MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems

Graham Gobieski
gobieski@cmu.edu
Carnegie Mellon University

Amolak Nagi
amolakn@andrew.cmu.edu
Carnegie Mellon University

Nathan Serafin
nserafin@andrew.cmu.edu
Carnegie Mellon University

Mehmet Meric Isgenc
mericisgenc@gmail.com
Carnegie Mellon University

Nathan Beckmann
beckmann@cs.cmu.edu
Carnegie Mellon University

Brandon Lucia
blucia@andrew.cmu.edu
Carnegie Mellon University

ABSTRACT

Ultra-low-power sensor nodes enable many new applications and are becoming increasingly pervasive and important. Energy efficiency is the key determinant of the value of these devices: battery-powered nodes want their battery to last, and nodes that harvest energy should minimize their time spent recharging. Unfortunately, current devices are energy-inefficient.

In this work, we present MANIC, a new, highly energy-efficient architecture targeting the ultra-low-power sensor domain. MANIC achieves high energy-efficiency while maintaining programmability and generality. MANIC introduces *vector-dataflow execution*, allowing it to exploit the dataflows in a sequence of vector instructions and amortize instruction fetch and decode over a whole vector of operations. By forwarding values from producers to consumers, MANIC avoids costly vector register file accesses. By carefully scheduling code and avoiding dead register writes, MANIC avoids costly vector register writes. Across seven benchmarks, MANIC is on average 2.8× more energy efficient than a scalar baseline, 38.1% more energy-efficient than a vector baseline, and gets to within 26.4% of an idealized design.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; **Data flow architectures**.

KEYWORDS

Vector, dataflow, low-power, sensor, energy-harvesting

ACM Reference Format:
Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358277

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10.
https://doi.org/10.1145/3352460.3358277

1 INTRODUCTION

The emergence of tiny, pervasively deployed, ultra-low-power sensor systems enables important new applications in environmental sensing, in- and on-body medical implants, civil infrastructure monitors, and even tiny chip-scale satellites. Existing systems for these applications suffer fundamental inefficiencies that demand new, extremely energy-efficient computer architectures.

Sensing workloads are increasingly sophisticated: Sensor devices collect data from a deployed environment and must process raw data to support applications. Processing varies and may entail digital signal processing (DSP), computing statistics, sorting, or sophisticated computations such as machine learning (ML) inference using a deep neural network (DNN) or a support vector machine (SVM). As processing sophistication has increased, sensor device capability also matured to include high-definition image sensors [61] and multi-sensor arrays [50], increasing sensed data volume.

This shift poses a challenge: how can we perform sophisticated computations on simple, ultra-low-power systems? One design is to offload work by wirelessly transmitting data to a more powerful nearby computer (e.g., at the "edge" or cloud) for processing. In offloading, the more data a sensor produces, the more data the device must communicate. Unfortunately, transmitting data takes much more energy per byte than sensing, storing, or computing on those data [32, 52]. While a high-powered device like a smartphone, with a high-bandwidth, long-range radio, can afford to offload data to the edge or cloud, this is not practical for power-, energy-, and bandwidth-limited sensor devices [26, 32].

Since offloading is infeasible, the alternative is to process data *locally* on the sensor node itself. For example, recent work [32] has shown how systems can use commodity off-the-shelf microcontrollers (COTS MCU) to filter sensed data so that only meaningful data (as defined by the application) are transmitted. Processing data locally at a sensor node eliminates most of the high energy cost of communication, but makes the device highly sensitive to the energy-efficiency of computation.

There are two key criteria that make a computation-heavy sensor system effective. First, the device must process data locally at a low operating power and with *extremely high energy-efficiency*. Second, the device must be *programmable* and general to support a wide variety of applications. These goals are in tension, since programmability often carries a significant energy penalty. Our goal is to design a highly programmable architecture that *hides microarchitectural complexity while eliminating the energy costs of programmability*.

Existing low-power architectures fall short: Ultra-low-power COTS MCUs used in many deeply embedded sensor nodes (e.g., TI MSP430, ARM M0+ & M4+) fail to meet the criteria for an effective sensor node. These MCUs are general-purpose, programmable devices that support a variety of applications. However, COTS MCUs pay a high power, energy, and performance cost for their generality and programmability (see the *COTS MCU* dot in Fig. 1).

Programmability is expensive in two main ways [7, 34, 39]. First, *instruction supply* consumes significant energy: in the best case, the energy of an instruction cache hit, and in the worst case, the energy of a main memory read and instruction cache fill. Lacking sophisticated microarchitectural features such as superscalar and out-of-order execution pipelines [41, 78], the energy overhead of instruction supply constitutes a significant fraction of total operating energy. Second, data supply through *register file (RF) access* also consumes significant energy. Together, we find that instruction and data supply consume 54.4% of the average execution energy in our workloads.

Programming pitfalls of architectural specialization: To combat the energy costs of generality, some recent work has turned to microarchitectural specialization, making a system energy-efficient at the expense of generality and programmability [13–15, 27, 51, 80]. Specialization customizes a system's control and datapath to accommodate a particular workload (e.g., deep neural networks [13, 15]), eliminating inessential inefficiencies like instruction supply and RF access. The downside of specialization is its inability to support a wide range of applications (see the *ASIC* dot in Fig. 1).

In contrast to specialization, another approach to programmable energy-efficiency is to target a conventional vector architecture (such as NVidia's Jetson TX2 [64], ARM NEON [2], or TI LEA [42]), amortizing the cost of instruction supply across a large number of compute operations. Unfortunately, vector architectures exacerbate the energy costs of RF access, especially in high-throughput designs with multi-ported vector register files (VRFs) [3, 49, 65], and so remain far from the energy-efficiency of fully specialized designs [34] (see the *classic vector* dot in Fig. 1).

The ELM architecture stands out among prior efforts as an architecture that targets ultra-low-power operation, operates with extremely high energy-efficiency, and retains general-purpose programmability [5, 7]. The key to ELM's efficiency is an *operand forwarding* network that avoids latching intermediate results and a distributed RF that provides sufficient register storage, while avoiding unfavorable RF energy scaling. Unfortunately, despite these successes, ELM faces fundamental limitations that prevent its widespread adoption. ELM makes significant changes to the architecture and microarchitecture of the system, requiring a full re-write of software to target its exotic, software-managed RF hierarchy and instruction-register design. This programming task requires expert-level assembly hand-coding, as compilers for ELM are unlikely to be simple or efficient; e.g., ELM itself cites a nearly 2× drop in performance when moving from hand-coded assembly to compiler-generated assembly [5]. While ELM supports general-purpose programs, it does so with a high programmability cost and substantial changes to software development tools (as shown in Fig. 1).

Our design and contributions: In this work we present MANIC: an efficient vector-dataflow architecture for ultra-low-power embedded systems. As depicted in Fig. 1, MANIC is closest to the *Ideal* design, achieving high energy-efficiency while remaining general-purpose and simple to program. MANIC is simple to program because it exposes a standard vector ISA interface based on the RISC-V vector extension [69].

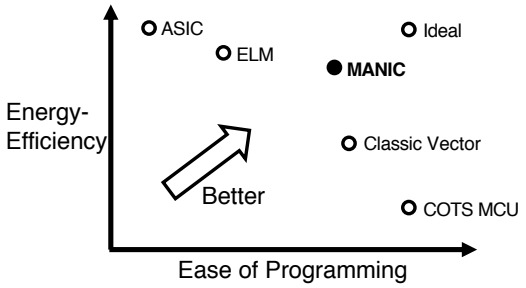


Figure 1: MANIC seeks to improve energy efficiency without compromising programmability.

MANIC achieves high energy-efficiency by eliminating the two main costs of programmability through its vector-dataflow design. First, **vector** execution amortizes instruction supply energy over a large number of operations. Second, MANIC addresses the high cost of VRF accesses through its **dataflow** component by forwarding operands directly between vector operations. MANIC transparently buffers vector outputs in a small forwarding buffer and, at instruction issue, renames vector operands to directly access the forwarding buffer, *eliminating read accesses to the VRF*. Additionally, MANIC extends the vector ISA with **kill annotations** that denote the last use of a vector register, *eliminating write accesses to the VRF*. The vector-dataflow architecture is efficient because MANIC amortizes the energy of tracking dataflow across many vector operations. MANIC thus eliminates a large fraction of VRF accesses (90.1% on average in our experiments) with simple microarchitectural changes that leave the basic vector architecture intact.

Finally, we have designed and implemented a code scheduling algorithm that exploits MANIC's operand forwarding to minimize VRF energy, while being *microarchitecturally agnostic*. In other words, it is *not* necessary to expose the details of the pipeline architecture or size of forwarding buffers to minimize VRF energy—a single code schedule is near-optimal across a range of microarchitectural design points.

We implement MANIC fully in RTL and use industry-grade CAD tools to evaluate its energy efficiency across a collection of programs appropriate to the deeply embedded domain. Using post-synthesis energy estimates, we show that MANIC is within 26.4% of the energy of an idealized design while remaining fully general and making few, unobtrusive changes to the ISA and software development stack.

2 RELATED WORK

High-data-rate embedded sensors demand energy-efficient computation. Battery-powered and energy-harvesting systems are primarily constrained by *energy-efficiency*, not performance. Despite increases in capability and efficiency, prior systems compromise either programmability, energy-efficiency, or both. Prior work on energy-efficient architecture uses datapath specialization, vector execution,

or compiler support, but does not apply to deeply-embedded, ultra-low-power systems. This section discusses these prior efforts to motivate MANIC and give context for our contributions.

2.1 Ultra-low-power embedded systems

An ultra-low-power embedded device combines sensors, compute components, and radios, and many are designed to capture data and send them over a radio link to a base station for offloaded processing. The offload model requires the sensor to communicate all sensed data. Unfortunately, communication over long distances has a high energy cost [26], creating a strong incentive to process sensed data locally.

Battery-powered sensing devices: Some devices are battery-powered [22, 44, 70] which limits lifetime and device duty cycle. With a single-charge battery, energy-efficiency determines lifetime until depletion. With a rechargeable battery, the number of recharge cycles limits lifetime. Recent work [44] suggests that batteries show little promise for compute-intensive workloads on COTS MCUs. Even a simple data-logger lasts only a few years deployed despite very sparse duty cycling and performing almost no computation. As duty-cycle and computational intensity increase (e.g., for machine learning on high-rate sensor data), lifetime will drop to weeks or days, limiting device applicability.

Energy-harvesting sensing devices: Energy-harvesting systems collect operating energy from their environment and use capacitive energy storage instead of a battery, eliminating complexity, duty-cycle, and lifetime limitations of batteries [19, 35, 36, 72, 87]. Energy-harvesting devices operate intermittently after buffering sufficient energy and otherwise sleep, waiting for energy. An energy-harvesting system's value is determined by the amount of work it can perform in a tiny energy budget, as once the energy is spent, the device turns off and must recharge.

Recent software and platform results [9, 10, 18, 19, 30, 32, 36, 37, 45, 53, 54, 56–58, 71, 79, 86] show that intermittently powered systems can execute sophisticated applications despite these limitations, and recent work has developed microarchitectural support for correct intermittent execution [38, 55, 59]. These prior efforts focused primarily on ensuring correct execution in intermittent systems, with few [30, 32, 54] optimizing performance and energy.

Relevance to MANIC: We observe that for deeply embedded systems and energy-harvesting systems, *value is largely determined by energy efficiency*. A more energy-efficient system operates more frequently and does more per operating period. Studying the frequency of power failures and the typically very low cost of JIT checkpointing [45, 58, 60], we conclude that microarchitectural support for correctness is not the most urgent research question in an intermittent architecture; architects should focus on efficiency first.

2.2 Vector architecture

MANIC's vector-dataflow design is informed by a long history of vector and dataflow architectures. Early vector machines exploited vector operations for supercomputing [21] and most commercially available architectures support vectors (e.g., AVX [28] and GPUs [20]). These vector designs target performance and operate at a power budget orders-of-magnitude higher than an ultra-low-power device. MANIC focuses on low-power operation, which leads to different choices than performance-optimized designs.

Vector execution is data-parallel and provides the energy-efficiency benefit of amortizing instruction supply energy (fetch, decode, and issue) across many operations. Unfortunately, a vector design requires a large vector register file (VRF), exacerbating register file access cost, especially in designs that require a VRF with many ports. Reducing VRF cost and complexity has been a primary focus of prior vector designs [3, 49].

T0 [3, 83] is a vector architecture with reconfigurable pipelines. Software controls datapaths to chain operations, eliminating VRF access within a chain. However, microarchitectural details of the datapath are exposed to software, requiring major software changes and recompilation.

CODE [49] reduces VRF cost by distributing the VRF among heterogeneous functional units. This design is transparent to software because CODE renames operands at instruction issue to use registers near an appropriate functional unit. Distribution lets CODE reduce VRF ports, but requires a routing network to send values between functional units.

Relevance to MANIC: Like this prior work, MANIC uses vector execution to reduce instruction supply overhead, but also includes additional techniques to lower VRF cost. MANIC reduces VRF ports by focusing on efficiency, not performance: MANIC uses a single functional unit per lane, requiring a minimum of VRF ports (2 read, 1 write). VRF access remains expensive, however, requiring MANIC to avoid VRF access when possible. MANIC's vector-dataflow execution model achieves this by relaxing the ordering of a typical vector execution, similar to SIMT in GPUs. Like T0, MANIC forwards operands to eliminate VRF access, but, unlike T0, does so transparently to software. Like CODE, MANIC renames operands to hide microarchitectural complexity, but, unlike CODE, does so to eliminate VRF access. It is this combination of techniques that lets MANIC achieve ultra-low-power operation without complicating software.

2.3 Dataflow architecture

MANIC eliminates VRF accesses by forwarding operands between instructions according to dataflow. Dataflow machines have a long history [23–25, 62] that includes changes to the programming and execution model to eliminate control and data movement overheads. More recent efforts identify dataflow locality [73, 77] as a key determinant in sequential code. Ample prior work in out-of-order (OoO) execution engines (i.e., restricted dataflow) uses *operator fusion* to improve performance and reduce RF pressure [4, 12, 48, 74, 75].

RSVP [17] uses SIMD execution, specialized for dataflow operation, focused on performance. However, RSVP requires writing programs in a custom dataflow language and only targets streaming workloads. Dyser [33], Plasticine [68], and Stream-dataflow [63] have revived spatial dataflow, enabling programmable designs at ASIC-like efficiencies. Their key drawback is the need to compile programs directly to a spatial fabric, which precludes microarchitectural change.

ELM [5] is perhaps the most related work to MANIC. ELM is a custom microarchitecture designed for low-power, embedded

operation. ELM uses restricted SIMD execution and operand forwarding to provide dataflow-like execution. ELM's complex register file hierarchy and forwarding mechanism are software-controlled, exposing microarchitectural details to the programmer and requiring expert-level, hand-coded assembly for maximum efficiency. Even with significant changes to the compiler toolchain, ELM poses a risk of unpredictable performance and high programming cost.

Relevance to MANIC: Like the work above, MANIC seeks to exploit dataflow to improve efficiency and, like RSVP and ELM, uses SIMD in order to amortize instruction supply cost. MANIC also relies upon operand forwarding like ELM to avoid RF reads and writes. However, unlike this prior work, MANIC seeks to hide microarchitectural complexity from the programmer. MANIC uses dataflow to inform control of a single lane functional unit instead of opting for a spatial fabric like in Dyser, Plasticine, and Stream-dataflow. And unlike ELM, which has a significant programming cost, MANIC relies on the standard vector extension to the RISC-V ISA with only a few optional changes.

2.4 Register file optimization

MANIC reduces vector register file energy by leveraging dataflow to avoid register file reads and writes. There are a number of prior works that also identify the register file as a performance and/or energy bottleneck.

In the GPU domain, Gebhart et al. and Jeon et al. observe that producers often have only a single consumer and consumers arrive shortly after a producer [31, 46]. Gerbhart et al. introduces a RF cache and operand buffer to forward values directly between instructions, reducing RF pressure [31, 84]. RF virtualization for GPUs has also been explored as a way to improve RF utilization [46, 81, 82]. These systems remap registers by dynamically allocating and deallocating registers among running threads in order to expose additional parallelism [81, 82] or to reduce the number of physical registers while maintaining performance [46].

For CPU designs, dead-value prediction and operator fusion can be used to reduce RF reads and writes. Dead-value prediction reduces RF pressure by identifying values that do not need to be persisted to the register file because they are not read again [8, 66]. Finally, compilers can mark dead values [47, 82] so that hardware can deallocate and free the physical register earlier.

Relevance to MANIC: MANIC exploits the same pattern of register liveness (i.e., values are not alive for very long) identified in prior work. However, instead of exploiting this pattern to improve performance or RF utilization, MANIC focuses on eliminating RF accesses to save energy. MANIC remaps registers similar to RF virtualization, but operates at a much finer granularity (remapping instruction-by-instruction vs. at program phases/thread barriers) and remaps registers explicitly following dataflow (vs. caching them). Lastly, MANIC uses compiler-generated dead-value hints to avoid VRF writes; to the best of our knowledge, it is the first to use compiler-generated hints in a vector design. Prior work on dead-value prediction is inapplicable to ultra-low-power designs because it requires expensive speculative recovery mechanisms to handle mispredictions.

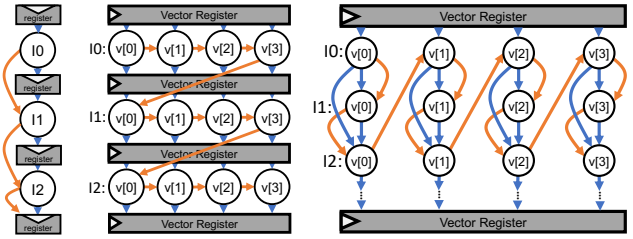


Figure 2: Different execution models. Orange arrows represent control flow, blue arrows represent dataflow. MANIC relies on vector-dataflow execution, avoiding register accesses by forwarding and renaming.

3 VECTOR-DATAFLOW EXECUTION

MANIC implements the *vector-dataflow* execution model. There are two main goals of vector-dataflow execution (Fig. 1). The first goal is to provide general-purpose programmability. The second goal is to do this while operating efficiently by minimizing instruction and data supply overheads. Vector-dataflow achieves this through three features: *(i)* vector execution, *(ii)* dataflow instruction fusion, and *(iii)* register kill points.

3.1 Vector execution

The first main feature of MANIC's execution model is vector execution. Vector instructions specify an operation that applies to an entire vector of input operands (as in ample prior work discussed in Sec. 2). The key advantage of vector operation for an ultra-low-power design is that control overheads imposed by each instruction — instruction cache access, fetch, decode, and issue — amortize over the many operands in the vector of inputs. Vector operation dramatically reduces the cost of instruction supply and control, which is a primary energy cost of general-purpose programmability. Vector operation is thus a key ingredient in MANIC's energy-efficiency.

Fig. 2 illustrates the difference between scalar execution and vector execution. Fig. 2a executes a sequence of instructions in a scalar fashion. Blue arrows show dataflow and orange arrows show control flow. Instructions proceed in sequence and write to and read from the register file to produce and consume outputs and operands. Fig. 2b executes the same sequence of instructions in a vector execution. The execution performs the vector instruction's operation on each element of the vector in sequence, consuming operands from and producing outputs to the register for each operation over the entire vector. Control proceeds *horizontally* across each of the vector's elements for a single vector instruction before control transfers *vertically* to the next vector instruction. Vector execution amortizes the control overhead of a scalar execution because a single instruction corresponds to an entire vector worth of operations.

3.2 Dataflow instruction fusion

The second main feature of MANIC's execution model is *dataflow instruction fusion*. Dataflow instruction fusion identifies windows of contiguous, dependent vector instructions. Dataflow instruction fusion eliminates register file reads by directly forwarding values between instructions within the window. Comparing to a typical vector machine illustrates the benefit of dataflow instruction fusion. In a typical vector machine, instructions execute independently and

each operation performs two vector register file reads and one vector register file write. Accessing the vector register file has an extremely high energy cost that scales poorly with the number of access ports [7, 49]. With dataflow instruction fusion, each instruction that receives a forwarded input avoids accessing the expensive vector register file to fetch its input operands. Avoiding these reads reduces the total energy cost of executing a window of vector instructions.

Fig. 2c illustrates the difference between vector execution and vector-dataflow execution in MANIC. Vector-dataflow first identifies data dependencies among a sequence of vector instructions in a fixed-size instruction window. After identifying dependences between instructions in the window, MANIC creates an efficient dataflow forwarding path between dependent instructions (using the forwarding mechanism described in Sec. 4). Fig. 2c shows a window of dependent operations made up of instructions `I0`, `I1`, and `I2`. Execution begins with the first vector instruction in the window (`I0`) and the first element of the vector (`v[0]`). However, unlike a typical vector execution, control transfers *vertically* first, next applying the second vector instruction to the first vector element. The orange arcs illustrate vertical execution of `I0`, then `I1`, then `I2` to the vector inputs represented by `v[0]`. After vertically executing an operation for each instruction in the window for `v[0]`, the orange arcs show that control steps horizontally, executing the same window of operations on the next element of the vector, `v[1]`. The blue arrows illustrate the dataflow forwarding captured by vertical execution in a window of vector-dataflow execution. The blue arrow from `I0` to `I2` shows that the value produced by `I0` is forwarded directly to `I2` without storing the intermediate result in the vector register file.

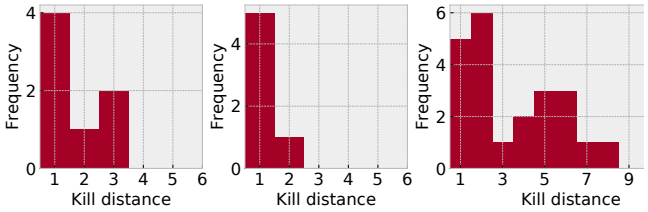
3.3 Vector register kill points

The third main feature of MANIC's execution model is its use of *vector register kill points*. A vector register is *dead* at a particular instruction if no subsequent instruction uses the value in that register. Hence, a dead value need not be written to the vector register file. The instruction at which a vector register becomes dead is the *kill point* for that register. Though MANIC forwards values between dependent instructions without going through the vector register file, MANIC normally must write each operand back to the vector register file because the operand may be used later in a later window.

However, if a program explicitly informs MANIC of each register's kill points, then MANIC can eliminate register file writes associated with those registers. We propose to tag each of an instruction's operands with an optional *kill bit* that indicates that the register is dead at that instruction, and its value need not be written back to the vector register file. Kill bits do not affect programmability because they are optional, a compiler analysis to identify dead registers is simple, and kill bits do not expose microarchitectural details, such as the size of MANIC's instruction window.

3.4 Applications benefit from vector-dataflow

We studied the core compute kernels in a wide variety of sensor node applications and found abundant opportunities for vector-dataflow execution. Regardless of MANIC's window size, an application has more exploitable vector dataflows if its sequences of dependent instructions tend to be shorter. The length of a dependent instruction sequence is characterized by the distance (or number of instructions) between a when register's value is produced and when that register is killed (the kill point). We deem this the *kill distance*. Shorter kill distances require fewer resources for forwarding in a window and make a window of any size more effective.



(a) SpMV *(b) Sparse conv* *(c) FFT*

Figure 3: Histograms of kill distances for three different applications. Distances skew left, suggesting values are consumed for the last time shortly after being produced.

We statically measured the distribution of kill distances for all registers in the inner loops of three kernels. The histograms shown in Fig. 3 suggest that kill distances tend to be short and that a reasonably small (and thus implementable) window size would capture dependencies for these kernels.

3.5 Synchronization and memory consistency

In MANIC, the vector unit runs as a loosely-coupled co-processor with the scalar core. As a result, MANIC must synchronize vector and scalar execution to ensure a consistent memory state. A typical sequentially consistent model would require frequent stalls in the scalar core to disambiguate memory and, worse, would limit the opportunity for forwarding in the vector unit. These issues could be avoided with microarchitectural speculation, including load-store disambiguation and mis-speculation recovery mechanisms, but we judge such mechanisms too expensive for ultra-low-power applications. Moreover, in practice, the scalar core and the vector unit rarely touch the same memory during compute-intensive program phases, so the mechanisms would be largely unused.

Instead, we add a new `vfence` instruction that handles both synchronization and memory consistency. `vfence` stalls the scalar core until the vector unit completes execution with its current window of vector-dataflow operations. MANIC's use of `vfence` operations is very similar to memory fences for concurrency in x86, ARM, and other widely commercially available processors [29]. Properly used, `vfence` operations cause the scalar and vector cores' executions to be sequentially consistent. In practice, this often means inserting a `vfence` at the end of the kernel.

As with any system relying on fences, the programmer is responsible for their correct use (i.e., avoiding data races). Relying on the programmer to avoid data races is practical since compilers struggle with alias analysis, reasonable because `vfences` are rare, and consistent with common practice in architectures and high-level programming languages [11, 43].

4 MANIC: ULTRA-LOW-POWER VECTOR-DATAFLOW PROCESSING

MANIC is a processor microarchitecture that implements the vector-dataflow execution model to improve energy efficiency while maintaining programmability and generality. MANIC's hardware/software interface is the most recent revision of the RISC-V ISA vector

extension [69]. MANIC adds a vector unit with a single lane to a simple, in-order scalar processor core. The vector unit has a few simple additions to support vector-dataflow execution: instruction windowing hardware and a renaming mechanism together implement forwarding between dependent instructions. With no modifications to the ISA, MANIC runs programs efficiently. With a minor ISA change, MANIC further improves efficiency by conveying register kill annotations; the microarchitecture uses these annotations to kill registers instead of incurring the cost of writing them to the vector register file.

4.1 Vector ISA

The software interface to MANIC's vector execution engine is the RISC-V ISA vector extension [69] and RISC-V code[1] will run efficiently on a MANIC system with only minor modifications to add `vfence` instructions for synchronization and memory consistency.

A programmer may further optionally recompile their code using our custom MANIC compiler to use minor ISA changes that support code scheduling and vector register kill annotations. We emphasize that these compiler-based features do not require programming changes, do not expose microarchitectural details, and are optional to the effective use of MANIC.

MANIC implements the RISC-V V vector extension. RISC-V V does not specify a fixed number of vector registers, but its register name encoding includes five bits for vector register names. We implement 16 vector registers, requiring four bits to name, and leaving a single bit in the register name unused. We use the extra bit in a register's name to convey kill annotations from the compiler to the microarchitecture. If either of an instruction's input registers has its high-order bit set, the encoded instruction indicates to MANIC that the register dies at the instruction. To support code scheduling, MANIC's optional compiler support runs the dataflow code scheduling algorithm (described in Sec. 4.5). After scheduling, the compiler analyzes definitions and uses of each register and adds a kill annotation to a killed register's name in the instruction at which it dies.

4.2 Microarchitecture

The foundation of MANIC's microarchitecture is an extremely simple, in-order, single-issue vector core with a single execution lane that is equipped with a single functional unit. MANIC adds four components to this base vector core to support vector-dataflow execution: (1) issue logic and a register renaming table; (2) an instruction window buffer; (3) an xdata buffer and (4) a forwarding buffer. Fig. 4 shows an overview of MANIC's design and how these components relate to one another.

Issue logic and register renaming: MANIC's issue logic is primarily responsible for creating a window of instructions to execute according to vector-dataflow. The issue logic activates once per window of instructions, identifying, preparing, and issuing for execution a window of dependent instructions over an entire vector of inputs. A key parameter of the issue logic is the length of the instruction buffer window, which we explain next. The issue logic analyzes a short sequence of instructions that has the same number of instructions as the instruction buffer can hold. The issue logic identifies dataflow

[1] Specifically, RISC-V E extension, which uses 16 architectural registers.

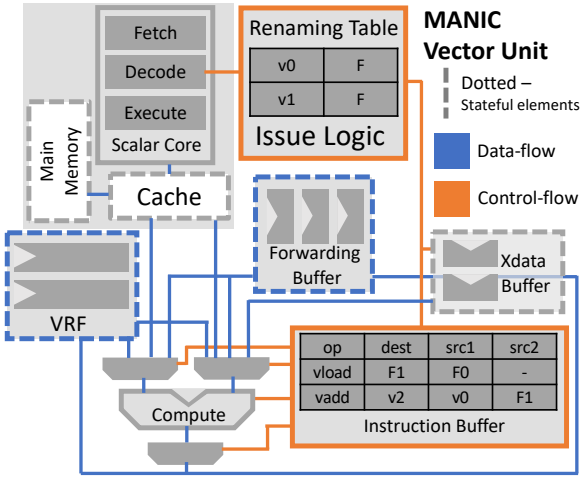


Figure 4: A block diagram of MANIC's microarchitectural components (non-gray region). Control flow is denoted with orange, while blue denotes dataflow. Stateful components (e.g. register files) have dotted outlines.

between instructions by comparing the names of their input and output operands. If two instructions are dependent — the output of one of the instructions is the input of another — MANIC should forward the output value directly from its producer to the input of the consumer, avoiding the register file. MANIC's issue logic implements forwarding by renaming the instructions' register operands to refer to a free location in MANIC's forwarding buffer, instead of to the register file. The issue logic records the renaming in MANIC's renaming table, which is a fixed-size, directly-indexed table, with one entry for each register that can be renamed in a window of instructions. After the issue logic identifies dependent operations and performs renaming for a window of operations, it dispatches the window of operations for execution.

Instruction window: MANIC uses its instruction window to store an issued window of instructions that have had their register operands renamed by the issue logic. The instruction window and its associated control logic (not pictured) determine what operation MANIC's single functional unit should execute next. The instruction window's control logic executes the operation represented by each instruction stored in the buffer. As Sec. 3 describes, execution proceeds first vertically, through the entire window, and then horizontally through the vector. A key feature of the instruction window's control logic is its ability to select an operand's source or destination. For input operands, the instruction window controls whether to fetch an operand from the vector register file or from MANIC's forwarding buffer (in the case of an operand being forwarded between instructions in the window). Likewise, for output operands, the instruction window controls whether to write an output operand to the vector register file, to the forwarding buffer, or to both.

Limits to window size: There are several classes of instructions that limit window size. These include stores, permutations, and reductions. Permutations and reductions require interactions between elements in a vector, which creates a *horizontal* dependence between operations on different vector elements. MANIC does not support forwarding for such operations because of the complexity

of the dependence tracking that they introduce. Instead, these operations execute one element at a time, ultimately writing to the vector register file.

A store also ends issuing for a window. A store may write to a memory location that a later operation loads from. Such a through-memory dependence is unknown until execution time. Consequently, MANIC conservatively assumes that the address of any store may alias with the address of any load or store in the window (i.e., in a later vector element). A store ends the construction of a window to avoid the need for dynamic memory disambiguation to detect and avoid the effect of such aliasing. We evaluated adding a *non-aliasing store* instruction that would allow MANIC to forward past stores, but this instruction improved energy-efficiency by less than 0.5% in our applications. This is because store instructions often naturally close windows (e.g. a `vfence` follows the store to ensure correctness). Thus, given the added programming complexity for minimum benefit, we conclude that such an instruction is unnecessary.

Xdata buffer: Some instructions like vector loads and stores require extra information (e.g. base address and stride) available from the scalar register file when the instruction is decoded. Due to the loosely coupled nature of MANIC, this extra information must be buffered alongside the vector instruction. Since not all vector instructions require values from the scalar register file, MANIC includes a separate buffer, called the xdata buffer, to hold this extra information. Entries in the instruction buffer contain indices into the xdata buffer as needed. During execution, MANIC uses these indices to read information from the xdata buffer and execute accordingly.

Forwarding buffer: The forwarding buffer is a small, directly-indexed buffer that stores intermediate values as MANIC's execution unit forwards them to dependent instructions in the instruction window. The issue logic lazily allocates space in the forwarding buffer and renames instruction's forwarded operands to refer to these allocated entries. The benefit of the forwarding buffer is that it is very small and simple, which corresponds to a very low static power and access energy compared to the very high static power and access energy of the vector register file. By accessing the forwarding buffer instead of accessing the vector register file, an instruction with one or more forwarded operands consumes less energy than one that executes without MANIC.

Efficient reductions: RISC-V V contains reduction instructions like `vredsum v1 v2`, which adds up all elements of `v2` and writes the sum into the first element of `v1`. MANIC relies on the forwarding buffer to avoid VRF accesses for reductions. Instead of writing partial results to the VRF, MANIC allocates space in the forwarding buffer for partial accumulation. The decode logic recognizes a reduction, allocates space, and remaps the second source operand and the destination to point to the entry in the forwarding buffer. During execution, MANIC will then use the partial result in the forwarding buffer as one source for the reduction (e.g., sum) and overwrite it with the new value as it is produced. This optimization re-purposes MANIC's existing dataflow mechanisms to save an entire vector-length of VRF reads and writes for reductions.

Structural hazards: There are three structural hazards that cause MANIC to stop buffering additional instructions, stall the scalar core, and start vector execution. The first hazard occurs when the instruction buffer is full and another vector instruction is waiting to be buffered. The second hazard occurs when all slots in the forwarding buffer are allocated and an incoming instruction requires a slot. Finally, the third hazard occurs when the xdata buffer is full and a decoded vector instruction requires a slot. The prevalence of each hazard depends on the size of the buffers associated with each. The first hazard is most common, while the other two tend to be rare.

4.3 Memory system

MANIC includes an instruction cache (icache) and a data cache (dcache). This departs from the designs of many commercial microcontrollers in the ultra-low-power computing domain, which do not have dcaches and have extremely small icaches on the order of 64 bytes [41]. However, we find that even small or moderately sized dcaches (512B) are effective in minimizing the number of accesses to main memory. We measured miss curves for the seven different application we consider; for each application there is an extreme drop-off in the number of misses for even small cache sizes, and with a 512B cache the curves are basically flat. Since the energy of an access to main memory dwarfs an access to the dcache, the dcache offers a significant reduction in energy.

Caching and intermittence: In the intermittent computing domain, improperly managed caches may lead to memory corruption because dirty data may be lost when power fails. As such, MANIC assumes a hardware-software JIT-checkpointing mechanism (like [9, 45, 58]) for protecting the caches and any dirty data. Checkpointing energy for cached data is virtually negligible because caches are very small relative to the operating period.

4.4 Putting it together with an example

We illustrate the operation of the issue logic, renaming table, instruction window, and forwarding buffer with an example of MANIC's operation, shown in Fig. 5. The figure starts with vector-aware assembly code that MANIC transforms into vector-dataflow operations by populating the renaming table and instruction buffer with information about the dataflow. Vector assembly instructions pass into MANIC's microarchitectural mechanisms as they decode to the issue logic and later execute.

Issuing instructions and renaming operands: The figure shows a three-instruction program and illustrates how the issue logic populates the instruction buffer and remaps registers for each instruction.

- `vload`: The issue logic records the load in the instruction window and, since the instruction is a vector load and requires a base address, also inserts the base address (`&a` forwarded from the scalar register file) into the xdata buffer. In addition, the issue logic writes an empty renaming entry to `v0` in the renaming table along with the index of the instruction in the instruction buffer. An empty renaming entry at execution time signifies a vector register write. However, during issue, an empty entry may be filled by an instruction added to the instruction window later during the same issue phase.
- `vmul`: The multiply instruction consumes two register operands that are not in the renaming table and, at execution time, will issue two vector register file reads. As with the load, the issue logic records the multiply's output register with an empty entry in the renaming table as well as the index of the multiply in the instruction buffer.

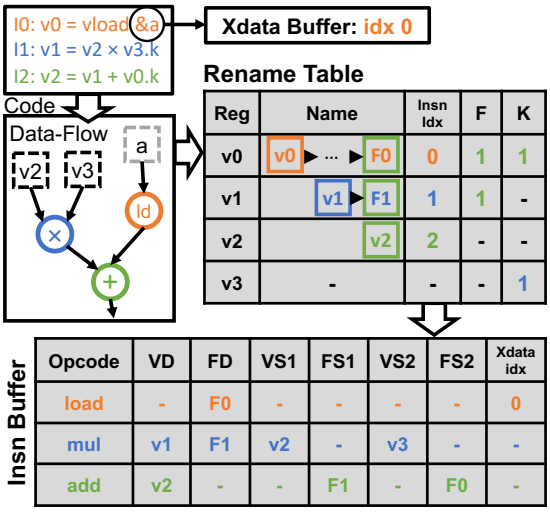


Figure 5: MANIC's issue logic constructs windows of instructions with dataflow. The rename table keeps track of registers and names, updating the instruction buffer when new opportunities for forwarding are identified.

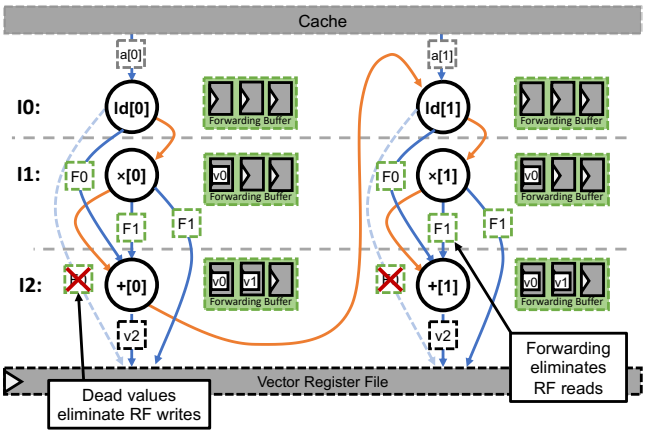


Figure 6: MANIC's microarchitecture components execute a window of instructions using forwarding according to dataflow across an entire vector of input.

- `vadd`: The add's inputs are `v0` and `v1` with the kill annotation indicating that the instruction kills register `v0`. The issue logic looks up each input operand in the renaming table and, finding both have valid entries, identifies this instruction as the target for forwarding. The issue logic remaps `v0` to refer to the first entry of the forwarding buffer and `v1` to the second position. The load instruction in the instruction buffer (found by the saved index in the renaming table) is updated and will store its result in `F0` instead of `v0`. Similarly, the multiply instruction is also updated and will store its result in `F1`, but since `v1` is not killed, it will still be written-back to the register file. The add instruction then will fetch its input operands from `F0` and `F1` instead of the vector register file. The kill annotations associated with `v3` and `v0` follow the re-written instructions into the instruction window, enabling their use during execution to avoid register file writes.

Executing a window of instructions: After issue, the window of instructions is ready to execute. Fig. 6 shows (via the orange control-flow arcs) how MANIC executes the entire window vertically for a single vector element before moving on to execute the entire window for the second vector element. The blue dataflow arcs show how MANIC forwards values between dependent instructions using its forwarding buffer. The green squares marked with "F" names are forwarded values. The figure also shows how MANIC uses a kill annotation at runtime. The registers with kill annotations (`v0` and `v3`) need not be written to the vector register file when the window completes executing, sparing the execution two vector register writes required by a typical vector execution.

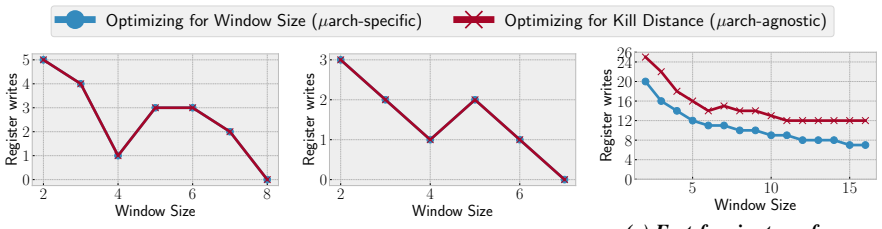
4.5 Microarchitecture-agnostic dataflow scheduling

MANIC's final feature is *microarchitecture-agnostic dataflow scheduling*. This feature is optional compiler support that re-orders vector instructions to make dependent operations as close as possible to one another. If dependent operations are closer together in an instruction sequence, then it is more likely that they will appear together in one of MANIC's vector-dataflow windows. By re-ordering operations to appear close together in a window, MANIC creates more opportunities to forward values from a producer instruction to its consumer, eliminating more vector register file accesses.

MANIC's dataflow scheduler does not compromise programmability or generality. The programmer need not understand the microarchitecture to reap the benefits of the dataflow scheduler. The dataflow scheduler minimizes the forwarding distance between dependent instructions, rather than targeting a particular window size. While not always optimal for a given window size, this microarchitecture-agnostic optimization prevents the compiler from being brittle or dependent on the microarchitectural parameters of a particular system.

To minimize forwarding distance between dependent instructions, MANIC's dataflow code scheduler uses *sum kill distance*. A vector register's kill distance is the number of instructions between when an instruction defines the register and when the value in the register is used for the last time (i.e., the register dies). The sum kill distance is the sum of all registers' kill distances across the entire program. To remain agnostic to the window size of particular MANIC implementation, the code scheduler minimizes the sum kill distance (which is equivalent to minimizing average kill distance). Sum kill distance is a proxy for the number of register writes in a program because if a register does not die during a window's execution, the system must write its value back to the register file. When sequences of dependent instructions are closer together, their intermediate values die more quickly, because registers need not remain live waiting for unrelated instructions to execute. A larger window accommodates dependence chains that include longer kill distances.

We implement dataflow code scheduling using brute force (exhaustive) search for small kernels containing fewer than 12 vector operations. For larger kernels (e.g., FFT), we implement dataflow code scheduling via simulated annealing that randomly mutates instruction schedules, while preserving dependences, to produce a new valid schedule, accepting this new schedule with some probability.



(a) Sparse matrix multiply *(b) Sparse convolution* *(c) Fast-fourier transform*

Figure 7: Code scheduling is microarchitecturally agnostic – minimizing the sum of kill distances is good proxy for minimizing register writes for specific window size.

Parameter	Possible Values	Optimal
Core Frequency	100 MHz	100 MHz
Scalar Register #	16	16
Vector Register #	16	16
Vector Length	16/32/64	64
Window Size	8/16/32	16
Main Memory Size	64 KB	64KB
Cache Line Size	4/8 B	4B
DCache Size	128/256/512 B	256B
DCache Associativity	1/2/4/8	8
ICache Size	64/128/256 B	128B
ICache Associativity	1/2/4/8	2

Table 1: Microarchitectural parameters.

Fig. 7 shows that the microarchitecture-agnostic minimization of the sum kill distance closely approximates a microarchitecture-specific approach that optimizes for a particular window size. The plot shows the number of register writes made by one iteration of the kernel's inner loop for a given window size using code optimized by the two different optimization criteria. The blue line shows the number of register writes of a *microarchitecture-specific* schedule, where window size is exposed to the compiler. The red line shows the number of writes for our *microarchitecture-agnostic* schedule based on sum kill distance. The two curves generally agree, suggesting that minimizing sum kill distance eliminates register writes with similar efficacy as when window size is exposed explicitly to the compiler. For the FFT kernel, the instruction window is broken by stores and permutations (Sec. 4.2), causing additional vector register file writes. This is a limitation of optimizing only for sum kill distance that we plan to address in future work.

5 METHODOLOGY

We implement MANIC entirely in RTL and evaluate the system using post-synthesis timing, power, and energy modeling.

5.1 Full-stack approach

We take a full-stack approach to implementing MANIC. We build a simulation infrastructure based on Verilator [76] that allows us to run full applications on top of MANIC. This infrastructure includes a custom version of Spike, modifications to the assembler, a custom LibC, a custom bootloader, a cache and memory simulator, and RTL for both MANIC and its five-stage pipelined scalar core. We develop a custom version of Spike [1] (the gold standard for functional RISC-V simulation) with support for vector instructions to verify RTL simulation. We extend the GNU assembler to generate the correct bit encodings for the RISC-V vector extension instruction set. We build a custom LibC and bootloader to minimize the overhead of startup and to support our architecture. We use a cache and memory simulator to model timing for loads and stores. Finally, we use a test harness built with Verilator to run full-application, cycle-accurate RTL simulations.

Post-synthesis modeling: We synthesize MANIC using the Cadence Genus Synthesis solution and a 45nm standard cell library. Genus generates post-synthesis timing, area, and generic power estimates. To get a better power estimate, we annotate switching for each application. Towards this end, we use our Verilator test harness to generate a SystemVerilog testbench for each application. Then we generate a value-change dump (VCD) file, using the Cadence Xcelium RTL simulator, our application testbench, and the post-synthesis RTL netlist. The VCD file represents switching of all internal signals in the netlist. Finally, we feed the VCD file back into Genus and leverage the Joules power estimation tool to estimate leakage and dynamic power for the entire application. For some applications, the VCD file that covers the entire execution of the application is too large to store. In these instances, we generate a VCD file that is representative of a region of interest.

Modeling memory energy: We model the power, energy, and latency of main memory, caches, and the vector register file using Destiny [67]. We use Destiny's SRAM RAM model for the vector register file and use its SRAM cache model for the caches. A large, non-volatile main memory is a prerequisite for an intermittently-powered system, which is one potential target of MANIC. We model main memory using Destiny's STT-MRAM model, because STT-MRAM has favorable energy characteristics and has high write endurance compared to alternatives like Flash and RRAM [16, 40, 85]. Each memory model was configured to optimize for low leakage using the LSTP device roadmap and 45nm technology node.

5.2 Baselines and applications

We compare MANIC to two baselines: an in-order, single-issue scalar core architecture and a simple vector architecture. We implement both baselines in RTL and follow the same process detailed previously for MANIC for estimating energy and power. The scalar and vector baselines share a scalar core model with MANIC (a simple, in-order, five-stage-pipelined core, representative of ultra-low-power systems). Each implementation also has separate instruction and data caches. The vector baseline is similar to MANIC, possessing a small instruction buffer, but, unlike MANIC, iterates through each vector instruction element by element and writes intermediate results to the vector register file. We also consider an idealized design that operates with a similar datapath as MANIC, but eliminates the main costs of programmability, namely instruction fetch/decode and scalar/vector register file access.

Design space exploration: For the baselines and MANIC as well as different configurations of each, we conduct a design space exploration to optimize microarchitectural parameters to reduce total energy. Holding all but one parameter constant, we identify the

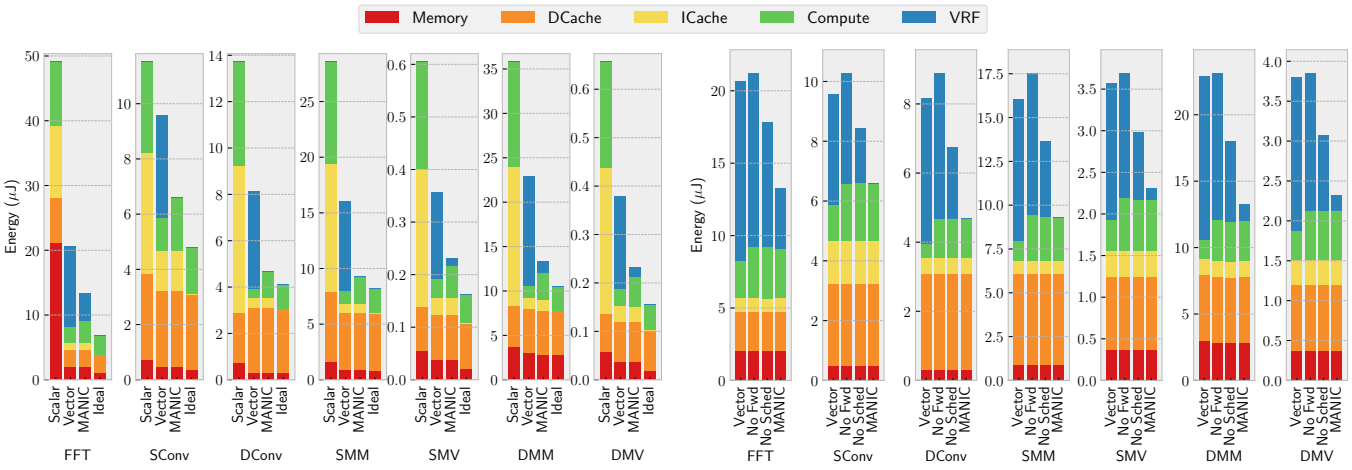


Figure 8: Full system energy of MANIC against various baselines across seven different applications. Bars (from left-to-right): scalar baseline, vector baseline, MANIC, and an idealized vector design with no instruction or data supply energy. MANIC is within 26.4% of the ideal design and is overall 2.8× more energy efficient than the scalar baseline and 38.1% more energy efficient than the vector baseline.

Figure 9: Impact of MANIC's optimizations on full system energy, comparing (from left-to-right): vector baseline, MANIC with forwarding disabled, MANIC without dataflow code scheduling, and full MANIC. Without forwarding, MANIC's added components slightly increase energy by <5%. Forwarding saves 15.5% system energy vs. the baseline, and kill annotations and dataflow code scheduling saves a further 26.0%.

optimal setting of the variable parameter, iterating through each parameter and selecting the best value for each in the next iteration until we reach a fixed point. Our design-space parameters are data and instruction cache sizes, cache line size (4 or 8 bytes), cache associativity (1, 2, 4, 8), vector length, and instruction window size. Unless specified otherwise, our architectural simulations use the parameters shown in Table 1.

Applications: We evaluate MANIC using seven different benchmarks representative of the ultra-low-power computing domain and similar in scope to prior work [6]. We implemented 2D fast-fourier transform (FFT), dense and sparse matrix-matrix multiply (DMM and SMM, respectively), dense and sparse matrix-vector multiply (DMV and SMV, respectively), and dense and sparse 2D convolution (DConv and SConv, respectively). These kernels are important because they are at the heart of many important applications including signal processing, graph analysis, machine learning (including deep neural networks), and communications. Often they account for majority of execution time and application energy – convolution alone in inference applications can account for more than 80% of energy [15]. For each benchmark, we wrote four variants: a version using SONIC [32] which enables safe execution on an intermittent system, a plain C version, a vectorized plain C version, and a vectorized plain C version that we scheduled using MANIC's dataflow code scheduler.

The plain-C, vectorized plain-C and scheduled versions use the same algorithm for each application, except for FFT. For FFT, the plain-C implementation uses the Cooley-Tukey algorithm, while the vectorized versions do a series of smaller FFTs leveraging the vector length and then combine these small FFTs for the output.

6 EVALUATION

We evaluated MANIC to show that it achieves near-ideal energy efficiency without sacrificing generality or programmability. MANIC is more energy efficient, uses less power, and is as fast or faster than a vector or scalar baseline.

6.1 MANIC is energy-efficient

Fig. 8 shows that MANIC is energy efficient, comparing full-system energy for several different architectures across seven workloads. Comparing the first three bars in the figure (scalar baseline, vector baseline, and MANIC) illustrates that MANIC uses less total energy than either baseline. On average, MANIC decreases total energy consumption by 2.8× and by as much as 3.7× over the scalar baseline. These results are likely to significantly *under-estimate* MANIC's improvement over a COTS system deployed today because we model a scalar baseline implemented in 45nm that uses less energy than a common commercial offering implemented in, e.g., 130nm [41]. MANIC reduces energy by 38.1% compared to the vector baseline and consistently decreases energy for all applications, decreasing energy by a minimum of 30.9% (for sparse convolution).

The key to this efficiency is MANIC's vector-dataflow model and kill annotations, which eliminate most eliminate vector register reads and writes. Consequently, vector register file access energy (blue segment) is significantly lower for MANIC than for the vector baseline.

MANIC has near-ideal energy-efficiency: Comparing MANIC to ideal in Fig. 8 shows how MANIC performs compared to an idealized design that removes instruction and data supply energy (i.e. fetch, decode, and RF accesses). In the idealized design, memory's energy cost dominates. MANIC is within 26.4% of the ideal energy consumption. These data show that the cost of programmability and generality in MANIC is low and near the ideal design.

MANIC (without forwarding) has small overhead: Fig. 9 shows the sensitivity of MANIC to various optimizations. The second bars in the figure represent MANIC with forwarding disabled. Compared to the the vector baseline (first bars in the figure), MANIC without

forwarding incurs only a small energy overhead of less than 5% for its microarchitectural additions (e.g., the instruction and forwarding buffers).

Forwarding saves significant energy: The third bars in Fig. 9 show MANIC with dataflow code scheduling disabled, meaning that there are no kill annotations and fewer opportunities to forward values. Even without code scheduling, MANIC decreases energy as compared to the the vector baseline because forwarding effectively eliminates vector register reads. On average, MANIC's forwarding without any special code scheduling reduces total energy by 15.5% over the vector baseline.

Dataflow code scheduling improves forwarding: Finally, comparing these bars to MANIC (last bars in the figure) shows that code scheduling improves MANIC's forwarding to eliminate vector register writes and further reduce reads (i.e., the blue segment is smaller with scheduling). On average, scheduling improves energy by 26.0% vs. without and, on dense and sparse convolution, MANIC is able to entirely eliminate *all* vector register file accesses.

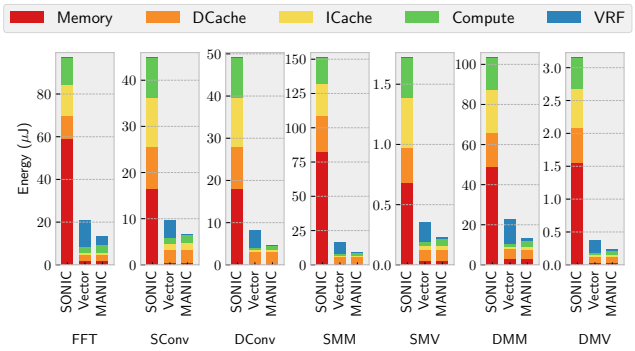


Figure 10: In the intermittent computing domain, MANIC with hardware JIT-checkpointing is 9.6× more energy efficient than SONIC [32], which maintains correctness in software alone.

MANIC with JIT-checkpointing outperforms SONIC: To evaluate how MANIC would perform under intermittent execution on an energy-harvesting system, we compare MANIC to SONIC [32], the state-of-the-art software runtime system for machine inference on energy-harvesting devices. Fig. 10 shows results for SONIC, the scalar baseline running plain C, and MANIC. MANIC significantly outperforms SONIC, by 9.6× on average. This is because MANIC has hardware support for JIT-checkpointing of architectural state before power failure, whereas SONIC suffers from increased accesses to the icache, dcache, and main memory to checkpoint progress in software. Scalar with JIT-checkpointing is 3.5× more efficient than SONIC. Note that SONIC's dcache is write-through because otherwise the dcache and main memory could be inconsistent and flushes after every store would be required for correctness.

6.2 MANIC is performant

Fig. 11 shows both the cycle and instruction counts for each application running on the scalar and vector baselines as well as MANIC. The scalar baseline runs on average 10.6× more instructions and 2.5× more cycles than both the vector baseline and MANIC. MANIC and the vector baseline do not differ in instruction counts and effectively do not differ in cycles counts either (vector dataflow execution

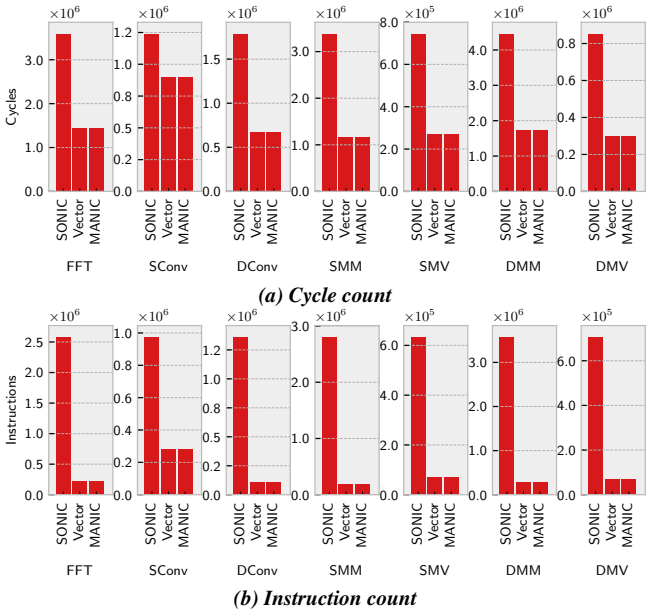


Figure 11: Instruction and cycle counts for seven benchmarks running on the scalar baseline, vector baseline, and MANIC. The vector baseline and MANIC effectively do not differ. Vector execution means that both run 10.6× less instructions and 2.5× less cycles than the scalar baseline.

requires no additional cycles and both implementations run the same program). Vector execution explains the difference in performance between the scalar baseline and the vector implementations. Not only does vector execution amortize instruction fetch and decode, but it also completely eliminates scalar instructions in inner-loop nests. For example, a loop index needs to be incremented significantly less (up to 64× less if the vector length is 64) in vector implementations compared to the scalar-only implementations.

Fewer instructions, less energy: The scalar baseline spends 54.4% of its energy on instruction fetch and register file access (the costs of programmability). Hence, theoretically, a 2.2× reduction in energy

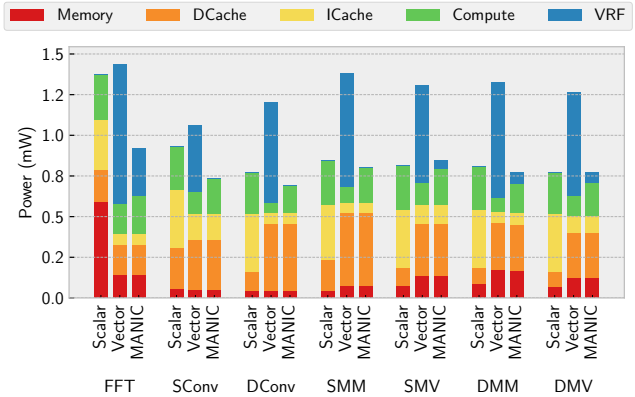


Figure 12: Power of the scalar baseline, vector baseline, and MANIC across seven benchmarks. MANIC uses 10.0% less power than the scalar baseline and, despite using less energy than the scalar baseline, the vector baseline actually uses 29.5% more power.

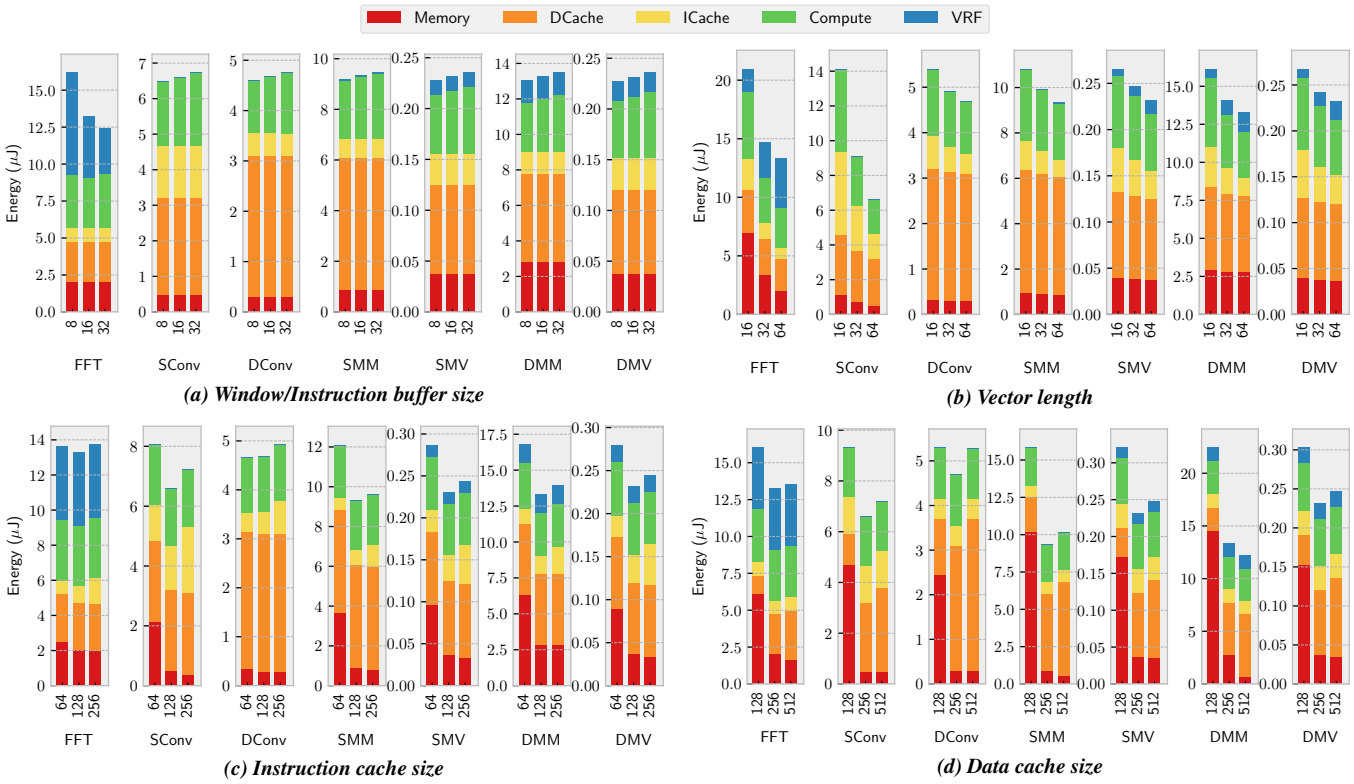


Figure 13: MANIC's sensitivity to its microarchitectural parameters: 16 is the best window size, larger vector lengths are generally better, and moderately sized caches are generally more energy efficient.

can be achieved by eliminating instruction supply and register file energy from the scalar baseline. However, MANIC achieves a decrease of 2.8×. What gives? The explanation is Fig. 11b: MANIC runs fewer instructions than the scalar baseline and effectively eliminates the need for certain scalar instructions in inner-loop nests. A single addition for address arithmetic, for example, can replace dozens of additions in the scalar baseline.

6.3 MANIC is power-efficient

Fig. 12 shows the average power in mW for the scalar and vector baselines as well as MANIC. Each bar is broken down in the same way as in Fig. 9. MANIC uses on average 10.0% less power than the scalar baseline, while the vector baseline actually uses 29.5% more power than the scalar baseline. (Recall that our goal is to save *energy*, not power.) Vector register file accesses (blue segment in Fig. 9) account for the differences in power. The vector baseline must access the VRF for every element of every vector instruction, while MANIC avoids many VRF accesses and thus uses less power.

6.4 Sensitivity

We characterize MANIC by studying the sensitivity of its energy consumption to window size (Fig. 13a), vector length (Fig. 13b), instruction cache size (Fig. 13c), and data cache size (Fig. 13d).

Window size of 16 is best: Fig. 13a suggests that larger window sizes have only a small energy cost and can provide significant improvement to kernels that can take advantage of the extra length. Excluding FFT, a window size of 8 is best, which is consistent with measured kill distances in Fig. 3. For FFT, a window size of 16 or 32 is beneficial. This is because FFT can forward values across loop iterations, allowing a window to span multiple iterations and expose additional dataflow opportunities. This phenomenon is not captured in our static code analysis in Fig. 3, which effectively considers only a single loop iteration.

Large vector lengths reduce icache energy: Fig. 13b shows that energy decreases as vector length increases. Forwarding eliminates 90.1% of total VRF accesses and, thus, despite a larger VRF having a higher access energy, longer vector lengths are still beneficial. The benefit, however, levels off as the proportion of energy spent on instruction supply (yellow segments) decreases as a fraction of the total energy. In our experiments, we stop the vector length at 64 because certain applications (like FFT) are designed to take advantage of vector lengths of 64 or less. This is an artificial stopping point and we plan further work to investigate even larger vector lengths.

A moderately-sized icache is sufficient: Fig. 13c illustrates the need for a relatively small instruction cache. The data show that a very small instruction cache is inefficient, forcing hardware to access main memory often. However, a large cache is unnecessary to capture instruction access locality and realize a large energy efficiency benefit. An instruction cache of 128B is sufficient for most benchmarks and increasing the size further only seems be a detriment due to higher access energy.

Relatively-small data cache is adequate: Fig. 13d shows that generally speaking a cache size of 256B is sufficient. A too small dcache,

leads to additional misses and accesses to main memory. For most benchmarks, except dense matrix multiplication, there is no reduction in energy for the largest cache size of 512B (if anything there is a slight increase due to increased access energy and leakage). Dense matrix multiplication still shows improvement in energy for the largest cache size, but the improvement is small compared to the improvement between 128B and 256B, suggesting that most of the benefit has been captured at 256B.

7 CONCLUSION

This paper described MANIC, an ultra-low-power embedded processor architecture that achieves high energy efficiency without sacrificing programmability or generality. The key to MANIC's efficient operation is its vector-dataflow execution model, in which dependent instructions in a short window forward operands to one another according to dataflow. Vector operation amortizes control overhead. Dataflow execution avoids costly reads from the vector register file. Simple compiler and software support helps avoid further vector register file writes in a microarchitecture-agnostic way. MANIC's microarchitecture implementation directly implements vector-dataflow with simple hardware additions, while still exposing a standard RISC-V ISA interface. MANIC's highly efficient implementation is on average 2.8× more energy efficient than an scalar core and is within 26.4% on average of an ideal design that eliminates all costs of programmability. Our results show that MANIC's vector-dataflow model is realizable and approaches the limit of energy efficiency for an ultra-low-power embedded processor.

REFERENCES

[1] Yunsup Lee Andrew Waterman, Tim Newsome. 2019. RISCV ISA Sim. https://github.com/riscv/riscv-isa-sim

[2] ARM. 2019. ARM NEON. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon

[3] Krste Asanovic, James Beck, Bertrand Irissou, Brian ED Kingsbury, and John Wawrzynek. 1996. T0: A single-chip vector microprocessor with reconfigurable pipelines. In *ESSCIRC'96: Proceedings of the 22nd European Solid-State Circuits Conference*. IEEE, 344–347.

[4] Görkem Aşılıoğlu, Zhaoxiang Jin, Murat Köksal, Omkar Javeri, and Soner Önder. 2015. LaZy superscalar. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 260–271.

[5] James Balfour, William Dally, David Black-Schaffer, Vishal Parikh, and JongSoo Park. 2008. An energy-efficient processor architecture for embedded systems. *IEEE Computer Architecture Letters* 7, 1 (2008), 29–32.

[6] James Balfour, Richard Harting, and William Dally. 2009. Operand registers and explicit operand forwarding. *IEEE Computer Architecture Letters* 8, 2 (2009), 60–63.

[7] James David. Balfour, William J. Dally, Mark Horowitz, and Christoforos Kozyrakis. 2010. *Efficient embedded computing*. Ph.D. Dissertation.

[8] Deniz Balkan, Joseph Sharkey, Dmitry Ponomarev, and Kanad Ghose. 2006. SPARTAN: speculative avoidance of register allocations to transient values for performance and energy efficiency. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 265–274.

[9] D. Balsamo, A. Weddell, A. Das, A. Arreola, D. Brunelli, B. Al-Hashimi, G. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP, 99 (2016), 1–1. https://doi.org/10.1109/TCAD.2016.2547919

[10] D. Balsamo, A.S. Weddell, G.V. Merrett, B.M. Al-Hashimi, D. Brunelli, and L. Benini. 2014. Hibernus: Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems. *Embedded Systems Letters, IEEE* PP, 99 (2014), 1–1. https://doi.org/10.1109/LES.2014.2371494

[11] Lorenzo Bettini, Pilu Crescenzi, Gaia Innocenti, and Michele Loreti. 2004. An environment for self-assessing Java programming skills in first programming courses. In *IEEE International Conference on Advanced Learning Technologies, 2004. Proceedings.* IEEE, 161–165.

[12] Anne Bracy, Prashant Prahlad, and Amir Roth. 2004. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 18–29.

[13] 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 *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*.

[14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[15] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. 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)*.

[16] Tsai-Kan Chien, Lih-Yih Chiou, Shyh-Shyuan Sheu, Jing-Cian Lin, Chang-Chia Lee, Tzu-Kun Ku, Ming-Jinn Tsai, and Chih-I Wu. 2016. Low-power MCU with embedded ReRAM buffers as sensor hub for IoT applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6, 2 (2016), 247–257.

[17] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The reconfigurable streaming vector processor (RSVPTM). In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 141.

[18] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

[19] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *ASPLOS*.

[20] NVIDIA Corporation. 2009. NVIDIA's next generation CUDA compute architecture: Fermi. (2009).

[21] Cray Computer. 2003. U.S. Patent 6,665,774.

[22] David Culler, Jason Hill, Mike Horton, Kris Pister, Robert Szewczyk, and Alec Wood. 2002. Mica: The commercialization of microsensor motes. *Sensor Technology and Design, April* (2002).

[23] Jack B Dennis. 1980. Data flow supercomputers. *Computer* 11 (1980), 48–56.

[24] Jack B Dennis and Guang R Gao. 1988. An efficient pipelined dataflow processor architecture. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 368–373.

[25] Jack B Dennis and David P Misunas. 1975. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, Vol. 3. ACM, 126–132.

[26] Adwait Dongare, Craig Hesling, Khushboo Bhatia, Artur Balanuta, Ricardo Lopes Pereira, Bob Iannucci, and Anthony Rowe. 2017. OpenChirp: A low-power wide-area networking architecture. In *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. IEEE, 569–574.

[27] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*.

[28] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. 2008. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper* 19, 20 (2008).

[29] HSA Foundation. 2018. HSA Platform System Architecture Specification. http://www.hsafoundation.com/standards/

[30] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. 2019. The What's Next Intermittent Computing Architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 211–223.

[31] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 235–246.

[32] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *ASPLOS*.

[33] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51.

[34] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 37–47.

[35] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 5–16. https://doi.org/10.1145/2809695.2809707

[36] Josiah Hester and Jacob Sorber. [n. d.]. Flicker: Rapid Prototyping for the Batteryless Internet of Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*.

[37] Josiah Hester, Kevin Storer, and Jacob Sorber. [n. d.]. Timely Execution on Intermi!ently Powered Ba!eryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*.

[38] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 228–240. https://doi.org/10.1145/3079856.3080238

[39] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, 10–14.

[40] Yiming Huai et al. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin* 18, 6 (2008), 33–40.

[41] Texas Instruments. 2017. MSP430fr5994 SLA. http://www.ti.com/lit/ds/symlink/msp430fr5994.pdf

[42] Texas Instruments. 2018. Low Energy Accelerator FAQ. http://www.ti.com/lit/an/slaa720/slaa720.pdf

[43] ISOCPP. 2019. CPP Spec. https://isocpp.org/std/the-standard

[44] Neal Jackson. 2019. lab11/permamote. https://github.com/lab11/permamote

[45] H. Jayakumar, A. Raha, and V. Raghunathan. 2014. QuickRecall: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*.

[46] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 420–432.

[47] Timothy M Jones, MFR O'Boyle, Jaume Abella, Antonio Gonzalez, and Oguz Ergin. 2005. Compiler directed early register release. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, 110–119.

[48] Ho-Seop Kim and James E Smith. 2002. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 71–81.

[49] Christos Kozyrakis and David Patterson. 2003. Overcoming the limitations of conventional vector processors. *ACM SIGARCH Computer Architecture News* 31, 2 (2003), 399–409.

[50] Gierad Laput, Yang Zhang, and Chris Harrison. 2017. Synthetic sensors: Towards general-purpose sensing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 3986–3999.

[51] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 369–381.

[52] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. 2004. Implementing Software on Resource-constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services (MobiSys '04)*. ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/990064.990095

[53] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 575–585. https://doi.org/10.1145/2737924.2737978

[54] Kaisheng Ma, Xueqing Li, Jinyang Li, Yongpan Liu, Yuan Xie, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. 2017. Incidental computing on IoT nonvolatile processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 204–218.

[55] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 526–537.

[56] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, Vancouver, BC, Canada.

[57] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 129–144. http://dl.acm.org/citation.cfm?id=3291168.3291178

[58] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-In-Time Checkpoints. In *PLDI*.

[59] J. San Miguel, K. Ganesan, M. Badr, and N. E. Jerger. 2018. The EH Model: Analytical Exploration of Energy-Harvesting Architectures. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 76–79. https://doi.org/10.1109/LCA.2017.2777834

[60] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. 2013. Idetic: A High-level Synthesis Approach for Enabling Long Computations on Transiently-powered ASICs. In *IEEE Pervasive Computing and Communication Conference (PerCom)*. http://aceslab.org/sites/default/files/Idetic.pdf

[61] Saman Naderiparizi, Mehrdad Hessar, Vamsi Talla, Shyamnath Gollakota, and Joshua R Smith. 2018. Towards battery-free {HD} video streaming. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 233–247.

[62] Rishiyur S Nikhil et al. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on computers* 39, 3 (1990), 300–318.

[63] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–429.

[64] Nvidia. 2019. Nividia Jetson TX2. https://developer.nvidia.com/embedded/develop/hardware

[65] D. Patterson, T. Anderson, and K. Yelick. 1996. A Case for Intelligent DRAM: IRAM. In *Hot Chips VIII Symposium Record*.

[66] Vlad Petric, Tingting Sha, and Amir Roth. 2005. Reno: a rename-based instruction optimizer. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 98–109.

[67] Matt Poremba, Sparsh Mittal, Dong Li, Jeffrey S Vetter, and Yuan Xie. 2015. Destiny: A tool for modeling emerging 3d nvm and edram caches. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 1543–1546.

[68] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 389–402.

[69] Riscv. 2019. riscv-v-spec. https://github.com/riscv/riscv-v-spec

[70] Anthony Rowe, Mario E Berges, Gaurav Bhatia, Ethan Goldman, Ragunathan Rajkumar, James H Garrett, José MF Moura, and Lucio Soibelman. 2011. Sensor Andrew: Large-scale campus-wide sensing and actuation. *IBM Journal of Research and Development* 55, 1.2 (2011), 6–1.

[71] Emily Ruppel and Brandon Lucia. 2019. Transactional Concurrency Control for Intermittent, Energy Harvesting, Computing Systems. In *PLDI*.

[72] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. 2008. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (Nov. 2008), 2608–2615.

[73] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 422–433.

[74] Peter G Sassone and D Scott Wills. 2004. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 7–17.

[75] Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, André Seznec, and Pierre Michaud. 2015. Long term parking (LTP): criticality-aware resource allocation in OOO processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 334–346.

[76] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*.

[77] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 291.

[78] Andreas Traber. [n. d.]. PULPino: A small single-core RISC-V SoC.

[79] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*. 17.

[80] 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. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 205–218.

[81] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 15.

[82] Dani Voitsechov, Arslan Zulfiqar, Mark Stephenson, Mark Gebhart, and Stephen W. Keckler. 2018. Software-Directed Techniques for Improved GPU Register File Utilization. *ACM Trans. Archit. Code Optim.* 15, 3, Article 38 (Sept. 2018), 23 pages. https://doi.org/10.1145/3243905

[83] John Wawrzynek, Krste Asanovic, Brian Kingsbury, David Johnson, James Beck, and Nelson Morgan. 1996. Spert-II: A vector microprocessor system. *Computer* 29, 3 (1996), 79–86.

[84] Mark Wyse. [n. d.]. Understanding GPGPU Vector Register File Usage. ([n. d.]).

[85] Yuan Xie. 2013. *Emerging Memory Technologies: Design, Architecture, and Applications*. Springer Science & Business Media.

[86] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 41–53.

[87] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. 2011. Moo: A batteryless computational RFID and sensing platform. *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep* (2011).