DiAG: A Dataflow-Inspired Architecture for General-Purpose Processors

Dong Kai Wang
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
dwang47@illinois.edu

Nam Sung Kim
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
nskim@illinois.edu

ABSTRACT

The end of Dennard scaling and decline of Moore's law has prompted the proliferation of hardware accelerators for a wide range of application domains. Yet, at the dawn of an era of specialized computing, left behind the trend is the general-purpose processor that is still most easily programmed and widely used but has seen incremental changes for decades. This work uses an accelerator-inspired approach to rethink CPU microarchitecture to improve its energy efficiency while retaining its generality. We propose DiAG, a dataflow-based general-purpose processor architecture that can minimize latency by exploiting instruction-level parallelism or maximize throughput by exploiting data-level parallelism. DiAG is designed to support any RISC-like instruction set without explicitly requiring specialized languages, libraries, or compilers. Central to this architecture is the abstraction of the register file as register 'lanes' that allow implicit construction of the program's dataflow graph in hardware. At the cost of increased area, DiAG offers three main benefits over conventional out-of-order microarchitectures: reduced front-end overhead, efficient instruction reuse, and thread-level pipelining. We implement a DiAG prototype that supports the RISC-V ISA in SystemVerilog and evaluate its performance, power consumption, and area with EDA tools. In the tested Rodinia and SPEC CPU2017 benchmarks, DiAG configured with 512 PEs achieves a 1.18× speedup and 1.63× improvement in energy efficiency against an aggressive out-of-order CPU baseline.

CCS CONCEPTS

• **Computer systems organization → Data flow architectures**; **Superscalar architectures**.

KEYWORDS

dataflow architecture, general-purpose, parallelism

ACM Reference Format:
Dong Kai Wang and Nam Sung Kim. 2021. DiAG: A Dataflow-Inspired Architecture for General-Purpose Processors. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446703

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446703

1 INTRODUCTION

At the twilight of transistor scaling, each technology generation will still shrink the transistor's dimensions but not so much its supply voltage. This disparity leads to a circuit utilization wall, or 'dark silicon' [6], resulting in the inevitable consequence that only a fraction of the circuit can be powered-on under the same power budget. To combat this issue, a wide range of application-specific hardware accelerators have been researched [2, 3, 8, 26], developed, and adopted by industry [16] to populate under-utilized die area. Today, the industry is diverse with all types of processing hardware targeting different markets and use cases. One end of the spectrum is populated by highly efficient but rigid ASICs dedicated for specific application domains. On the other end lies power hungry and inefficient, but easily programmed general-purpose processors. Between these extremes, we find middle ground architectures such as GPUs, DSPs, FPGAs, and the like occupying various points on the flexibility-efficiency trade-off line. This work seeks to retain the generality of traditional CPUs and significantly improve energy efficiency by applying accelerator design principles to processor microarchitecture.

To do so, we first identify design factors that make accelerators more efficient. Unless algorithm-level changes are involved, a program running on any hardware performs the same computations (additions, divisions, etc.) except control related operations such as branches and loop variables. An ideal processor should spend nearly all of its power on functional units performing computations and waste as little as possible on control and data movement. This is not the case for a modern out-of-order CPU that houses complex front-end control logic consisting of instruction fetching, decoding, register renaming, etc. that consume a significant percentage of total power to the point that computation logic consume as low as 3% [19]. However, this sacrifice is necessary to ensure correct execution of any application using its supported instruction set. Consider a simple example to illustrate this trade-off: we can reduce the cost of instruction decoding by reducing the number of opcodes in the ISA, but doing so also reduces generality since fewer opcodes means fewer operations are supported.

Three techniques: instruction reuse, data-level parallelism, and optimized data movement contribute to a domain-specific accelerator's superior efficiency. Instruction reuse [34] refers to repeated execution of the same program instructions, such as iterations over the same loop or repeatedly calling the same function. General-purpose CPUs are largely unaware of instruction reuse as every instruction must always be fetched, issued, and checked for dependencies whether it is first encountered or repeated in a loop iteration, only benefiting from locality in caches or pipeline shortcuts (e.g., loop-stream detectors). This is a major disadvantage as

many programs tend to spend most of their execution time in loops. In contrast, instruction reuse is inherent to accelerators since they are designed for applications that are reused enough to at least justify its existence. Accelerators eliminate instruction control overheads by having a handcrafted datapath for the application directly imprinted in hardware. With this in mind, we propose a hybrid design: a dataflow architecture that can fetch, decode, and execute instructions normally but, as it executes, also constructs a hardware datapath that replicates the program's dataflow graph. This datapath can then be reused when a loop is encountered and bypasses the costly front-end that would otherwise have to re-fetch and decode the same instructions again.

Exploiting parallelism in a program is crucial to achieving high performance. While CPUs can exploit instruction-level parallelism (ILP), they do so at the cost of control complexity through renaming registers and reordering instructions. Multicores exploit thread-level parallelism (TLP), but each core still suffers the same inefficiencies. On the hand, accelerators can effectively exploit data-level parallelism (DLP) by pipelining entire functions and programs. Since instruction control is nonexistent or centralized, each of its pipeline stage is solely a compute operation, i.e., only the execute stage of the instruction. We apply the same concept to pipeline our dynamic datapaths by grouping fixed sets of instruction operations to form pipeline stages and, ideally, scaling compute throughput with the number of functional units.

We propose DiAG, a dataflow-based general-purpose microarchitecture that can support typical RISC ISAs without modification. DiAG dynamically constructs an instruction dataflow graph (DFG) of the program as it executes that is similar to a mini-accelerator datapath. This graph exposes the locally available ILP within the program, allowing multiple instructions to execute concurrently. With the DFG, we can eliminate significant control overhead especially if the program harbors instructions reuse. Furthermore, when DiAG encounters parallelizable parts of the program that meet certain criteria, we apply pipelining at the thread-level to improve hardware utilization. In this mode, DiAG's throughput efficiently scales with the number of functional units.

The main contributions of this paper are as follows:

- We propose DiAG, a dataflow-inspired general-purpose processor architecture that can dynamically construct a reusable execution datapath to exploit ILP, TLP, and DLP.
- Unlike conventional dataflow architectures, DiAG directly maps instructions to hardware in program order, allowing simple support for branches and interrupts. We extend past methods that dynamically construct restricted DFGs in hardware [12, 21, 30] to enable architecture composability, instruction reuse, and loop pipelining. DiAG is designed as a plug-and-play processor that can support regular ISAs without additional baggage.
- We implement a variant of this architecture to support the RISC-V 32-bit ISA and evaluate its performance by simulation and its power consumption and area properties with EDA tools. We additionally test a barebones DiAG prototype on a Xilinx FPGA board. Finally, we provide some optional ISA extensions to enable thread pipelining to boost DiAG's performance.

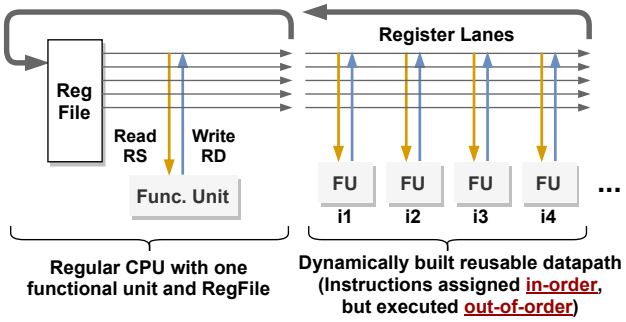


Figure 1: (*Left*) A single functional unit that reads and writes to a register file, essentially a single-issue in-order processor. (*Right*) DiAG extends the register file as register lanes that transport register values from one functional unit to the next.

Evaluation results using benchmarks from the Rodinia and SPEC CPU2017 benchmark suite show a 1.18× speedup in relative performance, and 1.63× improvement in energy efficiency on average against a 12-core 8-issue out-of-order CPU. The remainder of this paper is organized as follows: Section 2 is a high-level overview, Section 3 differentiates DiAG from past works on dataflow and superscalar architectures, Section 4 details how DiAG efficiently exploits ILP and DLP, Section 5 is a more detailed view of the DiAG architecture, Sections 6 and 7 evaluate our hardware implementation and performance, and finally Section 8 concludes the paper.

2 HIGH-LEVEL OVERVIEW

In theory, an ideal processor could consist of only structures that hold data and structures that process data. Such a processor built for one task can be handcrafted with the exact memory size and functional units needed. Though near-perfect efficiency is out of reach with generality, we propose an architecture that can: **1.** extract ILP and DLP with reduced control overhead, **2.** exploit dynamic instruction reuse, and **3.** maximize computation hardware utilization when possible.

At a glance, the DiAG architecture consists of a row of functional units and an interconnecting bus flowing through them. The interconnect, named register lanes, is designed to serve the combined roles of forwarding paths, the physical register file, and the reorder buffer in a traditional out-of-order processor. Each register is abstracted as a lane (wire bundle) that transports its value and status across functional units. In many ways, this is not fundamentally different from a typical dataflow processor equipped with a spatial array of functional units upon which the program's DFG can be mapped to. In DiAG however, instructions are assigned to functional units **in program order** but can execute **out-of-order** as soon as their operands from register lanes are available. The method of DFG construction in DiAG is closely related to CRIB [12] and earlier works that we discuss in the next section. During execution, a dynamic datapath arises as functional units are orderly loaded with program instructions. Since register lanes replace the register file, a restricted dataflow graph is implicitly formed as results from previous functional units are forwarded to the next. This construction

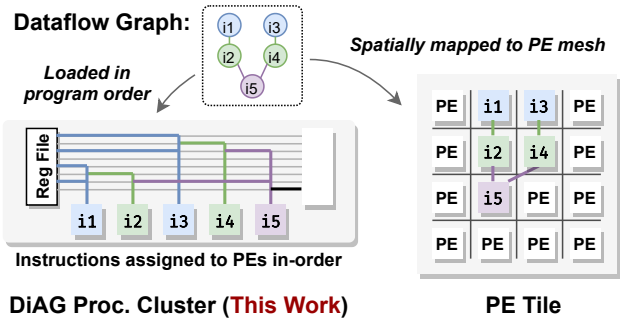


Figure 2: Mapping instructions to PEs. (*Left*) In DiAG, instructions are assigned in program order, register lanes form the DFG. (*Right*) Instructions are spatially mapped to a mesh of PEs, directly reflecting the DFG.

requires no reconfiguration because each functional unit simply loads its source operands from register lanes after its assigned instruction is decoded. Instructions can begin executing as soon as their source register lanes are valid, resolving any read-after-write (RAW) hazards. Figure 1 shows register lanes as an extension to the register file, note that a DiAG processor with only one functional unit is nearly identical to the back-end of an in-order single-issue CPU. By implicitly constructing the DFG, DiAG has performed the combined tasks of renaming registers, issuing, dispatching, and re-ordering instructions. We effectively reduce the complexity of the processor's front-end at the cost of increased hardware area for sparsely enabled functional units and register lanes. However, the DiAG architecture truly shines when the dynamically built datapath is reused across loop iterations. When a backward branch or jump is encountered, one or more parts of the datapath can be reused if the target address falls within the already constructed range. A reused datapath already has instructions loaded and decoded, and all data dependencies between instructions resolved, effectively leaving only the execute stage for each instruction. Finally, when we encounter a parallelizable loop that entirely fits in the DiAG processor, we can pipeline the datapath to further improve execution efficiency. By inserting pipeline registers between functional units, we can ideally scale DiAG's total throughput proportionally with the number of functional units.

3 RELATED WORK

DiAG uses a combination of many concepts and techniques from different areas of computer architecture such as dataflow computing, loop reuse, pipelining, composable architectures, etc. This section highlights similarities and differences between DiAG and relevant past works on dataflow architectures and superscalar techniques with similar themes.

3.1 Dataflow Architectures

Dataflow computing has been a subject of extensive research in computer architecture. While DiAG borrows many concepts from dataflow architectures, it strictly adheres to the von Neumann model of computing. Unlike most past works, DiAG can be used as a drop-in replacement for regular CPUs and supports generic RISC-like ISAs without requiring its own ecosystem of specialized compilers or languages. Thus, it is more appropriately considered a re-imagined out-of-order microarchitecture imbued with dataflow concepts.

3.1.1 The Dataflow Model. As its name implies, the dataflow computation model abstracts a program as a directed graph of operations, and data 'flows' internally through edges of the graph during execution [17]. Since nodes in the graph can perform their operations as soon as input operands are available, dataflow architectures can naturally exploit parallelism of operations.

3.1.2 Early Dataflow Architectures. Elegant on paper, many early implementations of dataflow architectures attempt to statically replicate the DFG in hardware. Examples of early architectures include DDM1 [4], the Hughes Dataflow Machine [37], and the MIT Dataflow Architecture [5]. However, pure dataflow architectures were limited in practicality since they could not easily support commonly used programming languages and data structures.

3.1.3 Hybrid von Neumann / Dataflow Architectures. Subsequent works [7, 11, 18, 22, 31, 39] combined dataflow techniques with von Neumann models of computation and memory. Architectures such as Tartan [28], TRIPS [32], and Conservation cores [38] also use a hybrid control flow model where the program's DFG is divided into subgraphs that are executed atomically. While DiAG's microarchitecture is inspired by the dataflow model, it differs from most past works in the following ways: **1.** Processing elements (PEs) are chained together in a line rather than arranged in 2D tiles. All PEs have assigned instructions regardless of the shape of the graph. **2.** Program instructions are assigned to PEs strictly in program order, and eventually 'commit' in-order. **3.** Register lanes form the interconnect between PEs, they dynamically construct a restricted dataflow graph and serve the same purpose as a reorder buffer.
DiAG addresses two important limitations of past works:
1. Granularity of control. Most dataflow architectures cannot fully handle control flow changes at the instruction level. They use compilers to break down a program into control-free sequences of code, e.g., 'blocks' in TRIPS [32] or 'waves' in Wavescalar [35]. These sequences are then mapped and executed block-wise in hardware. As a result, supporting an arbitrary branch instruction or precise interrupts is difficult to realize. DiAG does not decompose the program into subgraphs and handles all control flow changes at the instruction level. It supports precise interrupts and speculative execution fully (e.g., even if all instructions in a program are nested branches).
2. General compatibility. Most dataflow architectures require special instruction sets and/or compilers and/or software libraries to work with the hardware [10, 23, 29, 33, 35]. Thus, there is a high barrier to adoption as existing binaries for commonly used ISAs must all be recompiled to work on the platform. Furthermore, control limitations in **1.** only make it more difficult to support all application types. DiAG differs from past dataflow works in that instructions are mapped **in program order** but **execute out-of-order** as shown in Figure 2. This not only simplifies instruction control but also allows composability. We can view DiAG's DFG as a linearized version of the spatial DFG mapped to a tile of PEs, this will become very apparent in Section 4.

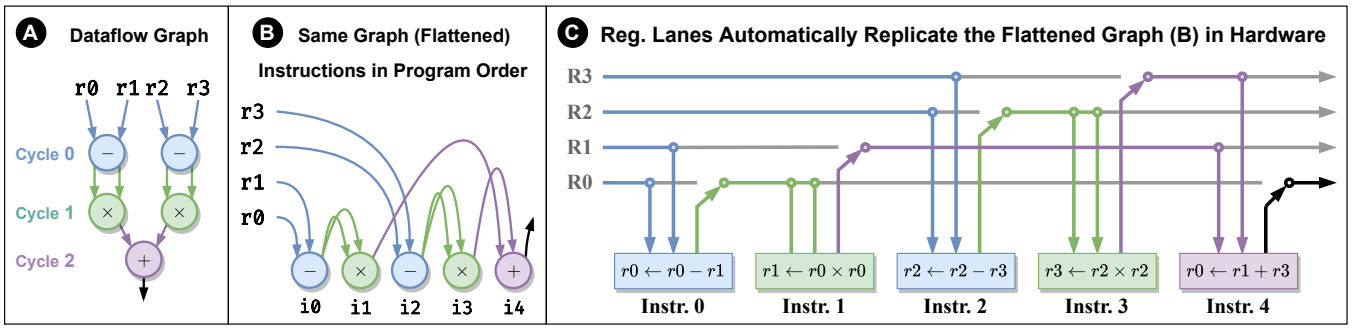


Figure 3: Dataflow execution of a program with five instructions. Instruction assigned to functional units in program order implicitly constructs the flattened dataflow graph of the program, i.e., the DFG in (A) and (B) are identical.

3.1.4 CGRAs. Coarse-grained reconfigurable architectures such as Piperench [9], DySER [10], and Stream-Dataflow [29] also utilize dataflow techniques, however CGRAs are generally intended to be co-processors or programmable accelerators, not a replacement for the main CPU. DySER, for example, is a dynamically configurable dataflow accelerator directly integrated into the processor pipeline that executes selected blocks of code in dataflow fashion like a large functional unit. As with the previous architectures, DySER uses a spatial grid of functional units and reconfigurable switches, similarly relying on the compiler to map the DFG. A main goal of DiAG is to be the central processor with full transparency and support for existing code.

3.2 Related Superscalar Techniques

Many aspects of DiAG are closely related to past works in microarchitectural design, superscalar techniques, and optimizations.

3.2.1 Out-of-Order Architectures. Out-of-order processors [36] can be considered as a type of restricted dataflow architecture [14]. DiAG has two main advantages and disadvantages compared to modern out-of-order processors. DiAG implicitly resolves data dependencies through its register lanes, eliminating most of the control structures necessary in out-of-order cores. Dynamic datapaths constructed by DiAG are reusable, thus loop iterations can execute at an efficiency close to accelerators. On the other hand, applications that are memory-centric or contain significant control divergence perform poorly since most cycles are wasted on stalls. DiAG will also inevitably use more die area although only a small fraction of it is powered at a time since PEs are only enabled when executing. A more detailed comparison between DiAG and out-of-order CPUs is discussed in Section 5.

3.2.2 Superscalar Techniques. Various similarly themed superscalar techniques have been proposed, though most are intended to optimize the design of out-of-order μarchs. ILDP [21] proposes an instruction set that allows the processor to group dependent instructions into strains that are executed in an interconnected back-end of PEs. However, ILDP retains of most of the front-end pipeline stages from fetch to register rename, and requires a special ISA with accumulator registers matching the number of PEs. Complexity-effective superscalar processors [30] also uses bypass logic to rapidly transfer dependent data between functional units. DiAG fully manifests this concept, replacing the register file and RAT with a full bypass circuit. CRIB [12] uses a very similar method to construct a dynamic datapath and eliminate front-end structures. DiAG is more elaborate in its design with full resource duplication at each PE. We extend CRIB's approach to the scale of hundreds of PEs not confined to the traditional CPU back-end; this allows us to additionally exploit full datapath reuse in loops, architecture composability, and thread-level pipelining.

Loop reuse techniques such as Revolver [13] seek to eliminate front-end redundancies when repeated instructions are executed in a loop. In DiAG, the entire datapath along with all functional units are duplicated and reused with optional pipelining. Clustered speculative multithreaded processors [27] spawn speculative threads from a single thread by detecting parallelizable sections of code at runtime, sharing some similarities with DiAG's thread pipeline mode. However, DiAG is non-speculative at the thread level and the former uses a multi-processor architecture. The mechanisms proposed in these works generally serve as extensions to the out-of-order microarchitecture whereas DiAG functions independently. Composable architectures such as CLP [20] and Core Fusion [15] can group multiple smaller processor elements and hardware resources to form a larger one using a flexible microarchitecture. DiAG applies a similar concept, however, we cluster individual PEs into different subgraphs, allowing the construction of multiple datapaths in the processor depending on properties of the application.

4 DATAFLOW-BASED EXECUTION

DiAG uses dynamic dataflow execution for serial parts of a program to minimize latency of execution. This is not an easy task considering that a major power burden of modern out-of-order processors lies in its complex front-end logic responsible for resolving control and data dependencies in the program. This overhead is necessary to allow aggressive dispatching of as many instructions as it can each cycle to maximize ILP. Consequently, even for floating-point operations, a significant chunk of total power spent executing each instruction is consumed by control structures such as the register alias table (RAT), reorder buffer (ROB), and reservation stations, rather than the functional units performing the operation. The purpose of these control structures is to: **1.** determine true data dependencies between instructions, **2.** dispatch instructions not waiting on dependencies to functional units each

cycle, and **3.** maintain a table of active instructions in program order for control hazards. We use an alternate method similar to CRIB [12] that accomplishes these tasks without explicit renaming or out-of-order issue by building a restricted DFG of the program dynamically in hardware. However, the graph need not be spatially mapped to a processor array, or even be actually constructed. Instead, it arises naturally when instructions are laid out sequentially.

4.1 Instruction Dataflow Graph

DiAG consists of a long row of functional units and each is assigned a single CPU instruction to execute. The instructions are assigned strictly in program order (e.g., functional units from left to right are assigned `i0, i1, i2, ...`). In most ISAs, regular instructions require up to two source registers (`RS1, RS2`) and write to one destination register (`RD`). However, registers values are transient by design, source registers of one instruction are destination registers written by some instruction before it. Rather than implementing a fixed register file, we use an interconnect of wires linking outputs of one functional unit to inputs of subsequent units.

For full generality, this interconnect converts the ISA's registers into wires of the same bit-width that can be accessed as it passes by the functional units. For example, an ISA with 16 registers, each 32-bit wide translates to 16 register lanes where each lane is **a 32-bit wire along with a valid bit**. Each functional unit reads its assigned instruction's source operands from the register lanes and writes its output to the destination register lane. To support writes to a lane, at each functional unit, a switch (2-input multiplexer) selects between propagating the lane's current value and the functional unit's output. This means that when a functional unit writes to a lane, it only changes the register lane's value for future functional units, not previous ones. The accompanying valid bit is also set to high when a functional unit writes its output. This bit allows subsequent units to be aware that the register lane's value is ready and correct. Finally, even though functional units are assigned instructions in program order, they can begin execution as soon as their inputs are valid, exploiting any available instruction-level parallelism.

We have now essentially constructed a restricted dataflow graph of the program. Viewed from a higher abstraction level, the row of functional units corresponds to computation nodes in the graph. Register lanes that are read and written correspond to edges in the graph. To better illustrate this concept, Figure 3 shows a simple example program that computes the Euclidean distance between two points. For simplicity, we assume the ISA has only four registers to work with. In Figure 3: **(A)** Shows the dataflow graph of the program with all dependencies between instructions. Assuming a one cycle latency for operations, the program completes in 3 cycles. **(B)** We flatten the same DFG from **(A)** by laying out instructions in program order, all edges remain unchanged. **(C)** Shows the design described in this section. This diagram simplifies the hardware and does not show unused multiplexers. Functional units are assigned instructions in program order (`i0` to `i4`). Each functional unit reads its inputs from the register lanes, and writes its output to the destination lane, overwriting its value and valid bit for subsequent functional units. We see in **(C)** that, even though functional units are assigned instructions in program order, they begin execution as soon as all inputs are valid. In the first cycle, both `i0` and `i2` have valid inputs from the register lanes and can begin execution. On the other hand, `i1` cannot begin because it takes register `r0` as input, and the `r0` lane is overwritten by `i0` whose output is currently invalid until it completes. Execution completes when all register lanes are valid at the last functional unit. Assuming a one cycle latency for each operation, we once again complete execution in three cycles, identical to the ideal case. In fact, it is clear that in **(C)**, we have implicitly constructed the same DFG shown in **(B)**, which in turn is simply a flattened view of the original DFG shown in **(A)**.

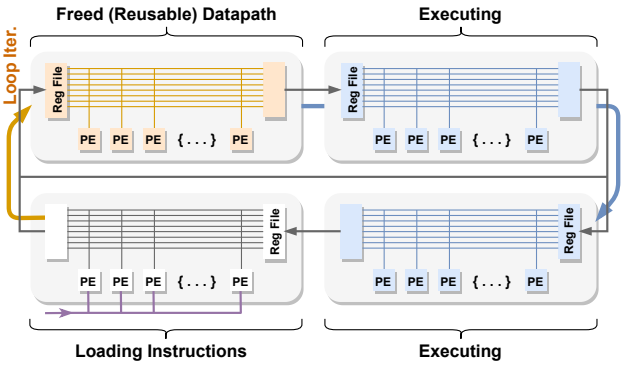


Figure 4: Four processing clusters: two are executing (blue), one is idle (orange), and the last is loaded with instructions. A loop can reuse this datapath.

4.2 Data Hazards

False register dependencies, namely write-after-read (WAR) and write-after-write (WAW) hazards do not obstruct ILP in DiAG. Normally, we cannot overwrite a register value if prior instructions still require the old value, but this is no issue for register lanes flowing in one direction. Recall that the output of each functional unit replaces the register value and valid bit for its destination register lane only for subsequent instructions. This is analogous to register renaming which assigns each instruction's destination register a new physical register, precluding prior instructions from overwriting its result. Hence, register writes of one instruction can complete at any time without affecting the instructions prior to it. Disregarding control hazards, DiAG can exploit all locally available ILP. If a program has instructions that are all independent, we can schedule as many instructions as there are functional units in one cycle, an issue width of up to infinite. In practice, however, we must account for wire delay and 2-input multiplexers placed at each functional unit, more details on timing are presented in Section 6.

4.3 Control Hazards

To support control flow changes such as branches and jumps in the program, we divide the long row of functional units into parts called processing clusters. This division is transparent to register lanes, whose values and control bits are connected from one cluster to another with a buffer in between. Figure 4 shows a design with 4 processing clusters that are chained in a circular connection. Functional units are interchangeably called processing elements (PEs) in

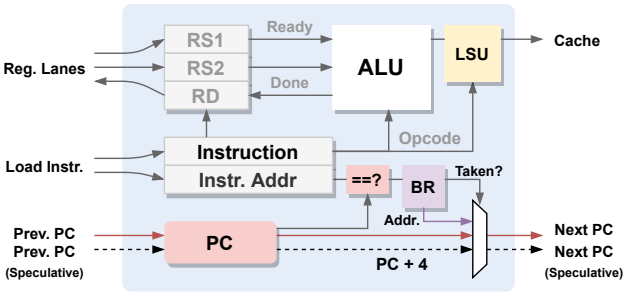


Figure 5: Control within a PE tracks the program counter (PC) and handles branches and jumps.

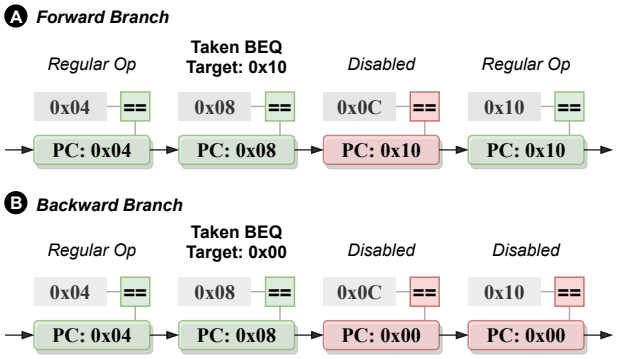


Figure 6: Example of a forward (A) and backward (B) branch instruction at address 0x08. When the branch is taken, the program counter (PC) no longer matches the next PE's instruction address and it is ignored.

all figures. Under serial execution where the program counter (PC) increments by one word every cycle, we simply load instructions into the next cluster while the current clusters execute. A cluster is freed if all its functional units have completed their instructions, thus, having just two clusters is enough to alternate between. Note that instructions are always assigned in program order; loading a single 64-Byte instruction cache line is enough to fill a 16 functional unit cluster in a 32-bit architecture.

Each PE has an instruction address register that holds address of the instruction assigned to it. In addition, the PC lane crosses every PE in a cluster. Control structures within each PE are shown in Figure 5. As instructions are simultaneously executing in the cluster, the PC crosses each instruction in program order allowing completion of memory stores (retiring instructions). The PC is normally incremented by 4 in each PE it passes (without actual addition) unless it encounters a branch or jump instruction.

4.3.1 Forward Branch. A positive offset jump or branch that is taken modifies the PC lane and sets its value to the jump or branch target address. As a result, the subsequent PEs' instruction addresses will no longer match the PC lane. This mismatch disables the functional unit and allows the PC lane to bypass all PEs until the next match is encountered. An example is shown in Figure 6 **(A)** where the third instruction (`0x10`) is skipped by the branch instruction. If the target address lies outside the current processing cluster, the next cluster will load the instructions at the target address.

4.3.2 Backward Branch. A backward branch is handled in mostly the same way as forward branches. Figure 6 **(B)** shows an example of a backward branch that is taken. As before, subsequent instruction addresses will not match the modified PC lane and will be disabled. However, in the case that the target address is in a cluster already present in the processor, such as the case shown earlier in Figure 4, we can reuse the dynamic datapath that is already constructed for the current loop. This enables instruction reuse and eliminates the overhead of instruction fetching and decoding. If the target address does not fall within the range of any cluster, it must be loaded from the instruction cache again.

4.4 Thread Pipelining

In contrast to the previous section, thread pipelining targets parallel parts of a program with the goal of maximizing throughput. We extend the dataflow architecture in the previous section by pipelining register lanes so that possibly all functional units are active when the pipeline saturates. If two different threads that can run concurrently, we use **spatial parallelism** to dedicate multiple rows of processing clusters to execute each in parallel, similar to multicores. If two threads are identical but process different data, we can additionally use **temporal parallelism**, i.e., thread-level pipelining to further improve execution efficiency in DiAG.

4.4.1 SIMT Pipeline. Thread pipelining is similar to loop pipelining where different iterations of a loop execute at different stages of the pipeline, though each iteration is still executed sequentially. An easy way to visualize the pipeline is to treat it as a generalization of the traditional instruction pipeline in microprocessors.

In the classic 5-stage pipeline, each instruction is broken down into five parts from fetch to write-back. Each pipeline stage then consists of dedicated hardware logic for one part. When a program runs, instructions flow through the pipeline and, if there are no stalls, the pipeline achieves a cycles per instruction (CPI) of exactly 1 with all stages busy. Though instructions are overlapped, each instruction's parts are always performed in correct order, i.e., no instruction will be decoded before fetched or executed before decoded. We follow the same intuition to construct a pipeline for threads rather than instructions. Rather than breaking down instructions into parts, we break down threads into instructions. This is possible since DiAG exploits instruction reuse to reduce each instruction to only the execute stage. Thus, each pipeline stage now executes a complete instruction, and each thread, carrying its register file, flows through the pipeline to complete execution as shown in Figure 7. We can modify our dataflow design to support thread pipelining by inserting pipeline registers between functional units. This graph is exaggerated to illustrate a point, in our design, pipeline registers are inserted between clusters, not each PE due to tremendous area cost. As before, functional units are assigned instructions of the thread in program order, however, from the perspective of each executing thread, its instructions are also executed in original program order.

The main benefit of instruction pipelining is achieving an ideal CPI of 1 with temporal parallelism. In the case of thread pipelining,

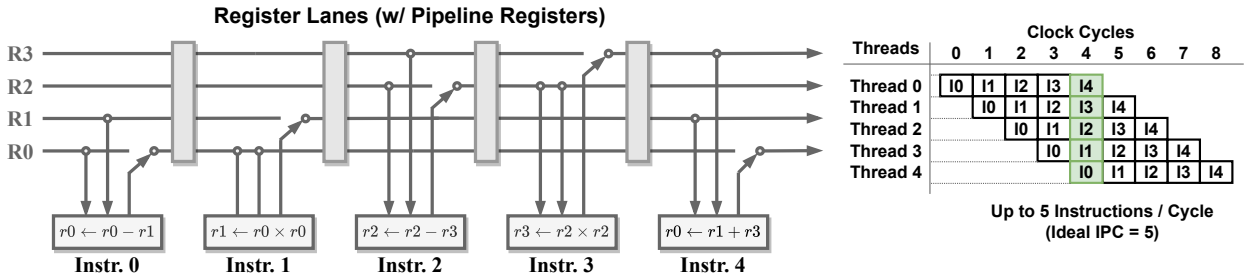


Figure 7: Thread pipelining with effectively only execute stages for each instruction. Ideally, all five instructions are active each cycle and `IPC = 5` in this example. (This figure is exaggerated, pipeline registers only exist between DiAG clusters, not each PE.)

we can ideally execute one thread per cycle, provided there are no stalls and enough PEs for all instructions. This also implies that the instruction throughput (IPC) achieved by the pipeline scales linearly with the number of PEs available. If the thread has fewer instructions than available PEs, we can spatially replicate the pipeline across clusters to maximize utilization of resources. This is a major advantage in comparison to multicores since rather than scaling IPC with more cores, we scale IPC with more PEs which have minimal control overhead. However, an ideal scenario with 1-cycle operations and no stalls is unrealistic and only represents a theoretical peak throughput. The pipeline can easily stall due to cache access misses and non-uniform execution time of instructions across clusters.

4.4.2 Data Hazards. While parallel threads can be executed in any order, instructions within each thread execute strictly in program order as it flows through the pipeline as shown on the right side of Figure 7, thus there are no data hazards. For each thread, `i0` always executes before `i1`, then `i1` before `i2`, and so on.

4.4.3 Control Hazards. There are unfortunate constraints to thread pipelining that limit its applicability to all types of parallel programs. For starters, there must be enough PEs to fit all instructions of the thread in the DiAG processor. Secondly, there cannot be backward jumps or conditional branches within the thread (e.g., nested loops). If they exist, they must be fully unrolled, otherwise the threads are executed sequentially. A backward branch cannot be realized when pipelined since past PEs are concurrently used by other threads. Fortunately, forward branches are easily handled since each thread carries its own PC through the pipeline. Like before, each PE is enabled only if its instruction address matches and the thread's PC. When a branch instruction is encountered and the branch is taken, the thread's PC is modified to the branch target address, effectively nullifying the subsequent instructions until the correct address is reached. Thus, control divergence is not as significant a problem here as it is in vectored processors.

5 GENERAL ARCHITECTURE

This section presents the general architecture of a DiAG processor complete with its control and memory subsystem. Much of the DiAG architecture is parametrizable with a multitude of possible design choices. An architect can optimize it for performance and efficiency in specific use cases. Although we assume an implementation supporting the RISC-V 32-bit ISA, the DiAG architecture we present is intended to be ISA agnostic and should work reasonably well with most general-purpose instruction sets.

5.1 Overall Organization

A high-level architecture diagram is depicted in Figure 8. DiAG is organized hierarchically by the following hardware divisions: dataflow rings, which contain processing clusters, which contain individual PEs. A dataflow ring is analogous to a CPU 'core', it connects multiple clusters together and is the smallest hardware unit that a program can run on. Each ring is independently equipped with a control unit responsible for handling instruction line fetches to its clusters, activating and freeing clusters, and thread-level control tasks. Note that multiple rings can be chained together to form a larger ring with a longer datapath. A DiAG processor can thus have multiple ring configurations to exploit different types of parallelism. Each processing cluster contains a row of PEs as we described in the previous section. Individual loads and stores are queued at the level of the processing cluster. Likewise, branch, jump, and call instructions whose target addresses fall outside the current cluster are also handled at the cluster level.

5.1.1 Instruction Fetching. Instructions are fetched at the granularity of I-Cache lines and are assigned in program order to PEs in a cluster. Each processing cluster should hold exactly one instruction cache line, note that I-Cache lines and register lanes share the same on-chip network for data transport. Our DiAG implementation has 16 PEs per cluster and 64-Byte cache lines, thus a single line fills all PEs in a cluster. A processing cluster can begin execution as soon as instructions are decoded, which takes one cycle after they are assigned to PEs. When the PC is branched to an instruction address not aligned to the cache line, we fetch and load the entire line to the cluster anyway. Instructions prior to our target address will be disabled anyways due to PC mismatching the instruction address. A standard direct-mapped instruction cache is used in our evaluations.

5.1.2 Functional Units. Since each functional unit is assigned only one instruction to execute at a time, we can consider its implementation as either a general-purpose ALU/FPU or a fine-grained reconfigurable compute unit. Using reconfigurable logic has the advantage of reduced area and power costs, but in turn achieves a

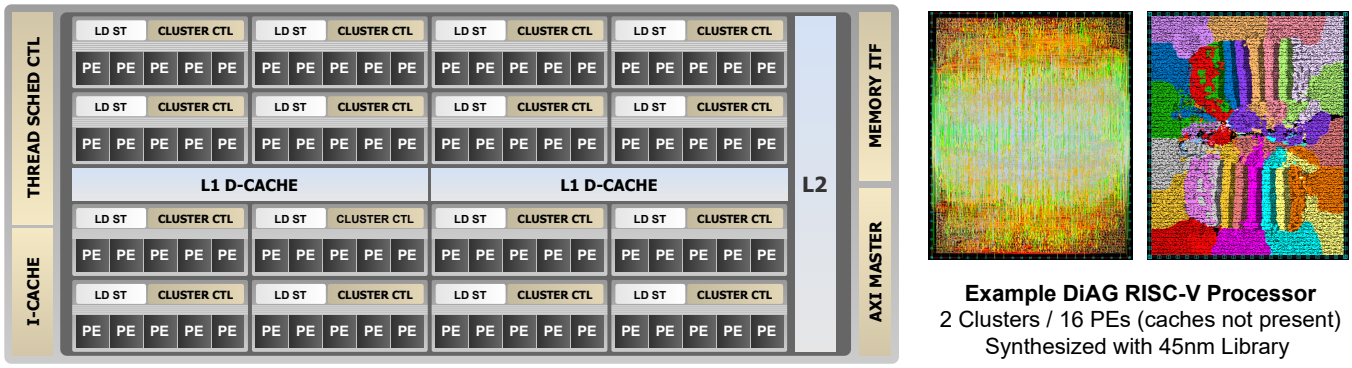


Figure 8: DiAG high-level organization and synthesized example with 2 clusters and 16 PEs.

lower frequency. The problem with a reconfigurable design here is that it must always be allocated enough area for the costliest instruction (`FDIV`). In the current prototype, we use a generic 32-bit integer ALU and FPU with some shared computation logic.

5.1.3 Centralized Control. While control transfer instructions are handled within each cluster, a control unit for each dataflow ring keeps track of instruction assignments to each cluster and execution progress. This unit maintains a hardware scheduling table that stores the head and tail clusters in use for each thread as well as their statuses. Its tasks include preemptively loading instruction lines, freeing completed clusters, and tracking PC. An on-chip 512-bit bus is present to transport partial register files from clusters that are not directly connected (in two cycles) for backward branches. As previously noted, this bus is also shared for loading I-Cache lines to clusters.

5.1.4 Interrupts and Exceptions. Since instructions are mapped to PEs in program order, DiAG can easily support precise interrupts. DiAG's register lanes serve the same purpose as a reorder buffer in an out-of-order processor. When an interrupt is encountered at instruction `i`, all instructions from `i+1, i+2, ...` are automatically disabled because the PE for instruction `i` modifies the PC lane to the target trap vector causing subsequent PEs have a PC mismatch. Finally, all previous instructions can write their values to the current processing cluster's register file. The next cluster is then loaded with instructions at the target PC, beginning the interrupt sequence. The same process applies to a branch misprediction; PEs can always execute at will, but the PC lane essentially retires instruction in-order like a reorder buffer.

5.2 Memory Subsystem

A robust memory subsystem is crucial to the overall performance of a DiAG processor. Under thread pipelining, this is further amplified since a missed load stalls the entire pipeline. We use a hierarchy of a memory lanes, cluster-level caches, banked L1 D-Caches, and a larger last-level cache. Memory accesses in each functional unit are first checked against memory lanes then routed to a load store unit at the cluster level, where the previously accessed line is stored. If missed, the request is queued and then sent to access the banked L1 D-Cache, where a secondary arbiter manages incoming requests from processing clusters. The L1 D-Cache in this case is technically a second level cache, and we choose a size in the range of 32-128KB depending on the configuration. Locally, at each cluster, we use memory lanes, which are essentially set-associative register lanes that transports memory data from PE to PE and enable access reordering. Data written by stores are temporarily stored in memory lanes that are passed to succeeding clusters and PEs for immediate access. There is significant room for further memory optimization for our work since, with instruction reuse, each PE is assigned a single memory instruction whose address likely changes in a fixed pattern each iteration. We expect that localized stride prefetching and more advanced techniques will be effective in DiAG, however they are beyond the scope of evaluation for this work.

5.3 Comparison with Superscalar CPU

We compare at large the design aspects of DiAG against typical out-of-order CPUs to highlight its benefits and drawbacks. We summarize how each instruction is processed in a standard CPU and in DiAG in Table 1. The second column shows the case for a purely serial program and the third column shows the case when a parallelizable loop fits on DiAG. Dataflow execution can theoretically achieve the performance of a wide out-of-order core without needing most of its control structures.

DiAG's main disadvantage (for serial programs especially) is the area consumed for a large number of PEs and register lanes. We can certainly use only two clusters with few PEs, but that would defeat the purpose of instruction reuse since most loops can no longer fit. Thus, we decide to duplicate hardware resources and maintain that generally only a small part of the circuit is active at once, and PEs are dormant until an instruction is ready to execute. Furthermore, the total number of operations performed for each program remains unchanged. This is disadvantage disappears for parallel parts of the program where a traditional processor must also invest in additional cores to achieve a high throughput. Under thread pipelining, IPC is expected to scale almost linearly with the number of functional units, eliminating the control overhead of multicores. Trading hardware area for efficiency is DiAG's philosophy under the dark silicon regime. We can treat DiAG's design as a temporal record of a processor's back-end laid out spatially in hardware with register lanes traveling in the direction of time.

Table 1: Comparison with out-of-order processor.

Stages and Structures	Out-of-Order Processor	DiAG (Initial)	DiAG (Reuse)
Fetch	Yes	Yes (Batch)	No
Decode	Yes	Yes	No
Issue	Yes	No	No
Issue Width	4-8 Instr.	Scalable	Scalable
Rename	Yes	No	No
Register File	Physical RF	Reg Lanes	Reg Lanes
Dispatch	Yes	No	No
Execute	Yes	Yes	Yes
Commit	Reorder Buffer	Reg Lanes	Reg Lanes

5.4 ISA Extensions

We extend the RISC-V ISA with two additional instructions to enable thread-level pipelining:

- `simt_s, rc, r_step, r_end, interval`. Spawns multiple loop instances by retaining the current register file in the cluster with the exception of the control register `rc`. This instruction must be followed by a `simt_e` instruction that is offset by `l_offset` and denotes the end of the current pipelined loop. The value and type of `r_step` determines how the control register changes and `r_end` determines the ending condition. Threads are only initiated once every `interval` cycles.
- `simt_e, rc, r_end, l_offset`. This instruction is used to mark the end of the current pipelined region, does not propagate all but the last thread's register lanes to the next processing cluster when the terminating condition is met.

These instructions are inserted at the beginning and end of parallelizable loops. Currently, loops that can be pipelined in DiAG (without negative offset branches) must be identified manually due to numerous restrictions.

6 HARDWARE IMPLEMENTATION

We implement a parametrizable DiAG processor prototype in SystemVerilog to evaluate its performance. We target the RISC-V 32-bit (RV32I) instruction set with optional multiplication (-M) and floating-point (-F) extensions. At this time, we do not have complete hardware support for environment call, system, and atomic instructions. Table 2 lists the different hardware configurations used for evaluation.

6.1 Hardware Synthesis

We synthesize our hardware design using Synopsys Design Compiler `L-2016.03-SP1` with a FreePDK 45nm library. We then perform basic place and route of the synthesized design using Synopsys IC Compiler for more accurate area and power estimations. Table 3 shows hardware area and power consumption breakdown of key components in DiAG, hierarchically. The L1 and L2 caches are modeled separately with CACTI [25] and are not present in the hardware design.

Table 2: DiAG configurations used for evaluation.

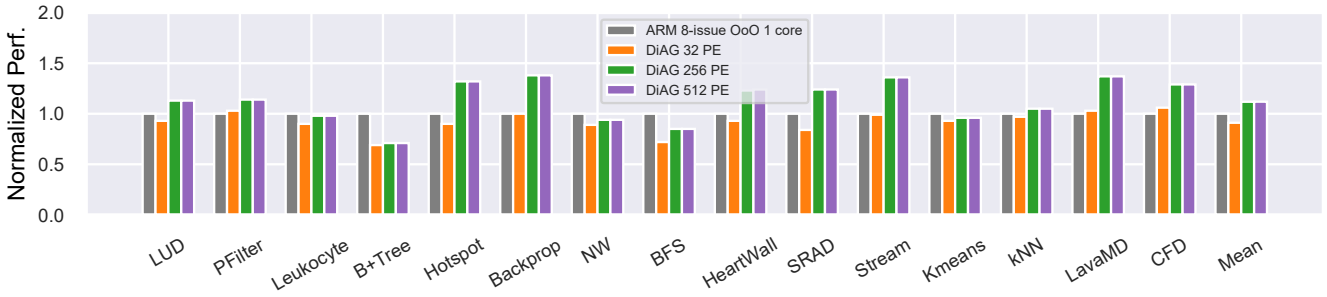
Configuration	I4C2	F4C2	F4C16	F4C32
ISA	RV32I	RV32IMF	RV32IMF	RV32IMF
PEs / Cluster	16	16	16	16
Total Clusters	2	2	16	32
Total PEs	32	32	256	512
Freq. (Sim.)	N/A	2.0GHz	2.0GHz	2.0GHz
Freq. (Syn.)	100MHz	1.0GHz	1.0GHz	1.0GHz
L1I Cache Size	32KB	32KB	32KB	32KB
L1D Cache Size	32KB	64KB	128KB	128KB
L2 Cache Size	N/A	4MB	4MB	4MB

Table 3: Hardware area and power breakdown by component. * Indicates estimations not entirely from synthesis.

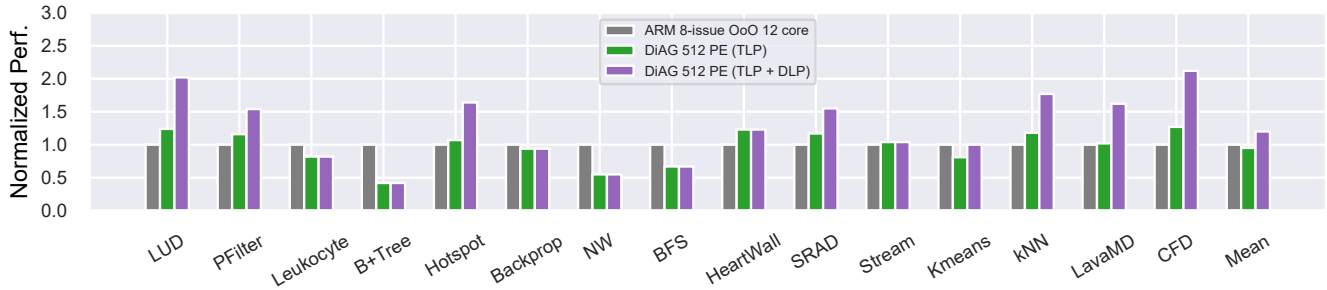
Component Name	Hardware Area	Total Power
F4C32 (TOP)	93.07 mm^{2*}	74.30 W*
PCLUSTER	2.208 mm^{2*}	2.104 W*
PE (w/ FPU)	97014 μm^2	120.4 mW
REGLANE	15731 μm^2	3.063 mW
INT ALU	1375.4 μm^2	0.774 mW
FPU (MUL / DIV)	66592 μm^2	105.2 mW
RV_DECODER	244.6 μm^2	0.019 mW

6.1.1 Hardware Area. Area is dominated by floating-point units that each occupy 68% of a PE and together occupy 48% of a processing cluster. Register lanes account for 16.3% of a processing cluster. In our design, register lane accesses are synthesized as a chain of simple MUXes. We do not have the resources for a custom physical design with shared read wires that could greatly reduce both area and delay. Although we are using an older technology library, the area cost of each PE is almost $10^5 \mu m^2$ and each cluster around $2mm^2$. This raises concerns as whether DiAG can be scaled to a 64-bit ISA with 64 register lanes. Indeed, direct scaling to 64 register lanes would notably increase hardware area. However, for a 64-bit design, we can first multiplex only the registers that are accessed by the current cluster. For example, a cluster with 16 instructions can at most access 32 different registers. Hence, the original 32 register lane design can still be used with some modifications.

6.1.2 Circuit Timing. Timing is met at 1.0GHz for a processing cluster with register lanes buffered every 8 PEs. The critical path in the cluster is the longest path on the register lane running from the first PE output to last PE's input within the cluster. Each lane passes through a 2-input MUX at each PE multiplexing the current value and write value. Thus, for a cluster with 16 PEs, we insert a full register buffer on all lanes between PE 8 and 9. We also pass timing with a 2.0GHz clock with a fixed two cycle delay from first to last register lanes, this may be preferred as integer ALUs can run at a higher frequency. Note, however, that the DiAG circuit is fully synchronous and all PEs across each cluster share the same clock domain.



(a) Single-Thread Performance (Rodinia)



(b) Multi-Thread Performance (Rodinia)

Figure 9: Performance results on the Rodinia benchmark suite.

6.1.3 Dynamic Power. Power consumption numbers are reported in Table 3 which assumes all PEs are powered on every cycle. However, during execution, PEs are only enabled when they execute after all input operands are available; we assume they are clock-gated in a similar fashion to FPUs in a regular processor.

6.2 FPGA Proof of Concept

We synthesize the integer-only model I4C2 (with 32 PEs) on a Xilinx VC709 FPGA board running at 100MHz. The DiAG processor uses an AXI4-Lite master port to access on-chip memories. A Microblaze processor is also present for monitoring purposes. We preloaded bare metal RISC-V programs in memory to verify basic functionality. Due to limitations of the FPGA, however, we do not use the platform for performance evaluation.

7 EVALUATION

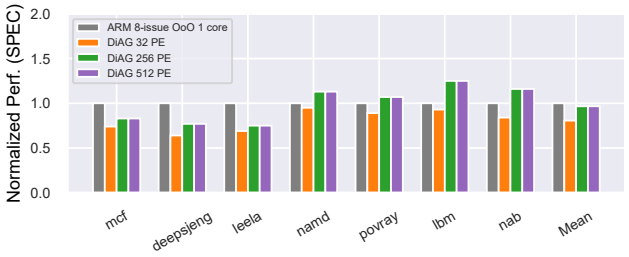
We evaluate DiAG's performance and energy efficiency using benchmarks from the Rodinia and SPEC CPU2017 benchmark suites against a 12-core 8-issue out-of-order ARM CPU baseline.

7.1 Methodology

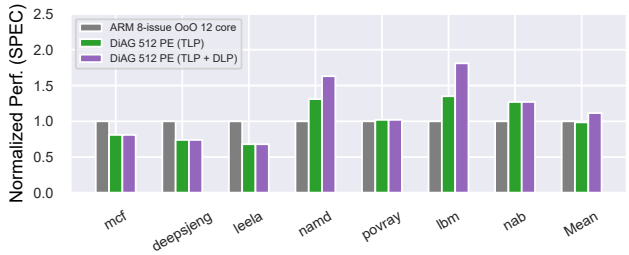
Hardware performance is modeled with RTL simulation using ModelSim on the F4C2, F4C16, and F4C32 configurations in Table 2. Our hardware prototypes do not have a full support of required system instructions, so we modify, delete, and circumvent non-profiled or non-critical sections of benchmark code to avoid all system calls. The modified benchmarks are then cross-compiled to ARM and to RISC-V with DiAG extensions inserted if applicable. For power estimations, we record component utilization each cycle in the RTL testbench. A disabled processing element or floating-point unit is assumed to be clock-gated, and we assume it does not consume dynamic power unless the instruction activates it. We then estimate total energy consumed based the fraction of dynamically active components each cycle. We use the `gem5` [1] simulator in Syscall Emulation (SE) mode to model an ARM CPU baseline (due to issues with RISC-V in the simulator). The ARM baseline runs at the same frequency and is aggressively configured to issue, dispatch, and retire up to 8 instructions with a 2 cycle latency for each of these stages. A similar memory hierarchy of 64KB L1 and 4-8MB unified L2 is selected for the ARM processor. Power consumption for simulation is estimated using McPAT [24]. To improve simulation speed, the SystemVerilog testbench module used for simulation is less detailed compared to the full design used for synthesis. Floating-point operations are modeled as fixed delays and performed with non-synthesizable real variables, and caches are also only modeled functionally with delays. Some benchmarks are not simulated in full due to speed limitations of RTL simulation, and results are projected based on a smaller input run. However, given the general regularity of these applications, we have tested various input sizes to check that projected results are reasonable.

7.2 Single and Multi-Thread Performance

7.2.1 Rodinia Benchmarks. Figure 9a shows performance results of Rodinia benchmarks running with one thread. Overall, the average performance of DiAG is 0.91×, 1.12×, and 1.12× compared to the baseline CPU for configurations with 32, 256, and 512 PEs respectively. Much like large ROB sizes, no noticeable improvement can be gained with more than 256 PEs for serial programs. Memory



(a) Single-Thread Performance (SPEC)



(b) Multi-Thread Performance (SPEC)

Figure 10: Performance results on benchmarks from the SPEC CPU2017 benchmark suite.

misses across account for nearly 74% of all stalls in DiAG PEs, this is a detriment in control and memory-bound applications where DiAG performs much worse than the CPU baseline. Nevertheless, we observe that even the DiAG configuration with 32 PEs can come close in performance in generally compute heavy benchmarks if we do not intend to exploit instruction reuse and thread pipelining.

In the multi-threaded case, DiAG is simply configured to run in 16-by-2 format, i.e., each thread is allocated to a dataflow ring with two clusters to alternate between. Our single-thread results show that 32 PEs is enough to extract most of the available ILP, however instruction reuse is all but sacrificed. Thus, to exploit TLP, we configure DiAG to concurrently run as many instances as possible. Figure 9b shows multi-thread performance result against the CPU baseline. The purple bar indicates temporal parallelism (SIMT pipelining) applied in addition to spatial parallelism with multiple threads. We manually identify or modify program regions in applicable benchmarks to enable thread pipelining while maintaining correctness. We observe that performance is slightly slower (0.95× on average) compared to the ARM CPU but elevates to 1.2× with thread pipelining enabled. The multicore CPU retakes its advantage in most benchmarks where we do not additionally apply SIMT pipelining. It is therefore important to configure DiAG with enough PEs to exploit reuse in most workloads to unlock its potential with thread pipelining. This performance degradation is also largely attributed to memory stalls due to load congestion, especially in the initial segments of the benchmark kernels.

7.2.2 SPEC Benchmarks. We additionally evaluate a subset of SPEC CPU2017 benchmarks. Note that benchmarks that use Fortran source code (e.g., `bwaves`) or are too difficult to modify to bypass unsupported system calls (e.g., `gcc`) are not evaluated. Identical

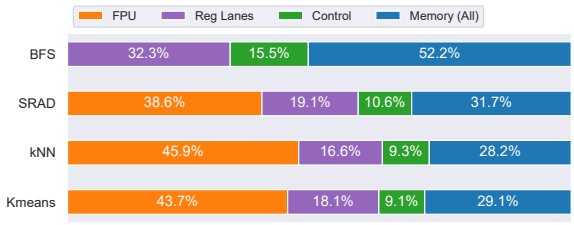


Figure 11: Energy consumption breakdown (%) by hardware component across four benchmarks.

baseline and DiAG configurations are used in these tests. Results in Figure 10a shows that the average single-thread performance for DiAG is 0.81×, 0.97×, and 0.97× compared to baseline for configurations with 32, 256, and 512 PEs respectively. For the multi-thread case in Figure 10b, DiAG with 512 PEs achieves the same average performance relative to baseline and a speedup of 1.15× with pipelining enabled. Individual results reveal the same trend as Rodinia benchmarks where DiAG excels in the more compute intensive applications and trails behind in memory-bound or control-dependent applications.

7.3 Performance and Utilization

We take a closer look at how energy is spent by the DiAG processor and inspect its performance bottlenecks.

7.3.1 Component Utilization. Figure 11 shows the energy consumption breakdown by hardware component in four Rodinia benchmarks. The FP unit is clock-gated when the PE executes a non-FP instruction, and consumes very little leakage power. Register lanes (including integer ALUs), memory (including LSUs and caches), and the remaining control logic are assumed to be always powered. In compute-heavy benchmarks, DiAG expends close to half of total energy consumed on the functional units, a desirable result however the 20% overhead on register lanes is nontrivial. It is no surprise that in graph traversal applications, energy consumption is largely dominated by memory and data movement.

7.3.2 Breakdown of Stalls. The main causes of stalled instructions averaged across the Rodinia benchmarks are listed below. For this statistic, we only count the source of stalls, not dependent instructions that are subsequently stalled.

- **73.6% - Memory stalls.** This includes cache misses, full LSU request queues, busy bus, etc. Memory stalls frequently arise due to load congestions, which are difficult to circumvent. Even with memory lanes, there is significant room for memory-side optimization in DiAG.
- **21.1% - Control flow changes.** When a branch misprediction occurs, all subsequent PEs are flushed. The correct instruction line must then be loaded into the current or next processing cluster, wasting at least 3 cycles.
- **5.3% - Other stalls.** Stalls caused mostly by structural hazards such as when the shared on-chip 512-bit bus for register lanes and instruction lines is busy or when no clusters in a dataflow ring is free to load new instructions.

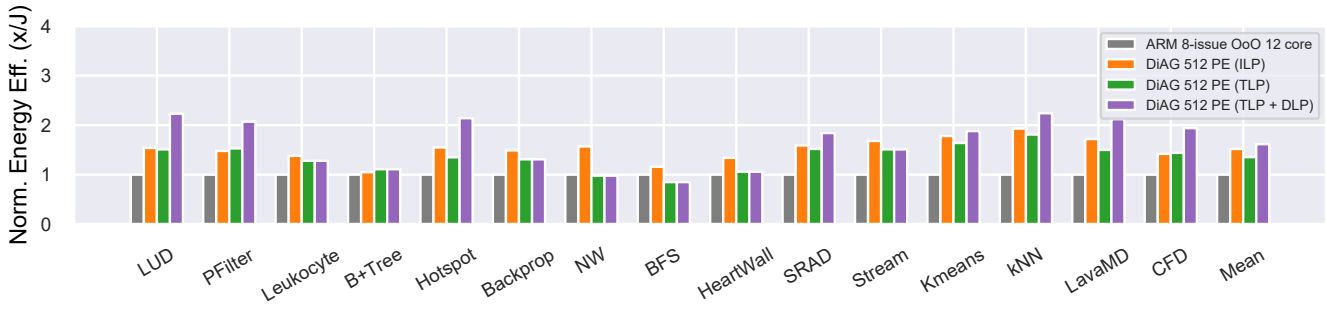


Figure 12: Energy efficiency results on the Rodinia benchmark suite.

Results in this section show that DiAG could be well improved with more advanced prediction and prefetching techniques, and benefit from a more robust memory subsystem. However, the distributed nature of PEs in DiAG make it difficult to directly replicate existing microarchitectural techniques employed by traditional CPUs. This also presents new opportunities, for example, since instructions assigned to each PE are fixed under reuse, we can implement FIFO structures for memory accesses as accelerator designs commonly use. Likewise, penalties due to unpredictable control flow changes can potentially be ameliorated by simultaneously constructing multiple speculative datapaths since DiAG's hardware resources are abundant but usually sparsely enabled.

7.4 Energy Efficiency

As noted previously, we measure dynamic power consumption based on the utilization of PEs and floating-point units. Static power is determined from synthesis and assumed to be constant. Figure 12 shows energy efficiency improvements in the Rodinia benchmarks measured as the inverse of total energy spent during execution. Despite, losing performance on some of the benchmarks evaluated, energy efficiency is improved across most benchmarks in both single and multi-threaded cases, with an average of 1.51× and 1.35× respectively, and 1.63× with SIMT pipelining enabled. Energy efficiency is improved largely due to eliminated control overheads as memory and computation structures account for nearly all of DiAG's power budget. This improvement is most apparent in programs with significant instruction reuse, where already constructed datapaths consume only dynamic power for functional units and register lanes. However, in memory-bound benchmarks, register lanes and the memory subsystem dominate energy consumption as shown in Figure 11. The overheads of data movement in register lanes thus outweigh the almost nonexistent cost of computation at each PE, yielding little to no improvements.

7.5 Discussion and Future Work

DiAG is an approach at general-purpose processor architecture where we adhere to an existing ISA and maintain transparency to software as a baseline requirement. These restrictions make DiAG processors viable as drop-in replacements, however, working within the confines of building DFGs dynamically in hardware has its limitations as we observe many of the same memory and control bottlenecks still obstruct performance. As benchmark results in this section demonstrate, DiAG can support all application types but, in its current form, excels most at computationally intensive programs. However, a main drawback of DiAG is the high hardware area cost that arises from the duplication of computation resources necessary to support any instruction at every PE. As a result, most of the hardware area is dynamically dead during execution unless thread pipelining is enabled. In the end, we improved energy efficiency while retaining generality at the cost of hardware area. This leads us to consider two possible future directions: reduce hardware area by resource sharing, or improve hardware utilization by supporting more execution modes. The first approach shares functional units within clusters not unlike a CPU's back-end. We inevitably sacrifice some performance due to structural hazards. The second approach keeps the current hardware layout and seeks to improve the utilization of PEs. We realize that PEs are generic and DiAG could easily be adapted to additionally support vector instruction sets. Under this approach, a DiAG processor can be temporally reconfigured or spatially partitioned to serve both the roles of main CPU and accelerator.

8 CONCLUSION

We proposed a dataflow-based architecture for general-purpose microprocessors that can dynamically construct a reusable execution datapath. DiAG exploits instruction-level parallelism and data-level parallelism while eliminating most of the control overhead of traditional out-of-order processors. This is done by replicating a dataflow graph of the program in hardware, which naturally reveals and resolves instruction dependencies. We evaluate DiAG against an aggressive out-of-order processor and conclude that it achieves similar performance under the same frequency but benefits from superior energy efficiency. The main disadvantage of this architecture is the die area cost of register lanes and hundreds of PEs equipped with floating-point hardware. However, under the dark silicon regime, the DiAG architecture is an example of 'spending' area to 'buy' energy efficiency without sacrificing generality. As it stands, the DiAG architecture is still in its infancy and presents many opportunities for future optimization.

ACKNOWLEDGMENTS

We would like to thank Daniel Sanchez and the anonymous reviewers for their helpful feedback. This work was supported by NSF award CNS-1705047.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

[2] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

[3] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[4] Al Davis. 1978. The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine. In *Proceedings of the 5th Annual Symposium on Computer Architecture (ISCA '78)*. Association for Computing Machinery, New York, NY, USA, 210–215. https://doi.org/10.1145/800094.803050

[5] Jack B. Dennis and David P. Misunas. 1974. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA '75)*. Association for Computing Machinery, New York, NY, USA, 126–132. https://doi.org/10.1145/642089.642111

[6] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108

[7] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2010. Task Superscalar: An Out-of-Order Task Pipeline. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, New York, NY, USA, 89–100. https://doi.org/10.1109/MICRO.2010.13

[8] Adi Fuchs and David Wentzlaff. 2019. The Accelerator Wall: Limits of Chip Specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, New York, NY, USA, 1–14. https://doi.org/10.1109/HPCA.2019.00023

[9] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. 2000. PipeRench: A Reconfigurable Architecture and Compiler. *Computer* 33, 4 (April 2000), 70–77. https://doi.org/10.1109/2.839324

[10] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (Sept. 2012), 38–51. https://doi.org/10.1109/MM.2012.51

[11] V. G. Grafe, George S. Davidson, James E. Hoch, and Victor Paul Holmes. 1989. The Epsilon Dataflow Processor. *SIGARCH Computer Architure News* 17, 3 (April 1989), 36–45. https://doi.org/10.1145/74926.74930

[12] Erika Gunadi and Mikko H. Lipasti. 2011. CRIB: Consolidated Rename, Issue, and Bypass. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/2000064.2000068

[13] M. Hayenga, V. R. K. Naresh, and M. H. Lipasti. 2014. Revolver: Processor architecture for power efficient loop execution. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, New York, NY, USA, 591–602. https://doi.org/10.1109/HPCA.2014.6835968

[14] Wen-Wei Hwu and Yale N. Patt. 1998. HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)* (Barcelona, Spain) *(ISCA '98)*. Association for Computing Machinery, New York, NY, USA, 300–308. https://doi.org/10.1145/285930.285989

[15] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. Association for Computing Machinery, New York, NY, USA, 186–197. https://doi.org/10.1145/1250662.1250686

[16] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Computer Architecture News* 45, 2 (June 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[17] Richard M. Karp and Raymond E. Miller. 1966. Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411. https://doi.org/10.1137/0114108

[18] Krishna M. Kavi, Roberto Giorgi, and Joseph Arul. 2001. Scheduled dataflow: execution paradigm, architecture, and performance evaluation. *IEEE Trans. Comput.* 50, 8 (2001), 834–846. https://doi.org/10.1109/12.947003

[19] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (Sept. 2011), 7–17. https://doi.org/10.1109/MM.2011.89

[20] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. 2007. Composable Lightweight Processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE, New York, NY, USA, 381–394. https://doi.org/10.1109/MICRO.2007.41

[21] Ho-Seop Kim and James E. Smith. 2002. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (Anchorage, Alaska) *(ISCA '02)*. IEEE Computer Society, New York, NY, USA, 71–81. https://doi.org/10.1145/545214.545224

[22] David Kuck, Edward Davidson, Duncan Lawrie, Ahmed Sameh, Chuanqi Zhu, Alexander Veidenbaum, Joel Konicek, Pen-Chung Yew, Kyle Andrew Gallivan, William Jalby, Harry Wijshoff, Randall Bramley, U. M. Yang, Perry Emrath, D. Padua, Rudolf Eigenmann, Jay Hoeflinger, G. Jaxon, Zhiyuan Li, T. Murphy, and J. Andrews. 1993. The Cedar System and an Initial Performance Study. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/165123.165157

[23] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. 1998. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VIII)*. Association for Computing Machinery, New York, NY, USA, 46–57. https://doi.org/10.1145/291069.291018

[24] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[25] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-Level Modeling for SRAM-Based Structures with Advanced Leakage Reduction Techniques. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '11)*. IEEE Press, New York, NY, USA, 694–701. https://doi.org/10.5555/2132325.2132479

[26] Andrea Lottarini, João P. Cerqueira, Thomas J. Repetti, Stephen A. Edwards, Kenneth A. Ross, Mingoo Seok, and Martha A. Kim. 2019. Master of None Acceleration: A Comparison of Accelerator Architectures for Analytical Query Processing. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 762–773. https://doi.org/10.1145/3307650.3322220

[27] Pedro Marcuello and Antonio González. 1999. Clustered Speculative Multithreaded Processors. In *Proceedings of the 13th International Conference on Supercomputing* (Rhodes, Greece) *(ICS '99)*. Association for Computing Machinery, New York, NY, USA, 365–372. https://doi.org/10.1145/305138.305214

[28] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/1168857.1168878

[29] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. *SIGARCH Computer Architecture News* 45, 2 (June 2017), 416–429. https://doi.org/10.1145/3140659.3080255

[30] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. 1997. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 206–218. https://doi.org/10.1145/264107.264201

[31] Gregory M. Papadopoulos and David E. Culler. 1990. Monsoon: An Explicit Token-Store Architecture. *SIGARCH Computer Architecture News* 18, 2SI (May 1990), 82–91. https://doi.org/10.1145/325096.325117

[32] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. 2004. TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP. *ACM Transactions on Architecture and Code Optimization* 1, 1 (March 2004), 62–93. https://doi.org/10.1145/980152.980156

[33] Aaron Smith, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, Kathryn S. McKinle, and Jim Burrill. 2006. Compiling for EDGE Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, New York, NY, USA, 185–195. https://doi.org/10.1109/CGO.2006.10

[34] Avinash Sodani and Gurindar S. Sohi. 1997. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 194–205. https://doi.org/10.1145/264107.264200

[35] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The WaveScalar Architecture. *ACM Transactions on Computer Systems* 25, 2, Article 4 (May 2007), 54 pages. https://doi.org/10.1145/1233307.1233308

[36] Robert Marco Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33. https://doi.org/10.1147/rd.111.0025

[37] Rex Vedder and Dennis Finn. 1985. The Hughes Data Flow Multiprocessor: Architecture for Efficient Signal and Data Processing. *SIGARCH Computer Architecture News* 13, 3 (June 1985), 324–332. https://doi.org/10.1145/327070.327290

[38] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. *SIGARCH Computer Architecture News* 38, 1 (March 2010), 205–218. https://doi.org/10.1145/1735970.1736044

[39] Toshitsugu. Yuba, Kei Hiraki, Toshio Shimada, Satoshi Sekiguchi, and Kenji Nishida. 1987. The SIGMA-1 Dataflow Computer. In *Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow* (Dallas, Texas, USA) *(ACM '87)*. IEEE, New York, NY, USA, 578–585. https://doi.org/10.5555/42040.42134