The TYR Dataflow Architecture: Improving Locality by Taming Parallelism

Nikhil Agarwal, Mitchell Fream, Souradip Ghosh, Brian C. Schwedock,[†] Nathan Beckmann

{nikhilag, mfream, souradip}@cmu.edu   b.schwedock@samsung.com   beckmann@cs.cmu.edu

Carnegie Mellon University   [†]Samsung

Abstract—**Architectures should aim to maximize parallelism within a machine's finite memories, but prior designs tend to extremes, either maximizing parallelism or minimizing state. In particular, prior unordered dataflow architectures suffer from a *parallelism explosion* that creates unbounded state, requires prohibitively large associative memories, and risks deadlock. The few architectures that successfully navigate the parallelism-state tradeoff are limited to embarrassingly parallel programs.**

TYR is a new, general-purpose unordered dataflow architecture that achieves high parallelism with bounded state. The key insight is that prior unordered dataflow architectures are overly conservative, unnecessarily allocating tags from a single, *global* tag space. TYR exploits program structure to break up tags into *local tag spaces* that operate independently. Local tag spaces eliminate tag competition between co-dependent parts of the program, provably guaranteeing forward progress with only two tags per local tag space. TYR thus opens the door to an efficient, scalable implementation of unordered dataflow. Simulation of parallel programs demonstrates that TYR achieves parallelism nearly identical to a naïve unordered dataflow architecture with orders-of-magnitude less state.

Keywords-**Dataflow, parallelism, locality**

I. INTRODUCTION

THERE IS A FUNDAMENTAL TRADEOFF in computing between parallelism and live state. The more work done in parallel, the more state is required to track it. Too much live state can lead to a loss of locality, so more parallelism is not always better. Generally, the optimal operating point is a middle ground with sufficient parallelism to keep a processor busy, but not so much that state overwhelms on-chip memory.

Limitations of existing architectures. Unfortunately, even though the parallelism-state tradeoff has been recognized for decades [4, 17, 32], no existing architecture satisfactorily navigates this tradeoff. At one extreme, sequential architectures like von Neumann CPUs reduce state by severely constraining parallelism. Recovering parallelism from a sequential architecture is challenging, and these architectures are hard to scale. At the other extreme, unordered dataflow architectures [52, 58] suffer from "parallelism explosion," as some parts of a computation run far ahead of others. Dataflow's unbounded state is a longstanding unresolved challenge with the architecture, which complicates implementation and risks deadlocking the machine. Existing techniques to limit state in dataflow are not general [52] or impose arbitrary orderings, losing parallelism [22, 66, 73].

[†]This work was done while the author was at Carnegie Mellon University.

Other approaches to trade off parallelism and state have limited scope. Data-parallel architectures (e.g., GPUs [11, 39], TPUs [37], scalable vectors [64, 71], and stream dataflow [54, 62]) assume a large amount of identical, independent work, allowing the microarchitecture to choose how much work to execute in parallel. Data-parallel architectures can thus dynamically throttle parallelism to fit live state within the machine's finite memories. While highly effective, this approach requires an SPMD programming model that does not suit all programs, and these architectures also impose other inessential constraints on parallelism (e.g., due to branch divergence in GPUs [80]).

Goal: Taming parallelism to improve locality. This paper seeks a general-purpose architecture that balances parallelism and live state. Fig. 1 depicts our goal and approach, sketching a dataflow graph (DFG) of instructions and their dependencies in a mock program. A sequential architecture like a CPU executes the DFG in "depth-first" fashion, limiting execution to a narrow slice of the program at a time and frequently backtracking for independent work that could have been executed in parallel. By contrast, the dataflow architecture executes the DFG in "breadth-first" fashion, eagerly expanding work as it becomes available and thus having to track an exponentially increasing amount of state. Our architecture, TYR, explores a wide, but bounded, segment of the DFG at a time to saturate parallelism without blowing up the amount of live state.
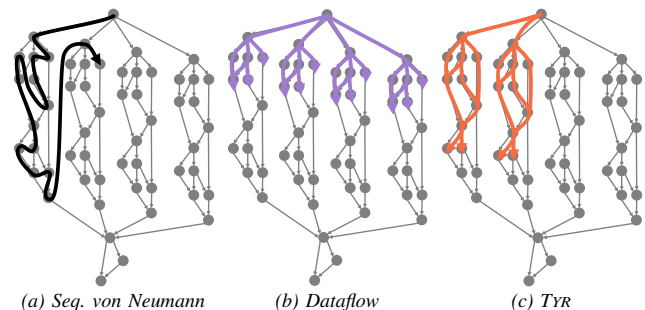

(a) Seq. von Neumann   *(b) Dataflow*   *(c) TYR*

Figure 1: How architectures execute a dynamic dataflow graph (DFG). Sequential von Neumann architectures have too little parallelism, exploring the DFG "depth-first". Dataflow architectures have too *much* parallelism, exploring the DFG "breadth-first" and exploding live state. TYR seeks a general-purpose solution that balances both, maximizing parallelism within a machine's finite resources.

Problem: Dataflow cannot safely limit parallelism. Given the well established difficulty of extracting parallelism from a

sequential architecture, our starting point is dataflow. Dataflow architectures use *tags* to disambiguate values from, e.g., different loop iterations. Naïvely, a simple way to throttle parallelism in a dataflow machine would be to limit the *tag space*, e.g., the number of loop iterations that can be active at once. However, limiting the tag space leads to deadlock, e.g., when an earlier loop iteration cannot proceed because all tags have been allocated to later, dependent loop iterations. Unable to safely limit the number of tags, prior dataflow architectures have been forced into large token stores, which have proved impossible to implement efficiently.

Key insight: A global tag space is unnecessary. We observe that this problem arises because all prior dataflow architectures allocate tags from a single, global tag space. Allocating tags from a global tag space guarantees that all tags are unique, but this is overly conservative. It is not a problem if the same tag is used by different parts of a program, *so long as one can be sure that the tags will never interact.*

TYR: A dataflow architecture with local tag spaces. TYR[1] is a new dataflow architecture that assigns each *concurrent block* in a program (i.e., function or loop body) its own *local tag space*. TYR exploits program structure to guarantee that the local tag spaces are *disjoint*, i.e., that their tags will never interact. Each local tag space can be tiny (two tags is enough) without risking deadlock. TYR can thus be implemented with small, private token stores and opens the door, for the first time, to a practical, scalable implementation of unordered dataflow.

TYR is general-purpose. We compile programs from C to TYR's dataflow ISA, breaking the program into local tag spaces at loop and function boundaries. TYR introduces new tag-management instructions to manage the local tag spaces. We describe the semantics required of `allocate` and `free` to guarantee forward progress on arbitrary programs, and we prove that TYR is deadlock-free and bounds the number of live tokens. TYR achieves high parallelism on arbitrary loops and acyclic call graphs, and it enables designers to trade off parallelism and state by scaling the number of tags.

Contributions. TYR is the first general-purpose architecture to bound state without artificially constraining parallelism or risking deadlock. This paper focuses on the high-level architectural primitives to navigate the parallelism-state tradeoff:

- We characterize the parallelism-state tradeoff in prior architectures and show that they all either limit parallelism, suffer from unbounded state, or have limited scope.
- We observe that global tag spaces in unordered dataflow architectures are overly conservative and are the root cause of parallelism explosion.
- We present *local tag spaces*, a general-purpose technique that exploits program structure to tame parallelism and improve locality.
- We introduce TYR, a new dataflow architecture and compiler, that implements local tag spaces.
- We prove that TYR is deadlock-free and bounds live state.

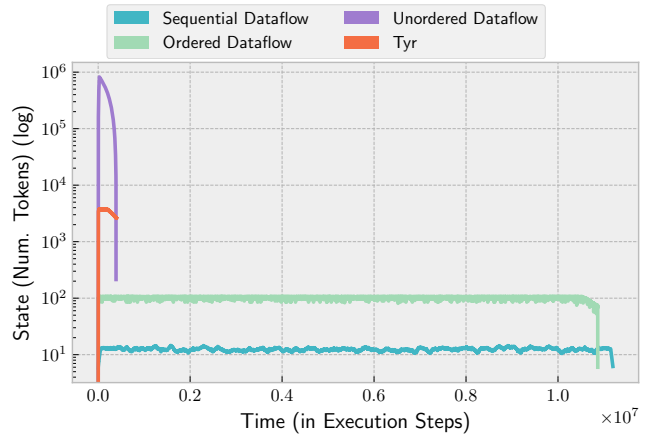[1]After the Norse god who slayed the monstrous wolf Fenrir [1, 72].

Figure 2: Results for sparse matrix multiplication (spmspm). Unordered dataflow suffers from parallelism explosion that exponentially increases live state. Sequential and ordered dataflow have poor performance. Only TYR achieves high performance with low state.

Summary of results. We evaluate TYR in high-level simulation against several prior architectures (Sec. II): sequential, ordered dataflow, and unordered dataflow. Our results show that TYR achieves the best of all worlds: it achieves nearly the same performance as a naïve unordered dataflow architecture with orders-of-magnitude less live state. Fig. 2 shows a representative execution trace of total live state (*y*-axis, log-scale) over timesteps (*x*-axis) for sparse matrix-matrix multiplication. Each trace ends when the program completes. Sequential and ordered dataflow architectures have little state, but also poor performance. Meanwhile, tagged dataflow architectures complete the program quickly, but have orders-of-magnitude more live state than other designs. Only TYR balances parallelism and locality, keeping state bounded while completing the program quickly.

II. BACKGROUND AND MOTIVATION

A. Resurgence of dataflow for programmable acceleration

Our work is motivated by the need for an efficient, general-purpose architecture. The end of Dennard scaling has made processors power-limited [26, 76], and sequential CPU architectures are horribly inefficient, wasting upwards of 99% of their energy [28, 36]. Some foresee a future of "dark silicon," where systems feature a wide variety of specialized accelerators [26,40,76]. Such accelerators have already gained traction in industry for some workloads [13,37].

However, specialized accelerators have serious drawbacks and deliver lower benefits than generally advertised. Amdahl's Law tells us that specialization is at odds with efficiency [29,35], since an application is quickly limited by whatever runs least efficiently. Custom hardware requires large and growing non-recurring engineering (NRE) costs [40], which only a few applications can justify. Specialized hardware is very disruptive to software, and there is no scalable and maintainable solution for integrating an increasing variety of accelerators into existing software development flows. Finally, the cost

and carbon footprint of idle silicon is considerable, meaning dark silicon is not the free lunch it initially appears [9, 33]. These considerations will limit specialized accelerators to a few high-value domains, like deep learning [12, 13, 37].

Most applications will rely on general-purpose (i.e., software-programmable) architectures. Recent spatial architectures, like coarse-grained reconfigurable arrays (CGRAs) and reconfigurable dataflow architectures (RDAs), are increasingly general-purpose and easy to program [6, 14, 15, 19, 25, 28–31, 38, 44–49, 54, 59, 60, 62, 65–67, 70, 74, 75, 77, 79, 81, 84, 85]. They have led to a resurgence of interest in dataflow [18, 19, 22, 28, 29, 53–56, 62, 79, 82].

B. Dataflow basics

Dataflow architectures represent programs as graphs of instructions, with edges denoting producer-consumer relationships. Each instruction follows the *dataflow firing rule* [22, 52], executing once all its inputs are available. There is no program counter or global order on instructions. Data is communicated between producers and consumers in packets called *tokens*.

Fig. 3 shows pseudocode and a corresponding dataflow graph for dense matrix-vector multiplication (dmv). The yellow circular nodes represent classic arithmetic and memory instructions, while the red triangles represent control-flow instructions. In a dataflow machine, control flow takes the form of steer instructions, not branches, that conditionally route a data token depending on a boolean decider token. For example, in Fig. 3b, a steer routes w around the loop based on the for-loop comparison (i.e., so long as i != m), and another steer routes w to the store after the loop (i.e., once i == m).
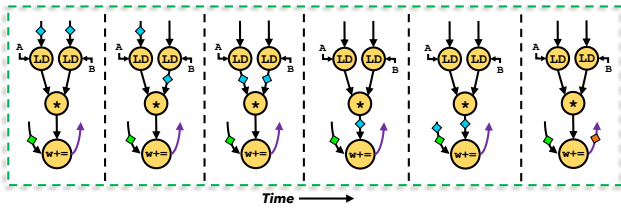
Figure 3: A running example: dense matrix-vector multiplication. We zoom in on the innermost loop body (green box) to demonstrate dataflow execution.

```
def dmv (A, B, Z, m, n):
    for i = 0..m:
        w = 0
        for j = 0..n:
            w += A[i,j] * B[j]
        Z[i] = w
```

(a) Pseudocode.
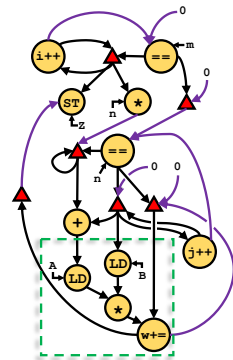
(b) Dataflow graph (DFG).

(c) Execution trace of innermost loop.

Fig. 3c illustrates tokens flowing through the innermost loop body of dmv. Tokens are shown as diamonds, and their color represents a *tag* that indicates, in this example, which

loop iteration they belong to. Initially, tokens from iteration 0 (cyan) arrive at the two loads and at w+= at the bottom. The loads immediately fire, since they have all inputs ready, but the add must wait for its remaining input. The load from B completes first, followed by the load from A. Once both return, the multiply is ready and fires, at last enabling w+= to fire. Finally, the updated value for w is sent back to the top of the loop, with a new tag representing iteration 1 (orange). Note that w+= does *not* fire after step 4 because, although input tokens are available, their tags do not match (green vs. cyan).

C. Surveying the parallelism vs. state tradeoff

To characterize how prior architectures balance parallelism and state, we introduce the notion of ***token synchronization***. A running program spawns many dynamic instances of the same instruction(s) (e.g., across iterations of a loop). For correct execution, architectures must route values (tokens) to the appropriate dynamic instruction (e.g., the right loop iteration). *Token synchronization* is merely how an architecture does this, i.e., how it disambiguates between dynamic instances of an instruction, either by ordering them or giving them unique names (or both). For example, a sequential architecture disambiguates loop iterations by serializing them, whereas a dataflow architecture assigns each iteration a unique tag.

Token synchronization is the lens through which we understand how prior architectures balance parallelism vs. state. We consider a wide range of sequential, ordered, unordered, and data-parallel architectures. This section surveys how each balances parallelism and state, using an execution trace of dmv to drive the discussion.
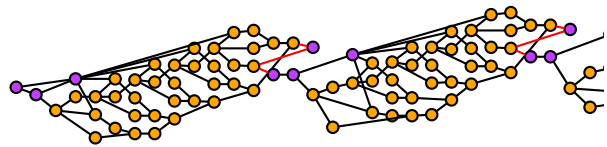
Figure 4: Partial execution trace of dmv.

Fig. 4 zooms into a trace of dmv. We show the dynamic execution graph, unrolling execution to show realized control flow and communication. Purple nodes are outer-loop instructions, and yellow nodes are inner-loop instructions. Black edges are communication, and red edges show other orderings required by the execution model. The width of the graph corresponds to time steps; height corresponds to parallelism. Parallel instructions running in the same time step are vertically adjacent, and the number of black edges crossing a vertical cut is the number of live tokens in the program at that point in time. The graph is simplified to show only necessary edges, i.e., redundant orderings are removed.

Sequential von Neumann (vN). At one extreme, vN architectures (i.e., CPUs) impose a sequential program order on instructions; *sequential ordering is vN's token synchronization scheme*. Fig. 5a illustrates this as red edges between pairs of subsequent instructions. These extra edges enforce a total ordering on instructions, modeling the program counter. In a
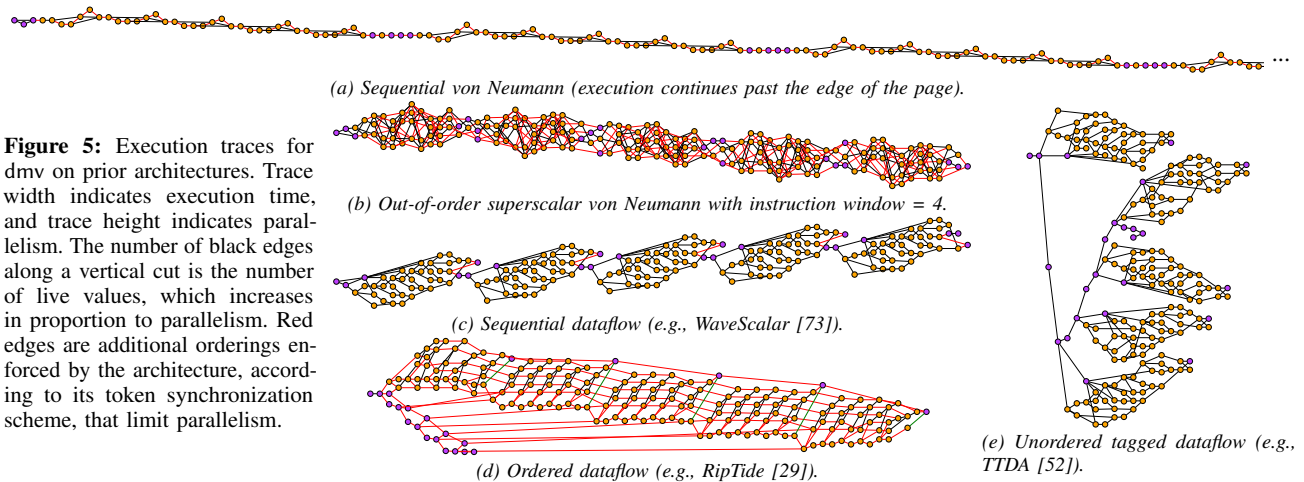
1186

(a) Sequential von Neumann (execution continues past the edge of the page).

Figure 5: Execution traces for dmv on prior architectures. Trace width indicates execution time, and trace height indicates parallelism. The number of black edges along a vertical cut is the number of live values, which increases in proportion to parallelism. Red edges are additional orderings enforced by the architecture, according to its token synchronization scheme, that limit parallelism.

(b) Out-of-order superscalar von Neumann with instruction window = 4.

(c) Sequential dataflow (e.g., WaveScalar [73]).

(d) Ordered dataflow (e.g., RipTide [29]).

(e) Unordered tagged dataflow (e.g., TTDA [52]).

(f) Data-parallel (e.g., vector, GPU).

sense, vN's sequential ordering restricts execution to a "depth-first" traversal of a program's full dynamic execution graph (Fig. 1). The result is minimal parallelism, so that program execution takes a long time (the graph is wide) but live state is minimized (the graph is short).

Parallelism can be increased by having multiple vN execution streams, i.e., multithreading. Token synchronization now includes a token's thread id, in addition to its vN ordering. Multithreading punts all of the problems of parallelism, including balancing parallelism and state, to the programmer.

Unordered (tagged) dataflow. At the other extreme, an unordered dataflow machine uses the most permissive token synchronization. Unordered dataflow architectures assign each dynamic instance of an instruction a unique name, or "tag." Instructions fire whenever inputs with matching tags are present. *Tag matching is unordered dataflow's synchronization scheme.* Data dependencies are the *only* constraint on parallelism, unleashing parallelism across the entire program. Examples include TTDA [52], Monsoon [58], and the Manchester Dataflow Machine [34]. Fig. 5e illustrates unordered dataflow. Execution is much different from sequential vN: all five inner-loop iterations execute in parallel, and execution is short (the graph is narrow) but with much more live state (the graph is tall). As the number of outer-loop iterations increases, parallelism explodes along with live state.

- ***Problem #1: Parallelism explosion.*** The abundant parallelism in unordered dataflow architectures is both a blessing and a curse. A longstanding problem with these designs is *parallelism explosion*, where runaway parallelism exhausts the machine's finite memory, hurting performance or even causing deadlock [4,17]. Parallelism explosion arises as the architecture greedily explores available work (Fig. 1). Prior unordered dataflow architectures recognized this problem but offered ad hoc solutions: e.g., TTDA limits the number of outstanding innermost loop iterations [52]. *Parallelism cannot be constrained without risking deadlock* (Sec. V).
- ***Problem #2: Implementation complexity.*** Unordered dataflow machines require an associative memory to match
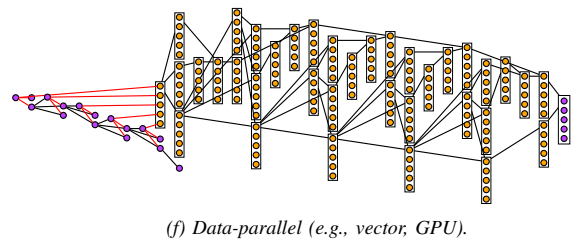
tags for instruction scheduling, which grows with parallelism because it must track all live tokens. Token stores are expensive to implement in hardware and scale poorly. Thus, as a corollary to Problem #1, *the inability to bound the size of token storage is a major unsolved implementation challenge in unordered dataflow architectures.*

Prior work has recognized these problems and has explored limiting parallelism by constraining tags. Unfortunately, *prior work on constrained tag spaces is not general-purpose.* TTDA targets innermost loops of scientific applications, limiting the number of tags available to some bound k (Sec. VIII). But k-bounding only applies to affine loops. By contrast, TYR's techniques are general-purpose, targeting, e.g., arbitrary loops and function calls.

Because they are unable to constrain tags, other unordered dataflow architectures are forced to track unbounded state. For example, the Manchester Dataflow Machine provides an "overflow unit" that spills tokens to main memory, delaying deadlock from unconstrained parallelism at the cost of extra data movement [34]. Deadlock is still possible if parallelism exhausts main memory.

Out-of-order vN. In between these extremes are architectures that trade off parallelism and state. The obvious hybrid of vN and dataflow is the out-of-order (OoO) superscalar processor. These architectures synchronize tokens through a combination of ordering and naming, executing instructions from within a window of the vN order and renaming values to disambiguate duplicates of the same instruction. Fig. 5b illustrates dmv on OoO with a window of 4 instructions; red edges now order

1187

instructions 4 or more apart in vN order. Parallelism increases by nearly $4\times$, and live state is kept small. However, OoO is still fundamentally vN: reordering is limited to a small region of the vN execution order, preventing the OoO processor from discovering parallelism across, e.g., outer-loop iterations. The scaling, efficiency, and implementation challenges of OoO processors are well documented [36, 73] and, despite decades of effort, remain unresolved.

Ordered dataflow. From the other direction, *ordered dataflow* machines impose a partial ordering on dataflow execution. These architectures synchronize tokens by having instructions communicate through FIFO queues. FIFO queues sequentialize execution of the *same* instruction but allow parallel execution of *different* instructions [28, 29, 31, 62, 69, 86]. Since tokens are ordered, there is no need for a tag, other than the instruction itself. The queue size also limits the number of dynamic instances of each instruction, applying back pressure to upstream instructions. Similar to vN, back pressure does not risk deadlock due to the global order enforced at each instruction.[2] Ordered dataflow thus avoids parallelism explosion.

However, ordered dataflow loses parallelism and performance. While ordered dataflow allows instruction-level parallelism and fine-grained pipelining, it is prone to stalls as long-latency operations block later instances of the same instruction from executing. Fig. 5d illustrates ordered-dataflow execution. Instructions within a single loop iteration run in parallel, but *across* loop iterations are serialized because they share a FIFO queue — see the red edges in the graph. Moreover, outer-loop instructions are pulled far forward in the execution (vs. the vN order) because there is no data dependence to prevent their execution earlier. The result is a middle ground between vN and unordered dataflow, both in performance and live state.

Ordered dataflow promises more parallelism than OoO, with simpler hardware than OoO or unordered dataflow. Most CGRAs are thus ordered-dataflow architectures. However, simplicity comes at increasing loss in parallelism with, e.g., the number of outer-loop iterations in dmv. Hence, *un*ordered (i.e., tagged) dataflow has recently re-gained interest in CGRAs for irregular programs with unpredictable latency, e.g., in sparse workloads, or with multiple threads [43, 68, 79, 81].

Sequential dataflow. Sequential dataflow architectures impose orderings even stricter than ordered dataflow. These architectures implement global ordering points, corresponding to the vN ordering at basic- or hyper-block boundaries, that limit parallelism from spanning code regions. Sequential dataflow processors use the dataflow firing rule within small regions, but this is not their main token-synchronization scheme. Rather, *sequential dataflow's token synchronization scheme is its total ordering on the basic- or hyper-blocks.* Examples include WaveScalar [73] and TRIPS [66]. WaveScalar orders instructions via a monotonically increasing tag, or "wave number." The wave number is advanced (incremented) at the end of each loop iteration or function call.

One might think that, by using tags, WaveScalar unleashes the unconstrained parallelism of unordered dataflow. However, because wave numbers must increase monotonically, instructions must wait until their wave number is resolved, which depends on the dynamic control flow of all earlier hyperblocks. In other words, instructions must wait for their "turn" in the global block-order. In fact, *all live values* must replicate the control flow of *all preceding blocks* to arrive at the right wave number [61]. Fig. 5c illustrates dmv in WaveScalar. Inner-loop iterations execute in parallel, but all tokens must advance alongside these inner loops to synchronize wave numbers. Outer-loop iterations (purple) are thus ordered globally between inner-loop nests, unlike in ordered dataflow. The result is closer to wide-issue OoO vN than to unordered dataflow, but implemented in a fully distributed fashion. Indeed, WaveScalar's (and TRIPS's) original motivation was to create a more scalable superscalar vN core. Accordingly, sequential dataflow architectures' parallelism is limited to the current block, akin to OoO's instruction window.

Data-parallel. Finally, data-parallel architectures like vector machines and GPUs apply the same sequence of instructions to many data concurrently. These architectures require that programs have an "embarrassingly parallel" structure to make this possible, corresponding to, e.g., some loop or map. Fig. 5f illustrates a data-parallel execution of dmv. First, there is a section of sequential code, in this case to compute the usable vector width and the offset of each row in the matrix. Then data-parallel execution starts wherein vectorized operations act on multiple data simultaneously, ending with a vectorized store for the entire output. Several recent dataflow architectures have adopted a similar approach, executing a data-parallel program as many independent streams [19, 54, 62, 65, 67, 69, 78, 79, 81].

The embarrassingly parallel program structure is essential to data-parallel architectures. Like unordered dataflow, data-parallel is not limited to a small instruction window; in dmv, data-parallel exploits parallelism across entire loop iterations. But unlike unordered dataflow, a data-parallel architecture is *free to choose as much parallelism as it wants* to maximize performance while staying within the machine's resources. This strategy is only safe for embarrassingly parallel programs, where each thread is doing independent work, and would otherwise risk deadlock.

Summary. Architectures seek to maximize parallelism within a machine's finite resources. Unordered dataflow architectures look promising, but parallelism explosion necessitates impractically large token stores to avoid deadlock. Other architectures (vN, OoO, ordered/sequential dataflow) add artificial orderings to the program execution that limit state at a severe cost in parallelism, usually limiting execution to a small window around a "program counter." Existing techniques that successfully navigate parallelism and state, e.g., TTDA's *k*-bounding and data-parallel architecture, are not general-purpose and only apply to embarrassingly parallel programs. TYR's contribution is to generalize these techniques across other program structures.

---

[2]Sec. VIII discusses deadlock in functional vs. imperative languages.

III. Local Tag Spaces

TYR is a new unordered dataflow architecture that achieves high parallelism with bounded state. As shown in Fig. 6, TYR breaks the tag space into several *local tag spaces* that correspond to program structure. Local tag spaces allow TYR to tame parallelism to fit token state within a machine's finite memories without risking deadlock, unlike prior unordered dataflow architectures. Moreover, TYR is fully general-purpose and makes no assumptions on program structure, unlike data-parallel approaches to parallelism.
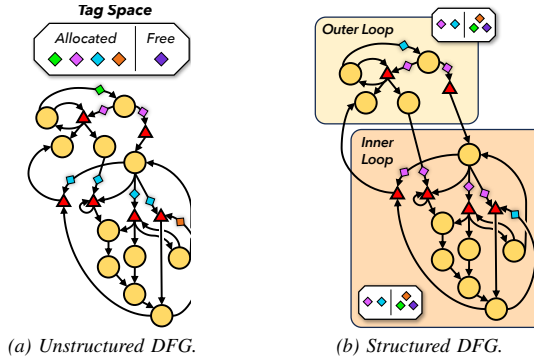

(a) Unstructured DFG.   *(b) Structured DFG.*

Figure 6: Local tag spaces allow TYR to balance parallelism and state, shown with DFGs of dmv. *(a)* Prior unordered dataflow architectures allocate tags from a global tag space, failing to exploit structure or bound tag usage. *(b)* TYR allocates tags from *local tag spaces* corresponding to program structure, allowing TYR to scale parallelism by tuning the number of tags. TYR ensures forward progress with just two tags per local tag space.

Insight: Unordered dataflow architectures over-synchronize. Fig. 6 shows the dataflow graph of dmv (pseudocode in Fig. 3a). Prior unordered dataflow architectures (Fig. 6a) execute an *unstructured* dataflow graph and allocate tags from a single, *global* tag space. By contrast, TYR (Fig. 6b) leverages program structure to break the dataflow graph into subgraphs, each with its own *local* tag space.

A global tag space is sufficient to ensure that there are no tag collisions, but we observe that it is far more conservative than necessary. A new tag is allocated whenever entering a loop, so tags from inner and outer loops can never interact. Prior designs are *synchronizing tokens that were never at risk of collision* to begin with. The inner and outer loops can allocate tags completely independently — in fact, it is safe for them to simultaneously use the same tag.

TYR leverages program structure via local tag spaces. TYR breaks the program into *concurrent blocks*, which allocate tags independently. A concurrent block is a group of static instructions that may have multiple instances executing concurrently, but has no internal concurrency. Each concurrent block thus requires a distinct tag, but requires no internal tag changes. In other words, concurrent blocks are the natural unit of tag management, akin to basic blocks or activation frames [50, 58]. Concretely, a concurrent block is a directed acyclic graph (DAG), i.e., straight-line or forward-branching

code. Within a DAG, instructions are not reused, so there is no possibility of concurrency. Concurrency naturally arises across loop iterations or function calls, so TYR breaks the program into concurrent blocks at loop and function boundaries.[3]

Concurrent blocks communicate through *transfer points*, which cut the dataflow graph at concurrent-block boundaries to translate tags between the local tag spaces. Tags from different concurrent blocks can never meet without having first been translated by a transfer point, and so aliasing across local tag spaces is impossible.

Local tag spaces in action. Fig. 7 shows the detailed dataflow graphs for dmv in naïve unordered dataflow and TYR. In naïve unordered dataflow (Fig. 7a), tags are allocated along every backedge (T nodes) from a single, global tag space. Thus, the inner and outer loop nests compete for tags, and a large number of tags are needed to avoid deadlock. By contrast, TYR (Fig. 7b) breaks dmv into two concurrent blocks (CBs) corresponding to the outer and inner loop. Each concurrent block gets its own local tag space, and the loops do not compete for tags. Forward progress can be guaranteed with minimal tags — as little as two per concurrent block.
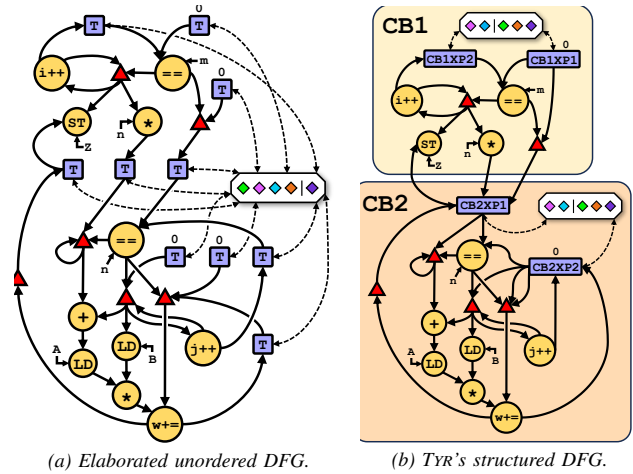

(a) Elaborated unordered DFG.   *(b) TYR's structured DFG.*

Figure 7: Unordered dataflow programs produce a large number of Tag Change (T) operations that compete to allocate tags from a single global tag space. TYR breaks dmv into two *concurrent blocks (CBs)*, each with its own local tag space. CBs are connected by *transfer points* (XPs), which transfer between tag spaces. Tags are allocated independently across CBs, and TYR ensures that a tag will always be available when needed to ensure forward progress.

Each concurrent block (CB) is guarded by transfer points (XPs) at its boundary. dmv has two transfer points per block: e.g., CB2XP1 from the outer loop to start a new inner-loop nest, and CB2XP2 along the backedge for each new inner-loop iteration. (Loops always have exactly two transfer points, unlike functions, which have one per caller.) A transfer point is responsible for allocating a new tag in the child's tag space

---

[3]Interprocedural analysis could allow for larger concurrent blocks, but using loop and function call boundaries is simple and safe. goto statements that transform into natural loops are supported, but gotos which create irreducible control flow [3], such as by jumping into the middle of a loop, are not [24].

for all incoming tokens (e.g., arguments), and resetting back to the parent's tag for all outgoing tokens (e.g., return values). In dmv, CB2XP2 allocates a new tag and assigns it to i, j, and w, and restores w's tag on exiting the inner loop.[4]

Local tag spaces allow TYR to always make forward progress. Forward progress cannot be guaranteed in prior unordered dataflow architectures because of unconstrained competition for tags — there is no way to prevent the last tag from being claimed by work that will later stall, waiting for something earlier to finish. TYR structures the tag space, enabling us to reason about the interactions between program regions and ensure that a tag will be available when it is needed (Sec. V).

Local tag spaces enable smaller, simpler, and more scalable hardware. For decades, unordered dataflow architectures have been hamstrung by large, associative memories needed to implement their unbounded global tag spaces. By contrast, TYR distributes tag management across many, independent, bounded local tag spaces. Thus, although an implementation is outside the scope of this paper, TYR is far more amenable to simple, scalable hardware than prior unordered dataflow architectures (Sec. VIII). Moreover, local tag spaces give systems a new knob to trade off parallelism and locality (Sec. IV-D).

IV. TYR ARCHITECTURE

TYR is a general-purpose unordered dataflow architecture that implements local tag spaces. TYR introduces new instructions to manage local tag spaces (Fig. 8) that prevent the system from running out of tags, even with only two tags per block. TYR is agnostic to programming language, is highly parallel, and guarantees forward progress with bounded state.

A. TYR instruction set

TYR has no program counter. Instead, instructions follow the *dataflow firing rule*, executing whenever their inputs are available (Sec. II-B). TYR's ISA, summarized in Table I, has four categories of instructions: arithmetic, memory, control flow, and token synchronization. TYR provides a standard set of arithmetic instructions. TYR implements control flow via steer instructions that conditionally route tokens to the taken path of a branch [22, 29, 73]. TYR supports conventional, mutable memory via loads and stores, converting memory ordering into explicit data dependencies in the dataflow graph [2, 29]. TYR's main contribution is its token-synchronization instructions, which allow it to bound token state. We first describe these instructions and then demonstrate them in an example.

Table I: TYR's instruction set.

Category	Instruction(s)
Arithmetic	$+, -, \times, \div, <<, \neq$, etc.
Memory	load, store
Control flow	steer, join
Token synchronization	allocate, free, changeTag, extractTag

[4]Alternatively, one could forward the parent context's live values to match the child, like WaveScalar [73]. But this strategy requires a global tag space and a much larger number of tag-management operations (Sec. VIII).
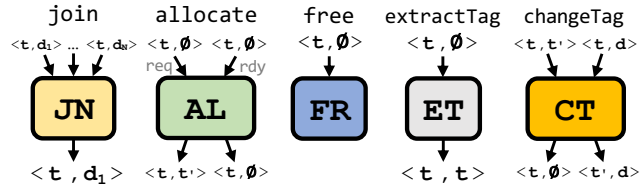

Figure 8: TYR's new instructions for token synchronization. Inputs and outputs are tagged tokens, denoted as <tag, data> = <t, d>. Control tokens are denoted with data = $\varnothing$.

Definition. *A context is a dynamic instance of a concurrent block, e.g., a single loop iteration or function invocation.*

join is an *n*-input barrier that waits for all inputs to arrive and then produces a copy of its left input. join has two roles in token synchronization: *(i)* join checks if a context will make forward progress, which is equivalent to waiting until all inputs are ready (Sec. V); and *(ii)* join checks if it is safe to free a tag, which is equivalent to waiting until all instructions have completed (see free below).

allocate generates a new tag when a context is spawned. All allocate instructions into the same concurrent block share the same free list. When tags are plentiful, allocate simply pops a tag from the free list. But, to guarantee forward progress, allocate will pop the last tag only if the context is ready.

allocate takes two zero-bit inputs, request and ready, with a special firing rule: request carries the parent context's tag and no data payload, <t, $\varnothing$>. If the free list has more than one tag, allocate immediately pops and returns the new tag, <t, free_list.pop>, and the ready input is consumed without effect when it eventually arrives. Otherwise, to guarantee forward progress, allocate waits for ready before popping and returning, consuming both inputs.[5]

Tail-recursive allocations (e.g., loops) are a special case for allocate because the parent and child are in the same tag space. A tail-recursive context cannot finish until it allocates its child, so a "spare tag" is required for forward progress. Thus, allocates on the "external" (i.e., not-tail-recursive) edge only pop from the free list if there are at least *two* tags remaining.

free returns a tag to the local tag space's free list. free consumes a single zero-bit input, <t, $\varnothing$>, and adds t to the free list. In order to safely free a tag, free must ensure that no other tokens exist with that tag. The compiler inserts a join before the free whose transitive fan-in reaches every instruction in the concurrent block.

The free barrier requires that all instructions create an output token with the same tag as their inputs. Most instructions satisfy this constraint automatically, but there are exceptions: store normally produces no output, so TYR adds a control output <t, $\varnothing$> for the free barrier, which is also used for memory ordering. changeTag (see below) normally generates a token with a different tag t', so TYR similarly adds a <t, $\varnothing$> output. steer *conditionally* generates a token on its primary output,

[5]allocate could return even earlier by tracking if *any* active context is ready, at the cost of additional complexity.
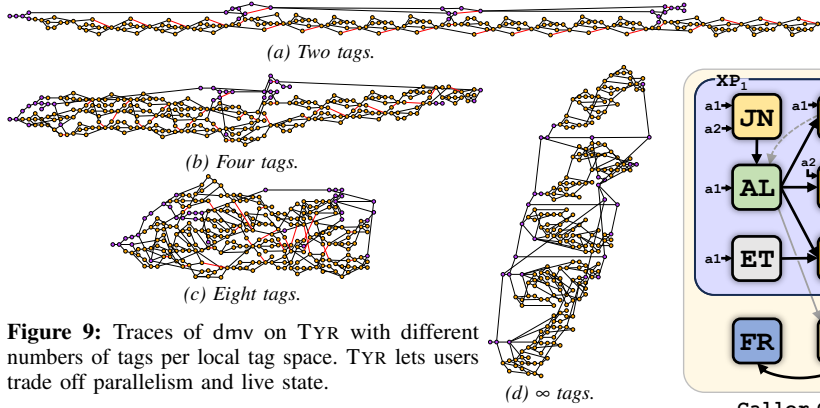
(a) Two tags.

(b) Four tags.

(c) Eight tags.

Figure 9: Traces of dmv on TYR with different numbers of tags per local tag space. TYR lets users trade off parallelism and live state.
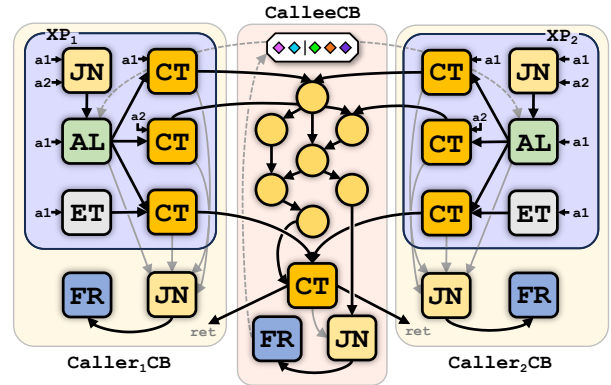
(d) ∞ tags.

Figure 10: TYR's concurrent-block linkage. All communication between concurrent blocks happens through changeTag instructions, ensuring tags never escape from their concurrent blocks.

so TYR adds an *unconditional* <t, ∅> output. And allocate's special firing rule means it can execute before ready arrives, so allocate generates <t, ∅> when the ready input is consumed. Correctly generating the free barrier for all cases was non-trivial, and required in any unordered dataflow architecture, yet we are unaware of any detailed discussion in prior work.

Finally, token synchronization in TYR requires two further instructions to manipulate tags, similar to TTDA [52]:

extractTag returns a tag as data: <t, ∅> → <t, t>.

changeTag replaces the tag and, for the free barrier, produces a control token: (<t, t'>, <t, data>) → (<t', data>, <t, ∅>).

When the token's destination is not statically known (e.g., function pointer or return to arbitrary caller), changeTag must also route the token to a dynamic location (i.e., instruction pointer and operand index) [52]. Loops and local functions can route tokens with ordinary control flow (i.e., steers).

B. Putting it all together: Concurrent block linkage in TYR

Fig. 10 shows how TYR uses these instructions to synchronize tokens from two transfer points into a single concurrent block. The transfer points are identical and share the block's free list. Each transfer point has an allocate (AL), which feeds *n* changeTag (CT) instructions to pass arguments to the child block [52]: The first are the parent's tag and return instruction, and the remaining are the program's arguments. The allocate is requested by the arrival of the first argument (a1),[6] and it is ready when all arguments arrive (see the join above the allocate). Note that, if multiple tags are available, the context starts executing as soon as a1 arrives, even if a2 has not yet arrived.

The context then executes normally, using the tag allocated by AL. Any exiting transfer points use changeTag to reset the tokens' tags and route them back to the parent context. Finally, the callee's join waits until all instructions have completed, whereupon the tag is freed back to the child's local tag space.

C. Compiling to TYR

TYR's compiles programs with LLVM [42] into the UDIR [2] intermediate representation, and then lowers UDIR to TYR. UDIR is a high-level dataflow IR that converts imperative

---

[6]The first is arbitrarily chosen; any argument suffices.

code (C/C++) into dataflow graphs, e.g., transforming branches into steers and memory-ordering dependencies into data dependencies. UDIR marks the boundaries of concurrent blocks with abstract enter and exit instructions, which can be lowered to many dataflow ISAs. We implement compiler passes to transform UDIR into TYR by converting enter and exit into the concurrent-block linkage shown in Fig. 10.

D. Trading off parallelism and live state

TYR gives architects a new knob to trade off parallelism and locality: the size of the local tag space. Fig. 9 demonstrates this feature by varying the number of tags per local tag space in dmv. With unlimited tags (Fig. 9d), TYR behaves identically to naïve unordered dataflow. However, TYR can tune the size of the tag space (Fig. 9a-c) to tame parallelism and save state. Even with few tags (e.g., eight in Fig. 9c), TYR achieves much higher parallelism than other architectures (Sec. II).

Local tag spaces unlock additional opportunities for fine-grained resource allocation. By *independently* sizing local tag spaces of *different program regions*, TYR can selectively control parallelism across a program. For instance, TYR can aggressively parallelize "hot code" (e.g., an affine innermost loop) by increasing its tags, and can reduce tags for outer-loops that stall waiting for earlier work to finish (Sec. VII-E).

V. DEADLOCK FREEDOM AND BOUNDED STATE

TYR achieves high parallelism with bounded state, unlike prior unordered dataflow architectures that produce an unbounded number of live tokens. Limiting the tag space in prior architectures can lead to deadlock when all tags are allocated to stalled work. Fig. 11 shows a simulation trace of dmv deadlocking on a naïve unordered dataflow architecture with 8 tags. As the system eagerly explores parallel work, all of the tags are allocated to outer loop iterations, so that the *first* iteration of the inner loop cannot finish because no tag is available for the second. Since outer-loop iterations are waiting for the inner loop to finish, the system deadlocks.

We now prove that TYR is deadlock-free and bounds live state. Intuitively, our proof shows that small pieces of TYR
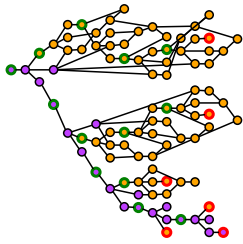
Figure 11: Deadlock seen in simulation: Unordered dataflow deadlocks on dmv with 8 tags. Nodes with green edges are successful tag allocations, while nodes with red edges are tag allocations which were pending when the system deadlocked. The number of tags required to successfully complete grows quickly with input size.

programs are deadlock-free and then composes them into larger, deadlock-free programs. Without loss of generality, we assume that cycles in the call graph (i.e., general recursion) have been transformed into tail recursion with an explicitly managed stack. This approach moves the unboundable state of general recursion from dataflow tokens into memory, but introduces memory ordering that may limit parallelism. We discuss an alternate strategy for recursion in Sec. VIII-B.

A. Proof of deadlock freedom

We first consider a program consisting of a single concurrent block, and then build up towards more complex programs.

Definition. *A* leaf block *is a concurrent block that does not contain any calls to other concurrent blocks.*

Assumption 1. *Once inputs are ready, a leaf block will eventually produce all outputs and free its tag.*

In other words, TYR's compiler is correct.

Lemma 1. *Leaf blocks are deadlock-free in TYR.*

Proof: For programs consisting of a single leaf block, the lemma follows immediately from Assumption 1. Concurrent blocks are DAGs, so tokens flow through the block in finite time, eventually reaching the output and the free.

By the same argument, for arbitrary programs, we only need to show that a ready context for the block will eventually receive a tag. TYR only allocates the last tag for a concurrent block to a ready context, so there is always either *(i)* an available tag or *(ii)* a ready context already executing, which will finish. Thus, if a tag is not immediately available, one will become available eventually, guaranteeing forward progress. ∎

Definition. *A* tail-leaf block *is a concurrent block that contains exactly one call, which is tail-recursive.*

Lemma 2. *Tail-leaf blocks are deadlock-free in TYR.*

Proof: A tail-leaf block context is guaranteed to complete so long as it can allocate a tag for its child context (i.e., the next loop iteration). TYR's allocate instruction ensures that at least one tag in a tail-leaf will always be available for the tail-recursive self edge by not allowing an external (i.e., not-tail-recursive) allocate to consume the last tag (Sec. IV-A). Thus, a tag will eventually be available for any tail-recursive child call, so by the same argument in Lemma 1, ready tail-recursive contexts will eventually complete. ∎

Theorem 1. *TYR is deadlock-free.*

Intuitively, deadlock-free concurrent blocks should compose to make deadlock-free programs. As long as a concurrent block's children do not deadlock, and it cannot internally deadlock, it must finish. Arbitrarily large programs can be proven deadlock-free by induction.

Proof: Per above discussion, the program is in a tail-recursive form. The call graph is, thus, a rooted DAG where the root is the entry point of the program (e.g., main). We will show by induction that every node in the graph is deadlock-free.

Base case: Consider a leaf node in the call graph. This node is either a leaf block or a tail-leaf block. By Lemma 1 or 2, it is deadlock-free.

Inductive step: Consider a concurrent block with calls to deadlock-free concurrent blocks. A context in this concurrent block will eventually produce all the inputs for any transfer point it contains. Because these callees are known to be deadlock-free, they will eventually produce return values. Once all return values have been received, Assumption 1 ensures that the context will finish and free its tag.

Thus every concurrent block in the call graph is deadlock-free, including the root, as is the entire program. ∎

TYR's local tag spaces are necessary for deadlock freedom. Note that both lemmas rely on TYR having local tag spaces for each concurrent block. Without local tag spaces, it is not possible to guarantee that a tag eventually becomes available for a ready context, as contention for tags across blocks can always starve a transfer point. Prior unordered dataflow architectures cannot even ensure that leaf blocks are deadlock-free.

TYR builds on prior deadlock-avoidance work. It is well known in networking that cycles present deadlock hazards, and the same is true in dataflow machines. TYR takes special care to handle loops by reserving one free tag to only be used by tokens already in the loop. Tokens waiting to be injected into the loop must wait for multiple available tags. This is similar to how bubble flow control prevents deadlock, where routing nodes refuse to accept packets from outside of a cycle in order to ensure the existence of a "bubble" in the cycle [10, 63].

B. Proof of bounded state

Prior unordered dataflow architectures have been unable to bound state because they are unable to bound the number of live tags. TYR can choose any number of tags greater than one and successfully execute all programs, which allows it to bound the total number of live tokens.

Theorem 2. *TYR bounds the number of live tokens.*

Proof: Consider a program with N static instructions and no more than M inputs per instruction. In all unordered (tagged) dataflow architectures, the number of tokens in flight with a single tag is bounded by the number of inputs to all instructions — there cannot be more than one token on an input with the same tag. Thus, any limit on the number of live tags T ensures that there are never more than $T \cdot N \cdot M$ live tokens. ∎
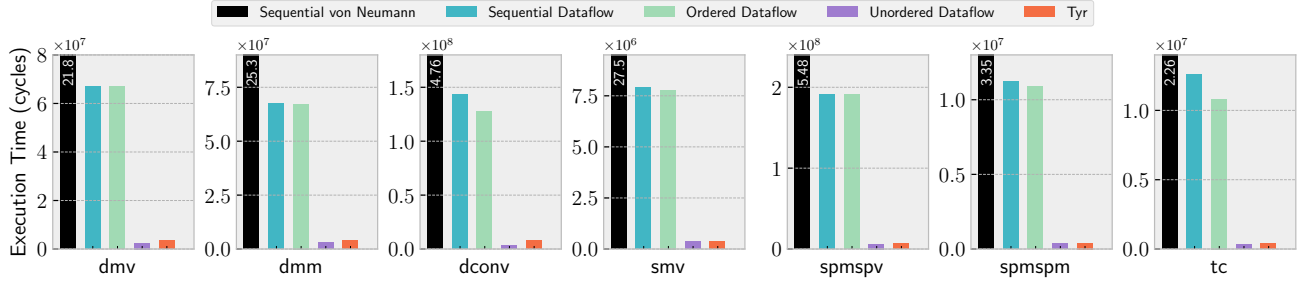
1192

Figure 12: Execution time across all apps and systems. TYR vastly outperforms von Neumann, sequential dataflow, and ordered dataflow, and is nearly identical to unordered dataflow.

Note this bound applies to all tagged dataflow architectures. But, unlike TYR, prior unordered dataflow is unable to bound the number of tags T for fear of deadlock. Thus, there is no bound on the total number of live tokens in these architectures.

VI. EXPERIMENTAL METHODOLOGY

Applications. We evaluate TYR on seven common, highly parallel benchmarks (Table II). These benchmarks were chosen to cover a variety of code patterns. The dense applications feature regular computation and simple control flow, while the sparse applications contain more complex, data-dependent control flow. They were compiled from unmodified C to DFGs for all dataflow architectures (including TYR's ISA) using UDIR (Sec. IV-C). Input sizes were chosen such that all systems executed between 50M and 1B dynamic instructions. TYR's advantage grows with increasing input size.

Dense applications were run on random inputs, as their execution is not data-dependent. smv and spmspv were run on real-world data from SuiteSparse [41]: smv on DNVS/trdheim [20,21] and spmspv on a subset of DIMACS10/M6 [5,21]. tc was run on a navigable small world graph [83].

Table II: Applications and their input sizes.

Application	Parameters
Dense matrix-vector (dmv)	Size: 4,096×4,096
Dense matrix-matrix (dmm)	Size: 256×256
Dense convolution (dconv)	Image: 512×512, filter: 11×11
Sparse matrix-vector (smv)	Size: 22,098×22,098, Non-zeros: 1,935,324
Sparse matrix-sparse vector (spmspv)	Size: 32,276×32,276, Matrix non-zeros: 74,482, Vector non-zeros: 1,638
Sparse matrix-sparse matrix (spmspm)	Size: 256×256, density: 5%
Triangle counting (tc)	Nodes: 16,384, edges: 206,107

Systems. We compare TYR against: *(i)* sequential von Neumann, *(ii)* sequential dataflow, *(iii)* ordered dataflow, and *(iv)* unordered dataflow (Sec. II-C). The size of the tag space(s) differs across architectures. TYR's local tag spaces each contain 64 tags. To ensure completion, unordered and sequential dataflow runs with unlimited tags. Ordered dataflow's FIFOs contain up to 4 tokens, which empirically minimizes peak state with minimal loss in performance [29].

Simulation. We evaluate TYR using an in-house simulator, which directly executes compiled DFGs. We model idealized execution to explore the ISA tradeoffs of dataflow architectures. Simulation assumes single-cycle execution for all instructions. We deliberately refrain from detailed microarchitectural timings, as doing so would risk clouding the ISA-level tradeoffs explored in this paper behind inessential microarchitectural assumptions. The systems can execute up to 128 instructions per cycle, including multiple instances of the same static instruction. We study sensitivity to these parameters below.

Metrics. We compare architectures on their parallelism and locality. We use execution time and IPC as measures of parallelism. Execution time is simply the number of cycles it takes to run a benchmark from start to finish. IPC and the number of live tokens are sampled each cycle during simulation.

VII. EVALUATION

We evaluate TYR to show that it is near-optimal in both parallelism and locality. TYR achieves similar performance to unordered dataflow with orders-of-magnitude less state.

A. TYR is fast

Fig. 12 shows the execution time for every app on each system. By gmean, TYR is 68× faster vs. vN, 22.7× vs. sequential dataflow, 21.7× vs. ordered, and 0.77× vs. unordered.

Fig. 13 explains the speedup by showing the distribution of IPC, plotted as a cumulative distribution function (CDF). The graph shows how frequently each system achieves a given IPC, so an ideal system would be an "⌐" shape. Unordered dataflow is nearly ideal, almost always saturating the machine's issue width, whereas sequential and ordered dataflow perform poorly, rarely executing more than ten instructions per cycle. vN is the worst, always executing one instruction per cycle. TYR's bounded tags slightly reduce IPC vs. unordered, but it still vastly outperforms vN, sequential dataflow, and ordered dataflow.

B. TYR reduces live state

Fig. 14 plots the number of live tokens during program execution on each system. Each bar shows both the maximum (unhatched) and mean (hatched) the *y*-axis in log scale. TYR reduces peak state by 572.8× vs. unordered dataflow on average, reducing peak tokens from over 15 M to just over
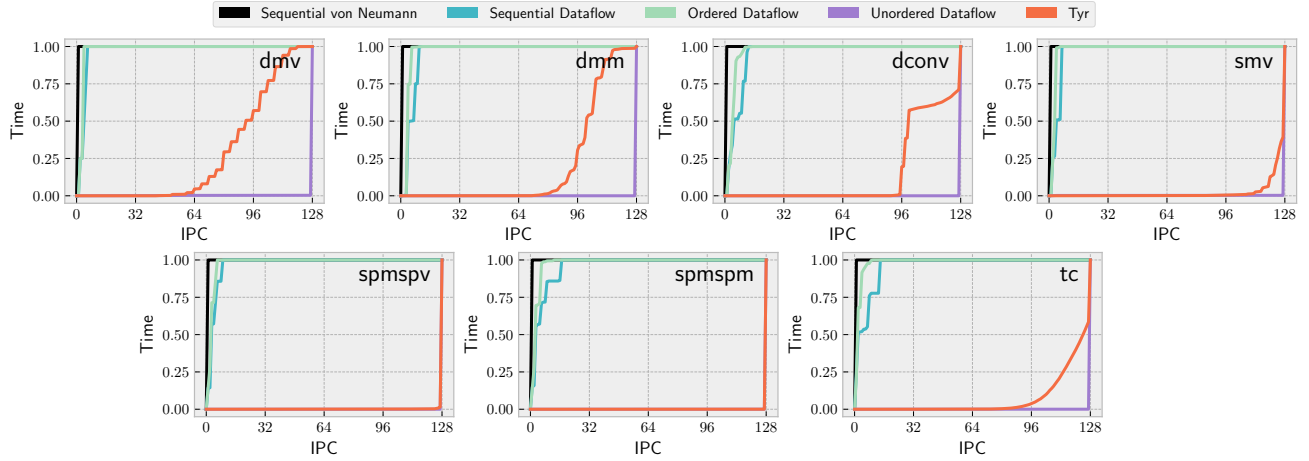
Figure 13: Cumulative distribution function (CDF) of the measured IPC of each system across all apps. TYR achieves high IPC, sometimes identical to unordered dataflow (e.g. spmspv and spmspm), but with much less state (Fig. 14). The other architectures run at much lower IPC.
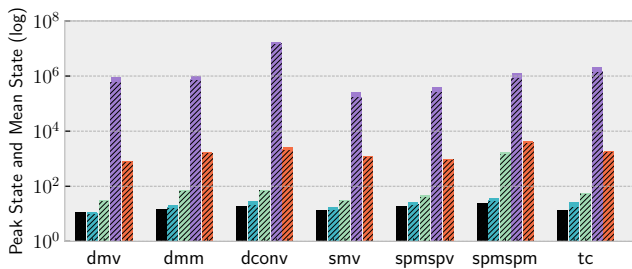

Figure 14: State during program execution, measured as number of live tokens (log scale). Each bar height shows the peak and the hatched region shows the average. TYR achieves peak state orders-of-magnitude lower than unordered dataflow while maintaining high performance (Fig. 12).

Figure 15: Execution time (top) and state (bottom) for TYR running dmv on a 512×512 inputs across issue widths from 16 to 512. TYR reduces execution time as issue width increases while maintaining consistent peak state.

2 K on dconv. TYR increases state vs. vN, sequential dataflow, and ordered dataflow (by 98.4×, 136× and 23×), but this is tunable in TYR (see below) and all are well within the constraints of a hardware implementation.

Comparing maximum and mean state, TYR is much closer to its peak on average, indicating that TYR is more consistently busy and achieves better resource utilization.

Fig. 2 (on page 1) shows a trace of state over time while executing spmspm. Unordered dataflow increases state exponentially as it greedily expands work and then slowly drains it at as the machine instruction width permits. By contrast, TYR's state quickly reaches its peak as local tag spaces are emptied and tag allocation is intentionally stalled, and TYR finishes in nearly the same execution time. Sequential and ordered dataflow also quickly reach their peak state, but thereafter are bottlenecked by a lack of parallelism and so their performance suffers.

C. TYR is scalable

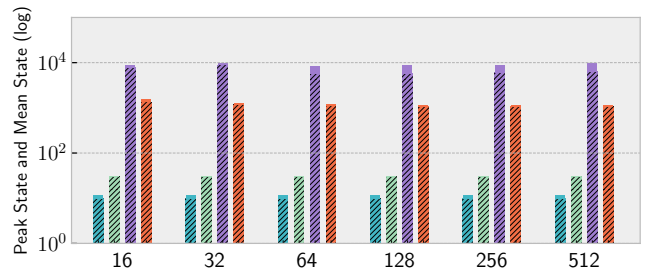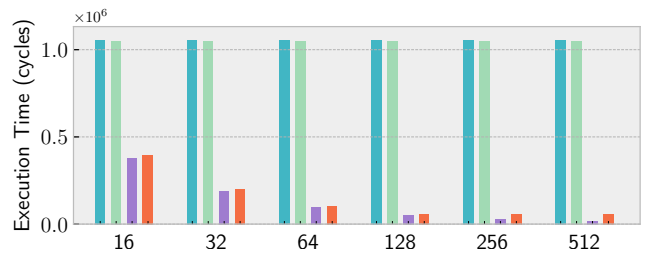Fig. 15 compares systems' performance and live state on spmspm while varying issue width. As expected, unordered dataflow and TYR see reduced performance benefits on narrow issue-width machines. Unordered dataflow achieves 15× speedup vs. sequential dataflow, and TYR achieves 10×. Unordered dataflow and TYR's performance improves steadily as issue width increases, but sequential and ordered dataflow see negligible gains due to their limited parallelism. The number of live tokens is fairly insensitive to issue width.

D. TYR's parallelism can be tuned to match system resources

Fig. 16 plots a trace of live tokens over time on spmspm in TYR as tag width varies from 2 to 512 tags per concurrent block. (Other applications show similar results.) TYR completes successfully, even with only two tags per concurrent block. Increasing the number of tags allows TYR to expand parallelism

aggressively and therefore achieve better performance, reducing the width of the trace. With issue width of 128, performance peaks at around t = 64 tags per block. On the same program, unordered dataflow would require multiple orders-of-magnitude more tags to complete without deadlocking.
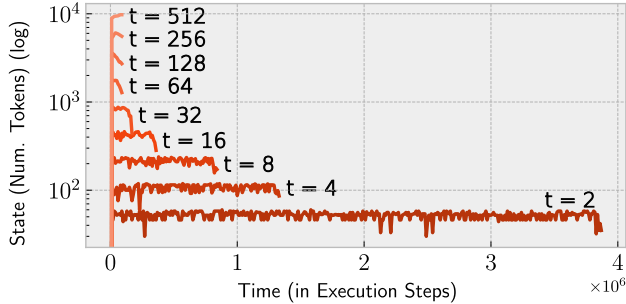

Figure 16: State vs. execution time for TYR across tag widths (t) from 2 to 512 on spmspm. TYR correctly executes even with 2 tags. Execution time improves until t = 64, showing that TYR can achieve high parallelism with few tags.

The number of tags required to extract maximum performance varies with issue width. Fig. 17a shows spmspv on a 128×128 matrix, measuring IPC as both issue width and tags per concurrent block are varied. Performance is bottlenecked when either issue width or tag spaces are too small. Fig. 17b shows the corresponding peak state, which increases as more tags are added. For a given issue width, performance increases with tags until the number of tags is half the issue width. Fig. 17c plots IPC and peak state with tags fixed at half issue width. At an issue width of 128, parallelism is saturated, and both performance and peak state stop increasing. Interestingly, once issue width is plentiful, peak state actually drops slightly because tokens no longer stall.

### E. TYR can tune parallelism across program regions

Thus far, we have assumed that all concurrent blocks should have equal-size local tag spaces, but this is not a requirement in TYR. Compilers or runtime systems could scale the tag space in each block to trade off parallelism and state. In some cases, reducing the number of tags can lower peak state without adversely affecting performance.

Fig. 18 plots a trace of live tokens over time on dmm for a baseline with 64 tags per concurrent block, and an optimized version with only 8 tags in the outermost loop. Because there is so much inner-loop parallelism available, few outer-loop iterations are needed to saturate the machine. Reducing the unnecessary outer-loop iterations reduces state by 28.5% without negatively affecting parallelism or performance.
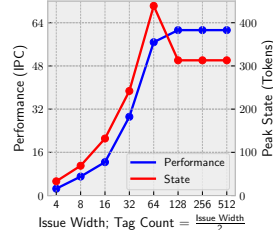
VIII. DISCUSSION

A. Roads not traveled

Prior work has recognized the parallelism explosion problem and proposed techniques to mitigate it. Generally, prior dataflow architectures observe that parallelism explosion makes tagging unscalable and expensive, and they try to mitigate this cost by
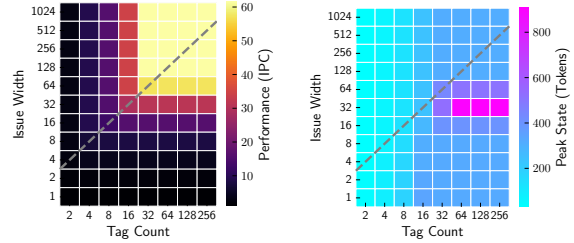
(a) TYR requires both sufficient issue width and tags to achieve peak performance.

(b) TYR's peak state increases with more tags but not with issue width.

Figure 17: IPC and live state in TYR on spmspv. High IPC requires many tags, which increases peak state. Sensible systems will scale issue width and tag count together, as shown by the gray line in *(a)* and *(b)*. In that regime, performance and peak state scale proportionally until all available parallelism is exploited.

(c) Performance vs. peak state when scaling tag count with issue width.
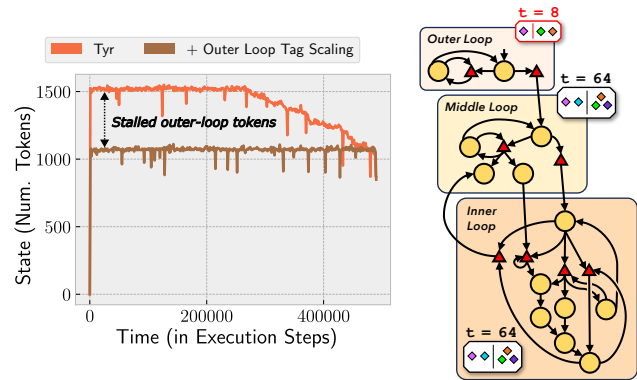

Figure 18: State vs. execution time for TYR with and without limiting the number of tags for the outermost loop in dmm. TYR enables the system to selectively scale parallelism across program regions. Reducing tags in the outermost loop from 64 to 8 reduces peak state by 28.5% with minimal performance impact.

reducing or simplifying tag management. Before arriving at TYR, we first tried building on these existing techniques, but we concluded that they left the root problem unresolved. Ultimately, *tagging is expensive because there are too many tags*. We thus realized that a true solution to parallelism explosion must cut the problem off at the source, i.e., by preventing unbounded parallelism from being generated in the first place.

The MIT Tagged Token Dataflow Architecture (TTDA). As discussed in Sec. II, TTDA introduces a *k*-bounding technique for affine loops that allocates a contiguous *block* of tags to the loop, instead of a tag per loop iteration. Within the loop tags are allocated simply by incrementing, and only *k* iterations can be live at a time. Culler's PhD dissertation [16] extended this approach to nested loops, reserving k_1 tags for innermost

loops, k_2 tags for the next loop, etc., and analyzed the impact on state for different $\{k_i\}$s.

k-bounding both bounds parallelism and simplifies tagging, and it is a highly effective solution to parallelism explosion — but only for nested affine loops. Naïvely applying the technique to other program constructs leads to deadlock.

In a sense, TYR generalizes k-bounding. However, we initially attempted this by generalizing k-bounding's block-allocation, *not* its parallelism bounds. Our "TTDA+" scheme supports functions and arbitrary loops by, e.g., allowing callers to donate a partially used block to a callee. Instructions only need to allocate a new tag once the entire block is exhausted. TTDA+ amortizes tag management and thereby improves scalability, but we eventually realized it is a bandaid. It is far more important to reduce the number of tags by *bounding parallelism*.

Monsoon. Monsoon introduces the explicit token store, where space is allocated in units of *activation frames* that store tokens for entire function bodies. Tags identify the activation frame *and* include a frame offset that is determined statically for each instruction, similar to conventional registers. When a function executes, most instructions can just use the frame offset and require no associative tag match.

Like Monsoon, TYR breaks programs into logical regions and manages tags per region. (TYR's concurrent blocks are analogous to Monsoon's activation frames.) However, whereas TYR leverages this structure to safely bound parallelism, Monsoon uses it to amortize tag management. In that sense, Monsoon is more similar to TTDA+ than Tyr.

Monsoon and TYR are complementary. We envision that an efficient microarchitecture for TYR would implement an explicit token store for concurrent blocks to amortize tag management, like Monsoon (see below).

WaveScalar. WaveScalar simplifies all tag management to increments (i.e., the WaveAdvance instruction) and, by limiting execution to a hyperblock in the vN order, also limits parallelism. WaveScalar thus seems like a promising solution to parallelism explosion.

However, WaveScalar must keep tags up-to-date for *all* live values, requiring an absurdly large number of WaveAdvances.[7] Follow-on work proposed *far-hoisting* to "fast-forward" values at the end of a loop [61]. But, like TTDA's k-bounding, far-hoisting is limited to simple loops.

We investigated a "WaveScalar+" architecture that generalized far-hoisting to nested loops and function calls. WaveScalar+ represents the wave number as an input to each code block and provides simple, composable rules for how to update the wave number across program constructs. We developed compilation passes akin to constant-folding that could completely eliminate WaveAdvances in many cases, e.g., on loops with known trip counts. By removing the WaveAdvance dependencies,

WaveScalar+ can also significantly improve parallelism, i.e., by executing multiple hyperblocks at once.

However, we ultimately moved away from sequential dataflow architectures like WaveScalar. These architectures neither strictly bound live state nor exploit all available parallelism, as they are limited to the vN block-order. Moreover, although WaveScalar+ reduced WaveAdvances substantially, their overhead remained significant.

Functional languages $\nLeftrightarrow$ parallelism explosion. TTDA and Monsoon were designed for functional dataflow languages like Id [51]. The conventional wisdom was that functional languages were uniquely prone to parallelism explosion and deadlock. Unlike imperative languages, their declarative nature allowed unintuitive iterative structures like:

```
def fib(N):
    for i in 1 ... N:
        if i > N-1:
            x[i] = 1
        else:
            x[i] = x[i+1] + x[i+2]
```

where, in order to make forward progress, the *last* iteration of the loop must execute *first*.[8] Hence, by bounding parallelism to $k < N$ iterations, the program deadlocks [52].

This paper demonstrates that parallelism explosion and deadlock are, in fact, independent of language choice. First, we have shown that parallelism explosion can lead to deadlock even in an imperative program (Fig. 11). TYR provably avoids these deadlocks.

Second, we observe that these "backwards executions" are not, in fact, unique to functional languages. They naturally arise in imperative languages through recursion:

```
def fib(N):
    if N <= 2:
        return 1
    else:
        return fib(N-1) + fib(N-2)
```

Here, the stack depth corresponds to k, precluding correct execution if $k < N$. One can code in a pure functional style within an imperative language like C, even emulating "pure dataflow" constructs like write-once memory (Id's I-structures). For such programs, the challenge is to support recursion with high parallelism, but without exploding token storage.

B. The path ahead

Parallel general recursion in TYR. Theorem 1 begins by transforming general recursion into tail recursion with an explicitly managed stack. This transformation is necessary because general recursion is inherently unbounded, requiring memory somewhere (i.e., the stack) for parent contexts in any implementation. With general recursion, our goal is not to bound state to some finite value, but to limit its growth (e.g., to call-tree depth, like vN), which a stack-based implementation does. Unfortunately, a stack-based implementation precludes parallelism between sibling calls because they share the same stack frame. Ideally, an implementation would enable parallelism across sibling calls while still bounding live state.

To improve parallelism, one could allocate a tree of activation frames [52, 58], but, done naïvely, this reintroduces parallelism

---

[7]As originally proposed, WaveScalar cross-compiles from a vN ISA, where register spilling limits the number of live values. As a general compilation target, WaveAdvance overhead is prohibitive.

[8]In Id, the loads to x[i+1] and x[i+2] stall until a value is written.

explosion. In future work, we will investigate a solution inspired by work stealing [7,8,27,57] that expands parallelism when resources are idle, but prioritizes ongoing work to avoid parallelism explosion, limiting live state to $O(T) \times$ the serialized execution peak state, where T is the number of tags.

Critically, by building on TYR, the amount of *token state* required remains strictly bounded, whether using a stack or tree of activation frames. The state *in memory* (i.e., for the stack or activation frames) may grow without bound, if required by the program, but the number of *live tokens* still follows Theorem 2.

An efficient microarchitecture for TYR. This paper has explored the architectural tradeoffs between parallelism and state, but leaves a detailed implementation to future work. Fortunately, TYR creates many opportunities to simplify tagged dataflow and improve its scalability.

Most importantly, TYR strictly limits the number of tags per concurrent block. TYR enables practical and efficient hardware to perform tagged dataflow execution, similar to an issue queue in an out-of-order superscalar. Moreover, execution is distributed across many concurrent blocks, enabling a clustered implementation with much smaller issue queues than a monolithic superscalar.

TYR's local tag spaces localize tag management to a small number of instructions, enabling distributed, scalable tag management in hardware. Given the large size of real programs, techniques must be developed to spill local tag spaces to and from memory, caching those used most frequently.

Finally, it is important to emphasize that TYR is an *architecture*, not a microarchitecture. Like other architectures, TYR's semantics admit many possible implementations. For example, we have described TYR as performing a tag match for every instruction, but TYR's concurrent blocks have no internal tag changes. TYR thus admits a hierarchical implementation where entire blocks are scheduled at once, like TRIPS [66], or where tags are matched only at the concurrent-block boundary, like Monsoon [58]. TYR also encodes all orderings as explicit dependencies in the DFG, but an implementation is free to speculate around dependencies (e.g., memory orderings or steers) to increase parallelism [23,73]. We intend to explore this large design space in future work.

IX. CONCLUSION

We have presented TYR, a dataflow architecture that is deadlock-free, general-purpose, highly parallel, and has bounded state. TYR exploits program structure to distribute tag allocation across *local tag spaces*, and introduces new tag-management instructions that guarantee forward progress. TYR compiles programs from unmodified C code, and our evaluation shows it nearly matches the performance of the best prior architecture with orders-of-magnitude less state. We have proven that TYR is deadlock-free and has bounded state. By defusing parallelism explosion, TYR unblocks the path for dataflow architectures in high-performance applications.

REFERENCES

[1] *The Poetic Edda*, Iceland, c. 1270.

[2] N. Agarwal, M. Fream, S. Ghosh, B. C.Schwedock, and N. Beckmann, "UDIR: Towards a Unified Compiler Framework for Reconfigurable Dataflow Architectures," in *IEEE Computer Architecture Letters*, 2023.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] Arvind and D. E. Culler, "Managing resources in a parallel machine," in *Proc. of the IFIP TC 10 Working Conference on Fifth Generation Computer Architectures*. NLD: North-Holland Publishing Co., 1986, p. 103–121.

[5] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, ser. Contemporary Mathematics, vol. 588. American Mathematical Society, 2013. [Online]. Available: http://dblp.uni-trier.de/db/conf/dimacs/dimacs2012.html

[6] 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

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *ACM SigPlan Notices*, vol. 30, no. 8, pp. 207–216, 1995.

[8] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, sep 1999. [Online]. Available: https://doi.org/10.1145/324133.324234

[9] E. Brunvand, D. Kline, and A. K. Jones, "Dark silicon considered harmful: A case for truly green computing," in *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, 2018, pp. 1–8.

[10] C. Carrion, C. Izu, J. Gregorio, F. Vallejo, and R. Beivide, "Ghost packets: a deadlock-free solution for k-ary n-cube networks," in *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing - PDP '98 -*, 1998, pp. 133–139.

[11] G. Chakrabarti, V. Grover, B. Aarts, X. Kong, M. Kudlur, Y. Lin, J. Marathe, M. Murphy, and J.-Z. Wang, "Cuda: Compiling and optimizing for a gpu platform," *Procedia Computer Science*, vol. 9, pp. 1910–1919, 2012, proceedings of the International Conference on Computational Science, ICCS 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050912003304

[12] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[13] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[14] 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.

[15] 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.

[16] D. E. Culler, "Managing parallelism and resources in scientific dataflow programs," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1989. [Online]. Available: https://hdl.handle.net/1721.1/14511

[17] D. E. Culler *et al.*, "Resource requirements of dataflow programs," in *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2, 1988.

[18] 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

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

[20] C. Damhaug, "Dnvs/trdheim," 2004.

[21] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," vol. 38, no. 1. New York, NY, USA: Association for Computing Machinery, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[22] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ISCA*, 1975.

[23] R. Desikan, S. Sethumadhavan, D. Burger, and S. W. Keckler, "Scalable selective re-execution for edge architectures," *SIGPLAN Not.*, vol. 39, no. 11, p. 120–132, oct 2004. [Online]. Available: https://doi.org/10.1145/1037187.1024408

[24] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, mar 1968. [Online]. Available: https://doi.org/10.1145/362929.362947

[25] 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.

[26] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. ACM, 2011.

[27] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, p. 212–223, may 1998. [Online]. Available: https://doi.org/10.1145/277652.277725

[28] 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 *ISCA*, 2021.

[29] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "Riptide: A programmable, energy-minimal dataflow compiler and architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 546–564.

[30] 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.

[31] 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.

[32] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–12.

[33] 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," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 854–867.

[34] J. Gurd, "The manchester dataflow machine," *Computer Physics Communications*, vol. 37, no. 1, pp. 49–62, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0010465585901353

[35] M. Hill and V. J. Reddi, "Gables: A roofline model for mobile socs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 317–330.

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

[37] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

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

[39] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, 2011.

[40] 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

[41] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The suitesparse

matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019. [Online]. Available: https://doi.org/10.21105/joss.01244

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO*, 2021.

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

[50] R. S. Nikhil, "Can dataflow subsume von neumann computing?" in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ser. ISCA '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 262–272. [Online]. Available: https://doi.org/10.1145/74925.74955

[51] R. S. Nikhil, "Id reference manual, version 90.1," MIT Lab. for Comput. Sci., 545 Technology Square, Cambridge, MA 02139, CSG-Memo 284-2, July 1991.

[52] R. S. Nikhil *et al.*, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on computers*, 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, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ISCA 44*, 2017.

[55] 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.

[56] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, 2015.

[57] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on gpus," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 181–192.

[58] 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

[59] 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.

[60] 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

[61] A. Petersen, M. Mercaldi, S. Swanson, A. Putnam, A. Schwerin, M. Oskin, and S. Eggers, "Reducing control overhead in dataflow architectures," in *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 182–191.

[62] 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.

[63] V. Puente, C. Izu, R. Beivide, J. Gregorio, F. Vallejo, and J. Prellezo, "The adaptive bubble router," *Journal of Parallel and Distributed Computing*, vol. 61, no. 9, pp. 1180–1208, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731501917460

[64] Riscv, "riscv-v-spec," Apr 2019. [Online]. Available: https://github.com/riscv/riscv-v-spec

[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] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement," in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (Proc. ISCA-49)*, 2022.

[69] N. Serafin, S. Ghosh, H. Desai, N. Beckmann, and B. Lucia, "Pipestitch: An energy-minimal dataflow architecture with lightweight threads," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1409–1422. [Online]. Available: https://doi.org/10.1145/3613424.3614283

[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] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker, "The ARM scalable vector extension," *CoRR*, vol. abs/1803.06185, 2018. [Online]. Available: http://arxiv.org/abs/1803.06185

[72] S. Sturluson, *The Prose Edda*, Iceland, c. 1220.

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

[74] 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.

[75] 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.

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

[77] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 412–425.

[78] M. Vilim, A. Rucker, and K. Olukotun, "Aurochs: An architecture for dataflow threads," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 402–415.

[79] 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.

[80] D. Voitsechov and Y. Etsion, "Control flow coalescing on a hybrid dataflow/von neumann gpgpu," in *MICRO 48*, 2015.

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

[82] B. Wang, M. Karunarathne, A. Kulkarni, T. Mitra, and L.-S. Peh, "Hycube: A 0.9 v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications," in *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2019, pp. 133–136.

[83] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998. [Online]. Available: https://doi.org/10.1038/30918

[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