The Vector-Thread Architecture

Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, Krste Asanovic

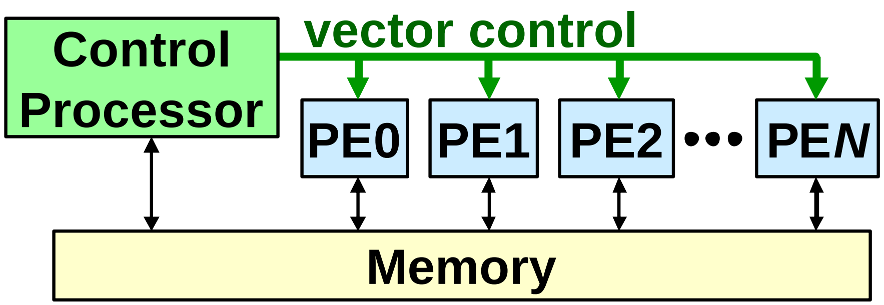
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA

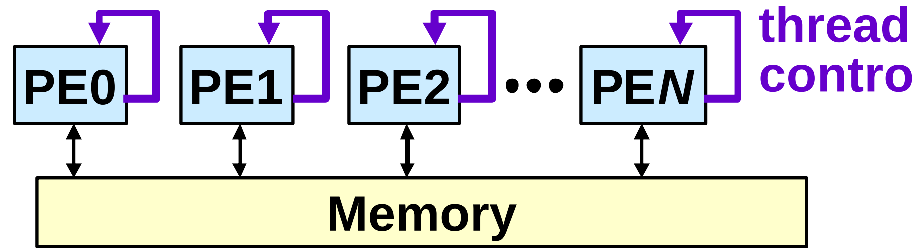
ISCA 2004

Goals For Vector-Thread Architecture

- Primary goal is **efficiency**
  - High performance with low energy and small area
- Take advantage of whatever **parallelism** and **locality** is available: DLP, TLP, ILP
  - Allow intermixing of multiple levels of parallelism
- Programming model is key
  - Encode parallelism and locality in a way that enables a complexity-effective implementation
  - Provide clean abstractions to simplify coding and compilation

Vector and Multithreaded Architectures



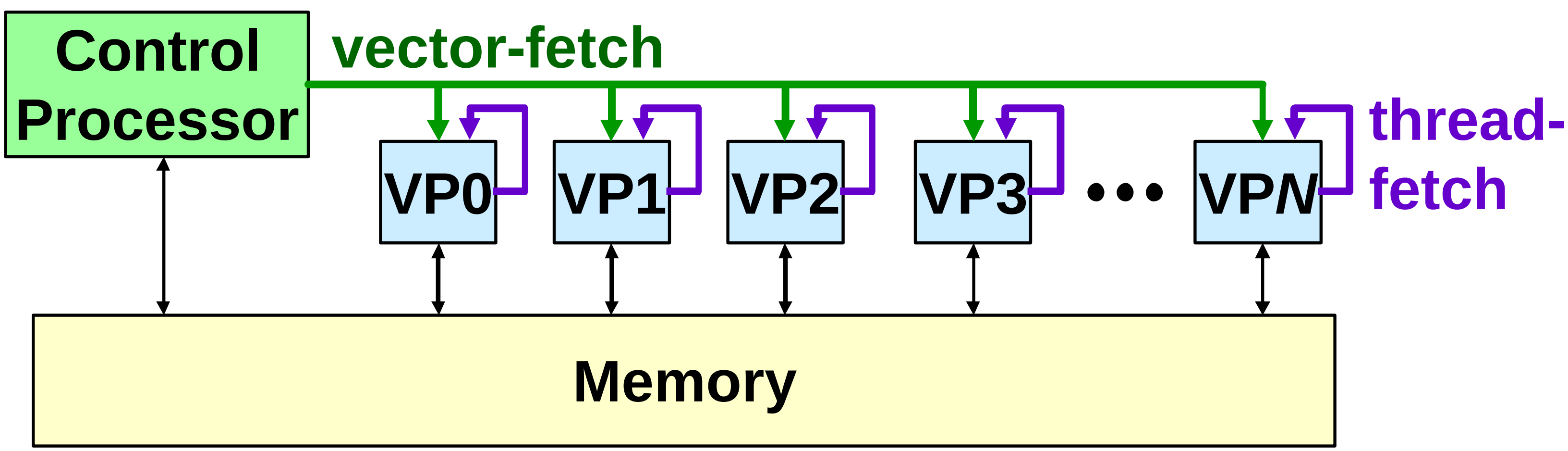


- Vector processors provide efficient DLP execution
  - Amortize instruction control
  - Amortize loop bookkeeping overhead
  - Exploit structured memory accesses
- Unable to execute loops with loop-carried dependencies or complex internal control flow

- Multithreaded processors can flexibly exploit TLP
- Unable to amortize common control overhead across threads
- Unable to exploit structured memory accesses across threads
- Costly memory-based synchronization and communication between threads

Vector-Thread Architecture

- VT unifies the vector and multithreaded compute models
- A control processor interacts with a vector of virtual processors (VPs)
- **Vector-fetch**: control processor fetches instructions for all VPs in parallel
- **Thread-fetch**: a VP fetches its own instructions
- VT allows a seamless intermixing of vector and thread control

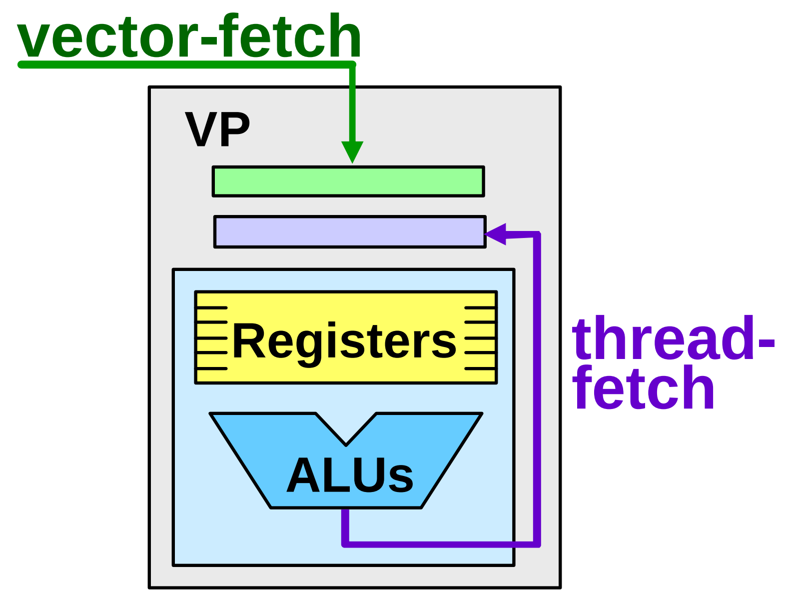


Outline

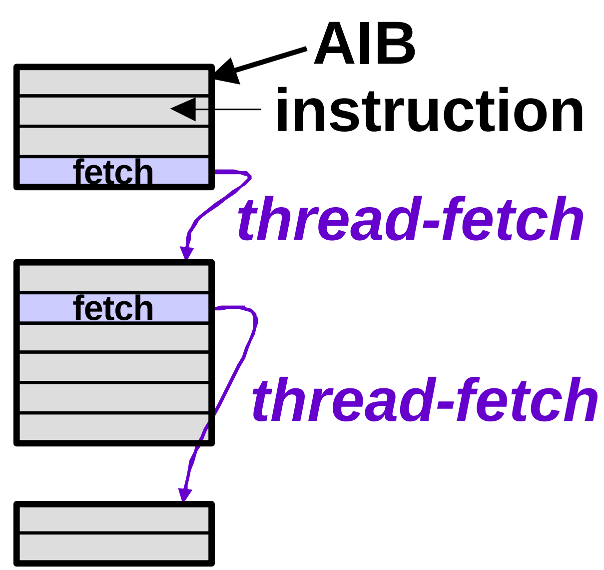
- Vector-Thread Architectural Paradigm
  - Abstract model
  - Physical Model
- SCALE VT Processor
- Evaluation
- Related Work

Virtual Processor Abstraction

- VPs contain a set of registers
- VPs execute RISC-like instructions grouped into atomic instruction blocks (AIBs)
- VPs have no automatic program counter, AIBs must be explicitly fetched
  - VPs contain pending vector-fetch and thread-fetch addresses
- A fetch instruction allows a VP to fetch its own AIB
  - May be predicated for conditional branch
- If an AIB does not execute a fetch, the VP thread *stops*

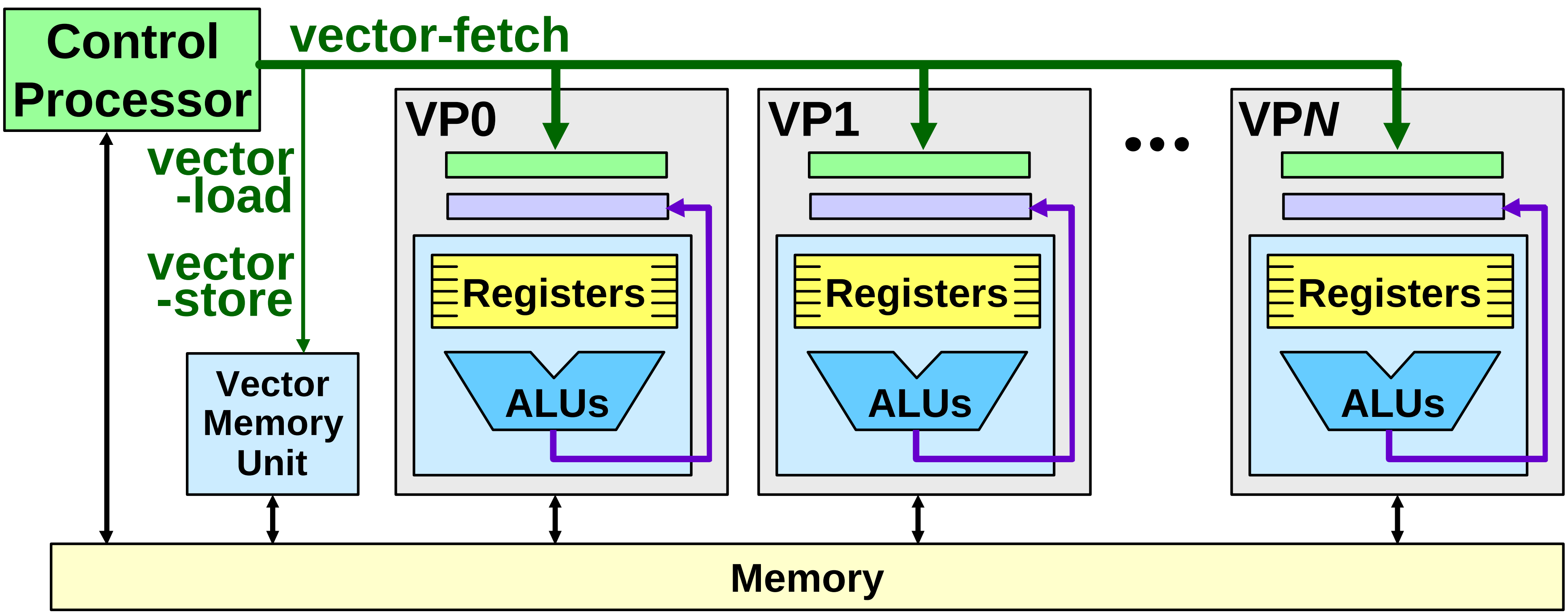


VP thread execution



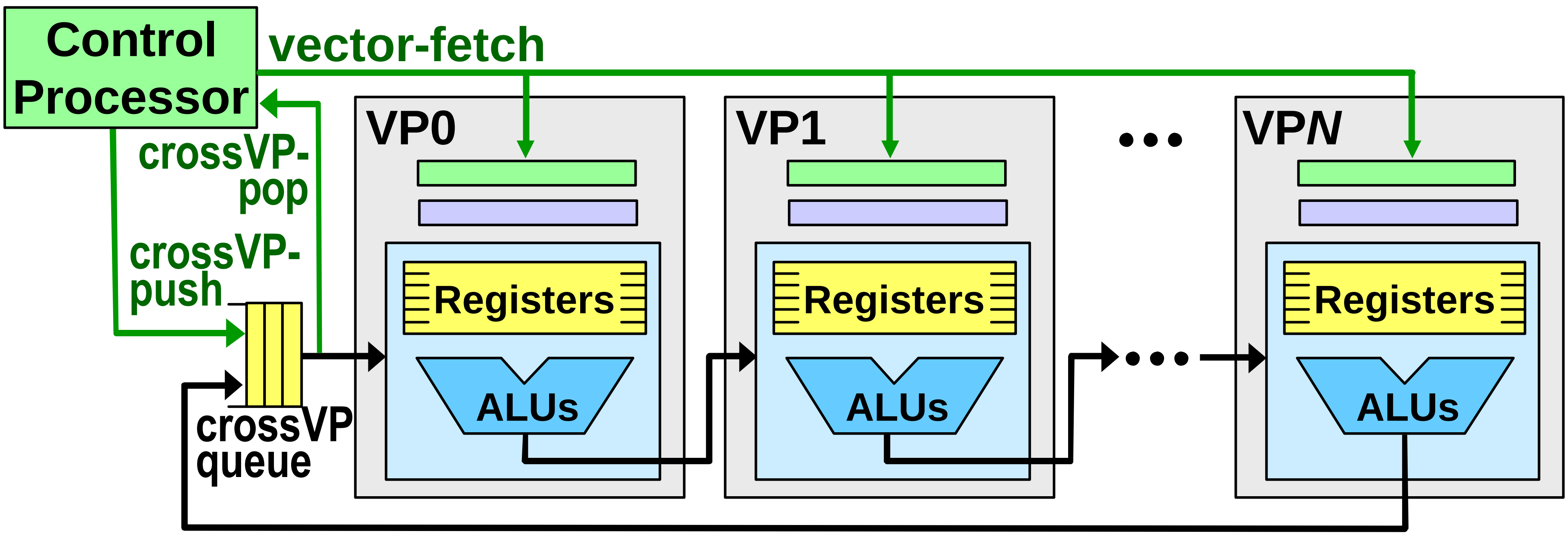
Virtual Processor Vector

- A VT architecture includes a control processor and a virtual processor vector
  - Two interacting instruction sets
- A vector-fetch command allows the control processor to fetch an AIB for all the VPs in parallel
- Vector-load and vector-store commands transfer blocks of data between memory and the VP registers



Cross-VP Data Transfers

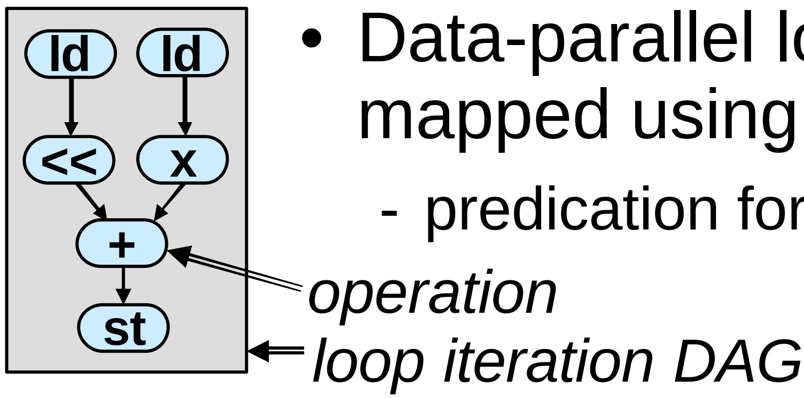
- Cross-VP connections provide fine-grain data operand communication and synchronization
  - VP instructions may target nextVP as destination or use prevVP as a source
  - CrossVP queue holds wrap-around data, control processor can push and pop
  - Restricted ring communication pattern is cheap to implement, scalable, and matches the software usage model for VPs



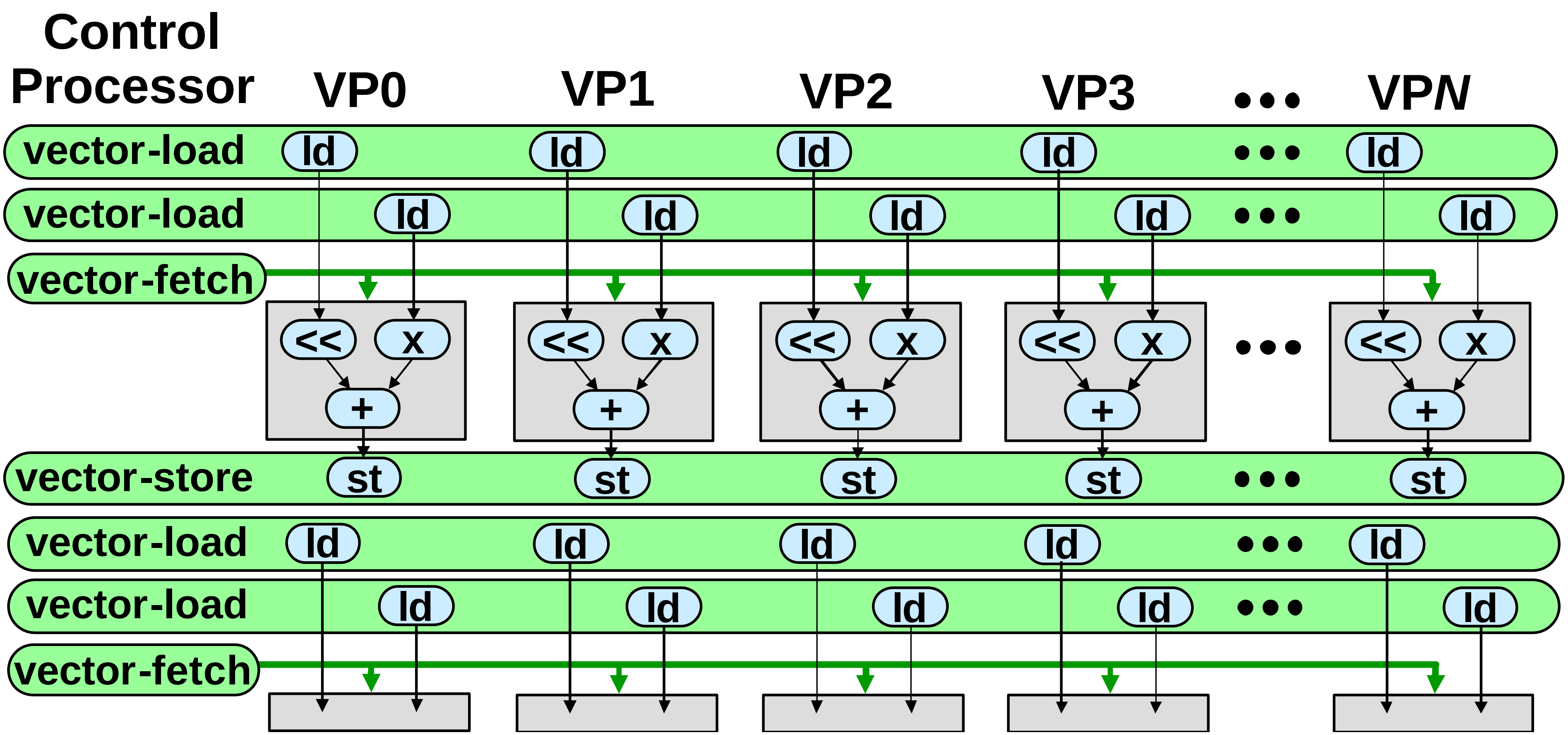
Mapping Loops to VT

- A broad class of loops map naturally to VT
  - Vectorizable loops
  - Loops with loop-carried dependencies
  - Loops with internal control flow
- Each VP executes one loop iteration
  - Control processor manages the execution
  - Stripmining enables implementation-dependent vector lengths
- Programmer or compiler only schedules one loop iteration on one VP
  - No cross-iteration scheduling

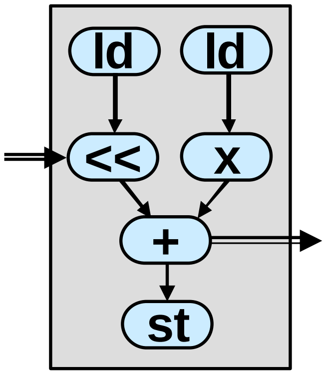
Vectorizable Loops



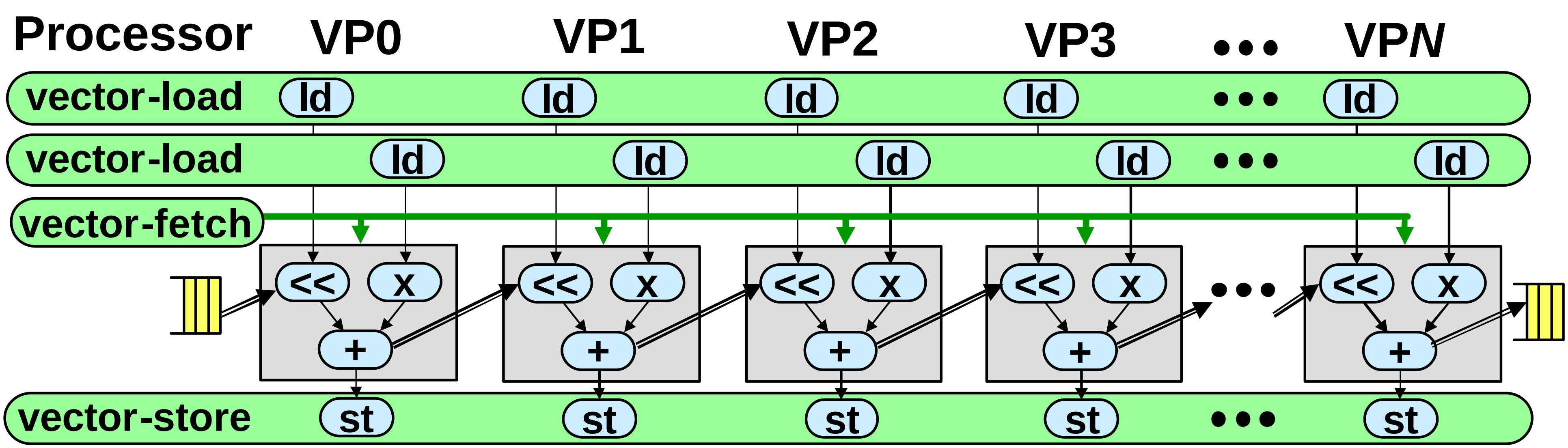
- Data-parallel loops with no internal control flow mapped using vector commands
  - predication for small conditionals



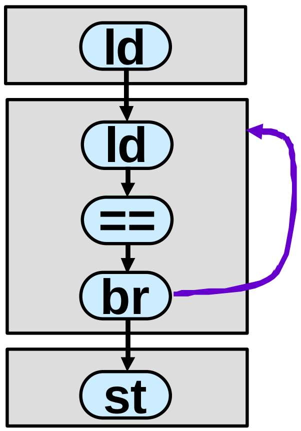
Loop-Carried Dependencies



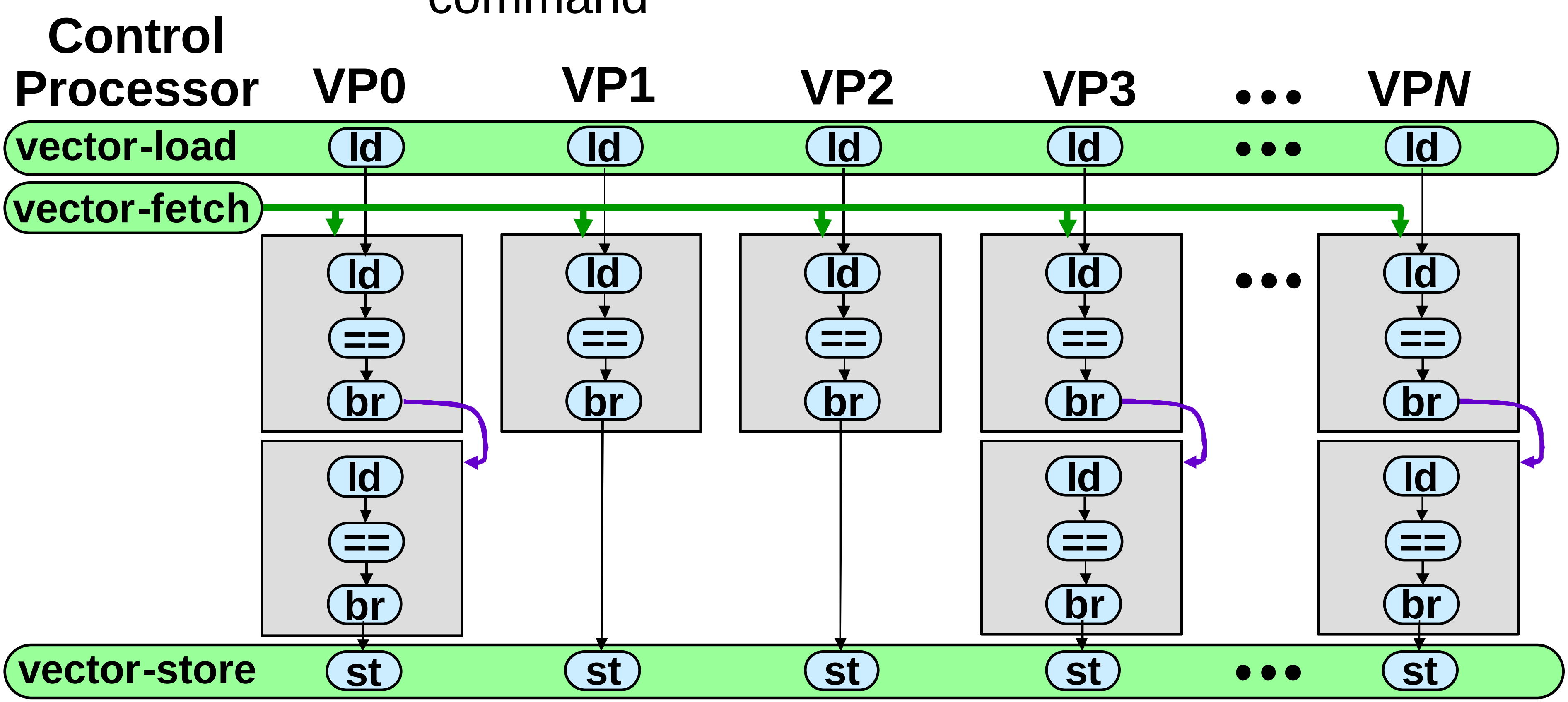
- Loops with cross-iteration dependencies mapped using vector commands with cross-VP data transfers
  - Vector-fetch introduces chain of prevVP receives and nextVP sends
  - Vector-memory commands with non-vectorizable compute



Loops with Internal Control Flow

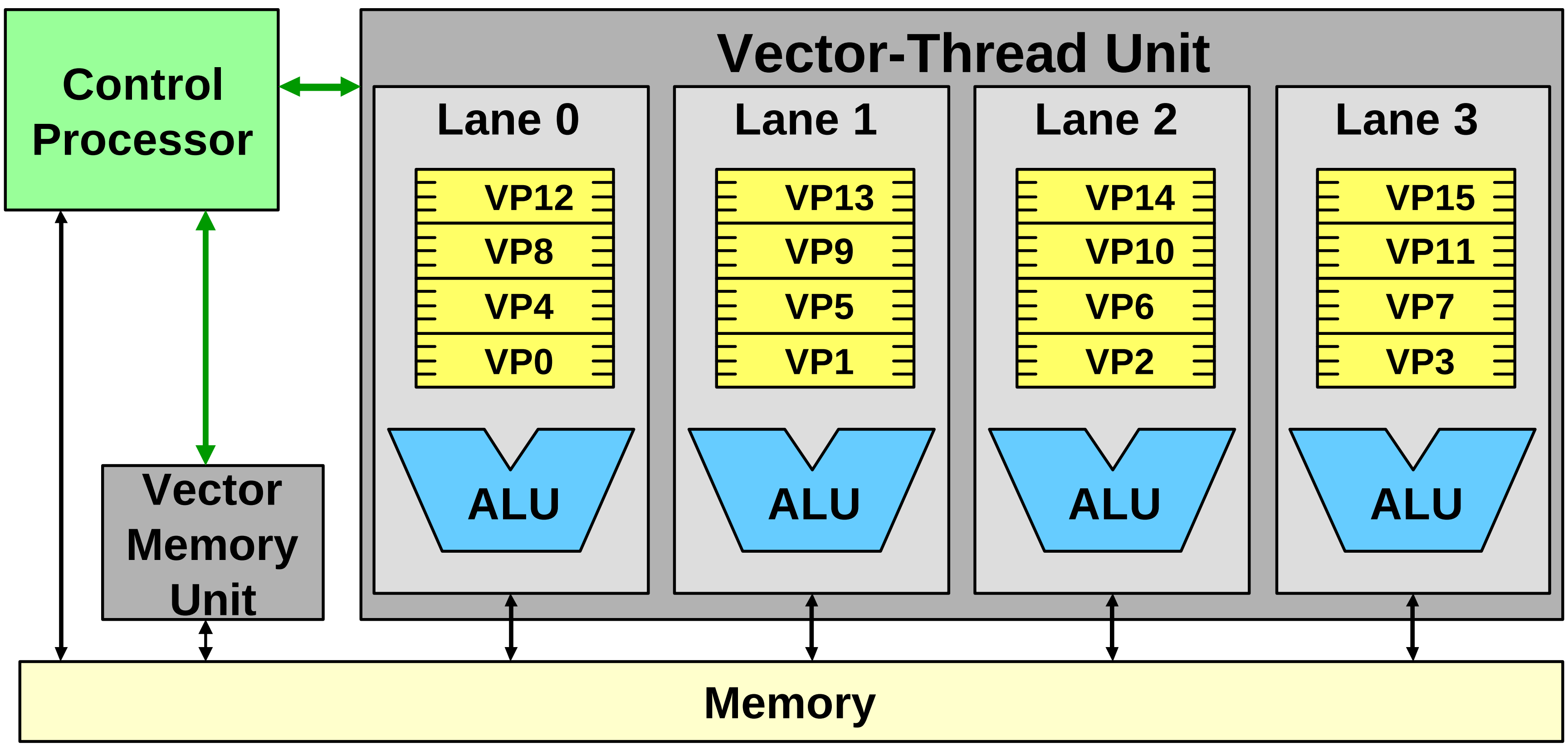


- Data-parallel loops with large conditionals or inner-loops mapped using thread-fetches
  - Vector-commands and thread-fetches freely intermixed
  - Once launched, the VP threads execute to completion before the next control processor command



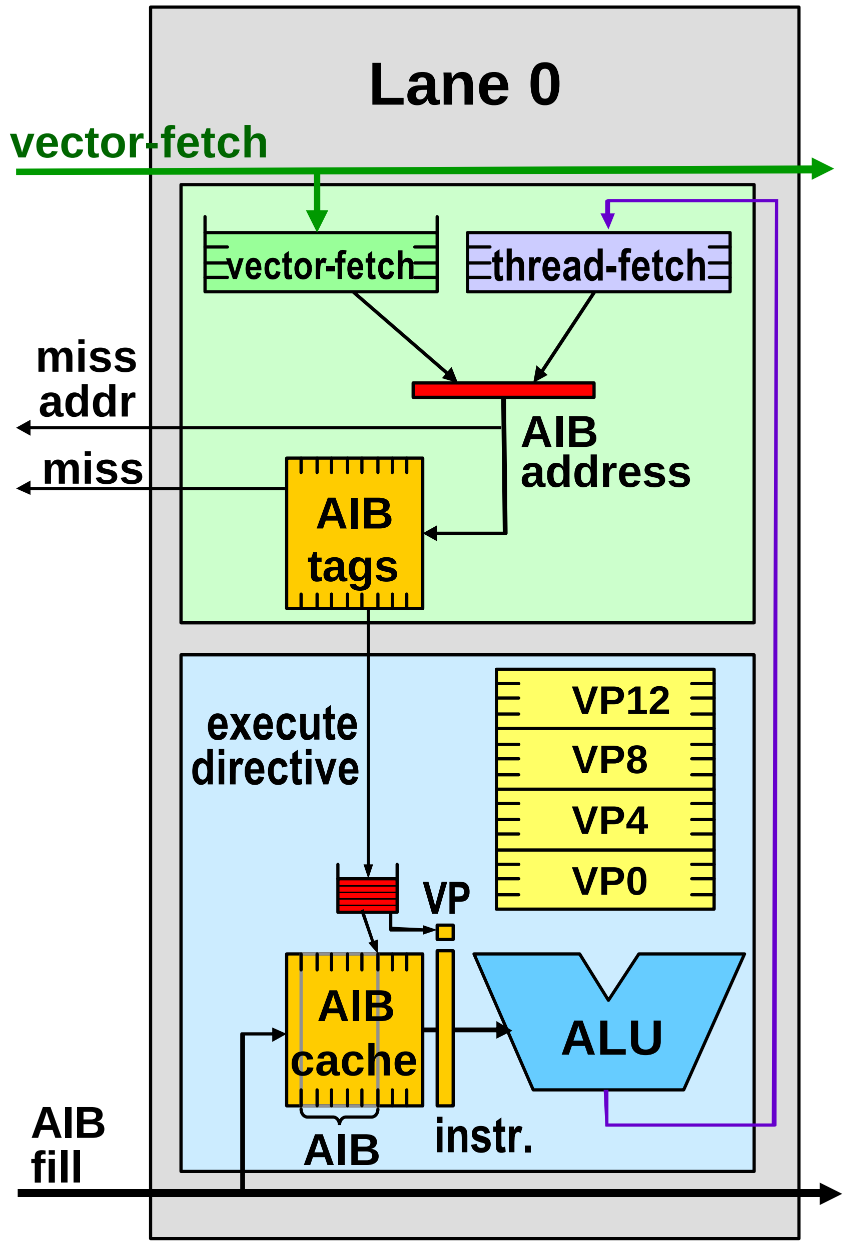
VT Physical Model

- A Vector-Thread Unit contains an array of lanes with physical register files and execution units
- VPs map to lanes and share physical resources, VP execution is time-multiplexed on the lanes
- Independent parallel lanes exploit parallelism across VPs and data operand locality within VPs



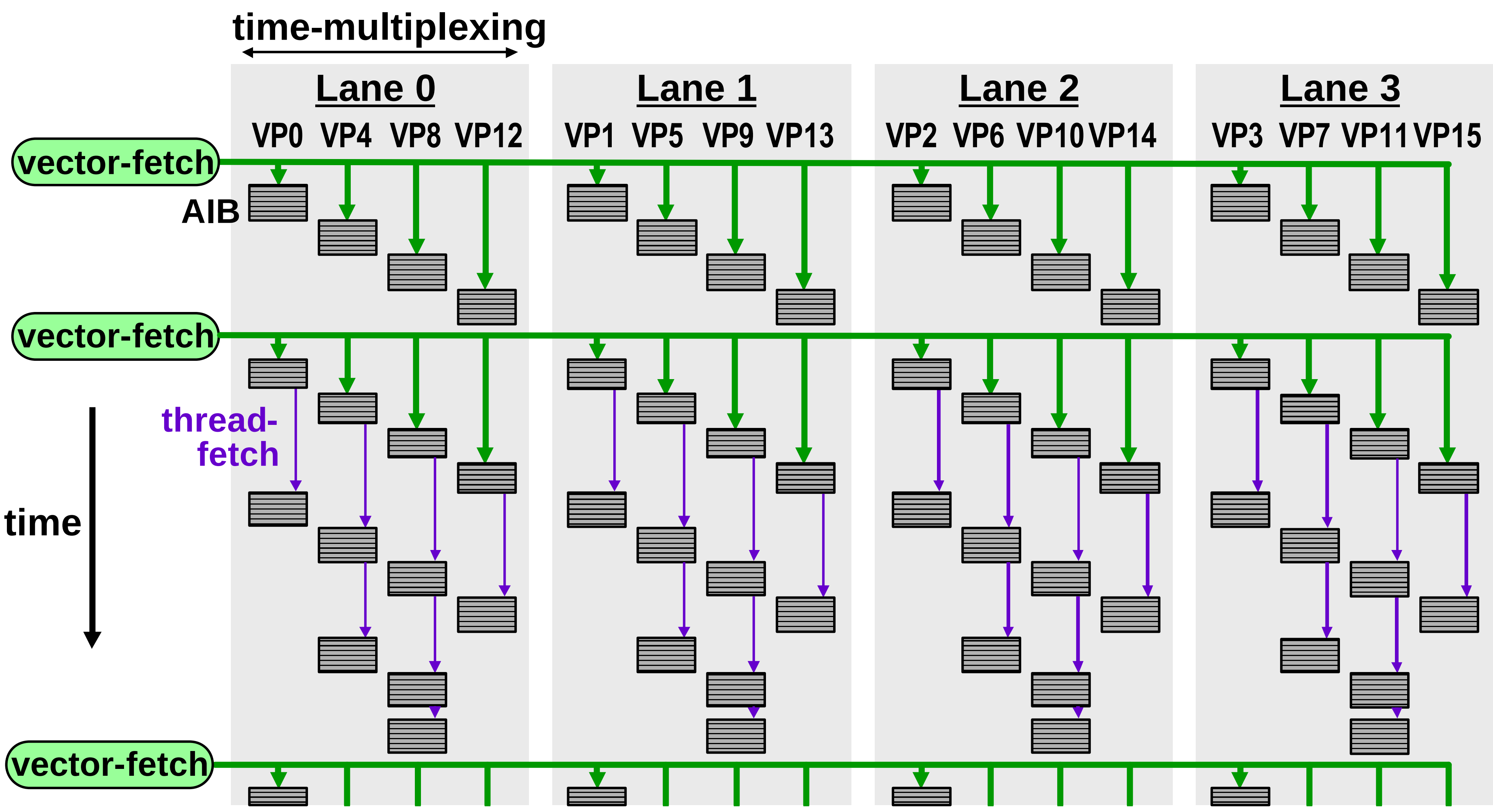
Lane Execution

- Lanes execute decoupled from each other
- Command management unit handles vector-fetch and thread-fetch commands
- Execution cluster executes instructions in-order from small AIB cache (e.g. 32 instructions)
  - AIB caches exploit locality to reduce instruction fetch energy (on par with register read)
- Execute directives point to AIBs and indicate which VP(s) the AIB should be executed for
  - For a thread-fetch command, the lane executes the AIB for the requesting VP
  - For a vector-fetch command, the lane executes the AIB for every VP
- AIBs and vector-fetch commands reduce control overhead
  - 10s—100s of instructions executed per fetch address tag-check, even for non-vectorizable loops



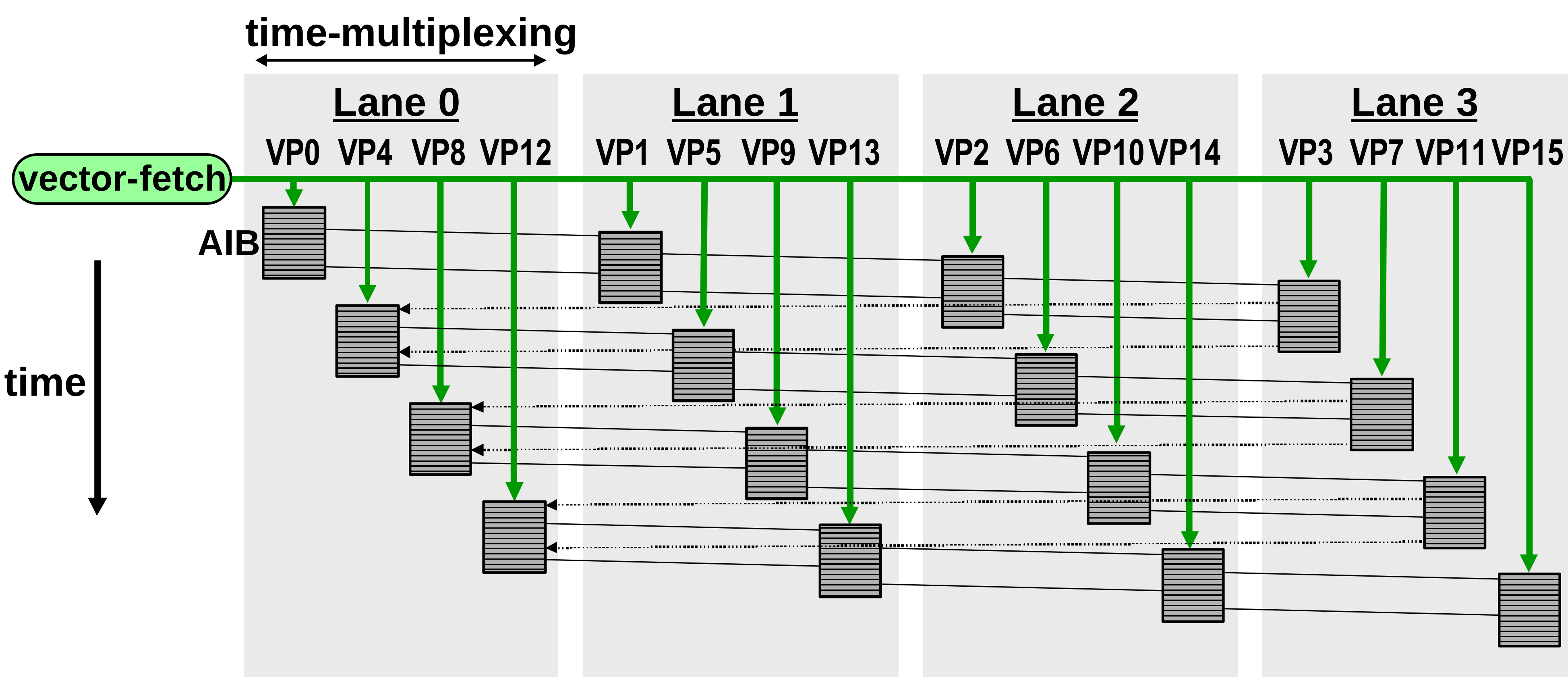
VP Execution Interleaving

- Hardware provides the benefits of loop unrolling by interleaving VPs
- Time-multiplexing can hide thread-fetch, memory, and functional unit latencies



VP Execution Interleaving

- Dynamic scheduling of cross-VP data transfers automatically adapts to software critical path (in contrast to static software pipelining)
  - No static cross-iteration scheduling
  - Tolerant to variable dynamic latencies

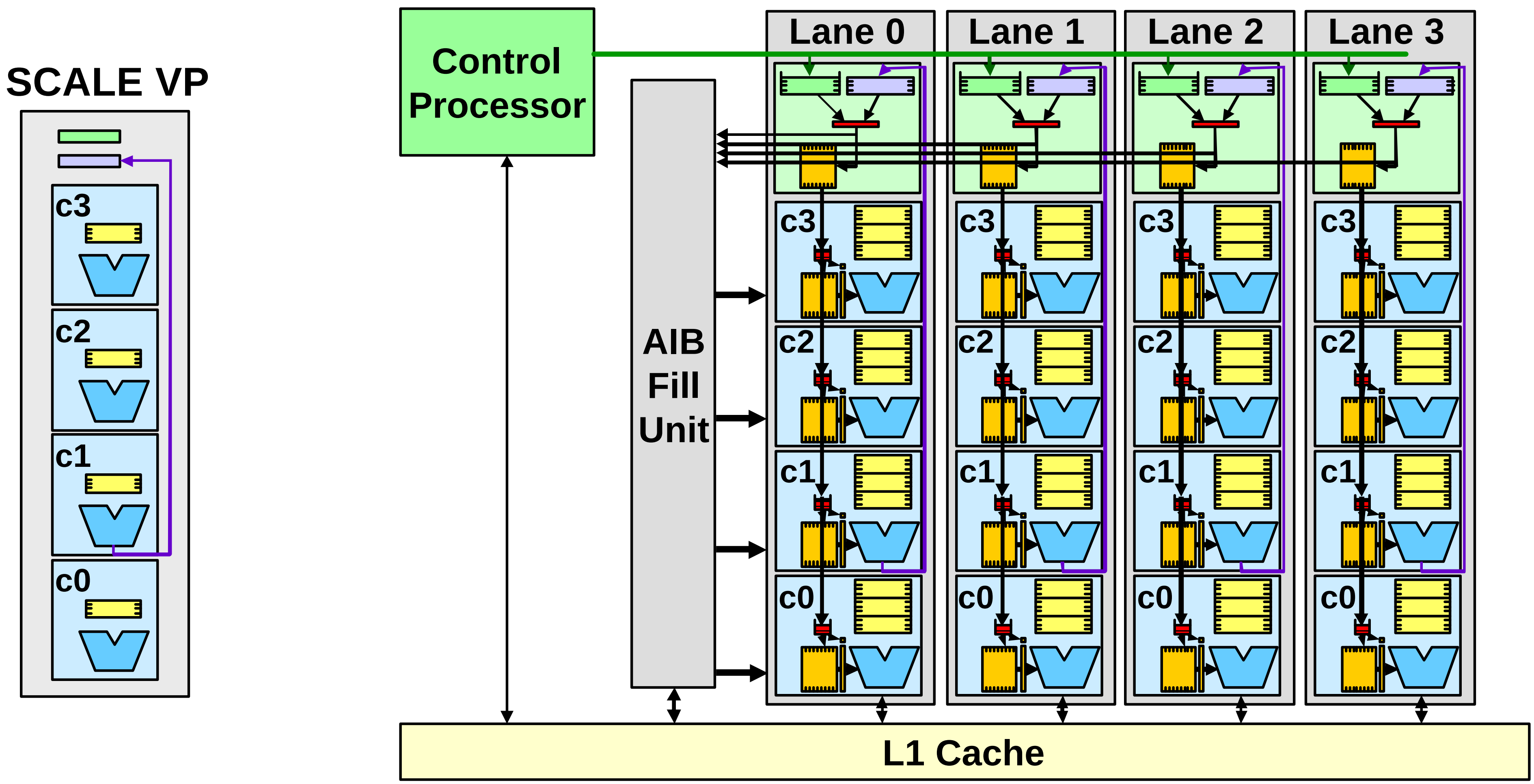


SCALE Vector-Thread Processor

- SCALE is designed to be a complexity-effective all-purpose embedded processor
  - Exploit all available forms of parallelism and locality to achieve high performance and low energy
- Constrained to small area (estimated 10 mm^2 in 0.18 µm)
  - Reduce wire delay and complexity
  - Support tiling of multiple SCALE processors for increased throughput
- Careful balance between software and hardware for code mapping and scheduling
  - Optimize runtime energy, area efficiency, and performance while maintaining a clean scalable programming model

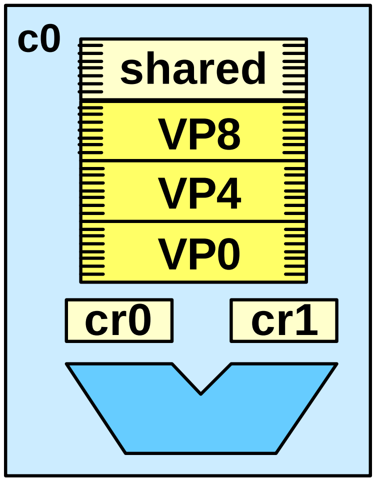
SCALE Clusters

- VPs partitioned into four clusters to exploit ILP and allow lane implementations to optimize area, energy, and circuit delay
  - Clusters are heterogeneous – c0 can execute loads and stores, c1 can execute fetches, c3 has integer mult/div
  - Clusters execute decoupled from each other

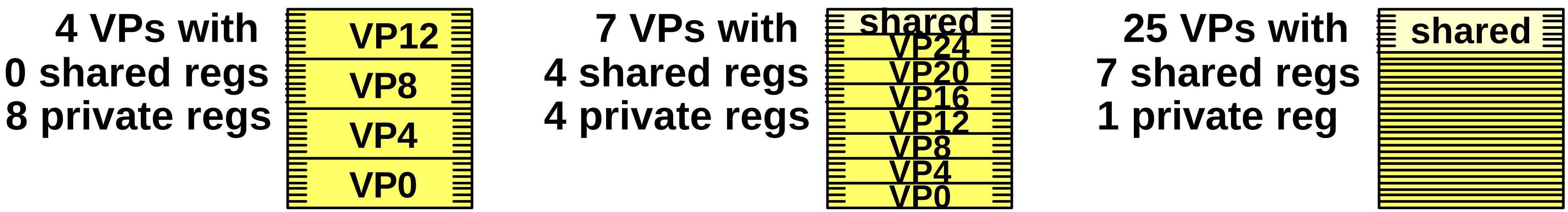


SCALE Registers and VP Configuration

- Atomic instruction blocks allow VPs to share temporary state – only valid within the AIB
  - VP general registers divided into private and shared
  - Chain registers at ALU inputs – avoid reading and writing general register file to save energy



- Number of VP registers in each cluster is configurable
  - The hardware can support more VPs when they each have fewer private registers
  - Low overhead: Control processor instruction configures VPs before entering stripmine loop, VP state undefined across reconfigurations



SCALE Micro-Ops

- Assembler translates portable software ISA into hardware micro-ops
- Per-cluster micro-op bundles access local registers only
- Inter-cluster data transfers broken into transports and writebacks

Software VP code:

cluster	***operation***		***destinations***
`c0`	`xor pr0, pr1`	®	`c1/cr0, c2/cr0`
`c1`	`sll cr0, 2`	®	`c2/cr1`
`c2`	`add cr0, cr1`	®	`pr0`

Hardware micro-ops:

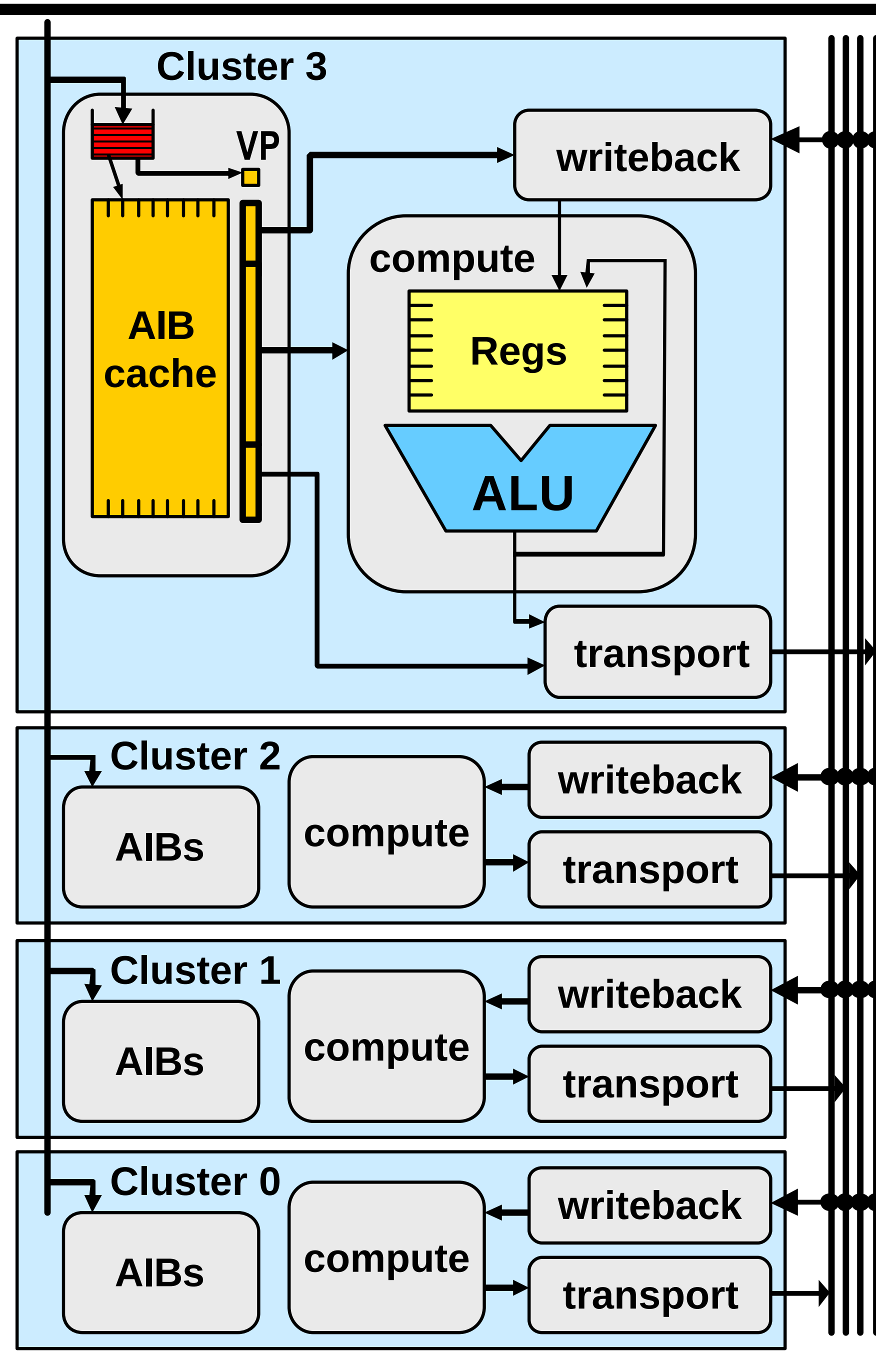
Cluster 0			**Cluster 1**			**Cluster 2**		
wb	**compute**	**tp**	**wb**	**compute**	**tp**	**wb**	**compute**	**tp**
	`xor pr0,pr1`	`®c1,c2`	`c0®cr0`	`sll cr0,2`	`®c2`	`c0®cr0`		
						`c1®cr1`	`add cr0,cr1®pr0`	

cluster micro-op bundle (spanning Cluster 0's wb, compute and tp columns)

Cluster 3 not shown

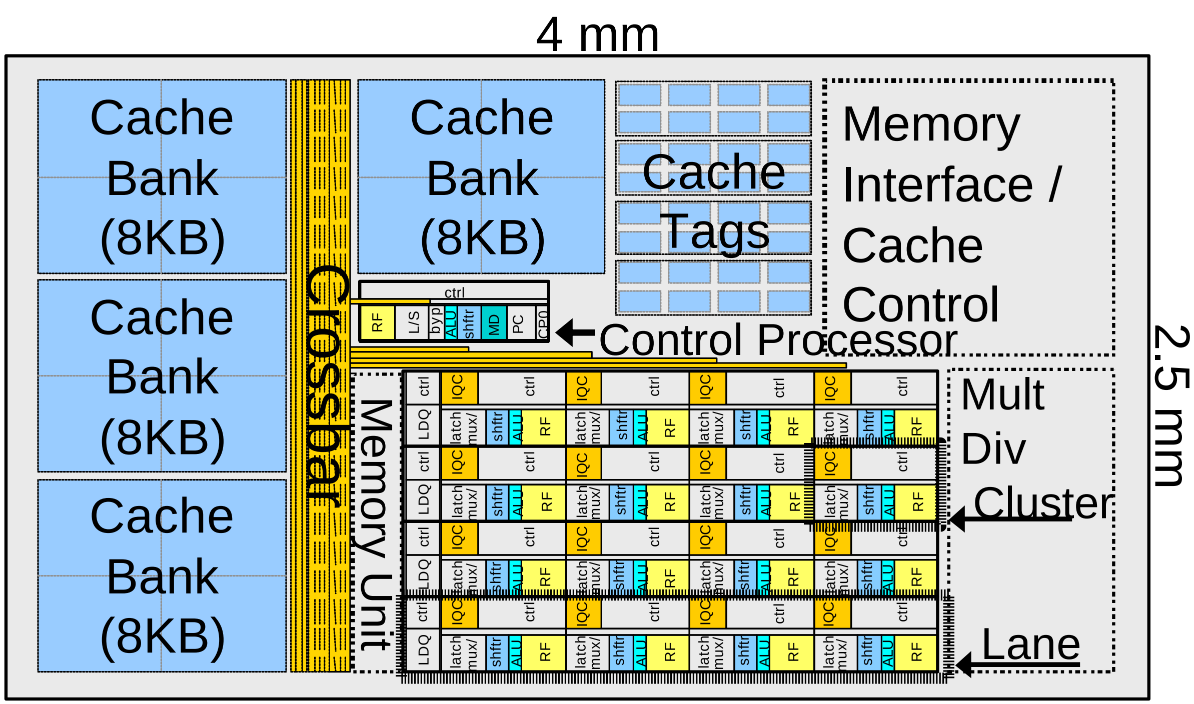
SCALE Cluster Decoupling

- Cluster execution is decoupled
  - Cluster AIB caches hold micro-op bundles
  - Each cluster has its own execute-directive queue, and local control
  - Inter-cluster data transfers synchronize with handshake signals
- Memory Access Decoupling *(see paper)*
  - Load-data-queue enables continued execution after a cache miss
  - Decoupled-store-queue enables loads to slip ahead of stores



SCALE Prototype and Simulator

- Prototype SCALE processor in development
  - Control processor: MIPS, 1 instr/cycle
  - VTU: 4 lanes, 4 clusters/lane, 32 registers/cluster, 128 VPs max
  - Primary I/D cache: 32 KB, 4x128b per cycle, non-blocking
  - DRAM: 64b, 200 MHz DDR2 (64b at 400Mb/s: 3.2GB/s)
  - Estimated 10 mm^2 in 0.18µm, 400 MHz (25 FO4)
- Cycle-level execution-driven microarchitectural simulator
  - Detailed VTU and memory system model

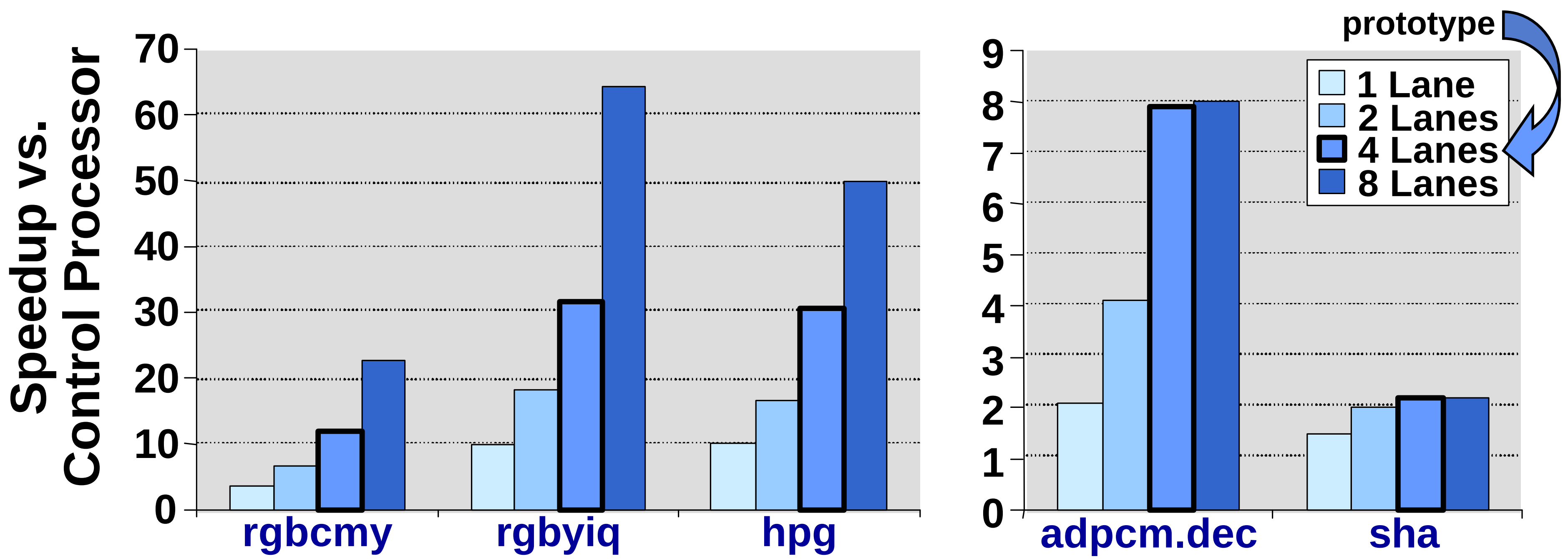


Benchmarks

- Diverse set of 22 benchmarks chosen to evaluate a range of applications with different types of parallelism
  - 16 from EEMBC, 6 from MiBench, Mediabench, and SpecInt
- Hand-written VP assembly code linked with C code compiled for control processor using gcc
  - Reflects typical usage model for embedded processors
- EEMBC enables comparison to other processors running hand-optimized code, but it is not an ideal comparison
  - Performance depends greatly on programmer effort, algorithmic changes are allowed for some benchmarks, these are often unpublished
  - Performance depends greatly on special compute instructions or sub-word SIMD operations (for the current evaluation, SCALE does not provide these)
  - Processors use unequal silicon area, process technologies, and circuit styles
- Overall results: SCALE is competitive with larger and more complex processors on a wide range of codes from different domains
  - See paper for detailed results
  - Results are a snapshot, SCALE microarchitecture and benchmark mappings continue to improve

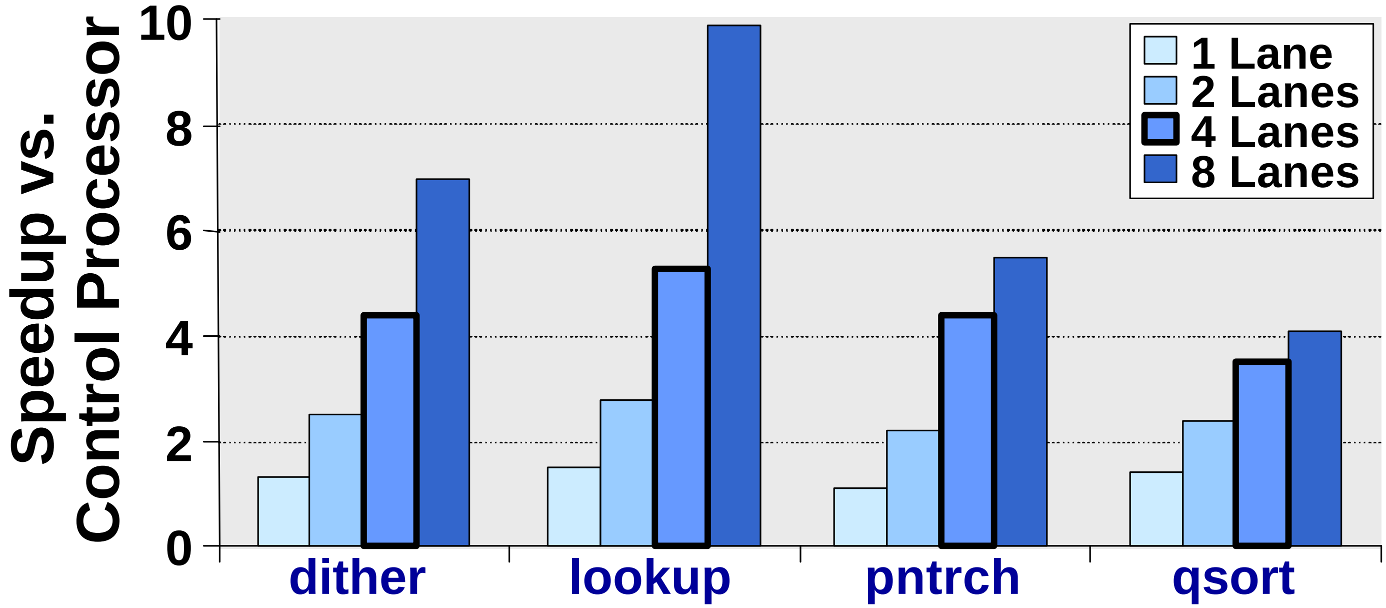
Mapping Parallelism to SCALE

- Data-parallel loops with no complex control flow
  - Use vector-fetch and vector-memory commands
  - EEMBC **rgbcmy**, **rgbyiq**, and **hpg** execute 6-10 ops/cycle for 12x-32x speedup over control processor, performance scales with number of lanes
- Loops with loop-carried dependencies
  - Use vector-fetched AIBs with cross-VP data transfers
  - Mediabench **adpcm.dec**: two loop-carried dependencies propagate in parallel, on average 7 loop iterations execute in parallel, 8x speedup
  - MiBench **sha** has 5 loop-carried dependencies, exploits ILP



Mapping Parallelism to SCALE

- Data-parallel loops with large conditionals
  - Use vector-fetched AIBs with conditional thread-fetches
  - EEMBC **dither**: special-case for white pixels (18 ops vs. 49)
- Data-parallel loops with inner loops
  - Use vector-fetched AIBs with thread-fetches for inner loop
  - EEMBC **lookup**: VPs execute pointer-chaising IP address lookups in routing table
- Free-running threads
  - No control processor interaction
  - VP worker threads get tasks from shared queue using atomic memory ops
  - EEMBC **pntrch** and MiBench **qsort** achieve significant speedups



Comparison to Related Work

- **TRIPS** and **Smart Memories** can also exploit multiple types of parallelism, but must explicitly switch modes
- **Raw**'s tiled processors provide much lower compute density than SCALE's clusters which factor out instruction and data overheads and use direct communication instead of programmed switches
- **Multiscalar** passes loop-carried register dependencies around a ring network, but it focuses on speculative execution and memory, whereas VT uses simple logic to support common loop types
- **Imagine** organizes computation into kernels to improve register file locality, but it exposes machine details with a low-level VLIW ISA, in contrast to VT's VP abstraction and AIBs
- **CODE** uses register-renaming to hide its decoupled clusters from software, whereas SCALE simplifies hardware by exposing clusters and statically partitioning inter-cluster transport and writeback ops
- Jesshope's **micro-threading** is similar in spirit to VT, but its threads are dynamically scheduled and cross-iteration synchronization uses full/empty bits on global registers

Summary

- The vector-thread architectural paradigm unifies the vector and multithreaded compute models
- VT abstraction introduces a small set of primitives to allow software to succinctly encode parallelism and locality and seamlessly intermingle DLP, TLP, and ILP
  - Virtual processors, AIBs, vector-fetch and vector-memory commands, thread-fetches, cross-VP data transfers
- SCALE VT processor efficiently achieves high-performance on a wide range of embedded applications