RipTide: A programmable, energy-minimal dataflow compiler and architecture

ABSTRACT

Emerging sensing applications create an unprecedented need for extreme energy efficiency in general-purpose processors. To achieve useful multi-year deployments on a small battery or energy harvester, these applications must avoid off-device communication and instead process most data locally. Recent work has proven ultra-low-power (ULP) coarse-grained reconfigurable arrays (CGRAs) as a promising architecture for this domain. Unfortunately, prior ULP designs require hand-written assembly and can only accelerate a fraction of program execution, and other CGRAs target high-performance applications that face different goals and challenges.

We present RIPTIDE, a co-designed compiler and CGRA architecture targeting high programmability and extreme energy efficiency. RIPTIDE provides a rich set of control-flow operators, letting it support arbitrary control flow and memory access on the CGRA fabric. RIPTIDE implements these primitives without tagged tokens to save energy; this requires careful ordering analysis in the compiler to guarantee correctness. RIPTIDE further saves energy and area by offloading most control operations into its programmable on-chip network, where they can re-use existing network switches. RIPTIDE's compiler is implemented in LLVM and its hardware in an industrial sub-28nm process. RIPTIDE compiles applications written in C while saving 25% energy vs. the state-of-the-art ULP CGRA and 6.6× energy vs. a von Neumann core.

1. INTRODUCTION

RECENT advances in machine learning, sensor devices, and embedded systems open the door to a wide range of sensing applications, such as civil-infrastructure or wilderness monitoring, public safety and security, medical devices, and chip-scale satellites [75]. To achieve long (e.g., 5+ year) deployment lifetimes, these applications rely on on-device processing to limit off-device communication. Computing at the extreme edge calls for ultra-low-power (<1 mW), *extremely energy-efficient*, and *general-purpose* processing [23].

Why general-purpose? The need for extreme energy efficiency suggests a role for application-specific integrated circuits (ASICs), but ASICs come with several major disadvantages. Computations in smart sensing applications are diverse, spanning deep learning, signal processing, compression, encoding, decryption, planning, control, and symbolic reasoning [22]. Only a general-purpose solution can support all of these, as it is infeasible to build an ASIC for every conceivable task [36, 68]. Moreover, the rapid pace of change in these applications (e.g., due to new machine learning algorithms [34]) puts specialized hardware at risk of premature obsolescence, especially in a multi-year deployment [68]. Finally, by targeting all computations, general-purpose designs can achieve much greater scale than specialized designs — perhaps trillions of devices [71]. Scale reduces device cost, makes advanced manufacturing nodes economically viable, and mitigates carbon footprint [28].

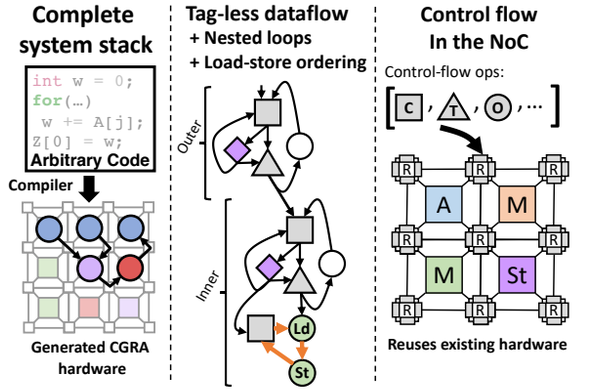

Figure 1: RIPTIDE is a co-designed compiler and CGRA microarchitecture that executes programs written in a *high-level language* with *minimal energy* and high performance. RIPTIDE *introduces new control-flow primitives* to support common programming idioms, like deeply nested loops and irregular memory accesses, while minimizing overhead. Finally, RIPTIDE implements its control-flow *in the NoC* to increase utilization and ease compilation.

Unfortunately, traditional programmable cores are very inefficient, typically using only 5% to 10% of their energy on useful work [21, 24, 33]. Architects' challenge is thus to reconcile generality and efficiency.

CGRAs are both programmable and efficient! Recent work has shown that coarse-grained reconfigurable arrays (CGRAs) can achieve energy efficiency competitive with ASICs while remaining programmable by software [21, 57, 84]. As shown in Fig. 1, a CGRA [4, 9, 12, 16, 19, 25, 26, 35, 43, 47–51, 55, 60, 62, 64–67, 70, 73, 74, 80, 81, 84, 85] is an array of processing elements (PEs) connected by an on-chip network (NoC). CGRAs are programmed by mapping a computation's control and dataflow onto the array, i.e., by assigning operations to PEs and configuring the NoC to route values between dependent operations. A CGRA's efficiency derives from avoiding overheads intrinsic to von Neumann architectures, specifically instruction fetch/control and data buffering in a centralized register file.

In the context of ultra-low-power sensing applications, SNAFU [21] is a CGRA framework designed from the ground up to minimize energy, in contrast to prior, performance-focused CGRAs (Sec. 2). SNAFU CGRAs reduce energy by 5× vs. ultra-low-power von Neumann cores, and they come within 3× of ASIC energy efficiency.

What's the problem? Amdahl's Law tells us that, to achieve significant end-to-end benefits, CGRAs must benefit the vast majority of program execution. This means CGRAs must provide a complete compiler and hardware stack that goes from application code to an efficient CGRA configuration. Unfortunately, prior CGRAs struggle to support common programming idioms efficiently, leaving significant energy savings on the table.

On the hardware side, many prior CGRAs only support simple, regular control flow, such as inner loops with streaming

memory accesses and no data-dependent control [21, 55, 64]. To support complex control flow, other CGRAs employ expensive hardware mechanisms, e.g., associative tags to distinguish loop iterations, large buffers to avoid deadlock, and dynamic NoC routing [52, 59, 72, 80]. In either case, the end result is wasted energy: from the extra instructions needed to implement control flow unsupported by the CGRA fabric, or from inefficiency in the CGRA microarchitecture itself.

On the compiler side, mapping large computations onto a CGRA fabric is perennial challenge. Heuristic compilation methods often fail to find a valid mapping [53, 63], and optimization-based methods lead to prohibitively long compilation times [8, 53]. Indeed, one reason that many CGRAs support only regular control is to limit the number of operations that must be mapped during compilation. Control flow can significantly increase the size of a computation's dataflow graph, increasing compilation time or preventing a computation from mapping successfully. To avoid these issues, some CGRAs (including SNAFU) require hand-coded assembly, raising a large barrier to adoption [21, 55, 87].

RipTide's solution. RIPTIDE is a co-designed CGRA compiler and microarchitecture that supports arbitrary control flow and memory access patterns without expensive hardware mechanisms. RIPTIDE targets emerging, highly energy-constrained applications and is designed from the ground up to minimize energy. RIPTIDE is easy to program: its compiler supports arbitrary nested control and loops as well as aliasing memory accesses. To save energy, RIPTIDE adopts a *steering* control paradigm [6, 18, 72], in which values are only routed to where they are actually needed (unlike predication- and selection-based control common in prior CGRAs [21, 66]). To support arbitrary nested control without tags, RIPTIDE introduces new control-flow primitives, such as the *carry gate*, which selects between tokens from inner and outer loops. To minimize operation count and ease compilation, RIPTIDE introduces new operations for common programming idioms, such as its *stream generator* that generates an affine sequence for, e.g., simple loops or streaming memory accesses.

RIPTIDE implements the above features efficiently in the both the compiler and hardware (Fig. 1). RIPTIDE compiles programs from a high-level language (currently, C) and employs novel analyses to safely parallelize operations. We observe that, with steering control flow and no program counter, conventional transitive reduction analysis fails to enforce all memory orderings, and we introduce *path-sensitive transitive reduction* to infer orderings correctly. RIPTIDE implements arbitrary control flow without associative tags by enforcing strict ordering among values, leveraging its new primitives like the carry gate for nested loops. RIPTIDE supports common idioms like affine generators directly in hardware, and its compiler recognizes these idioms in program code and maps them onto a single PE. Finally, RIPTIDE implements its new control flow primitives without wasting energy or PEs by *offloading control flow to the on-chip network*. The insight is that a NoC switch already contains essentially all of the logic needed for steering control flow, and with a few trivial additions it can implement a wide range of control primitives. Mapping control-flow into the NoC frees PEs for arithmetic and memory operations, so that RIPTIDE can support deeply nested loops with complex control flow on a small CGRA.

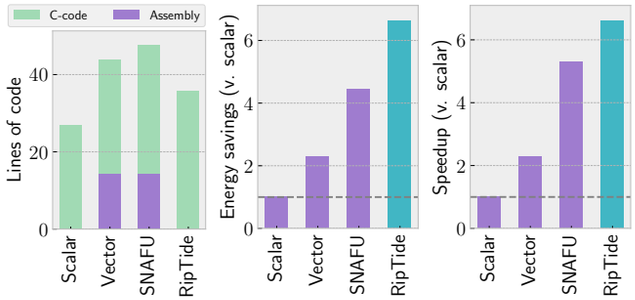

Figure 2: RIPTIDE improves energy-efficiency and performance over the state-of-the-art, while compiling programs from high-level C (vs. vector assembly in SNAFU).

Contributions. This paper contributes the following:

- ***Instruction set architecture:*** We co-design RIPTIDE's compiler and CGRA microarchitecture to provide a rich operation set that supports arbitrary control flow and irregular memory accesses with minimum execution energy. We identify common programming idioms and introduce new primitives to support them in fewer operations.
- ***Compiler:*** RIPTIDE compiles programs from high-level C code to an efficient CGRA configuration. RIPTIDE identifies and enforces all control-flow and memory orderings, introducing *path-sensitive transitive reduction* to safely prune unnecessary memory orderings.
- ***Hardware:*** RIPTIDE implements its operation set efficiently in hardware. It incorporates numerous techniques to minimize energy, including steering control flow and tagless dataflow firing. RIPTIDE *offloads control flow to the on-chip network*, freeing PEs for other useful work.
- ***Broader implications on architecture:*** We perform an in-depth case study of dense matrix-matrix multiplication, comparing RIPTIDE to an ASIC implemented in the same design flow. RIPTIDE is competitive on energy and performance, but consumes significantly more area than the ASIC. ASICs thus offer a cost advantage over CGRAs, but this advantage disappears in SoC designs with a large number of ASIC blocks. Given the large advantages gained by software programmability, we argue that energy-minimal CGRAs like RIPTIDE have a compelling edge over ASICs for the majority of computations.

Summary of results. We implement a complete RIPTIDE system in RTL and synthesize it in an industrial sub-28nm FinFET process with compiled memories. Across ten benchmarks, RIPTIDE reduces energy by 25% vs. SNAFU, the state-of-the-art programmable design, and improves performance by 17% (Fig. 2). At nominal voltage, RIPTIDE achieves 141MOPs/mW (including main memory) at 50MHz and takes $\approx$0.5mm^2. Compared to equivalent ASICs for `dmm`, `sort`, and `fft`, RIPTIDE consumes just 2.1$\times$ more energy. RIPTIDE achieves these benefits on software written in C, cf. hand-coded vector assembly in SNAFU.

Road map. Sec. 2 covers background, and Sec. 3 gives an overview of RIPTIDE. Secs. 4, 5, and 6 present RIPTIDE's architecture, compiler, and microarchitecture, respectively. Secs. 7 and 8 evaluate RIPTIDE. Sec. 9 concludes by discussing RIPTIDE's broader implications.

2. BACKGROUND

RIPTIDE is motivated by emerging energy-constrained sensing applications. CGRAs avoid inefficiencies of von Neumann cores, but prior CGRAs have limited programmability or efficiency, which are both significant affected by the CGRA's control-flow paradigm. This section reviews prior work to set the stage for RIPTIDE's contributions.

2.1 On-device processing at the extreme edge

Energy constraints. Applications such as infrastructure and wilderness monitoring, public safety, and tiny satellites collect sensor data continuously in long-lived deployments [17, 41]. Maintenance is impractical in these applications because devices are physically inaccessible, requiring years of maintenance-free operation on a small battery or energy harvester. These applications impose tight energy constraints; e.g., a AA battery over a five-year deployment provides an amortized power budget of only 65 µW. Devices with operating power over this limit must duty cycle, lowering application value.

Why on-device processing? Off-device communication takes orders-of-magnitude more energy than computing locally, creating a strong incentive for on-device processing of sensor data [23]. To reap the benefit of local processing, computation must consume minimum energy. Computing must also be fast to meet performance requirements.

ASICs are efficient, but too limited and expensive. The rapid pace of change in sensing applications rules out the application-specific integrated circuit (ASIC) as a solution. ASIC efficiency comes with a high cost for design, verification, and manufacturing, and only makes sense for stable workloads. ASICs' lack of programmability makes them likely to fall behind applications during a multi-year deployment. Applications then have only bad options: sacrifice application features for energy-efficiency on an obsolete ASIC, or sacrifice energy efficiency (and lifetime) to keep features up-to-date [69].

Programmable cores are inefficient. Unfortunately, traditional von Neumann cores are very energy-inefficient, burning most energy on instruction control and data movement. Instruction fetch, pipeline control, and register-file access easily consumes 90% of energy.

This inefficiency is not fundamental. Real programs have fine-grained instruction and data locality that von Neumann cores exploit poorly, as they time-multiplex instructions on an shared execution pipeline and communicate values through a register file. Exploiting this locality is the key to achieving energy-efficient programmability.

2.2 Coarse-grained reconfigurable arrays

A CGRA architecture [4, 9, 12, 16, 19, 25, 26, 35, 43, 47–51, 55, 60, 62, 64–67, 70, 73, 74, 80, 81, 84, 85] is a spatial array of processing elements (PEs) connected by an on-chip interconnect (NoC). A PE in a CGRA consumes inputs and produces outputs consumed by another PE, forming a pipeline corresponding to program dataflow. CGRA efficiency derives from avoiding control and data-movement overheads. A CGRA reduces instruction overheads by mapping operations to a PE, avoiding the need for instruction fetch and decode and simplifying control. A CGRA mitigates data movement overheads by avoiding large register files, instead moving operands through a NoC directly from producer PE to consumer PEs.

CGRA design space. A wide variety of CGRA architectures target different domains. CGRAs exist as standalone cores [48, 66, 72, 82], co-processors [10, 11, 25, 27, 30, 43, 56, 73], components of a processor pipeline [26, 38, 39] or memory hierarchy [40], or as accelerators [12, 14–16, 50, 51, 55, 62, 64, 65, 70, 79, 80, 85, 86]. These contexts expose a wide range of hardware design choices, including PE operation set, PE complexity, and NoC.

A CGRA's PEs typically include functional units for arithmetic, logic, and memory access, or specialized functionality [14–16, 21, 65, 79, 84, 86]. PEs may be homogeneous or heterogeneous; the latter is more area- and energy-efficient, but creates a combinatorically large design space [4].

Scheduling operations. For regular applications, a compiler can time and route operations, in a fabric often called a *systolic array*. For irregular applications, CGRAs often adopt *dynamic dataflow firing*, where an operation issues when its inputs arrive. Matching inputs is challenging, however, especially in high-performance designs with out-of-order execution [52, 59, 60, 72]. Out-of-order token matching requires tags and associative memories, which must be large enough to avoid deadlock, incurring a high energy cost. To mitigate these overheads, Revel [85] combines systolic and tagged dataflow in a single, heterogeneous fabric. *Ordered dataflow* entirely avoids tag matching by disallowing out-of-order execution so that inputs are always matched on arrival. But such designs must guarantee that re-orderings can never happen, which is particularly challenging in RIPTIDE's steering control paradigm (see below).

Performance vs. energy. Like nearly all processors, prior CGRAs primarily maximize performance under an area or power budget. As such, key CGRA metrics have been PE utilization and initiation interval (i.e., cycles between loop iterations). In contrast, RIPTIDE focuses on energy, with a CGRA's inherent parallelism yielding performance as a secondary benefit. Many design choices in RIPTIDE only make sense in this context.

SNAFU [21] is a recent framework for generating *energy-minimal* CGRAs. To minimize switching activity and buffering in the fabric, SNAFU maps a single operation to each PE and statically routes values between PEs. SNAFU also uses ordered dataflow without tags. These design choices reduce energy vs. performance-focused CGRAs. RIPTIDE targets the same domain, and uses SNAFU as its baseline design.

CGRA compilation. Compiling to a CGRA fabric is challenging. Similar to hardware synthesis, the compiler must find a layout of operations that fits within fabric resources with valid routes between all producers and consumers. In performance-focused CGRAs, the compiler must also reason about timing to maximize utilization and minimize initiation interval. With this vast search space, optimization-based methods often do not converge in a reasonable time [54, 58]. Most CGRA compilers use heuristics [4, 42, 54, 58, 61, 74, 83, 84] that can fail or produce poor mappings. Recent work proposed graph convolutional networks as a solution [46].

RIPTIDE's compiler uses an integer-linear program (ILP) solver [29] to find a mapping that minimizes routing distance.

Mapping is tractable because its compiler need not reason about timing or utilization and because RIPTIDE offloads control flow to its NoC, which does not consume scarce PEs.

2.3 Control flow in dataflow architectures

There are three competing control-flow models for dataflow execution: predication, selection (ϕ), and steering (ϕ^{-1}). Each has benefits and drawbacks. In an energy-constrained context, we observe steering is best because it avoids routing values to the not-taken branch paths.

Predication routes values unnecessarily. Predication is popular, especially in GPU and vector architectures [20, 31], converting conditional code to straightline code to simplify execution. In predication, only one side of a branch fully executes while the other side partially executes, passing through results from the enabled side to downstream consumers. Predication simplifies control flow in CGRAs because tokens arrive on every path, simplifying operand ordering. But predication has a performance and energy cost because values flow unnecessarily through the not-taken path. SNAFU uses predication for control flow and supports only simple, affine loops (disallowing back-edges generally).

Selection (ϕ) burns energy on paths not taken. Selection executes both sides of a branch fully, sending results to a mux that chooses between results using the branch decider. Selection maximizes performance because the branch condition and each side of the branch execute speculatively in parallel. Selection is common in recent CGRAs that focus on performance. However, selection wastes energy by throwing away work.

Steering (ϕ^{-1}) is most energy efficient. Steering was proposed in the original dataflow paper [18] and has been used notably in a few dataflow architectures [5,27,48,56,72]. Steering routes values to only the taken path of a branch based on the branch's decider. Steering serializes execution of the taken path on the branch decider, but avoids executing any operations on the not-taken path. RIPTIDE implements steering to minimize energy, as steering never fires unneeded operations. However, since loop iterations may take different paths through control flow, steering introduces the risk of token reordering, which would lead to incorrect results in RIPTIDE's ordered dataflow model. RIPTIDE guarantees correct ordering by inserting ordering operations, where necessary, in its compiler.

3. RIPTIDE OVERVIEW

RIPTIDE is a compiler and microarchitecture for ultra-low-power, energy-minimal CGRAs (Fig. 3). Its architecture is carefully co-designed across compiler and hardware to maximize flexibility while minimizing energy. RIPTIDE can compile and run programs written in high-level C with arbitrary control flow and memory access. RIPTIDE improves energy efficiency / performance by 25% / 17% vs. the state-of-the-art [21] and by 6.6× / 6.2× vs. a ULP core.

RIPTIDE's control-flow paradigm minimizes energy. RIPTIDE introduces a new control-flow paradigm, discussed in Sec. 4, that supports arbitrary programs without expensive tag-token matching hardware. Specifically, RIPTIDE adopts a steering (ϕ^{-1}) control-flow model, which minimizes energy by gating disabled paths so that values are only sent where they will be actually used. RIPTIDE introduces the *carry gate*, a new control-flow operator that supports nested loops without tags, as well as other operations to maintain memory and cross-iteration orderings efficiently.

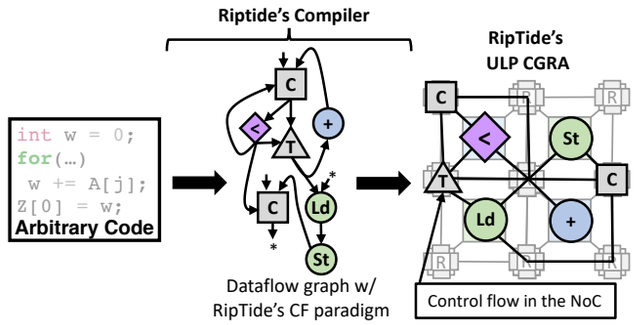

Figure 3: RIPTIDE is a compiler and CGRA microarchitecture that maps high-level code with arbitrary control flow and memory access to an energy-minimal CGRA fabric. To maximize efficiency while minimizing area, RIPTIDE implements control flow in the NoC.

RIPTIDE compiles arbitrary C-code to energy-minimal and performant CGRA configurations. RIPTIDE's compiler (Sec. 5) leverages LLVM [37] and several custom compiler passes to convert arbitrary programs to dataflow graphs schedulable onto RIPTIDE's CGRA hardware. RIPTIDE supports steering control and introduces a novel memory-ordering analysis to enforce load-store ordering with low overhead. Additionally, RIPTIDE applies several optimizations to dataflow graphs, including operation fusion (e.g., "streamifying" affine loops by fusing loop headers) to reduce operation count, improve performance, and ease mapping operations onto the fabric.

RIPTIDE's CGRA microarchitecture minimizes switching activity. RIPTIDE's CGRA microarchitecture prioritizes energy efficiency above all other metrics. It adopts and improves techniques from prior work [21] to reduce switching activity to minimize fabric energy: e.g., mapping exactly one operation per PE, firing operations without tags, and statically routing values in a bufferless NoC.

Offloading control flow to the NoC enables complex programs on small CGRA fabrics with negligible hardware overhead. RIPTIDE supports control-flow operations directly in the NoC by reusing existing NoC hardware. Control-flow operations are simple, but numerous. This means that allocating them to entire PEs is unnecessary and wasteful — programs are often unmappable if control-flow operations require PEs. Instead, RIPTIDE implements these operations directly in NoC routers by reusing existing switch crossbars, which already perform most of the required logic.

4. RIPTIDE INSTRUCTION SET

RIPTIDE provides a rich set of control-flow operators to support complex programs. Its ISA, shown in Table 1, has six categories of operators: arithmetic, multiplier, memory, control flow, synchronization, and streams. (Multiplication is split from other arithmetic because, to save area, only some PEs can perform multiplication.) We now highlight the control-flow, synchronization, and stream operators.

nzb: Table doesn't show control tokens for memory ops.

4.1 Control-flow operators

There are two fundamental control-flow operators: steer and carry. These operators are universal, i.e., sufficient to

Table 1: RIPTIDE's instruction set architecture (ISA).

Operator(s)	Category	Symbol(s)	Semantics
Basic binary ops	Arithmetic	$+, -, <<, !=$, etc.	a op b
Multiply, clip	Multiplier	$*$, `clip`	a op b
Load	Memory	`ld`	`ld` $base, idx$
Store	Memory	`st`	`st` $base, idx, val$
Select	Control Flow	`sel`	$cond$? $val0$: $val1$
Steer, carry, invariant	Control Flow	(`T` | `F`), `C`, `I`	See Fig. 4
Merge, order	Synchronization	`M`, `O`	See Fig. 4
Stream	Stream	`STR`	See Fig. 4

nzb: Revisit this. Do they cover order and merge?

implement RIPTIDE's control-flow paradigm. Operators are illustrated in Fig. 4. Whenever a value is read by an operator, it is implied that the operator waits until a valid token arrives for that value over the NoC. Tokens are buffered at the operator inputs if they are not consumed or discarded.

Steer. Steers (ϕ^{-1}) come in two flavors — True and False — and take two inputs: a decider, `D` and a data input, `A`. If `D` matches the flavor, then the gate passes `A` through; otherwise, `A` is discarded. Steers are necessary to implement conditional execution, as they gate the inputs to disabled branches.

Carry. Carry is the other fundamental control-flow operator. It represents a loop-carried dependency and takes a decider, `D`, and two data values `A` and `B`. Carry has the internal state machine shown in Fig. 4. In the *Initial* state, it waits for `A`, and then passes it through and transitions to the *Block* state. While in *Block*, if `D` is True, the operator passes through `B`. It transitions back to *Initial* when `D` is False, and begins waiting for the next `A` value (if not already buffered at the input).

nzb: Rename states to Outer and Inner for clarity?

Carry operators keep tokens ordered in loops, eliminating the need to tag tokens. By not consuming `A` while in *Block*, carry operators prevent outer loops from spawning a new inner-loop instance before the previous one has finished. (Iterations from one inner-loop may safely run in parallel, but entire instances of the inner loop may not.)

Invariant. The invariant operator is a slight variation of carry. It represents a loop invariant and can be implemented as a carry with a self-edge back to `B`. Invariants are used to generate a new loop-invariant token for each loop iteration.

4.2 Synchronization operators

Order. The order operator enforces cross-iteration ordering by making sure that tokens from different loop iterations appear in the same order, regardless of the control path taken within by each loop iteration. The operator takes three inputs: a decider, `D`, and two data inputs, `A` and `B`. Order is essentially a mux that passes through either `A` or `B`, depending on `D`. But note that only the value passed through is consumed.

Merge. The merge operator is used to enforce memory ordering by guaranteeing that multiple preceding operators have executed. It takes two inputs, `A` and `B`, and fires as soon as both arrive, passing `A` through.

4.3 Stream operators

Streams generate a sequence of data values, which are produced by evaluating an affine function across a range of inputs. These operations are used to accelerate loops governed by affine induction variables. A stream takes three inputs: `start`, `step`, and `bound`. It initially sets its internal `idx` to `start`, and then begins iterating a specified arithmetic operator `f` as `idx' = f(idx, step)`.

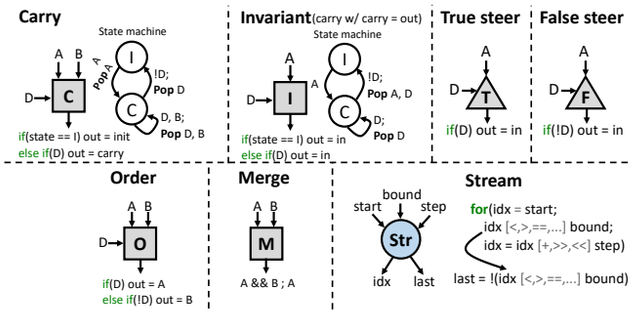

Figure 4: Semantics of new control-flow operations in RIPTIDE.

A stream operator produces two output tokens per iteration: `idx` itself, and a control signal `last`. `last` is False until `idx` reaches `bound`, whereupon it is True and the stream stops iterating. `last` is used by downstream control logic to, e.g., control a carry operator for outer loops.

5. RIPTIDE COMPILER

RIPTIDE compiles, optimizes, and maps high-level C code to RIPTIDE's CGRA fabric. Its compiler has a frontend, middle-end, and backend. The frontend compiles C to target-independent intermediate representation (IR) based on LLVM's IR. The middle-end optimizes the code in LLVM's IR, then translates to target-specific IR, represented as a dataflow graph (DFG) of the operator types from Sec. 4, including control flow and memory ordering. The backend takes target-specific IR as input and maps operators onto specific CGRA hardware units, producing a configuration bitstream directly usable for configuring the CGRA. Fig. 5 demonstrates the frontend and middle-end's compiler passes.

5.1 Memory-ordering analysis

RIPTIDE maps sequential code onto a CGRA fabric in which many operations, including memory operations, may execute in parallel. For correctness, some dependent memory operations must execute in a particular order. RIPTIDE's middle-end computes required orderings between memory operations present in the IR and adds control-flow operators to enforce those orderings. The compiler uses alias analysis to identify dependent memory operations that may access the same memory locations and may execute on the RIPTIDE fabric simultaneously, requiring ordering.

RIPTIDE computes ordering relations by analyzing memory dependences. A *memory dependence* is an ordered relation from one memory operation (source) to another (destination) that access same memory location, where the destination is reachable from the source and one (or both) are writes. An *ordering graph* (OG) is a digraph of the ordering relations required to ensure that parallel memory operations produce a result consistent with a sequential execution of the program.

Constructing an ordering graph. To build the OG, RIPTIDE uses alias analysis to identify memory operations that do not, may, or must access the same memory location (i.e., alias). RIPTIDE makes no assumptions on the alias analysis. RIPTIDE queries alias analysis for all memory operations and adds an edge to the OG for each pair that may or must alias. RIPTIDE need not consider self-dependences because repeated instances of the same memory operation are always ordered on its CGRA fabric. Fig. 6 shows a basic, unopti-

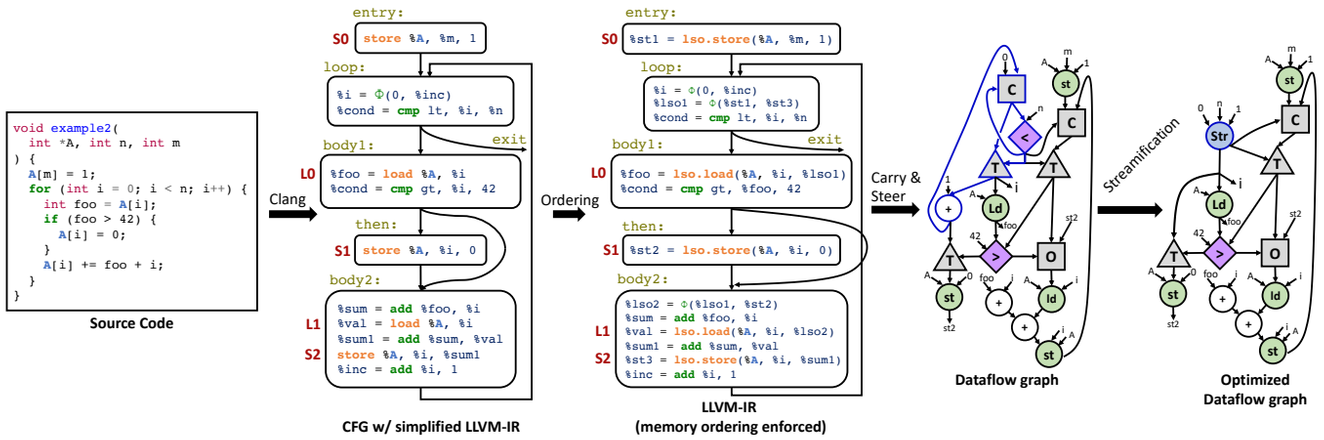

Figure 5: RIPTIDE's frontend and middle-end components. The frontend compiles C code to LLVM-IR using `clang`. The middle-end produces an optimized dataflow graph (DFG) that enforces memory ordering and RIPTIDE's control paradigm.

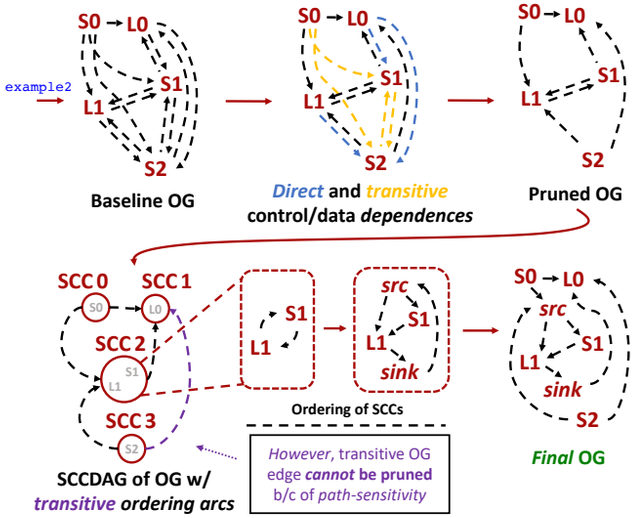

Figure 6: RIPTIDE's middle-end enforces memory ordering. For `example2`, an ordering graph (OG) that is iteratively pruned and reduced.

mized OG in the top left for an example function (`example2`).

Pruning the ordering graph. The OG as computed includes redundant ordering edges because data dependences and control dependences suffice to order some operations. A data dependence orders a pair of operations in RIPTIDE because the destination will not execute before receiving an input token produced by the source. RIPTIDE finds data dependences by walking LLVM's definition-use (def-use) chain from source to destination and removes arcs for dependent operations [77]. For instance, in `example2`'s CFG from Fig. 5, **S2** is data-dependent **L1**, so there need not be an ordering arc in the OG. This is reflected in the blue-outlined edge from **L1** to **S2** that is pruned in the OG in Fig. 6. Similarly, control dependences order some memory operations if the execution of the destination is control-dependent on the source. RIPTIDE analyzes the CFG to identify control dependences between memory operations and removes those orderings from the OG. In `example2`'s CFG from Fig. 5, the edge from **L0** to **S1** in Fig. 6 is pruned using this analysis.

Transitive memory-ordering analysis. Two dependent memory operations are transitively ordered if there is a path (of ordering arcs) in the OG from source to destination. RIPTIDE finds and eliminates edges in the OG between operations already transitively ordered by some other path in the OG corresponding to control- and data-dependence orderings.

To simplify its OG, RIPTIDE uses transitive reduction (TR) [2], which prior work deployed to simplify ordering relation graphs for parallel execution of loops [44, 45]. We apply TR to the OG, which converts a (potentially cyclic) ordering graph into an acyclic graph of strongly connected components (the SCCDAG). Traditional TR then eliminates transitive edges between SCCs, removes all edges in an SCC, and adds edges forming a simple cycle through all vertices in the SCC. We modify the algorithm in two ways to make it work for RIPTIDE's OG. First, we require the inserted cycle to add edges compatible with program order instead of being arbitrary. Second, the inserted cycle must add edges that respect proper loop nesting, avoiding edges directly from the inner to outer loop. To handle these edges, we add synthetic loop entry and exit nodes to each loop (shown as `src` and `sink` nodes at the bottom of Fig. 6). Any edge inserted that links an inner loop node to an outer loop node instead uses the inner loop's exit as its destination. Symmetrically, an edge inserted that links an outer loop node to an inner loop node has the inner loop's entry as its destination. With these two changes, the SCCDAG is usable for TR.

However, we observe that applying existing TR analysis to the OG in RIPTIDE fails to preserve required ordering operations. The problem is that a source and destination may be ordered along one (transitive) path, and ordering along another (direct) path may be removed as redundant. The transitive path's execution enforces ordering, but the direct path's execution does not, which is incorrect. This problem arises due to RIPTIDE's steering control and lack of a program counter to order memory operations. In Fig. 6, we show a scenario where path-sensitivity is critical. The transitive edge in the SCCDAG of the OG between **SCC3** (**S2**) and **SCC1** (**L0**) cannot be eliminated in TR because the alternative path through **SCC2** in the OG does not inherently capture the control-flow path from **S2** to **L0** via the backedge of the loop.

To correctly apply TR to remove redundant ordering edges, RIPTIDE uses *path-sensitive* TR. This analysis confirms that a transitive ordering path between a source and destination subsumes all possible control-flow paths between source and destination before removing any ordering arc from the OG.

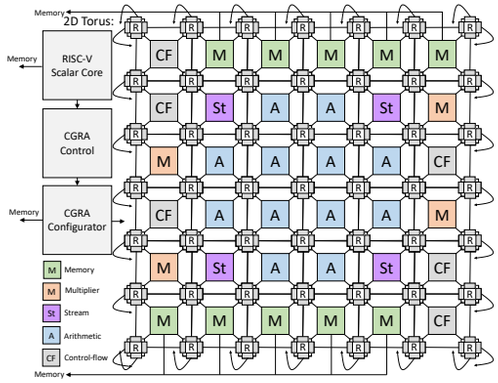

Figure 7: RIPTIDE's ULP CGRA fabric.

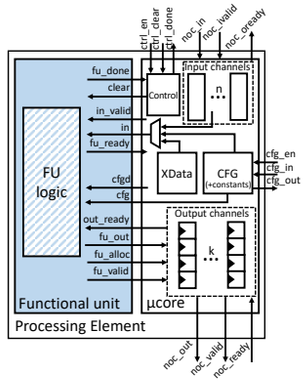

Figure 8: PE microarchitecture

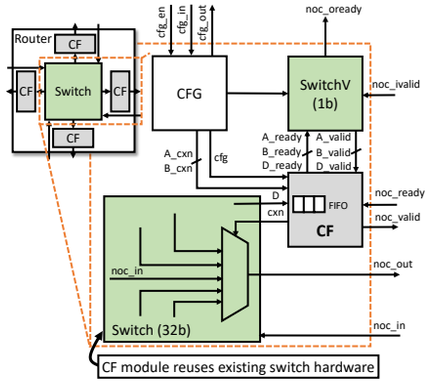

Figure 9: Router microarchitecture

With this constraint applied, RIPTIDE can safely use this modified TR on its OG. Prior efforts also incorporated control-flow to reduce ordering constraints [7, 44] or improve alias analysis accuracy to begin with [32].

Enforcing ordering constraints. Memory operations in RIPTIDE produce a control token on completion and can optionally consume a control token to enforce memory ordering. The middle-end encodes ordering arcs as defs and uses of data values in the IR (as seen in the IR transform of loads and stores in Fig. 5) before lowering them as dependences in the DFG. For a memory operation that must receive multiple control signals, the middle-end inserts merge operators (Sec. 4) to consolidate those signals.

5.2 Control-flow operator insertion

The compiler lowers its IR to use RIPTIDE's control paradigm by inserting RIPTIDE control-flow operators into the DFG.

Steer. The compiler uses the control dependence graph (CDG) [13] to insert steers. For each consumer of a value, the compiler walks the CDG from the producer to consumer and inserts a steer operator at each node along the CDG traversal. The steer's control input is the decider of the basic block that the steer depends on, and its data input is the value or the output of an earlier inserted steer.

Carry and invariant. For loops, the compiler inserts a carry operator for loop-carried dependences and an invariant operator for loop-invariant values into the loop header. A carry's data input comes from the loop backedge that produces the value. An invariant's data input comes from the loop pre-header. These operators should produce a token only if the next iteration of the loop is certain to execute; to ensure this behavior, the compiler sets their control signal to the decider of the block at the loop exit.

Order. If two iterations of a loop may take different control-flow paths that converge at a single *merge node*[1] in the loop body, either may produce a token to the merge node first. But for correctness, the one from the earlier iteration must produce the first token. The compiler inserts an order operator at a merge node in the CFG to ensure that tokens flow to the merge node in iteration order. The control signal `D` for the order operator is the decider of nearest common dominator of the merge node's predecessor basic blocks. Since the earlier iteration sends its control signal first, it blocks the later iteration until the earlier iteration completes.

[1]Not to be confused with merge *operators* in RIPTIDE.

5.3 Stream fusion

RIPTIDE performs target-specific operation fusion on the DFG to reduce required operations and routes by combining value *stream generators* with loop control logic and address computation logic. RIPTIDE supports stream operations and applies them for the common case of a loop with an affine loop governing induction variable (LGIV). A stream makes loop logic efficient by fusing the LGIV update and the loop exit condition into a single operation. In the DFG, loop iteration logic is represented by the exit condition, an update operation, the carry for the LGIV's value, and the steer that gates the LGIV in a loop iteration. The middle-end fuses these operations into a single stream operation and sets the stream's initial, step, and bound values. Fig. 5 shows stream compilation, where the operations for loop iteration logic (outlined in blue in the DFG) are fused into a stream operator. RIPTIDE uses applies induction variable analysis [1,3] to find affine LGIVs. RIPTIDE also identifies address computations, maps these to an affine stream if possible, and fuses the stream into the memory operation.

5.4 Mapping DFGs to hardware

RIPTIDE's backend takes a DFG and a CGRA topology description and generates scalar code to invoke RIPTIDE and a bitstream to configure the RIPTIDE fabric. The compiler's mapper uses a mixed integer linear program constraint problem formulation, the solution to which maps DFG nodes and edges to hardware PEs, CF-modules, and links. The mapper finds isomorphisms between the DFG and the CGRA fabric, while minimizing distance between operations, and maps control-flow operations onto the NoC (Sec. 6.4).

6. RIPTIDE MICROARCHITECTURE

RIPTIDE is an energy-minimal, ultra-low-power coarse-grained reconfigurable array (Fig. 7). The 6×6 fabric has a heterogeneous set of PEs connected via a bufferless, 2D-torus NoC. The fabric integrates a RISC-V scalar core and a 256KB (8×32KB banks) SRAM main memory.

6.1 Tagless dataflow scheduling

RIPTIDE implements asynchronous dataflow firing via *ordered dataflow* (Sec. 2). The fabric does not reorder tokens because RIPTIDE adds ordering operators where control may diverge. Tokens always match on arrival at a PE, obviating the need for tags. Tagless, asynchronous firing has a low hard-

nzb: Why?

ware cost (one bit per input plus control logic). Asynchronous dataflow firing lets RIPTIDE tolerate variable operation latency (e.g., bank conflicts) and eliminates the need for the compiler to reason about operation timing.

6.2 Processing elements

RIPTIDE's PEs perform all arithmetic and memory operations in the fabric. Fig. 8 shows the microarchitecture of a PE. The PE includes a functional unit (FU) and the µcore. The µcore interfaces with the NoC, buffers output values, and interfaces with top-level fabric control for PE configuration.

Functional units. The µcore exposes a generic interface using a latency-insensitive ready/valid protocol to make it easy to add new operators. Inputs arrive on `in_data` when `in_valid` is high, and are consumed when `fu_ready` is high. The FU reserves space in the output channel by raising `fu_alloc` (e.g., for pipelined, multi-cycle operations), and output arrives on `fu_data` when `fu_valid` is high. `out_ready` supplies back pressure from downstream PEs. The remaining signals deal with top-level configuration and control.

Communication. The µcore decouples NoC communication from FU computation. The µcore tracks which inputs are valid, raises backpressure on input ports when its FU is not ready, buffers intermediate results in output channels, and sends results over the NoC. Decoupling simplifies the FU.

Configuration. The µcore handles PE and FU configuration, storing configuration state in a two-entry *configuration cache* that enables single-cycle reconfiguration. Additionally, the µcore enables the fabric to overlap reconfiguration of some PEs while others finish computation on an old configuration.

PE types. RIPTIDE includes a heterogeneous of PEs:

- *Memory PEs* issue loads and stores to memory and have a "row buffer" that coalesces non-aliasing subword loads.
- *Arithmetic PEs* implement basic ALU operations, e.g., compare, bitwise logic, add, subtract, shift, etc.
- *Multiplier PEs* implement multiply, multiply + shift, multiply + fixed-point clip, and multiply-accumulate.
- *Control-flow PEs* implement steer, invariant, carry, select, merge, and order (Sec. 4) — but most of these are actually implemented in RIPTIDE's NoC (see below).
- *Stream PEs* implement common affine iterators (Sec. 4).

6.3 Bufferless NoC

RIPTIDE connects PEs via a statically configured, multi-hop, bufferless on-chip network with routers. Instead of buffering values in the NoC, PEs buffer values in their output channel. NoC buffers are a primary energy sink in prior CGRAs [21, 35], and RIPTIDE completely eliminates them. Similarly, RIPTIDE's NoC is statically routed to eliminate routing look-up tables and flow-control mechanisms.

6.4 Control flow in the NoC

Control-flow operations are simple to implement (often a single multiplexer), but there are often many of them. Mapping each to a PE wastes energy and area, and can render infeasible mapping to the CGRA. We observe that much of the logic required to implement control flow is already plentiful in the NoC. Each NoC switch is a crossbar that can be *re-purposed* to mux values for control. Implementing each control-flow operator simply requires routing values and manipulating upstream and downstream ready/valid signals.

```
cxn = A_cxn
forever:
    A_ready = D_ready = 0
    if A_valid && D_valid: # wait for A and D
        # if D is true, pass through A;
        # else discard A
        noc_valid = D
        A_ready = D_ready = noc_ready || !D
        if D: wait for noc_ready
```

(a) Steer (True flavor).

```
forever:
    # begin in Initial state
    if A_valid:
        cxn = A_cxn          # pass through A
        noc_valid = A_valid
        D_ready = A_ready = noc_ready
        B_ready = xxx        # don't care
    wait for noc_ready
    # transition to Block state
    do until D_valid && !D:
        cxn = B_cxn          # pass through B
        noc_valid = B_valid
        D_ready = B_ready = noc_ready
        A_ready = false      # hold A at input
        wait for noc_ready
```

(b) Carry.

Figure 10: Implementing control-flow operators using NoC control signals.

RIPTIDE's router microarchitecture is shown in Fig. 9. The router shares routing configuration and its data and valid crossbars with the baseline NoC. RIPTIDE adds a control-flow module (CFM) at output ports. The CFM determines when to send data to the router's output port and manipulates inputs to the data switch to select which data to send.

Control-flow module. The CFM takes eight inputs and produces five outputs that control router configuration and the dataflow through the network. The inputs are:

- `cfg`: configuration of the CFM;
- `A_valid`, `B_valid`, `D_valid`: whether inputs are valid;
- `D`: value of the decider;
- `A_cxn` and `B_cxn`: input ports for `A` and `B`; and
- `noc_ready`: backpressure signal from the output port.

From this, the CFM produces outputs:

- `A_ready`, `B_ready`, and `D_ready`: upstream backpressure signals that allow the CFM to block upstream producers until all signals required are valid;
- `noc_valid`: the valid signal for the CF's output; and
- `cxn`: which port (`A_cxn` or `B_cxn`) to route to the output port on the data switch.

Supported operations. The CFM can be configured for routing or for the control operators in Sec. 4. Implementing routing, e.g., `out = A`, is simple: `cxn = A_cxn`, `noc_valid = A_valid`, and `A_ready = noc_valid`.

Implementing other operators is slightly more involved, but requires only a small state machine. Fig. 10 is pseudocode for steer and carry operators (Sec. 4). A steer forwards `A` if `D` is true; otherwise, it discards `A`. To implement steer, the CFM waits for `A` and `D` to be valid. If `D` is true, then `noc_valid` is raised, and the `noc_ready` signal propagates upstream to `A` and `D` and the CFM waits for `noc_ready`, i.e., for the value to be consumed. If `D` is false, then `noc_valid` is kept low, and `A_ready` and `D_ready` are raised to discard these tokens.

Carry is a more complex control-flow operator. Carry begins in *Initial* state, waiting for a valid `A` token, which It forwards the token and transitions to *Blocked* state, where it forwards `B` until seeing a false `D` token.

Control-flow in the NoC adds small hardware overheads. Implementing control flow in the NoC is far more energy- and area-efficient than in a PE. The CFM deals only with narrow control signals and the 1b decider value `D`. It does not need to touch full data signals at all; these are left to the pre-existing data switch. Importantly, this means that the CFM adds no data buffers. Instead, the CFM simply raises the `*_ready` signals to park values in the upstream output channels until they are no longer needed.

By contrast, implementing control flow in a PE requires full data-width muxes and, if an entire PE is dedicated to control, an output channel to hold the results. Nevertheless, RIPTIDE is sometimes forced to allocate a PE for control flow. Specifically, if a control-flow operation takes a constant or software-supplied value, it currently requires μcore support.

Buffering of decider values. The CF module provides a small amount of buffering for decider values. This is because loop deciders often have high fanout, which means that the next iteration of a loop is likely blocked by one or more downstream consumers. To remove this limitation, RIPTIDE provides a small amount of downstream buffering for 1b decider signals, improving performance with minimal impact on area. The CFM uses run-length encoding to buffer up to eight decider values with just 3b of additional state.

7. EXPERIMENTAL METHODOLOGY

We evaluate a complete RIPTIDE system: the compiler built using LLVM and the microarchitecture fully implemented in RTL in an industrial, sub-28nm FinFET process.

Compiler. RIPTIDE's compiler passes extend LLVM 12.0 [37] and we compile workloads with (`-Oz`) to optimize code size. RIPTIDE's compiler middle-end uses LLVM's flow-insensitive alias analyses for memory ordering. The backend uses Gurobi 9.5 to solve its ILP to map to hardware [29].

Hardware. RIPTIDE is a complete RTL hardware implementation, including the RIPTIDE fabric, the RISC-V (RV32IMACE) scalar core, and a 256KB (8×32KB banks) SRAM main memory. We use Cadence Xcelium to simulate RTL to verify correctness and measure performance. We synthesize RIPTIDE using Cadence Genus and an industrial-grade, sub-28nm, high-threshold-voltage, FinFET PDK with compiled memories. To estimate power, we simulate full benchmarks post-synthesis and use Cadence Joules to estimate power from annotated switching activities.

Baselines. The evaluation compares to several baselines—scalar, vector, SNAFU, and three ASICs—also entirely in RTL, using the same design flow. All baselines and RIPTIDE use the same scalar core and main memory. The scalar baseline is a simple, six-stage microcontroller. The vector baseline adds a single-lane co-processor [24]. SNAFU is the state-of-the-art ULP CGRA, but requires hand-coded vector assembly programs. We compare to three custom ASICs on `dmm`, `sort`, and `fft` to evaluate the costs of programmability in RIPTIDE.

Benchmarks. We evaluate ten workloads important to the ULP domain running on random inputs. For the vector baseline, we vectorized all code by hand (except `dfs`, which does not vectorize well). SNAFU uses the vectorized code to generate its bitstreams. For RIPTIDE, we compile and run the plain scalar C implementation of each benchmark. The only exception is `sort`, for which we use merge sort on the scalar core and a C version of radix sort for RIPTIDE because it maps to the CGRA fabric.

8. EVALUATION

We evaluate RIPTIDE to show that it is easy to program in a high-language, uses 25% less energy than the state of the art, and improves performance by 17% on average and up to 2.5×. Moreover, control flow in the NoC is essential for large workloads and reduces energy by up to 2.25×.

8.1 Main results

RIPTIDE compiles high-level-language code to its fabric. RIPTIDE compiles, schedules, and runs ten applications on its 6×6 fabric. For all but `fft`, RIPTIDE offloads the entire benchmark onto the fabric, including outer loops. For `fft`, a 6×6 fabric does not have enough arithmetic or multiplier PEs, so we split `fft` into two separate functions. Further, RIPTIDE maps and runs `dfs`, which is *not possible* for the vector and SNAFU baselines (Xs in the figures).

nzb: Need a sentence on compilation time.

RIPTIDE saves energy. Fig. 11 presents energy of the scalar, vector, SNAFU, and ASICs normalized to RIPTIDE. RIPTIDE reduces energy by 6.6× vs. scalar, 3.1× vs. vector, and 25% vs. SNAFU. RIPTIDE uses less energy across the board. Fig. 11 breaks energy into memory, scalar, and vector/CGRA. RIPTIDE saves energy vs. scalar and vector because it does not fetch instructions, re-uses its configuration across many inputs, and forwards operands directly from producers to consumers. RIPTIDE uses less energy than SNAFU by reducing scalar computation: RIPTIDE runs outer loops on the fabric, but SNAFU runs them on the scalar core. Avoiding scalar work also eliminates instruction-fetch (memory) energy.

The only benchmark for which memory energy increases vs. SNAFU is `fft`. SNAFU uses scratchpads in the fabric for `fft`, which reduces main memory energy. Even without scratchpads, RIPTIDE shows an overall energy reduction. (RIPTIDE could use scratchpads as well in future work.)

`sconv`'s case shows how control-flow costs in RIPTIDE move from scalar core to the fabric (e.g., steer, carry). While RIPTIDE eliminates scalar cost (e.g., fetches), it adds fabric energy (vs. SNAFU) to support outer loops. Scalar execution is a small fraction of overall energy for `sconv`, so RIPTIDE provides no benefit on this benchmark. This result further shows that RIPTIDE's microarchitectural additions cost little energy.

Fig. 11 also compares RIPTIDE to hand-coded, fixed-function ASICs for `dmm`, `sort`, and `fft`. RIPTIDE uses just 53% more energy on average than the ASICs, while running applications compiled directly from C. RIPTIDE compares especially favorably to `dmm`, using just 32% more energy. The data show that the cost of RIPTIDE's programmability is low.

RIPTIDE is fast. Fig. 12 shows performance normalized to scalar. RIPTIDE is 6.2×, 3.4×, and 17% faster than vector, scalar, and SNAFU. RIPTIDE achieves this performance from C code without hand-coded assembly. RIPTIDE does especially well on `bfs`, with a 2.5× speedup vs. SNAFU. The benefit comes from RIPTIDE's ability to run even `bfs`'s irregular outer loop on the fabric, whereas SNAFU runs only inner loops on its fabric, causing a performance bottleneck on the

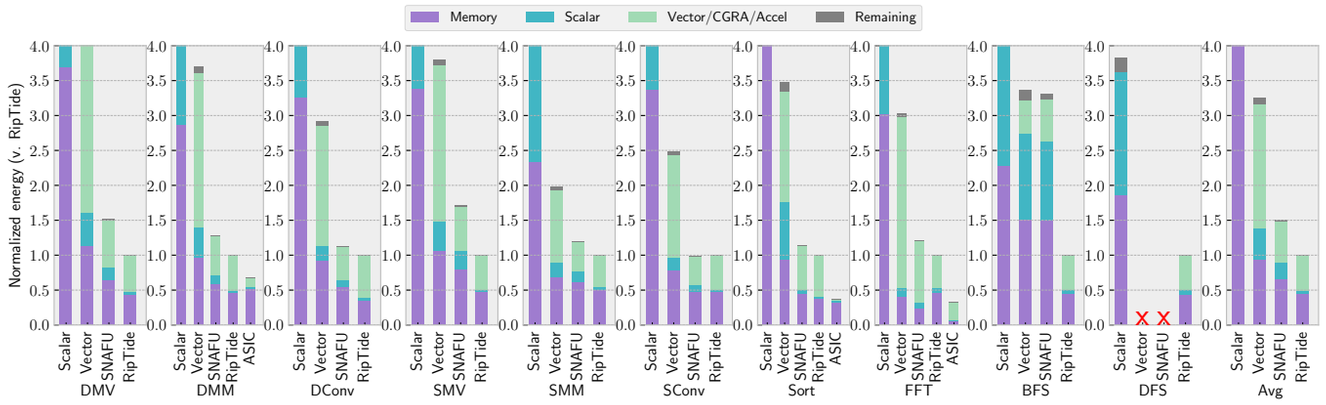

Figure 11: Shows the energy (normalized to RIPTIDE) of scalar, vector, SNAFU, RIPTIDE across ten benchmarks. RIPTIDE uses 25% less energy than SNAFU.

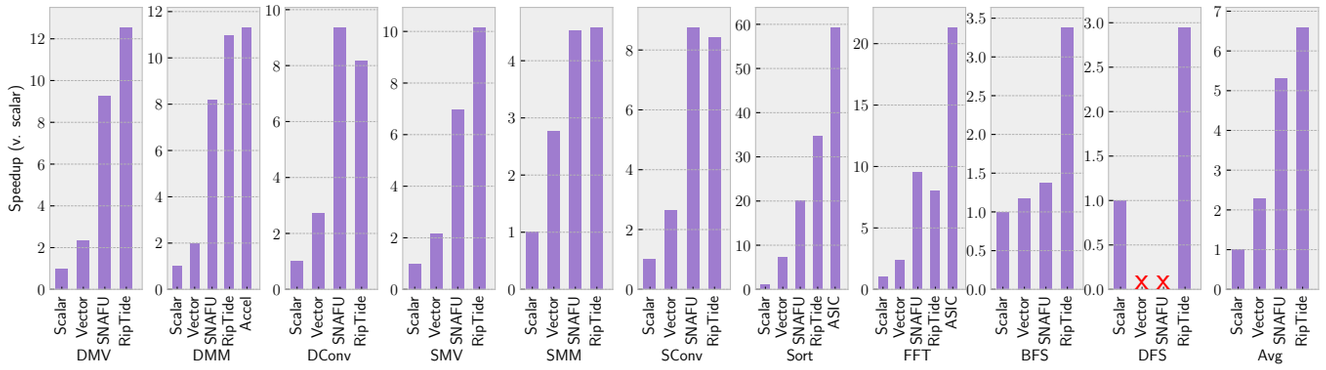

Figure 12: Shows the speedup v. scalar of scalar, vector, SNAFU, and RIPTIDE across ten benchmarks. RIPTIDE is 17% faster than SNAFU.

scalar core.

RIPTIDE is tiny and has extremely low power consumption. The complete RIPTIDE system is approximately 0.5mm^2 and operates between 320µW and 600µW, with negligible leakage (<3%) due to RIPTIDE's high-threshold-voltage process. Overall, the complete system, including memory, achieves 141 MOPS/mW vs. 110 MOPS/mW for SNAFU's full system.

8.2 Compiler characterization

RIPTIDE's compiler effectively optimizes dataflow graphs, reducing operation counts by 26% while enforcing memory ordering vs. an unoptimized DFG without ordering. The compiler also reduces programmer effort: RIPTIDE compiles from C with no hand-coded assembly, requiring just 8.7 added LoC on average over the original C (mostly for wrappers).

RIPTIDE's compiler reduces operation counts. Reducing operation count is important because they consume PEs in RIPTIDE's fabric. Fig. 13 shows operation counts by type with different optimizations applied. The first bar is an unoptimized DFG mapped to RIPTIDE. This graph requires many PEs to map to hardware and may yield incorrect results because it does not enforce memory ordering. The second bar adds streams, operator fusion, and redundant control flow elimination, reducing operation count by 32.6%. The third bar adds unoptimized memory ordering, which *increases* operations counts by 85.3% to ensure correctness. Mapping this graph to hardware is challenging due to its size. The fourth bar applies RIPTIDE's ordering optimizations (Sec. 5), reducing operation count (vs. unoptimized ordering) by 27.7%. The fifth bar adds programmer-inserted (`restrict`) annotations on pointers to better inform LLVM's alias analysis, reducing operation count by 18.2%. The last bar removes control-flow operations that map to RIPTIDE's NoC, reducing the number of operations on PEs by 37.5% and demonstrating the benefit of RIPTIDE's control flow in the NoC.

RIPTIDE reduces programmer effort. Fig. 15 counts code additions, including lines of code (LoC) in C, assembly, and `restrict` annotations. RIPTIDE has no hand-written assembly, compiling directly from C, while 32% and 27% of the LoC for the vector and SNAFU baseline are hand-written assembly. On average, vector adds 17 LoC vs. scalar, SNAFU adds 21 LoC vs. scalar, and RIPTIDE adds just 8.7 lines. Annotations in RIPTIDE represent a small fraction of the overall LoC, just 11.2% and, on average, the programmer adds 4.5 annotations per benchmark.

8.3 CF in the NoC saves energy & area

Fig. 14 quantifies the benefits of implementing control flow in the NoC (CFiN). From left to right, the plot shows base energy on RIPTIDE with CFiN, the same fabric with every control-flow operation on a PE, an "All PEs" larger fabric with all CF operations mapped to PEs, and a "Fused" fabric in which each PE supports one fused CF operation. RIPTIDE uses the least energy: 45% less than No CFiN, 42% less than All PEs, and 29% less than Fused. RIPTIDE's benefit stems from CFiN avoiding the overhead of a full PE. Other configurations also have unique problems. No-CFiN is possible only for `dmv` and `smv`, which are small enough to map to the same RIPTIDE fabric; other workloads have too many control-flow operations to map. The All PEs and Fused

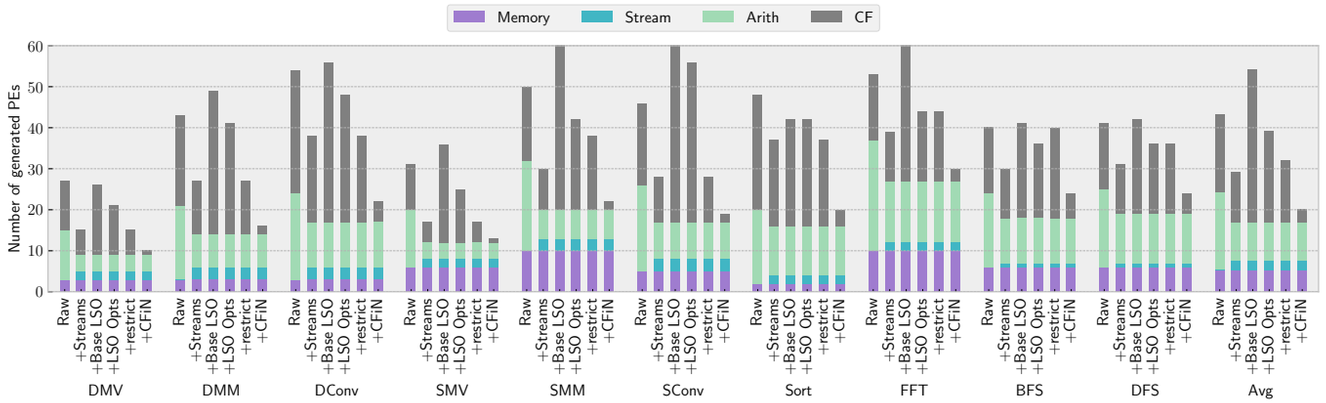

Figure 13: Shows operator counts for ten different benchmarks. Starting with an unoptimized, unordered baseline (Raw), compiler optimizations reduce operator counts while enforcing memory ordering, making it feasible to map benchmarks to hardware.

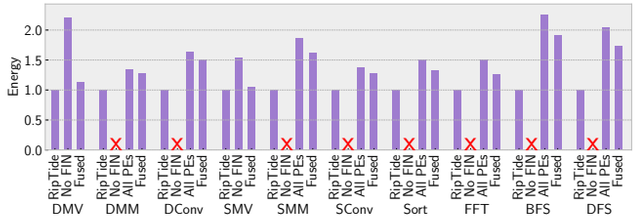

Figure 14: Quantifies the benefits of control flow in the NoC. RIPTIDE uses 45% , 42%, and 29% less energy than RIPTIDE w/ No CFiN, a fabric where all CF ops are PEs (All PEs), and a fabric that fuses CF ops into PEs (Fused).

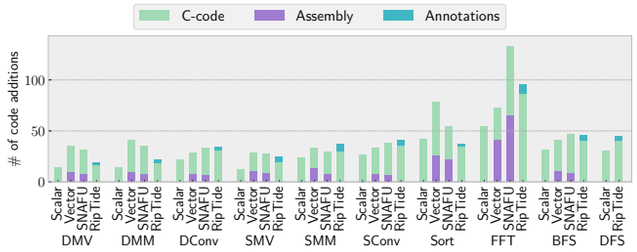

Figure 15: Shows the number of code additions for ten benchmarks running on scalar, vector, SNAFU, and RIPTIDE. RIPTIDE requires no hand-coded assembly unlike vector and SNAFU.

configurations add many control-flow PEs, wasting energy and area. In contrast, RIPTIDE is 22% and 17% smaller than All PEs and Fused, respectively.

9. IMPLICATIONS FOR GENERAL-PURPOSE ARCHITECTURE AND DARK SILICON

Fig. 11 and Fig. 12 compare the energy and performance for `dmm` on RIPTIDE vs. an equivalent ASIC. RIPTIDE does not compromise much on energy or performance — coming within 32% and 3%, respectively — but it is not a free lunch. There is a high area cost for RIPTIDE's programmability: RIPTIDE is 57× (only 9× for `fft`) larger than the ASIC. The question is, is RIPTIDE's programmability worth the extra area?

RIPTIDE area is inflated partly because of low utilization on PEs that perform outer loops. RIPTIDE only supports one operation per PE, so entire PEs are consumed even if an operation fires rarely. A future design could revisit this constraint to allow limited time-multiplexing, either through fine-grain [85] or coarse-grain [50] time multiplexing.

Regardless, the area difference shows potentially large cost savings from ASICs, so long as a computation is performed frequently enough to overcome ASICs' upfront design and verification costs. Standardized, pervasive tasks like JPEG compression and wireless communication protocols are good candidates for ASICs. But if the computation is prone to change or used infrequently, then this cost advantage rapidly disappears.

Some have proposed that, with increasing transistor budgets and stagnating power budgets, processors should embrace extreme heterogeneity and assemble a large number of distinct ASICs [76, 78]. The "garden of ASICs" approach lets architects do something with extra transistors, but it dramatically increases system design and verification cost. Moreover, the "garden of ASICs" approach creates herculean challenges in system integration, as there is no standard programming interface for ASICs, obsolescence is monotonic and inevitable, and programs must be somehow partitioned between ASICs and cores with accompanying data coordination issues.

RIPTIDE suggests an alternative approach. Rather than spend area on ASICs that will idle most of the time, instead build an energy-minimal, programmable dataflow fabric. The two designs take similar area with a few dozen ASICs. And the dataflow fabric is cheaper to design, more broadly applicable, and easier to use — programs can be simply compiled for a different target. Finally, as a general-purpose design, programmable dataflow fabrics can create a self-sustaining ecosystem that aggregates optimizations and achieves sufficient scale to justify cutting-edge silicon. All told, while dataflow fabrics like RIPTIDE are not a replacement for ASICs by any means, they could play an important role in improving the efficiency of general-purpose processing.

REFERENCES

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

[2] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972. [Online]. Available: https://doi.org/10.1137/0201008

[3] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 242–249. [Online]. Available: https://doi.org/10.1145/190347.190423

[4] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "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*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 918–932. [Online]. Available: https://doi.org/10.1145/3503222.3507772

[5] M. Budiu, P. Artigas, and S. Goldstein, "Dataflow: A complement to superscalar," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, 2005, pp. 177–186.

[6] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 2005, pp. 177–186.

[7] D.-K. Chen and P.-C. Yew, "Redundant synchronization elimination for doacross loops," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 459–470, 1999.

[8] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[9] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 184–189.

[10] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013, pp. 305–310.

[11] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 379–384. [Online]. Available: https://doi.org/10.1145/2333660.2333747

[12] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of cgra," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 9–16.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[14] V. Dadu, S. Liu, and T. Nowatzki, *PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators*. IEEE Press, 2021, p. 595–608. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00053

[15] V. Dadu and T. Nowatzki, *TaskStream: Accelerating Task-Parallel Workloads by Recovering Program Structure*. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–13. [Online]. Available: https://doi.org/10.1145/3503222.3507706

[16] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 924–939.

[17] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *ASPLOS 25*, 2020.

[18] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, no. 4, 1975.

[19] M. Duric, O. Palomar, A. Smith, O. Unsal, A. Cristal, M. Valero, and D. Burger, "Evx: Vector execution on low power edge cores," in *DATE*, 2014.

[20] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," *Intel white paper*, vol. 19, no. 20, 2008.

[21] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1027–1040.

[22] G. Gobieski, N. Beckmann, and B. Lucia, "Intermittent deep neural network inference," in *SysML*, 2018.

[23] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *ASPLOS*, 2019.

[24] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in *MICRO*, 2019.

[25] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, 2000.

[26] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, 2012.

[27] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 12–23.

[28] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H.-H. S. Lee, G.-Y. Wei, D. Brooks, and C.-J. Wu, "Chasing carbon: The elusive environmental footprint of computing," *IEEE Micro*, 2022.

[29] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[30] J. R. Hauser and J. Wawrzynek, "Garp: A mips processor with a reconfigurable coprocessor," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*. IEEE, 1997, pp. 12–21.

[31] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[32] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, p. 848–894, jul 1999. [Online]. Available: https://doi.org/10.1145/325478.325519

[33] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[34] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1–14.

[35] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *DAC*, 2017.

[36] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor, "Moonwalk: Nre optimization in asic clouds," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 511–526. [Online]. Available: https://doi.org/10.1145/3037697.3037749

[37] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, Mar. 2004.

[38] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: hybrid program analysis for determinism," 2012.

[39] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, "Dynaspam: Dynamic spatial architecture mapping using out of order instruction schedules," in *Proceedings of the 42nd Annual International*

Symposium on Computer Architecture, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 541–553. [Online]. Available: https://doi.org/10.1145/2749469.2750414

[40] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 417–433.

[41] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent Computing: Challenges and Opportunities," Dagstuhl, Germany, 2017. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7131

[42] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," in *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.* IEEE, 2002, pp. 166–173.

[43] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[44] S. Midkiff and D. Padua, "A comparison of four synchronization optimization techniques," in *Intl. Conf. on Parallel Processing*, vol. 2, 1991, pp. 9–16.

[45] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1485–1495, 1987.

[46] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[47] E. Mirsky, A. DeHon *et al.*, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources." in *FCCM*, vol. 96, 1996, pp. 17–19.

[48] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, 2006.

[49] T. Miyamori and K. Olukotun, "Remarc: Reconfigurable multimedia array coprocessor," *IEICE Transactions on information and systems*, vol. 82, no. 2, pp. 389–397, 1999.

[50] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[51] C. Nicol, "A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing," *WaveComputing WhitePaper*, 2017.

[52] R. S. Nikhil *et al.*, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on computers*, vol. 39, no. 3, 1990.

[53] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *PACT 27*, 2018.

[54] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: ACM, 2018, pp. 36:1–36:15. [Online]. Available: http://doi.acm.org/10.1145/3243176.3243212

[55] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ISCA 44*, 2017.

[56] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 298–310.

[57] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, no. 3, 2017.

[58] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *ACM SIGPLAN Notices*, vol. 48, no. 6, 2013.

[59] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, p. 82–91, may 1990. [Online]. Available: https://doi.org/10.1145/325096.325117

[60] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel *et al.*, "Triggered instructions: a control paradigm for spatially-programmed architectures," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013.

[61] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 166–176.

[62] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 370–380. [Online]. Available: https://doi.org/10.1145/1669112.1669160

[63] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik, "Chlorophyll: Synthesis-aided compiler for low-power spatial architectures," *SIGPLAN Not.*, vol. 49, no. 6, p. 396–407, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594339

[64] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *ISCA 44*, 2017.

[65] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector rda for sparsity," 2021.

[66] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *ISCA 30*, 2003.

[67] K. Sankaralingam, T. Nowatzki, G. Wright, P. Palamuttam, J. Khare, V. Gangadhar, and P. Shah, "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, 2021, pp. 1–20. [Online]. Available: https://doi.org/10.1109/HCS52781.2021.9567306

[68] M. Satyanarayanan, N. Beckmann, G. A. Lewis, and B. Lucia, "The role of edge offload for hardware-accelerated mobile devices," in *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 22–29. [Online]. Available: https://doi.org/10.1145/3446382.3448360

[69] M. Satyanarayanan, N. Beckmann, G. A. Lewis, and B. Lucia, "The role of edge offload for hardware-accelerated mobile devices," in *HotMobile*, 2021.

[70] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[71] P. Sparks, "A route to a trillion devices," *Arm WhitePaper*, 2017.

[72] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO 36*, 2003.

[73] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, "Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables," in *ISCA 45*, 2018.

[74] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 381–388.

[75] F. Tavares, "Kicksat 2," May 2019. [Online]. Available: https://www.nasa.gov/ames/kicksat

[76] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *DAC*, 2012.

[77] N. Vedula, A. Shriraman, S. Kumar, and W. N. Sumner, "Nachos: Software-driven hardware-assisted memory disambiguation for accelerators," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 710–723.

[78] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores:

reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, 2010.

[79] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 309–321.

[80] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," *ACM SIGARCH computer architecture news*, vol. 42, no. 3, 2014.

[81] D. Voitsechov, O. Port, and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," in *MICRO 51*, 2018.

[82] E. Waingold *et al.*, "Baring It All to Software: Raw Machines," in *IEEE Computer*, September 1997.

[83] M. A. Watkins, T. Nowatzki, and A. Carno, "Software transparent dynamic binary translation for coarse-grain reconfigurable architectures," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 138–150.

[84] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: synthesizing programmable spatial accelerators," in *ISCA 47*, 2020.

[85] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA*, 2020.

[86] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541961

[87] Y. Yang, J. S. Emer, and D. Sanchez, "Spzip: architectural support for effective data compression in irregular applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1069–1082.