Ripple: Asynchronous Programming for Spatial Dataflow Architectures

SOURADIP GHOSH, Carnegie Mellon University, USA
YUFEI SHI, Carnegie Mellon University, USA
BRANDON LUCIA, Carnegie Mellon University, USA
NATHAN BECKMANN, Carnegie Mellon University, USA

Spatial dataflow architectures (SDAs) are a promising and versatile accelerator platform. They are software-programmable and achieve near-ASIC performance and energy efficiency, beating CPUs by orders of magnitude. Unfortunately, many SDAs struggle to efficiently implement irregular computations because they suffer from an *abstraction inversion*: they fail to capture coarse-grain dataflow semantics in the application — namely asynchronous communication, pipelining, and queueing — that are naturally supported by the dataflow execution model and existing SDA hardware.

Ripple is a language and architecture that corrects the abstraction inversion by preserving dataflow semantics down the stack. *Ripple* provides *asynchronous iterators*, shared-memory atomics, and a familiar task-parallel interface to concisely express the asynchronous pipeline parallelism enabled by an SDA. *Ripple* efficiently implements deadlock-free, asynchronous task communication by exposing hardware token queues in its ISA. Across nine important workloads, compared to a recent ordered-dataflow SDA, *Ripple* shrinks programs by 1.9×, improves performance by 3×, increases IPC by 58%, and reduces dynamic instructions by 44%.

CCS Concepts: • **Software and its engineering → Dataflow architectures**; • **Computer systems organization → Parallel architectures**.

Additional Key Words and Phrases: spatial dataflow architectures, asynchronous programming

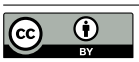
ACM Reference Format:
Souradip Ghosh, Yufei Shi, Brandon Lucia, and Nathan Beckmann. 2025. Ripple: Asynchronous Programming for Spatial Dataflow Architectures. *Proc. ACM Program. Lang.* 9, PLDI, Article 157 (June 2025), 28 pages. https://doi.org/10.1145/3729256

1 Introduction

Over the last decade, computing has turned to specialized architectures to scale performance and efficiency, from low-power sensing [32] to datacenter-scale machine learning [39, 66]. Fixed-function accelerators are fast and efficient, but workload diversity and non-recurring engineering costs precludes using a custom ASIC for every application [43].

Spatial dataflow architectures (SDAs) are a flexible, efficient alternative to specialized hardware accelerators [23, 32–35, 41, 54, 55, 58–61, 63, 67, 69, 71–73, 76, 81, 83–85, 87, 90]. An SDA and its compiler represents a program as a *dataflow graph* (DFG). The DFG's nodes are instructions, and its directed edges explicitly encode communication between instructions. There is no global instruction order (i.e., program counter) in the dataflow execution model. Instead, the *dataflow firing rule* governs execution: an instruction fires whenever its inputs are ready [27]. Upon firing,

Authors' Contact Information: Souradip Ghosh, Carnegie Mellon University, Pittsburgh, USA, souradip@cmu.edu; Yufei Shi, Carnegie Mellon University, Pittsburgh, USA, yfshi@cmu.edu; Brandon Lucia, Carnegie Mellon University, Pittsburgh, USA, blucia@andrew.cmu.edu; Nathan Beckmann, Carnegie Mellon University, Pittsburgh, USA, beckmann@cs.cmu.edu.



ACM 2475-1421/2025/6-ART157
https://doi.org/10.1145/3729256

an instruction typically generates a data *token* for dependent instructions that is sent through a *token queue*. The compiler maps the DFG onto the SDA's array of processing elements (PEs), or *fabric*, by placing instructions onto PEs and routing communication over a network on-chip (NoC). Token queues are routinely implemented in hardware as self-managing FIFOs, which buffer data tokens at PEs. Fig. 1 shows an SDA fabric and a small slice of the core SDA components: a single PE, the NoC, and hardware queues for input tokens.

Spatial distribution of compute and communication is key to an SDA's performance and efficiency. Spatial distribution exposes ample instruction-level parallelism (ILP), while dramatically reducing the switching activity of an SDA's PEs vs. a CPU [32, 76]. However, spatial distribution makes SDAs highly sensitive to *program size* because a DFG must fit onto an SDA to realize these advantages. A larger program requires more PEs and bigger NoC, which increases area and power. A larger program also often executes more instructions, wasting energy. Moreover, depending on fabric size, placing and routing a large program can take hours [7, 33, 62].

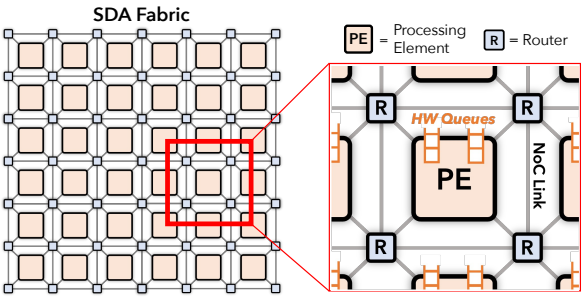


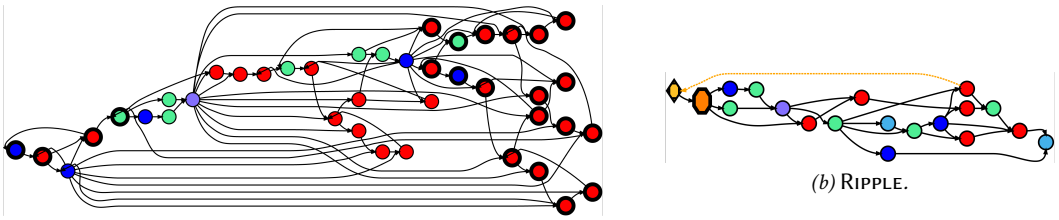
Fig. 1. Typical SDA fabric. Spatially distributed PEs execute instructions and communicate tokens over the NoC (via links and routers). Tokens are buffered at *hardware queues*.

Prior work has mainly focused on regular, affine loops, where dataflow compilers easily extract feed-forward DFGs from simple code regions (e.g., a dot product). These DFGs contain plentiful ILP and pipeline parallelism for SDAs to exploit [32, 41, 60, 67, 77, 84]; however, this approach neglects irregular code [23, 55, 69, 83]. Irregularity increases DFG size and reduces parallelism due to extra instructions for memory ordering and data-dependent control [33, 37, 64, 76].

Why do SDAs struggle on irregular applications? Many irregular programs feature regions of computation that execute and communicate data to other regions asynchronously. An SDA executes programs similarly: PEs fire asynchronously and hardware queues convey values from producers to consumers. These irregular algorithms and an SDA's execution model both express *task-level* or *coarse-grain dataflow parallelism*. SDAs should be well-suited to accelerate these workloads.

Unfortunately, existing SDAs commonly extract DFGs from sequential [8, 14, 23, 33–35, 41, 54, 55, 60, 71, 76, 81, 82, 87] or data-parallel [44, 73, 77, 84, 85] code that lack explicit primitives for dataflow parallelism. Instead, the programmer must express coarse-grain communication explicitly in code. The resulting DFG implements asynchronous communication via several control and memory instructions. Whereas, if only dataflow behavior had survived compilation, communication could map directly onto hardware token queues. The mismatch between high-level dataflow semantics and the low-level compiled code is an ***abstraction inversion*** that *(i) increases program size* and *(ii) requires conservative serialization* of memory operations.

Programming languages targeting SDAs should explicitly expose asynchrony and queueing, enabling the compiler to map dataflow operations directly to their corresponding hardware primitives. We introduce **Ripple**, a language and spatial dataflow architecture, which combines an asynchronous dataflow programming model with simple ISA extensions to eliminate the abstraction inversion. *Ripple* programs are composed of asynchronously communicating tasks written using *asynchronous iterators* that define queueing and pipelining semantics for tasks and their inputs. *Ripple* allows arbitrary structure, from feed-forward pipelines to feedback loops of



(a) Riptide. *(b)* Ripple.

Fig. 2. DFGs of breadth-first search (bfs) for Riptide and *Ripple*. *(a)* Riptide compiles C, requiring heavy control flow (red) to implement dataflow semantics. The DFG serializes execution. *(b)* *Ripple* directly supports token queues (golden/orange) and `atomics` (light blue), shrinking the DFG while exposing parallelism.

tasks that generate dynamic amounts of work. *Ripple*'s ISA abstracts SDA token queues, allowing the compiler to map asynchronous task communication nearly one-to-one onto SDA hardware.

By exposing dataflow semantics, *Ripple* decreases program size, increases parallelism, and saves energy. Fig. 2 demonstrates *Ripple*'s benefits; it shows the DFGs of breadth-first search (bfs) compiled for Riptide [33], a recent general-purpose dataflow architecture (left) and for *Ripple* (right). bfs traverses the graph using a FIFO queue, but its C implementation fails to express high-level queue semantics. Instead, the code implements a software queue using loads, stores, and data-dependent control flow. Riptide's compiler produces a large DFG that serializes execution to order the queue's loads and stores. Fig. 2a has a sea of control flow (red nodes) with few memory (green) and arithmetic (blue) instructions. On a real-world input, *83% of all dynamic instructions* were control flow.

The *Ripple* DFG (Fig. 2b) has fewer than half as many instructions. Traversal logic is implemented with an asynchronous code region, or `async`, that provides iterator semantics and a built-in work queue. bfs's queue semantics lowers directly to a single `spill` instruction (orange node), eliminating almost all control flow in the DFG. `spill` implements vertex queueing — including concurrent enqueue and dequeue operations — using existing hardware token queues; it automatically spills tokens to memory if the hardware queue is full. Moreover, *Ripple* supports `atomics` to synchronize concurrent memory accesses in asynchronous tasks, allowing concurrent updates in bfs. *Ripple* efficiently implements `atomics` via its `acquire` and `release` instructions (light blue nodes). `atomics` eliminate memory orderings imposed by C, further shrinking programs while increasing parallelism.

Contributions. This paper solves an abstraction inversion problem that hinders SDAs:

- *Language.* *Ripple* implements the asynchronous dataflow programming model to efficiently program SDAs. *Ripple*'s few primitives for explicit asynchronous communication, queueing, and pipelining are designed to map nearly one-to-one to SDA hardware (e.g., token queues).
- *Spatial Dataflow ISA.* *Ripple* introduces `queue` and `spill` instructions that expose token queues for safe, efficient task communication, and *Ripple* provides `acquire`/`release` instructions for efficient and composable shared-memory synchronization on SDAs.
- *Evaluation.* We implement a full *Ripple* software stack and architectural simulator. Across nine important workloads, *Ripple* achieves a gmean 3× speedup, gmean 58% gain in IPC, and nearly 2× reduction in program size and dynamic instruction count vs. Riptide, a recent SDA.

2 The Dataflow Execution Model

This section motivates *Ripple*'s contributions using a running example, bfs. Sec. 2.1 demonstrates how SDAs achieve high ILP through the dataflow execution model. Sec. 2.2 presents the abstraction inversion problem, which prevents SDAs and their compilers from extracting the ample dataflow parallelism present in many programs. *Ripple* solves the abstraction inversion problem by encoding dataflow semantics from the language down to the architecture (Sec. 3-Sec. 6).

2.1 A primer on dataflow execution

The dataflow model is the source of SDA performance. Dataflow execution achieves high ILP through three key properties: asynchronous instruction execution, fine-grained pipeline parallelism, and token queueing. We detail these properties by walking through a code section of bfs (Fig. 3). The program builds a search tree (`parents`) of a sparse graph using a software queue (`queue`).

How does dataflow execution work? Fig. 4 plots a DFG of L10-12, which performs neighbor updates in bfs. The DFG is implemented with the Riptide ISA, a recent general-purpose SDA [33]. For concreteness, we focus on Riptide's ISA, but the lessons generalize to other SDAs. Fig. 4 uses common instructions like `loads`, `stores`, and comparators (LD, ST, ==). The DFG uses steering control (ϕ^{-1} [21, 27]), routing values from producers to consumers using `steers` (T) instead of branches. T takes a data value and a conditional; if the conditional is `true`, it emits the data. Otherwise, it emits nothing. The top three nodes of the DFG perform the conditional in L10-11 to check for an unvisited neighbor: they load a neighbor, `nb = nbrs[i]`, then load `parents[nb]`, and compare it to −1. The bottom three nodes set `parents[nb]` to the current vertex, `v`. The DFG conditionally routes `nb` and `v` through Ts to the ST.

```
1. void bfs (u32 *ofs, u32 *nbrs,
2.  i32 *parents, u32 source):
3.  u32 *queue, head = 0, tail = 0
4.  void push(v): queue[tail++] = v
5.  u32 pop(): return queue[head++]
6.  bool empty(): return head == tail
7.  while (!empty()):
8.   u32 v = pop()
9.   for (i = ofs[v]..ofs[v+1]):
10.    u32 nb = nbrs[i]
11.    if (parents[nb] == -1):
12.      parents[nb] = v
13.      push(nb)
```

Fig. 3. bfs in C-like code.

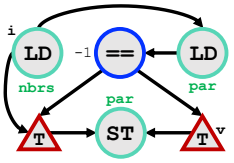


Fig. 4. DFG of L10-12 of Fig. 3. `parents` is abbreviated to `par`.

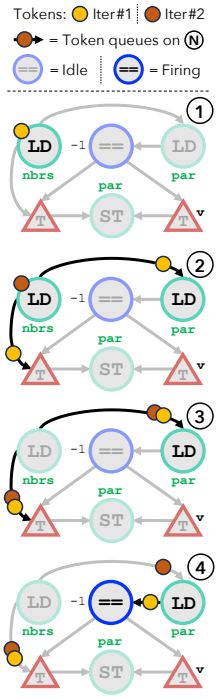


Fig. 5. Running Fig. 4 across two `for` loop iterations (L9, Fig. 3).

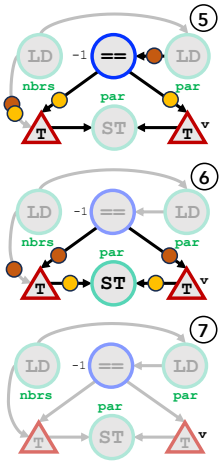
Fig. 5 shows an execution trace that highlights asynchronous instruction execution, pipeline parallelism, token queueing, and high ILP. This trace shows two iterations of the `for` loop (L9): the first (denoted by ● tokens) processes an unvisited neighbor, but the second (● tokens) does not. In this example, instructions fire and finish in one step, unless otherwise stated.

Step ①. A ● token that represents `i` is present and LD `nbrs` fires because it has all of its inputs. The LD takes a single step to finish. It produces output tokens that arrive at consumers on the next step.

Step ②. Output ●s from ① (sent along the darkened arcs) arrive and *queue* at both of LD `nbrs`'s consumers. One consumer, a T, does not yet have a complete set of tokens, so it does not fire. The other consumer, LD `par`, has all of its inputs (i.e., just the result of LD `nbrs`) and fires. However, load latency can vary, and this LD arbitrarily takes longer; it will continue past step ② and finish in ③. Meanwhile, a ● from the second iteration arrives at the LD of `nbrs`, enabling it to fire again and *pipeline* work over consecutive loop iterations. The second iteration's LD `nbrs` finishes in a single step and sends output ●s for the next step.

Step ③. The ●s from ② arrive and queue at both consumers. A ● queues behind the ● at LD `par` while the LD finishes up from step ②, and a ● also queues at the T while it awaits its other input. Neither consumer fires using ●s in ③. Once LD `par` completes, it outputs a ● that arrives in ④.

Step ④. LD `par`'s result arrives at ==, allowing it to fire. Concurrently, LD `par` fires again by dequeueing the ● and finishes in one step. Thus, LD `par` also pipelines tokens from consecutive iterations, just like LD `nbrs` in ①-②.



Step ⑤. Both △T s fire as they have full input sets (e.g., ●s). Since the first loop iteration finds an unvisited neighbor in this example, the ●s from (==) are `true`. △T s thus pass `nb` and `v` to arrive at the (ST) on ⑥. Simultaneously, (==) pipelines work, firing using the ●s from the (LD) of `par`.

Step ⑥. The (ST) fires using ●s and finishes in one step. Meanwhile, the △T s pipeline – this time, they fire using ●s. Since the second iteration does not update `parents`, the ●s from (==) are `false`. Since the condition is `false`, the △T s finish but produce nothing for the next step.

Step ⑦. All remaining tokens were consumed in ⑥. The execution ends.

Asynchronous dataflow firing and queueing are key to parallelism and throughput. Dataflow firing yields high ILP in bfs because independent instructions can fire asynchronously and in parallel as soon as inputs arrive. Multiple instructions fire in most steps (i.e., ② and ④-⑥) in Fig. 5. By contrast, a von Neumann program serializes instructions into a stream and forces the system to rediscover parallelism. Fig. 5 also achieves high pipeline parallelism by exploiting token queueing. Queueing decouples produces from consumers, allowing a producer to execute several times, even if its consumers are not ready (e.g., (LD) `nbrs` and its consumer △T in ①-③). This decoupling allows both loop iterations to pipeline through the DFG (e.g., in ④-⑦).

An SDA exploits this high ILP by spatially distributing a DFG on its fabric, allowing natural asynchronous instruction execution, pipelining, and queueing behavior inherent in the dataflow model. As shown in the example of an arbitrary SDA in Fig. 6, dataflow programs are commonly mapped onto an SDA fabric exactly as how they appear as a DFG [33, 35, 54, 73]. All six instructions are distributed across PEs, and DFG edges for communication are routed through the NoC. Token queues are implemented directly in hardware at each PE.

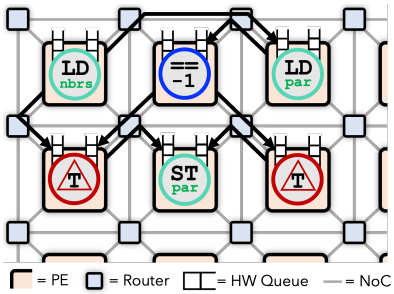


Fig. 6. An SDA implementing Fig. 5a. Instructions map to PEs and send tokens that buffer at hardware queues.

2.2 The dataflow abstraction inversion problem

Recent SDAs focus on extracting DFGs from sequential code [8, 14, 23, 33–35, 41, 54, 55, 60, 71, 76, 81, 82, 87] (often C), which lacks abstractions for asynchrony and queueing. Even when queueing is essential to the algorithm (as in bfs), programmers have no way to directly refer to queueing primitives that express asynchronous communication or token queues in hardware. Instead, the programmer writes a software queue with the same behavior as a hardware queue, but using many control-flow operations and memory accesses. *The queue abstraction is inverted*, leading to a serious performance problem. In contrast to Sec. 2.1, where the SDA achieves high ILP and IPC on a simple DFG of fine-grain data dependencies, *coarse-grained*, queue-based communication that spans the entire loop nest in bfs (Fig. 3) does not compile to a simple DFG. The extra instructions to support the software queue amounts to half of bfs's DFG (bold-bordered nodes in Fig. 2a).

Our observation: ignoring coarse-grain dataflow semantics leads to significant serialization. A compiler sees the implementation of **`queue`** as many unstructured memory dependences, rather than a dataflow dependence. If multiple accesses potentially alias and race, an SDA compiler enforces *memory ordering*: accesses are conservatively serialized at compile-time to follow program order, at the expense of performance. In bfs, memory ordering serializes **`push`** and **`pop`** (L8, L13, Fig. 3). Fig. 7 zooms into the (simplified) subgraph to show how **`push`** and **`pop`** are serialized. `loads` and `stores`

take an optional ordering input (red arcs). The `for` node abstracts loop iteration: every iteration, it emits `i` and a control token, `last`. `last` is `true` on the last iteration and `false` otherwise.

From the left, **pop** (i.e., the (LD)) produces `v`, the current vertex, and triggers the inner `for` loop to emit `i` and a neighbor (`nb`, L8-10, Fig. 3). If `nb` is unvisited (L11), it will **push** back to **queue** (L13). A steer (T) (top) routes `nb` to the `store` (ST) for **push**. Due to memory ordering, *all* **push**es in the inner loop must complete before the next **pop** can execute. Every execution of the (ST) emits an ordering token (outgoing red edge) that the bottom steer (T) gates. Only when `last` is `true` does the bottom (T) allow the latest ordering token from (ST) to pass through to (LD). In practice, several control-flow instructions implement each red edge and route tokens into and out of nested control structures. The effect compounds when multiple sets of aliasing reads and writes exist across the program.

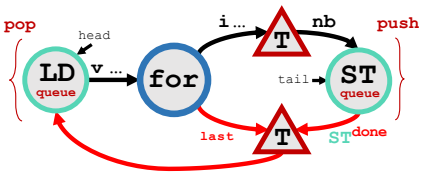


Fig. 7. Memory ordering of **pop** → **push** in Fig. 2a. Red edges indicate ordering.

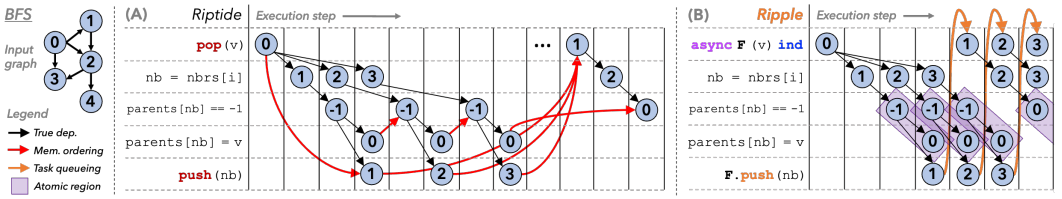


Fig. 8. A high-level pipeline diagram of bfs referencing C (Fig. 3) and *Ripple* code (Fig. 9). `parents` are initialized to (-1). (a) Riptide is marred with serialization of the software queue. (b) *Ripple* eliminates ordering with pipelined tasks and `atomics` for better IPC and speedup.

Memory ordering prevents asynchronous parallel execution. Fig. 8a shows a significant loss of parallelism from ordering using a pipeline diagram of bfs running on Riptide. The diagram illustrates the ordering (red edges) in Fig. 7: all unvisited vertices **push** before another is **pop**ped. Read-modify-writes (RMWs) to `parents` (L12) are also ordered. The ordering prevents the idealized, pipelined loop iterations shown in Fig. 5 and results in serialization (third-fourth rows of Fig. 8a). *This loss in parallelism stems from the lack of language-level abstractions that match dataflow execution.*

Efficiently ensuring memory correctness is a long-standing problem for all SDAs. Prior architectures and systems impose ordering in a variety of ways, but the net result is the same: serialization and performance degradation. Strict ordering models [76] and Riptide-style token-based ordering [15, 54] degrade parallelism, even when applied to code blocks to amortize costs [90]. Id's I- and M-structures [5, 9, 57] stand out as a mechanism to ensure memory correctness while retaining parallelism, but they require intrusive software changes. No existing solution ensures memory correctness without degrading performance due to serialization or making programming harder.

3 *Ripple* to the Rescue!

Ripple corrects the dataflow abstraction inversion by implementing the *asynchronous dataflow programming model*, in which programs are decomposed into asynchronous code regions that explicitly communicate data across regions. These abstractions enable the compiler to lower coarse-grain dataflow behavior to dataflow primitives implemented in SDA hardware. *Ripple* reduces DFG size and boosts parallelism by eliminating the inefficiency of software queues.

Ripple's prototype language provides *asynchronous iterators* or `asyncs`, each of which represents a task-parallel region of code. An `async` can contain arbitrary code written in an imperative language. `asyncs` execute asynchronously from one another. Communication between `asyncs` is explicit and

happens via queues that match the dataflow execution model. With *just a few changes to the code* to encode asynchronous dataflow and queueing, the resulting implementation in *Ripple* eliminates the memory ordering dependences that degraded performance.

Fig. 9 shows bfs implemented using *Ripple*. It has two asyncs, each of which has an implicit, dedicated task queue for its inputs. At runtime, `Init` (L2) kicks off the traversal with a *task*, or dynamic instance of an async, which pushes `source` to async `F`'s input queue. As `F` (L3-9) asynchronously receives inputs (e.g., `source`) to its task queue, it simultaneously dispatches tasks that run in *pipeline-parallel* fashion. We mark `F` with `ind`, expressing that its tasks are *independent*, with no memory ordering or data dependencies between any two tasks. As a result, `F`'s independent tasks use `atomics` to synchronize updates to `parents` (L6-9), avoiding memory ordering that would force a complete RMW to finish before another one starts. In *Ripple*, `atomics` pipeline, allowing many RMWs to launch before the first finishes. Finally, *Ripple* allows arbitrary, unrestricted task creation. Here, any task of `F` can `push` new inputs (i.e., unvisited vertices) back to create a future task of `F` (L11). *Ripple*'s execution enables high throughput processing over vertex neighbors.

```
1. def bfs (u32 *ofs, u32 *nbrs,
     i32 *parents, u32 source):
2.   async Init(): F.push(source)
3.   async F(u32 v) ind:
4.     for (i = ofs[v]..ofs[v+1]):
5.       u32 nb = nbrs[i]
6.       atomic<i32*>(&parents[nb]):
7.         if (parents[nb] == -1):
8.           F.push(nb)
9.           parents[nb] = v
```

Fig. 9. bfs in *Ripple*.

Fig. 2b shows that bfs gets much smaller, with a single backedge (orange) to a `spill` instruction using existing token queues. Fig. 8b shows how *Ripple* aggressively pipelines and concurrently processes vertices. Nodes ①-③ queue, dispatch, and pipeline using async (first row) and **`push`** (last row) while ⓪ finishes processing. Rows 3-4 of Fig. 8b show perfect pipelining because `atomics` execute in parallel. Ripple *achieves better performance with fewer resources.*

By embracing bfs's inherent dataflow behavior — i.e., queueing and asynchrony — *Ripple* removes bottlenecks that limit prior SDAs. First, async and push directly program token queues, obviating software queues. Second, explicit asynchrony removes serialization needed for sequential code. For bfs, *Ripple* need not sequentialize tasks, like the `while` loop in C demands. Vertices queue and drain safely and asynchronously. Third, *Ripple*'s architecture eliminates nearly all overheads for task queueing and dispatch. `queue` and `spill` primitives are handled by hardware queues that efficiently self-manage a buffer of tokens and dynamically route tokens to and from memory when needed. Queueing and dispatch thus appear automatic in the language. Ripple *need no extra runtime support for* asyncs *and few changes to monolithic code to reap benefits.*[1]

4 Programming with *Ripple*

The *Ripple* language implements the asynchronous dataflow programming model, which explicitly expresses dataflow semantics. *Ripple* provides a small set of language features designed around the dataflow execution model and existing SDA hardware. This section explores those features in the context of our *Ripple* prototype implementation. Sec. 3 showed an example *Ripple* program.

4.1 Asynchronous programming in *Ripple*

An asynchronous iterator or async defines a code region that accepts inputs through an implicit *queue*. async automatically iterates over inputs and dispatches work as a *task* (a dynamic instance of the async). Fig. 10a shows the syntax of an async. An async has a name, an arbitrary number of input arguments that forms a task input tuple (L1-2), and a body (scoped between L2 and L4) containing arbitrary code (C code in our prototype).

[1]See Appendix for more examples of C code vs. *Ripple* code.

A *Ripple* task executes with sequential semantics. Tasks of different `async`s run in parallel, and tasks of the same `async` pipeline in dataflow fashion. An `async` dispatches tasks following dataflow queueing semantics: they drain and launch in the FIFO order of the `async`'s input queue. Tasks do not need to finish in-order, e.g., if tasks take diverging control paths. An `async` with an empty input tuple (e.g., `Init` in Fig. 9) runs immediately.

```
async Name ([type _],
  ...) [ind] {
  // body
}
```

(a) `async` *syntax.*

```
async P (u32 x) ind {
  atomic<u32*>(&A[x]){
    A[x]++;
}}
```

```
async Q (u32 y) ind {
  atomic<u32*>(&A[y]){
    A[y] = y - 1;
}}
```

(b) Shared-memory updates with `atomic`*s.*

Fig. 10. `atomic`s synchronize racing shared-memory updates.

Independent work. By default, *Ripple* orders any dependencies across tasks of the same `async` in dispatch order (e.g., serializing updates to the same `static` variable). A programmer can, instead, mark an `async` as *independent* or **`ind`**, indicating the absence of dependences between tasks of that `async`. The absence of dependences exposes more pipeline parallelism. `ind`, however, does not prevent data races between pipelined tasks of the same `async`. Instead, the programmer must synchronize accesses to shared memory using `atomic`s, which Sec. 4.2 describes. Applying `ind` to an `async` essentially converts it from DOACROSS parallelism to pipelined DOALL parallelism [42]. Fig. 9 implements bfs with `ind` to aggressively pipeline unvisited neighbor updates.

`push` is *Ripple*'s only inter-task communication primitive. `push` creates and sends a tuple of task inputs to a named `async`, automatically queueing those inputs at that `async` (L2 and L8 of Fig. 9). *Ripple* allows feed-forward and cyclic communication among `async`s using `push`. *Ripple*'s architecture supports sending an unrestricted amount of data from one `async` to another, allowing the user to avoid thinking about communication bottlenecks or deadlocks related to queue capacity (Sec. 5).

`push` is also *Ripple*'s main task coordination primitive, with fence-like behavior that enforces dataflow ordering. A `push` does not execute until the completion of all reads, writes, and pushes that precede that `push` on its dynamic control-flow path from the entry point of the `async`.

4.2 Shared memory synchronization in *Ripple*

Tasks synchronize their accesses to shared memory using `atomic`s, designed after atomic sections or blocks [17, 18, 31]. Fig. 10b shows the syntax of an `atomic`. `atomic` is parameterized, taking an arbitrary-sized, typed list of input memory addresses, which the user reads from or writes to in the `atomic`'s body. `atomic`s have acquire-release semantics similar to two-phase locking [10, 30]. The body of an `atomic` executes only after acquiring exclusive access rights for all input addresses. The `atomic` releases its access rights when its body finishes executing. *Ripple* `atomic`s allow multiple inputs that may alias, and instances of an `atomic` with disjoint sets of input addresses pipeline. *Ripple* implements these semantics in its ISA (Sec. 5), keeping the programming model simple.

Ripple*'s memory model* is a data-race-free (DRF) memory consistency model [1], which gives undefined semantics to a program execution with a data race. `atomic`s order concurrent accesses to the same memory location(s), eliminating data races. `atomic`s have "all-or-nothing" atomicity semantics, and all accesses in an `atomic` appear to other `atomic`s to happen simultaneously. A valid execution of a *Ripple* program corresponds to a partial order over memory operations. Operations in the partial order are unordered, except for (i) the serialization of operations with direct dataflow dependences (including explicit ordering dependences) and (ii) the serialization of memory operations in `atomic`s that have overlapping input address sets.

4.3 Why is Ripple designed this way?

Each *Ripple* language primitive is designed to integrate with existing SDA hardware primitives without overly complicating the programming interface or the architecture. While *Ripple* is not the first to introduce asynchronous tasks (see Sec. 9), its implementation of the asynchronous dataflow programming model yields a concise language-to-hardware mapping.

Explicit dataflow semantics. Capturing dataflow behavior in code means expressing *asynchronous queue-based communication* and *coarse-grain pipeline parallelism* (Sec. 2). *Ripple*'s asynchronous iterators and communication primitives abstract exactly these properties: *(i)* `async` defines queueing semantics for task inputs and pipelining semantics for task execution; *(ii)* `push` enables arbitrary asynchronous task communication. Combined with its ISA, *Ripple* solves the abstraction inversion problem by preserving application-level dataflow semantics through compilation.

`async` *and* `push` *precisely capture an SDA hardware token queue's functionality.* In the dataflow model, a DFG edge represents an instruction dependence, and an SDA implements the dependence with a hardware token queue. A hardware queue is self-managing: it asynchronously receives input tokens and automatically drains whenever its associated PE is ready to fire another instruction (using the universal dataflow firing rule). `async` and `push` are designed to express dataflow dependencies just as an SDA does; they capture hardware queueing behavior in a number of ways. First, `push` defines an explicit dataflow dependence between `async`s in a program. Just as DFG dependencies are implemented by SDA token queues, a dependence defined by `push` is captured with `async`'s input queueing semantics. `async` defines an *implicit*, opaque work queue for its inputs that is only modified by `push`, rather than exposing queues to the user (e.g. via a special type). Limiting user control (e.g., by eliminating random access or preventing arbitrary queue instantiation [17, 28]) streamlines the API without losing the abstraction of token queues. Second, `async` defines iterator semantics to invoke automatic task dispatch rather offering user control over task launch. This design precisely expresses the automatic nature of draining a hardware queue. Any explicit "pop" or "receive" primitive that complements `push` would be a mismatch. Third, `push` is synchronization-free in software and automatically sends task inputs; it precludes the need for any runtime system. Instead, *Ripple* exposes hardware queues and arbitration in its ISA (Sec. 5), allowing the architecture to efficiently handle all task creation and routing.

Ripple*'s* `atomic` *simplifies both the language and architecture.* `atomic` alone can express arbitrary shared-memory synchronization, eliminating the need for a library of common, low-level primitives (e.g., compare-and-swap or atomic-increment). This design also reduces the need for complex architectural support; *Ripple*'s ISA adds only two instructions to directly implement `atomic` (`acquire` and `release`, Sec. 5.4). Common low-level atomics are implemented by composing `acquire` and `release` rather than introducing a specialized instruction for each one.

Eliminating ordering bottlenecks. *Ripple* is designed to make it easy to expose coarse-grained pipeline parallelism. An *independent* (`ind`) `async` frees the compiler to optimize without adding memory ordering across tasks. Additionally, `atomics` synchronize shared-memory accesses with high performance; they pipeline and eliminate conservative memory ordering to reduce DFG size.

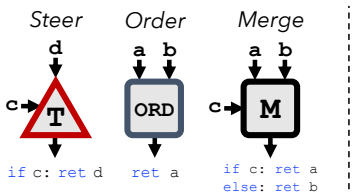
Simplicity. Motivated by the observations from Sec. 2.1-Sec. 2.2, *Ripple extends* a mainstream language, namely C in our implementation, rather than proposing an entirely new language. This strategy avoids a complete rewrite of programs, unlike other dataflow languages (Sec. 9). Rather, *Ripple* provides a small set of primitives that directly express the coarse-grained dataflow parallelism missed by mainstream SDA compilers. The *Ripple* version of bfs in Fig. 3 highlights the simplicity: an `async` and an `atomic` replaces the serialized `while` loop and the rest remains the same.

5 The *Ripple* Architecture

Ripple introduces simple dataflow ISA extensions that implements the queueing and synchronization primitives in *Ripple*'s asynchronous dataflow programming model.

5.1 Baseline architecture

Ripple builds on the Riptide ISA, which is a general-purpose ISA that supports full program execution in the *ordered dataflow* execution model. Sec. 2.1 demonstrated the ordered dataflow model, in which a PE fires its instruction after all of its inputs arrive via FIFO token queues (TQs). The ISA supports standard arithmetic and memory instructions and converts memory ordering into direct data dependencies. *Ripple* uses token steering to implement conditional control-flow (as shown in Sec. 2.1); the left side of Fig. 11 shows *Ripple*'s main control-flow instructions.



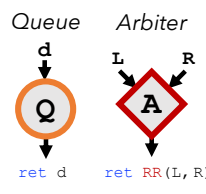


Fig. 11. Key instructions in *Ripple*. It borrows `steer`, `order`, and `merge` from [33] (left). It adds queue and `arbiter` (right).

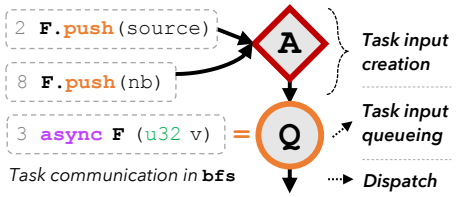


Fig. 12. `arb` (A) and queue (Q) (shown for bfs) perform task creation, queueing, and dispatch.

5.2 *Ripple* architecture extensions

Two simple ISA primitives implement TQs and arbitration for multiple pushes to a TQ. The right side of Fig. 11 shows these extensions.

`queue` instructions expose hardware TQs to the compiler. queues have a single data input and output, passing its input token unmodified to its output. A queue can be marked to **`spill`**, which indicates the need to back the queue's input with memory (i.e., dynamic capacity). A `spill` is architecturally identical to a queue, but its unlimited capacity to buffer tokens prevents deadlocks stemming from overflow of a TQ in hardware (Sec. 5.3 describes this problem in more detail).

`arbiter` (arb) collects inputs pushed from different locations and fairly merges them into a TQ. `arb` takes two data inputs (L and R) and produces one output by selecting round-robin from inputs. If a single input is present, `arb` still fires, passing it as output.

`signal-arbiter` (sarb) is identical to `arb` but outputs the side chosen (not data): 0 for L, 1 for R.

`queue` *and* `arbiter` *directly implement language-level asynchronous communication.* An `arb`-queue pair is allocated for each `async` input. An `arb` selects tokens from pushes to an `async`, where each `push` is a separate `arb` input. We demonstrate using bfs (Fig. 9) and the DFG of `async` F's input queue in Fig. 12. The DFG shows a single `arb`-queue pair for F's input, `v`. In Fig. 12, there are two pushes to F, requiring a single `arb`. Note the simplicity in Fig. 12 over the same program compiled from C (Fig. 2a); only two *Ripple* instructions are needed to support the same dataflow queueing semantics that Fig. 2a implements with a sea of control flow (red, dark-bordered nodes). By abstracting token queues, *Ripple* language primitives can directly program existing SDA hardware.

Ripple *supports any number of* `async` *inputs and* `push`*es by composing arbitration instructions.* Multi-input `async`s require more care than single-input ones (e.g., F in bfs). Fig. 13a shows multiple `push`es to `async` F, which has two inputs. Fig. 13b shows the resulting `arb`-queue pairs for each input. The risk is that distributing input tuples without synchronization can erroneously

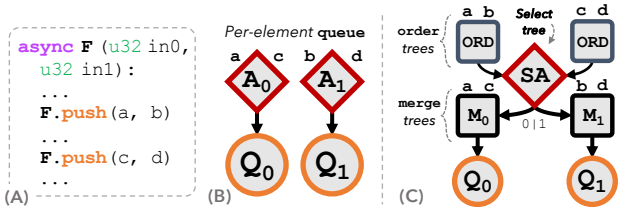


Fig. 13. General task input synchronization. (a) A multi-input async, F, and 2 pushes to it. (b) arb-queue pairs per element. (c) order-, select-, and merge-trees ensure tuples are ready, picks one, and emits per-element tokens to queue.

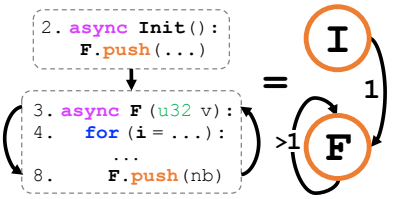


Fig. 14. Task dependence graph (TDG) of bfs.

interleave tokens that arrive at different times. *Ripple* requires logic to choose only a complete input tuple. The logic has three parts: *(i)* tuple synchronization, *(ii)* tuple arbitration, and *(iii)* element-wise token selection. Fig. 13c illustrates an `order-tree`; it waits until an entire tuple is ready (e.g., both `a` *and* `b` in `<a,b>`). Once a tuple is ready for arbitration, its `order`-tree emits a token that feeds a *select-tree* of `sarb`s. The select-tree emits boolean signals that encode which tuple it picked. Fig. 13c shows a select-tree, comprising a single `sarb` since there are only two tuples (`<a,b>` and `<c,d>`). Finally, the signals guide the chosen tuple, routing tuple-element-wise to their respective queues. `merge`-trees perform element-wise selection; each `merge` outputs a token based on a condition. In Fig. 13c, a `merge`-tree per tuple element takes the select-tree as its condition input (e.g., `SA` $\longrightarrow$ M_0). `arb`s in Fig. 13b lower to `merge`-trees in Fig. 13c to perform element-wise selection. `order`-, select- and `merge`-trees grow logarithmically with the number of pushes.

5.3 Preventing deadlock in *Ripple* tasks

While *Ripple*'s ISA abstracts hardware queues and implements arbitrary communication between asyncs, the resulting DFGs can produce *deadlock* when those queues do not have enough capacity. *Ripple* identifies deadlock in DFGs and avoids deadlock using the `spill` instruction.

How might* Ripple *programs deadlock? bfs (Fig. 9 and Fig. 14) illustrates the risk for deadlock. `async F` (L3-8) pipelines tasks without memory ordering between them. The *asynchrony* of both task creation and dispatch allows for rapid, unrestricted queueing of inputs.

The dynamic rate of task creation in `async F` is higher than the rate of dispatch because each task may push many neighbors while processing a single vertex. With a finite-size queue implementation, `F` exhausts its resources, blocking more inputs from being pushed. This blocking cascades through the DFG, eventually forming a cycle and deadlocking the program when nothing can fire.

Ripple captures this notion using a *task dependence graph* (TDG) [25, 79] that attaches relative rates of communication to dependent tasks. Fig. 14 shows the TDG for bfs lined up with code snippets. Nodes are asyncs (Ⓕ and Ⓘ) and arcs are task creation points (i.e., pushes). Importantly, when `F` dispatches, it pushes at most `i` inputs back to `F` (L8); `i>1` if a vertex has many unvisited neighbors. The TDG encodes the cycle and the relative rate of task creation to dispatch on each arc. Ⓕ↻ is then labeled with a variable rate: >1. *Variable rates along a cycle indicate a possible deadlock.*

***The key observation: task inputs must have space to queue to guarantee forward progress in* Ripple.** Cyclic task communication is a property of a *Ripple* program. For instance, bfs *must* complete the `for` loop (lines 4-8 in Fig. 14) to finish one of `F`'s tasks, so it must have space to queue the new inputs for `F`. Common deadlock avoidance schemes fail to support cyclic, dynamic routes; they suggest eliminating cycles [25] or adding minimal, finite buffering [68] unsuitable for dynamic token queueing. Imposing extra control flow to throttle progress yields a DFG similar to one extracted from C, bloating it and eliminating any asynchrony or parallelism.

Only dynamically sized, on-demand buffering avoids deadlock in* Ripple *programs. Once a queue runs out of buffering, *Ripple* automatically *spills* task inputs to a *backing queue* (BQ) in memory. *Ripple* accomplishes spilling with the specialized `spill` instruction. `spill` is functionally equivalent to queue, but it automatically reroutes tokens to the BQ whenever `TQ` storage runs out using an internal state machine (see Sec. 6.2 for its microarchitecture). A single instruction to manage both `TQ` and BQ *(i)* retains the asynchrony of task input creation and dispatch and *(ii)* avoids extensive resource consumption of distributing logic across many added instructions. *Ripple* prevents deadlock by analyzing the TDG and marking queues as `spill`s in the compiler.

5.4 Architecture for *Ripple* `atomic`s

Ripple introduces `acquire` (`acq`) and `release` (`rel`) instructions to directly implement `atomic`'s acquire-release semantics. `acq` and `rel` make requests (`req`) to the SDA memory system to acquire rights to an address and release it. `acq` and `rel` can compose to handle multi-input `atomic`s and aliasing input addresses, while maintaining high performance.

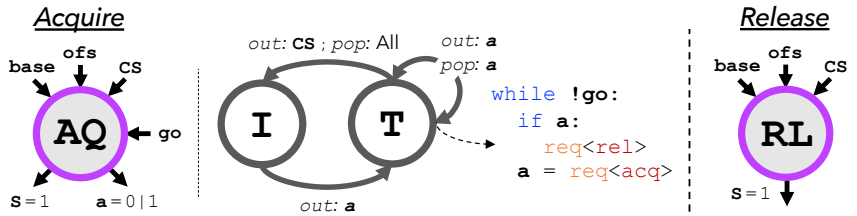


Fig. 15. `acq`/`rel` implement *Ripple* `atomic`s and acquire/release addresses. `acq` has a state machine (middle).

***Synchronizing memory accesses using* `acq` *and* `rel`.** `acq` and `rel` are outlined in Fig. 15 and put in action in a read-modify-write `atomic` (Fig. 16). Each address requires an `acq`-`rel` pair that envelops `load`s and `store`s in the DFG (i.e., two pairs in the example). To guarantee atomicity, *all* `acq`s must acquire before proceeding with dependencies. `acq`s acquire in parallel and synchronize with a reduction tree that controls if they proceed. Otherwise, all `acq`s must release, if necessary, and try again. The DFG shows a single-node tree (Ⓐ) that reduces and routes to both `acq`s.

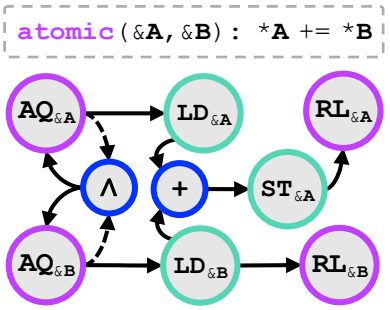


Fig. 16. DFG of an `atomic`. `acq`-`rel` pairs sandwich `load`s/`store`s for atomicity. `acq`s synchronize first (dashed arcs).

`acq` takes 4 inputs: a `base` and `offset` (`ofs`) to form an address, an optional control signal (`CS`) if ordered with other `atomic`s, and a signal, "go", that decides if it should retry. `acq` has two outputs: `a`, a status returned from an address acquire request (`req<acq>`), and `S`, a final success status.

Each `acq` starts at state `Init` (Ⓘ) once it receives `<base,offset,[CS]>` and sends out `req<acq>`. It then switches to `Try` (Ⓣ) and runs the loop in Fig. 15. Until `go=1`, the `acq` passes the return value from `req<acq>` as output `a` to the reduction tree and waits for a new `go` input. Importantly, an `acq` must *release* (i.e., `req<rel>`) an acquired address and retry if other `acq`s failed (i.e., `<a=1,go=0>`). Once `go=0`, `acq`s revert to Ⓘ and send `S`, shown as solid DFG arcs from `acq`s to dependent memory instructions (e.g., $\texttt{AQ}_{\texttt{\&A}} \rightarrow \texttt{LD}_{\texttt{\&A}}$). They then consume inputs and wait for a new address.

`rel` takes `base`, `ofs`, and `CS`. It invokes a release request (`req<rel>`) once it receives `CS` from completed memory operations upstream (e.g., $\texttt{ST}_{\texttt{\&A}} \rightarrow \texttt{RL}_{\texttt{\&A}}$). Like `acq`, `rel` emits `S`, an ordering token. `rel`s do *not* need to synchronize. A concurrent `atomic` with the same input set cannot

start unless *all* addresses are acquired, allowing even a single unreleased address to block another `atomic`. This feature maintains atomicity and enables `rels` to remain *disjoint* (as seen in Fig. 16).

5.5 Preventing deadlock in *Ripple* `atomics`

`atomics` must be have enough token buffering from `acqs` → `rels` in the DFG so acquired addresses are not held perpetually. `atomics` that produce a live-out (e.g., an atomic `load`) consumed by another `atomic` can cause deadlock if both share aliasing inputs. Consider a consumer `atomic` that waits on an address that a producer `atomic` has acquired but has not released. A "backpressure" or blocking cascade (similar to Sec. 5.3) can stall instructions in the producer, preventing it from executing its `rels`. The data dependence in the DFG that causes backpressure *and* the dynamic memory dependence between `atomics` creates an implicit deadlock-inducing cycle.

Ripple *prevents deadlocks between* `atomics` *by reapplying* `spill`. `spills` are inserted at any live-outs that yield data dependencies between `atomics` to provide abundant buffering. On-demand buffering enables forward progress so `atomics` can eventually release all acquired addresses.

6 *Ripple*'s Implementation

We implement a full *Ripple* system: a language prototype, compiler, and spatial dataflow architecture.

6.1 *Ripple*'s language prototype and compiler

We built the *Ripple* language prototype on top of C, using its types and scoping. *Ripple*'s `async` mimics C function signatures (e.g., Fig. 9), simplifying frontend compilation without adding new conventions. *Ripple* generates optimized DFGs, lowering *Ripple* code to LLVM-IR and expanding prior work [33] to insert control flow, enforce memory ordering, and prevent deadlock.

Ripple*'s frontend* generates a single, well-formed control flow graph (CFG) in LLVM-IR for each program (see Fig. 17 for an example). Frontend passes convert `push`, `async` iteration, and `ind` to intrinsic functions (see `%a = @A.iter()` in block `A.body`). Each `atomic` is similarly translated; a pair of `_acq`/`_rel` intrinsics surrounds its scope. `async` and `atomics` are converted to basic blocks via `clang` (e.g. `async A` is `A.body`, `if`, and `else`). To convert several `asyncs` to one CFG, the frontend adds glue code (gray blocks) and transforms `async` subgraphs to loops. A well-formed CFG importantly avoids irreducibility [2] and maintains assumptions made by LLVM (e.g., in dominance analysis). Glue code is not lowered further; it only streamlines compiler analyses.

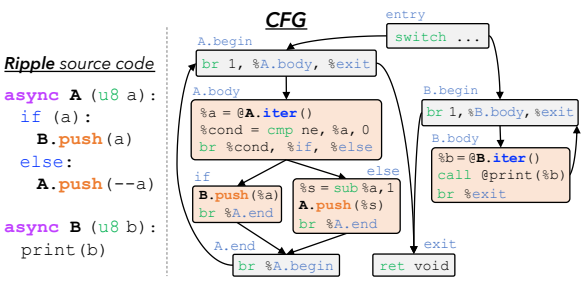


Fig. 17. *Ripple*'s frontend produces a well-formed control flow graph (CFG, as abridged LLVM-IR) with glue code (gray blocks) and `async` code (bright-colored blocks).

***Memory ordering for* Ripple.** Before lowering to the DFG, *Ripple*'s compiler performs memory ordering analysis, similar to [33]. *Ripple* borrows Riptide's analysis framework, which (*i*) leverages alias analysis to build an ordering graph, (*ii*) optimizes it with a specialized transitive reduction, and (*iii*) inserts data dependencies to enforce happens-before arcs.

Ripple's compiler builds on this by handling `atomics`. `_acq`/`_rel` pairs become nodes in the ordering graph. New happens-before arcs are added: *(a)* `_acq` → paired `_rel`, *(b)* `_acq` → `loads`/`stores` in its `atomic`, *(c)* `loads`/`stores` in an `atomic` → its `_rel`, *(d)* `_rel` → *all reachable and aliasing* `_acq`. *(a)* and *(b)* enforce that an `atomic` must acquire its addresses first. *(c)* compels reads and

writes to finish before releasing. *(d)* orders `atomics` with other `atomics` that have aliasing input addresses (inferred via alias analysis) in the same `async`. The compiler transitively reduces the ordering graph to prune arcs and enforces remaining edges. Importantly, atomic reads/writes are only ordered with each other and their `_acq`/`_rel` pair. Different `atomics` then run concurrently.

DFG generation. *Ripple* creates a DFG from LLVM-IR. It leverages `steer` and `stream` insertion from Riptide for control flow and optimized loop iterators. Next, every `async` parameter maps to an `arb`-queue pair in the DFG. `push` intrinsics are converted to input arcs for `arb`s and iterators (e.g., `.iter()`) to outgoing arcs from queues. When necessary, `order`-, select-, and `merge`-trees are added for multi-input `async`s or >2 `push`es to an `async`. Finally, `atomics` are lowered by mapping each `_acq`/`_rel` intrinsic to `acq`-`rel` pairs in the DFG per address. A reduction tree of `and`s are inserted for multi-input `atomics`. Otherwise, the "go" input is configured to 1 for a single-`acq` `atomic`.

Spill analysis. The *Ripple* compiler prevents deadlock by marking or inserting `spill`s. It first builds a task dependence graph (TDG, Sec. 5.2) from each `push` in the program. Nodes are `async`s and arcs are `push`es. Arcs are conservatively labeled with a variable rate (>1) if the associated `push` resides in a loop; i.e., each task of the `push`'s parent `async` can create many tasks itself. Arcs are labeled with 1 otherwise. Variable-rate arcs along a TDG cycle indicate potential deadlock. *Ripple* then picks any queue in the DFG that corresponds to a node along a cycle and marks it as `spill`.

The compiler also prevents deadlocks for `atomics` (Sec. 5.5). It first checks for dependent `atomics`; i.e., if any live-out from an `atomic` in the DFG reaches another `atomic`. If dependent `atomics` access aliasing memory, a `spill` is inserted after each live-out that reaches the other `atomic`.

6.2 *Ripple* microarchitecture

We describe the microarchitecture of a PE for `spill` and a `directory` that manages acquire/release requests for `atomics`. Both require small hardware changes to integrate with a real SDA system.

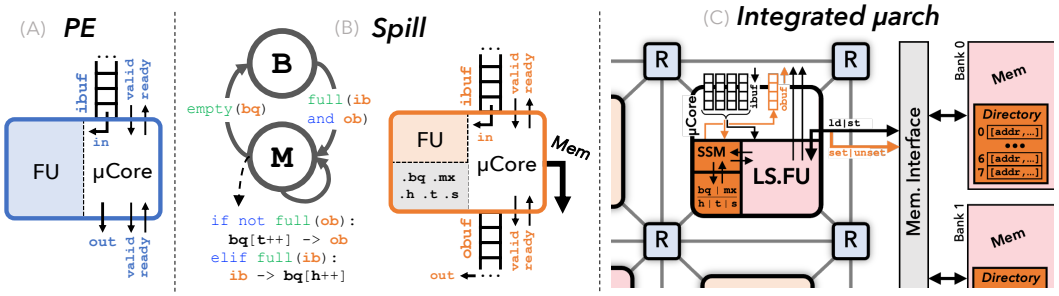


Fig. 18. (a) *Ripple* PE microarchitecture (μarch). (b) `spill`'s μarch has both `ibuf` and `obuf` (abv. `ib`, `ob`) and redirects or drains tokens with a FSM and internal state (gray portion). (c) *Ripple*'s μarch components (orange) added to Riptide's memory PE (left; e.g., the `spill` state machine or SSM) and memory subsystem (right).

Microarchitecture for spilling. `spill` prevents deadlock with on-demand buffering. We propose a PE for `spill` borrowing from Riptide's PE. A PE (Fig. 18a) has a functional unit (FU) and μcore that communicates data and signals through the network-on-chip (NoC) using a `ready`/`valid` protocol. PEs use dedicated `TQ`s at inputs (i.e., an input buffer or `ibuf`) for token storage.

The `spill` PE (Fig. 18b) has an `ibuf` and `obuf` (output buffer) to increase `TQ` space. Once `TQ`s fill up, `spill` moves tokens to a backing queue (BQ) in memory, treated as a ring buffer. A state machine reroutes or drains tokens (from `ibuf` or BQ). The `spill` PE requires registers for a pointer to the BQ (`bq`), max capacity (`mx`), head/tail indices (`h`/`t`), and a bit, `s`, to track the state machine.

`spill` starts in the `Buffer` (Ⓑ) state, where tokens transfer `ibuf` → `obuf`. If both are full, `spill` transitions to `Memory` (Ⓜ), where tokens route to the BQ and drain from `ibuf` → BQ → `obuf`. In

Ⓜ, the PE has two choices (see code in Fig. 18b). It *(i)* fills the obuf by loading from the BQ (i.e., prefetching [76]) or *(ii)* stores to the BQ if its ibuf is full. spill (i.e., blocks instructions upstream) if its ibuf saturates while rerouting tokens. Once the BQ is empty, spill moves back to Ⓑ.

***Microarchitecture for* atomics.** *Ripple* uses a directory (dir), a hardware memory structure, to manage acquired addresses and talk with acq and rel using its API: set() and unset(). Fig. 19 shows pseudocode for each; set() implements req<acq> and unset() implements req<rel>. dir stores a tuple of metadata per address: <addr, atmid, ref, iid> to allow aliasing input sets and atomic instances to *pipeline*.

```
1  def set(addr, atmid, iid) -> bool:
2    if addr in dir: # There's an entry ...
3      if dir[addr].atmid != atmid: ret F # No match
4      elif dir[addr].iid != iid: ret F # No match
5      else: dir[addr].ref += 1 ; ret T # Match!
6    # @param addr isn't set, make a new entry
7    dir[addr] = (atmid, iid, ref=0) ; ret T
8
9  def unset(addr, atmid, iid) -> bool:
10   assert {addr, (atmid, iid, ref=Any)} in dir
11   dir[addr].ref -= 1 # Always decrement
12   if dir[addr].ref == 0: del dir[addr] # Released
13   ret T
```

Fig. 19. Pseudocode for the set() and unset() API. This API exposes the dir to acq and rel and also manages its metadata.

The *atomic ID* (atmid) is a static numbering of an atomic based on source code. It enables an atomic to safely acquire the *same* address multiple times if it appears in an input set. set checks for a matching atmid (L3) if a dir entry exists for addr, indicating an acq of the same static atomic reserved addr already. Similarly, dir records a *reference count* (ref) on acquisition (L5-7) so rel can safely release addr multiple times using unset. ref decrements for each unset() (L11), which removes the entry for addr when ref is 0 (L12). The language need not force atomics to use statically differentiable addresses because the directory manages it.

Finally, the directory records an *instance ID* (iid) per entry, that enumerates dynamic instances of an atomic. iid disambiguates instances of the same atomic (i.e., atmid) in case two concurrent instances have identical inputs. Atomicity is violated if multiple instances are allowed to run on shared inputs, so set blocks a new instance from progressing on an iid mismatch (L4). The iid enables the same atomic to safely *pipeline* concurrent instances with high throughput.

Integrating* Ripple *into an existing SDA fabric. *Ripple*'s microarchitectural extensions require few hardware changes. Fig. 18c shows the changes on Riptide's SDA fabric in orange. Riptide features several types of PEs, and *Ripple* leverages Riptide's memory PE to implement spill, acq, and rel. The memory PE already supports most of the logic required: spill, acq, and rel reuse the same load-store FU (LS.FU in Fig. 18c) and datapath to access memory or make set/unset requests. With spill's state machine (SSM), registers, and obuf added, the memory PE has similar hardware complexity to other stateful SDA hardware implemented in prior work, including Riptide's stream PE or Softbrain's memory stream engines [60]. Fig. 18c shows *Ripple*'s directory integrated into Riptide's banked memory subsystem, where the memory interface directs set/unset requests. Each directory bank holds 8 entries; across Riptide's 8 banks, the directory needs just 1KB total.

7 Methodology

Systems. We compare *Ripple* to Riptide, building a full *Ripple* software stack and modeling both systems in a simulator. We also evaluate against a sequential von Neumann (vN) architecture (modeling 1 instruction per cycle or IPC) on performance and program size metrics. The vN system is modeled by directly executing LLVM-IR instructions. We count arithmetic, memory, and branching instructions but exclude all others (e.g., ϕ-nodes) for a fairer comparison to the SDAs.

Compiler. Compilers for all systems use LLVM 14 [48] and apply -Oz to reduce code size.

Architectural simulator. We evaluate *Ripple* and Riptide using an in-house, event-driven dataflow simulator that directly executes DFGs.

Modeling Riptide: The simulator accurately models Riptide's SDA fabric, including pipelined PE execution, bufferless NoC, and instruction timing. Each PE is assigned one instruction that it executes for the duration of the program; the SDA does not support dynamic time-multiplexing [63]. We assume DFGs successfully map as one unit and hence do not model fabric reconfiguration. In fairness to Riptide, the simulator implements `ibufs` for instruction inputs at each PE (4 slots per `ibuf`) — follow-on work to Riptide found that input buffers improved performance with small hardware overheads [73]. Any token transferring between buffers (e.g., from a `spill`'s `obuf` to its consumer's `ibuf`) incurs a 1-cycle delay. Simulation of Riptide in its original configuration (i.e., with `obufs`) was within 10% of its RTL execution cycles reported in [33] on two sample programs (bfs and dense matrix-vector multiplication).

Modeling arithmetic and control-flow instructions: Arithmetic instructions have 1-cycle latency. We simulate control flow for Riptide with 0-cycle latency and no buffering. This models Riptide's combinational control-flow-in-NoC (CFiN) technique, wherein bufferless NoC routers perform control flow with no delay for better performance (e.g., over control flow on PEs). We empirically confirmed Riptide's findings that it performs better with CFiN.

Modeling Ripple: We run *Ripple* programs using the *Ripple* microarchitecture that extends the Riptide fabric (Sec. 6.2). Simulation is identical to Riptide apart from control flow. *Ripple* executes most control flow on a PE because it prefers buffering for better throughput and pipelining. *Ripple* only executes reduction trees (e.g., for atomics or accumulators) combinationally, where latency is more critical than throughput. `arb` and `queue` instructions have 1-cycle latency, and `spill` has 1-cycle latency when it only drains from its hardware queue (i.e., the `Buffer` state).

Modeling memory: `loads`, `stores`, and `spills` talk to a single-level, 8-way banked memory (e.g., like Riptide's hardware). Each bank queues memory requests and processes one per cycle with 1-cycle latency; bank conflicts can increase latency. *Ripple*'s `dir` is also 8-way banked, with 8 entries per bank (as in Sec. 6.2). `dir` has two request queues for `set` and `unset`. Each `dir` bank runs one `set()` and one `unset()` per cycle with 1-cycle latency each.

Applications. We evaluate *Ripple* on nine workloads in graph analytics and linear algebra, demonstrating that SDAs can efficiently support irregular code. By contrast, Riptide was evaluated on workloads with simple dependencies (e.g., dense linear algebra). Our irregular workloads highlight the dataflow abstraction inversion problem solved by *Ripple*.

Breadth-first search (bfs) is Fig. 3 and Fig. 9. *Topological sort* (ts) partially orders nodes of a directed acyclic graph. *K-core decomposition* (kc) produces a maximal induced subgraph, S, s.t. $deg(v) \geq K,\ \forall\ v \in |V_S|$ ($K = 128$). *Connected components* (cc) assigns components to vertices via label propagation. *Single-source shortest path* (sssp) finds the shortest path from a source to all vertices. *PageRank-Delta* (prd) [53] ranks vertices, only refining ranks that change $> \delta$ in each step. *Matrix addition* (ma) performs a weighted sum of sparse matrices (dense output). *Stencil* (st) is a 2D, 5-point stencil using the Jacobi method (`iters` = 100). Finally, we evaluate *Dense matrix-vector multiplication* (dmv) to cover workloads that SDAs like Riptide already accelerate well.

Ripple and Riptide (i.e., C) implement the same algorithm for each program. Graph workloads follow data-driven, iterative algorithms with high parallelism and fast convergence properties [26, 36, 46, 52, 88]. We use push-style updates [11]. ma and st are derived from Polybench [89].

Inputs. Workloads were evaluated on two synthetic or real-world inputs each (Table 1). Inputs range in size and cover different structures (e.g. uniform v. power-law graphs).

8 Evaluation

We evaluate *Ripple* and show that it improves performance, IPC, performance-per-area, and shrinks program size over Riptide, a state-of-the-art, general-purpose ordered dataflow architecture.

App(s)	Graph		$\|V\|$	$\|E\|$	d_{avg}	d_{max}
[bfs, kc, cc, sssp]	KR	kron_g500-logn17	131K	5.1M	78	30K
	CA	roadNet-CA	2M	2.8M	2.8	12
prd	WG	web-Google	916K	5.1M	11.1	6.4K
	WS	web-Stanford	282K	2.3M	16.4	38.6K
ts	DL	DAG:Linux.call	320K	1.1M	7.2	80K
	DS	DAG:Synthetic	1M	4.4M	8.4	24

App	Matrix ($n \times n$)		n	Structure
ma	S0	Random-Sp-0	2^{14}	Sparse (nnz_{avg}: 16)
	S1	Random-Sp-1	2^{14}	Sparse (nnz_{avg}: 512)
st	SM	Small-Grid	2^{8}	Dense
	LG	Large-Grid	2^{9}	Dense
dmv	D0	Random-D-0	2^{11}	Dense
	D1	Random-D-1	2^{12}	Dense

Table 1. *Inputs.* Real inputs are fetched from SuiteSparse [45] (KR [6]; CA, WG, WS from [50]; DL is the SCC-DAG of [75]). DS is the SCC-DAG of a random Erdős-Réyni graph [20, 29]. S0, S1, D0, and D1 are random, uniform matrices. SM and LG are dense grids produced by Polybench. sssp adds random weights in [1, 128] to inputs.

8.1 Performance and resource utilization

Ripple *is fast.* Fig. 20 shows *Ripple*'s and Riptide's speedups over the vN baseline. Both *Ripple* and Riptide significantly outperform the sequential baseline by leveraging dataflow execution to extract high ILP (gmean 14.2× and 4.7×, respectively). Crucially, asynchronous tasks allow *Ripple* to unlock parallelism that Riptide cannot exploit from C, amounting to gmean 3× speedup over Riptide. Queue operations are performance-critical in many irregular programs, and *Ripple*'s `async` iterators and `atomics` make these fast by eliminating software queues and memory ordering. They pipeline

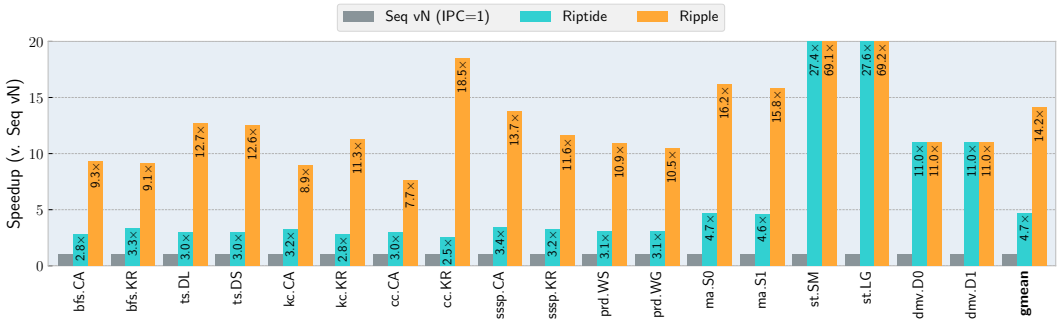


Fig. 20. Speedup of *Ripple* and Riptide over a sequential von Neumann (vN) baseline (IPC=1) on all applications. *Ripple* is faster than Riptide by gmean 3× and the vN baseline by gmean 14.2×.

better than Riptide, accounting for most of the speedup. For example, *Ripple*'s cc sees a massive 7.3× speedup over Riptide on the power-law graph KR because queueing and atomic updates for label propagation are on the critical path. ma shows the benefit of *Ripple* avoiding serialization on independent, pipelineable `asyncs`, while Riptide is forced to conservatively serialize ma's computation (which additionally disrupts pipelining), leading to *Ripple*'s 3.5× advantage. *Ripple* is on-par with Riptide's performance in its best domain: dense linear algebra. Riptide's compiler extracts maximal pipeline parallelism from dmv's simple loop nest to produce a feed-forward DFG with no memory ordering, leading to a 11× improvement over the vN baseline. *Ripple* performs just as well; however, expressing dmv's computation with `asyncs` offers little advantage because affine loops already express dmv's parallelism well.

Ripple *massively improves resource usage.* While *Ripple* programs enjoy fewer total (static) instructions, they execute substantially *more* IPC than Riptide. Fig. 21 plots IPC (split by instruction type), measured as number of PE firings divided by execution time. *Ripple* improves IPC by gmean 58% and routinely achieves >10 IPC on *smaller* programs.

Ripple eliminates parallelism-killing artificial dependences to achieve high IPC. It executes more computational (arithmetic and memory) operations each cycle than the SDA baseline, which instead spends most of its time on control operations. Asynchronous tasks translate to better pipelining and high ILP. `arb` and `spill` often execute on every cycle, indicating high task throughput. st helps illustrate the effect of memory ordering instructions. Both Riptide and *Ripple* have high IPC, but

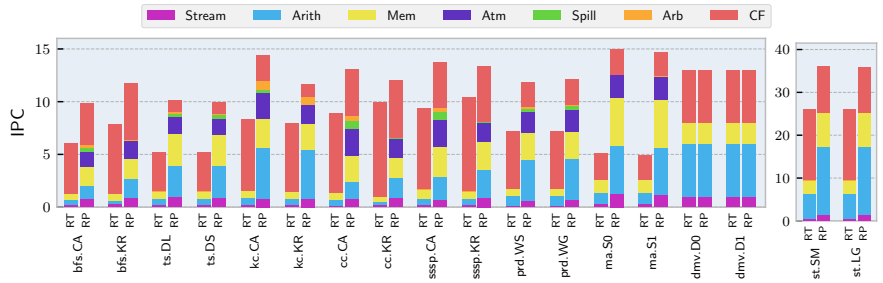


Fig. 21. IPC split by operator type. *Ripple* improves IPC on all applications (by gmean 58%) over Riptide.

Riptide wastes cycles on memory ordering. *Ripple* updates data rows in parallel (using red-black coloring), leading to its 2× memory IPC. *Ripple* implements a coarse-grain pipeline for dmv that compiles to the same DFG as from C, allowing *Ripple* and Riptide to achieve the same IPC on dmv.

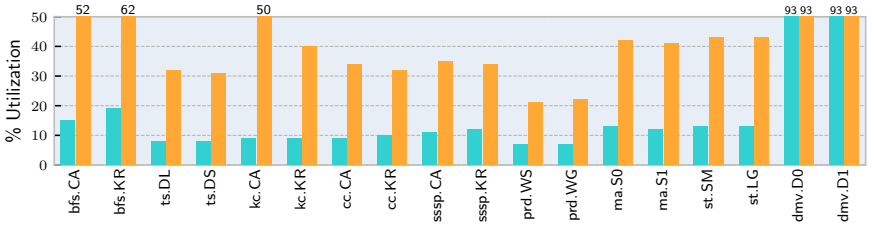


Fig. 22. Utilization of *Ripple* (orange) vs. Riptide (teal). *Ripple* shows higher utilization by eliminating control-flow bottlenecks while improving concurrency via tasks.

Fig. 22 shows utilization, defined as the IPC divided by program size (i.e. number of DFG nodes). *Ripple*'s high utilization on irregular workloads (often 30-40%) showcases its unique ability to improve parallelism while requiring fewer static instructions. prd is an exception; much of its time is spent in a computationally intensive loop for each vertex that updates residuals. Nonetheless, *Ripple* almost triples its utilization against the baseline. Reducing control flow is key to utilization gains, as *Ripple* requires fewer resources to run a program while still performing with higher IPC. *Ripple* and Riptide both perfectly pipeline dmv, allowing near 100% utilization.

***Improved resource usage coupled with small programs translates to high area-normalized performance for* Ripple.** Fig. 23 shows gains in performance-per-area – execution time divided by program size – for *Ripple* over Riptide. We use DFG size as a proxy for physical area consumption. *Ripple* improves by gmean 5.8× through reduced resource usage (smaller DFGs) and large speedups (more parallelism).

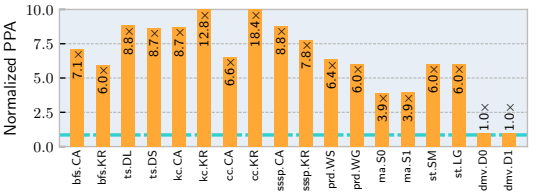


Fig. 23. Performance-per-area gains of *Ripple* over Riptide, with a gmean 5.8× improvement.

8.2 Characterizing *Ripple* programs

Ripple *produces small programs.* *Ripple* reduces program size by gmean 1.9× (and as much as 3×) over Riptide's dataflow programs and by gmean 1.4× over the sequential von Neumann (vN) system, as Fig. 24 shows. Riptide incurs high control flow overhead from excessive memory ordering and software queueing. vN programs need less control flow than Riptide; branching requires fewer instructions than steering, and a program counter enforces all ordering by default. However, vN programs still need many instructions to implement software queueing. *Ripple* tackles both sources of static instruction overhead by significantly reducing memory ordering (e.g., with

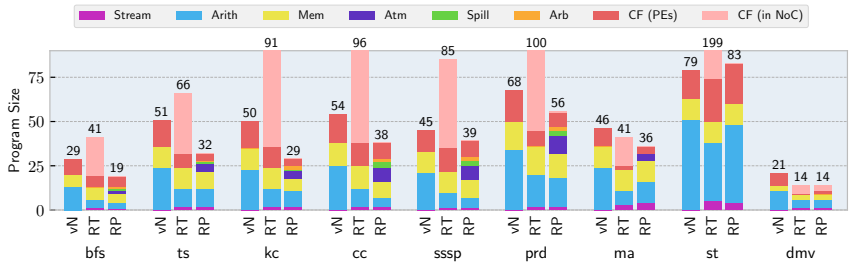


Fig. 24. Program size plotted as the number of DFG nodes or static instructions (split by operator type) for the vN system (vN), Riptide (RT), and *Ripple* (RP). Control flow for RT and RP is split by PEs (dark red) and control-flow-in-NoC (light red). RP shrinks program size by gmean 1.9× over RT and gmean 1.4× over vN.

`atomics`) and eliminating software queues (with `arb` and `spill`). As a result, *Ripple* programs have far fewer control flow operations over Riptide. ma shows other code size benefits of asynchronous programming. It does not use a software queue, but *Ripple* shrinks its DFG by splitting it into independent `asyncs` that pipeline with `atomics`. The resulting code has no ordering control flow across its loops, and no spilling, because its task graph is acyclic. Both *Ripple* and Riptide produce small, nearly identical DFGs for dmv, as affine loops are easy to express and compile.

8.3 Estimating *Ripple*'s energy costs

Ripple's benefits come without high energy overheads, and with a decrease in key sources of energy consumption. We focus on dynamic instruction count and memory usage as proxies for energy.

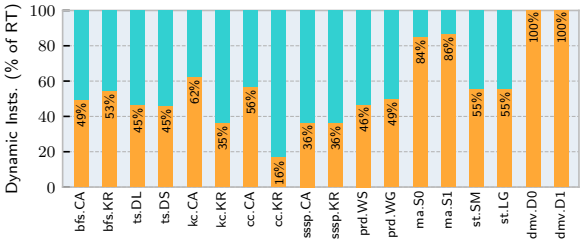


Fig. 25. *Ripple* (orange) reduces dynamic instructions, shown as a % of Riptide's execution (teal), by 44% on avg.

Ripple *executes fewer dynamic instructions.* *Ripple* reduces dynamic instructions by 44% on average (Fig. 25). *Ripple* eliminates the majority of instructions that implement software queues, which are especially common in graph applications. Additionally, kc, cc, sssp, and prd implement asynchronous, iterative algorithms that process a vertex many times (e.g. path relaxation in sssp). These algorithms finish faster when they discover high-degree vertices more quickly, which can depend on vertex processing order and timing. *Ripple* very quickly discovers vertices in a traversal and aggressively pipelines their processing, leading to faster convergence and shorter runtimes. Riptide instead serializes discovery and processing.

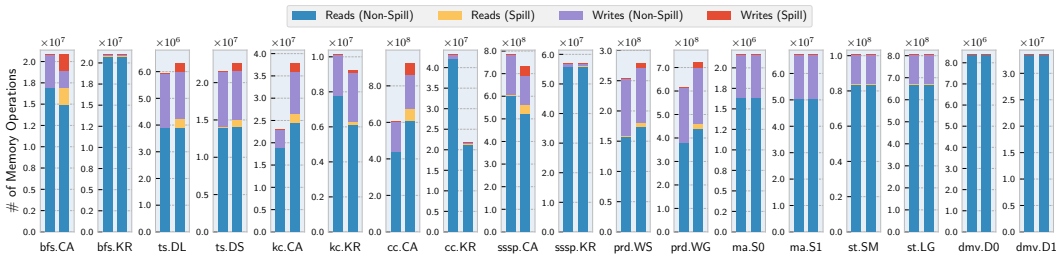


Fig. 26. Memory usage of *Ripple* vs. Riptide, split into spilling and non-spilling reads and writes.

Ripple *incurs modest spilling overheads.* Fig. 26 shows total memory operations executed for each application, and *Ripple*'s bars account for extra spill-induced reads and writes. Spilling to memory accounts for 7.6% of total memory operations on average in applications that need it. bfs.CA incurs the highest spill percentage at 18.8% of all memory operations. bfs and ts have

the highest per-application spill proportion at 9.9% and 9.7%, respectively. Generally, the graph applications spill more on uniform, high-diameter inputs (e.g. CA). Nevertheless, this tradeoff is a win for high throughput and performance across all applications.

Ripple *experiences reasonable but higher memory usage.* Fig. 26 also shows that *Ripple* can increase memory usage, often in iterative, work-inefficient programs. kc.CA and cc.CA execute 66% and 53% more memory instructions, respectively, with *Ripple*. Suboptimal vertex visiting order (i.e. work scheduling) coupled with aggressive pipelining degrade work efficiency since more vertices are revisited than needed compared to Riptide. kc.KR and cc.KR reduce memory usage similar to their reductions in dynamic instructions with *Ripple*: they discover high-degree vertices and propagate updates quickly. *Ripple* requires fewer memory instructions for sssp on both inputs, likely because its traversals differ from kc and cc. These results point to a need for better work scheduling, either based on priority, rank, or dynamic information [22].

Other factors contribute to higher memory usage. In ts, spilling and slight implementation differences in bookkeeping of in-degrees come to increase overall memory usage by 6.2%. *Ripple* executes 14% more memory operations on prd because it aggressively pipelines tasks, causing it to update residuals of some neighbors that have already converged. prd also reloads data from the graph on occasion to fetch neighbors; it reduces pipeline bubbles but increases memory usage.

When work efficiency is not in play, *Ripple* has identical memory usage as Riptide. *Ripple*'s bfs has no memory overheads with spilling since the traversal queues each vertex only once. ma, st, and dmv also have no change in memory usage since they compute a fixed number of updates. As with spilling, *Ripple* prefers to trade off reasonable memory overheads for much higher performance.

8.4 Common patterns in *Ripple* code

We identified common patterns and pitfalls from our experience writing applications in *Ripple*.

Expressing different forms of task parallelism. We found that *Ripple* is excellent at expressing worklist parallelism [65]. `async`'s queues naturally track outstanding work. Take prd: only outgoing neighbors of unconverged nodes (rank $> \delta$) need to be refined and `push`ed back to the work queue. Users need not iterate through all vertices in each round and poll for convergence.

Ripple also expresses DAG parallelism well. End-to-end asynchrony and `atomics` allow the user to discover task dependencies (often a DAG) *and* process them simultaneously. *Ripple*'s ts shows this pattern; it begins sorting as soon as any zero in-degree node is found, improving concurrency over the common strategy of computing a task graph and processing it in separate, disjoint stages [38].

Finally, *Ripple* encodes coarse-grain pipelines efficiently. We find that using `push` as coordination primitive to split computation leads to a natural pipeline structure. st exhibits this well; we use `push` to stagger and trigger updates at row-level granularity in a red-black colored grid. [2]

***Decoupling reads and writes across* `asyncs`.** Many RMWs or `atomics` in a single `async` can hamper performance because of memory ordering. We find that decomposing one `async` into multiple often naturally decouples and batches reads and writes with no memory ordering imposed across `asyncs`. Instead, we use `push` as a fence for correctness. The 5-point stencil update in st is an example; queries are decoupled from updates to the grid.

***Correctly using* `atomic` *with* `ind`.** Correct synchronization is difficult in parallel code, and the use of `atomic` is no different. `asyncs` marked `ind` add extra complexity because their tasks aggressively *pipeline*. Using `atomic` in the presence of `ind` is crucial to eliminate races from potentially many in-flight tasks from the *same* `async`. The `atomic` in bfs (Fig. 9) shows this in practice.

[2] See Appendix for C and *Ripple* source code for ts and st.

9 Related Work

Ripple builds on prior work in asynchronous dataflow, dataflow languages, and dataflow ISAs.

9.1 Programming for SDAs

Sec. 2 outlined the pitfalls of SDA compilers that extract DFGs from mainstream, sequential code. Few prior general-purpose languages target SDAs. Prior languages that do require a complete implementation overhaul, often including both software and hardware changes. We divvy these languages and SDAs by their parallel programming paradigm.

Implicit asynchronous parallelism. The functional dataflow language Id [4] assumes *all* code is implicitly parallel by default. However, functional languages like Id make common computation idioms difficult to express because they do not support mutable state. Id requires separate hardware support, called I- and M-structures [5, 9], to enable this common feature. Id also requires several instructions to launch parallel work on its target dataflow architecture [3].

Ripple takes an incremental approach: A few `asyncs` are enough to inform the compiler about high-level dataflow semantics in the program. Fine-grain parallelism can be successfully expressed and extracted by the compiler within an `async`. Moreover, by starting from common SDA hardware with shared memory, *Ripple*'s architecture remains simple and efficient.

Data parallelism. Recent SDAs like Plasticine [67] and SGMF [84] exploit massive fine-grained data parallelism replicated across onto large fabrics. They are programmed with data-parallel code: SGMF uses CUDA while Plasticine implements parallel patterns from the Spatial language [44]. While highly performant on regular code, users must morph iterative, irregular applications into map-reduce blocks that can underutilize the SDA's feed-forward pipeline stages [67]. Consequently, subsequent work continually adds language features and hardware to support workloads from other application domains [69, 70, 83, 85] (e.g., dMT-CGRA's message passing in CUDA threads).

Ripple, instead, starts from a point of generality and adds asynchronous communication to the language because it best models dataflow execution and SDA hardware; only few hardware changes are needed to achieve high performance *and* small programs.

Structured pipeline parallelism. Many SDA-adjacent systems target code decomposed into coarse-grain pipeline stages; StreamIt [79] and Raw [86] are archetypes of this model. StreamIt efficiently implements streaming programs as coarse-grained pipelines with FIFO communication. Raw supports StreamIt programs on a distributed architecture of many scalar cores. StreamIt and Raw do not support fine-grained dataflow execution like *Ripple* and other SDAs (they support message passing between cores). However, they analyze and optimize pipelines using the synchronous dataflow model [49], which restricts the structure of StreamIt pipelines. All pipeline stages, feedback loops, and asynchronous messaging channels [80] must exhibit statically analyzable bounds.

Ultimately, structured pipelines are unable to express arbitrary, asynchronous communication that governs dataflow firing and application-level dataflow behavior seen in irregular code like bfs. *Ripple*, instead supports arbitrary, unrestricted communication between `asyncs`, and can easily encode general pipelines *and* feedback loops without deadlock.

9.2 Conventional asynchronous and task-parallel languages

Message passing [19, 74], actor [24, 51], coordination [13], and task-parallel [12, 16, 18, 28, 47, 78] languages provide primitives that share core features of asynchronous dataflow programming and *Ripple*. For instance, MPI implements explicit asynchronous communication while Cilk supports asynchronous thread launch. Galois supports iterative execution, `go` offers first-class FIFO `channels`, and Habanero-Java (HJ) provides its own `async` and data-driven tasks.

So, what sets* Ripple *apart? A parallel language should enable programmers to express the kind of parallelism efficiently supported by the target architecture. For instance, CUDA forces programmers to write data-parallel programs, which yields high parallelism on GPUs. Likewise, *Ripple*'s asynchronous iterators express exactly the parallel semantics supported by SDAs. As a result, *Ripple* programs map efficiently to dataflow hardware by construction. This is the key to *Ripple*'s benefits: high pipeline parallelism and small programs, and no software runtime. Other languages do not map efficiently because they target conventional multicores or distributed systems, and they consequently express parallelism that does not map one-to-one to SDA hardware.

`go` and Cilk are languages that may impose too little or too much structure in task communication, respectively. In particular, `go` lacks structure in `channel` manipulation to effectively represent SDA queues; `channels` can be created and destroyed arbitrarily, drained imperatively, and passed by reference. SDA queues always accept and dispatch tokens automatically, which requires only a simple primitive to `push` data to a queue; the rest of its functionality is automatic in *Ripple*. On the other hand, Cilk imposes too much structure in task communication, requiring parent-child task joining. In *Ripple*, tasks can communicate arbitrarily, which maximizes asynchrony.

In addition, Galois and HJ support parallel execution models unfit for SDAs. Galois iterators support speculative, transactional worklist parallelism, where worklist data structures can be programmatically defined. HJ supports data-driven tasks, but at the wrong granularity; the programmer must stitch together all dataflow behavior explicitly with futures and `async-await` statements for each computational step. By contrast, *Ripple* features pipeline-parallel, asynchronous iterators amenable to dataflow execution on SDAs. *Ripple* programmers only need to specify coarse-grained asynchronous communication, and *Ripple* relies on the compiler to extract fine-grained dataflow parallelism, obviating the need for a `future` for each value.

Finally, due to the mismatch between language and architecture, these languages require complex software runtime systems to effectively extract parallelism. *Ripple* does not need one because it directly maps asynchronous dataflow parallelism to SDA hardware exposed by *Ripple*'s ISA.

9.3 SDAs for irregular workloads

Few SDAs directly optimize for irregular code through hardware changes. UE-CGRA [81] identified underutilization with irregular code on spatial hardware. However, they apply VLSI techniques to accelerate only inner loops. Several SDAs optimize sparse tensor code via specialized tensor iteration [23, 69, 73] or efficient near-data spatial computation [40]. However, these SDAs need extra hardware and data-parallel expression of sparse computation (e.g., `parfor`); this diverges from *Ripple*'s approach and goals (as discussed with data-parallel SDAs in Sec. 9.1). Fifer [55] is a high-performance SDA that targets decoupled irregular code [56] to hide long-latency memory operations through aggressive pipelining. However, Fifer does not support arbitrary cyclic dependencies. *Ripple* and Fifer also differ in their approaches and goals: Fifer adds massive task queues between pipeline stages that ensures ample work to hide memory latency. By contrast, *Ripple* solves the abstraction inversion problem to make better use of existing hardware queues, without modification.

10 Conclusion

This paper presents *Ripple*, a language and spatial dataflow architecture that implements the asynchronous dataflow programming model. *Ripple* explicitly encodes and maps high-level asynchrony, pipelining, and queueing semantics in irregular computation to an efficient dataflow representation. *Ripple* eliminates control-flow and memory ordering bottlenecks that plague DFGs extracted from sequential code by preserving dataflow semantics down the hardware-software stack. On important irregular workloads, and over a state-of-the-art SDA, *Ripple* improves performance by gmean 3×, reduces DFG size by nearly half, improves IPC, and slashes dynamic instructions.

A Appendix

Fig. A1 lists reference C and *Ripple* code for two workloads we evaluate, *Topological sort* (ts) and *Stencil* (st), that demonstrate how *Ripple* expresses different kinds of parallelism. *Ripple*'s implementation of ts exploits dynamic task parallelism, where the DAG task graph is discovered and processed simultaneously (in `Init` and `Sort`, respectively). *Ripple*'s st implementation constructs a coarse-grained pipeline that exploits red-black coloring of the stencil grid. Rows of red points and black points are computed concurrently, but staggered for correctness using `push`.

```
void topsort(u32 N, u32 *sorted,
  u32 *inOfs, u32 *inNbrs, // incoming
  u32 *outOfs, u32 *outNbrs, // outgoing
  u32 *inDegrees, u32 *queue):
 u32 head = 0, tail = 0, sI = 0
 void push(v): queue[tail++] = v
 u32 pop(): return queue[head++]
 bool empty(): return head == tail
 for (v = 0..N): // find 0-inDeg nodes
   u32 inDeg = inOfs[v+1] - inOfs[v]
   if (!inDeg): push(v)
   else: inDegrees[v] = inDeg
 while (!empty()): // sort
   u32 v = pop(), sorted[sI++] = v
   for (i = outOfs[v]..outOfs[v+1]):
     u32 out = outNbrs[i]
     if (!--inDegrees[out]):
       push(nb)
```

```
def topsort(u32 N, u32 *sorted,
  u32 *inOfs, u32 *inNbrs, // incoming
  u32 *outOfs, u32 *outNbrs, // outgoing
  i32 *inDegrees):
async Init(): // find 0-inDeg nodes
  for (v = 0..N):
    u32 inDeg = inOfs[v+1] - inOfs[v]
    // Atomic add to inDegrees allows
    // Init to discover work *and* Sort
    // to process tasks concurrently.
    atomic<i32*>(&inDegrees[v]):
      inDegrees[v] += inDeg;
      if (!--inDegrees[v]):
        Sort.push(v)
async Sort(u32 v) ind:
  Record.push(v)
  for (i = outOfs[v]..outOfs[v+1]):
    u32 out = outNbrs[i]
    atomic<i32*>(&inDegrees[out]):
      if (!--inDegrees[out]):
        Sort.push(out)
async Record(u32 v):
  static u32 sI = 0
  sorted[sI++] = v
```

```
void stencil(u32 n, u32 steps
  f32 A[n][n], f32 B[n][n], f32 α):
  for (0..steps):
    // Memory ordering on A and B
    // spans the entire loop nest.
    for (i = 1..n-1):
      for (j = 1..n-1): // compute B from A
        B[i][j] = α * (A[i][j] + A[i-1][j] +
          A[i+1][j] + A[i][j-1] + A[i][i+1)
    for (i = 1..n-1):
      for (j = 1..n-1): // compute A from B
        A[i][j] = α * (B[i][j] + B[i-1][j] +
          B[i+1][j] + B[i][j-1] + B[i][i+1)
```

```
const DEPTH = 3 // 5-point pattern spans 3 rows
def stencil(u32 n, u32 steps, f32 A[n][n], f32 α):
  async GenRedRows():
    for (0..steps*2):
      for (redRow = 1..n-1):
        ProcessRed.push(redRow)
  async ProcessRed(u32 i):
    // Compute red cells of row i
    for (j = ((i%2)+1)..n-1; j += 2):
      f32 sum = α * (A[i][j] + A[i-1][j] +
        A[i+1][j] + A[i][j-1] + A[i][j+1])
      WriteRed.push(i, j, sum)
  async WriteRed(u32 i, u32 j, f32 sum):
    // Writes and reads are decoupled to remove
    // unnecessary memory ordering. Instead of
    // atomics, push coordinates the pipeline.
    A[i][j] = sum;
    GenBlackRow.push(i, j)
  async GenBlackRow(u32 redRow, u32 col):
    // At the start of a red row (col <= 2),
    // a black row DEPTH=3 rows behind is ready.
    if (col <= 2):
      u32 blackRow = (redRow - DEPTH + n) % n
      ProcessBlack.push(blackRow)
  async ProcessBlack(u32 i):
    // Compute black cells of row i
    for (j = (!(i%2)+1)..n-1; j += 2):
      f32 sum = α * (A[i][j] + A[i-1][j] +
        A[i+1][j] + A[i][j-1] + A[i][j+1])
      WriteBlack.push(i, j, sum)
  async WriteBlack(u32 i, u32 j, f32 sum):
    A[i][j] = sum;
```

Fig. A1. Abridged reference C and *Ripple* code for *Topological sort* (left) and *Stencil* (right).

Acknowledgments

We thank Nathan Serafin, Nikhil Agarwal, Mitchell Fream, our anonymous reviewers, and our shepard for their time and feedback. This work was generously funded by NSF grants CCF-2403144

and CCF-1845986. Souradip Ghosh was supported by the U.S. Department of Energy Computational Science Graduate Fellowship (DESC0022158).

References

[1] S. V. Adve and M. Hill. 1990. Weak Ordering - A New Definition. In *International Symposium on Computer Architecture*.

[2] A. Aho, R. Sethi, and J. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

[3] Arvind, David E. Culler, and Kattamuri Ekanadham. 1988. *The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures*. Technical Report. MIT. Memo 278.

[4] Arvind, Kim P Gostelow, and Wil Plouffe. 1978. *The (preliminary) Id report: an asynchronous programming language and computing machine (revised)*. Technical Report. University of California, Irvine (UCI). https://escholarship.org/uc/item/0rr7573w

[5] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (oct 1989), 598–632. https://doi.org/10.1145/69558.69562

[6] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner (Eds.). 2013. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*. Contemporary Mathematics, Vol. 588. American Mathematical Society. http://dblp.uni-trier.de/db/conf/dimacs/dimacs2012.html

[7] Mahesh Balasubramanian and Aviral Shrivastava. 2022. PathSeeker: a fast mapping algorithm for CGRAs. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 268–273.

[8] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: A Systematic Framework for Heterogeneous CGRA Realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 918–932. https://doi.org/10.1145/3503222.3507772

[9] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. 1991. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, Berlin, Heidelberg, 538–568.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[11] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (Washington, DC, USA) *(HPDC '17)*. Association for Computing Machinery, New York, NY, USA, 93–104. https://doi.org/10.1145/3078597.3078616

[12] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.* 30, 8 (aug 1995), 207–216. https://doi.org/10.1145/209937.209958

[13] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. 2010. Concurrent Collections. *Sci. Program.* 18, 3–4 (Aug. 2010), 203–217. https://doi.org/10.1155/2010/521797

[14] Mihai Budiu. 2003. *Spatial Computation*. Ph. D. Dissertation. Carnegie Mellon University. https://apps.dtic.mil/sti/tr/pdf/ADA461132.pdf

[15] Mihai Budiu and Seth Copen Goldstein. 2002. *Pegasus: An Efficient Intermediate Representation*. Technical Report CMU-CS-02-107. Carnegie Mellon University.

[16] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark) *(PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 51–61. https://doi.org/10.1145/2093157.2093165

[17] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (aug 2007), 291–312. https://doi.org/10.1177/1094342007078442

[18] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[19] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. 2013. Integrating Asynchronous Task Parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. 712–725. https://doi.org/10.1109/IPDPS.2013.78

[20] Gabor Csardi and Tamas Nepusz. 2006. The igraph software package for complex network research. *InterJournal* Complex Systems (2006), 1695. https://igraph.org

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[22] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. *PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators.* IEEE Press, 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[23] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 924–939.

[24] William J. Dally and Andrew A. Chien. 1988. Object-oriented concurrent programming in CST. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications - Architecture, Software, Computer Systems, and General Issues, C^3P, Pasadena, California, USA, January 19-20, 1988*, Geoffrey C. Fox (Ed.). ACM, 434–439. https://doi.org/10.1145/62297.62346

[25] W. J. Dally and C. L. Seitz. 1987. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Computers* 36, 5 (1987).

[26] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V. Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. 2019. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT) (PACT '19).* 15–28. https://doi.org/10.1109/PACT.2019.00010

[27] Jack B Dennis and David P Misunas. 1975. A preliminary architecture for a basic data-flow processor. In *ISCA.*

[28] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

[29] P. Erdös and A. Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae Debrecen* 6 (1959), 290.

[30] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (nov 1976), 624–633. https://doi.org/10.1145/360363.360369

[31] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2014. The nesC language: a holistic approach to networked embedded systems. *SIGPLAN Not.* 49, 4S (jul 2014), 41–51. https://doi.org/10.1145/2641638.2641652

[32] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: an ultra-low-power, energy-minimal CGRA-generation framework and architecture. In *ISCA.*

[33] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2023. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (<conf-loc>, <city>Chicago</city>, <state>Illinois</state>, <country>USA</country>, </conf-loc>) *(MICRO '22).* IEEE Press, 546–564. https://doi.org/10.1109/MICRO56248.2022.00046

[34] 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).

[35] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012).

[36] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11).* Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1941553.1941557

[37] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. 1993. Performance studies of Id on the Monsoon dataflow system. *J. Parallel Distrib. Comput.* 18, 3 (jul 1993), 273–300. https://doi.org/10.1006/jpdc.1993.1065

[38] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Trans. Parallel Distrib. Syst.* 33, 6 (June 2022), 1303–1320. https://doi.org/10.1109/TPDS.2021.3104255

[39] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).

[40] Rohan Juneja, Pranav Dangi, Thilini Kaushalya Bandara, Zhaoying Li, Tulika Mitra, and Li shiuan Peh. 2025. Nexus Machine: An Active Message Inspired Reconfigurable Architecture for Irregular Workloads. arXiv:2502.12380 [cs.AR] https://arxiv.org/abs/2502.12380

[41] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *DAC.*

[42] Ken Kennedy and John R. Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[43] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. 2017. Moonwalk: NRE Optimization in ASIC Clouds. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 511–526. https://doi.org/10.1145/3037697.3037749

[44] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[45] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* 4, 35 (2019), 1244. https://doi.org/10.21105/joss.01244

[46] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. 2009. How much parallelism is there in irregular applications?. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Raleigh, NC, USA) *(PPoPP '09)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1504176.1504181

[47] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/1250734.1250759

[48] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[49] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[50] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[51] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Trans. Embed. Comput. Syst.* 20, 4, Article 36 (May 2021), 27 pages. https://doi.org/10.1145/3448128

[52] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: a new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (Catalina Island, CA) *(UAI'10)*. AUAI Press, Arlington, Virginia, USA, 340–349.

[53] Frank McSherry. 2005. A uniform approach to accelerated PageRank computation. In *Proceedings of the 14th International Conference on World Wide Web* (Chiba, Japan) *(WWW '05)*. Association for Computing Machinery, New York, NY, USA, 575–582. https://doi.org/10.1145/1060745.1060829

[54] Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. 2006. Tartan: evaluating spatial computation for whole program execution. *ACM SIGARCH Computer Architecture News* 34, 5 (2006).

[55] Quan M Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO*.

[56] Quan M. Nguyen and Daniel Sanchez. 2023. Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1262–1274. https://doi.org/10.1109/HPCA56546.2023.10071026

[57] R.S. Nikhil. 1991. ID Reference Manual. Memo 284-2.

[58] Rishiyur S Nikhil et al. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on computers* (1990).

[59] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *PACT 27*.

[60] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *ISCA 44*.

[61] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *ACM SIGARCH Computer Architecture News*, Vol. 43.

[62] 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. *ACM SIGPLAN Notices* 48, 6 (2013).

[63] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. 2013. Triggered instructions: a control paradigm for spatially-programmed architectures. *ACM SIGARCH Computer Architecture News* 41, 3 (2013).

[64] Andrew Petersen, Martha Mercaldi, Steve Swanson, Andrew Putnam, Andrew Schwerin, Mark Oskin, and Susan Eggers. 2006. Reducing control overhead in dataflow architectures. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 182–191.

[65] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 12–25. https://doi.org/10.1145/1993498.1993501

[66] Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Karen Li, et al. 2024. SambaNova SN40L: Scaling the AI Memory Wall with Dataflow and Composition of Experts. In *57th IEEE/ACM International Symposium on Microarchitecture (MICRO-57)*.

[67] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *ISCA 44*.

[68] V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo, and J.M. Prellezo. 2001. The Adaptive Bubble Router. *J. Parallel Distrib. Comput.* 61, 9 (sep 2001), 1180–1208. https://doi.org/10.1006/jpdc.2001.1746

[69] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. arXiv:2104.12760 [cs.AR]

[70] Alexander C. Rucker, Shiv Sundram, Coleman Smith, Matthew Vilim, Raghu Prabhakar, Fredrik Kjolstad, and Kunle Olukotun. 2024. Revet: A Language and Compiler for Dataflow Threads . In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–14. https://doi.org/10.1109/HPCA57654.2024.00016

[71] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA 30*.

[72] Karu Sankaralingam, Tony Nowatzki, Greg Wright, Poly Palamuttam, Jitu Khare, Vinay Gangadhar, and Preyas Shah. 2021. Mozart: Designing for Software Maturity and the Next Paradigm for Chip Architectures. In *IEEE Hot Chips 33 Symposium, HCS 2021, Palo Alto, CA, USA, August 22-24, 2021*. IEEE, 1–20. https://doi.org/10.1109/HCS52781.2021.9567306

[73] Nathan Serafin, Souradip Ghosh, Harsh Desai, Nathan Beckmann, and Brandon Lucia. 2023. Pipestitch: An energy-minimal dataflow architecture with lightweight threads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (<conf-loc>, <city>Toronto</city>, <state>ON</state>, <country>Canada</country>, </conf-loc>) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1409–1422. https://doi.org/10.1145/3613424.3614283

[74] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1996. *MPI: The Complete Reference*. The MIT Press.

[75] Svend Haugaard Sorensen. 2013. Linux call graph (version 3.7.10).

[76] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *MICRO 36*.

[77] Cheng Tan, Nicolas Bohm Agostini, Tong Geng, Chenhao Xie, Jiajia Li, Ang Li, Kevin J. Barker, and Antonino Tumeo. 2022. DRIPS: Dynamic Rebalancing of Pipelined Streaming Applications on CGRAs. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 304–316. https://doi.org/10.1109/HPCA53966.2022.00030

[78] Sagnak Tasirlar and Vivek Sarkar. 2011. Data-Driven Tasks and Their Implementation. In *2011 International Conference on Parallel Processing (ICPP '11)*. 652–661. https://doi.org/10.1109/ICPP.2011.87

[79] W. Thies, M. Karczmarek, and S. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *CC*.

[80] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. 2005. Teleport messaging for distributed stream programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/1065944.1065975

[81] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-elastic cgras for irregular loop specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 412–425.

[82] Naveen Vedula, Arrvindh Shriraman, Snehasish Kumar, and William N Sumner. 2018. NACHOS: Software-Driven Hardware-Assisted Memory Disambiguation for Accelerators. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 710–723. https://doi.org/10.1109/HPCA.2018.00066

[83] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. 2021. Aurochs: An architecture for dataflow threads. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 402–415.
[84] Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. *ACM SIGARCH computer architecture news* 42, 3 (2014).
[85] Dani Voitsechov, Oron Port, and Yoav Etsion. 2018. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. In *MICRO 51*.
[86] E. Waingold et al. 1997. Baring It All to Software: Raw Machines. In *IEEE Computer*.
[87] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *HPCA*.
[88] Joyce Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. 2015. Scalable Data-driven PageRank: Algorithms, System Issues, and Lessons Learned. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.
[89] Tomofumi Yuki and Louis-Noel Pouchet. 2016. PolyBench 4.2.1: The polyhedral benchmark suite.
[90] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1041–1054. https://doi.org/10.1109/ISCA52012.2021.00085

Received 2024-11-15; accepted 2025-03-06