Using Dynamic Dependence Analysis to Improve the Quality of High-Level Synthesis Designs

Rafael Garibotti, Brandon Reagen, Yakun Sophia Shao, Gu-Yeon Wei and David Brooks
Harvard University
Email: {rgaribotti, reagen, shao, guyeon, dbrooks}@eecs.harvard.edu

***Abstract*—High-Level Synthesis (*HLS*) tools that compile algorithms written in high-level languages into register-transfer level implementations can significantly improve design productivity and lower engineering cost. However, HLS-generated designs still lag handwritten implementations in a number of areas, particularly in the efficient allocation of hardware resources. In this work, we propose the use of dynamic dependence analysis to generate higher quality designs using existing HLS tools. We focus on resource sharing for compute-intensive workloads, a major limitation of relying only on static analysis. We demonstrate that with dynamic dependence analysis, the synthesized designs can achieve an order of magnitude resource reduction without performance loss over the state-of-the-art HLS solutions.**

I. Introduction

With the end of Moore's Law on the horizon, hardware designers and architects have turned their attention to hardware accelerators. By committing certain functions to specialized hardware, designers are able to increase energy efficiency by orders of magnitude, making accelerators a promising way to delay the end of technology scaling and continue scaling performance. As the industry enters the era of specialization, there is a clear need to improve hardware design productivity and reduce engineering costs. Hence, high-level synthesis (*HLS*) tools, which automatically transform high-level algorithms (e.g., written in C/C++) into register-transfer level (*RTL*) implementations, have been gaining momentum in both academia and industry.

Despite the increased maturity of HLS tools, several limitations still exist to generate efficient designs. One well-known challenge in high-level synthesis is the inability to adequately use available resources. Operations like multipliers can be expensive in terms of area and power. If a single functional unit (*FU*) can be shared across multiple operations without performance degradation, then a better quality of results (*QoR*) can be achieved. Unfortunately, efficient resource sharing within a given performance target is NP-hard, making it difficult to efficiently solve at synthesis time [1]. Exhaustively sweeping all possible resource sharing parameters exponentially increases the design space exploration time, making this exploration prohibitively expensive and detrimental to the productivity benefits of HLS. This shortcoming motivates the need for novel techniques to adeptly and intelligently guide the HLS tool toward higher QoR designs.

In this work, we propose the use of dynamic dependence analysis to profile the dynamic behavior of applications, specifically targeting resource allocation in compute-intensive workloads. This paper presents mechanisms to help generate a higher QoR in accelerator designs using existing HLS tools. We apply our technique to two state-of-the-art, commercial HLS tools to show that this problem is fundamental and not an artifact of a specific tool. We demonstrate how dynamic dependence analysis can provide hints for resource allocation constraints to generate more resource efficient designs without performance loss. Our evaluations show that with dynamic dependence analysis we can achieve an order of magnitude savings of resource allocation reduction for compute-intensive workloads.

II. Background and Motivation

A. Static and Dynamic Analysis for HLS

HLS uses a control data flow graph (*CDFG*) to represent workloads, and all the scheduling and synthesis phases utilize this information. A completely static approach tends to be conservative, as HLS tools are bound to generate correct hardware. One source of conservativeness comes from resource allocation, where static analysis tends to over provision hardware resources to ease scheduling and binding at the expense of implementation efficiency.

To better leverage dynamic information in the synthesis process, researchers have begun applying dynamic profiling information to generate better designs. Recent studies include the use dynamic information to multi-cycle infrequent branches and to better approximate dynamic value ranges to more aggressively reduce bitwidths [2], [3]. However, both of the approaches use traditional software profiling or software instrumentation, without special consideration of the hardware it targets. Our work extends the existing dynamic profiling for HLS because hardware characterization avoids numerous HLS iterations, which reduces the synthesis time.

B. Resource Sharing in HLS

1) HLS Synthesis Flow: A typical HLS flow decomposes the behavioral synthesis process into three basic stages: resource allocation, scheduling, and binding [4]. There have been significant efforts in each of these stages to optimize for resource sharing. However, there is a sequential dependency in this process. If not guided by the user, the resource allocation stage profiles the control data flow graph to estimate the required resources with little consideration for resource sharing, leading to a conservative estimation. Resource availability is an input to the scheduling stage, which formulates the

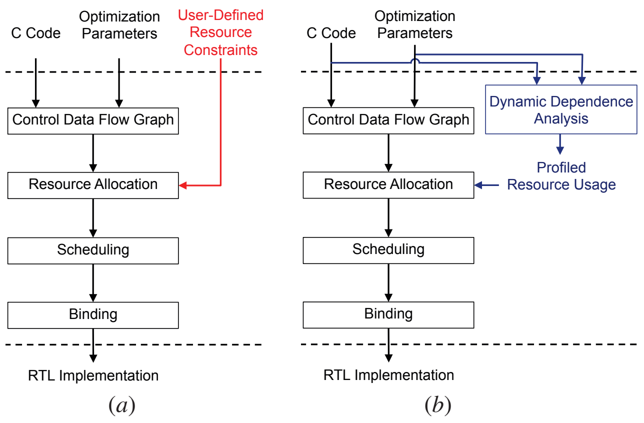

Fig. 1. Difference between (*a*) state-of-the-art and (*b*) proposed HLS synthesis flow, highlighting the dynamic dependence analysis added to HLS flow.

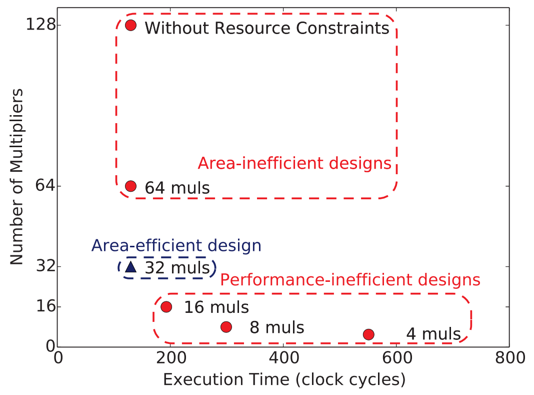

Fig. 2. HLS can over-provision functional units if resource constraints are not set or are set too high. On the other hand, overly aggressive resource constraints can lead to performance degradation.

scheduling problem as an optimization problem, using either linear integer programming [5] or a system of different constraints [6], to produce a legal schedule that satisfies various constraints. Though efficient for a small number of operations, the compilation time increases significantly when it comes to a large number of nodes in the graph, as the complexity of optimization process grows polynomially with the number of nodes to schedule [7]. The binding stage, where most of resource sharing optimizations happen [8], has very limited room to explore resource sharing opportunities, as scheduling has already been done. To achieve better resource sharing, iterative resource optimization process across these stages has also been proposed, but this leads to significantly longer compilation time [9].

One way to break such inter-dependency across these stages is to expose the resource allocation stage to the user, as shown in Figure 1.a. For example, commercial HLS tools including Vivado HLS, Catapult C, and Stratus HLS can take user-defined resource constraints for the number of functional units. If users set the resource constraints optimally, based either on detailed application knowledge [10] or handwritten-RTL reference designs [11], efficient resource sharing can be achieved with the same target performance.

However, exposing the resource allocation stage to the user alone does not solve the resource sharing problem, it just offloads the responsibility of finding the optimal resource requirement from HLS tools to users. To truely achieve better resource utilization, users have to exhaustively sweep all possible resource requirements in conjunction with other optimization parameters, which exponentially increases the size of the design space.

2) Impact of Resource Sharing: Figure 2 illustrates the impact of resource constraints for blocked matrix multiplication (*BB_GEMM*). All designs are generated using the same optimization parameters (loop unrolling and array partition set to 32 with pipelining). The only difference across these HLS-generated designs is the number of resources allocated. Without any user-defined resource constraints, HLS generates the largest design, using 130 multipliers. The remaining designs result from sweeping the number of multipliers from 64 to 4 via setting a resource allocation directive provided by HLS. We find that with only 32 multipliers, we can achieve the same performance as the original, over-provisioned design with 130 multipliers. However, if the resource constraint is set too low (from 4 to 16 multipliers in this case), we observe performance loss due to aggressive resource sharing.

III. Methodology

The design flow proposed in this paper is shown in Fig. 3 and can be spilt into three parts: (A) commercial tools, (B) an academic tool, and (C) pre-processing step.

A. Commercial Tools

FPGA target: We use a commercial available HLS tool that targets FPGA devices. All reported FPGA numbers are from a Virtex-7 FPGA (part number: xq7v585t).

ASIC target: To demonstrate our techniques' generality across different HLS tools and backends, we use a different HLS tool compatible with ASIC design flows. The ASIC design flow uses a commercial, 40nm CMOS technology.

All synthesized versions of RTL are simulated using Modelsim; simulation results provide performance numbers and validation for each design. FPGA area numbers come from the synthesis report while ASIC results are from Design Compiler.

B. Academic Tool

Aladdin is an accelerator modeling framework based on dynamic data dependence graph (*DDDG*) analysis [12]. In this work, instead of using Aladdin as an architectural simulator to estimate the power, performance, and area of a desired accelerator, we leverage the Aladdin-scheduled DDDG to guide designers to set resource constraints for compute-intensive workloads.

Unlike HLS, Aladdin *does not generate RTL*. Therefore, we modify the Aladdin-scheduled DDDG to profile specific information on a desired design and use it to create directives to guide HLS tools and to assist designers in generating more efficient designs. DDDG analysis is more than 40× faster than HLS [12], introducing negligable overheads when included as a pre-processing step to the HLS design process.

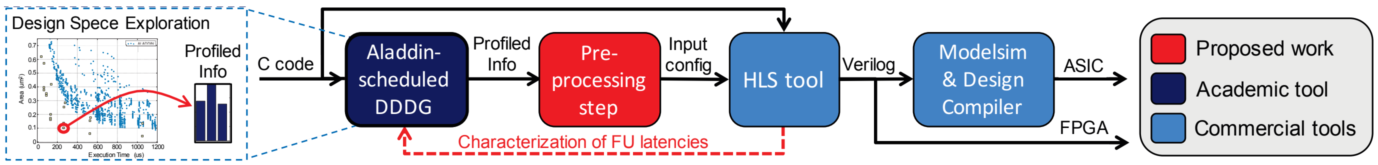

Fig. 3. Detailed design flow that integrates academic and commercial tools intended to improve the quality of HLS-generated designs.

C. Pre-Processing step

The pre-processing step aims to join academic models and commercial products to bring achievable benefits for HLS-generated designs. The pre-processing step consists of three different parts: (1) characterization of FU latencies, (2) DDDG optimization and scheduling, and (3) directives generation.

1) Characterization of FU latencies: A detailed characterization of operation latencies from our target platform are included in the Aladdin-scheduled DDDG to accurately model the FU latencies. Besides an adaptation of the original version [12], this is the key difference between using software profiling and our DDDG approach. While software profiling information provides the behavior of the used input algorithm, our approach also considers the operation frequency to determine the available FUs. This difference is crucial to generate an efficient design. For example, if a higher frequency is set, default single-cycle multiplier must be replaced by a multi-cycle multiplier, which directly impacts the scheduling and the chosen FU implementation.

2) DDDG Optimization and Scheduling: We take advantage of the fast simulation speeds to explore design spaces. After a quick evaluation of the algorithm, we choose the desired design point and extract the profile information produced by Aladdin-scheduled DDDG, as illustrated in Fig. 3 (left).

A DDDG is built through an LLVM instrumentation step that dumps a dynamic IR execution trace containing information like instruction opcodes, register IDs, data values, and memory addresses. Analysis of register and memory dependencies based on the trace yields an original DDDG that only contains the true read-after-write data dependencies. To bound the DDDG with more realistic dependence assumptions, the analysis framework also finds places in the graph where control flow and data dependencies cannot be disambiguated statically, and it adds edges (true dependencies) at these locations. Such manipulations to the original DDDG open various optimization opportunities at the node, loop, and memory level. Optimization parameters such as loop unrolling and pipelining are also taken into consideration here. The final optimized DDDG is the input to the DDDG scheduler.

The scheduling step consists of two parts: a forward step that minimizes the critical path and a backward step that rebalances the graph to optimize resource usage. While the final schedule is not necessarily *optimal*, it is sufficient to significantly constrain the search space by setting the upper bound of resource usage and generate a better design. We profile the rescheduled DDDG to find the maximum number of FU required at any single cycle. This is the optimistic resource usage estimate that we automatically provide for the HLS tool as a hint, as illustrated in Figure 1.b.

3) Directives Generation: Profile information extracted from Aladdin-scheduled DDDG is used to generate an input configuration (directives). Each of the commercial HLS tools have a different syntax, offering a wide number of possibilities to prune your design. Technical challenges lie in understanding these tools and create a proper directive that assists equally both HLS tools to generate an efficient design. These particular directives ensure integration with commercial HLS tools.

IV. Experimental Results

This section presents how dynamic-based analysis can assist HLS to increase the quality of HLS generated designs. We use MachSuite [13] to demonstrate the improvement efficient resources sharing has in compute-intensive workloads. The benchmarks are written in C, conforming to both commercial HLS standards as well as Aladdin.

Fig. 4 presents the area-saving opportunities exposed by our approach for *STENCIL-2D*. We evaluated two design spaces: first using a commercial HLS targeting ASIC platforms, including loop unrolling and array partition range from 1 to 64 with pipelining parameter to create our *baseline HLS*. Then, dynamic analysis hints are applied which set the minimum number of resources to allocate, called as *HLS with pre-processing analysis*. For illustration purposes, only Pareto points are presented in Fig. 4.

Exploiting the same design space, identical Pareto performance points are found (horizontal points), meaning that our hint-enhanced results do not degrade performance. With respect to area, we observe that for sequential designs, which are designs with few parallelism optimization parameters (right design points), the baseline HLS analysis can find the optimal resource constraint in the synthesis process. This is because such designs have little opportunity for resource sharing.

For the same reason, on small kernels like *TRIAD* has no improvement with our pre-processing step. This kernel has no loop carried dependencies, making the static analysis from HLS tools be powerful enough to handle it. This shows that the benefits of our proposal are tightly correlated with the complexity of the workloads and the parallelism of the designs.

Table I demonstrates our benefits in a realistic scenario. Here, we find examples where unoptimized designs would not fit within a FPGA, our optimized implementations could not only now fit but also incured no performance loss. These MachSuite workloads had loop unrolling and array partition set to 32 with pipelining.

Standard HLS produces designs with DSP slices above those available in the chosen FPGA. However, when the pre-processing step is applied, the same design fits into the FPGA. In addition, DSP slices savings are always bigger than FF or

TABLE I

FPGA RESOURCE SAVINGS FOR A HIGH PARALLEL CONFIGURATION USING HLS WITH PRE-PROCESSING ANALYSIS.

Benchmark	Model	DSP (Available: 1260)		FF (Available: 728400)		LUT (Available: 364200)	
		Total Used	Utilization	Total Used	Utilization	Total Used	Utilization
BB_GEMM	Baseline	2048	162.5 %	50438	6.9 %	40070	11.0 %
	Pre-Proc.	512	40.6 %	31877	4.4 %	49426	13.6 %
STENCIL-2D	Baseline	3240	257.1 %	34278	4.7 %	37882	10.4 %
	Pre-Proc.	1080	85.7 %	17670	2.4 %	72234	19.8 %
LAPLACIAN-FILTER	Baseline	2198	174.4 %	66768	9.2 %	69774	19.2 %
	Pre-Proc.	384	30.5 %	37624	5.2 %	77932	21.4 %
NNET	Baseline	16384	1300.3 %	200711	27.6 %	155674	42.7 %
	Pre-Proc.	256	20.3 %	22407	3.1 %	80026	22.0 %

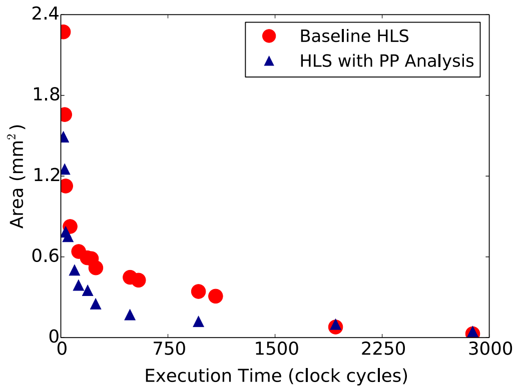

Fig. 4. Area-saving opportunities exposed by our approach for *STENCIL-2D*, highlighting that parallel designs are more affected by HLS with PP analysis.

LUT. For some workloads, like *BB_GEMM* and *STENCIL-2D*, LUTs present an increase in resource utilization, due to the cost of MUXes from resource sharing. When more FUs are being shared, more MUXes need to be introduced. However, as DSP slices are scarcer than LUT, our approach is justified by better balancing the available resources. On the contrary, when our approach is applied to ASIC design flow, the cost of additional MUXes are dwarfed by the cost of FUs.

Our lightweight pre-processing step has a direct impact on the overall C-to-RTL synthesis time. Besides the additional time of the dynamic analysis, which is almost negligible compared with HLS stages. For complex designs, it accounts for less than 1% synthesis time. The greatest impact occurs in scheduling and binding stages, which drastically changes the synthesis time. Results show an improvement in binding stage. However, this entails a worsening in the scheduling time. This is due to the increased complexity to assign control steps to operations subject to resource constraints. For example, to achieve the best area-savings in *NNET*, the scheduling stage has become too complicated. This made the the scheduling time be $36\times$ higher than binding time, causing an overall delay of 24.73 $\%$ at synthesis time.

In short, our proposed pre-processing step increases the quality of the generated accelerator designs by improving the efficiency of resource sharing using commercial HLS tools. However, for complex workloads where greater area-savings are expected, the synthesis time can be impaired.

V. Conclusions

Efficient resource sharing and allocation for compute-intensive workloads is still a challenge for HLS tools. To increase the quality of HLS-generated designs, we have proposed a lightweight pre-processing step that provides design hints and automatically profiles a resource estimate before invoking HLS tools. Results evaluated in two commercial HLS tools show up to $64\times$ DSP slice savings can be achieved without any performance loss. These results increase the acceptability of generated designs through HLS solutions.

Looking forwards, this work will pave the way for an automatic tool for standardization across existing HLS solutions.

Acknowledgment

This work was partially supported by CNPq, PUCRS and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and DARPA under Contract #: HR0011-13-C-0022. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] O. Arcas-Abella et al. *An empirical evaluation of High-Level Synthesis languages and tools for database acceleration*. In FPL, 2014, pp. 1–8.
[2] S. Hadjis et al. *Profiling-driven Multi-cycling in FPGA High-level Synthesis*. In DATE, 2015.
[3] M. Gort and J.H. Anderson. *Range and bitmask analysis for hardware optimization in high-level synthesis*. In ASP-DAC, 2013, pp. 773–779.
[4] S. Hadjis et al. *Impact of FPGA architecture on resource sharing in high-level synthesis*. In FPGA, 2012, pp. 111–114.
[5] C.-T. Hwang et al. *A formal approach to the scheduling problem in high level synthesis*. In TCAD, vol. 10, no. 4, pp. 464–475, 1991.
[6] Z. Zhang and B. Liu. *SDC-based modulo scheduling for pipeline synthesis*. In ICCAD, 2013, pp. 211–218.
[7] J. Cong and Z. Zhang. *An efficient and versatile scheduling algorithm based on SDC formulation*. In DAC, 2006, pp. 433–438.
[8] A. Canis, J.H. Anderson and S.D. Brown. *Multi-pumping for resource reduction in FPGA high-level synthesis*. In DATE, 2013, pp. 194–197.
[9] J. Cong, B. Liu and J. Xu. *Coordinated resource optimization in behavioral synthesis*. In DATE, 2010, pp. 1267–1272.
[10] P. Li et al. *Resource-Aware Throughput Optimization for High-Level Synthesis*. In FPGA, 2015, pp. 200–209.
[11] Available online: *http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf*.
[12] Y.S. Shao et al. *Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures*. In ISCA, 2014, pp. 97–108.
[13] B. Reagen et al. *MachSuite: Benchmarks for accelerator design and customized architectures*. In IISWC, 2014, pp. 110–119.