Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis

Steve Dai[1], Ritchie Zhao[1], Gai Liu[1], Shreesha Srinath[1], Udit Gupta *[2], Christopher Batten[1], Zhiru Zhang[1]

[1]School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
[2]Computer Science, Harvard University, Cambridge, MA

{hd273, rz252, gl387, ss2783}@cornell.edu, ugupta@g.harvard.edu, {cbatten, zhiruz}@cornell.edu

Abstract

Current pipelining approach in high-level synthesis (HLS) achieves high performance for applications with regular and statically analyzable memory access patterns. However, it cannot effectively handle infrequent data-dependent structural and data hazards because they are conservatively assumed to always occur in the synthesized pipeline. To enable high-throughput pipelining of irregular loops, we study the problem of augmenting HLS with application-specific dynamic hazard resolution, and examine its implications on scheduling and quality of results. We propose to generate an aggressive pipeline at compile-time while resolving hazards with memory port arbitration and squash-and-replay at run-time. Our experiments targeting a Xilinx FPGA demonstrate promising performance improvement across a suite of representative benchmarks.

1. Introduction

Over the past few years, high-level synthesis (HLS) has become an increasingly popular alternative to traditional register-transfer level (RTL) designs [4]. HLS automatically generates digital circuits from a behavioral specification, greatly improving productivity over the traditional tedious hardware design process. Pipelining is one of the most widely used optimizations in HLS because it allows successive loop iterations (or function invocations) to be overlapped during execution, effectively exploiting parallelism with fewer resources compared to outright hardware duplication.

Conventional HLS pipelining typically leverages modulo scheduling [12], a compile-time optimization which creates a static schedule for a single loop iteration that can be repeated at a fixed *initiation interval* (`II`). The modulo scheduling algorithm analyzes the program's control-data flow graph along with resource, data dependence, and other constraints to minimize the `II` while ensuring that the pipeline does not encounter hazards during execution. Specifically, the statically generated schedule must not allow multiple operations to access the same physical resource within a single cycle (*structural hazards*) and must ensure that dependences between memory loads and stores are not violated (*data hazards*). The need to avoid these two types of hazards on memory accesses often limit the throughput of the synthesized pipeline.

* Udit Gupta was affiliated with School of Electrical and Computer Engineering, Cornell University during the course of this work.

FPGA '17, February 22-24, 2017, Monterey, CA, USA
© 2017 ACM. ISBN 978-1-4503-4354-1/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3020078.3021754

```
1 for (j=0...num_edges){
2  int s = e[j].src;
3  int d = e[j].dst;
4
5  if (v[s]<0 && v[d]<0){
6   v[s] = d;
7   v[d] = s;
8 }}
```

(a) Source code.

Cycle		
0	`e[j].load`	(line 2/3)
1	`v[s].load`	(line 5)
2	`v[d].load`	(line 5)
3	`v[s].store`	(line 6)
4	`v[d].store`	(line 7)

(b) Schedule for one iteration.

Figure 1: **Maximal Matching example** — (a) Source code in C-like syntax. (b) Static schedule produced by conventional HLS pipelining with `II=4`. Only `load` and `store` operations are shown while others (e.g., comparisons) are omitted.

HLS pipelining makes extensive use of memory dependence and alias analysis to identify dependences and disambiguate memory accesses (for convenience, we use dependence and alias analysis interchangeably in subsequent discussions). Such techniques attempt to classify each pair of memory accesses as no-alias or must-alias if the analysis is conclusive, or may-alias if the analysis is inconclusive. Static alias analysis is able to return fairly accurate dependence information for programs with compile-time analyzable control flow and highly regular memory access patterns, allowing efficient pipeline schedules to be created. However, such static analysis techniques are ineffective against programs that contain conditional and/or data-dependent memory operations with memory addresses unknown at compile-time, making it difficult to prove the absence of aliases. As a result, the dependence information will be inexact and contains may-alias pairs that have to be treated as must-alias by the scheduler to ensure hazard-free execution under all circumstances.

While existing pipelining techniques are effective at generating high-throughput hardware for regular dataflow-centric applications with well-structured data access patterns, they cannot efficiently synthesize *irregular programs* (e.g. graph algorithms, data analytics, sparse matrix computations) because these programs exhibit data-dependent control flow, irregular memory dependence patterns, and dynamic workloads. In particular, irregular programs incur structural and/or data hazards caused by conditional and/or data-dependent memory operations whose occurrence pattern cannot be accurately predicted by static compiler analysis, even with advances in polyhedral model [10, 11]. To ensure functional correctness, the pipelining algorithm must conservatively assume that these hazards always occur, even if they rarely or never do in practice. Consequently, conservative static pipelining leads to pessimistic performance as the pipeline stalls needlessly to avoid hazards which may be *infrequent* during actual execution.

We illustrate this performance gap using Maximal Matching in Figure 1(a), a common graph algorithm that computes the set of independent edges without common vertices in a graph. The kernel examines the two endpoints of each edge of a graph and checks if

	Cycles																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
j=0	`e.ld`	`v.ld`	`v.ld`	`v.st`	`v.st`												
j=1					`e.ld`	`v.ld`	`v.ld`	~~`v.st`~~	~~`v.st`~~								
j=2									`e.ld`	`v.ld`	`v.ld`	~~`v.st`~~	~~`v.st`~~				
j=3													`e.ld`	`v.ld`	`v.ld`	~~`v.st`~~	~~`v.st`~~

(a) Execution following static schedule in (b) incurs 17 cycles. `II=4` for all iterations.

	Cycles																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
j=0	`e.ld`	`v.ld`	`v.ld`	`v.st`	`v.st`												
j=1					`e.ld`	`v.ld`	`v.ld`	~~`v.st`~~	~~`v.st`~~								
j=2							`e.ld`	`v.ld`	`v.ld`	~~`v.st`~~	~~`v.st`~~						
j=3									`e.ld`	`v.ld`	`v.ld`	~~`v.st`~~	~~`v.st`~~				

(b) Ideal execution incurs only 13 cycles. `II=2` after Cycle 4.

Figure 2: **Execution of Maximal Matching** — Assume a single-ported memory for each array. ~~`v.st`~~ indicates a `store` to array `v` that is not executed due to false conditional branch. (a) Execution following the static schedule in Figure 1(b). (b) Ideal latency-optimal execution.

they are marked. If not, the algorithm updates the vertices at the endpoints using two conditional stores. Note that there are conditional loop-carried dependences between the `load` operations on `line 5` and the `store` operations on `line 6/7`. However, these stores are executed infrequently for a dense graph because only a small subset of its edges are independent. Figure 1(b) shows the static schedule for one iteration of the loop. For this schedule, each array is mapped to a single-ported memory so only one memory access per array is allowed in each cycle. To avoid potential structural and data hazards, conventional techniques will pipeline this design to an `II` of 4 cycles, which results in a 17-cycle execution latency, as shown in Figure 2(a), for the first four iterations processing a fully-connected graph. Six cycles are unused because the condition on `line 5` in Figure 1(a) is evaluated false for all iterations except `j=0`, thus no stores need to be performed for those iterations.

To take advantage of the infrequent nature of conditional memory accesses and inter-iteration memory dependences in this case, a better solution would be to launch new iterations more frequently to increase the efficiency of the pipeline by saturating the available memory bandwidth. As shown in the ideal execution in Figure 2(b), aggressively launching a new iteration every two cycles from Cycle 4 onward reduces the execution latency to 13 cycles. However, aggressive pipelining causes structural hazards, when the `stores` from the current iteration collide with `loads` from the next iteration, as well as data hazards, when the loop-carried dependence is violated. For example, if the stores to array `v` in iteration `j=1` were executed, they would collide with the loads from array `v` in iteration `j=2`. Moreover, a dependent load in iteration `j=2` may read from an address in array `v` before a store in iteration `j=1` writes to the same address, violating the inter-iteration read-after-write dependence between these memory accesses.

We propose to address the performance gap between conservative and aggressive pipelining by *speculatively* executing each iteration, launching each iteration before hazard-free execution can be guaranteed, and rely on a hardware dynamic hazard resolution mechanism to resolve any hazard that actually occurs. To achieve high throughput using this approach, two problems must be addressed: first, aggressive pipelining must be performed without pessimistically assuming that conditional or data-dependent hazards always occur; second, hazards that actually occur must be detected and resolved appropriately at runtime .

In this paper we propose a set of synergistic techniques which enable dynamic hazard resolution in pipeline synthesis. We address the scheduling problem by *virtualizing* the memory interface to make memory accesses appear independent. Virtualization hides structural hazards and dependences between memory operations, allowing any conventional HLS tools to perform aggressive scheduling without the need for programmer intervention. We next introduce hazard resolution logic which resolves structural hazards via port arbitration and data hazards via pipeline squashing. The hazard resolution hardware is automatically generated based on the number of virtual memory ports, the type (read or write) of each port, and the possible data dependences between memory accesses.

While our techniques are generally applicable to structural and data hazards for any expensive or limited hardware resources, this paper emphasizes memory-related hazards, because memory ports constitute a scarce resource and memory dependences are a common limiting factor of pipeline throughput in irregular programs. In particular, we focus on irregular loops with conditional memory accesses and inter-iteration memory dependences whose access patterns cannot be asserted at compile time. Our approach works for truly dynamic data dependences for which speculation, hazard detection, and replay are necessary for high-throughput pipelined execution. Our techniques provide the most performance benefit when the conditional accesses and data dependences are infrequent. Our approach is especially relevant as FPGA devices continue to attain higher memory bandwidths [6]. Specifically, our major technical contributions are threefold:

1. We identify a considerable performance gap in the HLS of irregular programs due to conservative nature of static pipelining in face of infrequent data-dependent dynamic hazards.
2. To our best knowledge, we are the first to propose and study structural hazard resolution and speculative execution as dynamic pipelining techniques to bridge this performance gap.
3. We compose our generated RTL with pipelines synthesized by a commercial HLS tool to achieve significant performance improvement on a suite of irregular benchmarks.

The remainder of the paper is organized as follows: Section 2 illustrates our dynamic hazard resolution techniques; Section 3 examines implementation details and discusses experimental results; Section 4 discusses related work; Section 5 concludes with the overall insight of this work.

2. Dynamic Hazard Resolution for HLS

We propose three synergistic techniques to augment the HLS-synthesized pipeline with dynamic hazard resolution to address the performance gap caused by static alias analysis and scheduling. Figure 3 illustrates the overall architectural template for the augmented pipeline with Maximal Matching from Figure 1(a), composed of an accelerator synthesized with a virtualized memory interface connected to a hazard resolution unit (HRU) customized for the Maximal Matching application. The HRU can be further divided into a data hazard resolution unit (D-HRU) and a structural hazard resolution unit (S-HRU). The HRU dynamically resolves structural and data hazards that occur in the Maximal Matching pipeline and communicates with memory. Our approach does not require any modification to current pipelining algorithms. HRU logic is automatically generated based on the schedule of the synthesized pipeline and a set of may-alias memory access pairs obtained from static analysis and/or user-specified directives. Our techniques also benefit from more accurate alias analysis to decrease the number of may-alias pairs and reduce the complexity of the customized HRU. We will use Maximal Matching to illustrate the customizable architecture.

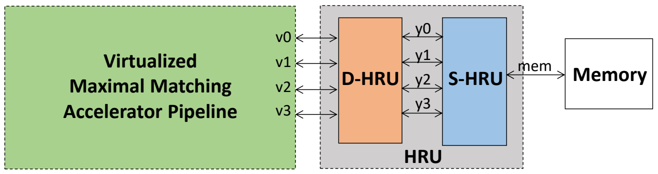

Figure 3: **Architectural template for the composed Maximal Matching accelerator** — HLS synthesized Maximal Matching pipeline with customized hazard resolution unit (HRU) consisting of a data hazard resolution unit (D-HRU) and a structural hazard resolution unit (S-HRU). A version with four virtual ports is shown.

```
for (j=0...num_edges){
 int s = e[j].src;
 int d = e[j].dst;

 if (v0[s]<0 && v1[d]<0){
  v2[s] = d;
  v3[d] = s;
}}
```

Figure 4: **Maximal Matching example** — Virtualized source code in C-like syntax. Compared to Figure 1(a), accesses to array `v` have been replaced by accesses to `v0`, `v1`, `v2`, and `v3`, respectively.

2.1 Memory Interface Virtualization

Although we have identified a significant opportunity in improving the performance of synthesized pipelines by deferring the handling of infrequent hazards to runtime, we cannot take advantage of this opportunity unless we can easily reduce the pipeline `II` below what is deemed safe by the HLS tool. While it is possible to modify existing pipelining algorithms for this purpose, doing so would not be generally applicable to any HLS flows. It will also limit our ability to evaluate our techniques leveraging existing HLS tools.

We propose to relax infrequent resource and memory dependence constraints by *virtualizing* the memory interface to enable aggressive pipelining. Virtualization is a source-to-source transformation that alters each conditional or may-alias memory operations to access its own independent array. This technique decouples physical memory ports from the scheduling process to remove memory port constraints and inter-iteration memory dependences. In the perspective of the scheduler, the transformed memory operations do not share a common resource and thus do not alias. Hiding these infrequent hazards from the pipeline scheduler enables aggressive `II` reduction. Although the virtualized design contains more memory ports than the non-virtualized design, these ports interface with the HRU and will be arbitrated for actual physical memory ports, as shown in Figure 3.

To relax the constraints in Maximal Matching, we can virtualize its memory port interface by modifying the source code as shown in Figure 4 where the accesses to the same array `v` are transformed into accesses to four different arrays `v0`, `v1`, `v2` and `v3`. With this transformation, the HLS tool no longer sees the dependence between those memory operations and no longer encounters memory port conflict because each memory operation accesses a different array. Assuming two physical memory ports and the same schedule as that in Figure 1(b), Figure 5 shows the execution trace of the first few iterations for virtualized Maximal Matching pipelined to `II=2`. There

	Cycles									
	0	1	2	3	4	5	6	7	8	9
j=0	e.ld	v.ld	v.ld	v.st	v.st					
j=1			e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~			
j=2					e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~	
j=3							e.ld	v.ld	v.ld	...

Figure 5: **Execution of virtualized Maximal Matching** — With two physical memory ports and design pipelined to `II=2`. Note that there exist two instances of potential dynamic data hazard between `v.st` in `j=0` and `v.ld` in `j=1`.

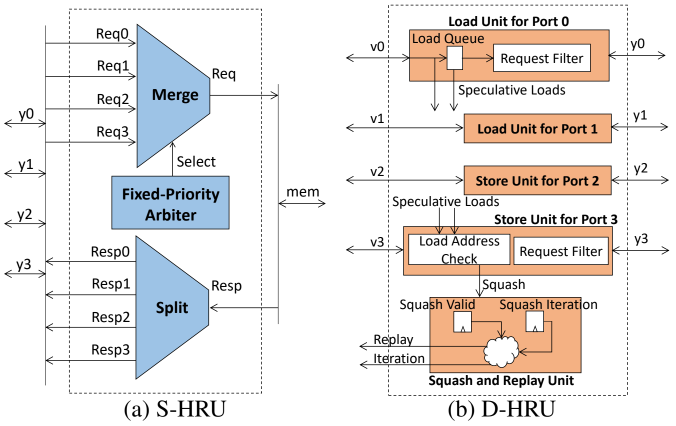

Figure 6: **Hazard resolution units (HRUs) for Maximal Matching** — (a) Structural hazard resolution unit (S-HRU). (b) Data hazard resolution unit employing speculative squash-and-replay (D-HRU).

exist two instances of potential dynamic data hazards between `v.st` in iteration `j=0` and `v.ld` in `j=1`.

2.2 Structural Hazard Resolution

We first discuss the implications of aggressive scheduling on resources. Having bypassed unnecessarily conservative resource constraints during scheduling, it is necessary to complement the synthesized virtualized pipeline with an S-HRU to resolve structural hazards caused by infrequent conditional memory accesses that actually occur during runtime. While our proposed scheduling scheme with virtualization relaxes the constraints on memory ports, the number of physical memory ports is limited in reality. An S-HRU is required to appropriately arbitrate memory accesses that present at the virtual memory ports into a limited number of available physical memory ports. If there is only one physical memory port available for Maximal Matching, `v.st` from iteration `j=0` cannot execute in parallel with `v.ld` from iteration `j=1` as shown in Figure 7. In this case, the S-HRU prioritizes `v.st` in Cycle 3 and stalls `v.ld` until Cycle 4. Subsequent operations are similarly arbitrated and stalled. As shown in Figure 6(a), the S-HRU implements a fixed-priority arbitration policy that always services request(s) from the earliest iteration(s). This policy preserves consistency for some speculatively executed memory accesses to reduce the need for squash-and-replay. The policy is also important for preventing deadlock and allowing the pipeline to flush in case of stall.

While dynamic hazard resolution is able to arbitrate competing memory requests, it also allows an aggressively pipelined design to capture unused memory bandwidth when a conditional memory access is not executed due to a false conditional branch. As shown in Figure 7, dynamic memory port arbitration allows `v.ld` operations in iteration `j=2` to capture the unused memory bandwidth from `v.st` operations that are not executed in iteration `j=1` due to a false

	Cycles									
	2	3	4	5	6	7	8	9	10	11
j=0	v.ld	v.st		v.st						
j=1	e.ld		v.ld		v.ld	~~v.st~~	~~v.st~~			
j=2					e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~	
j=3							e.ld	v.ld	v.ld	...

Figure 7: **Structural hazard resolution** — With a single-ported memory, memory access from an earlier iteration is prioritized while others are stalled. `v.st` from `j=0` is prioritized over `v.ld` from `j=1` even though they are in the same cycle on the HLS-generated schedule. `v.ld` operations from `j=2` capture the unused bandwidth of `v.st` operations from `j=1` that are not executed.

conditional branch. If the conditional accesses in Maximal Matching are infrequently executed, we can observe that the effective II will be very close to the target II of the aggressive pipeline.

The application-specific S-HRU architecture automatically generated for Maximal Matching is shown in Figure 6(a) targeting one physical memory port. Because there are four independent array accesses in the virtualized design in Figure 4, the customized S-HRU is composed of a merge unit with four input buses (Req0, Req1, Req2, and Req3) that arbitrates four incoming memory requests from the virtual request ports of the accelerator to the single physical memory request port (Req). Similarly, the S-HRU also includes a split unit that routes any memory response from the single physical memory response port (Resp) back to the appropriate virtual response port (Resp0, Resp1, Resp2, or Resp3) of the accelerator. A fixed-priority arbiter determines the priority of requests in the same cycle by always servicing request from the earliest iteration.

The proposed approach is able to elastically adapt to memory bandwidth that varies over time. This is especially applicable to emerging accelerator-rich architectures where many accelerators share the same memory ports [3]. For these architectures, statically assigning memory ports is either inefficient or impractical. In Section 3, we show that our hazard resolution techniques can effectively adapt to varying memory bandwidth.

2.3 Data Hazard Resolution

In addition to resource constraints, our aggressive scheduling scheme also relaxes inter-iteration dependence constraints by optimistically assuming that may-alias memory accesses would never alias. To ensure correct pipeline execution for the occasions when memory accesses do alias, however infrequent, we further complement the synthesized virtualized pipeline with a D-HRU to resolve runtime aliases not considered during static scheduling. In Figure 5, since the conditional accesses are actually executed in iteration j=0, v.ld in iteration j=1 is executed at Cycle 3 before v.st in j=0 is executed in Cycle 4. If the addresses of these may-alias memory accesses actually alias during runtime, the execution shown in Figure 5 would violate inter-iteration read-after-write dependence.

We propose to speculatively execute may-alias memory operations and perform squash and replay if the alias actually occurs during runtime. For Maximal Matching, we can speculatively execute the load operations from array v and squash and replay them only when memory aliasing is detected during the execution of a may-alias store. As shown in Figure 8, v.ld operations in iteration j=1 execute speculatively but are later squashed when v.st in j=0 executes in Cycle 4 and detects alias with the speculative v.ld from j=1 executed in Cycle 3. Due to the squash, iteration j=1 replays starting from Cycle 5, the cycle immediately after the alias is detected. On the other hand, v.ld operations executed speculatively in j=2 do not get squashed because v.st operations in iteration j=1 are not executed and cause no alias.

We propose a customized data hazard resolution unit with squash-and-replay capability (D-HRU) to enable a fully speculative pipeline. To prevent speculatively executed memory accesses from corrupting states, D-HRU is automatically generated to selectively include load queues and/or store queues to buffer speculatively executed requests until they are committed to memory. In addition, D-HRU selectively instantiates store-to-load forwarding unit to forward not yet committed store data. While the loads and stores reside in the queue, they are checked by other committing loads and stores to detect any mis-speculation. D-HRU implements a squash-and-replay mechanism that is able to cancel and replay any mis-speculated iterations.

	Cycles									
	3	4	5	6	7	8	9	10	11	12
j=0	v.st	v.st								
j=1	~~v.ld~~	~~v.ld~~	e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~			
j=2		e.ld			e.ld	v.ld	v.ld	~~v.st~~	~~v.st~~	
j=3							e.ld	v.ld	v.ld	...

Figure 8: **Speculative squash-and-replay** — The execution of v.st in Cycle 4 detects alias with v.ld executed in Cycle 3. Executed v.ld operations in Cycle 3 and 4 from j=1 are squashed (indicated by ~~v.ld~~). Iterations j=1 and onward are then replayed.

While the idea of speculation is borrowed from complex superscalar processors, the customized architecture of an HLS synthesized pipeline provides a unique opportunity to greatly simplify the complexity of the speculation logic. As such, our hardware generation algorithm is designed to instantiate the minimum subset and minimum number of the aforementioned hardware modules to support the alias pattern of a particular application for a specific schedule. In Figure 6(b), the customized D-HRU instantiates a Load Unit for port 0 to buffer incoming load requests for array v0 (v[s].load), because these requests may alias with store requests on port 3 from array v3 (v[d].store). The size of the queue is determined by the difference between the worst-case schedule distance from the load to any potentially aliased stores (3 cycles in this case) and the II of the pipeline. Thus we need only one entry in the load queue to buffer incoming requests at port 0 for II=2.

In Figure 6(b), the Store Unit for port 3 instantiates a Load Address Check unit that reads the speculative load addresses from the Load Queue for port 0 and check whether the current store request in port 3 (v[d].store) aliases with any speculative load requests from port 0 (v[s].load). If so, it sends a squash signal to the Squash and Replay Unit which squashes and replays the appropriate iterations. Request Filter is instantiated as part of the Load Unit or Store Unit to drop squashed requests. No store buffers need to be instantiated because the store operations are not speculatively executed. Load Unit for port 1 and Store Unit for port 2 implement load buffer and load check similar to those of port 0 and 3, respectively.

3. Experiments

While we implement a source-to-source transformation to virtualize the memory interface of the design, we develop a highly-parameterized hardware generation algorithm to automatically generate the minimum amount of HRU logic necessary for the particular application and compose the HRU logic with the synthesized pipeline to achieve high performance. The hardware generation algorithm leverages profiling and dependence analysis passes to extract infrequent may-alias memory access pairs, along with meta-data extracted from the schedule of the synthesized design to intelligently instantiate and connect HRU modules based on the architectural templates described in Section 2.

During hardware generation, the algorithm first extracts the necessary meta-data from the results of pipeline synthesis, including the II of the schedule, schedule distance between infrequent may-alias memory accesses, and the number of synthesized virtual memory ports. Then the algorithm generates the S-HRU based on the number of virtual and physical ports. Afterward, the algorithm instantiates a D-HRU if there exists dependence that must be resolved dynamically. The algorithm automatically customizes the composition of the D-HRU based on the number of virtual ports, a list of dependences, and the specification of each dependence. More specifically, a D-HRU is selectively composed of custom-size load/store queue, data forward unit, squash unit, replay unit, and filter units for resolving speculatively executed load and store operations.

We implement the hardware generation algorithm within a Python-based hardware modeling framework, which supports concurrent structural hardware modeling and provides a collection of tools for simulating and translating Python RTL models to Verilog [9]. To compose the synthesized pipeline with customized HRU, we instantiate a top-level model that integrates each HLS-generated accelerator design with appropriately-parameterized HRU models. Valid-ready interfaces are implemented to communicate between hardware units and stall the circuit when necessary.

Table 1: QoR comparison between baseline (`base`), alternative with structural hazard resolution only (`s-hru`), and alternative with structural and data hazard resolution (`all`) using a single-ported memory. Target clock period is 5ns. Designs include Sorting (SORT), Connected Components Labeling (CC), Histogram (HIST), Maximal Matching (MM), Matrix Power (MATPOW), and N-Queens Algorithm (NQ).

Design	Latency (cycles)			Clock Period (ns)			#LUTs			#FFs			#DSPs		
	base	s-hru	all	base	s-hru	all	base	s-hru	all	base	s-hru	all	base	s-hru	all
SORT	3917	3153	3084	4.2	4.1	4.2	624	651	879	717	806	992	4	4	4
CC	2114	1513	1131	4.9	4.6	4.6	899	1047	1224	874	1308	1472	0	0	0
HIST	78014	78014	39382	3.9	4.1	4.2	508	592	652	529	602	753	2	2	2
MM	1973	1440	1150	3.9	4.3	4.6	487	586	960	479	516	852	0	0	0
MATPOW	16018	10529	6131	4.4	4.4	4.4	826	1789	1850	1228	2232	2413	13	12	12
NQ	2280	1344	1344	4.3	4.4	4.6	512	563	609	406	445	555	1	1	1

This composition can be simulated and synthesized with conventional tools. Both the C-level programs and composed RTL models are synthesized with Vivado 2015.1 targeting Xilinx Virtex-7 FPGA. Quality of results (QoR) is obtained post place-and-route, and performance is obtained from cycle-accurate RTL simulation.

3.1 Benchmarks

To understand the implications of our proposed approach, we experiment with designs that exhibit data-dependent structural and data hazards from a range of application domains. Our experiments emphasize irregularity typically not found in regular applications where current HLS tools excel. We discuss two applications in detail.

```
1 for (i=0; i<N; ++i){
2  int m = feature[i];
3  float wt = weight[i];
4  if (m>THRESHOLD){
5   float x = hist[m];
6   hist[m] = x + wt;
7 }}
```

(a) `CountIf Histogram`

```
1 for (k=1; k<=m; k++){
2  for (p=0; p<nz; p++){
3   x[k][row[p]] +=
4      a[p]*x[k-1][col[p]];
5  }
6 }
```

(b) `Matrix Power`

Figure 9: **Irregular loop kernels with conditional hazards** — (a) `CountIf Histogram` constructs a weighted histogram of an array of features above a specified threshold. Array `hist` incurs conditional hazards. (b) `Matrix Power` computes the set of vectors $A^i\vec{x}$ for $i = [0, m]$. Array `x` incurs conditional hazards.

In Figure 9(a), each iteration of the `CountIf Histogram` kernel increases the bin indexed by the current feature value by adding the current weight if the feature value is above a specified threshold. There is an inter-iteration read-after-write dependence between the `load` on `line 5` and the `store` on `line 6`, which may cause data hazards if a subsequent iteration reads from the same histogram bin before the current iteration writes to it. While such memory aliasing is usually rare during histogram computation, it is impossible to assert the absence of such alias without prior knowledge of the sequence of feature values. Thus the HLS tool must create a conservative schedule such that a subsequent iteration reads from the histogram after the current iteration finishes writing to the histogram. Moreover, static scheduling unconditionally allocates a memory port for each memory access in a cycle even if the access is conditional. This results in inefficient utilization of memory bandwidth when the conditional accesses are predicated false at runtime.

In Figure 9(b), the `Matrix Power` kernel computes the set of vectors $A^i\vec{x}$ for $i = [1, m]$. With A stored as a coordinate list of `(row, column, value)` tuples, this kernel performs m sparse matrix-vector multiplications. The indirect memory accesses `x[k][row[p]]` and `x[k-1][col[p]]` on `line 3` and `line 4` present a potential inter-iteration read-after-write dependence because the result of $A^i\vec{x}$ depends on that of $A^{i-1}\vec{x}$. To ensure functional correctness without complete knowledge of the runtime values of `row[p]` and `col[p]`, the HLS tool must conservatively execute `load` from `x[k-1][col[p]]` only after `store` to `x[k][row[p]]` from a previous iteration has been completed.

3.2 Results

In Table 1, we first compare the achieved latency, clock period, and resource usage between the baseline designs, alternative designs with S-HRU only, and alternative designs with both D-HRU and S-HRU. The baseline designs consist of the highest-throughput pipelines generated by Vivado HLS, while our alternative designs are virtualized versions of the baseline designs synthesized with the same commercial tool but augmented with dynamic hazard resolution. With a single-ported memory, Table 1 shows that our alternative designs are able to achieve a significant latency reduction compared to the baseline designs with reasonable timing and area overhead. Note that the Histogram design excludes the condition in Figure 9(a) to present a case in which S-HRU provides no benefit.

The amount of speedup is dependent on the input data pattern, number of executed conditional memory accesses, and the available physical memory bandwidth. Table 1 breaks down how latency improves with only structural hazard resolution and both structural and data hazard resolution. For Histogram, including only S-HRU provides no performance benefit because the pipeline throughput is limited by a long inter-iteration dependence cycle. In this case, it is necessary to incur the overhead of the D-HRU. On the other hand, N-Queens reaps no benefit from speculation because structural hazard resolution has indirectly helped resolve any dynamic data dependence due to the limited number of memory ports. In this case, it is sufficient to include only the S-HRU.

By studying different design points, Table 1 demonstrates the inherent trade-off between performance gain and area. Our proposed techniques are important because loops that exhibit data-dependent hazards are often dominated by memory accesses. This is the reason that only a limited amount of compute resources are necessary. In addition, these loops usually contain only a couple of may-alias pairs, and thus require relatively lightweight hazard resolution logic that keeps timing and area well-contained.

We further study the effect of increasing memory bandwidth on performance and area by varying the number of physical memory ports. Figure 10 shows the speedup of each design for one to four memory ports normalized to the latency of the single-port case. For Sorting, Connected Components, Histrogram, and N-Queens, performance saturates beyond two memory ports because these designs contain at most two unconditional memory accesses in each cycle most of the time. These two unconditional accesses need to be arbitrated only when there are less than two physical ports. Having more than two physical ports does not help because there aren't enough pipelined parallel accesses in these designs to utilize any ports beyond the two required. On the other hand, Maximal Matching and Matrix Power continue to reap the benefit of increasing memory bandwidth beyond two ports because both designs contain many memory accesses that can execute in parallel. Having a large number of memory accesses at each cycle allows the design to take full advantage of the available bandwidth.

Table 2 compares the achieved clock period and resource usage for different numbers of memory ports for Maximal Matching. Al-

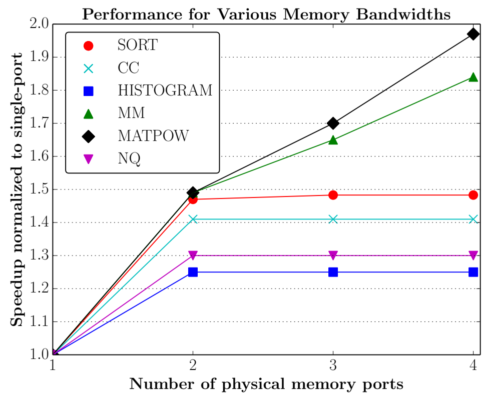

Figure 10: **Performance comparison for different memory bandwidths** — Speedup is normalized to the latency of single-ported memory case. The speedup saturates beyond two memory ports for designs with less pipelined parallelism or fewer memory accesses.

ternative designs with two to four ports incur `1.22x` to `1.70x` LUT counts and `1.02x` to `1.19x` FF counts with comparable timing. These overheads originate from the S-HRU shown in Figure 6(a) and apply equally to any benchmark. With an increasing number of physical memory ports, more complicated arbitration logic is needed to assign pending requests from the virtual ports to the available physical ports, which explains the increasing resource usage. According to Figure 10, Maximal Matching using four physical ports achieves over `1.8x` speedup compared to the single-ported case, which justifies the `1.70x` LUT and `1.19x` FF overhead.

Table 2: Timing and area overhead for increasing number of memory ports for Maximal Matching. No DSPs are used.

#Ports	Clock Period (ns)	#LUTs	#FFs
1	4.6	960	852
2	4.5	1171 (1.22x)	872 (1.02x)
3	4.5	1322 (1.38x)	941 (1.10x)
4	4.5	1629 (1.70x)	1010(1.19x)

4. Related Work

Many academic and commercial HLS tools, such as Vivado HLS [4] and LegUp [2], leverage static pipelining techniques to synthesize high-performance designs. Recent work in flushing-enabled pipelining [5] and multithreaded pipelining [13] extends these techniques to support dynamic memory behaviors. ElasticFlow enables the pipelining of irregular loop nests [14]. Zhao et al. synthesize irregular program by decoupling data structures from algorithms [15].

Alle et al. propose a runtime memory disambiguation technique where the address of a store is sent out before the store itself, allowing hardware to check whether an infrequently aliasing operation is expected to cause a hazard [1]. This information is leveraged to enable more aggressive pipeline `II`. We differentiate from this approach by considering structural in addition to data hazards for additional performance gain. We also study speculative execution to overcome the limitations pointed out by Alle et al.

Liu et al. extend polyhedral analysis to synthesizes pipelines that switches between aggressive (fast) execution, when hazards can be safely ignored, and conservative (slow) execution, when hazards are expected [7, 8]. Unlike this class of non-speculative stalling approach, our proposed approach does not require exact compile-time analysis to achieve high throughput. Our techniques tackle dynamic hazard resolution more broadly by emphasizing sophisticated runtime mechanisms complemented by relatively simple compile-time analysis. Nevertheless, our approach can benefit from the compile-time analysis proposed in this work.

5. Conclusions

Existing HLS tools rely on static pipelining techniques that extract parallelism only at compile-time, and are therefore not competitive for irregular programs with dynamic parallelism. As a result, we aim to create adaptive pipelining techniques that dynamically extract parallelism at run-time and efficiently handles statically unanalyzable program patterns. We address the problem of augmenting the HLS pipeline with application-specific dynamic hazard resolution to effectively resolve infrequent data-dependent structural and data hazards without sacrificing throughput. Our proposed approach achieve substantial performance improvement for a range of applications. For future work, it would be interesting to explore the tradeoff between maximum effective pipeline throughput and complexity of the hazard resolution logic. It would also be useful to develop scheduling techniques to reduce the overhead of hazard resolution.

Acknowledgments

This work was supported in part by NSF Awards #1149464, #1337240, #1453378, #1512937, and DARPA YFA D15AP00096.

References

[1] M. Alle, A. Morvan, and S. Derrien. Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2013.

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

[3] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-Rich Architectures: Opportunities and Progresses. *Design Automation Conf. (DAC)*, Jun 2014.

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

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

[6] M. Deo, J. Schulz, and L. Brown. Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge. *Intel White Paper*, 2016.

[7] J. Liu, S. Bayliss, and G. Constantinides. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, May 2013.

[8] J. Liu, J. Wickerson, and G. Constantinides. Loop Splitting for Efficient Pipelining in High-Level Synthesis. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, May 2016.

[9] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

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

[11] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-Based Data Reuse Optimization for Configurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2013.

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

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

[14] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang. ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2015.

[15] R. Zhao, G. Liu, S. Srinath, C. Batten, and Z. Zhang. Improving High-Level Synthesis with Decoupled Data Structure Optimization. *Design Automation Conf. (DAC)*, Jun 2016.