Lin-Analyzer: A High-level Performance Analysis Tool for FPGA-based Accelerators

Guanwen Zhong[1], Alok Prakash[1], Yun Liang[2], Tulika Mitra[1] and Smail Niar[3]
[1]School of Computing, National University of Singapore
[2]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China
[3]LAMIH, University of Valenciennes, France
{guanwen,tulika}@comp.nus.edu.sg, alok.prakash@outlook.com, ericlyun@pku.edu.cn,
smail.niar@univ-valenciennes.fr

ABSTRACT

The increasing complexity of FPGA-based accelerators, coupled with time-to-market pressure, makes high-level synthesis (HLS) an attractive solution to improve designer productivity by abstracting the programming effort above register-transfer level (RTL). HLS offers various architectural design options with different trade-offs via pragmas (loop unrolling, loop pipelining, array partitioning). However, non-negligible HLS runtime renders manual or automated HLS-based exhaustive architectural exploration practically infeasible. To address this challenge, we present Lin-Analyzer, a high-level accurate performance analysis tool that enables rapid design space exploration with various pragmas for FPGA-based accelerators without requiring RTL implementations.

1. INTRODUCTION

Field programmable gate arrays (FPGAs) have the advantages of reconfigurability, customization and energy efficiency. They are widely used in embedded domains such as automotive, wireless communications, and others that demand high performance with low energy consumption. The increasing capacity and improving power efficiency in latest FPGA devices, such as 16nm UltraScale+ from Xilinx and 14nm Stratix 10 from Altera, have also contributed to their adoption in high-performance computing domains such as data-centers. However, the architectural flexibility comes at the cost of complex hardware programming models, making FPGA development a challenging and time consuming task even as the time-to-market constraints continue to tighten.

High-level synthesis (HLS) tools have been developed, both in academia [4][5] and industry [1][17], to address this challenge by enabling automated translation of applications written in high-level languages (such as C/C++, SystemC, and others) to register-transfer level (RTL). But the various optimization options offered by HLS tools (e.g., loop unrolling, loop pipelining, array partitioning) make it non-trivial to choose appropriate options in synthesizing an application on an FPGA even with the assistance of HLS.

DAC '16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2898040

Hence, several authors [3][8][9][10][11][13][14] have proposed compiler-assisted static program analysis, similar to the commercial HLS tools, to estimate accelerator performance while exploring the large design space. However, static program analysis suffers from its inherently conservative dependence analysis [4][5][12] that can potentially lead to false dependences between the operations and limit the exploitable parallelism by the FPGA-based accelerators, ultimately introducing large inaccuracies in the predicted performance. In addition, to improve the estimation accuracy, some techniques rely on HLS tools to obtain performance for a few design points and then extrapolate for the rest. The time required in the HLS tools ranges from hours to days depending on the number of design points to be synthesized and the configuration choices.

We propose a dynamic analysis method that exploits runtime information to obtain true dependences between operations and hence accurately predicts the accelerator performance. This also obviates the need to use HLS tools, resulting in a rapid and reliable design space exploration (DSE) tool. In particular, our contributions are two-fold:

- We introduce a high-level analysis tool, Lin-Analyzer, to perform fast and accurate FPGA performance estimation as well as DSE according to different optimizations (loop unrolling, pipelining and array partitioning) without generating any RTL implementations.
- Lin-Analyzer can identify bottlenecks of different FPGA implementations when applying diverse optimizations. It can assist designers in evaluating different architectural options in the context of high-level synthesis and better understand the performance impact of different accelerator design choices.

The goal of Lin-Analyzer is to perform an early design space exploration and recommend the best suited optimization configuration for an application when mapped to FPGAs. The HLS tool should then be invoked with the suggested configuration to obtain the final synthesized accelerator. In other words, Lin-Analyzer is complementary to HLS tools. Experimental evaluation confirms that Lin-Analyzer returns the optimal recommendation while navigating complex design spaces within seconds to minutes.

2. RELATED WORK

Table 1 summarizes the state-of-the-art techniques used for performance estimation and architectural exploration of hardware accelerators based on either FPGAs or ASICs.

Table 1: Current State-of-the-art vs. Lin-Analyzer

Reference	Accuracy	Design Space Complexity	DSE Time	Static/ Dynamic	Target
Bilavarn [3] [TCAD'06]	Medium	Low (Loop Unrolling)	minutes	Static	FPGA
Boucle [10] [DSD'13]	Medium	Med (Loop Unrolling+ Array to registers)	minutes	Static	FPGA
Liu [8] [DAC'13]	High	High (Loop Unrolling+ Loop Pipelining+Array Partitioning (MemBW))	hours	Static+ HLS	ASIC
Zhong [14] [ICCD'14]	High	Low (Loop Unrolling+ dataflow)	minutes	Static+ HLS	FPGA
Pham [9] [DATE'15]	High	High (Loop Unrolling+ Loop Pipelining+Array Partitioning (MemBW))	hours	Static+ HLS	FPGA
Schafer [11] [TECS'12]	High	High (Loop Unrolling+ Loop Pipelining+Array Partitioning (MemBW))	hours	Static+ HLS	FPGA
So [13] [PLDI'02]	High	Low (Loop Unrolling+ Loop Tiling+data reuse)	minutes	Static+ HLS	FPGA
Shao [12] [ISCA'14]	High	High (Loop Unrolling+ Loop Pipelining+Array Partitioning (MemBW))	minutes (small input size)	Dynamic	ASIC
Proposed Lin-Analyzer	High	High (Loop Unrolling+ Loop Pipelining+Array Partitioning (MemBW))	seconds to minutes	Dynamic	FPGA

As the design space of hardware accelerators can get prohibitively large in the presence of the various pragma options, the existing works rightly focus on pruning this design space for rapid exploration. The majority of the current state-of-the-art techniques use static compiler-assisted program analysis to prune the design space [3][8][9][10][11][13][14]. Moreover, [3][10][13][14] consider only *loop unrolling* and ignore the other two prominent optimizations (*loop pipelining* and *array partitioning*) that have significant impact on accelerator performance and resource consumption. [8][14] use HLS tools to improve the accuracy of the predicted performance of the accelerators, while using machine learning techniques or regression analysis to further prune the design space. Most of these works have limited design space or only focus on one benchmark [8] in an effort to find optimal application-specific accelerators. [9][11] perform more extensive design space exploration, however the usage of commercial HLS tools in their frameworks significantly increases the exploration time to hours or even days in some cases. In addition, the reliance on static analysis introduces inaccuracies, to varying extent, in the performance prediction due to the lack of memory disambiguation information.

In [12], the authors propose a pre-RTL power-performance simulator for ASIC accelerators. We have similar goal. But as we target FPGA-based accelerators, the different kind of resources (DSP, BRAM) along with memory bandwidth bring in additional complexity during resource-constrained scheduling. In contrast, [12] only considers memory bandwidth (number of read/write ports per memory bank and number of memory banks) while using an ASAP-like scheduling algorithm. Moreover, instead of analyzing the entire trace, as in [12], Lin-Analyzer only focuses on relevant *sub-traces* that makes it very fast even for applications with relatively large input size. Lastly, Lin-Analyzer can also assist hardware designers in developing good quality FPGA-based accelerators by identifying potential application bottlenecks after performing a set of optimizations.

3. MOTIVATING EXAMPLE

Manual or automated HLS-based DSE is a time-consuming process due to non-negligible HLS runtime and large design space. Listing 1 shows the nested loop of Convolution3D kernel from Polybench suite [15]. We use a commercial HLS tool, Vivado HLS [17], to generate FPGA-based accelerators for this application with three different pragma combinations specified in Table 2. The HLS runtime varies from seconds to hours for different choices of pragmas. As the internal workings of the Vivado HLS tool is not available publicly, we do not know the precise reasons behind this highly variable synthesis time.

Listing 1: Convolution3D example

```
...
/* c11 to c33 are constant values of a window*/
loop1: for (i = 1; i < NI - 1; ++i) {
  loop2: for (j = 1; j < NJ - 1; ++j) {
    loop3: for (k = 1; k < NK -1; ++k) {
      B[i][j][k]=c11*A[i-1][j-1][k-1]+
                 c13*A[i+1][j-1][k-1]+c21*A[i-1][j-1][k-1]+
                 c23*A[i+1][j-1][k-1]+c31*A[i-1][j-1][k-1]+
                 c33*A[i+1][j-1][k-1]+c12*A[i+0][j-1][k+0]+
                 c22*A[i+0][j+0][k+0]+c32*A[i+0][j+1][k+0]+
                 c11*A[i-1][j-1][k+1]+c13*A[i+1][j-1][k+1]+
                 c21*A[i-1][j+0][k+1]+c23*A[i+1][j+0][k+1]+
                 c31*A[i-1][j+1][k+1]+c33*A[i+1][j+1][k+1];
    }
  }
}
```

Table 2: Xilinx Vivado HLS Runtime for Convolution3D

Input Size	Loop Pipelining	Loop Unrolling	Array Partitioning	Vivado HLS Runtime
32*32*32	Disabled	loop3 factor=30	A, cyclic, 2 B, cyclic, 2	44.25 seconds
	loop3, yes	loop3 factor=15	A, cyclic, 16 B, cyclic, 16	1.78 hours
	loop3, yes	loop3 factor=16	A, cyclic, 16 B, cyclic, 16	3.25 hours

Table 3: Convolution3D DSE: Exhaustive vs. Lin-Analyzer.

Input Size	Design Space	DSE Time	
		Exhaustive HLS-based	Lin-Analyzer
32*32*32	120	10 days*	29.30 seconds

*The HLS tool runs for a long time with few design points with complex pragmas and we terminate it after running for 10 days

Due to the long HLS runtime, exhaustive HLS-based DSE is infeasible as shown in Table 3. To assist designers in finding good-quality accelerator designs through appropriate pragma settings, it is vital to obtain performance estimations early in the design stage without generating RTL implementations. This also enables rapid architectural exploration. Hence, we develop Lin-Analyzer, a pre-RTL, high-level performance analysis tool for FPGA-based accelerators. Table 3 demonstrates that Lin-Analyzer can navigate the complex design space `Convolution3D` kernel with 120 design points under 30 sec.

4. THE LIN-ANALYZER FRAMEWORK

Lin-Analyzer leverages dynamic analysis and uses dynamic data dependence graphs (DDDGs) generated from program traces to represent the dataflow of the accelerator to be designed. A DDDG is a directed, acyclic graph in which nodes represent operations and edges denote data dependences between the nodes. DDDG representation is suitable for accelerator design [2][12]. Based on the DDDG, Lin-Analyzer can model performance of FPGA-based accelerators directly from algorithms written in high-level languages, such as C/C++, without generating RTL implementations.

4.1 Framework Overview

Lin-Analyzer framework is shown in Figure 1. It takes in a high-level specification (C/C++) of an algorithm without any modifications as input and generates its execution trace. According to optimization pragmas (loop unrolling, loop pipelining and array partitioning) provided by the designer, Lin-Analyzer extracts a *sub-trace* and generates the corresponding DDDG. After essential optimizations on the DDDG, Lin-Analyzer schedules the nodes with resource constraints to obtain an early performance estimate of the accelerator corresponding to the input algorithm. As it only

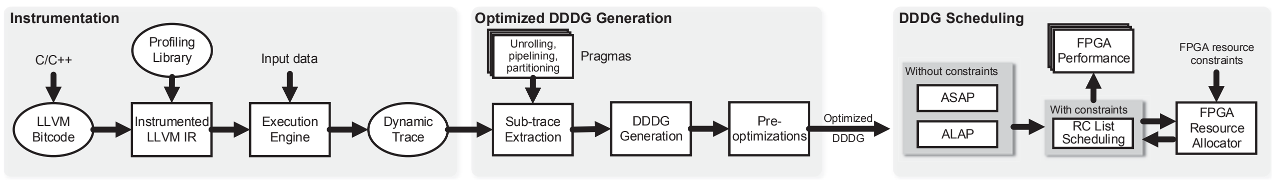


Figure 1: The Lin-Analyzer Framework

focuses on DDDG of the relevant sub-trace and uses fast scheduling algorithm, Lin-Analyzer has minimal runtime even for applications with relative large input size and complex optimization combination, for example, complete loop unrolling, pipelining and large array partitioning factors.

4.2 Instrumentation

An execution trace of the application specified in C/C++ is required for generating the DDDG. Lin-Analyzer leverages the Low-Level Virtual Machine (LLVM) [6] for instrumentation and trace collection. The core of LLVM is a Static Single Assignment (SSA) based Intermediate Representation (IR). The IR is machine-independent and uses unlimited virtual registers. Code analysis, optimization and modification are performed on IR via LLVM passes.

We implement an instrumentation pass by adding profiling functions to instrument the IR for trace collection. After instrumentation, the *Execution Engine*, an LLVM Just-in-Time (JIT) compiler integrated in Lin-Analyzer, is invoked to perform execution of the instrumented IR and generate run-time trace. The generated trace contains runtime instances of static instructions, including instruction IDs, opcodes, operands, virtual register IDs, memory addresses (for load/store instructions) and basic block IDs. A *Profiling Library*, developed in-house, is used to record the trace.

4.3 Optimized DDDG Generation

A dynamic trace typically contains millions or even billions of instruction instances. So Lin-Analyzer only focuses on a *sub-trace* based on the pragma settings, constructs a DDDG of the sub-trace, and optimizes it.

4.3.1 Sub-trace Extraction

Consider a nested loop $L = \{L_1, .., L_i, .., L_K\}$, where K is the number of loop levels in L and L_K is the innermost loop level. Loop unrolling can be applied at any loop level. We can handle both perfectly nested loops (e.g., Listing 1) and non-perfectly nested loops with statements between different loop levels. Given an unrolling factor tuple $\{U_1, .., U_i, .., U_K\}$, where U_i is the unrolling factor of loop L_i, we extract sub-trace of U_i iterations of L_i if all inner loop levels $\{L_{i+1}, L_{i+2}, ..., L_K\}$ are completely unrolled; otherwise, we only collect U_K iterations of L_K as the sub-trace.

Loop pipelining is only applied at one loop level L_i in L. When loop pipelining is applied to L_i and $i \neq K$, all the nested inner level loops $L' = \{L_{i+1}, ..., L_K\}$ are completely unrolled irrespective of their unrolling factors. The sub-trace in this case consists of instruction instances of L'. On the other hand, if $i = K$, then the sub-trace consists of the instruction instances in U_K iterations of L_K.

4.3.2 DDDG Generation & Pre-optimizations

An un-optimized DDDG is generated first from the sub-trace. Nodes in the DDDG represent dynamic instances of LLVM IR instructions, while edges denote dependences between nodes including register/memory-dependence. Edges only consider true dependences and does not include anti-/output dependences. Control dependences are not considered as we are dealing with dynamic traces.

Before scheduling, we perform tree-height reduction to decrease the height of long expression chains and expose potential parallelism similar to Shao's work [12]. The generated DDDG contains supporting instructions and dependences between loop index variables that are not relevant to accelerators. To focus only on actual computations, we remove supporting instructions and dependences by assigning zero latency to the associated nodes.

We also perform optimizations to remove redundant load-/store operations to save memory (BRAM) bandwidth. For example, if a store is a direct predecessor of a load with the same memory address, then the load is redundant and can be eliminated by adding edges between the store and successors of the load. If two loads have the same memory address and there is no store operation in between with the same address, the load operations can be reduced to one load node. After removing redundancies, we map each unique memory address (corresponding to load/store operations) to a memory bank according to array partitioning factor.

4.4 DDDG Scheduling

Once an optimized DDDG is generated, Lin-Analyzer performs Resource-Constrained List Scheduling (RCLS) for performance prediction. The inputs to RCLS are the DDDG and a priority list obtained from As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling policies. These algorithms schedule nodes with the following assumptions: (1) hardware functional units associated with nodes follow the default setting of Vivado HLS including their types, latencies, and resources consumption (e.g., a 32-bit floating-point addition node is mapped to a pipelined floating-point add (FADD) unit with 5-cycle latency consuming 2 DSP and zero BRAM); (2) each memory bank only supports two reads and one write simultaneously; (3) nodes related to supporting instructions (e.g., loop index computation) are assigned zero latency; and (4) resource constraints are modeled for only DSP and BRAM, as these resources are the bottlenecks in most accelerator designs.

With the above assumptions, ASAP policy schedules a node as soon as possible when its predecessors are all completed and returns minimum latency for the DDDG. ALAP policy schedules nodes at the latest opportunity. Timestamps of nodes after ALAP scheduling are fed into RCLS scheduling and serve as a priority list. ASAP and ALAP schedules are generated without resource constraints. However, it is impossible to generate FPGA-based accelerators with infinite resources. Therefore, to find a feasible schedule of minimum latency under limited FPGA resources, we employ resource-constrained list scheduling algorithm.

In RCLS algorithm, a node is scheduled when all of the following conditions are satisfied: (1) its predecessors have all completed; (2) it has the highest priority among the unscheduled ready nodes; (3) resources are sufficient for allocating the node. When scheduling nodes, resources allocation and release are managed by FPGA Resource Allocator (FRA). For example, to schedule a single-precision floating-point multiplication node, FRA checks whether there exists an unoccupied floating-point multiplication unit (FMUL). If all FMUL units are occupied, FRA allocates 3 DSP blocks from the resource budget to create a new FMUL unit and records the occupied status of the unit; otherwise, FRA allocates a previously generated FMUL unit for the node. For pipelined functional units (floating-point operations), FRA releases the unit in the next cycle after a node using this functional unit is scheduled (assuming single-cycle initiation interval for the pipelined functional units), while for non-pipelined units (integer operations), FRA releases the unit when the associated node finishes its execution.

When all nodes in the DDDG have been scheduled and finished execution, RCLS algorithm terminates and returns the finial schedule and latency for the DDDG.

4.5 Enabling Architectural Exploration

The current state-of-the-art HLS tools provide optimization pragmas for users to explore and evaluate diverse hardware architectures. Loop unrolling, pipelining and array partitioning are the three prominent pragmas that have significant impact on hardware performance and resource utilization. Hence, we designed Lin-Analyzer to support these three pragmas and enable rapid architectural exploration.

Loop Unrolling: To simulate loop unrolling with factor u, Lin-Analyzer extracts u loop iterations as *sub-trace* to generate a DDDG as explained in Section 4.3.1. After removing dependences between loop index variables in the DDDG, u loop iterations can be executed in parallel if there is no loop carried dependences across different iterations. RCLS schedules nodes in the optimized DDDG and returns latency of u iterations of the loop (IL). Lin-Analyzer then takes into account the IL, loop bounds and the unrolling factor u, to predict performance of the accelerator.

Loop Pipelining: Accelerator performance, with loop pipelining enabled, is determined by a constant initiation interval (II) of the loop, where the II is defined as the interval between the start of consecutive iterations. Instead of generating schedule with loop pipelining, Lin-Analyzer calculates the minimum initiation interval (MII) by Equation 1 to predict performance,

$$MII = \max(RecMII, ResMII) \tag{1}$$

where $RecMII$ and $ResMII$ are the recurrence-constrained and resource-constrained MII, respectively. MII is used as an approximation for the II to reduce the size of the sub-trace required by Lin-Analyzer and hence its runtime. $RecMII$ is introduced by loop-carried dependences. Lin-Analyzer calculates $RecMII$ by using Swing Modulo Scheduling described in [16]. $ResMII$ is calculated as,

$$ResMII = \max(ResMII_{mem}, ResMII_{op}) \tag{2}$$

$$ResMII_{mem} = \max_m\left(\left\lceil \frac{R_m}{RPorts_m} \right\rceil, \left\lceil \frac{W_m}{WPorts_m} \right\rceil\right) \tag{3}$$

$$ResMII_{op} = \max_n\left(\left\lceil \frac{Fop\_Par_n}{Fop\_used_n} \right\rceil\right) \tag{4}$$

where $ResMII_{mem}$ is limited by memory bandwidth and $ResMII_{op}$ is constrained by number of floating-point functional units. R_m and W_m denote the number of memory reads and writes of array m inside a pipeline stage respectively, while $RPorts_m$ and $WPorts_m$ represent number of available read and write ports of array m respectively. Fop_Par_n is the maximum number of floating-point functional unit of type n that can run in parallel and this value is obtained from ALAP scheduling without resource constraints. Fop_used_n is the number of floating-point functional unit of type n used in RCLS scheduling. Using IL for the pipeline depth, predicted MII as well as the loop bounds, Lin-Analyzer estimates FPGA performance with loop pipelining enabled utilizing the equation in [7].

Array Partitioning: This technique partitions program arrays into multiple memory banks to improve memory bandwidth. It is a commonly used array optimization in FPGA domain, as memory blocks in FPGAs have limited number of read/write ports. In this work, we assume that each memory partition has two read ports and one write port. Similar to Vivado HLS [17], Lin-Analyzer supports *block*, *cyclic* and *complete* array partitioning strategies (one-dimensional partitions). Due to the page limitation, further details on *block*, *cyclic* and *complete* partitioning strategies have not been included here, but can be found in [17].

Lin-Analyzer enables array partitioning by mapping addresses of memory nodes (load and store) in the DDDG to memory banks. FRA keeps track of read/write ports used for each memory bank in each cycle and only allows up to 2 read or 1 write memory accesses for the same memory bank. Memory bank number is calculated as follows:

$$Bank\_N_m = \begin{cases} (addr_m)/(\lceil size_m/pf \rceil) & \text{if block (5a)} \\ (addr_m)\ modulo\ (pf) & \text{if cyclic(5b)} \end{cases}$$

where $addr_m$ is a memory address within array m, $size_m$ is size of array m, and pf is the partition factor. For *complete* array partitioning, we do not need to consider memory-port constraints as the entire array is placed in registers.

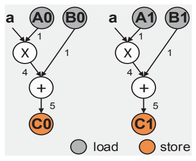
An Example: Figure 2 shows an example to demonstrate how Lin-Analyzer predicts FPGA performance. Let us assume that the FMUL unit has 4-cycle latency, while the FADD unit has 5-cycle latency. FMUL and FADD are pipelined hardware components. Memory load and store operations have 1-cycle latency. These FPGA node latencies are obtained from Vivado HLS. In this example, let us assume the designer wants to apply loop unrolling with factor of 2, no loop pipelining and cyclic partitioning of array C with partitioning factor 2.

Lin-Analyzer collects an execution trace of this program from LLVM and extracts *sub-trace* of two loop iterations based on loop unrolling factor for generating a DDDG. After performing optimizations on the DDDG, the optimized DDDG, shown in Figure 2b, reflects dataflow nature of the FPGA-based accelerator. Edges in the DDDG represent flow dependences and FPGA node latencies are added to edges as edge weights. Figure 2c shows the schedule generated by RCLS algorithm. As each memory partition has two read ports, memory load operations for array A and B can be executed in the same cycle. As array C has cyclic partitioning with partitioning factor 2, the two memory stores ($C0$ and $C1$) access different memory banks and can be executed in the same cycle. Figure 2c illustrates a feasible schedule with two FADD and FMUL units. The IL for the two loop iterations is 11 cycles. Therefore, the total latency of the hardware accelerator is $IL * N/2$ cycles where N is the loop bound.

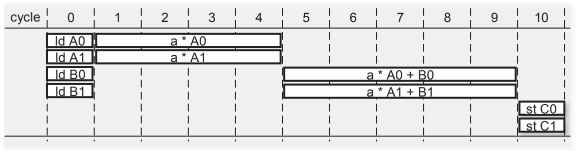
Similar to this example, Lin-Analyzer can rapidly estimate the performance for other combinations of pragmas

```
...
/* float a,A,B,C */
for(int i=0;i<N;i++){
  C[i] = a*A[i]+B[i];
}
...
```

(a) Source Code



(b) DDDG



(c) Scheduling Graph

Figure 2: An Example

without generating RTL implementations. This, in turn, enables rapid architectural exploration in the order of seconds to minutes over a large design space. However, it should be noted that Lin-Analyzer, like other dynamic analysis tools relying on profiling [12], can suffer from inaccuracy in performance prediction if the behavior of the application differs significantly across different program inputs. In such cases, it is imperative to carefully select a representative program input for trace generation. Also Lin-Analyzer only optimizes for performance and lets the accelerator use all the available FPGA resources if necessary. In future, we plan to explore area-performance tradeoff in accelerator design.

5. EXPERIMENTAL RESULTS

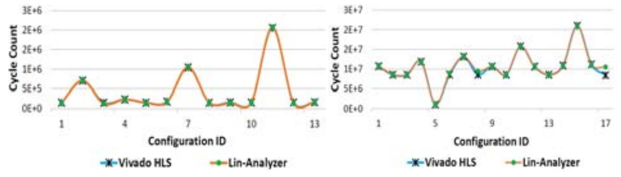
We use 10 applications from Polybench benchmark suite [15] to validate the proposed tool. The input data size for each application is chosen such that the application can use the resources (LUT, BRAMs, DSP, etc.) available on our target FPGA device, Xilinx ZC702 Evaluation Kit [17]. We use Xilinx Vivado HLS version 2014.4 and set the accelerators to run at 100MHz. Experiments are conducted on a PC with an Intel Xeon CPU E5-2620 core at 2.10Hz with 64GB RAM, running Ubuntu 14.04 OS.

5.1 Estimation Accuracy

Lin-Analyzer predicts accelerator performance considering loop unrolling, pipelining and array partitioning. As Vivado HLS might add false loop-carried dependences when using array partitioning, we report estimation accuracy separately according to optimization pragmas.

5.1.1 Loop Unrolling and Pipelining

Figure 3 shows the cycle counts predicted by Lin-Analyzer against the ones obtained from Vivado HLS for `Convolution3D` and `MM` applications considering loop unrolling and pipelining pragmas (due to page limitation, we only plot two applications). The X-axis denotes the various configurations, each of which represents a pragma combination (e.g., loop unrolling factor of 2 and pipelining outer loop for one configuration). The cycle counts predicted by the Lin-Analyzer tool matches the Vivado HLS tool very closely for both applications. Similar results were also obtained for the remaining applications. Table 4 demonstrates the average percentage difference between the cycle counts predicted by Lin-Analyzer and the ones obtained from the Vivado HLS for the same configuration across all combinations of loop unrolling and pipelining pragmas. Lin-Analyzer can accurately predict accelerator performance with less than 5.2% difference compared to Vivado HLS for all 10 benchmarks.



(a) Convolution3D (b) MM

Figure 3: Accuracy of Lin-Analyzer with loop unrolling and pipelining pragmas compared to Vivado HLS

Table 4: Difference between Lin-Analyzer estimated performance and the ones obtained from Vivado HLS with loop unrolling and pipelining pragmas

Benchmark	ATAX	BICG	CONV2D	CONV3D	GEMM
Difference (%)	2.68	1.24	1.63	3.75	3.25
Benchmark	GESUMMV	MM	MVT	SYR2K	SYRK
Difference (%)	5.15	2.69	2.05	1.32	2.78

5.1.2 Array Partitioning

Figure 4 compares the results of Lin-Analyzer and Vivado HLS for `Convolution3D` application while considering loop unrolling, pipelining and array partitioning pragmas. The code listing for this application was discussed earlier in Section 3. The X-axis denotes the array partitioning factors from 1 to 16 in step of 2. Dashed lines denote cycle counts predicted by the Lin-Analyzer, while solid lines represent results from Vivado HLS. (ui-Pj) denotes a pragma combination with loop unrolling factor i for the innermost loop and pipelining applied on loop level j.

As shown in Figure 4, Lin-Analyzer can predict the performance trends accurately with varying partitioning factors. However, for a particular combination, the cycle count predicted by Lin-Analyzer may differ to some extent from the one obtained from Vivado HLS. This is not a limitation of Lin-Analyzer and can be explained as follows. In these configurations, Vivado HLS, while relying on static analysis, conservatively adds *false loop-carried dependences* leading to higher recurrence II values dominated by these dependences. Increasing memory bandwidth by applying array partitioning, therefore, does not help to reduce the II in Vivado HLS. This can be observed in the solid blue line showing the results for the ($u3$-$P3$) configuration in which Vivado HLS is unable to improve the cycle counts by exploiting array partitioning. A hand-written RTL code, however, can effectively exploit this feature as predicted by Lin-Analyzer. Moreover, we also simulate the RTL code generated by Vivado HLS and find that there are redundant memory reads for some configurations when using array partitioning. This further deteriorates accelerator performance in HLS designs when compared to an optimized hand-written RTL code.

From the above discussion, it is also evident that Lin-Analyzer can assist designers in generating efficient FPGA-based accelerators using HLS tools by better understanding design bottlenecks. Besides, Lin-Analyzer can also help HLS developers to detect potential issues in HLS tools.

5.2 Rapid Design Space Exploration

The key motivation behind the proposed Lin-Analyzer tool is to rapidly evaluate the accelerator performance while using various pragma combinations such as loop unrolling, loop pipelining and array partitioning. The design space considered in this work is shown in Table 5.

Table 6 lists the different application kernels used in this work in column 1, while column 2 shows the number of loops in each of these applications. Column 3 shows the total num-

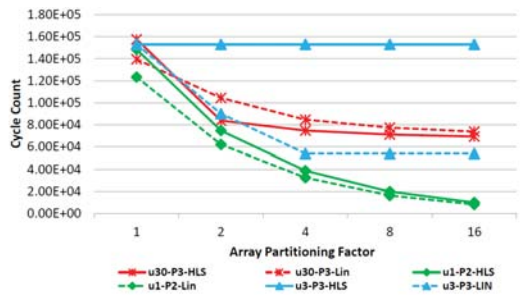


Figure 4: Lin-Analyzer vs. Vivado HLS for Convolution3D considering loop unrolling, pipelining and array partitioning. False loop-carried dependences cause Vivado HLS to create inefficient designs leading to difference with estimations from Lin-Analyzer.

Table 5: Pragma settings used in Lin-Analyzer

Pragmas	Range
Loop unrolling factor	Divisors of loop bound N
Loop pipelining	$[True,\ False]$
Array partitioning	factor: [1 :: 2 :: 16] (the set of values from 1 to 16 in steps of 2) type: $[cyclic,\ block,\ complete]$

ber of configurations considered for each application while exploring various loop unrolling factors, pipelining options, and array partitioning factors/types. To evaluate the accuracy of Lin-Analyzer, we also used Vivado HLS tool to synthesize accelerators for the same configurations and recorded HLS runtime and cycle count of the generated hardware accelerators. In Table 6, this is shown as *Exhaustive* method.

Columns 4 and 5 in Table 6 compare the optimal design points returned by exhaustive DSE with HLS and Lin-Analyzer. The tuple denotes configurations recommended by *Exhaustive* and Lin-Analyzer, respectively, that achieves the best performance for each application. It has the format: *(partitioning factor, loop unrolling factor, pipeline level)*. The pipeline level i indicates that the pipelining pragma is applied to loop level L_i as explained in 4.3.1. The results confirm that Lin-Analyzer recommends exactly the same configurations as *Exhaustive* method.

The exploration time of *Exhaustive* method is shown in column 6. Lin-Analyzer profiles an application in the first step to create dynamic trace. The time taken for this step is shown in column 7, while column 8 shows the DSE time. Profiling incurs one-time overhead and this overhead is amortized. The last column in Table 6 shows the total runtime for Lin-Analyzer. Comparing columns 6 and 9, it can be seen that Lin-Analyzer takes a fraction of the time needed by *Exhaustive* approach while arriving at the same result.

The asterisk for `BICG` and `GESUMMV` signify that for these applications there is no change in performance obtained from Vivado HLS, when partitioning factor varies from 1 to 16. This is a limitation of Vivado HLS, as it does not exploit array partitioning as expected, to improve performance. In contrast, Lin-Analyzer correctly predicts the effect of array partitioning and thereby finds the best configuration, i.e. (16,1,1) for both `BICG` and `GESUMMV`.

It should also be noted that unlike the existing state-of-the-art techniques that invoke HLS tools, Lin-Analyzer does not rely on HLS tools. Hence, its runtime scales linearly with increasing design space complexity (more unrolling factors, pipeline levels, etc.), therefore making it attractive during DSE of complex FPGA-based accelerators.

Table 6: DSE Results. Configuration Format (array partitioning factor, loop unrolling factor, pipeline level)

	Loop Levels	Design Space	Configuration		Total DSE Time (seconds)			
			Exhaustive	Lin-Analyzer	Exhaustive	Lin-Analyzer		
						Profiling	DSE	Total
ATAX	2	85	16,1,1	16,1,1	3855.74	1.28	7.57	8.85
BICG	2	95	(1-16),1,1 *	16,1,1	8989.35	7.40	15.20	22.60
CONV2D	2	62	16,1,1	16,1,1	26573.13	5.48	17.28	22.76
CONV3D	3	65	16,1,2	16,1,2	21586.68	12.85	9.62	22.47
GEMM	3	85	1,1,2	1,1,2	36579.48	176.38	8.99	185.37
GESUMMV	2	85	(1-16),1,1 *	16,1,1	17318.30	1.88	10.17	12.05
MM	3	85	1,1,2	1,1,2	7986.90	153.07	8.30	161.37
MVT	2	95	16,1,1	16,1,1	7696.61	3.57	11.31	14.88
SYT2K	3	85	1,1,2	1,1,2	50714.11	239.24	10.76	250.01
SYRK	3	85	1,1,2	1,1,2	20769.15	160.05	8.73	168.78

6. CONCLUSION

In this paper we proposed Lin-Analyzer, a high-level performance estimation tool that accurately and rapidly predicts the performance of FPGA-based accelerators. Lin-Analyzer relies on the dynamic data dependence graph (DDDG) to avoid the false data dependences created by the static analysis techniques used in most existing techniques including commercial HLS tools. This results in an accurate performance estimation of FPGA-based accelerators without resorting to time-consuming HLS runs. The tool also helps in identifying design bottlenecks while exploring various pragmas such as loop unrolling, pipelining, and array partitioning. Lastly, Lin-Analyzer can assist HLS developers in identifying potential limitations of the HLS tool.

7. ACKNOWLEDGMENTS

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2012-T2-1-115 and French-Singaporean Merlion 2012 PhD Project.

8. REFERENCES

[1] Cadence Inc. C-to-Silicon Compiler, 2015. www.cadence.com.
[2] Austin T. et al. Dynamic Dependency Analysis of Ordinary Programs. In *ISCA*, 1992.
[3] Bilavarn S. et al. Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations. *TCAD*, 2006.
[4] Canis A. et al. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *FPGA*, 2011.
[5] Cong J. et al. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *DAC*, 2006.
[6] Lattner C. et al. LLVM: a Compilation Framework for Lifelong Program Analysis Transformation. In *CGO*, 2004.
[7] Li P. et al. Resource-Aware Throughput Optimization for High-level Synthesis. In *FPGA*, 2015.
[8] Liu H. et al. On Learning-based Methods for Design-space Exploration with High-Level Synthesis. In *DAC*, 2013.
[9] Pham N. et al. Exploiting Loop-array Dependencies to Accelerate the Design Space Exploration with High Level Synthesis. In *DATE*, 2015.
[10] Prost-Boucle A. et al. A Fast and Autonomous HLS Methodology for Hardware Accelerator Generation under Resource Constraints. In *DSD*, 2013.
[11] Schafer B. et al. Divide and Conquer High-level Synthesis Design Space Exploration. *TECS*, 2012.
[12] Shao Y. et al. Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *ISCA*, 2014.
[13] So B. et al. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In *PLDI*, 2002.
[14] Zhong G. et al. Design Space Exploration of Multiple Loops on FPGAs Using High Level Synthesis. In *ICCD*, 2014.
[15] Pouchet L. PolyBench/C 3.2. http://web.cse.ohio-state.edu/~pouchet/software/polybench/.
[16] Lattner T. An Implementation of Swing Modulo Scheduling with Extensions for Superblocks. Master's thesis, 2005.
[17] Xilinx Inc., 2015. www.xilinx.com.