Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis

Ritchie Zhao, Mingxing Tan, Steve Dai, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{rz252, mingxing.tan, hd273, zhiruz}@cornell.edu

Abstract

Traditional techniques for pipeline scheduling in high-level synthesis for FPGAs assume an additive delay model where each operation incurs a pre-characterized delay. While a good approximation for some operation types, this fails to consider technology mapping, where a group of logic operations can be mapped to a single look-up table (LUT) and together incur one LUT worth of delay. We propose an exact formulation of the throughput-constrained, mapping-aware pipeline scheduling problem for FPGA-targeted high-level synthesis with area minimization being a primary objective. By taking this cross-layered approach, our technique is able to mitigate the pessimism inherent in static delay estimates and reduce the usage of LUTs and pipeline registers. Experimental results using our method demonstrate improved resource utilization for a number of logic-intensive, real-life benchmarks compared to a state-of-the-art commercial HLS tool for Xilinx FPGAs.

1. Introduction

Over the past few years, high-level synthesis (HLS) has emerged as a powerful tool for managing the increasing size and complexity of hardware designs. HLS allows engineers to build circuits using behavioral-level constructs, resulting in improved productivity and time-to-market over traditional register-transfer level (RTL) design flows. HLS is particularly useful for applications that repeatedly execute one or more compute heavy kernels inside loops, which are widespread in domains such as signals processing, cryptography, and machine learning. As a consequence, one of the most widely implemented techniques in HLS is pipelining, which synthesizes a datapath that allows successive iterations of a loop or function to execute before the current iteration has finished. This improves the throughput of the final circuit at a low resource cost.

Traditional pipeline synthesis typically uses a software compilation technique known as modulo scheduling [18] to generate a static schedule for a single loop iteration which can then be repeated at an interval known as the *initiation interval (II)*. The schedule plays a singular role in HLS by determining the requisite number of pipeline stages and inserting register boundaries into the untimed code. One major weakness of the existing modulo scheduling technique is its lack of awareness of the downstream synthesis optimizations such as logic synthesis and technology mapping. Each operation is typically assumed to incur a set delay based on the operation type (add, multiply, etc.), which is approximately the case when targeting general-purpose processors. However, this assumption fails to consider low-level optimizations when the target is a LUT-based FPGA. In particular, it is possible to map a large logic network to a few levels of LUTs during the technology mapping step. We illustrate how the existing approach results in an overly conservative schedule using the following example.

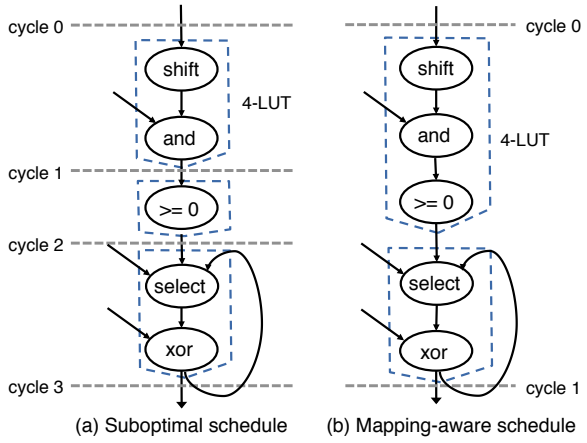


Figure 1: Pipeline schedule for a Reed-Solomon encoder: Target clock period is 5ns; each logic operation or LUT incurs a 2ns delay. (a) Suboptimal schedule requires 3 LUTs and 3 pipeline stages; (b) Optimal schedule requires 2 LUTs and 1 pipeline stage.

Figure 1 shows the data-flow graph (DFG) of a kernel in Reed-Solomon encoding [1], a prominent error-correction algorithm. For illustrative purposes we will assume the FPGA uses 4-input LUTs, the target clock period is $5ns$, and the target II is one cycle. Existing scheduling techniques assume an additive delay model based on pre-characterized delays, and generate a three-stage pipeline (left). In practice, however, it is possible to map the entire kernel to only two LUTs, which can be chained combinationally in one cycle (right).

Crucially, it is not possible to obtain the optimal schedule using a traditional flow where pipeline synthesis and technology mapping are performed in separate steps. Given the mapping-agnostic delay estimates, the modulo scheduler will pessimistically insert registers to meet timing. Downstream technology mapping must then respect these register boundaries and is unable to shorten the pipeline. To bridge the illustrated QoR gap, the scheduling algorithm must be made aware of the underlying LUT-based hardware and potential mapping optimizations.

In this paper we present a mixed integer linear programming (MILP) formulation of modulo scheduling to perform mapping-aware pipeline synthesis. Our method allows the MILP to select the optimal mapping for each operation to minimize LUT and register utilization. While an ILP is inherently unscalable to large designs, our primary goal is to demonstrate the improvements in area that can be realized using this cross-layered pipelining approach compared to a state of the art commercial HLS tool for Xilinx FPGAs. Experiments performed on a set of real-life benchmarks from a va-

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2744769.2744801

riety of domains show the applicability of our ideas. More specifically, our main contributions are as follows:

1. We are the first to propose an exact formulation of pipeline scheduling which is capable of minimizing resource usage, including LUTs and registers, under a throughput constraint while considering technology mapping.
2. We study a diverse set of real-life applications and demonstrate the considerable gap in QoR between a state-of-the-art commercial HLS tool and our MILP solution.

The rest of this paper is organized as follows: Section 2 examines previous work on pipeline scheduling and technology mapping algorithms, as well as recent efforts to improve pipeline synthesis in the context of HLS; Section 3 presents our cut enumeration algorithm and MILP formulation; We report experimental results in Section 4 and conclude the paper in Section 5.

2. Related Work

Modulo scheduling is a well-known compiler optimization technique to realize software pipelining [18]. It is also extensively used in the HLS context for enabling loop and function pipelining. For example, state-of-the-art HLS tools such as LegUp [4] and Vivado HLS [10] make use of a mathematical programming formulation know as system of difference constraints (SDC) to support modulo scheduling for hardware pipelining synthesis [22, 3]. Other developments to incorporate memory port reduction [2], pipeline flushing [12], polyhedral analysis [15], and multithreading [20] continue to advance the pipelining capabilities of HLS.

For an FPGA, technology mapping is the process of covering a network of logic gates with LUTs [6]. While some prominent mapping techniques include FlowMap [8], CutMap [9], and DaoMap [5], there exists numerous mapping techniques that optimize for LUT depth [8] or area [11, 5]. Meanwhile, Pan et al. proposed a retiming-based technology mapping technique that considers mapping for register repositioning to achieve the minimum clock period [17, 16]. Area-efficient mapping is an important step in achieving good QoR in the physical implementation flow. Unfortunately, while latency-optimal technology mapping can be achieved in polynomial time [8], area-optimal mapping is NP-hard [14]. An ILP-based algorithm for minimum-area LUT mapping is proposed in [7]. Unlike our approach, however, it eschews the usage of cuts. Recent research has focused on integrating mapping with the upstream tool flow to explore additional optimizations in high-level analysis. In particular, Zheng et al. propose a flow that iterates between upstream scheduling and downstream mapping and place-and-route and uses post-physical implementation timing for re-scheduling [23]. Most recently, Tan et al. propose a constrained scheduling algorithm that considers depth-optimal LUT mapping [19]. However, this work does not address pipelined designs and focuses on minimizing latency.

3. Mapping-Aware Modulo Scheduling

The key idea behind our approach is to perform pipeline scheduling while taking into consideration LUT mapping. To accomplish this, we extend the technique of cut enumeration from conventional technology mapping algorithms into the HLS domain. A brief description of traditional cut enumerations is as follows.

Let v be a node on a graph representing a bit-level logic network. Then we define O_v, a *cone* of v, as a sub-graph of v and its predecessors such that there exists a path from any node in O_v to v that is entirely contained within O_v. The cut of O_v, denoted as C_v, is then defined to be the set of nodes not in O_v with an edge pointing to a node in O_v. A cone and its associated cut is defined to be K-feasible if the cut contains K or fewer nodes. Because a K-input LUT can implement any K-bounded logic network, a K-feasible cone can be mapped to a single K-input LUT.

Cut enumeration is the process of identifying the set of all K-feasible cuts for every node in the graph. Afterward this is done, the technology mapping problem becomes one of selecting a set of cuts whose cones cover each node in the graph, while minimizing some objective such as total latency or LUT area.

Typically, cut enumeration is done on a bit-level directed acyclic graph (DAG). However, the pipeline scheduling problem operates on a word-level control data flow graph (CDFG). Our approach is to use a modified cut enumeration algorithm to find the cut set of each operation, and construct an MILP using the cut information which simultaneously schedules each operation while selecting an optimal set of cuts which covers the CDFG. Our MILP performs modulo scheduling, which attempts to schedule one iteration of a loop to meet a target initiation interval. More formally, here is our area-minimizing modulo scheduling problem formulation:

Given: (1) A CDFG for a function or loop whose edges capture inter-iteration and intra-iteration data dependences between operations; (2) A target clock period T_{cp}; (3) A target initiation interval II; (4) A set of constraints C including latency constraints, cycle time constraints, and resource constraints; (5) Characterized delays for operations on a target FPGA device using K-input LUTs.

Goal: Find a minimum area modulo schedule for the operations so that no constraints in C are violated, and within each cycle, there exists a feasible K-input LUT mapping that meets T_{cp}.

3.1 Word-Level Cut Enumeration

An intuitive approach to word-level cut enumeration is to break down the word-level DFG into a bit-level graph [21] and use a traditional method. Tan et al. proposed a word-level cut enumeration algorithm, but only for simplicity purposes since the scheduling approach in their paper can handle both bit-level and word-level graphs [19]. In this work, bit-level decomposition would generate an enormous number of cuts and make an MILP approach intractable. A word-level cut enumeration algorithm is therefore necessary, and we present the technique in detail below.

Bitwise operations such as `AND/OR/XOR` are straightforward because the operations on different bits are completely independent. The key challenge arises for non-bitwise operations, such as shifting or arithmetic, where a single bit of the output might depend on multiple bits of each input operand. For instance, given an addition operation $out[1:0] = in_1[1:0] + in_2[1:0]$, the most significant output bit $out[1]$ would depend on four input bits: $in_1[0]$, $in_1[1]$, $in_2[0]$, and $in_2[1]$, coming from two nodes in_1, in_2 on the DFG. To address this problem, we use a bit-level dependence tracking technique on the word-level DFG. Instead of just identifying dependent values, our algorithm also tracks all dependent bits of each value. To limit our analysis to those operations which are mapped to LUTs, we define a *black box (BB)* operation as one which does not map to LUTs, (i.e., memory access operations). The following applies then to all non-BB nodes in the DFG.

Let $v[j]$ denote the a bit j of the operand v. We then classify all operations into three classes, and define the DEP function for each class of operations as follows:

- Bit-wise operations (`AND/OR/XOR`): each output bit only depends on a single bit of each input operand. For example, the DEP for operation $out = in_1 \ \& \ in_2$ is defined as: $DEP(out[j]) = \{in_1[j], in_2[j])\}$.
- Shifting operations (`LSFHIT/RSHIFT`): each output bit depends on a shifted single bit of each input operand. For example, the DEP for operation $out = in_1 >> s$ is defined as: $DEP(out[j]) = \{in_1[j+s]\}$.

Algorithm 1: $CutGen(CDFG)$

input : $CDFG$ – control data flow graph
output: CUT – cut set for all CDFG nodes
`// Initialize the trivial cut for each node.`
foreach *node* v *in* $CDFG$ **do**
  $CUT_v = \{\{v\}\}$
$L \leftarrow$ list of CDFG nodes in topological order
`// Iteratively update cut set`
while $L \neq \Phi$ **do**
  get the head node v from L
  if v *is not a primary input or black box operation* **then**
    $newCutSet = mergeCuts(v)$
    if $newCutSet \neq CUT_v$ **then**
      $CUT_v \leftarrow newCutSet$
      append v's successors to L

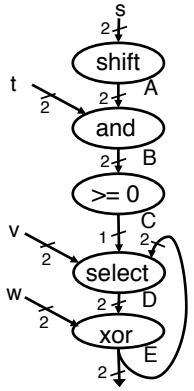
- Arithmetic operations (`ADD/SUB/CMP`): each output bit can depend on multiple bits of each input operand. For example, the DEP for operation $out = in_1 + in_2$ is defined as follows: $DEP(out[j]) = \{in_1[j], in_1[j-1], ..., in_1[0], in_2[j], in_2[j-1]..., in_2[0]\}$.

The DEP function over a word-level value $DEP(v)$ is further defined as the union set of $DEP(v[j])$ for each bit $v[j]$. Based on the DEP function, we compute the K-feasible cut set for each node in the DFG by merging the K-feasible cuts for all of its inputs. Suppose v's inputs are u_1, u_2, ..., u_p, with associated cut sets CUT_{u_1}, CUT_{u_2}, ..., CUT_{u_p}, where CUT_u is a collection of cuts and each cut $C_i \in CUT_u$ is K-feasible. The K-feasible cut set for v can be computed as follows:

$$\begin{aligned} &CUT_v = mergeCuts(u_1, ..., u_p) = \\ &\{C' = \bigcup_{C_i \in CUT_{u_i}} \{DEPS(C_i)\}, \text{if } |C'| < K\} \\ &\text{where } DEPS(C_i) = \bigcup_{t \in C_i} \{DEP(t)\} \end{aligned} \tag{1}$$

Our cut enumeration algorithm iteratively applies Equation (1) to each node until all K-feasible cuts are obtained. Algorithm 1 lists the pseudocode of our cut enumeration algorithm. We maintain a work list for nodes that need to be updated. Initially, the work list contains all operations, and the cut set for each node v is the trivial cut $\{\{v\}\}$. For each node in the work list, we apply Equation (1) to compute the new cut set. If a new cut is added for a node, we update its cut set and add all its successors to the work list. We remove a node from the work list each time it is visited. The algorithm terminates when the work list becomes empty. Note that for each black-box operation, we simply force its cut set to be its trivial cut. Previous studies have shown that cut enumeration is an exponential algorithm with respect to K [11]. Nevertheless, the actual runtime for cut enumeration is typically very fast as the value of K is small in practice ($K \leq 6$).

Figure 2 demonstrates the cut enumeration for the example listed in Figure 1. The original Reed-Solomon application uses 32-bit operations, but for simplicity, we use 2-bit operations in this figure. Cut enumeration for A and B are relatively simple. Each bit of A depends on a single shifted bit of input s, while each bit of B depends on a single bit of each input operand t and A. In general, operation C would be treated as an arithmetic operation, but in this example, the comparison "`B >= 0`" is actually testing whether the most significant bit is zero or one. In this case, our algorithm will identify that the output of C only depends on the highest bit of each input based on bit-level dependence tracking. Our algorithm can also handle the cycle which arises from a loop-carried dependence when processing nodes D and E.



CUTS[A[i]] = { {s[i-1]} }
CUTS[B[i]] = { {A[i], t[i]},
{s[i-1], t[i]}
CUTS[C[0]] = { {B[1]},
{t[1], A[1]},
{t[1], s[0]} }
CUTS[D[i]] = { {C[0], v[i], E[i]},
{B[1], v[i], E[i]},
{t[1], A[1], v[i], E[i]},
{t[1], s[1], v[i], E[i]},
{C[0], v[i], w[i], D[i]},
{B[1], v[i], w[i], D[i]},}
CUTS[E[i]] = { {W[i], D[i]},
{W[i], v[i], C[0], E[i]},
{W[i], v[i], B[1], E[i]},
{W[i], v[i], C[0], D[i]},
{W[i], v[i], B[1], D[i]} }

Figure 2: Cut enumeration for the Reed-Solomon decoder.

3.2 MILP Formulation of Modulo Scheduling

We formulate the modulo scheduling problem as a mixed integer linear program (MILP). Given a word-level dependence graph G and the cut set CUT_v of each graph node v computed via Algorithm 1, our formulation aims to compute an area-efficient pipelined schedule while respecting the following constraints:

LUT cover constraints – For each cut of v, we create a binary variable $c_{v,i}$, denoting whether cut i is selected for node v. We also define the binary variable $root_v$ as the sum of $c_{v,i}$ for all i. Conceptually, if $root_v = 1$ then v is the root of a cone that will be mapped to a LUT. Otherwise v will be mapped inside the cones of other nodes. Note that by the definition of $root_v$, the MILP can only select one cut for each node. Equation (3) ensures that each primary output (PO) is a root node, while Equation (4) ensures that the inputs of a selected cut are themselves root nodes.

$$root_v \in \{0, 1\}$$

$$root_v = \sum_i c_{v,i} \ \ \forall v \tag{2}$$

$$\sum_i c_{v,i} = 1 \ \ \forall v \in \{POs\} \tag{3}$$

$$c_{v,i} \leq root_u \ \ \forall u \in CUT_v[i] \tag{4}$$

Dependence constraints – The MILP uses a set of binary scheduling variables $s_{v,t}$, $0 \leq t \leq M$ where M is the bound on pipeline latency, to denote whether operation v is assigned to cycle t. Equation (5) below constrains each operation to a single clock cycle, and Equation (6) defines S_v, an integer variable whose value is the actual cycle assigned to v.

$$\sum_t s_{v,t} = 1 \tag{5}$$

$$S_v = \sum_t t * s_{v,t} \tag{6}$$

Given a dependence between operations u and v with a distance $dist_{u,v}$ [1] , we need to guarantee that operation u in iteration k is

[1] A distance of 0 indicates an intra-iteration dependence, while a non-zero distance indicates an inter-iteration dependence.

finished before we start the operation v in iteration $k + dist_{u,v}$. This can be captured by Equation (7):

$$S_u - S_v - II \cdot dist_{u,v} \leq 0 \tag{7}$$

Cycle time constraints – To consider the possibility of combinationally chaining multiple operations in a clock cycle, the MILP also assigns to each variable a start time L_v within the cycle, Note that L_v is a real number with bounds $0 \leq L_v \leq T_{CP}$, where T_{CP} is the target clock period. Suppose d_v is the delay of operation v in nanoseconds, Equation (8) ensures a combinational circuit will not across clock boundaries:

$$L_v + d_v \leq T_{CP} \tag{8}$$

For each dependence between nodes u and v, we must make sure u will not start later than v if they are scheduled in the same cycle. Equation (9) ensures this constraint:

$$(S_u - S_v - II \cdot dist_{u,v}) * T_{CP} + (L_u - L_v + c_{v,i} \cdot d_u) \leq 0 \quad \text{where } u \text{ is in } CUT_v[i]. \tag{9}$$

Essentially, Equation (9) enforces a difference between L_u and L_v if and only if u and v are scheduled in the same cycle (i.e., when $S_v - S_u - II \cdot dist_{u,v} = 0$). If the selected cut of v does not contain u ($c_{v,i} = 0$) then u and v will be constrained to have the same start time in the same cycle (i.e., $L_u = L_v$ and $S_u = S_v$), and will thus be mapped into the same LUT.

Register constraints – Register usage is captured by considering the liveness of each operation. We define binary variables $live_{v,t}$ to denote whether the result of operation v is live on cycle t. We shall show that $live_{v,t}$ can be calculated using binary variables $def_{v,t}$ and $kill_{u,t}$ and adding the constraint in Equation (12) for each u in $CUT_v[i]$:

$$def_{v,t} = \sum_{z=0}^{t-\lfloor d_v/T_{CP} \rfloor} s_{v,z} \tag{10}$$

$$kill_{v,t} = \sum_{z=0}^{t} s_{v,z} \tag{11}$$

$$def_{u,t} - kill_{v,t} - (1 - c_{v,i}) \leq live_{u,t} \tag{12}$$

Here $def_{v,t} = 1$ if the results of v is available on or before cycle t, and $kill_{v,t} = 1$ if the inputs of v are killed on or before cycle t. Thus Equation (12) ensures $live_{u,t} = 1$ on each cycle t where the results of u is defined and at least one fanout of u has not yet executed. The term $(1 - c_{v,i})$ ensures that if $CUT_v[i]$ is not selected, the liveness constraint between u and v is inactivated.

Using $live$ we can define the maximum number of registers needed on each cycle m, taking into account that operations separated by exactly II cycles will execute concurrently in a pipeline:

$$Reg(m) = \sum_{t \in T'} \sum_{v \in V} Bits(v) * live_{v,t} \quad \text{where } T' = \{t \mid t \bmod II = m\} \tag{13}$$

Here $Bits(v)$ is the number of bits in the value produced by v.

Resource constraints – In this work we consider resource constraints only for black-box operations. Because the cut set for such operations will always contain only the trivial cut, we do not need to consider mapping for them. The resource constraints in our MILP are therefore identical to those in a canonical modulo pipelining formulation [18]. Let r denote a resource type, X_r the number of that resource used for the design, N_r the number of that resource available, and $R(v)$ the resource type for a black-box operation v. Then for each r:

$$\sum_{t \in T'} \sum_{v \in V'(r)} s_{v,t} - X_r \leq 0, \forall m : 0 \leq m < II \quad X_r \leq N_r \quad \text{where } T' = \{t \mid t \bmod II = m\} \quad \text{and } V'(r) = \{v \mid R(v) = r\} \tag{14}$$

Objectives – The objective of the MILP is to minimize the weighted sum of the number of root nodes and the number of registers:

$$minimize \quad \alpha \cdot \sum_{v \in V} Bits(v) * root_v + \beta \cdot \sum_{m=0}^{II-1} Reg(m) \tag{15}$$

Here α and β are user-defined parameters to trade-off the optimizations on LUT and register usage. Note that our MILP formulation can be easily extended to consider other type of resources such as embedded DSP blocks.

4. Experimental Results

Our setup leverages a popular commercial HLS tool which compiles C/C++ into Verilog or VHDL targeting Xilinx FPGAs. The tool uses LLVM as its front-end compiler, and we implemented our technique as an additional LLVM pass written in C++ which is applied after compilation and other optimizations, but before the tool performs scheduling. By reordering instructions and inserting wait statements, we are able to enforce a custom schedule. No other changes were made to the program, and we were able to verify from the synthesis report that our schedule was honored by the HLS tool. The rest of the HLS flow (binding, etc.) was unchanged. We used IBM ILOG CPLEX as the MILP solver, and Xilinx Vivado 2014.3.1 as the tool to implement the generated RTL. All timing and area numbers below were obtained post place and route.

To model the delays of each operation for scheduling, we back annotated delay values parsed from the schedule report of the HLS tool for the black-box operations. To allow our experiments to finish in a timely fashion, we restricted the MILP solver to run for at most 60 minutes and return the best solution found. For all benchmarks, a feasible solution was found in this amount of time. However, this means that not all schedules we generated were optimal. The values of α and β were set to 0.5 in all experiments.

In order to make a fair evaluation of our mapping-aware pipeline synthesis approach, we have included three sets of results for each benchmark: the commercial HLS tool, the MILP without mapping consideration (referred to hereon as MILP-base), and the full MILP (referred to hereon as MILP-map). MILP-base is implemented by skipping the cut enumeration step, and assigning to each node only the trivial cut. We show results for two MILPs to understand how much of the measured improvement is due to the consideration of mapping and how much is due to the exact nature of the MILP versus the heuristic algorithm used by the commercial HLS tool.

Table 1 displays the name and information about each of our benchmarks. We divided the benchmark set into two classes:

1. **Kernels** - Compute intensive loops or functions commonly used in applications. These benchmarks are almost entirely composed of logical and arithmetic operations.
2. **Applications** - Complete real-life applications with practical use from a diverse set of domains. These benchmarks are more complex and contain more black-box operations.

Each benchmark is fully pipelined to an II of 1. For the kernels, the entire function can be executed on the resulting datapath once

Table 1: Resource usage comparison: Target clock period is 10ns. CP = achieved clock period; LUT = # of look-up tables; FF = # of flip-flops. The percentages next to each column is calculated relative to the HLS tool.

Design	Domain	Description	Method	CP(ns)	LUT	%	FF	%
CLZ	Kernel	Count the number of leading zeros in a 64-bit value	HLS Tool	5.43	171		221	
			MILP-base	4.29	152	(-11.1%)	226	(+2.3%)
			MILP-map	5.55	99	(-42.1%)	43	(-80.5%)
XORR	Kernel	XOR reduction for an array of elements	HLS Tool	5.55	3394		257	
			MILP-base	5.55	3394	(+0.0%)	257	(+0.0%)
			MILP-map	4.59	3264	(-3.8%)	0	(-100.0%)
GFMUL	Kernel	Efficient Galois field multiplication	HLS Tool	1.64	41		27	
			MILP-base	1.69	44	(+7.3%)	28	(+3.7%)
			MILP-map	3.36	39	(-4.9%)	0	(-100.0%)
CORDIC	Scientific Computing	Coordinate Rotation Digital Computer	HLS Tool	8.24	1313		631	
			MILP-base	5.19	1663	(+26.7%)	646	(+2.4%)
			MILP-map	7.58	1220	(-7.1%)	298	(-52.8%)
MT	Scientific Computing	Mersenne Twister pseudorandom number generation	HLS Tool	5.70	681		843	
			MILP-base	6.03	623	(-8.5%)	842	(-0.1%)
			MILP-map	7.17	640	(-6.0%)	526	(-37.6%)
AES	Cryptography	Advanced Encryption Standard	HLS Tool	5.27	4860		4720	
			MILP-base	5.33	4564	(-6.1%)	5316	(+12.6%)
			MILP-map	5.55	4475	(-7.9%)	2441	(-48.3%)
RS	Communication	Reed-Solomon decoder	HLS Tool	8.71	6493		8206	
			MILP-base	6.45	7308	(+12.6%)	7114	(-13.3%)
			MILP-map	9.26	6656	(+2.5%)	3856	(-53.0%)
DR	Machine Learning	Digit recognition using k-nearest neighbours algorithm	HLS Tool	5.29	1264		1365	
			MILP-base	5.55	1070	(-15.3%)	1088	(-20.3%)
			MILP-map	5.15	963	(-23.8%)	999	(-26.8%)
GSM	Communication	Global system for mobile communications	HLS Tool	7.76	1706		1231	
			MILP-base	8.53	1543	(-9.6%)	1074	(-12.8%)
			MILP-map	9.83	1766	(+3.5%)	493	(-60.0%)

per cycle. For the applications, the input data set is processed at the rate of one element per cycle.

4.1 Kernel Results

In this section we present the results for the three kernels. CLZ counts the number of leading zeros in an integer. GFMUL computes a Galois-Field product of two integers using efficient shifts and logical operations. Finally, XORR performs xor operations over an array of inputs.

The kernels saw essentially no improvement in QoR between the HLS tool to MILP-base and significant reductions in resource usage from between the HLS tool to MILP-map. In CLZ MILP-map cut down the pipeline latency from 7 to 1, reducing the number of FFs required by 81% compared to MILP-base. In GFMUL and XORR, MILP-map was able to recognize that the entire pipeline can be implemented in a single combinational stage, eliminating registers altogether. MILP-map was also able to decrease the number of LUTs used by 19% over the three kernels.

To explain how MILP-map obtained such enormous FF savings, we will look at XORR in detail. XORR specifies an xor reduction over an array, which was optimized by the HLS tool into a reduction tree with depth 9. From the scheduling reports we found that the HLS tool assigns a delay of 1.37 ns to each xor operation, and generated a 2-stage pipeline since the critical path through the tree contains 9 xor operations. MILP-base generates an identical schedule. By considering mapping, however, MILP-map saw that multiple xor operations can be mapped into a LUT, and that the critical path easily fits within a single cycle. The FFs saved are precisely the pipeline registers removed through this optimization, which can be very significant in the case of a wide pipeline such as a reduction tree. Conservative delay estimates in existing tools produce sizable timing slack in each of the kernels, and our mapping-aware algorithm exploits this to pack more operations into each cycle, thus shortening the overall pipeline and reducing resource counts.

4.2 Applications Results

Here we examine the results for practical applications. Note that RS utilize GFMUL as a kernel in its computations.

The most convincing results appear in CORDIC, MT, and AES, where MILP-map was able to reduce flip-flop registers by an average of 46% over the HLS tool. In contrast, MILP-base was unable to achieve any register savings on these designs, and in fact incurred an average penalty of 4.8%. MILP-base was able to show some improvement in RS, DR, and GSM compared to the commercial tool, obtaining an average FF saving of 16%. But even on these designs, MILP-map achieved superior QoR, achieving a further reduction in FFs of 39% over MILP-base. Once again, the results show that the area savings realized by MILP-map were due to new optimizations opened up through consideration of mapping, not the differences between a heuristic and an MILP scheduling approach. Some of the variation in the results of MILP-base on the application benchmarks could be attributed to imprecise delay estimates, as we were not able to obtain a delay from the HLS tool for every operation.

4.3 MILP Runtime and Practicality

Table 2 shows the size of each benchmark as well as the runtime of the MILP for each benchmark using both MILP-base and MILP-map. CPLEX performs presolve optimizations such as substitution and eliminating redundant constraints, and we noted that the runtime scaled primarily with the number of unique constraints in the MILP. The number of constraints is based on the total number of

cuts enumerated in the DFG, and is the primary reason MILP-map is much slower than MILP-base.

Table 2: Runtime of CPLEX for each benchmark, not including cut enumeration or ILP construction. Zeros indicate runtime too short to be measured.

	MILP Solver Runtime (s)		
Design	**LLVM Instrs**	**MILP-base**	**MILP-map**
CLZ	387	9.0	27.6
XORR	2047	0.0	1622.0
GFMULT	86	0.0	0.0
CORDIC	304	3.7	41.7
MT	236	5.3	3602.0
AES	1809	314.1	3600.0
RS	2503	0.2	1.8
DR	282	3600.0	3603.0
GSM	324	107.6	3603.0
Mean	813.2	404.3	1641.8

The resource constrained pipeline scheduling problem is known to be NP-hard, and exact formulations have always been deemed unscalable, with nearly all commercial algorithms being heuristic in nature. This efficiency gap is reflected in our data, with the commercial tool finishing in seconds compared to MILP-map's 60 minutes on some designs. Nevertheless, it would not be unreasonable to use MILP-map to trade-off runtime for improved design quality in the case of small kernels. Similar observations about the viability of exact ILP methods in scheduling are made in [13].

5. Conclusions and Future Work

We present an area-efficient pipeline synthesis approach for HLS which uses a word-level cut enumeration technique and an MILP formulation to perform mapping-aware modulo scheduling. We test our algorithm on a variety of kernel benchmarks and practical applications, obtaining improved LUT and FF usage compared to both a state-of-the-art commercial HLS tool and a mapping-agnostic MILP approach. While an MILP formulation is unscalable in general, we have limited the runtime of the solver to 60 minutes, which is tolerable given the non-trivial improvements in QoR. Future work includes incorporating mapping awareness into a scalable heuristic pipeline scheduling algorithm as well as investigating other logic synthesis optimizations such as exploring different logic decompositions of the circuit during mapping.

6. Acknowledgements

This work was supported in part by NSF CAREER Award #1453378, NSF XPS Award #1337240, and a research gift from Xilinx, Inc.

References

[1] A. Agarwal, M. C. Ng, and Arvind. A Comparative Evaluation of High-Level Hardware Synthesis Using Reed–Solomon Decoder. *IEEE Embedded Systems Letters*, 2(3):72–76, 2010.

[2] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 3(3):15, 2010.

[3] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, Sep 2014.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 33–36, Feb 2011.

[5] D. Chen and J. Cong. DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 752–759, 2004.

[6] D. Chen, J. Cong, and P. Pan. FPGA Design Automation: A Survey. *Foundations and Trends in Electronic Design Automation*, 1(3):139–169, 2006.

[7] A. Chowdhary and J. P. Hayes. Area-Optimal Technology Mapping for Field-Programmable Gate Arrays Based on Lookup Tables. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(7):999–1013, 2005.

[8] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(1):1–12, 1994.

[9] J. Cong and Y.-Y. Hwang. Simultaneous Depth and Area Minimization in LUT-based FPGA Mapping. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 68–74, 1995.

[10] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, Apr 2011.

[11] J. Cong, C. Wu, and Y. Ding. Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 29–35, 1999.

[12] S. Dai, M. Tan, K. Hao, and Z. Zhang. Flushing-Enabled Loop Pipelining for High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2014.

[13] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Author Retrospective for Optimum Modulo Schedules for Minimum Register Requirements. *Int'l Conf. on Supercomputing*, pages 35–36, 2014.

[14] A. H. Farrahi and M. Sarrafzadeh. Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(11):1319–1332, 1994.

[15] A. Morvan, S. Derrien, and P. Quinton. Polyhedral Bubble Insertion: a Method to Improve Nested Loop Pipelining for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32(3):339–352, 2013.

[16] P. Pan, A. K. Karandikar, and C. Liu. Optimal Clock Period Clustering for Sequential Circuits with Retiming. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 17(6):489–498, 1998.

[17] P. Pan and C.-C. Lin. A New Retiming-Based Technology Mapping Algorithm for LUT-based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 35–42, 1998.

[18] B. R. Rau. Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture (MICRO)*, pages 63–74, Nov 1994.

[19] M. Tan, S. Dai, U. Gupta, and Z. Zhang. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2015.

[20] M. Tan, B. Liu, S. Dai, and Z. Zhang. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 718–725, 2014.

[21] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong. Bit-level Optimization for High-Level Synthesis and FPGA-Based Acceleration. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 59–68, 2010.

[22] Z. Zhang and B. Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 211–218, 2013.

[23] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen. Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 1–10, 2014.