Enabling Adaptive Loop Pipelining in High-Level Synthesis

Steve Dai, Gai Liu, Ritchie Zhao, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
Email: {hd273,gl387,rz252,zhiruz}@cornell.edu

***Abstract*—Loop pipelining is an important optimization in high-level synthesis (HLS) because it allows successive loop iterations to be overlapped during execution. While current HLS pipelining approach achieves high performance for loops with regular and statically analyzable program patterns, it remains challenging to pipeline loops with irregular memory accesses, irregular dependence patterns, and unbalanced workload. The lack of support for dynamic program behaviors results in conservatively synthesized pipelines that sacrifice performance for maintaining presumed regularity. In this paper, we survey some of our recent work that addresses these challenges using a coordinated dynamic-static approach for enabling high-throughput pipelining of irregular loops. We propose to augment the HLS pipeline with dynamic scheduling to adapt to data-dependent behaviors, while employing static compile-time optimizations to minimize the hardware overhead associated with runtime optimization. Experimental results demonstrate that our proposed techniques can significantly improve effective pipeline throughput while conserving hardware resources.**

I. Introduction

The persistent effort to improve performance-per-watt has led to widespread adoption of heterogeneous architectures in which complex and specialized hardware accelerators play a central role in delivering the next leap in performance and energy efficiency. However, the promise of architectural heterogeneity comes at the expense of the agility of hardware/software development. As the number and diversity of hardware accelerators continue to scale alongside rapid and disruptive architectural changes, it is becoming increasingly difficult, if not already unsustainable, to design these accelerators with the traditional manual register-transfer-level (RTL) methodology. The need to wrestle with low-level RTL details prevents designers from exploring a sufficient number of design points necessary for identifying the appropriate tradeoff.

High-level synthesis (HLS) has emerged as a promising alternative to the RTL design methodology to enable productive modeling of hardware. HLS raises the level of input abstraction from hardware to software by automatically transforming a software program in C/C++/SystemC to a timed hardware design in Verilog or VHDL. HLS allows designers to quickly describe hardware using software languages and apply various optimizations using hardware-specific directives provided by the HLS tool.

Loop pipelining is one of the most important optimizations in HLS because it allows successive loop iterations to be overlapped during execution. Loop pipelining is typically performed by modulo scheduling, a software pipelining techniques that creates a static schedule for a single loop iteration; the same schedule can be repeated for subsequent iterations at a constant initiation interval (`II`) [1]. As loops are common in typical HLS applications, loop pipelining is a vital in enabling high-performance yet area-efficient hardware.

The conventional HLS pipelining approach employs static analysis to extract program behavior and is well-suited for applications that exhibit structured computation and data access patterns. In contrast, irregular programs are characterized by less-regular data and computation patterns that are often unknown until runtime, presenting fundamental challenges to existing pipelining techniques which mainly rely on static compile-time analysis and optimizations to exploit parallelism and data locality. Without the means to adapt to dynamic program behavior at runtime, HLS must pipeline irregular programs conservatively with lower throughput to ensure correctness at the cost of performance.

In this paper, we describe our synergistic approach to enabling adaptive loop pipelining by reviewing several of our work in this direction [2], [3], [4], [5]. On one hand, we augment the HLS-synthesized pipeline with run-time optimizations to exploit and adapt to data-dependent behaviors. On the other hand, we employ compile-time optimizations to minimize the overhead introduced by more intelligent hardware. The combination of dynamic and static optimizations promises significant improvement in performance while keeping hardware complexity in control. Specifically, we will describe techniques to address the following challenges in HLS pipelining: 1) pipelining with variable memory access latency [2], 2) pipelining with variable-bound inner loop [3], [4], and 3) hazards-aware pipelining [5].

The rest of the paper is organized as follows: Section II provides a more detailed motivation for the adaptive pipelining problem; Section III illustrates techniques to address each of the three challenges; Section IV surveys the line of related work in dynamic HLS; Section V provides additional insights into the problem.

II. Motivation

Listing 1 demonstrates a typical loop pipelining scenario in which the size of the workload, memory access pattern, and dependence pattern are statically analyzable. First, this loop nest has fixed loop bounds in Lines 1 and 2. As a result, the total number of loop iterations and memory accesses are known at compile-time. Second, array accesses in the loop body in Line 3 are indexed by loop induction variables and do not depend on any input data. The absence of data-dependent memory accesses allows HLS to partition memory for best throughput and optimize the order of accesses for the best data locality. Third, the distance of the inter-iteration read-after-write dependence in Line 3 for array `A` is constant. The regular dependence pattern between `A[i][j-1]` and `A[i][j]` dictates that HLS can parallelize across the first dimension of array `A`.

```
1 for(i=0; i<4; i++){
2   for(j=1; j<4; j++){
3     #pragma pipeline
4     A[i][j]=A[i][j-1]*j;
5   }
6 }
```

Listing 1. Pipelining inner loop

```
1 for(i=0; i<4; i++){
2   #pragma pipeline
3   A[i][1]=A[i][0]*1;
4   A[i][2]=A[i][1]*2;
5   A[i][3]=A[i][2]*3;
6 }
```

Listing 2. Unrolling inner loop

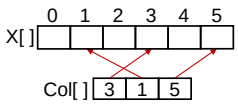
As shown in Listing 1, HLS can pipeline the inner loop of the loop nest to target one inner loop iteration per cycle. This approach achieves parallelism across different inner loop iterations while the outer loop iterations execute sequentially.

On the other hand, HLS can also pipeline the outer loop by first unrolling the inner loop as shown in Listing 2. This approach allows HLS to target one outer loop iteration per cycle. Regardless of the loop level at which pipelining is applied, HLS assumes fixed memory access latency to facilitate the creation of a static schedule for hardware to be synthesized.

Unlike the program in Listing 1, however, irregular programs operate on less-regular data structures and exhibit data-dependent workload, irregular memory access patterns, irregular data dependences, as well as difficult-to-predict memory access latency. While significant amounts of parallelism exist in many irregular programs, the potential data locality and parallelism cannot be easily uncovered and exploited using static techniques. In these cases, static pipelining must conservatively assume the worst-case program behavior, resulting in low-throughput pipelines.

1) Irregular memory access patterns: Listing 3 shows a sparse matrix vector multiplication (SPMV) kernel and illustrates the irregular memory access patterns in array `X`. Specifically, the matrix `A` is stored in the compressed sparse row format and requires an indirect array access `X[Col[c]]` to read an element from vector `X`. Due to poor data locality, this indirect access potentially incurs high cache miss rate. Such variable-latency operation stalls the entire HLS pipeline and leads to substantial performance degradation.

```
1 for(r=0; r<N; r++){
2   sum = 0;
3   for(c=Row[r]; c<Row[r+1]; c++)
4     sum += A[c]*X[Col[c]];
5   O[r] = sum;
6 }
```



Listing 3. SPMV code

2) Irregular workload patterns: Loops in an irregular program may have uneven work distribution across different loop iterations. In the SPMV kernel in Listing 3, the trip count of the inner loop is the number of non-zero elements in each row of the input sparse matrix. This trip count (i.e. workload) varies for different iterations. Because current HLS tools allocate hardware resources at compile time, it must target the worst-case trip count. Irregular workloads across iterations result in suboptimal resource utilization in the synthesized hardware.

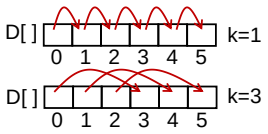
3) Irregular dependence patterns: Irregular programs often exhibit irregular data dependence patterns, especially in inter-iteration dependences. Examples include (1) infrequent dependences, and (2) dependences with variable iteration distance. Listing 4 shows a dynamic programming kernel, where the dependence distance for `D` varies with `k` at runtime.

A third example of irregular data dependence involves updates to shared data. Listing 5 shows a histogram kernel where each iteration updates a bin count. Different iterations can be executed out-of-order and in parallel as long as updates to individual elements of the `H` array are atomic. However, the dependence analysis techniques in modern HLS tools typically only handle static, affine data access patterns. The irregular parallelism in Listing 5 will not be discovered by the tool, resulting in low performance in the synthesized hardware.

III. Adaptive Pipelining Techniques

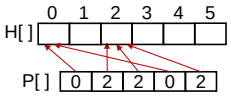
It is evident that irregular programs pose serious challenges to current pipelining techniques. In this section, we describe three complementary techniques to address different aspects of this challenge. While differing in purpose, all three techniques

```
1 for(k=0; k<N; k++){
2   for(i=0; i<N; i++){
3     val = f(D[i], D[i-k]);
4     D[i] = max(D[i], val);
5   }
6 }
```



Listing 4. Dynamic programming code

```
1 for(i=0; i<N; i++){
2   c = category(P[i]);
3   H[c]++;
4 }
```

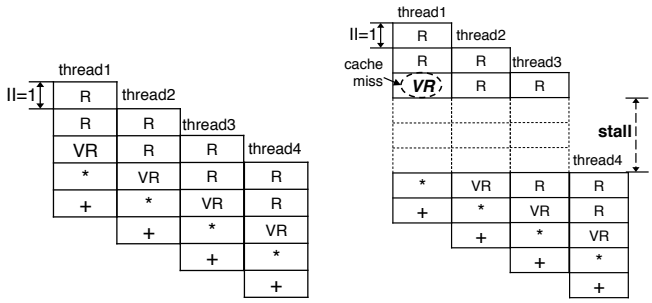


Listing 5. Histogram code

aim to synergistically combine dynamic and static optimizations to create high-performance yet area-efficient pipelines. This is done by synthesizing additional hardware to dynamically adapt pipeline execution to runtime-dependent program behaviors. The overhead of the new hardware is carefully minimized using both theoretical analysis and specialization to the target application.

A. Multithreaded Pipelining: Pipelining with Variable Memory Access Latency

As described in Section II-1, the need to access data from external shared memory incurs difficult-to-predict latency that is not expected by conventional pipelining techniques. As these techniques assume in-order execution of loop iterations and that the schedule is executed in lockstep, the entire pipeline is inevitably stalled until the completion of the variable-latency operation. As shown in Figure 1, while conventional HLS can pipeline the SPMV loop to `II=1`, the generated schedule can be executed without any stall only under the ideal condition where all data can be obtained perfectly with a fixed latency. If an internal stall occurs due to a cache miss, the current iteration as well as all subsequent iterations must be stalled to wait for the data. Throughput suffers as a result of the stall.



(a) Normal pipeline execution (b) Pipeline stall

Fig. 1. Pipeline stall due to the external memory access [2].

While reserving additional pipeline stages for each variable-latency operation may be a viable alternative, a deeper pipeline would come with additional hardware overhead. On one hand, it would be difficult to predict exactly how many stages should be added. On the other, a deeper pipeline still enforces in-order execution, which results in sub-optimal throughput based on our experiments. As a result, we introduce out-of-order thread execution to improve performance. Consider each iteration as a thread, the proposed multithreaded pipelining scheme allows subsequent threads to proceed when the current thread is blocked. More specifically, the blocked

thread releases its occupied resources by saving all live values at the time of the suspension to a context buffer so that the subsequent threads can continue without unnecessary stalling. The suspended thread will be swapped back into the pipeline once the variable-latency operation is completed.

As shown in Figure 2, a context buffer added alongside the pipeline provides the main support for out-of-order thread execution. The context buffer serves as a dedicated hardware storage to store the context of suspended threads. To minimize the overhead of the context buffer, we provide both an exact formulation and a heuristic to devise a context-aware schedule that minimizes the total bitwidth of all live values when a thread is suspended. Such a schedule reduces the required size of the context buffer and thus the hardware overhead incurred to obtain better throughput. Based on our experimental results, it is evident that multithreaded pipelining is able to achieve significant improvement in throughput while incurring much less resource overhead than deep pipelining. We refer the reader to [2] for detailed values.

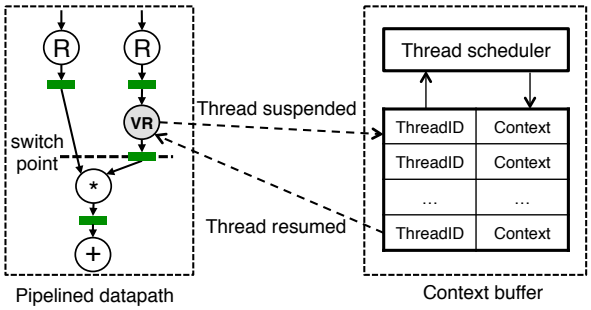


Fig. 2. Basic mechanisms of a multithreaded pipeline [2]: The pipelined datapath is extended with context switching support; Each context buffer saves the thread context when a thread is suspended; The thread scheduler decides which thread will be resumed if there are multiple ready threads in the buffer.

B. ElasticFlow: Pipelining with Variable-Bound Inner Loops

As discussed in Section II-2, irregular loops nests may incur unbalance workload due to variable inner loop bound. To pipeline the outer loop in this case, an alternative is to first transform the dynamic-bound inner loop into a fixed-bound loop using the worst-case bound, if known, that can handle any extreme cases of input data. However, this design choice presents two major problems: 1) Unrolling the inner loop based on worst-base bound is very inefficient in area. Many resources replicated from unrolling will spend most of their time idle. 2) Worst-case loop bound cannot be statically determined for many loops. The fixed-bound transformation approach cannot be applied without endangering correctness of execution.

To address these challenges, we synthesize an irregular loop nest into a multi-stage dataflow architecture called ElasticFlow. Figure 3 demonstrates the ElasticFlow architecture for a loop nest with two fixed-bound inner loops `A` and `C`, and one dynamic-bound inner loop `B`. While each stage in the architecture handles one of the three inner loops, we map the stage with dynamic-bound inner loop `B` to an array of loop processing units (LPUs). To adapt to the variable latency of the different instances of the inner loop, the architecture consists of a distributor to dynamically distribute work to and a collector to collect results from the LPUs based on their utilization. Each LPU contains the full datapath and control for executing the inner loop to completion. To optimize the number of LPUs, we employ an integer linear program (ILP) to statically allocate LPUs under hardware area constraints to meet the expected throughput requirement of the nested loop. As part of the optimization, the ILP decides whether an LPU should be instantiated such that it can be shared among different inner loops. This decision addresses the tradeoff between incurring more area per LPU and better work balancing.

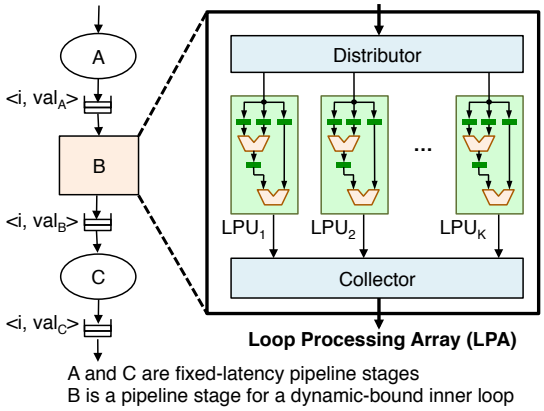


Fig. 3. **ElasticFlow Architecture** [3], [4] – An irregular loop nest is transformed into a multi-stage dataflow pipeline. Each dynamic-bound inner loop is mapped to a loop processing array (LPA), which consists of multiple loop processing units (LPUs). The loop iteration ID (`i`) and live values (val_A,val_B,val_C) are passed through the FIFOs between pipeline stages.

For experiments, we compare the performance and resource usage between irregular loop nests generated by commercial HLS tool and our ElasticFlow approach. We also vary the number of LPUs for each benchmark to study the performance-area tradeoff for ElasticFlow. Based on these experiments, we observe that ElasticFlow consistently outperforms results from the HLS tool in terms of performance. We also observe that while baseline HLS results are bottlenecked by the dynamic-bound inner loops, increasing the number of LPUs with ElasticFlow proportionally improves the performance of most designs. Not surprisingly, ElasticFlow is able to achieve better performance due to enhanced load balancing. Detailed experimental results can be found in [3], [4].

C. Dynamic Hazard Resolution: Pipelining under Data and Structural Hazards

As described in Section II-3, dependence information from static compiler analysis is inexact for irregular programs. These programs contain "may-alias" pairs that are treated as "must-alias" by the HLS tool to ensure hazard-free execution under all circumstances. While existing pipelining approach must conservatively assume that these hazards always exist, they rarely or never do in practice. Consequently, static pipelining incurs pessimistic performance as the synthesized pipeline stalls needlessly to avoid hazards which may be infrequent during actual execution. From our experiments, the performance gap is indeed significant when the hazards are infrequent.

To address the performance gap caused by infrequent hazards, we first synthesize a speculative pipeline using conventional HLS pipelining by bypassing infrequent dependence and resource constraints. Doing so allow iterations without hazards to execute aggressively and achieve the highest possible throughput. We then augment the HLS-synthesized pipeline with dynamic hazard resolution unit (HRU) customized specifically for the particular application to dynamically resolve any hazards incurred by other iterations. Figure 4 illustrates the overall architectural template for the augmented pipeline. As

shown in the figure, the HRU can be further divided into a data hazard resolution unit (D-HRU) and a structural hazard resolution unit (S-HRU) and can be instantiated separately based on the types of hazards present in the program.

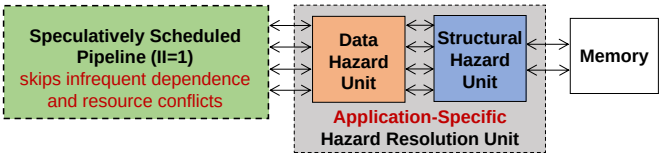


Fig. 4. **Architectural template for the augmented pipeline [5]** — HLS synthesized pipeline with customized hazard resolution unit (HRU) consisting of a data hazard resolution unit (D-HRU) and a structural hazard resolution unit (S-HRU).

Figure 5 shows an example of the HRU which is statically customized based on the specific application. As shown in Figure 5(a), an S-HRU consists of a merge unit that arbitrates incoming memory requests to the memory system, and a split unit that routes any memory response from the memory system back to the accelerator. These units provide the pipeline with the ability to achieve higher throughput by elastically adapting to varying demand for memory and more fully saturating the available memory bandwidth. The S-HRU is customized based on the number of ports required by the accelerator and the memory system to minimize any overhead introduced.

The D-HRU enables fully speculative pipeline execution by preventing the speculatively executed operations from corrupting states. As shown in Figure 5(b), D-HRU selectively including load queues and/or store queues to buffer speculatively executed memory requests until they are committed to memory. In addition, it also selectively instantiates store-to-load forwarding unit to forward not yet committed store data. While 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.

While our experimental results demonstrate significant speedup with hazards-aware pipelining for a number of irregular loops, the amount of speedup depends on the actual input data pattern and available memory bandwidth. Depending on the use case, one may achieve maximum speedup with either only S-HRU or D-HRU. Ultimately, it is a design tradeoff between performance gain and area. From our experience, 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. Furthermore, hazards oftentimes occur infrequently, which makes significant speedup possible. Detailed experimental results are available in [5].

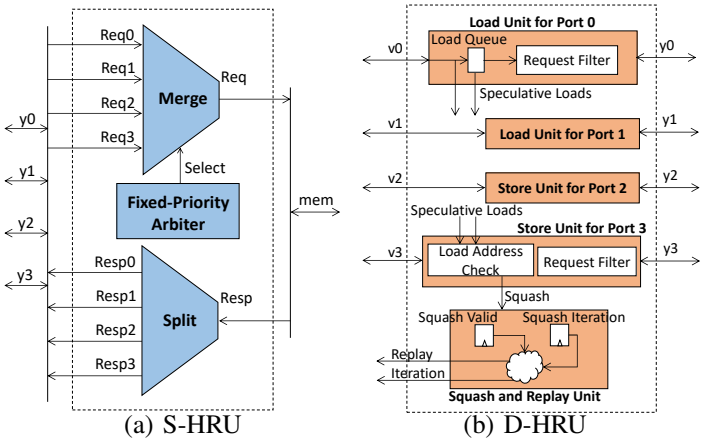


Fig. 5. **Hazard resolution units (HRUs) [5]** — (a) Structural hazard resolution unit (S-HRU). (b) Data hazard resolution unit employing speculative squash-and-replay (D-HRU).

IV. Related Work

Loop pipelining has been widely employed by both commercial and academic HLS tools, such as Vivado HLS [6] and LegUp [7], to improve the performance of the synthesized RTL. It is typically enabled by modulo scheduling, a software pipelining approach for increasing instruction-level parallelism by interleaving multiple iterations of a loop [1]. There are a number of work that continue to advance the frontier of loop pipelining [8], [9], [10]. Like simple loop pipelining, nested loop pipelining in software [11] has also been extended to hardware for accelerating hardware-synthesized loop nests [12]. Advances in polyhedral analysis enables automatic analysis, parallelization, streaming, and data reuse of regular loop nests with static bounds [13], [14]. Most recently, Lattuada and Ferrandi propose to unroll the outer loop while vectorizing inner loop instructions to improve performance. While this approach can be applied even if the inner loops cannot be parallelized, the number of inner loop iterations must not depend on the outer loop iteration [15].

There is a long line of work in increasing the effectiveness of HLS pipelining techniques for irregular programs that exhibit non-deterministic workload, unpredictable memory access latency, and irregular dependence. Dai et al. enable flushing for HLS pipeline to address the undesirable effects of pipeline stalls [16]. Halstread and Najjar develop the CHAT compiler to accelerator SPMV using a multithreaded datapath on FPGA [17]. Liu et al. decompose irregular loop into serial and parallel stages in which replicated datapaths are used in the parallel stages to accelerate the main computations of the loop [18]. Choi et al. develop the capability to generate multithreaded parallel hardware architectures in HLS using Pthreads and OpenMP in which threads are mapped to multiple copies of the same accelerator [19]. Kocberber et al. propose a decoupled pipeline architecture for implementing a reconfigurable accelerator for hash indexing [20]. Similar to ElasticFlow, a hashing unit distributes work to a parallel array of walker units and an output unit collects and combines the results from the walker units.

There is also significant interest in the effective handling of runtime hazards in HLS pipelines. Alle et al. propose runtime memory disambiguation, which allows the synthesized pipeline to check whether an infrequently aliasing operation is expected to cause a hazard and stall if deemed appropriate. This work aims to customize the memory disambiguation hardware so that it has "good enough" accuracy with minimal hardware footprint and critical path overhead [21]. Liu et al. propose to generate pipelines that dynamically select among multiple schedules during runtime. This approach enables aggressive (fast) pipeline execution when the pipeline is known to be hazards-free, and conservative (slow) execution when there is a possibility of hazards [22], [23]. Josipovic et al. present a specialized out-of-order load-store queue as an efficient interface between accelerator and memory to enable hazards-aware accelerator execution [24]. In general, there is a large volume of other work related to adaptive pipelining that cannot be enumerated due to the space limit.

V. Discussions

It is important to emphasize that adaptive pipelining underlies an inherent tradeoff between performance gain achieved and area overhead incurred. While this paper demonstrates that static pipelining is ineffective for irregular programs from emerging applications, static pipelining achieves state-of-the-arts performance for regular programs common in HLS-targeted applications. Rather than to abandon static pipelining, we have learned to embrace static scheduling for its efficiency, and apply adaptive pipelining as necessary to achieve the required performance target. While we provide static optimization techniques to minimize the overhead incurred by adaptive pipelining, the overhead cannot be completely eliminated and may still be substantial in many cases.

It is evident that the ideas behind our HLS adaptive pipelining approach bear similarities to those used in CPUs and GPUs, as the well-known advantages of these techniques apply equally to HLS pipelines. For example, we apply multithreading to switch loop iterations into and out of the HLS pipeline to increase concurrency and hide variable memory latency. We exploit data-level parallelism across different instances of inner loops to synthesize an array of parallel processing units similar to optimizations for GPUs. Furthermore, we also enable speculation to execute memory operations before the presence of hazards can be determined to prevent needless stalling of the pipeline for infrequent occurrence of hazards. As for CPU and GPU, all these techniques attempt to more fully utilize the available resources to minimize unnecessary resource idling that negatively impacts performance.

While our adaptive pipelining approach benefits from the advantages of aforementioned CPU and GPU optimizations, the application-specific nature of our techniques also holds a unique position in alleviating the disadvantages of the optimizations, namely the overhead in area, performance and timing introduced by more complex hardware mechanisms. For multithreading, we leverage static optimization to minimize the context width and thus the size of the context buffer. For ElasticFlow, we perform static resource allocation based on common-case loop bound and optimally determine using a linear program whether processing units should be shared among different loops. For hazards-aware pipelining, we instantiate the minimum amount of hardware based on the schedule distance between speculative pair of memory accesses to achieve the lowest-complexity hardware that maintains program correctness. By specializing the hardware overhead incurred for adapting the HLS pipeline to runtime program behaviors, we can achieve complexity-effective pipelining of irregular loops.

Acknowledgment

We would like to acknowledge Dr. Mingxing Tan for his contribution to the multithreading and ElasticFlow techniques summarized in this paper, Dr. Bin Liu for his contribution to the multithreading technique, as well as Shreesha Srinath and Prof. Christopher Batten for their contribution to the dynamic hazard resolution technique. This work was supported in part by NSF XPS Award #1337240, NSF CAREER Award #1453378, NSF CRI Award #1512937, DARPA Young Faculty Award D15AP00096, and a research gift from Xilinx, Inc.

References

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

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

[3] 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)*, 2015.

[4] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang, "Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017.

[5] 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," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[6] 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)*, 2011.

[7] 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)*, 2011.

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

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

[10] R. Zhao, M. Tan, S. Dai, and Z. Zhang, "Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis," *Design Automation Conf. (DAC)*, 2015.

[11] J. Ramanujam, "Optimal Software Pipelining of Nested Loops," *Int'l Parallel Processing Symp. (IPPS)*, 1994.

[12] D. Petkov, R. Harr, and S. Amarasinghe, "Efficient Pipelining of Nested Loops: Unroll-and-Squash," *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2001.

[13] 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)*, 2013.

[14] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2013.

[15] M. Lattuada and F. Ferrandi, "Exploiting Vectorization in High Level Synthesis of Nested Irregular Loops," *Journal of Systems Architecture (JSA)*, 2017.

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

[17] R. J. Halstead and W. Najjar, "Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads," *Intl'l Conf. on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2013.

[18] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August, "CGPA: Coarse-Grained Pipelined Accelerators," *Design Automation Conf. (DAC)*, 2014.

[19] J. Choi, S. Brown, and J. Anderson, "From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs," *Int'l Conf. on Field Programmable Technology (FPT)*, 2013.

[20] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases," *Int'l Symp. on Microarchitecture (MICRO)*, 2013.

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

[22] 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)*, 2015.

[23] 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)*, 2016.

[24] L. Josipovic, P. Brisk, and P. Ienne, "An Out-of-Order Load-Store Queue for Spatial Computing," *ACM Transactions in Embedded Computing Systems (TECS)*, 2017.