# Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs

Lana Josipović, Atri Bhattacharyya, Andrea Guerrieri, and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)

***Abstract*—When applications have unpredictable memory accesses or irregular control flow, dataflow circuits overcome the limitations of statically scheduled high-level synthesis (HLS). If memory dependences cannot be determined at compile time, dataflow circuits rely on load-store queues (LSQs) to resolve the dependences dynamically, as the circuit runs. However, when employed on reconfigurable platforms, these LSQs are resource-expensive, slow, and power-consuming. In this work, we explore techniques for reducing the cost of the memory interface in dataflow designs. Apart from exploiting standard memory analysis techniques, we present a novel approach which relies on the topology of the control and dataflow graphs to infer memory order with the purpose of minimizing the LSQ size and complexity. On benchmarks obtained automatically from C code, we show that our approach results in significant area reductions, as well as increased performance, compared to naive solutions.**

## I. Introduction

High-level synthesis tools, both commercial and academic, typically rely on static scheduling to produce high-throughput pipelines [27], [5]. However, in applications where memory accesses cannot be disambiguated at compile time, these tools make pessimistic assumptions on potential read-after-write and write-after-write dependences, therefore producing suboptimal schedules and resulting in lower performance. In contrast, dataflow or latency-insensitive protocols [6], [9], [25], [11] implement dynamically scheduled circuits, in which components communicate locally using a handshake mechanism and exchange data as soon as all conditions for execution are satisfied (i.e., when all data and control dependences are resolved). Due to their ability to adapt the schedule during runtime when a data hazard is detected, dataflow circuits have recently been explored as an efficient HLS approach to handle applications with irregular memory accesses [18].

Any HLS compiler must assure that memory accesses by the generated circuit to the same address happen in the same order as specified in the original program. A violation of this condition results in a data race and may lead to incorrect execution. To correctly handle memory accesses which may arrive at the memory interface out of order, dataflow designs rely on load-store queues (LSQs) to dynamically resolve memory dependences by appropriately reordering dependent accesses [3], [17], [26]. While issuing every load and store request to memory through an LSQ guarantees correctness, this solution is unattractive: LSQs have been shown to incur significant resource overheads as well as power and clock degradation with queue size, and especially on FPGAs [26]. Although standard HLS optimizations can, in certain cases, help in reducing this cost, analyzing memory access patterns of a dataflow circuit in order to decide where an LSQ is redundant remains a challenge—there is no predetermined, static schedule to provide information of the temporal ordering of memory accesses, so any two accesses targeting the same memory location may potentially conflict and, therefore, may require an LSQ. In this work, we explore a novel memory analysis technique which, based on the flow of data through the circuit, determines the activation ordering of certain memory accesses. This information allows us to rule out specific dependences and simplify the memory interface accordingly, leading to significant area savings and a tangible timing advantage.

## II. Motivation

The code in Figure 1 shows a loop with multiple memory accesses. Above the code is a section of the dataflow circuit which represents the loop kernel. One should notice that the dataflow circuit has no central controller and all dataflow units are connected to their neighbors with handshake signals to indicate the validity of new data and readiness to accept operands; each operation takes place as soon as the operands are available and the unit to execute it is ready. Load and store memory operators are connected to the memory system to get and put data, respectively. As illustrated in Figure 1, there are several possibilities for connecting the loads and stores of the dataflow circuit to memory. All design points in the figure are semantically correct for this particular example; they represent different optimization degrees of the memory interface.

In Figure 1a, all memory accesses are connected to memory using a single LSQ. For maximum parallelism, the LSQ executes requests out of order but ensures that no load is executed before an outstanding store to the same address—or forwards the corresponding value if and when available. This solution guarantees correctness without requiring any memory analysis, yet it is extremely resource-expensive, as it requires a large queue to maintain high parallelism and to consume all incoming requests at a high rate.

By exploiting methods such as alias analysis, one can disambiguate memory accesses when possible and connect them to independent memory ports using multiple, smaller LSQs (Figure 1b). Analyzing memory access patterns using techniques such as polyhedral analysis would allow us to simplify the design even further by removing LSQs in cases where the loads and stores targeting the same memory provably never access the same memory locations (Figure 1c). However, whenever this is not the case (i.e., the loads and stores might access the same locations at some point in time), standard techniques do not allow us to optimize our design any further—in a dataflow circuit, accesses may arrive at the memory interface out of order and an LSQ is needed to prevent a

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT47387.2019.00031

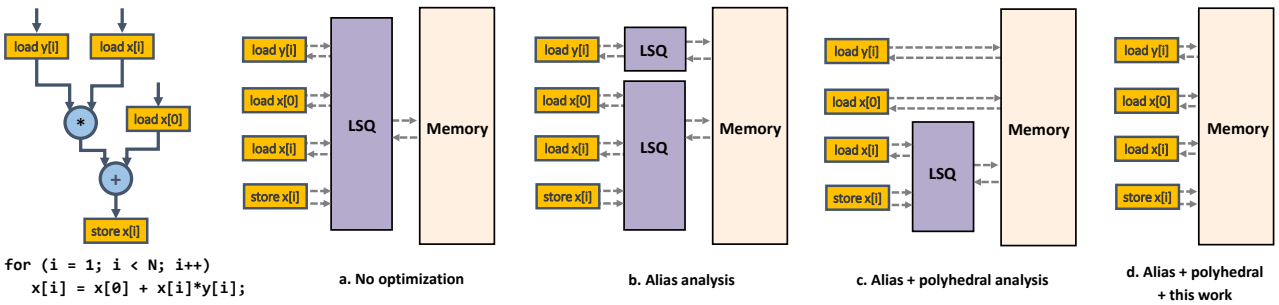



Fig. 1: Connecting a dataflow circuit to memory. The dataflow circuit on the left, which represents the loop kernel datapath of the code below, can be naively connected to memory using a single, large LSQ to reorder memory accesses (Figure 1a). Standard memory optimizations allow us to disambiguate memory accesses targeting different memories (Figure 1b) as well as to simplify the interface in cases where a memory access provably does not collide with any other access (Figure 1c). The optimal configuration (Figure 1d) is obtained using a specialized analysis for dataflow circuits which we present in this work. In this particular case, our analysis concludes that, since the load *must* occur before the store to the same memory location (i.e., the load is a certain producer of data for the store), the two accesses naturally occur in order and the LSQ can be omitted.

hazard. The main contribution of our work is an analysis which particularly targets dataflow circuits and, based on the flow of data through the dataflow graph, makes it possible for us to rule out specific data hazards through memory. In our example, it proves that a violation of the write-after-read dependence is impossible—the load of x[i] is the data producer for the store of x[i] and the two accesses can therefore never occur out of order. This information enables us to completely remove the LSQ, as given in Figure 1d.

In the rest of the paper, we discuss our methodology for optimizing the memory interface of dataflow circuits. In Section III, we provide the necessary background on dataflow circuits and discuss existing memory optimization techniques which more conventional forms of HLS regularly exploit. In Section IV, we detail the particularity of our circuits which prevents us from fully relying on existing methods and we introduce our new technique for analyzing memory accesses in an out-of-order dataflow circuit. We evaluate the effectiveness of our technique to simplify the memory interface in Section V.

## III. Background

In this section, we describe dataflow circuits and their properties which are relevant for this work. We then review memory analysis techniques employed by standard HLS approaches. We will later discuss their usage in the context of dataflow circuits and illustrate that, on their own, they are not always sufficient to create an optimal memory interface.

### A. Dataflow circuits

Latency-insensitive protocols implement dynamically scheduled dataflow circuits. These circuits are built out of dataflow components which use a handshake mechanism to exchange pieces of data (i.e., *tokens*). A description of how arbitrary functionality specified in imperative code can be implemented as a pipelined dataflow circuit has been presented recently [18] and we largely follow this methodology in our work. Although the exact circuit topology is irrelevant for this paper, it is important to note that the circuits we consider are organized into sections corresponding to basic blocks (BBs), i.e., pieces of code with no conditionals. All control flow statements are implemented between the BBs and each BB contains a dataflow graph (DFG) of program instructions.

The dataflow circuits we consider respect the property that, on any acyclic path, pieces of data arrive to each operator *strictly in order* [19]. In the example in Figure 1, the data of each loop iteration will arrive to each load and store component in the order specified by the original program (e.g., the load of y[1] *must* occur before the load of y[2]). However, different dataflow operations may be executed out of order (e.g., the load of x[2] may occur before the store to x[1], which is why, in general, the memory interface requires a load-store queue to ensure correct read-write ordering.

### B. Alias Analysis

The memory interface illustrated in Figure 1a corresponds to that of a compiler which does not perform any analysis of the memory accesses. Such a compiler must assume that each access in the code can point to any addressable value in memory [24] and therefore conservatively connects all accesses of the circuit to memory using a single monolithic LSQ.

Alias analysis groups pointers into sets such that different groups never access the same memory locations [24], [1]. HLS tools typically rely on alias analysis to simplify the memory interface by connecting different alias groups to different memories or independent memory ports. In our case, alias groups would connect to independent LSQs (see Figure 1b) which either insist on a single memory system (e.g., in cloud FPGA applications) or on different memories (e.g., separate block RAMs, like standard HLS tools employ [27]). Without loss of generality, we indicate in Figure 1 a single monolithic memory system with appropriate arbitration between ports.

### C. Polyhedral Analysis

The polyhedral model is a linear-algebraic representation of a program which provides strong static analysis capabilities for *Static Control Parts (SCoPs)*. SCoPs are side-effect free regions of a program in which all control flow decisions and memory accesses are known at compile time [13].

The loop from Figure 1 is an example of a SCoP in which the iterator $i$ is bound by the constraints $1 \leq i < n$ and increased

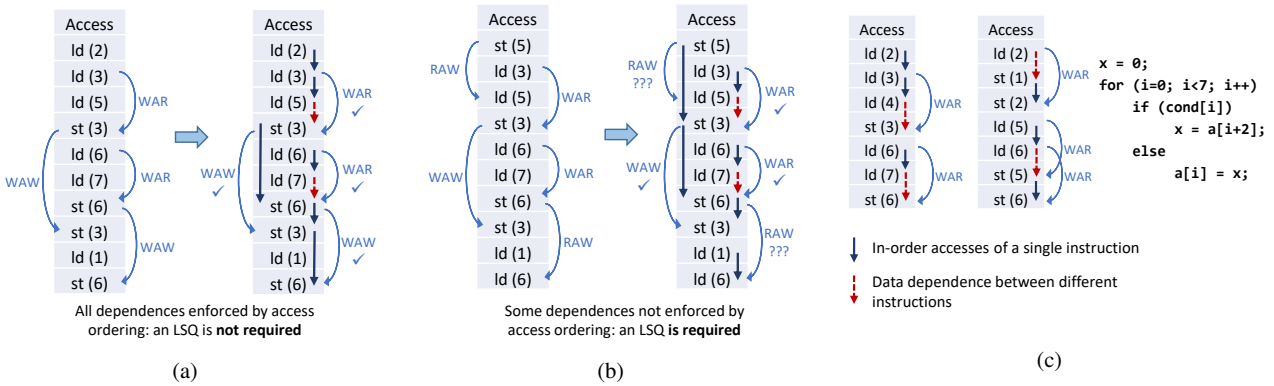



Fig. 2: Figures 2a and 2b show memory traces of two programs with a single load and a single store instruction, which contain RAW, WAR, and WAW dependences. The figures on the right show how we can exploit the ordering information between accesses to conclude that certain dependences will always be honored. Our analysis identifies data dependences between instructions (dashed arrows) which exclude access reordering and eliminate the need to use an LSQ. Figure 2c shows two (out of many) possible memory traces of the code in the figure. Because a data dependence will *always* occur between a load access and its subsequent store access, the circuit does not require an LSQ.

by 1 in each iteration. Polyhedral analysis can generate the sequence of array indices (i.e., memory addresses) that each memory instruction will access, which gives us information on all *read-after-write (RAW)*, *write-after-write (WAW)*, and *write-after-read (WAR)* dependences. In the example in Figure 1, there is a WAR dependence between the load and the store access of the same iteration. Because the load of x[0] only accesses the disjoint set $\{0\}$, we can simplify the memory interface to achieve the configuration from Figure 1c.

## IV. Memory Interface Optimizations

It should be clear from the previous section that standard analysis techniques allow us to group memory instructions into a maximal number of *memory sets* such that the address of an instruction in one given set can never collide with the address of an instruction in another set (i.e., an LSQ is not needed across sets). Yet, without further analysis, each memory set with more than a single instruction *must* employ an LSQ to handle all RAW, WAW, and WAR dependences between the instructions. This is the very situation shown in Figure 1c and we will detail in Section V our choice of standard analysis techniques to achieve this result. In the rest of this section, we will describe our original effort to simplify the memory interfaces of dataflow circuits past this design point, as suggested by Figure 1d.

### A. The Ordering Problem

Let us assume that, using standard analysis techniques, we have clustered all memory instructions into a maximal number of mutually independent memory sets, as indicated above. We will focus here on a program with only two instructions in a single set; we will generalize our approach to sets with multiple instructions in Section IV-D. The same reasoning we here describe applies to programs with multiple sets by considering each set independently from the others.

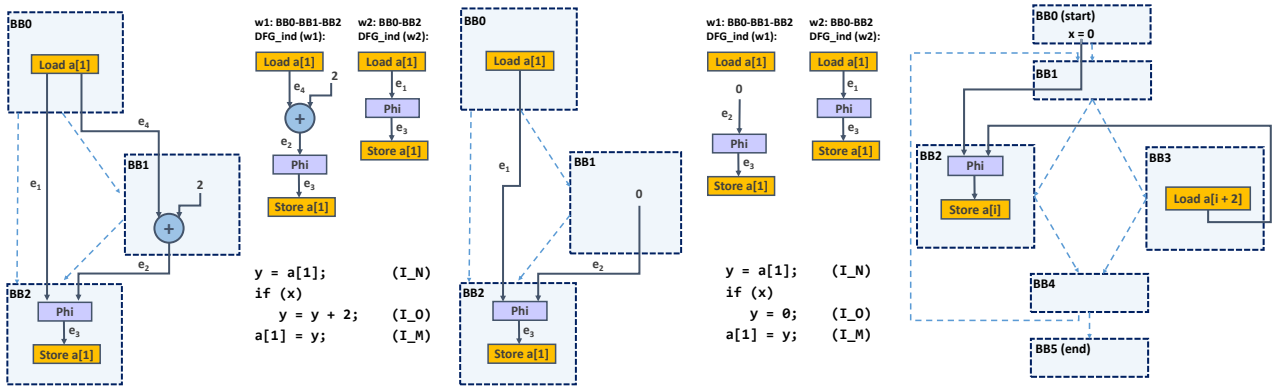
We consider a program with a single load and a single store instruction in a memory set. In fact, the ordering relations between load and store instructions will be the main focus of our analysis; although the ordering of colliding memory accesses of two store instructions matters as well, our analysis will not achieve any memory interface simplification in this case, as we will later observe. On the other hand, the ordering of a pair of load instructions does not impact correctness as none of the load accesses modifies the memory state.

Figures 2a and 2b show examples of sequential memory traces of two independent programs, which both contain a load and a store instruction. The purpose of a dynamically scheduled dataflow circuit is to execute each instruction as soon as its arguments are known, exactly as in a superscalar out-of-order processor [15]. Therefore, we generally need to assume that the circuit might try to reorder the shown sequences in any possible way. Some of these reorderings might result in semantically incorrect execution, as indicated in the figure. For instance, both sequences feature WAR dependences (i.e., there is a store access which writes into the same memory address that some load which precedes it in program order reads) and WAW dependences (i.e., there is a store access which writes into the same memory address as some store which precedes it in program order). The second sequence also contains RAW dependences (i.e., there is a load access which reads from the same memory address into which some preceding store writes). Although all other accesses can be reordered in the interest of execution speed, dynamically scheduled processors [15] and circuits [17] need an LSQ to enforce correct ordering across the accesses with dependences. If we could otherwise ensure the correct ordering of these accesses, we would be able to omit the LSQ.

### B. Exploiting Data Dependences

To remove the LSQ from the memory interface of a pair of load-store instructions, we need to reason about the ordering of their accesses in the execution of a dataflow circuit. There are two sources of information on which we can rely.

Firstly, each instruction performs its accesses strictly in program order—this property is guaranteed by the construction strategy of the dataflow circuit, as indicated in Section III-A. Secondly, there might be a data dependence between a load access and the following store access which would guarantee their correct ordering—if the load produces the data necessary




(a) (b) (c)

Fig. 3: Code snippets and their control/data flow graphs which we use to illustrate the global instruction dependence property in Section IV-C. In the circuit in Figure 3a, the dependence property $I_N \xrightarrow{\text{GID}} I_M$ holds. This is not the case for the two instructions in Figure 3b because the load is not a predecessor of the store along both control-flow paths. Figure 3c corresponds to the example of Figure 2c.

to compute the store value, there is no way for the store to run ahead in program execution.

The first property directly guarantees that WAW dependences are never a concern among write accesses of a single store instruction. The combination of the first and the second property ensures that any WAR dependence of a store access with *any* prior load access is maintained: if the store access has a data dependence with the preceding load access, and the load accesses execute in order, all previous load accesses will have completed before the store access, thus honoring any WAR.

However, the properties above do not help us reason about RAWs: there is no data dependence to rely on because the store is not a data producer for any other instruction, including the load. A store access may be arbitrarily ordered with respect to some subsequent load access and an LSQ is required to ensure correctness if their addresses collide.

In summary, a pair of load-store instructions does not require an LSQ among themselves if (1) all WAR dependences of the original program are enforced by a data dependence between each store access and its preceding load access in the sequence and (2) there are no RAW dependences between the load and the store instruction. As determining the presence and absence of RAW dependences, among others, is often possible using standard analysis (as discussed in Section III), we will here focus on the first property which is original for this work.

The right sequences of Figures 2a and 2b illustrate visually how the two properties above enable us to exclude certain dependences, assuming that a data dependence exists. In Figure 2a, both properties hold and guarantee that all dependent accesses are correctly ordered, so an LSQ is not needed. However, in Figure 2b, the second property does not hold and an LSQ is required.

Both conditions for omitting the LSQ hold for the example from Figure 1: each iteration has a WAR dependence between the load and the store access of the same iteration which is always honored because the load produces the data for the store. Furthermore, there are no RAW dependences in the program. Of course, this is a trivial case where the load and the store instruction are directly connected through the very datapath of the loop. A more interesting example is given in Figure 2c. This code exhibits many possible memory traces (depending on the if-condition) which contain WARs across different (and not necessarily consecutive) loop iterations. To make sure that WARs are honored, we need to make sure that *all* possible traces have a data dependence between any load access and the store access that immediately follows. We will formalize this property in the following section.

### C. Global Instruction Dependence

In this section, we describe the property of two instructions which guarantees that their order in the dataflow execution is equivalent to their order in the sequential program execution. As described in Section III-A, we consider a *control flow graph (CFG)* of a program which is organized into BBs such that each BB contains a DFG of instructions (we denote the BB that instruction $I$ belongs to as $BB_I$).

The dependence property that we will introduce applies to the *induced DFG* of a walk through the program CFG, where, as customarily, a *walk* is any sequence of BBs directly connected by control edges in the CFG (contrary to a path, a walk admits to visit every BB and traverse any edge an arbitrary number of times).

**Definition 1.** An *induced DFG* of a CFG walk $w$, denoted as $DFG_{ind}(w)$, is a DFG composed of a succession of the DFGs of the BBs traversed in $w$, repeated as many times as each BB is visited; the live-ins of each DFG are connected exclusively to the live-outs of the predecessor DFG.

Our analysis aims to determine *Global Instruction Dependence* of a pair of instructions.

**Definition 2.** Instructions $N$ and $M$ are *globally dependent* (written as $N \xrightarrow{\text{GID}} M$) if, for every CFG walk $w$ starting with $BB_N$, ending with $BB_M$, and containing $BB_N$ and $BB_M$ only once, $N$ is the predecessor of $M$ in $DFG_{ind}(w)$.

This property implies that, for every possible control flow sequence in which $N$ and $M$ execute, there is always a data dependence between $N$ and $M$ which enforces their in-order execution.

To illustrate, consider the examples in Figure 3. The memory instructions $I_N$ and $I_M$ belong to $BB_0$ and $BB_2$, respectively. The CFG objects and edges are given in dashed in the figure. To determine whether a dependence relation exists between $I_N$ and $I_M$, we need to consider all walks from $BB_0$ to $BB_2$, which are equivalent for both examples in the figure: $w_1 = [BB_0, BB_1, BB_2]$ and $w_2 = [BB_0, BB_2]$. The $DFG_{ind}$ of each walk is given in the figures. In both $DFG_{ind}(w_1)$ and $DFG_{ind}(w_2)$ of Figure 3a, $I_N$ is the predecessor of $I_M$ (i.e., the value to be written by $I_M$ has a data dependence on the load $I_N$). Regardless of which CFG path is taken, the execution of $I_M$ implies that $I_N$ must have executed already, so $I_N \xrightarrow{\text{GID}} I_M$ and the WAR dependence between the two instructions is always honored. The same relation does not hold for the same instructions in Figure 3b, because $I_N$ is not the predecessor of $I_M$ in $DFG_{ind}(w_1)$. If walk $w_1$ is executed, there is no way to guarantee the ordering of these two instructions—the store may execute before the load which would result in a data hazard.

To summarize, instruction dependence ($L \xrightarrow{\text{GID}} S$) ensures that any WAR dependence between a load instruction $L$ and a store instruction $S$ is honored for every possible execution of the program. As mentioned earlier, we cannot exploit the same property to reason about RAW and WAW dependences, i.e., it will never hold that $S \xrightarrow{\text{GID}} L$ or $S \xrightarrow{\text{GID}} S$ because a store instruction never produces data and cannot be a predecessor of any instruction.

### D. From Two Memory Instructions to Many

The properties from Section IV-B with the original one formalized in Section IV-C provide us with the knowledge about the activation order of a certain load-store pair and enable us to guarantee correct execution between them. Our algorithm exploits this information to reduce the number of instructions (i.e., the number of connections to an LSQ) of memory sets with more than two instructions as well.

Concretely, a load instruction $L$ can be removed from memory set $M$ if the two properties introduced in Section IV-B hold for each store instruction $S$ of $M$:

1) $L \xrightarrow{\text{GID}} S$, i.e., any WAR dependence between $L$ and $S$ is enforced by a data dependence.
2) There are no RAW dependences between any of the accesses of $L$ with any of the accesses of $S$.

If all WAR dependences between the load $L$ and every store of the memory set are provably maintained in order and there are no RAW dependences with any store of the set, $L$ does not need an LSQ and can be removed from the set.

After certain load instructions have been removed from a memory set using the properties above, it is trivial to re-evaluate the set to remove all store instructions which no longer have conflicting accesses with any of the remaining instructions in the set.

### E. Why Not CFG Dominance?

If one is familiar with classic compiler analysis, it may appear that our Global Instruction Dependence should, in fact, be the classic notion of CFG dominance and post-dominance. These notions describe the relations between BBs in a CFG as follows: (1) A basic block $BB_N$ *dominates* basic block $BB_M$ if every path from the entry of the graph to $BB_M$ must go through $BB_N$. (2) A basic block $BB_M$ *post-dominates* basic block $BB_N$ if all paths to the exit of the graph starting at $BB_N$ must go through $BB_M$ [24].

It might seem that we could use these properties to describe a relationship between a load and a store instruction. However, they would not suffice, as the examples in Figure 3 clearly illustrate: the CFGs of the two programs are the same (i.e., in both cases, $BB_0$ dominates $BB_1$ and $BB_2$, $BB_2$ post-dominates $BB_0$ and $BB_1$), yet the memory instructions exhibit different dependence relations, as discussed in Section IV-C. Therefore, determining instruction dependence using these CFG properties might lead to incorrect results.

One could consider formulating properties similar to dominance at the instruction level: informally, one could check whether, on each path from program entry to a store, one passes through the load, or whether every path from the load to the exit passes through the store. While not incorrect, this formulation would be restrictive. Consider the example from Figures 2c and 3c: in this case, neither of these properties would hold (e.g., there is a path from the start of the program to the store which does *not* pass through the load); one would conclude that dominance does not exist and would conservatively place an LSQ. Our formulation is more general and captures situations when there is effectively a load before the store.

### F. Another Ordering Guarantee

Our definition is quantified on all walks through the CFG which have a single passage through the BB of the load and the store. This constraint provides us with the ordering of the *last* execution of the load before the execution of the store—if the dependence relation holds for these two accesses, all previous load accesses and all successive store accesses can be ordered with respect to this load-store pair. Yet, if the CFG has a cycle which does *not* contain the load or the store BB, the absence of a load-store dependence may be detectable only after multiple traversals the cycle, as illustrated in Figure 4—the lack of data dependence is present only after two cycle traversals (i.e., in walk $w_2$) and, based on Section IV-D, would require an LSQ between the load and the store. But there is more.

As discussed in Section III-A, accesses of the same instruction *always* execute in order—we have already used this property to order load and store accesses, but it applies to other instructions as well. This particularity adds implicit dependence edges between multiple instances (corresponding to multiple accesses) of the same instruction in induced DFGs and provides us with additional dependence relations.

**Property.** Each instruction instance in a $DFG_{ind}(w)$ has an ordering dependence on any preceding instance of the same instruction.

In Figure 4a, this property holds for the *phi* and *add* instructions which repeat (we indicate all ordering dependences with red dotted edges in Figure 4b); the ordering between the instances of the leftmost *phi* implies a dependence between the load and the store along *every* walk—in $w_2$, the load is

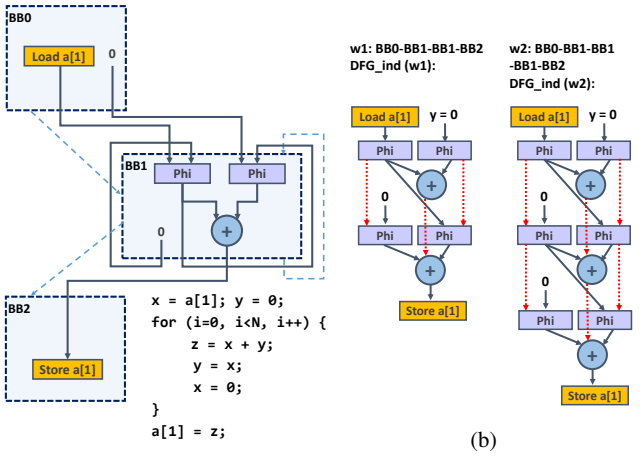



Fig. 4: The circuit in Figure 4a exhibits an absence of load-store dependence only after two traversals of the CFG cycle through $BB_1$, as illustrated in Figure 4b (path $w_2$). However, multiple instances of the same instruction are guaranteed by construction to execute in order. These ordering dependences are indicated with dotted arrows; they provide additional dependences between instructions (in this case, a dependence between the load and the store in path $w_2$).

not a data predecessor of the store, but the new ordering edges enforce their ordering and allow us to disconnect them from the LSQ. In the following section, we will formalize how this property enables us to further simplify the memory interfaces by simplifying the assessment of dependences.

### G. How Long a Walk Does One Need?

Our current definition requires checking *all* CFG walks to determine load-store dependences; in certain cyclic CFGs, it requires checking an infinite number of walks, which is not quite feasible. In the rest of this section, we discuss how to use the property described in the previous section to restrict our search to a finite subset of walks, making the test practical.

**Theorem.** If instructions $N$ and $M$ are dependent on the $DFG_{ind}(p)$ of a path $p$ from $BB_N$ to $BB_M$, they are also dependent on $DFG_{ind}(w)$ of any walk $w$ from $BB_N$ to $BB_M$ which contains $p$ (i.e., which includes all BBs and CFG edges belonging to path $p$).

We here consider a path to be a walk in which all vertices are distinct, except possibly the first and the last [4].

**Proof.** Given a path $p$ from $BB_N$ to $BB_M$ through any $BB_P$, $p = [BB_N, ..., BB_P, ..., BB_M]$, a dependence between $N$ and $M$ (denoted as $N \rightarrow M$) on $DFG_{ind}(p)$ implies that there exists at least one instruction $P$ in $BB_P$ such that $N \rightarrow P$ and $P \rightarrow M$. Let us consider now a walk $w$ which covers a single CFG cycle and contains $p$; the cycle has a CFG edge which connects into $p$ in $BB_P$; $w$ visits $BB_P$ multiple times, that is, $w = [BB_N, ..., BB_{P_{\text{first}}}, ..., BB_{P_{\text{last}}}, ..., BB_M]$, where $BB_{P_{\text{first}}}$ is the first visit to $BB_P$ in the walk and $BB_{P_{\text{last}}}$ is the last one. The dependence over the path $p$ ensures that $N \rightarrow P_{\text{first}}$ and $P_{\text{last}} \rightarrow M$. The property of Section IV-F also guarantees $P_{\text{first}} \rightarrow P_{\text{last}}$. Thus, transitively, $N \rightarrow M$ also on $DFG_{ind}(w)$. The same reasoning applies for all walks with whatever cycles or control-flow merges into $p$: a dependence holds across each BB where $w$ connects into $p$ and transitively holds across all of them. ■

This theorem restricts Definition 2 Section IV-C to paths instead of generic walks–we hence refine it to formulate *Global Instruction In-order Dependence (GIID)*:

**Definition 3.** Instructions $N$ and $M$ are *globally in-order dependent* (written as $N \xrightarrow{\text{GIID}} M$) if, for every CFG path $p$ starting with $BB_N$, ending with $BB_M$, and containing $BB_N$ and $BB_M$ only once, $N$ is the predecessor of $M$ in $DFG_{ind}(p)$.

The decision to remove a load instruction $L$ from a memory set can accordingly account for GIID instead of GID (see the first property of Section IV-D): if $L \xrightarrow{\text{GIID}} S$, any WAR dependence between $L$ and $S$ is enforced by a data dependence. We rely on this formulation in our experiments in Section V.

## V. Evaluation

In this section, we demonstrate the ability of our optimization approach to reduce the area and performance cost of the memory interface of dataflow circuits.

### A. Memory Analysis Implementation

We implement our memory optimization, detailed in Section IV, as an LLVM pass which determines the appropriate memory interface of a dataflow circuit. We exploit the alias analysis pass of LLVM (i.e., BasicAA [22]) to determine whether two pointers alias and to disambiguate arrays that target different memory regions. To extract the information about the memory access patterns, we rely on the ScopInfo pass of the Polly framework [14]. This pass detects SCoP regions within a program and creates a polyhedral description of the memory accesses it contains. We use this information to determine whether two instructions have RAW, WAW, and WAR dependences. Whenever an instruction is not part of any SCoP (i.e., Polly does not provide us with the memory access patterns), we connect it to an LSQ to ensure correctness.

To determine whether a dependence relationship holds between a load-store pair, we employ depth-first search (DFS) to find all CFG paths from the load to the store BB. We extract the $DFG_{ind}$ of each path and perform a DFG traversal across it to determine whether the load precedes the store. The information on the memory access patterns and the GIID relation determined in this step (see Section IV-G) enable us to simplify the memory interfaces, as described in Section IV-D.

### B. Experimental Methodology

In the rest of this section, we compare three design points: (1) designs without any optimization of the memory interface, which qualitatively correspond to the interface of Figure 1a, where all memory accesses of the circuit are connected to memory using a single LSQ; (2) designs optimized using the information provided by standard techniques (i.e., alias analysis and polyhedral analysis), and (3) our designs which, in addition to standard information, use the methodology from Section IV to simplify the memory interface.

| Benchmark | Memory access pattern |
|---|---|
| **Memory loop** | x[i] = *func* (x[0], x[i], y[i]) |
| **Scalar multiply** | x[i] = *func* (x[i]) |
| **Image revert** | x[i][j] = *func* (x[i][j]) |
| **Weighted sum** | x[i] = *func* (x[i], x[i-1], x[i+1], y[i], y[i-1], y[i+1]) |
| **Threshold** | {x[i], y[i], z[i]} = *func* (x[i], y[i], z[i]) |
| **Video filter** | x[i] = *func* (x[i]), y[i] = *func* (y[i]), z[i] = *func* (z[i]) |
| **Histogram** | x[y[i]] = *func* (x[y[i]], z[i]) |
| **Matrix power** | x[i][y[j]] = *func* (x[i][y[j]], x[i-1][w[j]], z[i]) |

TABLE I: Memory access patterns of the benchmarks we evaluate. The loop iterators of each kernel are given as $i$ and $j$. Function *func* is application-specific.

The circuits we consider are synthesized automatically from C code using the strategy by Josipović et al. [18]. Our LSQ implementation is specialized for dataflow circuits [17]. Because optimally sizing the LSQs is out of the scope of this paper, we manually choose the minimal power-of-2 size which allows for maximum achievable loop parallelism (i.e., all designs are pipelined to achieve the optimal loop initiation interval for the particular benchmark and memory configuration). We connect disambiguated memory instructions to separate dual-port BRAMs with a single-cycle read and write latency, either directly or through an LSQ.

We functionally verify all designs using ModelSim. We obtain the number of clock cycles from the simulation and the clock period (CP) from the post-routing timing analysis with Vivado to calculate the execution time. Vivado place-and-route gives us the resource usage (i.e., the FPGA slices, LUT, FF, and DSP count). All designs target a Xilinx Kintex-7 FPGA.

### C. Benchmarks

The designs that we evaluate are simple but realistic kernels from literature [12], [20], [10] which contain different memory access patterns, summarized in Table I: (1) *Memory loop* is the example from Figure 1 which we have discussed in Section II. (2) *Scalar multiply* reads values of a vector and rescales each value by a constant factor before storing it back into the same memory location. (3) *Image revert* is a nested loop in which the value of each image pixel (stored in a 2-dimensional array) is reverted by subtracting it from a constant. (4) *Weighted sum* updates each value of a vector to the weighted sum of itself and its neighboring vector values. The weights corresponding to each element are stored in a separate vector. (5) *Thresholding* is an edge detection kernel with a conditional statement inside a loop: if the pixel intensity is less than some fixed constant, it is replaced it with a black pixel. (6) *Video filter* is a simple video processing application which, in a nested loop, applies a filtering function on each video pixel. (7) *Histogram* calculates the weighted histogram of an array of features. The same histogram bin may be updated in consecutive loop iterations—this potential read-after-write dependence cannot be determined at compile time. (8) *Matrix power* computes a series of matrix-vector multiplications in a nested loops. The row and column coordinates of the read and written matrix elements are unknown at compile time.

### D. Results

Table II summarizes our comparison results. All cases where no memory optimization is applied (*Naive*) need a single, large LSQ. The number of LSQ ports corresponds to the total number of reads and writes within the kernel. The execution time suffers due to two effects: (1) the large number of LSQ entries degrades frequency [17] and (2) multiple ports simultaneously insist on the same LSQ and, consequently, the same dual-port BRAM, which causes memory port congestion and limits parallelism.

Standard memory analysis techniques (*Standard*) disambiguate memory accesses targeting different arrays. All applications with accesses to multiple arrays benefit from the reduced port count and, in certain cases, reduced LSQ depth. Consider, for instance, *Video filter*: the naive implementation had a large LSQ with 6 ports which connected three arrays to memory (i.e., $x$, $y$, and $z$, as indicated in Table I); this optimization step splits the single LSQ into three smaller LSQs (one for each of the arrays). Apart from disambiguating accesses of different memories, standard analysis determines read-only accesses which can be connected to memory separately from the LSQ. This is the case, for instance, for $x[0]$ in *Memory loop* and for accesses to arrays $y$ and $z$ in *Histogram* and *Matrix Power*. However, in all applications, all read and write instructions which access the same memory locations still require an LSQ.

In all applications apart from *Histogram* and *Matrix Power*, our technique (*This Work*) finds timing relations between instructions which enable us to simplify or to completely remove the LSQ from the memory interface, resulting in significant area savings (although remarkable, note that the area savings due to the removal of the complex LSQ circuitry are probably exaggerated by the simplicity of the kernels we consider). In *Weighted sum*, an LSQ is still required to handle the loop-carried RAW dependence between the load of $x[i+1]$ and store to $x[i]$ (see Table I), yet even a queue of minimal depth sustains maximal throughput. *Histogram* and *Matrix Power* always require an LSQ to handle memory dependences which cannot be determined at compile time (i.e., the kernels are not SCoPs and polyhedral analysis cannot extract the accessed indices). These examples are representative of cases where dynamic scheduling of dataflow circuits is superior to static HLS and an LSQ is essential—in any static approach, the loops cannot be pipelined due to the possible read-after-write dependences; in contrast, dataflow circuits exploit the LSQ to maximize parallelism, as others have noticed before us [18].

## VI. Related Work

Although LSQs are routinely exploited in out-of-order processors [23], they are useless for standard, statically scheduled HLS, as the produced circuits exhibit solely in-order behavior. Techniques for analyzing memory access patterns, such as alias analysis and memory dependence analysis [2], [8], as well as optimizations to improve memory bandwidth, such as array partitioning and memory reuse [7], have been extensively studied in the context of static HLS. Dataflow circuits can

| Benchmark | Optimization | Interface | CP ($ns$) | Execution Time ($\mu s$) | Speedup | Slices | LUTs | FFs | DSPs |
|---|---|---|---|---|---|---|---|---|---|
| **Memory loop** | No opt. | LSQ (d8, p4) | 6.7 | 13.6 | – | 1390 | 4755 | 1751 | 3 |
| | Standard | LSQ (d8, p2) | 6.3 | 9.5 | 1.4× | 1213 (-13%) | 3892 (-18%) | 1475 (-16%) | 3 |
| | This Work | – | 4.3 | 4.4 | 3.1× | 107 (-92%) | 316 (-93%) | 282 (-84%) | 3 |
| **Scalar multiply** | No opt. | LSQ (d8, p2) | 5.8 | 8.8 | – | 1092 | 3506 | 1512 | 3 |
| | Standard | LSQ (d8, p2) | 5.8 | 8.8 | 1.0× | 1092 (-0%) | 3506 (-0%) | 1512 (-0%) | 3 |
| | This Work | – | 4.0 | 4.0 | 2.2× | 100 (-91%) | 262 (-93%) | 317 (-79%) | 3 |
| **Image revert** | No opt. | LSQ (d8, p2) | 6.3 | 5.7 | – | 1013 | 3425 | 1455 | 0 |
| | Standard | LSQ (d8, p2) | 6.3 | 5.7 | 1.0× | 1013 (-0%) | 3425 (0%) | 1455 (-0%) | 0 |
| | This Work | – | 6.3 | 5.7 | 1.0× | 123 (-88%) | 392 (-89%) | 287 (-80%) | 0 |
| **Weighted sum** | No opt. | LSQ (d8, p7) | 6.5 | 71.5 | – | 1640 | 5272 | 2312 | 9 |
| | Standard | LSQ (d4, p4) | 5.4 | 48.6 | 1.5× | 628 (-62%) | 1776 (-66%) | 1442 (-38%) | 9 |
| | This Work | LSQ (d2, p2) | 4.0 | 36.0 | 2.0× | 313 (-81%) | 772 (-85%) | 1013 (-56%) | 9 |
| **Threshold** | No opt. | LSQ (d4, p6) | 13.6 | 136.5 | – | 440 | 1356 | 831 | 0 |
| | Standard | 3 LSQ (d2, p2) | 11.1 | 88.9 | 1.5× | 350 (-20%) | 917 (-32%) | 847 (+2%) | 0 |
| | This Work | – | 11.1 | 33.4 | 4.1× | 183 (-58%) | 587 (-57%) | 425 (-49%) | 0 |
| **Video filter** | No opt. | LSQ (d16, p6) | 8.8 | 23.9 | – | 4356 | 15546 | 3596 | 9 |
| | Standard | 3 LSQ (d8, p2) | 7.6 | 10.4 | 2.3× | 3408 (-22%) | 11282 (-27%) | 4366 (+21%) | 9 |
| | This Work | – | 6.6 | 6.2 | 3.9× | 385 (-91%) | 1073 (-93%) | 933 (-74%) | 9 |
| **Histogram** | No opt. | LSQ (d16, p4) | 8.6 | 60.5 | – | 3925 | 13677 | 3292 | 2 |
| | Standard | LSQ (d16, p2) | 7.8 | 15.1 | 4.0× | 3512 (-11%) | 12043 (-12%) | 3125 (-5%) | 2 |
| | This Work | LSQ (d16, p2) | 7.8 | 15.1 | 4.0× | 3512 (-11%) | 12043 (-12%) | 3125 (-5%) | 2 |
| **Matrix power** | No opt. | LSQ (d16, p5) | 8.7 | 32.9 | – | 4364 | 14697 | 3617 | 7 |
| | Standard | LSQ (d16, p3) | 7.7 | 15.1 | 2.2× | 3808 (-13%) | 12792 (-13%) | 3225 (-11%) | 7 |
| | This Work | LSQ (d16, p3) | 7.7 | 15.1 | 2.2× | 3808 (-13%) | 12792 (-13%) | 3225 (-11%) | 7 |

TABLE II: Timing and resources of dataflow circuits which exploit our memory interface optimization (*This Work*), compared to circuits with naively built memory interfaces (*No opt.*) as well as memory interfaces created using standard optimizations (*Standard*). The LSQs employed by each design are listed under *Interface*, together with the queue depth (e.g., d8 indicates an LSQ of depth 8) and the number of connected ports (e.g., p2 indicates that two memory accesses of the circuit connect to the LSQ). All LSQs are connected to a dual-port block RAM.

benefit from these techniques as well and we exploit some of them (e.g., alias analysis) in this work.

The conservatism of static scheduling damages performance when applications exhibit irregular memory access patterns. Many HLS approaches based on static scheduling have incorporated some dynamic behavior to handle certain classes of irregular applications. Dai et al. create application-specific data hazard detection logic [10] to resolve dependences during execution, whereas Liu et al. [21] generate multiple schedules which are interchanged dynamically, based on the actual dependences. On the other hand, latency-insensitive protocols have been explored as a way to create truly dynamically scheduled circuits [6], [9], [11]. Such circuits have no predetermined schedule and operations may execute in any order. To handle potentially dependent memory accesses, some approaches resort to serializing memory accesses at a performance penalty [16]. Others use LSQs to maximize parallelism [3], [17], but the conservative LSQ placement and sizing cause a significant resource overhead. We address this issue in this work by providing an analysis technique which simplifies the memory interface of dataflow circuits and reduces resource consumption.

## VII. Conclusions

In HLS, producing dataflow circuits is the key to unleash the performance that, in the world of programmable systems and on generic control-dominated applications, only out-of-order dynamically scheduled processors can achieve. On the memory side, aggressively reordering accesses in dataflow circuits implies the use of LSQs, qualitatively similar to those used in processors. The problem is that spatial dataflow circuits essentially need an LSQ port for every access, quickly making LSQs prohibitive in cost and terrible in performance, especially on FPGAs; the risk of negating any advantage dynamic scheduling is extremely concrete. It seems intuitive that in the case of spatial circuits one should be able to use information from the source code to disambiguate accesses and partition monolithic LSQs into smaller ones as well as bypass some of the LSQs. This paper has shown how to build such a network of LSQs for arbitrary applications. In particular, we exploit the fact that data dependences in the original code imply sequential execution of some accesses in the corresponding dataflow circuit; this guaranteed sequencing may remove the need for some LSQs or ports thereof. On average, we have shown that a careful design can reduce by 66% the design area and improve performance by a factor $2.8\times$ compared to a naive approach. Although dataflow circuits have been an object of research in various communities, from computer architecture to asynchronous circuit design, we believe that little has been done yet to understand the challenges of building appropriate memory interfaces for dataflow designs and minimizing their cost. This paper strives to make some progress in this direction.

## Acknowledgments

Lana Josipović is supported by a Google PhD Fellowship in Systems and Networking.

## REFERENCES


[1] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, first edition, 1998.

[2] J. R. Appel and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, first edition, 2001.

[3] M. Budiu, P. V. Artigas, and S. C. Goldstein. Dataflow: A complement to superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–86, Austin, Tex., Mar. 2005.

[4] P. J. Cameron. *Combinatorics: topics, techniques, algorithms*. Cambridge University Press, first edition, 1994.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24:1–24:27, Sept. 2013.

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-20(9):1059–76, Sept. 2001.

[7] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic memory partitioning and scheduling for throughput and power optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 697–704, San Jose, Calif., Nov. 2009.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, Apr. 2011.

[9] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 657–62, San Francisco, Calif., July 2006.

[10] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 189–194, Monterey, Calif., Feb. 2017.

[11] S. A. Edwards, R. Townsend, and M. A. Kim. Compositional dataflow circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 175–184, Vienna, Sept. 2017.

[12] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Pearson, fourth edition, 2017.

[22] The LLVM Compiler Infrastructure. *http://www.llvm.org*.

[13] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, pages 1–6, Chamonix, Apr. 2011.

[14] T. C. Grosser. *Enabling polyhedral optimizations in LLVM*. PhD thesis, 2011.

[15] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011.

[16] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu. Elastic CGRAs. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 171–80, Monterey, Calif., Feb. 2013.

[17] L. Josipović, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):125:1–125:19, Sept. 2017.

[18] L. Josipović, R. Ghosal, and P. Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 127–36, Monterey, Calif., Feb. 2018.

[19] L. Josipović, A. Guerrieri, and P. Ienne. Speculative dataflow circuits. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 162–71, Seaside, Calif., Feb. 2019.

[20] R. Kastner, J. Matai, and S. Neuendorffer. Parallel programming for FPGAs. *ArXiv e-prints*, arXiv:1805.03648, May 2018.

[21] J. Liu, S. Bayliss, and G. A. Constantinides. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. In *Proceedings of the 23rd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 159–62, Vancouver, May 2015.

[23] I. Park, C. L. Ooi, and T. Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings of the 36th International Symposium on Microarchitecture*, San Diego, Calif., Dec. 2003.

[24] L. Torczon and K. Cooper. *Engineering a Compiler*. Morgan Kaufmann, second edition, 2011.

[25] M. Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign*, pages 171–80, Cambridge, MA, July 2009.

[26] H. Wong, V. Betz, and J. Rose. Efficient methods for out-of-order load/store execution for high-performance soft processors. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 442–445, Kyoto, Dec. 2013.

[27] Xilinx Inc. *Vivado High-Level Synthesis*.