Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures

Quan M. Nguyen
Massachusetts Institute of Technology
qmn@csail.mit.edu

Daniel Sanchez
Massachusetts Institute of Technology
sanchez@csail.mit.edu

ABSTRACT

Coarse-grain reconfigurable arrays (CGRAs) can achieve much higher performance and efficiency than general-purpose cores, approaching the performance of a specialized design while retaining programmability. Unfortunately, CGRAs have so far only been effective on applications with *regular* compute patterns. However, many important workloads like graph analytics, sparse linear algebra, and databases, are *irregular applications* with unpredictable access patterns and control flow. Since CGRAs map computation statically to a spatial fabric of functional units, irregular memory accesses and control flow cause frequent stalls and load imbalance.

We present Fifer, an architecture and compilation technique that makes irregular applications efficient on CGRAs. Fifer first *decouples* irregular applications into a feed-forward network of pipeline stages. Each resulting stage is regular and can efficiently use the CGRA fabric. However, irregularity causes stages to have widely varying loads, resulting in high load imbalance if they execute spatially in a conventional CGRA. Fifer solves this by introducing *dynamic temporal pipelining*: it time-multiplexes multiple stages onto the same CGRA, and dynamically schedules stages to avoid load imbalance. Fifer makes time-multiplexing fast and cheap to quickly respond to load imbalance while retaining the efficiency and simplicity of a CGRA design. We show that Fifer improves performance by gmean 2.8× (and up to 5.5×) over a conventional CGRA architecture (and by gmean 17× over an out-of-order multicore) on a variety of challenging irregular applications.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; **Reconfigurable computing**.

KEYWORDS

CGRAs, reconfigurable architectures, pipeline parallelism

ACM Reference Format:
Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480048

1 INTRODUCTION

General-purpose processors are woefully inefficient: they routinely spend less than 1% of their energy executing computation [23], and spend most of their energy and area on instruction interpretation overheads and general but expensive latency-tolerance mechanisms, like out-of-order execution and speculation. With Moore's Law waning, it is crucial to reduce this bloat. While specializing hardware to each application achieves maximum performance and efficiency, it is inflexible. Ideally, we want architectures that approach the efficiency of full specialization, while being programmable and capable of executing a wide range of applications.

Coarse-grain reconfigurable arrays (CGRAs) are a promising approach to achieve this goal. CGRAs implement a sea of spatially distributed functional units that can be configured and connected with switches to create high-throughput datapaths. Prior work has explored and implemented a wide range of CGRA designs, either as standalone accelerators [11, 24, 37, 48, 50] or tightly integrated coprocessors [20, 36, 52, 64].

Unfortunately, CGRAs are restricted to *regular applications*, i.e., those with structured access patterns and control flow, like dense linear algebra. These features are necessary to produce a high-performance pipeline that can be spatially and statically mapped to a CGRA fabric.

By contrast, CGRAs struggle with *irregular applications*, i.e., those with unstructured memory accesses (like indirections) and control flow (like data-dependent branches). These applications arise in many important domains, like graph analytics, sparse linear algebra, sparse deep learning, and databases. CGRAs are ill-equipped to handle these operations: faced with a long-latency operation, like a cache miss, they simply stall; and even if misses are rare, irregular control flow causes load imbalance that leaves most of the fabric idle.

In this paper, we present *Fifer*, an architecture and compilation technique that makes irregular applications efficient on CGRAs. Fifer combines two key techniques:
(1) Extracting regular stages from irregular applications: We show that an application's irregular accesses and control can be *decoupled* from its regular computation, which can then be efficiently processed by the CGRA. This approach divides the computation into a pipeline, i.e., a feed-forward network of stages. These stages are connected with latency-insensitive channels, like FIFO queues, to tolerate unpredictable latencies. Importantly, this approach produces *regular* stages that can be turned into high-throughput datapaths mapped to a CGRA fabric and confines irregularity to happen across pipeline stages.

Despite this transformation, conventional CGRAs are still inefficient: while each stage maps well to a CGRA fabric, irregularity causes wide variations in work across stages. CGRAs, being pure

spatial architectures, cannot accommodate these variations and suffer from load imbalance. This necessitates Fifer's second key technique:
(2) Dynamic temporal pipelining: To avoid load imbalance, Fifer *temporally pipelines* CGRA-based architectures: multiple stages are time-multiplexed into the same CGRA fabric, with a scheduler dynamically choosing which stage to run based on the availability of work. This avoids load imbalance by dedicating more cycles to stages with more work. To be efficient, reconfigurations should be infrequent (occurring every few hundred cycles), and brief (lasting tens of cycles). We introduce fast reconfiguration mechanisms to make this possible.

Prior work has also explored adding time-multiplexing to CGRAs: Triggered Instructions [45] maps multiple operations onto each element of the fabric, and each element selects a ready operation to execute each cycle. This fine-grain time-multiplexing tolerates imbalance, but requires substantial additions to a CGRA to support such frequent switching. By contrast, Fifer reconfigures at coarser granularity: switching between large, regular chunks of operations over several cycles. Because our program transformation coarsens work into chunks that are switched less often, we can use much simpler scheduling hardware to achieve load balance on the CGRA fabric. By analogy with general-purpose cores, Triggered Instructions is the CGRA counterpart to fine-grained multithreading, whereas Fifer is the CGRA counterpart to coarse-grained multithreading.

To implement Fifer, we make three simple modifications to an existing CGRA: *(1)* frequent and rapid reconfiguration, *(2)* buffers acting as queues decoupling spatial and temporal pipelines, and *(3)* logic to further decouple irregular memory accesses. We prototype Fifer in a system with multiple *processing elements* (specialized cores), each with its own CGRA fabric and private cache. We show that Fifer scales well to large systems by combining spatial and temporal pipelining. We implement Fifer's major components in RTL and show that its additions are simple and cheap.

Our evaluation shows Fifer outperforms an OOO multicore by over gmean 17× while using much less area. We also compare Fifer to a CGRA-based architecture that cannot time-multiplex stages. Fifer outperforms this baseline by gmean 2.8×, and by up to 5.5×, across a variety of challenging irregular applications.

In summary, we make the following contributions:

- We identify the challenges of irregular applications on reconfigurable spatial architectures.
- We present a technique to decouple applications across sources of irregularity for effective CGRA mappings.
- We introduce Fifer, a CGRA-based architecture that time-multiplexes multiple configurations onto its processing elements to avoid load imbalance.
- We implement Fifer and evaluate its effectiveness on a wide range of applications, demonstrating its applicability.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce CGRAs as part of a broader taxonomy of spatial and temporal architectures; CGRAs are a purely spatial architecture. Our concrete example illustrates the challenges of accelerating irregular applications in CGRAs, and motivates Fifer's contributions—essentially, adding a cheap temporal component to CGRAs. Finally, we compare Fifer to related prior work.

2.1 Spatial and temporal architectures

We can broadly classify architectures into two classes [6, 20]: *temporal* architectures keep data at a fixed location and, over time, apply different operations to this data by changing the configuration of the processor; by contrast, *spatial* architectures keep the configuration of the processor fixed and process data by streaming it across spatially distributed elements of the processor.

General-purpose cores are *temporal architectures*, because, over time, cores continuously change the operations (instructions) they apply to data kept in the same place (in the register file, or in caches that achieve high reuse by exploiting locality). But continuously switching the activities in the processor pipeline over time (by fetching and decoding instructions, and trying to execute many in parallel) is expensive.

By contrast, coarse-grained reconfigurable architectures (CGRAs) are *spatial architectures*: they are structured as a spatially distributed grid of functional units, each performing a fixed (but configurable) operation. Functional units are composed to perform more complex computations, and data is moved among functional units through a network of simple switches. CGRAs achieve programmability without the need to continuously fetch and decode instructions. But their pure spatial approach has disadvantages: the size of the program is limited by the number of functional units, and to achieve high performance, these units must be configured as a *pipeline*, to allow many functional units to be active simultaneously.

Given their promise, much prior work has proposed CGRA-based architectures, either as standalone accelerators [11, 17, 24, 37, 42, 48] or tightly integrated within the pipeline of a general-purpose processor [20, 36, 52, 64]. Prior work has also designed compilation techniques [15, 33, 43, 65] and applied them to a multitude of application domains, including neural networks [57], data analytics [16, 62], and signal processing [47, 66]. Some GPUs now include specialized units featuring spatial execution, like NVIDIA's Tensor Cores [9], which, though less programmable, resemble CGRAs.

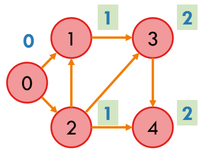
Systems often mix temporal and spatial approaches at different granularities. For example, some multicores, like Raw [58], are temporal within each core, but feature queue-based communication to build spatial pipelines across cores; and DySER [20] integrates a CGRA within each core, using spatial execution selectively. However, to reap the benefits of spatial architectures, we should execute most of the work spatially and use temporal execution infrequently. This is why Fifer is built to use spatial execution at the innermost levels of the application (through CGRAs), and temporal execution (through time-multiplexing) at coarser timescales to amortize reconfiguration overheads.

2.2 Challenges of irregular applications

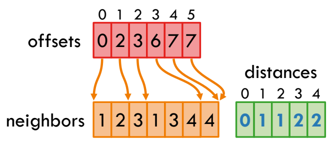
CGRAs are amenable to creating *static spatial pipelines*, in which an application is split into pipeline stages and mapped to functional units across the fabric. To perform a particular computation, operands are passed from one functional unit to the next in this *fixed* pipeline. These are highly effective for regular applications, where the data access patterns and control flow are highly predictable.

```
define bfs(src):
  distances[src] = 0
  cur_fringe = [src]
  cur_dist = 1
  while not cur_fringe.empty():
    for v in cur_fringe:
      start = offsets[v]
      end = offsets[v+1]
      for e in range(start, end):
        ngh = neighbors[e]
        dist = distances[ngh]
        if dist is unset:
          distances[ngh] = cur_dist
          next_fringe.push(ngh)
    cur_fringe = []
    swap(cur_fringe, next_fringe)
    cur_dist += 1
```

(a) Pseudocode for sequential BFS.



(b) An example graph G.



(c) G's CSR representation and, at right, the output (distances) produced by BFS.

Figure 1: An implementation of breadth-first search (BFS).

However, mapping *irregular* applications to spatial architectures is more complex, because their unpredictable latencies and control flow can significantly impact CGRA throughput. The key insight to enabling effective mappings of irregular applications is to recognize that they are rich in otherwise-*regular* computation, but interspersed with *irregular* memory accesses and control flow. Moreover, irregular applications decompose well—Pipette [41] exploited this property in general-purpose cores, but not yet for specialized architectures. This brings us to Fifer's first major contribution: enabling effective mappings to reconfigurable spatial architectures by *partitioning stages across sources of irregularity.*

For a concrete example, we use BFS, a common graph algorithm that finds the distance from a source vertex `src` to all vertices reachable from it. Fig. 1(a) shows example code for serial BFS. The graph in Fig. 1(b) is stored in the widely used compressed sparse row (CSR) format [39, 51, 54], shown in Fig. 1(c) beside the `distances` array, which results from running BFS on this graph. BFS is a challenging irregular workload due to its multiple levels of indirection: it uses elements from `cur_fringe` to access `offsets`, which is then used to access `neighbors`, which in turn is used to access `distances`.

To transform BFS into a pipeline, we isolate each long-latency memory access into its own stage, indicated by differently shaded code in Fig. 1(a). First, the process current fringe stage reads vertices from `cur_fringe`, whose neighbors are identified in the enumerate neighbors stage. For each of these neighbors, the fetch distances stage loads the distance of this neighbor, which is checked against the current distance from the source by the update data stage. By adding queues to communicate data from one stage to the next, we produce the pipeline shown in Fig. 2(a). By placing at most one long-latency operation in each stage, no memory access in any of these stages will keep computation in other stages from proceeding, unless input queues run empty or output queues become full.

This transformed BFS can now be mapped to a spatial architecture, but it will suffer from poor performance due to load imbalance across stages. The baseline architecture that we use in this paper (Sec. 3) has multiple *processing elements* (PEs), each with a separate CGRA fabric, and PEs can communicate through FIFO queues and access memory individually. Fig. 2(b) shows how our transformed BFS can be executed in this architecture by mapping each stage to a PE. But this approach is still a *static spatial pipeline*, and quick variations in load across stages will cause stalls on empty or full queues, resulting in poor utilization.

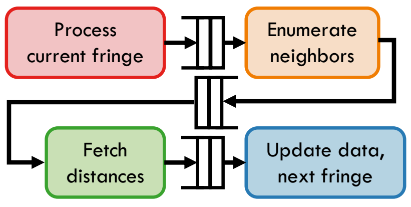
Fifer's other key insight addresses the shortcomings of static spatial pipelines by observing that a stage need not be fixed to a PE for the lifetime of a computation. Instead of placing stages on physically distinct PEs and creating a spatial pipeline, we can *temporally pipeline stages by time-multiplexing them onto the same PE.* Fig. 2(c) shows how Fifer maps BFS using this approach. We call this *dynamic* temporal pipelining, because a scheduler dynamically switches across stages based on the availability of work (e.g., when a stage runs out of work, the ready stage with the most work is switched in). Switches happen every few tens to hundreds of cycles, long enough to amortize reconfiguration overheads, and short enough to keep queue and memory footprint low (as these grow the further stages are decoupled).

2.3 Prior spatial and temporal CGRAs

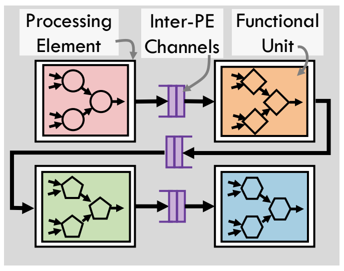
Fifer is not the first proposal to add a temporal component to a spatial architecture, but to the best of our knowledge, it is the first to do so at this granularity.

At one extreme, prior work has proposed time-multiplexing at the cycle level: Triggered Instructions [45] is a spatial architecture with an array of PEs that communicate through latency-insensitive channels. Each PE holds a limited number of instructions (e.g., 16), which become ready depending on runtime conditions (e.g., the availability of a value in a queue). Each PE chooses among one of the ready instructions each cycle. Though this cycle-level approach improves utilization, it comes at a cost: each PE is more complicated than the functional unit in a CGRA, and PEs communicate through queues rather than registers.

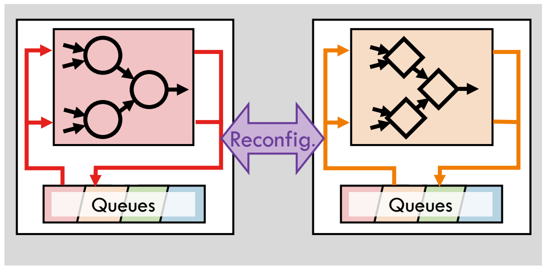
At the other extreme, run-time reconfiguration (RTR), a feature of commercial FPGAs (another kind of spatial architecture), has been used to time-multiplex configurations when a design exceeds



(a) A pipeline-parallel implementation of BFS.



(b) Mapping BFS as a spatial pipeline on current reconfigurable spatial architectures.



(c) **This paper:** map BFS (2 of 4 stages shown) as a dynamic temporal pipeline to a time-multiplexed reconfigurable fabric on a *single* PE.

Figure 2: Mapping breadth-first search (BFS) to spatial architectures.

available resources. However, this happens at coarse timescales—hundreds of microseconds [40]—much longer than the tens of cycles needed to effectively load balance stages of a pipeline or tolerate memory latency. Moreover, prior use of RTR only targeted applications that already map well to spatial architectures, like sorting [28] and streaming [7], or HLS-generated pipelines targeting applications with abundant data parallelism [56].

Fifer lies in the middle of these extremes, reconfiguring every few 10s–100s of cycles. This avoids expensive modifications to CGRAs and amortizes reconfiguration overheads, yet suffices to avoid load imbalance and achieve high utilization.

It is useful to contrast these techniques with general-purpose processors. They are analogous to *multithreading*, where a core switches among multiple threads of execution to improve utilization. Triggered Instructions is the CGRA analog to fine-grained multithreading [4, 25, 29] (FGMT), where the core time-multiplexes threads cycle by cycle; Fifer is the CGRA analog to coarse-grained multithreading [1–3, 26] (CGMT), where the core switches across threads less frequently, to tolerate long-latency events (e.g., on every L2 cache miss); and RTR is the spatial analog to software-only context-switching of threads by the operating system. Just as CGMT requires simpler core changes than FGMT, Fifer requires simpler changes than Triggered Instructions (quantitatively, the difference between these is larger since CGRAs do not already have a temporal component).

2.4 Other related work

Previous implementations of multithreading on CGRAs have not explored accelerating irregular applications by organizing them as pipelines; for example, one approach [50] instead focuses on switching between many disparate applications.

The SGMF architecture [63] and its descendant dMT-CGRA [64] adapt a GPGPU's data-parallel computation to flow through a CGRA. Its multithreading does not refer to the reconfiguring of a CGRA fabric for pipeline-parallel programs but instead to the many identical data-parallel threads of the SIMT execution model.

Pipette [41] implements pipeline-parallel programs by mapping each stage to a thread in a multithreaded OOO core. Fifer is inspired by Pipette and adapts several of its mechanisms to CGRAs, as we explain later. However, the two approaches have major differences. For example, Pipette reuses the physical register file for queue storage, which places an upper bound on decoupling. Furthermore, as previously discussed, modern OOO cores suffer from high frontend overheads and cannot achieve the same compute density as a reconfigurable fabric. Fifer operates at the high-throughput regime of Little's Law: larger queue buffers are needed to support the commensurately higher-throughput reconfigurable fabric.

Sec. 9 discusses additional prior work, including application-specific accelerators and decoupled access-execute (DAE) architectures.

Unlike prior work, Fifer combines spatial pipelining and temporal pipelining at *coarse-grain timescales* (10s–100s of cycles) to amortize the costs of reconfiguration yet effectively tolerate latencies of the memory hierarchy, with *coarse-grain computation* (machine word width, not bit granularity), to address the limited throughput and flexibility of prior architectures.

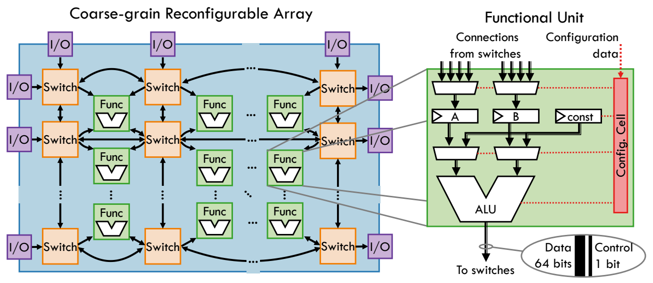


Figure 3: Coarse-grain reconfigurable array (CGRA, left) and functional unit design (right).

We achieve better utilization by *(1)* time-multiplexing the fabric at the individual PE level for improved load balance, and *(2)* enabling fast communication between stages on the same PE. Our novel programming model—structuring applications as pipeline-parallel stages of computation that can be time-multiplexed on the same processing element—overcomes the limitations of prior architectures and enables Fifer's performance benefits.

3 BASELINE CGRA ARCHITECTURE

Because there are many CGRA designs, we first introduce the baseline CGRA architecture to make the discussion concrete. Fifer then builds on this baseline.

CGRA fabric: Fig. 3 shows the major components of a coarse-grain reconfigurable array (CGRA): at a high level, it is a grid of *functional units* connected together with switches. Each functional unit, shown in detail in Fig. 3, contains an integer ALU similar to one in a general-purpose processor, capable of elementary operations (arithmetic, shifts, bitwise operations) at machine word width (e.g., 64 bits). Each PE also incorporates a few double-precision fused multiply-add (FMA) units to support floating-point workloads.

Configuration cells (registers) at each functional unit specify the operation of the ALU; additional configuration cells specify the connectivity of switches passing operands between functional units. Because conventional CGRAs are reconfigured rarely, their configuration cells use slow but simple write mechanisms, like register scan chains. (Fifer contributes a fast reconfiguration mechanism in Sec. 5.1.)

Inputs and outputs enter and leave the reconfigurable array through ports at the edges of the grid. The reconfigurable array is internally pipelined; that is, functional units are separated by registers as shown in Fig. 3, and the longest input-output path through functional units sets the latency of a given configuration. Registers also allow the CGRA to retain program state, e.g., to track loop iteration counts or accumulate values over loop iterations.

Multi-PE CGRA architecture: A single CGRA fabric is sometimes used as a functional unit or coprocessor [17, 20, 52] to accelerate small kernels. However, our goal is to handle full algorithms autonomously, without having general-purpose cores as intermediaries. To this end, our baseline architecture, shown in Fig. 4, consists of multiple *processing elements* (PEs), each of which integrates a CGRA fabric, a private L1 cache, and mechanisms for queue-based communication. All PEs share a (highly banked) last-level cache.

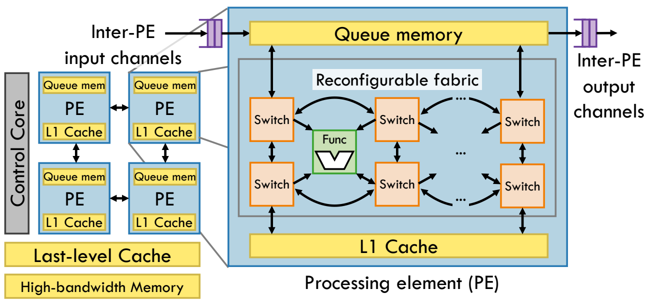


Figure 4: Baseline spatial architecture and design of a processing element (PE) with CGRA fabric.

This approach is preferable to having a single, very large CGRA fabric for two reasons. First, it enables having multiple independent private caches, so the system can achieve high memory throughput and exploit locality. Second, it provides decoupled communication between PEs. Because functional units inside each CGRA fabric are tightly coupled through *rigid* pipelines, a single stall would quickly propagate through the whole fabric. By contrast, PEs communicate with each other through FIFO queues, so when one incurs a stall (e.g., due to a cache miss), other PEs will not necessarily stall.

Our baseline implements inter-PE queues through a flexible interface: the switches at the edges of the fabric can dequeue from input queues and enqueue to output queues. Queues are stored in a small *queue memory* in each PE (a 16 KB SRAM in our implementation). This queue memory can be statically divided among multiple queues, each of which is managed as a circular buffer.

Mapping applications: To use this multi-PE design, applications are divided in *stages*, each of which is mapped to a PE. Then, these stages communicate through queues.

To use this baseline system well, it is crucial that all stages proceed at roughly the same rate: if one stage produces more inputs at a higher rate than its consumer, it will be bottlenecked by its consumer and frequently stall on a full output queue. Conversely, a too-fast consumer will spend many cycles waiting for input. Note that queues provide decoupling, but only against *temporary* mismatches in throughput, e.g., due to a cache miss. Long-running throughput differences will, over time, leave queues full or empty, and make the whole program run at the pace of the rate-limiting stage. Because queues do not provide unbounded decoupling, we consider this design a *static* spatial pipeline, even though it is decoupled.

Prior work has proposed different techniques to use this kind of static pipeline well, such as replicating slow stages [19]. But this requires having known and stable rates, which is not the case with irregular applications.

4 EXTRACTING REGULAR STAGES FROM IRREGULAR APPLICATIONS

We present a new technique to map irregular applications to CGRA fabrics—this is a prerequisite for Fifer, but also for our baseline architecture. The key insight, as we explained in Sec. 2.2, is to first *partition the application into stages across sources of irregularity.* This produces *regular stages* and confines the irregularity to happen

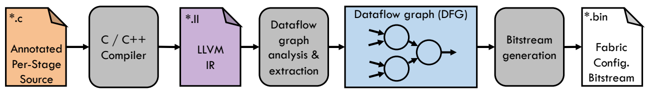
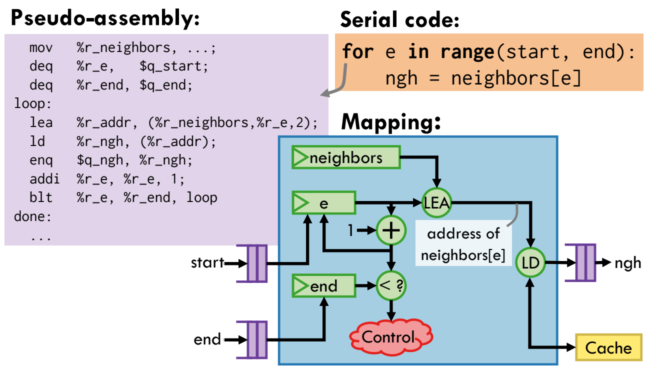


Figure 5: Process for transforming a pipeline stage's annotated source code into a Fifer PE fabric configuration.


Pseudo-assembly:

```
  mov    %r_neighbors, ...;
  deq    %r_e,   $q_start;
  deq    %r_end, $q_end;
loop:
  lea    %r_addr, (%r_neighbors,%r_e,2);
  ld     %r_ngh, (%r_addr);
  enq    $q_ngh, %r_ngh;
  addi   %r_e, %r_e, 1;
  blt    %r_e, %r_end, loop
done:
  ...
```

Serial code:

```
for e in range(start, end):
    ngh = neighbors[e]
```

Figure 6: Example mapping of BFS's enumerate neighbors stage to a CGRA.

across stages. Then, these regular stages are efficiently mapped to a CGRA. In our implementation, the first step is manual, while the second is automated.

Partitioning: A program may be split at arbitrary locations into arbitrarily many stages, but judiciously decoupling programs is essential for good performance. Our overarching objective is to decouple irregular parts of the computation, such as unpredictable memory accesses and control flow, from more regular parts. Therefore, we split a program at every long-latency load, so that loads issued by a given stage are consumed by a different stage. For example, in BFS (Sec. 2.2), each loop nest level contains such a load, so each stage corresponds to a level. Finally, stages that are too large to fit on one PE can always be divided into multiple smaller stages.

Though this partitioning process is manual, it is systematic. It is similar to Pipette's [41] approach to partition irregular applications across threads, but whereas Pipette can rely on prefetchers and OOO execution to avoid splitting across predictable accesses, we have no such prediction mechanisms. We believe such approach is automatable, e.g., using techniques like DSWP [49], or relying on domain-specific languages that perform deep program transformations [27, 69]. We leave this to future work.

Mapping: Fig. 5 shows the process of transforming partitioned serial code into configurations for a CGRA. We generate LLVM intermediate representation (IR) for each stage, which represents low-level operations on data and their dependences. An automated tool examines the LLVM IR and produces a dataflow graph (DFG) using the actual operations that can be performed by a PE's functional units. The DFG is modified to receive its inputs and send its outputs via queues. A final bitstream generation step transforms the DFG into a bitstream that, when configured into a CGRA, carries out the computation represented by the DFG.

To concretely demonstrate this process, consider the enumerate neighbors stage from BFS, shown in Fig. 6. This stage dequeues the `start` and `end` positions of the edge list of a vertex, and produces neighbor vertex ids. The operations required to carry out

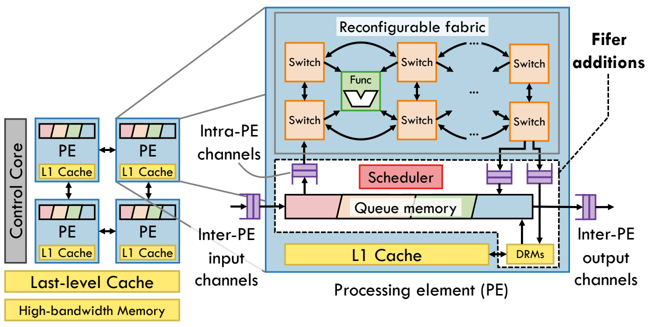


Figure 7: Fifer's modifications to a reconfigurable spatial architecture and processing elements, surrounded by a dashed line. (Queues, colored purple, are only illustrative and virtualized in the queue memory.)

the serial code (in orange) for this stage are represented with the pseudo-assembly code (in purple) resembling the lowered LLVM IR produced by the compiler. Each pseudo-assembly instruction corresponds to an operation assigned to a functional unit in the PE.

When mapped to a DFG, this stage computes addresses (LEA) in the `neighbors` array, performs the dereference (LD), and passes the neighbor (ngh) to the next stage. Some additional logic (+ and < ?) determines whether we have finished this edge list.

Values are processed in the order they arrive; once dequeued, they flow through one functional unit per cycle. Some control logic (the red cloud) triggers dequeues of new `start` and `end` values. In addition to driving enqueues and dequeues, per-PE control logic also orchestrates reconfigurations (Sec. 5.1) and stalls the pipeline for cache misses on coupled loads (Sec. 5.4).

Inter-stage control flow: In irregular applications, stages often need to communicate control flow decisions to other stages; for example, in BFS, all stages need to know when the current distance from the source vertex has changed. Since stages communicate through queues, it is natural to pass this control information through the same queues. We extend queues to carry control or data values. Since control values are infrequent, we compile stages to handle either multiple input data values or one control value per cycle. Sec. 5.5 gives implementation details.

5 FIFER ARCHITECTURE

Static spatial pipelines have several limitations reducing their effectiveness on irregular applications. Fifer overcomes these limitations by augmenting spatial pipelines with *temporal pipelines*. We organize our discussion of Fifer to describe:

(1) how multiple stages are time-multiplexed onto the same PE through the reconfiguration process,
(2) how to extend queues to communicate between stages on the same PE, and
(3) how to further decouple long-latency memory accesses.

Fig. 7 shows our modifications to the baseline system. We initially focus on a single-PE Fifer system that implements pure temporal pipelining. In Sec. 5.6 we discuss how multi-PE Fifer leverages both temporal and spatial pipelines.

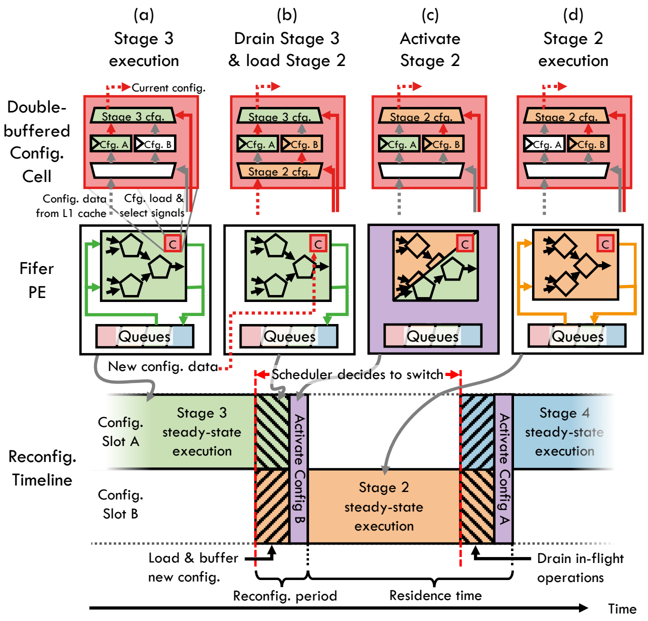

(a) Stage 3 execution
(b) Drain Stage 3 & load Stage 2
(c) Activate Stage 2
(d) Stage 2 execution
Current config.
Double-buffered Config. Cell
Stage 3 cfg.
Stage 2 cfg.
Cfg. A
Cfg. B
Config. data from L1 cache
Cfg. load & select signals
Fifer PE
C
Queues
New config. data
Scheduler decides to switch
Reconfig. Timeline
Config. Slot A
Config. Slot B
Stage 3 steady-state execution
Activate Config B
Stage 2 steady-state execution
Activate Config A
Stage 4 steady-state execution
Load & buffer new config.
Drain in-flight operations
Reconfig. period
Residence time
Time

Figure 8: Reconfiguration process in a Fifer PE, depicting role of double-buffered configuration cells.

5.1 Rapid reconfiguration

Each Fifer PE may select from not just one, but *many* possible configurations, so a PE may represent any part of a temporal pipeline throughout a program's execution.

Fifer is designed to transparently switch stages so that any ready stage begins executing as soon as possible. When a stage is scheduled onto a PE, functional units are configured with the operations needed by the stage. Reconfiguration also establishes connections between these functional units, as well as any input/output queues, registers, and memory connections. Any state required by the application, such as fixed constants, are loaded into the appropriate registers.

Unlike configurations for the baseline system, Fifer's configurations are stored in cacheable memory and loaded from the L1 cache. Configuration cells are chained, so that configurations can be loaded over multiple cycles. For example, our L1 supports a bandwidth of 64 bytes/cycle, and our 16×5 fabric requires about 360 bytes of configuration, so configuration cells are divided in 6 groups (with 5 groups consisting of a row of ALUs and switches, and one group being the last row of switches). Each cycle, the L1 serves 64 bytes of configuration data, which are propagated though the chained configuration cells. Thus, over 6 cycles (plus the L1 latency), the new configuration is loaded in place.

Loading the new configuration, as described by the previous paragraph, forms step (1) of a three-step reconfiguration process. Step (2) drains the in-flight operations from the current configuration. Step (3) activates the new configuration. Fifer introduces *double-buffered configuration cells* so that steps (1) and (2) take place in parallel. Consider the process in which Stage 3 is currently running on a Fifer PE and now needs to switch to run Stage 2. Fig. 8 shows the behavior of a configuration cell (top row) and how it changes the currently executing stage of a Fifer PE (middle row).

As soon as the scheduler begins the reconfiguration process, Stage 3 stops accepting new inputs and begins draining in-flight operations, as Fig. 8(b) shows. In parallel with draining in-flight operations, the PE begins loading the new configuration. This parallel loading is enabled by Fifer's double-buffered cells, which offers two configuration slots (Cfg. A and Cfg. B): one containing the current configuration and one to receive the new configuration. The PE loads new configuration data into the unused configuration slot (e.g., Cfg. B, since Cfg. A is currently used for Stage 3). Double-buffering allows us to overlap execution of the current configuration with the loading of the next configuration, which is essential to reducing the cost of reconfiguration. In practice, most configurations have over 6 pipeline stages, so draining them takes longer than loading the new configuration and is the dominant cost of reconfigurations for most applications.

Once remaining in-flight operations finish and the new configuration is loaded, the fabric *activates* the new configuration. Within the double-buffered configuration cells in Fig. 8(c), a multiplexer switches from reading Cfg. A to Cfg. B, so now Stage 2 becomes active. We account for this process with an *activation time*, a dead time of two cycles. At last, in Fig. 8(d), Stage 2 commences execution.

The old configuration may have written to state elements, like registers, that need be preserved across reconfigurations. As the new configuration loads, the contents of these state elements are drained out along with the old configuration to the L1 cache.

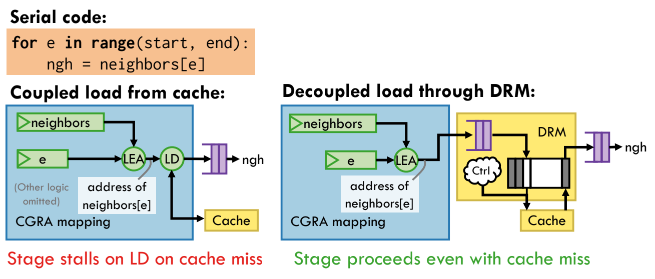
We define the *reconfiguration period* to be the time spent performing all of these operations: draining in-flight operations, loading the new configuration, and activating it. The *residence time* for a given stage is the time between activating that stage and the activation of the next stage (and thus includes the reconfiguration period). Fifer's effectiveness relies on keeping the reconfiguration period small—no more than a few dozen cycles—and maximizing residence times to many hundreds of cycles. We evaluate the effect of reconfiguration period on performance in Sec. 8.3.

5.2 Scheduling reconfigurations

Fifer extends each PE with a simple *scheduler* that dynamically switches among stages. Which stage is scheduled onto a particular PE depends on which input queues have values available and which output queues have space.

To keep utilization high and amortize the overall cost of reconfigurations, the scheduler follows a simple policy. First, it keeps a PE configured to the current stage until it is blocked by a full output queue or an empty input queue. Second, when it must select a new stage, the scheduler examines the occupancies of the queues used by the other stages. Of the unblocked stages (i.e., those with non-empty input queues and non-full output queues), the scheduler selects the stage with the greatest amount of work available in its input queues. By selecting stages with more work, the scheduler reduces the number of reconfigurations.

We also tried other scheduler policies, such as a round-robin scheduler or finer-grain multithreading, but found that these did not work as well. This makes sense: the application work done is nearly constant regardless of the scheduling policy, so processing the stage with the most work reduces the amount of reconfigurations; alternative policies increase reconfiguration frequency.



```
for e in range(start, end):
    ngh = neighbors[e]
```

Figure 9: Using a DRM to further decouple stages by separating address generation and fetching from cache.

5.3 Communicating in temporal pipelines

Because Fifer allows a producer stage to communicate with a consumer stage that may be located at the same PE, we introduce *intra-PE* queues. We augment the queue buffer described in Sec. 3 with more head/tail pointers to store these additional queues. These intra-PE queues offer high-bandwidth communication between stages located on the same PE, so it is advantageous to decouple applications such that stages communicating frequently reside on the same PE. Of course, as before, a producer stage on one Fifer PE can enqueue data destined for a consumer stage at a different PE.

For the purposes of evaluation, the baseline spatial architecture and Fifer have the same amount of queue buffer per PE. This means that Fifer, which can fit many more pipelines than the baseline, could have less effective space per queue. However, as we will see, modest decoupling suffices to achieve high utilization.

5.4 Further decoupling memory accesses

Memory accesses have widely varying needs: some may be irregular and cause stalls so they need to be decoupled; others may be known to rarely cause cache misses and so they do not merit further decoupling. Thus, Fifer PEs offer both *decoupled* and *coupled* load interfaces. The conventional, coupled load interface is a simple connection to the cache and stalls the PE on cache misses. Simple memory access patterns, like streaming linearly through memory, do not need to be decoupled, and would be suitable for this interface.

However, some accesses are known to miss frequently, causing lengthy stalls. Fifer allows these accesses to be further decoupled from stages with decoupled reference machines (DRMs), which are small finite state machines that interact with the PE through Fifer's decoupled interface and are shown in Fig. 9 (right). Instead of sending addresses directly to the cache, the PE instead enqueues the address into the input queue of a DRM. Now, the DRM performs the memory access on the PE's behalf and places the result into an output queue to be consumed by the next stage. DRMs are similar to reference accelerators from Pipette [41].

DRMs can be configured in two ways: in *dereference mode*, the DRM interprets input operands as addresses whose values in memory will be enqueued in the output queue. In *scanning mode*, the DRM interprets a pair of input operands as a range of addresses to sequentially fetch and enqueue. Other modes, like strided accesses to traverse arrays of structs, could be easily added to DRMs to reduce use of the reconfigurable fabric for address generation. We did not find the need for other modes in our benchmarks.

DRMs complement the high-throughput CGRA fabric by providing a mechanism for performing decoupled memory accesses. Unlike memory accesses initiated by the fabric, memory accesses issued by DRMs may complete out-of-order. Furthermore, unlike CGRA fabric configurations, DRMs are configured once, at initialization, so they may continue performing accesses regardless of which stage is currently scheduled on the PE.

Fig. 9 depicts the use of both the coupled and decoupled memory interfaces: in BFS's enumerate neighbors stage, getting the neighbor id requires dereferencing the `neighbors` array, but including a conventional load (left, the `LD` operation) in this stage leaves this stage vulnerable to stalls caused by loads from `neighbors` that will inevitably miss in cache. It would be better to send these addresses to a DRM instead (right), decoupling address generation of the enumerate neighbors stage from latencies incurred by the memory accesses.

In this example, a DRM is configured to the *dereference* mode at initialization. Now, the enumerate neighbors stage is solely responsible for computing the *address* from where to load `ngh`—that is, the memory address where `neighbors[e]` is stored. The stage generates this address through the `LEA` operation, which uses the index `e` and base address `neighbors`. This shortened stage enqueues this address to its output queue, which the DRM receives as its input. The DRM now performs the memory access, which will obtain the neighbor id `ngh` as a result. Once this value is available (and all other accesses ahead of this one have completed), `ngh` is placed in the output queue to be sent to the consumer stage. Even if the DRM's accesses to the `neighbors` array cause cache misses, the enumerate neighbors stage can continue enqueueing values to the DRM, stalling only when the DRM's input queue becomes full.

5.5 Control flow

Sometimes, producers may need to communicate control flow decisions to downstream consumer stages. In such cases, producers may enqueue *control values*, which are values that cannot be mistaken for data. These values can cause PEs to change some local state (such as updating the current BFS fringe), wait to synchronize with other PEs, or switch to another configuration. Using control values cheaply implements point-to-point synchronization where global synchronization is unnecessary or cumbersome.

We implement control values by adding an additional *control bit*, shown in Fig. 3, to communication channels and buffers to indicate whether the value is a control value. We similarly augment the functional units to allow special handling of control values: for example, it may be used to select between operand A or B, or used to predicate operations, e.g., whether to enqueue a particular value. Control bits travel alongside data; this way, their order can be used to delineate boundaries between sets of data or delineate iterations. The data traveling with the control bit can also be used to distinguish between different conditions (e.g. the end of an iteration versus the end of the program).

5.6 Multi-PE Fifer to exploit data parallelism

Up to this point, we have focused on temporal pipelines running on a single Fifer PE. We now explore how spatial pipelines can be used to leverage data parallelism. While our programming model focuses on making pipeline parallelism easy to exploit, data parallelism and pipeline parallelism are complementary; by exploiting them together, Fifer offers advantages that are not available when exploiting data parallelism alone. We now present two techniques that exploit data parallelism *within* a PE and *across* PEs.

SIMD-style parallelism within a PE: Because we can split applications into as many stages as needed, each stage can be arbitrarily small. When a stage implements data-parallel computation, the *datapath* obtained from its DFG can be replicated to use the PE's fabric as much as possible by filling unused functional units and switches. For example, a 16×5 grid of functional units can be configured as four copies of a datapath that fit on a smaller 4×5 grid, yielding a potential 4× throughput improvement. Our applications provide abundant opportunities to take advantage of SIMD-style parallelism: for example, the many edge list accesses performed by graph applications can be launched in parallel.

These datapaths run *in lockstep*: if multiple input elements are available at once, they can be dequeued as a group and processed simultaneously, up to the number of replicated datapaths. Control values are always handled serially: in a given cycle, a PE can dequeue multiple data values but will always dequeue a single control value.

This process is analogous to how vector processors, SIMD instructions, and GPUs exploit data parallelism: running multiple copies of the same operation in lockstep. When fewer than the maximum supported number of elements are available at the input, some datapaths are left unused, just like masked-off lanes in vector processing. However, Fifer's decoupled execution model makes exploiting SIMD-style data parallelism more efficient. For example, to get good GPU lane utilization in graph algorithms, edges from multiple vertices need to be processed in the same warp, leading to complex marshaling of vertex and graph metadata. Instead, Fifer's queue-based approach allows decoupling the processing of vertices and edges across stages, so they are grouped independently and easily fill the parallel datapaths.

Replicated temporal pipelines across PEs: Temporal pipelines can coexist with spatial pipelines; we spatially partition the data across multiple PEs, each running its own temporal pipeline. Fig. 10 replicates the BFS pipeline from Fig. 2(a) across four PEs, each pipeline processing a fraction of fringe and updating separate parts of the graph.

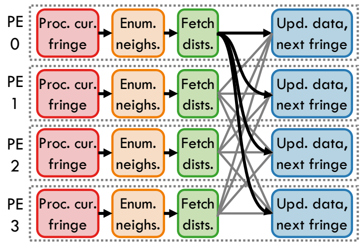


Figure 10: Replicated 4-PE BFS with Fifer.

Pipeline parallelism and queue-based communication enable a crucial optimization: instead of synchronizing through shared memory, each pipeline—that is, a processing element—sends neighbors "owned" by a different pipeline on another processing element. In BFS, this sharding is represented by the cross-PE communication between fetch distances, the third stage, and update data and next fringe, the final stage. By avoiding the need for shared-memory synchronization, applications scale better than by exploiting data parallelism alone.

Item	Area
Reconfigurable fabric, 16 × 5 func. units	0.91 mm^2
and 4× double-precision FMA units	0.15 mm^2
16 KB queue SRAM	0.054 mm^2
4× decoupled reference machines (DRMs)	0.0029 mm^2
32 KB data cache	0.22 mm^2
Total area (per PE)	1.34 mm^2

Table 1: Implementation costs for major components of a Fifer PE.

Inter-PE queues use credit-based flow control [12] to implement backpressure and handle multiple producers. Credits are associated to free queue space. Each destination queue divides credits evenly across producers, and a producer stalls when it runs out of credits.

Importantly, each PE in this replicated pipeline is still a dynamic temporal pipeline, so it *independently* reconfigures itself in response to varying load. Thus, different PEs can work on different stages. This scheme offers an additional dimension of load balance: not only is work distributed across PEs, but each PE also tolerates a different distribution of work among stages. This allows us to employ simpler partitioning schemes—for example, by examining bits of the neighbor id—rather than resorting to more complex techniques like work stealing.

6 FIFER IMPLEMENTATION

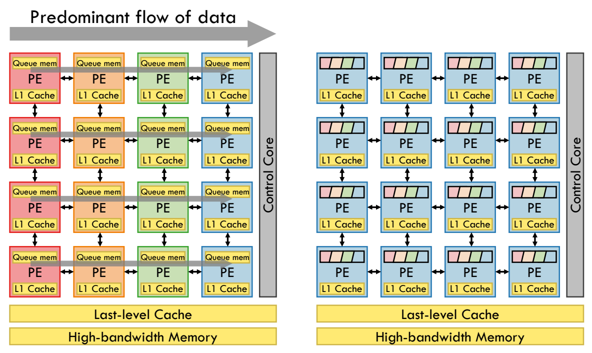
We implement the Fifer architecture by writing and synthesizing RTL for its major components.

The CGRA in each PE is a 16 × 5 grid of functional units surrounded by switches, a scaled-up version of the DySER fabric [20]. We use the CGRA-ME [8] framework to generate RTL for this fabric. CGRA-ME's output Verilog only uses a simple register scan chain to implement its configuration; we replace this with double-buffered configuration cells (Sec. 5.1) to allow loading a new configuration while the current configuration's remaining in-flight operations complete. To make loading configurations fast, Fifer loads configuration data served at the L1 width, not through a serial scan chain.

Our virtualized queues are stored in a 16 KB buffer, and each PE contains a 32 KB data cache. To support the floating-point operations used in some of our benchmarks, we synthesize several double-precision fused multiply-add (FMA) units and distribute them evenly across the fabric. The decoupled reference machines (DRMs), which launch and track decoupled memory accesses, add little additional area cost.

We synthesized these components with Yosys [67] and the 45 nm FreePDK45 library [38], closing timing at 2 GHz, and summarize the area used in Table 1. The memory arrays in caches and queue storage were estimated with CACTI [5]. Overall, each PE is 4.6% of the area of a core in the same technology node (45 nm Nehalem [59]), and each PE has higher arithmetic intensity. To account for this difference, in the evaluation, CGRA-based systems use 4 PEs for each OOO core of the baseline, which is conservative area-wise. (Our evaluation in Sec. 8 uses Skylake cores with larger structures, which makes our estimate even more conservative, even considering scaling to Skylake's 14 nm node.)

We transform our evaluated applications in two steps. We first derive the pipeline-parallel version by manually dividing code into stages using the systematic approach described in Sec. 2.2 and Sec. 4.



(a) Baseline reconfigurable architecture with static spatial pipelines. (b) Fifer architecture with dynamic temporal pipelines.

Figure 11: Baseline and Fifer system implemenations.

Then, we use per-stage data parallelism by replicating the dataflow graph until we fill the PE fabric.

Whenever a Fifer PE changes configuration, it takes a minimum of 12 cycles (loading the new configuration from the L1 cache is 10 cycles, plus 2 cycles for the activation time), but as previously discussed in Sec. 5.1, draining in-flight operations may increase this time.

7 EXPERIMENTAL METHODOLOGY

7.1 Evaluated systems

We implement and evaluate Fifer using cycle-level simulation. For the serial and OOO cores, we use a detailed event-driven cycle-level simulator based on Pin [35], using timing models from Pipette [41]. Core parameters are comparable to an Intel Skylake core with 6-way OOO issue. We also create a cycle-level simulator to evaluate our CGRA-based systems; it simulates executing stages using mapping information produced by CGRA-ME [8].

We model core and uncore energy at 22 nm for the OOO systems with McPAT [31] and use prior work to estimate HBM energy consumption [44]. Energy consumption for the reconfigurable fabric is based on Synopsys Design Compiler post-synthesis power estimates and scaled from 45 nm to 22 nm.

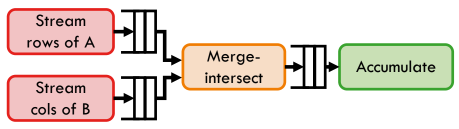
Fig. 11(b) shows our modeled spatial reconfigurable architectures, including 16 PEs with buffers serving as queue storage. A control core is responsible for initialization and teardown of a Fifer program as well as interactions requiring a general-purpose CPU (such as calls to the OS).

Comparison systems: Our baseline is the 16-PE system depicted in Fig. 11(a). Each stage of an application is mapped to a single PE, and this mapping remains fixed throughout the run. As a result, the predominant flow of data through the pipeline is also fairly fixed; for example, in BFS the predominant flow of data would be from left to right, as indicated by the gray arrow. While the baseline and Fifer systems have the same amount of space allocated for queues on each PE, the static system notably lacks the scheduler. The baseline system also retains DRMs, to focus our analysis on the effects of time-multiplexing on load balance.

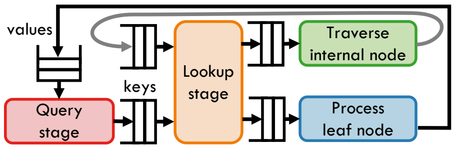
We also compare Fifer to serial and 4-core parallel implementations running on out-of-order cores with aggressively sized core structures, provisioned similarly to Intel's Skylake [13]. Table 2 summarizes our evaluated systems' parameters.

PEs	16 PEs, 2 GHz, 16 × 5 func. unit mesh, 32 KB L1 cache (8-way set-associative, 4-cycle latency)
Fifer	Up to 16 queues per PE, virtualized on a 16 KB buffer
Cores	1 or 4 cores, 2 GHz, x86-64 ISA, Skylake-like: 6-wide out-of-order issue, 32 KB L1 cache (8-way set-associative, 4-cycle latency), 256 KB L2 cache (8-way set-associative, 12-cycle latency)
LLC	2 MB/core *or* 512 KB/PE, 16-way set-associative, 40-cycle latency
Main mem	120-cycle latency, 256 GB/s high-bandwidth memory

Table 2: Configuration parameters of the evaluated system.



(a) Pipeline diagram for SpMM.



(b) Pipeline diagram for Silo.

Figure 12: Application pipelines for Silo and SpMM.

7.2 Benchmarks

We evaluate Fifer on six applications from graph analytics, sparse linear algebra, and databases. For each application, we start from a state-of-the-art serial implementation and derive a pipeline-parallel version. These applications were adapted from Pipette [41].

Breadth-first search (BFS), first described in Sec. 2, determines the distance of graph vertices from a source vertex. Our baseline OOO serial and multicore versions are based on work-efficient PBFS [30]. **Connected components (CC)** launches multiple breadth-first searches to discover connectivity of graph vertices. **PageRank-Delta (PRD)** [51] is an extension of PageRank that only visits vertices when the change in their PageRank values exceeds a threshold. **Radii estimation (Radii)** estimates the diameter of a graph by launching searches from a random subset of graph vertices. The Ligra [51] framework supplies the baseline implementations of these benchmarks, which also serve as the basis for our optimized Fifer versions.

Sparse matrix-matrix multiplication (SpMM) multiplies two compressed matrices: one in the CSR and the other in the Compressed Sparse Column (CSC) format. It is essential for sparse linear algebra and manifests in many application domains, including dynamic simulation and numerical solvers. We evaluate an *inner product*, or *output-stationary*, SpMM implementation, which computes the product C of matrices A and B one output element at a time by performing the inner product of a row from A and a column from B. Fig. 12(a) depicts a pipeline-parallel implementation of SpMM.

When performing the inner product of two vectors, only locations in both vectors where both elements are non-zero will affect the final result. In a compressed representation, this means that only non-zeros occurring at coordinates present in *both* vectors will affect the inner product. The merge-intersect stage iterates through coordinate lists of the row and column vectors in tandem, outputting coordinates that exist in both of them. The actual non-zero values located at these coordinates are then multiplied and accumulated into the final product by the accumulate stage.

Domain	Graph	Vertices	Edges	Avg. deg.
Human collaboration (Hu)	`coAuthorsDBLP-symmetric`	299K	1.9M	6.4
Dynamic simulation (Dy)	`hugetrace-00000`	4.6M	14M	3.0
Circuit simulation (Ci)	`Freescale1`	3.4M	19M	5.6
Internet graph (In)	`as-Skitter`	1.7M	22M	12.9
Road network (Rd)	`USA-road-d-USA`	24M	58M	2.4

Table 3: Input graphs, sorted by the number of edges.

Domain	Matrix	Size ($n \times n$)	Avg. nnz/row
File sharing (FS)	`p2p-Gnutella31`	62,586	2.4
Graph as matrix (Gr)	`amazon0312`	400,727	8.0
Gel electrophoresis (GE)	`cage12`	130,228	15.6
Electromagnetics (EM)	`2cubes_sphere`	101,492	16.2
Fluid dynamics (FD)	`rma10`	46,835	49.7
Structural (St)	`pwtk`	217,918	52.9

Table 4: Input matrices, sorted by average non-zeros per row.

Merge-intersection is a challenging property of SpMM because of its difficult-to-predict data-dependent traversal of the row and column vectors. Similar intersections also manifest in other applications, like database joins.

Silo [61], an in-memory database, performs lookups to B+tree indexes. Silo traverses the B+tree by examining the current node. If it is a leaf node, it checks for the presence of a value and returns it (lookup stage). Otherwise, it is an internal node, and it returns it to the queue for another dereference (traverse internal node). This pipeline, shown in Fig. 12(b), thus includes a cycle, shown by the gray arrow. Cycles in Fifer's stages are allowed so long as the amount of work performed is bounded: each internal node enqueues *at most* one additional node on the cyclical path. The Fifer implementation of Silo pipelines multiple lookups to overlap multiple memory accesses at once. Organizing Silo this way enlarges its memory footprint; we scale the queue memory down to 4 KB in our experiments to better fit the LLC.

As shown for BFS in Fig. 10, each pipeline is replicated so that each PE has a self-contained four-stage pipeline. The pipelines for CC, PRD, and Radii also use this structure. In SpMM, each PE is responsible for a contiguous set of rows of the output matrix; and in Silo, database operations are striped evenly across the PEs.

Sampling: To keep simulation times reasonable in PRD and Radii, we sample a subset of iterations. In SpMM, we multiply a subset of rows and columns. Even with this sampling, simulations are long (e.g., ~2 billion cycles on the largest inputs), so no warmup is needed. On other benchmarks, we run the entire input.

Input sets: BFS, CC, PRD, and Radii use five large, real-world graphs that include road networks, Web connectivity graphs, and academic collaboration graphs, listed in Table 3. SpMM uses six diverse sparse matrices, listed in Table 4. Silo uses the YCSB-C workload [10] on a 52 GB dataset.

8 EVALUATION

We compare Fifer to state-of-the-art data-parallel baseline implementations running on a generously provisioned out-of-order core. We compare our technique to an OOO baseline, then show that our approach achieves better utilization, and thus better performance, than a pipeline of static, single-stage PEs.

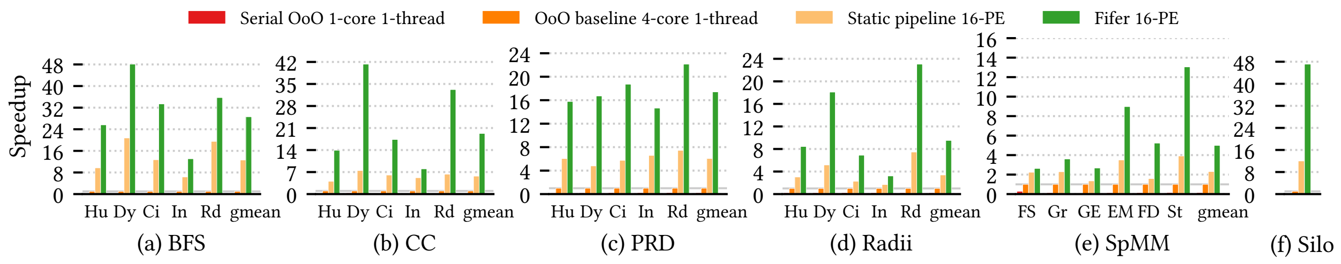


Figure 13: Per-input performance of all evaluated applications.

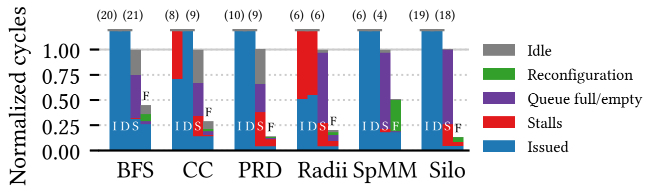


Figure 14: Breakdown of cycles spent executing each application, normalized to the static pipeline and averaged across inputs. (I: OOO serial, D: OOO multicore, S: Static pipeline, F: Fifer)

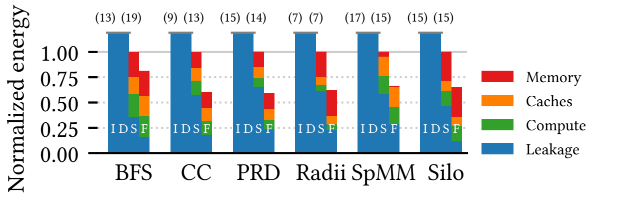


Figure 15: Breakdown of energy, normalized to the static pipeline and averaged across inputs. (I: OOO serial, D: OOO multicore, S: Static pipeline, F: Fifer)

8.1 Fifer outperforms the OOO baseline

The OOO core baseline, despite its significant area overheads, fails to perform well because it cannot effectively handle these applications' unpredictable memory accesses and control flow. Because it is a temporal architecture that sequences instructions one after another, it also suffers from low arithmetic intensity compared to the CGRA fabric. As a result, the static pipeline and Fifer are 25× and 72× faster than serial, respectively.

8.2 Fifer outperforms static pipelines

Fig. 13 shows the performance of our evaluated systems normalized to the performance of the *OOO multicore* (not serial). Fifer outperforms the static pipeline by gmean 2.8× and by up to 5.5× (CC with the Rd input). This speedup comes from Fifer's ability to change contexts in response to available work at each PE. For example, in BFS, speedups are best on graphs with high outdegree, where Fifer's many dynamic temporal pipelines working in parallel achieve better throughput than the baseline's few static pipelines.

To better understand how each system spends its execution time, Fig. 14 shows the breakdown of cycles spent executing each benchmark. We report the proportion of time spent by a core using the CPI stack methodology [14]. We extend this methodology to our PEs as well. Each group of bars reports breakdowns of each variant across benchmarks (averaged across inputs), relative to the static pipeline baseline. Each bar within a group reports cycles for one system, broken down in cycles spent *(1)* performing useful computation, and waiting on *(2)* backend or CGRA stalls (due to non-decoupled loads), *(3)* full or empty queues, *(4)* reconfigurations (for Fifer), or *(5)* idle stalls (when a PE is completely inactive while waiting for other PEs, e.g., a barrier).

As expected, a significant source of slowdowns in the serial and data-parallel systems is waiting on the backend (in red), which includes waiting for memory accesses and OOO core structures becoming full. In the static pipeline and Fifer systems, the breakdown also includes stalls resulting from full or empty queues (in purple). Finally, for Fifer, time spent reconfiguring is shown in green. In spite of this, Fifer performs better because it overlaps useful work, like completing memory accesses, as these reconfigurations occur.

These cycle breakdowns help us understand why Fifer performs consistently better across applications: as expected, the static pipeline spends a significant fraction of time stalled on full or empty queues (purple bars). Reconfiguration stalls are significant in SpMM because it is a *control-intensive* application: the merge-intersect stage intersects values at very high throughput, and when it reaches the end of an input row or column, it directs the producer to stop fetching unneeded data. In relatively sparse matrices, such as FS and Gr, with averages of 2.4 and 8.0 non-zero elements per row, respectively, merge-intersections complete rapidly, resulting in queues emptying more often and triggering more reconfigurations. In SpMM, despite frequent reconfigurations, Fifer is gmean 2.2× faster.

We also examine the energy consumed by each system in Fig. 15. Each bar indicates the dynamic energy consumed by the memory hierarchy (red and orange), dynamic energy consumed by cores or PEs (green), and energy consumed due to leakage currents (blue). The systems with OOO cores not only suffer from considerable leakage currents but also consume significant dynamic energy per instruction. By contrast, the systems with reconfigurable PEs consume much less energy, due to their reduced area and higher performance: the static pipeline achieves gmean 12× better energy efficiency than the OOO multicore across all applications. Using Fifer further improves energy efficiency because applications complete faster and reduce the energy lost to leakage. In SpMM, the static pipeline suffers from increased main memory energy usage due to the larger memory footprint created by separate PEs working on different parts of the matrix; Fifer avoids this by keeping the entire pipeline's working set at a single PE. Otherwise, dynamic energy consumed by the reconfigurable fabric and the memory hierarchy in Fifer and the static pipeline remain largely the same. Overall, Fifer reduces gmean energy consumption by 1.5× over the static pipeline and by 19× over the 4-core OOO system.

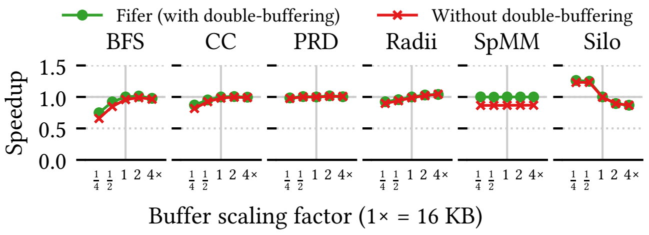


Figure 16: Fifer performance as the size of per-PE queue memory grows and how double-buffered configuration cells affect speedup.

8.3 Sensitivity to queue size and reconfiguration time

We now study Fifer's sensitivity to queue size. Fig. 16 reports gmean performance relative to the default configuration, a 16 KB queue memory, as it changes from 4 KB (0.25× the default) to 64 KB (4× the default). A second line allows us to study the effect of using Fifer's double-buffered configuration cells.

Fig. 16 shows that applications are sensitive to these parameters in different ways. First, BFS is mainly sensitive to queue sizes: its performance nearly halves with a 4 KB memory due to insufficient decoupling. CC, PRD, and Radii, which also benefit from the additional queue space, show similar trends in speedup as queue size increases. These applications do not see significant changes with the addition of double-buffered configuration cells. Because larger queues make reconfigurations less frequent, slower reconfigurations are irrelevant for large queue sizes.

Next, SpMM is mainly sensitive to reconfiguration latency: its performance reduces by about a quarter without the double-buffered configuration cells. As mentioned, SpMM is a control-intensive application; precisely because of this reason, SpMM's performance is flat across queue sizes: larger queues let producers fetch further ahead, but this is not used because merge-intersections redirect producers every few elements. Without double-buffered configuration cells, PEs cannot overlap loading a new configuration from memory as they complete the old stage's in-flight operations.

Finally, while Silo is insensitive to reconfiguration latency, its performance somewhat *decreases* as queue size increases. The larger queues enable so much parallelism that it significantly strains the memory hierarchy: with a 64 KB queue buffer, the working set of a stage matches the L1 size, and L1 hit rates fall from 66% to 62%. This increases pressure on the LLC and adds non-decoupled L1 misses that stall the PEs. This shows that extremely excessive decoupling can cause memory footprint issues, just as how prefetchers can trigger extra cache misses when running too far ahead.

Table 5 lists the average time a configuration resides on a PE, as well as the time needed to complete a reconfiguration for each benchmark. A typical application, BFS, executes each stage for about 140 cycles before reconfiguring, and spends around 13 cycles in reconfiguration, most of which is spent completing the previous configuration's in-flight operations. Applications on average spend 448 cycles per configuration with about 19.7 of those cycles spent reconfiguring. SpMM, with its frequent switching, shows why double-buffered configuration cells are crucial: being able to load the new configuration and finishing the current one in parallel minimizes dead time on a PE. Finally, the average time spent in a configuration is correlated to queue capacity; quadrupling queue storage increases the average residence time to 1488 cycles.

Application	BFS	CC	PRD	Radii	SpMM	Silo	Mean
Avg. residence time	140	279	927	564	30	1490	**448**
Avg. reconfig. period	12.5	13.9	20.4	27.7	12.6	60.1	**19.7**

Table 5: Average residence time of a configuration and time needed to complete reconfiguration (draining the old configuration, loading and activating the new configuration), in cycles.

Finally, we also evaluated a system that can perfectly overlap loading a new configuration with completing the previous configuration's operations, achieving zero-cost reconfiguration. This system improves performance by just 10% gmean (and up to 1.8× on SpMM's `Gr` input). We conclude that this alternative design is a poor tradeoff, as it incurs too much complexity for its limited benefits (for example, due to stalls from the outgoing stage, functional units would have to interleave the execution of multiple stages).

8.4 Sensitivity to stage count

So far we have evaluated fully decoupled application pipelines: we have used our partitioning technique to split the application into stages across every long-latency load. This produces regular stages that can execute on a CGRA fabric efficiently. Because Fifer time-multiplexes stages, splitting the program aggressively is the right approach. But for the static pipeline, the tradeoff is less clear: because each stage runs on a separate PE, having many stages improves decoupling but may worsen load imbalance.

To study this effect, we now consider alternative application pipelines where we judiciously merge stages to try to improve the performance of the static pipeline. For example, in BFS, the source-centric stages (processing the fringe, enumerating neighbors, and fetching distances) can be merged into a single stage, reducing a 4-stage pipeline to two stages. This pipeline still decouples across the most expensive indirection and, in the static pipeline, allows twice the number of parallel pipelines to be instantiated. However, the resulting first stage of this pipeline will incur stalls due to long-latency loads. In general, we choose which stages to merge by focusing on less-frequent indirections, merging low-activity stages, and keeping stage logic small enough to still fit in a single PE.

Fig. 17 shows the results of combining stages compared to the original (i.e., fully decoupled) Fifer and static pipeline, normalized to the performance of the *original* static pipeline and averaged across inputs. Overall, applications see very different effects. In BFS, because combining stages reintroduces coupling, the merged static pipeline is 4.4× slower than the original. CC sees a similar tradeoff as BFS, but PRD and Radii become slightly faster. In SpMM, stages are combined so that a single PE carries out the entire matrix multiplication for its share of rows. This exploits more data-parallelism at the expense of decoupling and benefits from small matrices like FS and Gr, which, as we discussed previously, cause Fifer to switch very frequently. Due to high speedups in those matrices, the merged static pipeline is gmean 4% faster than Fifer across inputs (Fifer

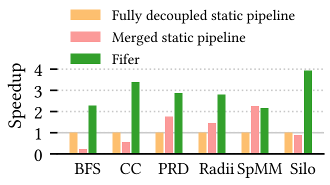


Figure 17: Performance of applications with merged stages.

using this coupled pipeline for the inputs that benefit from it and the decoupled pipeline for the others is 12% faster). Finally, Silo sees a slight performance degradation from the merged pipeline.

9 RELATED WORK

We now discuss other related work not covered in Sec. 2.

Application-specific accelerators often employ spatial architectures to improve throughput and reuse, especially when faced with irregular applications' unpredictable compute latencies and memory access patterns. To cope with this, prior techniques specialize their hardware to the problem. For example, SCNN [46] targets compressed sparse convolutional neural networks by directly embedding knowledge of the data structure format into the accelerator. Graphicionado [22] proposes pipelines for graph processing, and Q100 [68] proposes a spatial accelerator for databases. These designs trade a high degree of specialization for high performance in a narrow problem domain. As a result, prior efforts to accelerate these irregular applications often culminate in accelerators that are heavily tuned to the specifics of an application—benefiting only those whose needs are exactly met by the technique.

Decoupled access-execute (DAE) architectures [18, 21, 53, 60] allow a dedicated memory access unit to run ahead to fetch operands for a specialized execution unit. While some even exploit multithreading [55], DAE systems' overly rigid specifications, which strictly dictate the types of operations that occur in each specialized unit, cause tight dependences across units that limit performance.

Other pipeline-parallel techniques use pipelines built into reconfigurable fabric to exploit task parallelism [32], but do not time-multiplex configurations for better utilization. Coarse-grain pipelined accelerators (CGPAs) [34] also use HLS to make pipelines for FPGAs and ASICs, but these lack load balancing mechanisms.

10 CONCLUSION

In this work, we observed that reconfigurable spatial architectures can significantly improve compute intensity, but the unpredictable memory accesses and control flow of irregular applications are significant obstacles to good performance. We proposed Fifer, which augments a spatial reconfigurable fabric with dynamic temporal pipelines, allowing us to create pipeline-parallel applications whose stages are time-multiplexed onto this fabric. By layering temporal pipelines atop spatial pipelines, we demonstrated significant utilization and performance improvements on key application domains. Fifer thus makes efficient acceleration of irregular applications practical for reconfigurable architectures.

ACKNOWLEDGMENTS

We thank Maleen Abeydeera, Nithya Attaluri, Joel Emer, Axel Feldmann, Parker Huntington, Hyun Ryong (Ryan) Lee, Muhammad Shahir Rahman, Nikola Samardzic, Yifan Yang, Victor Ying, Guowei Zhang, and the anonymous reviewers for their feedback. This work was supported in part by DARPA SDH under contract HR0011-18-3-0007 and by Semiconductor Research Corporation under contract 2020-AH-2985. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David A. Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. 1995. The MIT Alewife Machine: Architecture and Performance. In *Proc. ISCA-22.*

[2] Anant Agarwal, John Kubiatowicz, David A. Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. 1993. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro* 13, 3 (1993).

[3] Anant Agarwal, Beng-Hong Lim, David A. Kranz, and John Kubiatowicz. 1990. APRIL: A Processor Architecture for Multiprocessing. In *Proc. ISCA-17.*

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian D. Koblenz, Allan Porterfield, and Burton J. Smith. 1990. The Tera computer system. In *Proc. ICS'90.*

[5] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM TACO* (2017).

[6] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. 2004. Spatial computation. In *Proc. ASPLOS-XI.*

[7] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. 2000. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proc. FPL.*

[8] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Helge Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *Proc. ASAP.*

[9] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021).

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proc. SoCC.*

[11] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proc. MICRO-52.*

[12] William James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks.*

[13] Jack Doweck and Wen-fu Kao. 2016. Inside 6th Gen Intel Core: New Microarchitecture Code Named Skylake. (2016). Hot Chips.

[14] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. 2006. A performance counter architecture for computing accurate CPI components. In *Proc. ASPLOS-XII.*

[15] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. SPR: an architecture-adaptive CGRA mapping tool. In *Proc. FPGA.*

[16] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proc. HPCA-22.*

[17] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R. Reed Taylor. 2000. PipeRench: A Reconfigurable Architecture and Compiler. *Computer* 33, 4 (2000).

[18] James R. Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P. B. Schechter, and Honesty C. Young. 1985. PIPE: A VLSI Decoupled Architecture. In *Proc. ISCA-12.*

[19] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. ASPLOS-XII.*

[20] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically Specialized Datapaths for energy efficient computing. In *Proc. HPCA-17.*

[21] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: decoupled supply-compute communication management for heterogeneous architectures. In *Proc. MICRO-48.*

[22] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. MICRO-49.*

[23] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *Proc. ISSCC.*

[24] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic CGRAs. In *Proc. FPGA.*

[25] Harry F. Jordan. 1985. *Parallel MIMD Computation: HEP Supercomputer and Its Applications.* MIT Press.

[26] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. 2004. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro* (2004).

[27] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. In *Proc. OOPSLA.*

[28] Dirk Koch and Jim Tørresen. 2011. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proc. FPGA.*

[29] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro* 25, 2 (2005).

[30] Charles E. Leiserson and Tao B. Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. SPAA*.

[31] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. MICRO-42*.

[32] Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, Yao Wang, and Shaojun Wei. 2017. Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware. In *Proc. ISCA-44*.

[33] Dajiang Liu, Shouyi Yin, Guojie Luo, Jiaxing Shang, Leibo Liu, Shaojun Wei, Yong Feng, and Shangbo Zhou. 2019. Data-Flow Graph Mapping Optimization for CGRA With Deep Reinforcement Learning. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* (2019).

[34] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. 2014. CGPA: Coarse-Grained Pipelined Accelerators. In *Proc. DAC-51*.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. PLDI*.

[36] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proc. FPL*.

[37] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth Copen Goldstein, and Mihai Budiu. 2006. Tartan: evaluating spatial computation for whole program execution. In *Proc. ASPLOS-XII*.

[38] Nangate Inc. 2008. The NanGate 45nm Open Cell Library. http://www.nangate.com/?page_id=2325.

[39] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proc. SOSP-24*.

[40] Marie Nguyen and James C. Hoe. 2018. Time-Shared Execution of Realtime Computer Vision Pipelines by Dynamic Partial Reconfiguration. In *Proc. FPL*.

[41] Quan Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *Proc. MICRO-53*.

[42] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proc. ISCA-44*.

[43] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A general constraint-centric scheduling framework for spatial architectures. In *Proc. PLDI*.

[44] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John M. Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems. In *Proc. MICRO-50*.

[45] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Clayton Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy L. Allmon, Rachid Rayess, Stephen Maresh, and Joel S. Emer. 2013. Triggered instructions: a control paradigm for spatially-programmed architectures. In *Proc. ISCA-40*.

[46] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proc. ISCA-44*.

[47] Guiqiang Peng, Leibo Liu, Sheng Zhou, Shouyi Yin, and Shaojun Wei. 2020. A 2.92-Gb/s/W and 0.43-Gb/s/MG Flexible and Scalable CGRA-Based Baseband Processor for Massive MIMO Detection. *IEEE J. Solid State Circuits* (2020).

[48] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proc. ISCA-44*.

[49] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled Software Pipelining with the Synchronization Array. In *Proc. PACT-13*.

[50] Aviral Shrivastava, Jared Pager, Reiley Jeyapaul, Mahdi Hamzeh, and Sarma B. K. Vrudhula. 2011. Enabling Multithreading on CGRAs. In *Proc. ICPP*.

[51] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proc. PPoPP*.

[52] Hartej Singh, Guangming Lu, Eliseu M. Chaves Filho, Rafael Maestre, Ming-Hau Lee, Fadi J. Kurdahi, and Nader Bagherzadeh. 2000. MorphoSys: case study of a reconfigurable computing system targeting multimedia applications. In *Proc. DAC-37*.

[53] James E. Smith. 1982. Decoupled access/execute computer architectures. In *Proc. ISCA-9*.

[54] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proc. VLDB* (2015).

[55] Michael Sung, Ronny Krashinsky, and Krste Asanović. 2001. Multithreading Decoupled Architectures for Complexity-effective General Purpose Computing. *SIGARCH Comput. Archit. News* (2001).

[56] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. 2014. Multithreaded pipeline synthesis for data-parallel kernels. In *Proc. ICCAD*.

[57] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. 2015. A CGRA-Based Approach for Accelerating Convolutional Neural Networks. In *Proc. MCSoC*.

[58] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. 2002. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. In *Proc. MICRO-35*.

[59] Michael E. Thomadakis. 2008. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. (2008). Hot Chips.

[60] Nigel P. Topham and Kenneth McDougall. 1995. Performance of the decoupled ACRI-1 architecture: the perfect club. In *Proc. HPCN*.

[61] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proc. SOSP-24*.

[62] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: Accelerating Machine Learning from Relational Data. In *Proc. ISCA-47*.

[63] Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. In *Proc. ISCA-41*.

[64] Dani Voitsechov, Oron Port, and Yoav Etsion. 2018. Inter-Thread Communication in Multithreaded, Reconfigurable Coarse-Grain Arrays. In *Proc. MICRO-51*.

[65] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *Proc. ISCA-47*.

[66] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *Proc. HPCA-26*.

[67] Clifford Wolf. 2014. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[68] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. 2014. Q100: the architecture and design of a database processing unit. In *Proc. ASPLOS-XIX*.

[69] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt - A High-Performance DSL for Graph Analytics. In *Proc. OOPSLA*.