beyond static circuit analysis for finding multi-cycling opportunities by using software profiling to guide schedule changes in HLS that create new multi-cycling opportunities. To our knowledge this has not been considered by prior work, and can be combined with other approaches such as [10] to leverage multi-cycling by exploiting visibility both into the delays of an HLS circuit as well as infrequently exercised computational paths.

III. Static Multi-Cycle Path Analysis

Multi-cycle paths occur frequently in HLS-generated circuits when an operation completes more than one cycle before its result is consumed – we refer to such a path as having *multi-cycle slack*. A naturally occurring multi-cycle path is illustrated in Fig. 1, in which the multiplication E*F, not required until clock cycle 3, is allowed two cycles to complete.

Our work is implemented in the LegUp open-source HLS tool developed at the University of Toronto [5], which is implemented within the LLVM compiler framework [6]. Within LLVM, the program is represented in an intermediate representation (IR), resembling assembly code. LegUp performs the traditional HLS steps on the IR: allocation, scheduling, binding, and HDL generation. Two compiler-related concepts are necessary to understand the proposed multi-cycling approach: 1) a *basic block* is a straight-line segment of code with a single entry point (at the beginning) and a single exit point (at the end); and 2) a *PHI* instruction is a control-flow related instruction within the LLVM IR that essentially implements a multiplexer selecting one of several values, depending which basic block executed prior to the block containing the PHI. For example, consider a basic block C, which may be reached by either of basic blocks A or B. A PHI allows a value in C to be set according to whether control transferred from A or from B.

After scheduling in HLS, analysis of the LegUp schedule is automatically performed to identify all instances of multi-cycle slack and print constraints for Altera's Quartus II Timing Analyzer, indicating to the timing analyzer that the paths are permitted extra cycles. This is done by traversing all register-to-register paths post-scheduling, including paths scheduled across basic blocks, and calculating cycle delays.

In addition to analyzing naturally occurring multi-cycle paths, datapath de-pipelining is performed to create additional multi-cycle opportunities. In general, hardware generated by LegUp is pipelined and able to process new data each clock cycle, including loops, dividers and floating point operations. However, instruction-level parallelism inferred from the C source does not always permit an initiation interval of 1 for these pipelines. While circuits can accept new data each clock cycle, this is not always observed in C benchmarks, which describe algorithms sequentially and are at times translated into hardware where only one or a few hardware units are active at a time. Because datapaths do not always benefit from pipeline parallelism, an alternative is to fully de-pipeline datapaths and instead designate them as multi-cycle paths (of equivalent latency). In such cases, the data feeding these multi-cycle paths is held in registers for multiple clock cycles.

A. De-pipelining Algorithm

The first step in de-pipelining is to identify where registers should remain, either to maintain functional correctness or where fully combinational logic would lead to unacceptable speed degradations. These cases are summarized here:

1. Block RAMs: Block RAMs in FPGAs are never combinational; rather, they always have registers at the inputs (and optional registers at the outputs). Load operations are therefore multi-cycle path sources, while store operations are multi-cycle path destinations. In addition, loads act as multi-cycle destinations when the load address calculation completes in multiple clock cycles, terminating at the address input port.
2. FSM state registers: We cannot eliminate the FSM state registers. However, next-state logic driving the FSM state registers is permitted to operate across multiple clock cycles and benefit from de-pipelining. Such next-state computations correspond to branch-related computations in the software, and these next-state computations can be multi-cycled when there exists cycle slack between the computation of a value and the clock cycle when it is used within a branching condition.
3. Function calls: LegUp creates separate Verilog modules for each C function. The module inputs (function arguments) and the module output (function return value) are registered, meaning that call operations act as both sources or destinations for multi-cycle paths.
4. Basic block boundaries: Multi-cycle paths may span basic block boundaries, in that a path starting in one basic block may terminate in another basic block. However, registers are still included for PHI operations and for computations whose results are consumed (used) in a different

basic block. Motivation for this decision is discussed at the end of this section.

5. Pipelined/shared hardware: For multi-cycling to work correctly, the data at the input of the multi-cycled path must be held in registers for the duration of the path – multiple clock cycles. Shared hardware that receives/produces new data every cycle cannot be part of a multi-cycle path. An example of this relates to memory: while LegUp supports local RAMs for data local to a function, globally accessible data is stored in shared memory. Shared memory uses dual-port RAMs, which implies at most two memory operations per clock cycle. It is common for new data to be read each clock cycle, which does not permit global RAMs to directly drive multi-cycle paths. This problem is solved by storing each loaded value feeding multi-cycle paths in a unique register, as discussed in Section V.

These five comprise all cases of multi-cycle path sources and destinations, and we refer to these as *path separators*. Separators correspond to nodes in the program's control-dataflow graph (CDFG), as represented by the LLVM IR. Path separators form the start/end points of multi-cycle paths.

After HLS scheduling, the location of each of the path separators is stored, and a traversal is performed over the control-dataflow graph (CDFG) representing the circuit to identify each separator-to-separator pair, print the appropriate multi-cycle timing constraint, and remove all intermediate, unneeded registers. The traversal is implemented as a series of backwards depth-first searches, each starting at a separator and traversing backwards to find all separators preceding it. Pseudocode is shown in Algorithm 1: For each separator S, fix this separator as a multi-cycle destination. Then, traverse backwards along all paths rooted at S until another separator is reached along each path. The first separator found along each path is the multi-cycle path source, while S is the multi-cycle path destination.

```
1  for each separator S do
2      Current = S
3      for all drivers D of Current in CDFG do
4          if D is a separator then
5              Store the path from D→S
6          else
7              Remove any register between D and Current
8              Repeat steps 3-8 for Current = D (recursion)
9          end
10     end
11 end
```

Algorithm 1: Identifying all multi-cycle paths.

Following the traversal, all multi-cycle paths have been stored (step 5 in Algorithm 1) and we print constraints for Altera's Quartus II Timing Analyzer, indicating to the timing analyzer that the paths are permitted extra cycles. By default, Synopsys multi-cycle timing constraints are of the following form (shown here for setup time) [7]

```
set_multicycle_path -from [get_registers { source_separator }] -to [get_registers { dest_separator }] -setup <slack #>
```

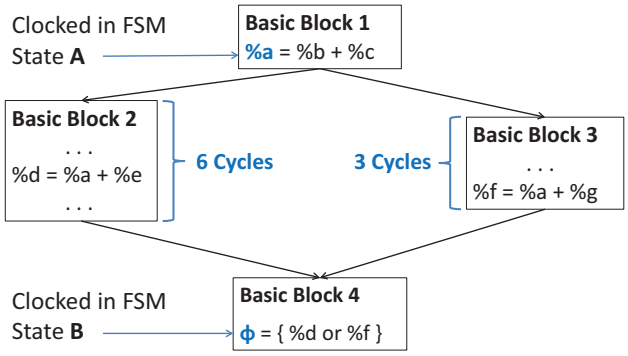

Fig. 2. Multi-cycle paths across basic blocks can have unbalanced latencies despite starting and terminating at the same registers.

Two such constraints are printed for each separator-to-separator pair, one for setup and one for hold. For a path with multi-cycle slack of n ($n \geq 2$), its setup slack is specified as n and its hold slack as $n-1$ (which instructs the timing analyzer to take the nth edge as the capturing edge and move the hold check back to the launch edge [11]). Because scheduling information is available for each operation, it is known exactly in which FSM state every register is enabled (registers use clock enable circuitry and a given register is only enabled in certain FSM states). The multi-cycle slack for a path is thus calculated as the number of FSM states between the source and destination separators being clocked (enabled). An area of caution is that multi-cycle paths may cross basic blocks. This complicates slack calculation because paths contain branches, which can lead to unbalanced path latencies.

For example, consider the circuit with two multi-cycle paths shown in Fig. 2. The operation %a is calculated in basic block 1, and then used in both basic blocks 2 and 3. Because %a is used in a basic block other than the basic block where it was defined, the result of operation %a is stored in a register (indicated in bold blue), during FSM state A. This registered value is then used in additional computations, for example computation %d in basic block 2 and computation %f in basic block 3, depending on the branch outcome from basic block 1. Finally, both computations %d and %f are used in basic block 4, requiring a PHI operation (multiplexer) at the input of basic block 4. The result of this PHI is then also stored in a register, during FSM state B, resulting in two multi-cycle paths in this circuit: both paths begin at register %a and terminate at the PHI register, with one path through basic block 2 and the other through basic block 3.

The goal is to calculate the multi-cycle slacks for these two paths, i.e. the number of FSM states between states A and B. Because the intermediate basic blocks have different latencies, the state difference depends on the branch taken: if the branch through basic block 2 is taken, then the FSM will reach state B six states (cycles) after A. If the branch through basic block 3 is taken, the FSM will reach state B three states after A. Thus, the two multi-cycle paths have *unbalanced latencies* of six and three, despite sharing source and destination separators.

Such scenarios of unbalanced latencies are common and cannot be addressed by the timing constraints outlined above. If a *single* multi-cycle slack constraint is printed for every separator-to-separator pair, then when that pair has *multiple* paths with unbalanced latencies the *shortest* latency of all these

paths must be selected to prevent timing violations (indeed, setup violations were observed during timing simulation when the minimum slack is not used). This makes higher-latency paths between these two registers critical in the overall circuit.

One solution is to restrict paths across basic blocks – if multi-cycle paths cannot span basic blocks, then unbalanced latencies are not a problem. However, we observed that the greatest speedups attained through multi-cycling are due to cycle-slack analysis across basic blocks. The other extreme is not allowing any registers between basic blocks (for example, not even with PHI operations), however this would lead to an exponential number of such unbalanced paths, as the depths of the multi-cycle paths increase. Another solution is in between these extremes, in which pipeline registers are judiciously added back into the circuit to "break up" such conflicting paths. A more obvious solution we investigated was to apply `-through` constraints on these paths. Multi-cycle timing constraints may specify a set of nodes through which the path must pass in order for the constraint to be applied:

```
set_multicycle_path -from [get_registers {
source_separator }] -to [get_registers { dest_separator
}] -setup <slack #> -through [get_nets { net_1 net_2 }]
```

This seems like a perfect solution, as it allows separate slacks to be applied for each signal. Unfortunately, the later synthesis and technology mapping steps may optimize away such intermediate signals. For example, such an intermediate signal may be "covered" within a LUT in the technology mapping, and therefore made invisible to static timing analysis tools. Because we cannot rely on the synthesis tool to keep the particular signal names, one option is to add Altera's `synthesis keep` attribute to all intermediate signals. While directives can be added to preserve intermediate wires, this introduces additional delays and increases circuit area. This is discussed further in Section V.

IV. Profiling-Driven Multi-Cycling

Extending the latency of a multi-cycle path requires modifying the control logic (FSM) that manages when the values computed by the path are consumed. This can be very useful in manual circuit design, wherein the latency (in cycles) of a critical path can be increased slightly, leading to a small increase in overall circuit latency, but ensuring that the path is no longer critical. We devised a method to achieve the same result in HLS, using software profiling.

Profilers, such as `llvm-prof` (built into the LLVM compiler [6]) indicate which parts of a circuit (program) will be used less frequently. Consider the example software program in Fig. 3(a). This program has a conditional branch, which either executes basic block B or C. Software profiling reveals that on a standard execution of this program with a typical input dataset, branch B is taken 3% of the time, while branch C is taken 97% of the time. Once this software program is converted to hardware by HLS (Fig. 3(b)), the software profiling data indicates that computations performed by the circuitry corresponding to B will only be used by sub-circuit D ∼3% of the time. If additional latency is then added to sub-circuit B, it will have only a minor impact on the overall circuit's total execution cycles. However, extending the latency of computations (i.e. the schedule) in sub-circuit B ensures that the critical path of the overall circuit is not within B.

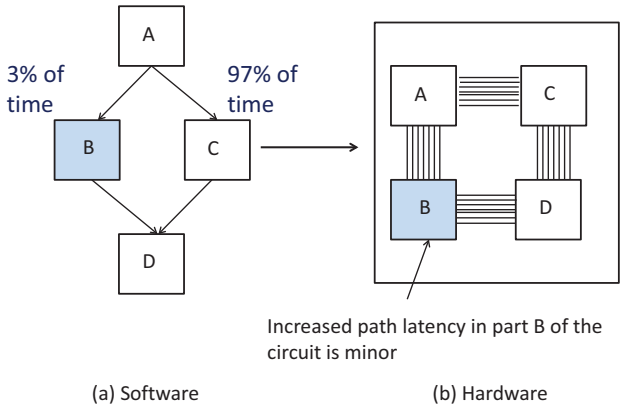

Fig. 3. An illustration of software profiling, in which branch B executes infrequently and therefore comprises a small percentage of total execution cycles in this circuit.

Extending the latency of infrequently executed basic blocks is achieved by specifying a cutoff frequency, `FREQ_THRESHOLD`, and identifying all basic blocks whose execution frequency is less than this threshold. The execution frequency of a basic block is reported by the LLVM profiler, and is defined as a percentage: the number of executions of a *specific* basic block divided by the total number of executions of *all* basic blocks For example, if a software run executes 100 basic blocks in total (dynamically), then a basic block which executed five times has a frequency of 5%. The latency of all paths in these basic blocks is then extended by a fixed amount, e.g. 1 clock cycle.

The distribution of basic blocks by execution frequency for one CHStone benchmark circuit [8], `dfmul`, is shown in Fig. 4a. The impact of software profiling with various cutoff frequencies applied to `dfmul` is shown in Fig. 4b. As the cutoff frequency increases, more and more basic blocks fall below this threshold and more multi-cycle paths become dilated. This increases both clock frequency and overall latency (# of clock cycles to complete). The key is identifying when the $FMax$ gains exceed the latency gains by the greatest amount, as shown in Section V.

The scheduling algorithm in LegUp is based on the system-of-difference (SDC) formulation, which formulates scheduling mathematically as a linear program (LP) [9] that can be handled by a standard solver. In the SDC scheduling formulation, variables are introduced for each computation in the CDFG and correspond to the state number in which the computation will execute. LP constraints are introduced to enforce dependencies between computations (i.e. ensure that dependent operations are scheduled later than their dependencies), as well as to enforce clock period and resource constraints.

A modification to the linear program allows the scheduled state of any operation to be extended arbitrarily, by constraining an operation to be scheduled a number of cycles later than its original cycle number assignment (in an initial ASAP scheduling). If this is done for an operation which is a multi-cycle path destination, then the latency of the entire path has been extended. For example, if a store operation is currently constrained to have `state` $\geq$ `4` within its basic block, then the latency of a multi-cycle path terminated at this store can be extended by one cycle by applying the constraint `state` $\geq$

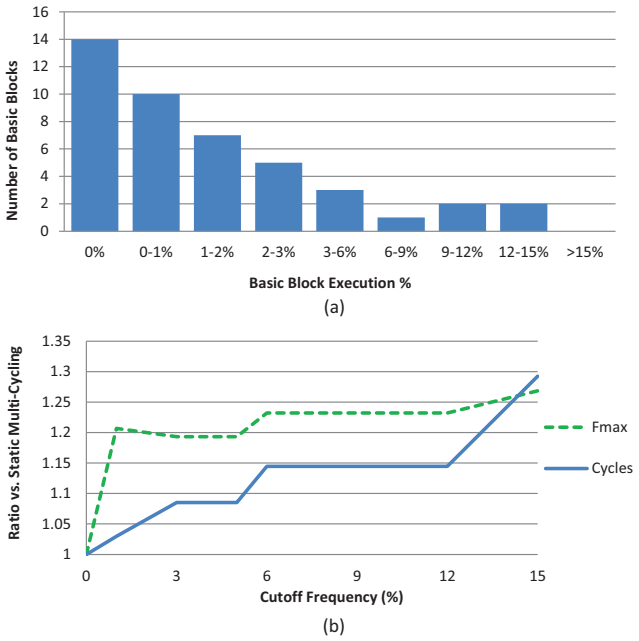

Fig. 4. (a) The distribution of basic blocks by frequency is shown here for the `dfmul` benchmark. In nearly all benchmarks, the majority of basic blocks fall below the 3% execution threshold. (b) The impact on $FMax$ and clock cycle latency when profiling-driven multi-cycling is applied to `dfmul` with varying cutoff frequencies.

5. This is summarized in Algorithm 2. Note that the combined run-time of algorithms 1 and 2 is a few seconds for all circuits.

The (path separator) operations in infrequently executed basic blocks which are pushed to later cycles include store operations, function calls, and operations used across basic blocks. In addition, whenever a basic block contains an operation whose state has been delayed, the basic block also has its total latency extended by the same amount. This is done by delaying branch and return instructions, which terminate these basic blocks. Note that line 8 of Algorithm 2 specifies a $\geq$ rather than an equality because sometimes an equality constraint is not possible, for example if a resource-constrained operation (such as a memory operation) is pushed to a state which already contains the maximum number of such operations.

We experimented with the algorithm parameters and observed that while a cutoff frequency in excess of 5% provided large increases in $FMax$, the increased latency became too high to justify the clock frequency speedups. Additionally, we tried dilating paths as a function of their basic block frequency, for example extending paths in basic blocks of frequency 1% by 2 cycles and paths in basic blocks of frequency 2-3% by 1 cycle. However, the extra latency rarely provided additional $FMax$ and the best results were obtained by extending paths by 1 cycle with a frequency cutoff of 1-3%, i.e. with `calculate_delayed_state` just incrementing the current state of S by 1.

V. Experimental Study

Static multi-cycling and profiling-driven multi-cycling were applied to the CHStone HLS benchmark suite [8] and dhrystone. These benchmarks have built-in correctness checking and input vectors representative of a standard workload. The

```
1  Schedule all operations using SDC ASAP formulation
2  Get all path separators and their scheduled state
3  for all path separators S do
4      B = the basic block containing S
5      freq = get_execution_frequency(B)
6      if freq < FREQ_THRESHOLD then
7          new_state = calculate_delayed_state(S, freq)
8          Add LP constraint: state[S] ≥ new_state
9      end
10 end
11 Solve the new LP
```

Algorithm 2: Re-scheduling to extend latencies.

HLS scheduling was constrained to a target period of $6ns$ for all circuits, which we observed to produce the lowest time-area product both with and without multi-cycling. Experiments target the Altera DE4 board which contains the Stratix IV FPGA. Following HLS, benchmarks were synthesized, placed and routed using the Quartus II software version 11.1. Post-routing timing-driven simulation was executed to verify correctness of the schedule changes and multi-cycle constraints.

Table I shows the wall-clock time (cycle latency $\times$ clock period) (in μS), $FMax$ (in MHz), cycle latency (in # cycles), and area (Stratix IV ALMs (adaptive logic modules)) for three flows. As a baseline, all circuits were generated using the latest version of the LegUp HLS tool, to which the multi-cycling optimizations were then applied. This baseline is shown in the first set of columns (base). The second set of columns then shows circuit speedups with static multi-cycling (static MC). The final set of columns uses profiling-driven multi-cycling (profiling-driven MC), where for each circuit we ran the algorithm with a cutoff of 1%, 2%, and 3%. All cutoff frequencies gave similar results but depending on the distribution of basic blocks some frequencies worked best for different circuits. Future work includes determining this cutoff automatically per-circuit by analyzing the histogram in Fig. 4a.

As expected, not all circuits benefit equally from multi-cycling due to critical paths not being in the datapath. For example if there are large delays associated with FSM logic or handling memory accesses then multi-cycling cannot provide speedups, and in fact adding additional latency attempting to improve data paths will slow circuits down. In these circuits (`jpeg`, `dfsin`) the cutoff frequency of 0% was chosen, i.e. falling back to static multi-cycling.

On the other hand, datapath-critical circuits speed up as much as 30% from multi-cycling and an additional 17% from software profiling. We observed that software profiling gives its greatest speedups when the most timing-critical computations occur in infrequently executed basic blocks. Notice from the geomeans that on average software profiling raises $FMax$ by $\sim$13% over the baseline, while only increasing latency by 1%. Software profiling works well because it achieves clock frequency speedups with very *little increase* to cycle latency, as opposed to *very large* frequency speedups with a *substantial* increase in latency. To see why this is, refer again to the example in Fig. 4. Because most basic blocks execute infrequently, the largest "gap" between $FMax$ and cycles occurs at low frequency cutoffs, i.e. $FMax$ "shoots upwards" sooner than latency does because most of the basic blocks are concentrated at these low frequencies.

TABLE I
SPEED PERFORMANCE AND AREA RESULTS.

	Base				Static MC				Profiling-Driven MC			
	Time	FMax	Cycles	ALMs	Time	FMax	Cycles	ALMs	Time	FMax	Cycles	ALMs
adpcm	178.8	146	26104	6351	126.1	207	26104	5599	118.1	226	26698	5664
aes	57.5	163	9372	8691	65.5	143	9372	7720	59.2	159	9410	7709
blowfish	981.2	190	186428	7010	981.2	190	186428	5447	981.2	190	186428	5447
dfadd	3.0	249	746	2071	3.0	251	746	1969	2.5	302	764	2007
dfdiv	9.1	215	1956	4343	7.2	272	1956	3229	6.8	289	1970	3192
dfmul	1.1	254	272	1478	0.9	292	272	1325	0.8	353	280	1358
dfsin	388.2	152	59132	12170	396.9	149	59132	11401	396.9	149	59132	11401
gsm	29.0	204	5906	4734	29.0	204	5906	4555	29.0	204	5906	4555
jpeg	12271.5	102	1251692	15488	10979.8	114	1251692	14159	10979.8	114	1251692	14159
mips	23.6	264	6228	1646	22.2	281	6228	1523	21.9	285	6254	1519
motion	33.7	250	8420	2047	33.4	252	8420	1932	32.2	263	8460	1939
sha	1120.1	229	256500	5903	1176.6	218	256500	4982	1107.8	232.72	257805	5058
dhrystone	31.0	250	7760	2693	30.3	256	7760	2530	29.9	268	8020	2541
geomean	70.79	198.33	14042.93	4420.12	66.90	209.90	14042.93	3934.18	63.57	223.18	14187.82	3953.23
ratio	1.000	1.000	1.000	1.000	0.945	1.058	1.000	0.890	0.898	1.125	1.010	0.894

Some circuits exhibited the unbalanced paths discussed in Section A. We circumvent this problem by adding a check for excessively unbalanced paths and aborting the multi-cycle flow if detected. This triggered for both `blowfish` and `gsm`, which remain on par with the baseline. The check did not trigger for `aes` however, which became 14% slower with static multi-cycling. Profiling optimizations then improved the `aes` wall-clock time by 11%, bringing it back towards the baseline. Empirically, we disable multi-cycling when two paths sharing source- and destination-separators have a slack difference of 40 cycles or more. We considered a more aggressive cutoff (e.g. to "catch" aes) but in fact most circuits show some unbalanced paths and we prefer a more relaxed constraint to generalize well to other circuits (i.e. not "over-fit" to our benchmarks).

Multi-cycling also reduces total register usage by 26% (geomean) due to de-pipelining. Combinational logic remains flat (decreases by 1%), resulting in a total area reduction of 11% (Stratix IV ALMs). Area-delay product is therefore improved by 20% on average using profiling-driven multi-cycling relative to the baseline. Profiling-driven multi-cycling consumes 0.4% more ALMs than static mult-cycling, due to additional FSM logic. One optimization to yield additional area savings is a more judicious allocation of registers on memory output ports. Because loads from global memory occur nearly every clock cycle in most circuits, the loaded values cannot be fed into multi-cycle paths. This problem was solved by inserting a unique register between the memory output port and each multi-cycle path, but an enhancement would be to only insert these registers when the loaded values will not persist for the duration of their multi-cycle.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we considered multi-cycling within high-level synthesis for FPGAs and proposed the idea of using software profiling to guide scheduling changes to improve multi-cycling results. Specifically, our approach extends the hardware schedules for code segments that are infrequently executed in a bid to raise $FMax$ with little increase to cycle latency. Results for an Altera Stratix IV FPGA showed mean 10% wall-clock time improvement, with datapath-critical circuits speeding up in excess of 30% from static multi-cycling and an additional 17% from software profiling. Area was reduced by 11%, producing an area-delay product improvement of 20%.

Future work involves combining multi-cycling with loop pipelining, considering whether the $FMax$ increase provided by multi-cycling could justify higher loop initiation intervals.

REFERENCES

[1] *OpenCL for Altera FPGAs*, http://www.altera.com/products/software/opencl/opencl-index.html.

[2] *Xilinx: Vivado Design Suite*, http://www.xilinx.com/products/design_tools/vivado/vivado-webpack.htm.

[3] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral level multi-cycle path analysis," in *IEEE FPL*, Porto, Portugal, 2013, pp. 1–8.

[4] *Stratix-IV Data Sheet*, Altera, Corp., 2014.

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

[6] *The LLVM Compiler Infrastructure Project (http://www.llvm.org)*, LLVM, 2014.

[7] *Altera: SDC and TimeQuest API Reference Manual*, http://www.altera.com/literature/manual/mnl_sdctmq.pdf.

[8] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, no. 0, pp. 242 – 254, 2009.

[9] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *IEEE/ACM DAC*, 2006, pp. 433–438.

[10] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen, "Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs," in *ACM/SIGDA FPGA*, 2014, pp. 1–10.

[11] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs*, Springer, 2009, pp. 260–272.