ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests

Mingxing Tan[1,2], Gai Liu[1], Ritchie Zhao[1], Steve Dai[1], Zhiru Zhang[1]

[1]School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
[2]Google Inc., Mountain View, CA
{mingxing.tan, gl387, rz252, hd273, zhiruz}@cornell.edu

Abstract

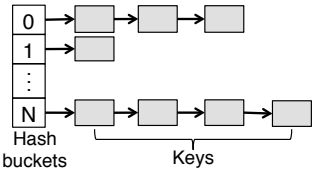
Modern high-level synthesis (HLS) tools commonly employ pipelining to achieve efficient loop acceleration by overlapping the execution of successive loop iterations. However, existing HLS techniques provide inadequate support for pipelining irregular loop nests that contain dynamic-bound inner loops, where unrolling is either very expensive or not even applicable. To overcome this major limitation, we propose ElasticFlow, a novel architectural synthesis approach capable of dynamically distributing inner loops to an array of loop processing units (LPUs) in a complexity-effective manner. These LPUs can be either specialized to execute an individual loop or shared amongst multiple inner loops for area reduction. We evaluate ElasticFlow using a variety of real-life applications and demonstrate significant performance improvements over a widely used commercial HLS tool for Xilinx FPGAs.

1. Introduction

Diminishing benefits of technology scaling and challenges in physical design over the past decade have prompted extensive research and development into architectural alternatives to the general-purpose processor. In particular, specialized accelerators have been employed in a multitude of applications and settings to deliver improvements in performance and energy efficiency. To meet the engineering demands of creating such hardware accelerators in a timely and cost-effective manner, high-level synthesis (HLS) has seen growing use over traditional register-transfer level (RTL) design methodologies. By automatically compiling specifications written in a software programming language into RTL and providing common architectural optimizations in the form of directives or pragmas, HLS is capable of significantly improving both the productivity and quality of hardware design.

One widely used optimization technique is pipelining, which allows successive loop iterations (or function invocations) to begin before the previous iteration has finished. Modern HLS tools can automatically pipeline a single loop or perfect loop nests, but are typically unable to efficiently address *irregular loop nests* that contain dynamic-bound inner loops. Unfortunately, such loop nests are commonplace in a variety of important application domains such as scientific computing, social analytics, and in-memory databases, as they are an inextricable part of key operations such as sparse matrix-vector multiplication, graph traversal, and hash lookup. Generating efficient accelerators for applications in these domains remains a serious challenge for contemporary HLS tools [20].

The difficulty presented by irregular loop nests is a consequence of the fundamental inability of existing pipelining techniques to handle elastic workloads. We illustrate this using an example. Figure 1(a) depicts a hash table which employs separate chaining, and Figure 1(b) shows `keysearch`, a kernel from the Memcached application [7] which performs a series of key lookups by first computing the hash bucket, and then pointer chasing over the keys in that bucket using a while loop. In this example we are interested in building a pipelined accelerator for this kernel which can achieve a throughput of one lookup per cycle. While it is possible to pipeline



(a) A common hash table implementation which uses separate chaining to resolve collisions.

```
for (k : keys_to_find) {
#pragma pipeline
  // Stage A
  hv = Jenkins_hash(k);
  p = hashtbl[hv].keys;

  // Stage B
  while (p && p->key!=k)
    p = p->next;

  // Stage C
  format_output(p);
}
```

(b) `keysearch` kernel using a `while` loop.

```
for (k : keys_to_find) {
#pragma pipeline
  hv = Jenkins_hash(k);
  p = hashtbl[hv].keys;

  for (i=0; i<M; i++) {
  #pragma unroll
    if (p && p->key!=k)
      p = p->next;
  }

  format_output(p);
}
```

(c) Modified `keysearch` using a `for` loop unrolled `M` times.

Figure 1. Irregular Loop Nest Example – `keysearch`, a kernel from the Memcached benchmark which performs a series of hash lookups; (a) Hash table representation; (b) Original code using a dynamic-bound inner loop to do pointer chasing; (c) Modified code using an unrolled fixed-bound inner loop of `M` iterations.

the outer loop without modifying the inner loop, the resulting design will be bottlenecked by the throughput of the inner loop, which is much lower than the throughput target.

An alternative is to transform the dynamic-bound loop in Figure 1(b) into a fixed-bound loop, which can be done by either manually changing the code or annotating static loop bounds in the HLS tool. The result is shown in Figure 1(c), where we have assumed `M` is the worst-case length of the collision chain in the hash table. While it is now possible to unroll the inner loop and achieve our throughput target, two major problems remain: (1) The design is very inefficient in area – a good hash function ensures that the vast majority of hash buckets contains at most one key, which requires only one loop iteration. However, we are forced to unroll `M` copies of the loop body to handle the extreme cases of the few buckets that contain multiple keys, appropriating resources which will spend most of their time idle; (2) For many loops, the worst-case loop bound cannot be statically determined. The bound-annotation approach cannot be applied in this case without endangering program correctness because the maximum number of keys in a hash bucket is often unknown.

The fundamental problem showcased by this example is that existing pipelining techniques attempt to statically allocate and bind resources for an elastic workload. This leads to unnecessary resource wastage as the design must handle the worst-case load, which is often unbounded. To efficiently address irregular loop nests, we argue that a new synthesis approach is needed for generating hardware capable of dynamically optimizing loop execution.

```
for (i = 0; i < num_imgs; i++) {
#pragma pipeline
  val = imgdiff[i];
  count = 0;

  while (val) {
    count++;
    val = val & (val - 1);
  }

  pc[i] = count;
}
```

(a) PC

```
for (i = 0; i < num_rows; i++) {
#pragma pipeline
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

(b) SPMV

```
for (i = 0; i < num_vertices; i++) {
#pragma pipeline
  v = vertice[i]; tmp = dist[i];

  for (e = v.edge; e; e = e->next) {
    j = e.target_vertice_id;
    if (dist[j] + e.weight < tmp)
      tmp = dist[j] + e.weight;
  }

  newdist[i] = tmp;
}
```

(c) SSSP

Figure 2. Representative Irregular Loop Kernels – (a) `Population Count (PC)` counts the ones in a bit vector, with an inner loop bound equal to the number of set bits. (b) `Sparse Matrix-Vector Multiplication (SPMV)` accesses each element in a sparse matrix, and has an inner loop bound dictated by the number of non-zero matrix entries; (c) `Single-Source Shortest Path (SSSP)` implements one iteration of the Bellman-Ford algorithm, which updates the distance of each node by examining each of its neighbors in the inner loop.

In this paper we propose *ElasticFlow* – a novel, complexity-effective technique which enables resource-efficient pipelining of irregular loop nests. Returning to the example in Figure 1(b), ElasticFlow generates a dataflow pipeline architecture containing an array of *loop processing units* (LPUs), each of which continuously executes an entire while loop to completion. Each iteration of the outer loop dynamically dispatches each of its inner loops (i.e., `Stage C`) to an LPU, allowing multiple outer loop iterations to execute in a pipelined fashion. ElasticFlow exploits decoupled pipeline parallelism to enable outer loop pipelining similar to the coarse-grained pipelined accelerators (CGPA) architecture [13] discussed in Section 6. In addition, ElasticFlow achieves improved resource efficiency and increased performance compared to CGPA by further enhancing the elastic nature of the pipeline; ElasticFlow supports out-of-order execution of outer loop iterations, as well as adaptive resource reallocation to allow an LPU to be appropriated for different inner loops at runtime. A summary of our major contributions include:

1. We propose a novel pipelined architecture and associated synthesis techniques to effectively accelerate irregular loop nests that contain dynamic-bound inner loops in HLS.
2. We propose an adaptive resource reallocation technique to reduce hardware overhead and improve pipeline performance for loop nests containing multiple dynamic-bound inner loops.
3. We systematically study the trade-off between performance and resource usage in terms of the number of LPUs and buffer sizes. Experimental results on a suite of practical application kernels demonstrate substantial performance improvements over a best-in-class commercial HLS tool for Xilinx FPGAs.

The rest of the paper is organized as follows: Section 2 provides an overview of irregular loop nests and the challenges they pose; Section 3 and 4 respectively detail the ElasticFlow architecture and synthesis approach; Section 5 presents experimental results; Section 6 examines related work, followed by conclusions in Section 7.

2. Irregular Loop Nests

In this paper we define an irregular loop nest as one which contains one or more dynamic-bound inner loops. Such loop patterns often arise from operations on less-regular data structures such as sparse matrices, graphs and hash tables. Figure 2 shows three real-world application kernels which exhibit this pattern. Notably, each application uses a different data structure which is usually sparse in practice – input values for `PC` tend not to span all the bits, sparse matrices for `SPMV` by definition contain very few non-zero entries compared to the number of matrix columns, and graphs for `SSSP` tend to be sparsely connected. This means that in the real world, the inner loops in these applications will almost never require the worst-case number of iterations.

The key challenge of pipelining irregular loop nests arise from the compile-time-unknown inner loop bounds, which inhibit static compiler transformations. Unrolling is also extremely inefficient even if the loop pattern possesses a known worst-case bound, as common-case execution will leave most resources idle. One approach is to partially unroll the inner loop and execute different iterations concurrently on multiple hardware copies. Unfortunately, this requires there to be no carried dependences in the inner loop, which is not the case in many practical applications as Figures 1 and 2 demonstrate. However, we note that in these examples there are *no outer-loop-carried dependences involving the irregular inner loops*, allowing these inner loop instances from different outer loop iterations to be executed in parallel. Indeed, we observe that such dependence patterns are commonplace in many important applications, leading us to consider an approach where we pipeline across different outer loop iterations by parallelizing the execution of multiple instances of an inner loop.

3. ElasticFlow Architecture

To address the challenges introduced by irregular loop nests, we present the ElasticFlow architecture, which implements such loop patterns as a multi-stage dataflow pipeline. Figure 3 provides an example which roughly corresponds to the `keysearch` kernel in Figure 1(b). Stage B implements a dynamic-bound inner loop as a loop processing array (LPA), which consists of multiple loop processing units (LPUs) as well as a *distributor* to distribute work to and a *collector* to collect results from the LPUs. Each LPU contains the full datapath and control for executing the inner loop. Other operations in the loop nest become fixed-latency pipelined stages (i.e., A and C), which can be synthesized using traditional pipelining techniques. We use FIFOs to connect successive stages and transfer outer loop iteration IDs and live variables.

3.1 sLPA Architecture

A natural approach is to map each dynamic-bound inner loop to a *single-loop processing array* (sLPA) containing multiple *single-loop processing units* (sLPUs) which execute that inner loop until completion. [1] Figure 4 illustrates pipelining using sLPAs with eight iterations of `keysearch`. Figure 4(a) shows the baseline approach taken by existing HLS tools, where every inner loop iteration executes serially on stage `B`. The throughput of stage `B` becomes the

[1] We focus on two-level loop nests for the rest of this paper, but ElasticFlow can be generalized to multi-level loop nests through hierarchical pipelining.

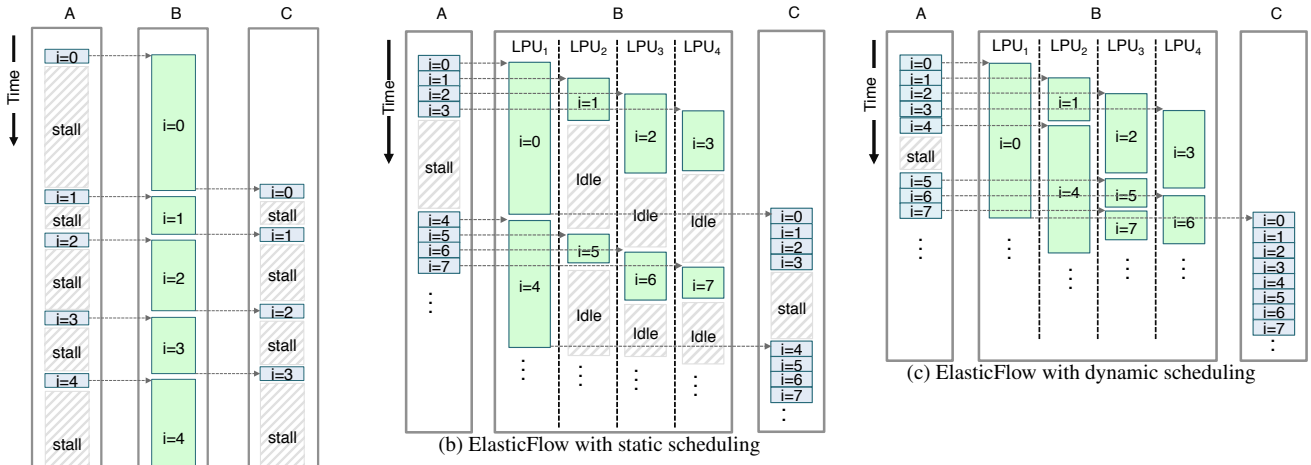


Figure 4. Execution on Different Pipeline Architectures – The irregular loop nest in this example is mapped to three pipeline stages (`S1`, `S2`, `S3`), where `S2` implements a dynamic-bound inner loop. Eight outer loop iterations are shown. (a) Baseline approach uses a sequential datapath for `S2`, resulting in frequent pipeline stalls due to low inner loop throughput; (b) ElasticFlow with four parallel LPUs for `S2` can improve throughput by overlapping different inner loop instances, but the LPUs are underutilized due to the static scheduling policy; (c) Dynamic scheduling can further increase throughput by improving LPU utilization.

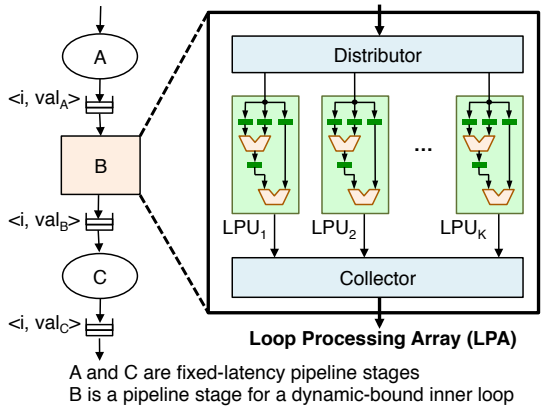


Figure 3. ElasticFlow Architecture – An irregular loop nest is transformed into a multi-stage dataflow pipeline. Each dynamic-bound inner loop is mapped to a loop processing array (LPA), which consists of multiple loop processing units (LPUs). The loop iteration ID (`i`) and live values (val_A,val_B,val_C)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

bottleneck, and the rest of the pipeline is frequently stalled. Figure 4(b) shows the sLPA approach using four sLPUs. In this example each outer loop iteration is statically assigned an LPU based on its ID modulo four. While throughput is improved, resource efficiency is poor as the workload is skewed towards LPU one, resulting in periods of idling on the other three LPUs. Indeed, it is not possible for a static scheduling policy to efficiently handle an arbitrary unbalanced workload. To guarantee resource efficiency, ElasticFlow employs a dynamic scheduling policy where an outer loop may dispatch its dynamic-bound inner loop to a free LPU. The resulting improvement in LPU utilization and throughput is shown in Figure 4(c). This runtime mechanism marks a fundamental difference between existing pipelining techniques in HLS and ElasticFlow. While it is possible to duplicate inner loop modules via unrolling and achieve parallelization in current HLS tools, the number of loop copies must be chosen at compile time to handle the worst-case loop bound, resulting in enormous resource inefficiency. In contrast, the number of ElasticFlow LPUs for a given dynamic-bound inner loop can target the common case, achieving maximum throughput most of the time while conserving resource.

It is important to note that although different outer loop iterations begin executing on an LPA in-order, they may finish out-of-order because the latency of each inner loop varies depending on the outer loop iteration, and cause incorrect results for many programs. To address this, the *collector* of an LPA can be configured to implement a *reorder buffer (ROB)* that ensures results are produced in the order indicated by the loop iteration ID.

Because each LPU continuously executes an entire inner loop to completion, inner loop carried dependencies are naturally honored. As discussed in Section 2, ElasticFlow assumes there are no outer loop carried dependencies involving stages synthesized to LPAs, and our synthesis algorithm ensures only loop nests adhering to this assumption will be mapped to this architecture.

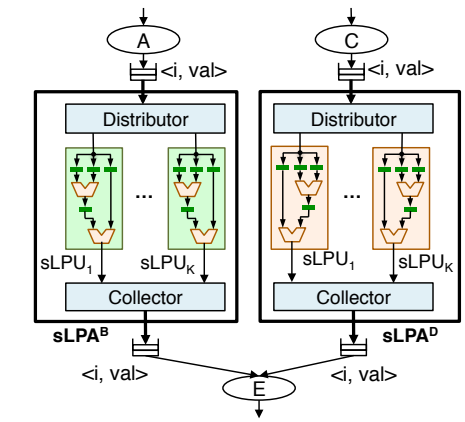
3.2 mLPA Architecture

The ElasticFlow architecture can further enable resource sharing at runtime by using *multi-loop processing units* (mLPUs), which have the ability to execute different dynamic-bound inner loops. We illustrate the advantage of this design with Database Join (`dbjoin`) in Figure 5(a), a common operation which combines the entries in two hash tables so they can be written to a single new table. The kernel uses two dynamic-bound inner loops (i.e., `B` and `D`), which is synthesized into two sLPAs with three LPUs each as in Figure 5(b). Figure 5(c) shows the execution of an example workload which is heavily unbalanced, such that loop B requires many more iterations than loop D. This causes idling in $sLPA^D$ as $sLPA^B$ bottlenecks pipeline progress.

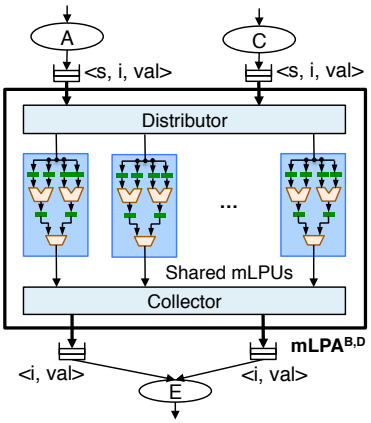
As before, this inefficiency is a consequence of static resource assignment for an elastic workload. Figure 5(d) shows an alternative architecture where the sLPAs are fused. The resulting *multi-loop processing array* (mLPA) contains four mLPUs, each of which is capable of executing either loop. An additional parameter, the stage IDs (`s`), must be passed into the mLPA to configure the mLPU to execute the desired loop. As Figure 5(e) demonstrates, the mLPUs can be dynamically reallocated for loop `B` or `D` depending on workload distribution, improving both resource utilizations and re-

```
for (i = 0; i < num_keys; i++) {
#pragma pipeline

  // stage A: look up hashtbl1
  k = input_keys[i]
  hv1 = Jenkins_hash1(k);
  p = hashtbl1[hv].keys;

  // stage B: dynamic-bound loop
  while (p && p->key!=k)
    p = p->next;

  // stage C: look up hashtbl2
  k = input_keys[i]
  hv2 = Jenkins_hash2(k);
  q = hashtbl2[hv].keys;

  // stage D: dynamic-bound loop
  while (q && q->key!= k)
    q = q->next;

  // stage E: merge the results
  return format_out(p, q);

}
```

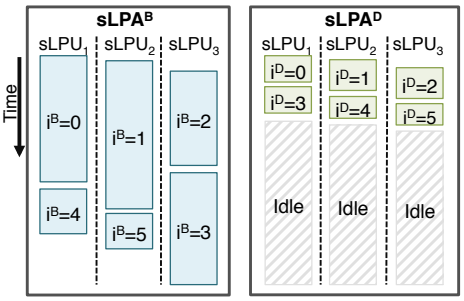
(a) Kernel for `dbjoin`



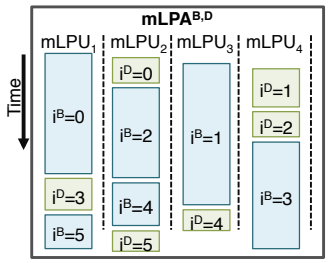
(b) Architecture without LPU sharing



(d) Architecture with LPU sharing



(c) Execution of six loop iterations on (b)



(e) Execution of six loop iterations on (d)

Figure 5. LPU Sharing and Adaptive Resource Reallocation – This example shows sharing and no-sharing architectures for the Database Join (`dbjoin`) kernel, (a) `dbjoin` code showing the five pipeline stages (`A, B, C, D, E`), where `B` and `D` contain dynamic-bound inner loops; (b) ElasticFlow architecture with two separate `sLPAs`: $sLPA^B$ for stage `B` and $sLPA^D$ for stage `D`; (c) Execution of six iterations on the no-sharing architecture; (d) ElasticFlow architecture with LPU sharing, where the `mLPA` consists of a set of `mLPUs` that can be shared between stage `B` and `D`. (e) Execution on the sharing architecture; `sLPU` = single-loop processing unit; `mLPU` = multi-loop processing unit; `sLPA` = single-loop processing array; `mLPA` = multi-loop processing array; `s` = stage ID (i.e., `B` or `D` for this example); i^B, i^D = loop iteration ID for stage `B` or `D`; `val` = live-in values for the downstream pipeline stages.

ducing pipeline stalls. Compared to sLPUs, an mLPU trades off additional area for increased throughput on unbalanced workloads. These trade-offs are further explored in Section 5.2.

4. ElasticFlow Synthesis

ElasticFlow synthesis maps an irregular loop nest to the architecture proposed in Section 3, which requires partitioning the loop nest into multiple stages, identifying inner loop candidates to form the LPAs, and synthesizing these loop bodies into sLPUs and mLPUs. The process considers resource sharing among different inner loops to optimize area usage, scheduling of workload onto LPUs to maximize throughput, and buffer sizing to avoid pipeline stalls.

Given a dependence graph for an irregular loop nest that captures both intra-iteration and inter-iteration data and control dependences, our synthesis algorithm first applies dependence analysis [8] to identify all dynamic-bound inner loops and partition each of these inner loops into separate stages. The subgraphs preceding or succeeding each dynamic-bound inner loop will be partitioned into their own stages, resulting in a coarse-grained directed acyclic graph (DAG) composed of stages. Figure 5 provides an example, where the kernel is partitioned into five stages (`A`-`E`), and the two dynamic bound inner loops are assigned two separate stages `A` and `B`. A stage containing a dynamic-bound loop will be synthesized as part of an LPA while other stages will be pipelined using conventional techniques. As such, ElasticFlow supports dependences among different outer loop iterations of all stages except `A` and `B` because `A` and `B` are bound to LPAs.

4.1 LPU Allocation

Under hardware area constraints, ElasticFlow synthesis enables static allocation of LPUs to meet the expected throughput requirement of the nested loop, TP, defined as the number of outer loop iterations per cycle. For the sLPA architecture, we allocate $U_i = [II_i \cdot (B_i - 1) + L_i] \cdot TP$ number of sLPUs for each inner loop i, where II_i denotes the achievable initiation interval of inner loop i from pipeline synthesis, B_i denotes the common-case bound of inner loop i from profiling the loop, and L_i denotes the latency in cycles of a single iteration of loop i, also obtained from synthesis. While a number of dedicated sLPUs proportional to the common-case bound of each loop achieves good performance for balanced, common-case workloads, this approach is less flexible under unbalanced workloads where loop bounds vary greatly over time. As demonstrated in Section 3.2, replacing dedicated sLPUs with shared mLPUs that are able to execute multiple loops allows us to maintain high throughput in this scenario. However, there is an inherent trade-off between performance and area. While creating more mLPUs improves performance by allowing adaptive assignment of resources for different loops depending on the workload, an mLPU typically consumes more area than an sLPU because it contains the hardware to execute multiple inner loops.

To address this trade-off, ElasticFlow synthesis optimizes performance given the resource usage of each type of LPU and the area of the sLPA architecture. We formulate an integer linear programming program as shown in Equation (1). LPUs are classified into different types depending on which loops share the particular LPU. For the example in Figure 5, LPUs are classified into three types: stage `B` only, stage `D` only, and stages `B/D`. The first two are sLPUs, and the latter is an mLPU. Since there are usually only a few inner loops, it is reasonable to enumerate the different types of LPUs. Given K types of LPAs, S_k^j denotes the area of resource j of an LPU in a type-k LPA and can be obtained from synthesizing each type of LPU individually. Given N inner loops, we also classify LPAs into N degrees where a degree-n LPA contains LPUs that can be shared among n loops.

In Equation (1), D_k indicates the degree of a type-k LPA with type-k LPUs, denoted LPA k. A_{total}^j represents the area constraint of resource j and is derived as a user-specified fraction of the number of resources required to synthesize the baseline sLPA architecture. n_k is a nonnegative integer variable that represents the number of LPUs needed for LPA k. r_{ik} is a binary variable that represents whether loop i is bound to LPA k. $T(i)$ denotes the set of LPAs on which loop i can execute. Equation (1b) constrains the total area of all the allocated LPUs. Equation (1c) prevents over-allocation of LPUs. Equation (1d) ensures that each inner loop is allocated to only a single type of LPA, and Equation (1e) enforces that loops are mapped only to compatible LPAs. The objective of the optimization is to maximize the weighted sum of the total degree of sharing and the total number of LPUs, where α and β are the weights that can be defined by the user. This objective acts as a proxy for performance by balancing the degree of sharing with the number of LPUs.

$$\text{maximize } \alpha \sum_{k=1}^{K} \sum_{i=1}^{N} D_k r_{ik} + \beta \sum_{k=1}^{K} n_k \quad \text{subject to} \tag{1a}$$

$$\sum_{k=1}^{K} S_k^j n_k \leq A_{total}^j \ \ \forall j \tag{1b}$$

$$\sum_{i=1}^{N} U_i r_{ik} \geq n_k \ \ \forall k \tag{1c}$$

$$\sum_{k=1}^{K} r_{ik} = 1 \ \ \forall i \tag{1d}$$

$$r_{ik} = 0 \ \ \forall k \notin T(i) \tag{1e}$$

4.2 Distributor and Collector Synthesis

Each LPA contains a distributor and a collector for assigning incoming inner loop instances to, and gathering results from LPUs, respectively. The distributor contains a *scheduler* which employs a dynamic work distribution policy – when an inner loop instance is available to be executed, the scheduler evaluates the busy/idle states of the LPUs in a round-robin manner and assigns the loop instance to the first idle LPU. For mLPAs, the scheduler gives priority to the inner loop with the lowest outer loop iteration ID.

As a result of dynamic loop bound, different inner loop instances may finish execution and exit their LPUs out of order. For applications that require results to be produced in order, we synthesize an ROB in the collector. Each LPU stores its results to a location in ROB in the order of increasing inner loop iteration ID, where the head of ROB stores the smallest loop iteration ID that is being processed. At each cycle, the ROB examines its head entry, and outputs the result if the head contains valid data. If the ROB is full, it applies back pressure to the distributor to prevent further work distribution until additional space frees up.

4.3 Deadlock Avoidance

ElasticFlow communicates live-in and live-out values in bundles between pipeline stages. For the sLPA architecture, artificial deadlock may occur if the number of in-flight outer loop iterations, consisting of those being processed by LPUs and those waiting in ROB, is greater than the total number of entries in the ROB. In this case, LPUs eventually stall because ROB does not have enough available entries to accept new data, and at the same time lacks all the data needed to proceed with reordering. To avoid such deadlock, we design the collector so that the number of in-flight outer loop iterations is limited to be no greater than the number of available entries in the ROB. This guarantees that every in-flight outer loop iteration will find an entry in the ROB once it finishes execution. As a result, the sLPAs are guaranteed to be deadlock free.

If we consider each sLPA as a single compute node, our dataflow architecture contains no cycles and forms a DAG. As proven in [12], such a system cannot deadlock if no inputs are filtered – an input to a node always results in an output. Because each live-in bundle always result in a live-out bundle per outgoing FIFO, the sLPAs are deadlock free and our system encounters no deadlock if it contains sLPAs only.

For mLPA architectures, we allocate one ROB for each inner loop sharing a particular mLPA, and limit the number of in-flight outer loop iterations of an inner loop to be no greater than the size of its corresponding ROB. If there is no data dependency between the shared inner loops, the resulting dataflow network forms a DAG, and the result from [12] guarantees no deadlock.

Now we show intuitively that our architecture remains deadlock free even if there exists data dependency between the shared inner loops. Assume that the mLPA architecture encounters a deadlock caused by two dependent inner loops that share an mLPA. In this situation, the oldest uncommitted outer loop iteration of the producer loop will be held up at the head of the ROB of its mLPA because execution cannot proceed to the corresponding consumer loop, which does not have an LPU to run on. However, the restriction on the number of in-flight outer loop iterations dictates that an mLPU will eventually free up when younger instances of the producer loop finish execution and move to the ROB. The consumer loop will then be able to execute using one of these freed LPUs and consume the producer result. This contradicts with the assumption that the consumer loop has been blocked due to insufficient resources. Hence, we conclude that the mLPA architecture is also deadlock free.

4.4 Buffer Sizing

It is important to suitably size the ROB to maximize the utilization of the LPUs. The distributor will be stalled when a long-latency loop iteration blocks the head of ROB. As a result, the LPUs cannot process new outer loop iterations, and the system becomes underutilized. However, there is no one-size-fits-all design since different applications may have drastically different loop latency patterns. Here we propose a profiling-driven approach to estimating the size of the ROB.

We profile a given application with representative sample datasets to obtain four key parameters for each dynamic-bound inner loop: the maximum latency L_{max}, the minimum latency L_{min}, the average-case latency L_{avg}, and the standard deviation σ of the loop latencies. Assume that there are K LPUs in the LPA, and consider the worst-case scenario where the head of ROB is blocked by a loop with L_{max}. In the meantime, the other $(K-1)$ LPUs are free to continue executing other inner loop instances. We assume that these inner loop instances have a latency of $(L_{avg} - 3\sigma)$, where the -3σ term accounts for the latency deviation from the mean. In situations where $L_{avg} - 3\sigma < L_{min}$, we use L_{min} instead to avoid over-pessimistic estimations. We define a parameter S using

the following equation:

$$S = \frac{L_{max}}{\texttt{max}(L_{avg} - 3\sigma, L_{min})}(K-1) + 1 \quad (2)$$

To increase LPU utilization by minimizing pipeline stalling, we require the ROB size to be no less than S. To reduce the logic overhead of ROBs, we round up S to the nearest power of two as the estimated ROB size.

A second consideration is the sizing of *delay lines*, which forward data that do not need to enter an LPA but must nevertheless proceed down the pipeline. A delay line is implemented as a FIFO that connects the stages before and after the LPA, and should be large enough to hold all in-flight data waiting to be consumed with the results from the LPA, or the pipeline will be stalled. The worst-case scenario for the delay line is similar to that of the ROB. Essentially, the delay line entries D should be no fewer than the number of possible in-flight loop instances in the LPA when the LPA is blocked by a loop instance with the maximum latency L_{max}, i.e., $D \geq S + K$. Compared with Equation (2), the additional term $+K$ corresponds to the fact that the K LPUs have also received new tasks by the time the long-latency loop instance finishes execution.

5. Experimental Results

Our setup leverages a widely used commercial HLS tool, which uses the LLVM compiler [10] as its front end and compiles a behavioral C/C++ program into Verilog or VHDL targeting Xilinx FPGAs. To our best knowledge, the tool employs modulo scheduling to pipeline a function or a loop nest, where all inner loops must be completely unrolled. Dynamic-bound inner loops will therefore prevent the functions or loop nests from being pipelined. We implement our ElasticFlow synthesis algorithm as an additional LLVM pass, which is applied after compilation and other optimizations. The pass automatically partitions the irregular loop nest into multiple stages as described in Section 4. Each stage in the loop nest is placed into a separate function. Stages with dynamic-bound inner loops will be mapped to sLPAs or mLPAs as described in Section 3, while other stages are pipelined using the default algorithm in the commercial tool. Our LLVM pass also inserts FIFOs between different stages and annotates `dataflow` directives to force different stages to run in a pipelined dataflow manner. We push the results through the same HLS engine to perform RTL code generation. The generated Verilog RTL design is implemented by Xilinx Vivado 2014.1 targeting a Virtex-7 FPGA device with 5ns target clock period. All timing and area numbers are obtained post place and route. We evaluate ElasticFlow using a variety of real-life applications from search engine, graph processing, database, scientific computing, image processing, and security. Table 1 briefly describes these applications. Each application contains one or more dynamic-bound inner loops that the commercial HLS tool cannot effectively pipeline.

Table 1. Descriptions of ElasticFlow Benchmarks.

Design	Description
bgcd	Stein's binary GCD algorithm for greatest common divisor
cfd	Computational fluid dynamics solver
dbjoin	Database join operation
digitrec	Digit recognition based on KNN algorithm
pagerank	Popular website ranking algorithm
spavg	Computing the mean value of each row in a sparse-matrix
spmv	Sparse matrix-vector multiplication kernel
sssp	Bellman-Ford single-source shortest path algorithm

5.1 Performance and Resource Usage Comparison

Table 2 shows the performance and resource usage comparison between the commercial HLS tool (i.e., the baseline approach) and our proposed ElasticFlow approach. We also vary the number of LPUs for each benchmark to study the performance-area tradeoff for ElasticFlow. With our synthesis flow, the user either use the default LPU allocation automatically generated by the tool, or manually specify the number of LPUs for each individual loop. Not surprisingly, for all designs, ElasticFlow consistently outperforms the baseline in terms of performance. We note that increasing the number of LPUs proportionally improves the performance of most designs, where dynamic-bound inner loops bottleneck the throughput of the pipeline without ElasticFlow.

Table 3. Elasticflow vs. Aggressive Unrolling Comparison – The unroll* approach applies a user-specified worst-case unroll factor based on the profiling results of the worst-case loop bounds (120 for `dbjoin` and 100 for `spmv`), which may be unsafe if the actual loop bound exceeds the user-specified unroll factor.

Design		LAT	CP	SLICE	LUT	FF
dbjoin	Unroll*	386	4.5	6679	10632	21187
	ElasticFlow	389	5.0	2019	6493	4239
spmv	Unroll*	365	4.4	2327	2884	6319
	ElasticFlow	372	4.4	632	1894	1412

An alternative approach to pipelining the outer loop is to fully unroll all inner loops, if the worst-case bounds of these loops are known. Obviously, such a complete unrolling approach can lead to good performance, but would incur significant area overhead. Moreover, the worst-case loop bounds may not be easily determinable for many real-life applications. Table 3 compares an aggressive unrolling approach with ElasticFlow on two benchmarks, where we optimistically specify the worst-case loop bounds based on profiling results. Specifically, the unrolling factor for `dbjoin` is 120, and `spmv` is unrolled by 100 times. While these unrolled designs may not function correctly when the actual loop bound exceeds the specified unroll factor, they are still useful references for measuring the quality of results of ElasticFlow. According to Table 3, ElasticFlow achieves similar performance with aggressive unrolling, but requires significantly less resource usage (by 3-4x).

5.2 LPU Sharing

We evaluate the effectiveness of our LPU sharing technique with two representative designs which contain more than one inner loop. `dbjoin` comprises two pointer-chasing inner loops that can share the same array of LPUs as shown in Figure 5(a); `cfd` contains two dynamic-bound inner loops that perform similar iterative fluid dynamics computations. As detailed in Section 4.1, we enforce an area constraint such that the total area of the mLPA designs are similar to that of the corresponding sLPA designs.

Table 4 demonstrates the latency reduction and resource usage of LPU sharing. For each benchmark, we provide two design points where the comparable sLPA design contains 8 or 16 LPUs. In addition, we are able to have equal numbers of mLPUs for `cfd`, since its two inner loops are structurally similar. For `dbjoin`, we allocate seven mLPUs (`dbjoin-A`) and 14 mLPUs (`dbjoin-B`), re-

Table 4. LPU Resource Sharing – Latency reduction and resource overheads for `cfd` and `dbjoin`. `cfd-A`: 8 mLPUs vs. 8 sLPUs; `cfd-B`: 16 mLPUs vs. 16 sLPUs; `dbjoin-A`: 7 mLPUs vs. 8 sLPUs; `dbjoin-B`: 14 mLPUs vs. 16 sLPUs.

Design	LAT Reduction	Slice Overhead	LUT Overhead	FF Overhead
cfd-A	34.7%	3.8%	9.3%	3.3%
cfd-B	31.5%	5.2%	8.9%	3.5%
dbjoin-A	21.3%	7.0%	9.7%	-10.5%
dbjoin-B	21.6%	5.7%	9.8%	-12.7%

Table 2. Performance and Resource Usage Comparison – `base` is the design generated by the commercial HLS tool; [`n`] represents the proposed `ElasticFlow` approach with `n` LPUs. `LAT` = latency of the entire design in # of cycles; `CP` = clock period in ns (target `CP` is set to 5ns); `SLICE` = # of slices; `LUT` = # of lookup tables; `FF` = # of flip-flops; `Speedup` = speedup of `ElasticFlow` over `base` in terms of latency.

Design	LAT	CP	SLICE	LUT	FF	Speedup	Design	LAT	CP	SLICE	LUT	FF	Speedup
bgcd-base	48770	4.5	500	1564	942		pagerank-base	702	3.7	181	432	543	
bgcd-[2]	24402	4.5	574	1808	1109	2.0x	pagerank-[2]	358	4.2	267	691	761	2.0x
bgcd-[4]	12228	4.9	778	2547	1357	4.0x	pagerank-[4]	193	4.3	441	1154	1197	3.6x
bgcd-[8]	6147	4.9	1161	3868	1853	7.9x	pagerank-[8]	113	4.5	806	2202	2071	6.2x
cfd-base	5059	4.6	3874	10595	10856		spavg-base	2604	4.4	1025	3067	2926	
cfd-[2]	2597	4.7	4703	12717	13139	1.9x	spavg-[2]	1336	4.9	1073	3261	3073	1.9x
cfd-[4]	1675	5.0	6534	17639	17447	3.0x	spavg-[4]	704	4.9	1184	3564	3281	3.7x
cfd-[8]	1675	5.4	9589	27167	26077	3.0x	spavg-[8]	480	4.9	1401	4295	3697	5.4x
dbjoin-base	2248	4.5	1180	3538	2319		spmv-base	2719	4.0	157	420	460	
dbjoin-[2]	1157	4.6	1328	3831	2667	1.9x	spmv-[2]	1370	3.8	243	620	596	2.0x
dbjoin-[4]	627	4.7	1503	4704	3191	3.6x	spmv-[4]	697	4.4	376	1019	868	3.9x
dbjoin-[8]	389	5.0	2019	6493	4239	5.8x	spmv-[8]	372	4.4	632	1894	1412	7.3x
digitrec-base	275402	4.0	312	807	510		sssp-base	780	3.8	169	445	502	
digitrec-[2]	137717	4.0	366	1018	624	2.0x	sssp-[2]	399	4.0	241	702	678	2.0x
digitrec-[4]	68877	4.4	456	1358	852	4.0x	sssp-[4]	208	4.2	416	1212	1030	3.8x
digitrec-[8]	34464	4.0	686	2097	1308	8.0x	sssp-[8]	122	4.4	756	2319	1736	6.4x

spectively. As shown in Table 4, using m-LPA can further improve the performance by 21%–34% with similar area, showing the effectiveness of the LPU sharing technique. For `dbjoin`, the mLPA designs require fewer FFs as we observe that many flip-flops are mapped to shift register LUTs (SRLs).

5.3 Reorder Buffer Sizing

Figure 6 demonstrates the performance impact of ROB sizing as well as the effectiveness of the ROB size estimation scheme described in Section 4.4. To simplify the ROB logic, we restrict the buffer size to be a power of two. As illustrated by the figure, our profiling-based approach for estimating the buffer size can reasonably predict the optimal ROB sizes that are free from stalling. For several benchmarks, the estimation matches exactly with the optimal ROB size. For `dbjoin` and `spavg`, the estimation provides a reasonable upper bound of the buffer size, which still ensures that the LPAs can achieve the best performance.

5.4 Comparison with CGPA

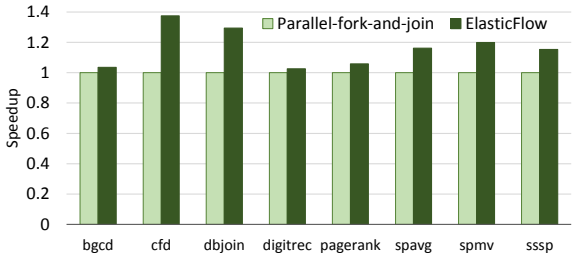


Figure 7. Performance Comparison between ElasticFlow and Parallel-Fork-and-Join – `ElasticFlow` is our approach; `Parallel-fork-and-join` implements the CGPA architecture [13] using sLPAs. We use eight LPUs for all the designs.

We note that the CGPA architecture can be implemented using sLPAs with a special distributor-collector scheme, termed as parallel-fork-and-join in [13]. The parallel-fork distributes inner loop instances based on a predetermined mapping between outer loop iteration IDs and the available LPUs (i.e., static scheduling illustrated in Figure 4(b)). The parallel-join collects outputs only after all LPUs have produced valid results. With parallel-join, ROB is not used although we still need to allocate sufficient amount of buffers in the distributor to temporarily hold the results from LPUs. Figure 7 shows that ElasticFlow can achieve up to 37% performance improvement compared to the alternative approach that resembles CGPA. We also observe that similar resource usage between two approaches. We attribute the additional speedup to LPU sharing with m-LPAs, and the more adaptive scheduling and load balancing enabled by ROBs.

6. Related Work

Loop pipelining is typically enabled by modulo scheduling [17], a software pipelining technique for extracting instruction-level parallelism across loop iterations, and is an important optimization implemented in various academic and commercial HLS tools, including Vivado HLS [4], Altera SDK for OpenCL [5], and LegUp [3]. Recent advances in pipeline flushing [6], multithreading [19], and runtime dependency analysis [1] aim to achieve even higher-performance designs, along with techniques to optimize area by reducing the usage of registers [21], LUTs [18, 22], and memory ports [2]. Despite these optimizations, existing approaches mostly focus on pipelining simple loops and are ineffective for more complex loop nests seen in many programs.

Like simple loop pipelining, nested loop pipelining was first introduced in the software domain [16] and later extended to hardware synthesis [14]. Recent optimizations in HLS employ polyhedral analysis to enable automatic parallelization, streaming, and data reuse of regular nested loops with affine data access patterns [15]. Such compiler analysis performs polyhedral code transformation to optimize the design for performance and area based on the program pattern. However, polyhedral analysis is performed at compile time and is mostly useful for regular loop nests with static bounds. Lattuada and Ferrandi propose a technique to parallelize irregular loop nests by unrolling the outer loop and vectorizing certain instructions [11]. However, this technique requires that the inner loop bound does not depend on the outer loop iteration, making it inapplicable to our benchmarks.

The CGPA framework generates coarse-grained pipelines for a loop nest by partitioning it into parallel and non-parallel sections. CGPA employs replicated data-level parallelism to create multiple identical copies of the parallel section and applies decoupled pipeline parallelism to separate the parallel and sequential sections with a set of FIFOs [13]. While CGPA is similar to ElasticFlow with sLPAs, ElasticFlow achieves additional performance improvement by enabling out-of-order execution and dynamic scheduling

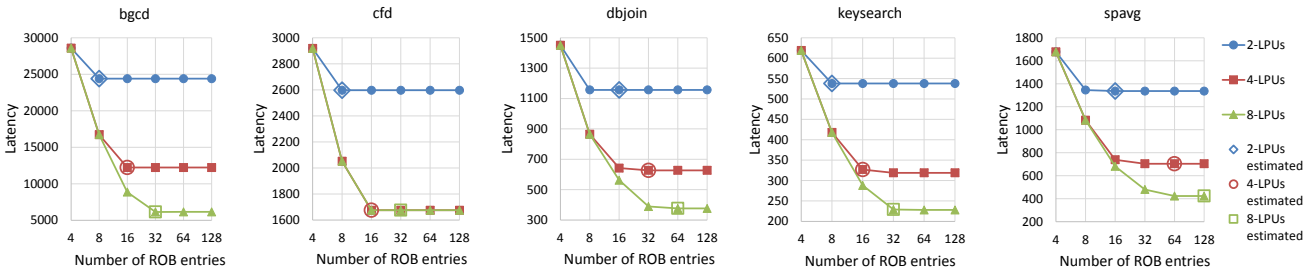


Figure 6. Performance Impact of Reorder Buffer (ROB) Size and the Estimated ROB Size using Equation (2).

of inner loop instances. In addition, ElasticFlow realizes better resource efficiency and potentially higher throughput by optimizing the allocation and sharing of LPUs with the mLPA architecture. We also study buffer sizing for both ROB and delay line and propose a runtime policy that guarantees the absence of deadlock.

Kocberber et al. propose Widx, a reconfigurable accelerator for hash indexing in database systems which uses decoupled pipeline architecture similar to ElasticFlow [9]. In Widx, a hashing unit distributes work to a parallel array of walker units, with the result combined in an output unit. While bearing some similarity, Widx is a specialized architecture for accelerating a single class of operation, whereas ElasticFlow is a technique for addressing a more general problem of pipelining irregular loop nests.

7. Conclusions

We presented ElasticFlow, a novel hardware architecture and associated synthesis techniques, which can efficiently address the pipelining of irregular loops nests containing dynamic-bound inner loops in HLS. ElasticFlow generates a dataflow pipeline architecture containing arrays of loop processing units, on which multiple instances of inner loops can be executed concurrently. We further study the complications of enabling out-of-order loop execution to improve throughput, and propose adaptive resource sharing schemes that enable the reallocation of loop execution units during runtime in response to workload imbalance for improved resource efficiency. Experimental results over a variety of real-life benchmarks show that ElasticFlow is able to achieve substantial performance improvement over a best-in-class commercial HLS tool targeting Xilinx FPGAs.

Acknowledgements

This work was supported in part by NSF CAREER Award #1453378, NSF XPS Award #1337240, and a research gift from Xilinx, Inc.

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] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 3(3), Sep 2010.

[3] 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.

[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] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown. OpenCL for FPGAs: Prototyping a Compiler. *Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 3–12, Jul 2012.

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

[7] B. Fitzpatrick. Distributed Caching with Memcached. *Linux journal*, 2004(124):5, Aug 2004.

[8] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[9] 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)*, pages 468–479, Dec 2013.

[10] C. Lattner and V. Adve. LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation. *Int'l Symp. on Code Generation and Optimization (CGO)*, pages 75–86, Mar 2004.

[11] M. Lattuada and F. Ferrandi. Exploiting Outer Loops Vectorization in High Level Synthesis. *Architecture of Computing Systems (ARCS)*, pages 31–42, Mar 2015.

[12] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain. Deadlock Avoidance for Streaming Computations with Filtering. *Int'l Symp. on Parallelism in Algorithms and Architectures (SPAA)*, Jun 2010.

[13] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August. CGPA: Coarse-Grained Pipelined Accelerators. *Design Automation Conf. (DAC)*, pages 1–6, Jun 2014.

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

[15] 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.

[16] J. Ramanujam. Optimal Software Pipelining of Nested Loops. *Int'l Parallel Processing Symp. (IPPS)*, pages 335–342, Apr 1994.

[17] B. R. Rau. Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture (MICRO)*, pages 63–74, Nov 1994.

[18] M. Tan, S. Dai, U. Gupta, and Z. Zhang. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2015.

[19] 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.

[20] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS. *Int'l Conf. on Field Programmable Technology (FPT)*, pages 362–365, Dec 2013.

[21] Z. Zhang and B. Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 211–218, Nov 2013.

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