Decoupled Vector Runahead

And other kinds of runahead

Ajeya Naithani, Jaime Roelandts, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout

MICRO 2023

Presented by Matt Ngaw and Yufei Shi

Ajeya Naithani

PhD + Postdoc @ Ghent University

now Professor @ Eindhoven University of Technology

Jaime Roelandts

PhD @ Ghent University

Sam Ainsworth

Honorary Fellow, Assistant Professor @ University of Edinburgh

Timothy M. Jones

Professor @ University of Cambridge

Lieven Eeckhout

Full Professor @ Ghent University

Review: Runahead Execution HPCA 2003

- ★ ROB (instruction window) commits in-order
  - ○ Problem: *Can be blocked by a long-latency memory access*

- ★ Traditionally, pipeline stalls until memory access resolves
  - ○ Alternative: *Speculatively execute past the blocking instruction to prefetch*

Pros	Cons
Accurate prefetching; follows program path	Executes *everything*; not all instructions are useful
Low hardware overhead; reuses much of existing hardware	Performance overhead of flushing+refilling pipeline

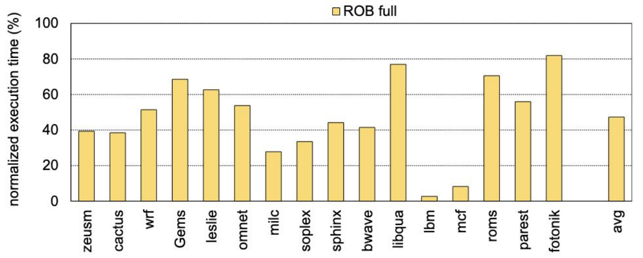


Fig. 1: Fraction of the execution time the ROB is full for memory-intensive benchmarks. *An out-of-order processor stalls on a full ROB for about half the time.*

Precise Runahead HPCA 2020

★ Runahead executes everything
- Solution: *Find and execute only instructions needed to generate memory accesses*

★ Runahead incurs overhead from flushing/refilling
- Solution 1: *Do not release processor state (AKA preserve the IQ, PRF, and ROB)*
- Solution 2: *Reuse availability in IQ and PRF to execute in runahead mode*

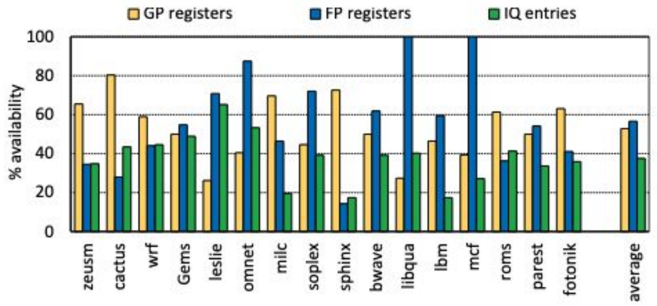


Fig. 4: Percentage general-purpose (GP) registers, floating-point (FP) registers and issue queue (IQ) entries that are available upon a full-window stall due to a long-latency load blocking commit. *About half the issue queue and physical register file entries are available upon a full-window stall.*

Vector Runahead MICRO 2021, Micro 2021 Top Pick

- ★ What about dependence chains?
  - for (i = 0..100) { x += B[hash(A[i])]; }

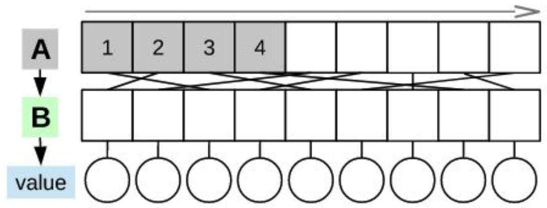
- ★ Why wait for the front-end to deliver future instructions?
  - Insight: *Generate your own memory instructions!*
  - Akin to hardware prefetching techniques
  - If right, automatically exposes lots of MLP

- ★ Same instruction, different data
  - Insight: *Vectorize these auto-generated instructions*
  - Performance of compute within loops can be improved via SIMD

Vector Runahead MICRO 2021, Micro 2021 Top Pick

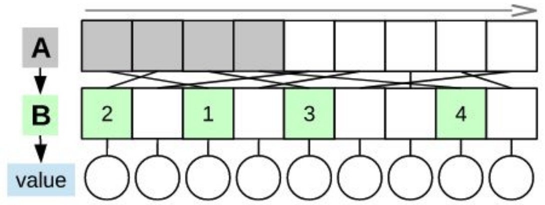
```
for (int x=0; x<N; x++)
    y += B[hash(A[x])]->value;
```

Precise Runahead



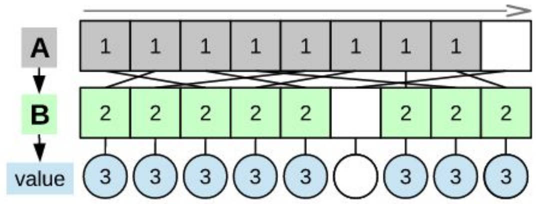
Data from A prefetched during runahead
B cannot be prefetched since depends on A

Precise Runahead + HW Prefetch



Data from A known from prefetcher
Data from B prefetched during runahead
Values cannot be prefetched since depends on B

Vector Runahead



Data from A fetched + stalled for during VR
Data from B fetched + stalled for during VR, after A
Data from value fetched + stalled for during VR, after A + B

Decoupled Vector Runahead

MICRO 2023 Best Paper
Micro 2023 Top Pick

2x speedup*

- ★ ROBs have grown bigger over the years
  - ○ Condition for runahead mode: *still full ROB?*
- ★ VR delays termination until done vectorizing dependent chains
  - ○ *What if stalling load finishes before then?*
- ★ Not all workloads have predictable loops
  - ○ *Data-dependent, dynamic number of iterations?*
- ★ Not all workloads have loops with many iterations
  - ○ *Multiple instances of inner-loop?*
- ★ Control flow complicates vectorization of loop iterations
  - ○ *Different memory accesses patterns?*

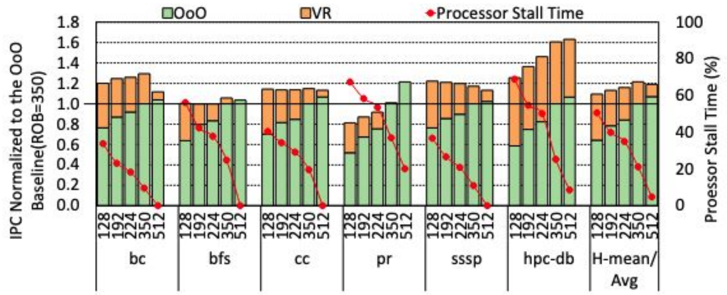
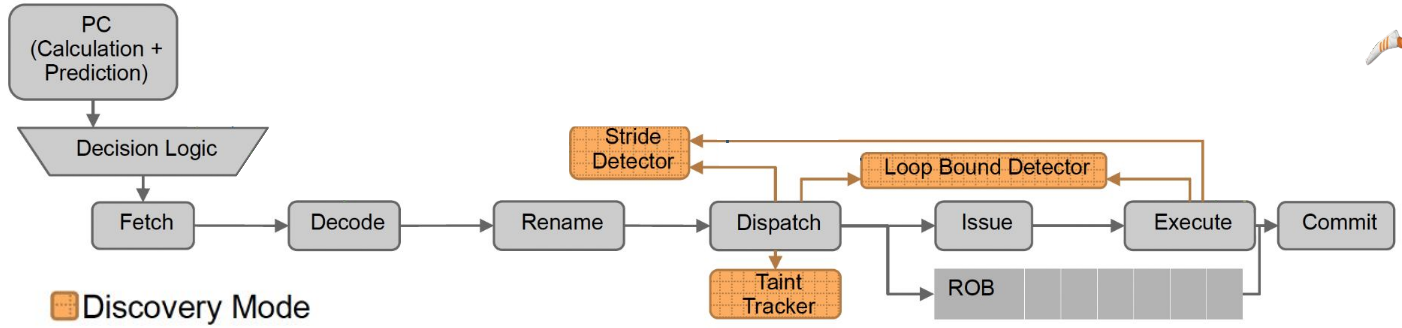


Figure 2: Performance of an OoO core and VR, normalized to a baseline 350-entry ROB OoO core (left axis), and processor stall time due to a full ROB (right axis), as a function of ROB size. *The performance gain of VR diminishes with increasing ROB size, and for some benchmarks overall performance even decreases.*

DVR::Discovery Mode



★ Stride Detector identifies a striding load and its stride

★ Look for the most suitable candidate for DVR

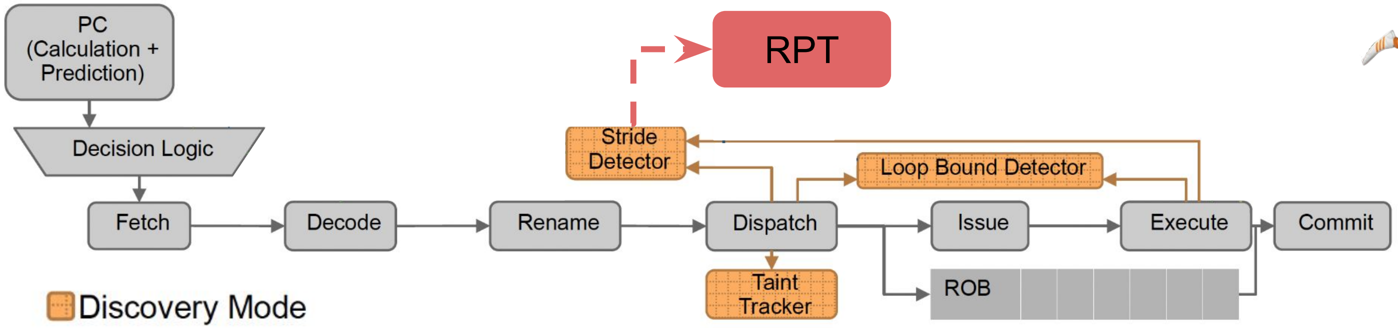
★ Derive loop bounds

★ Discover dependent loads

★ Stride Detector reaches the same striding load again

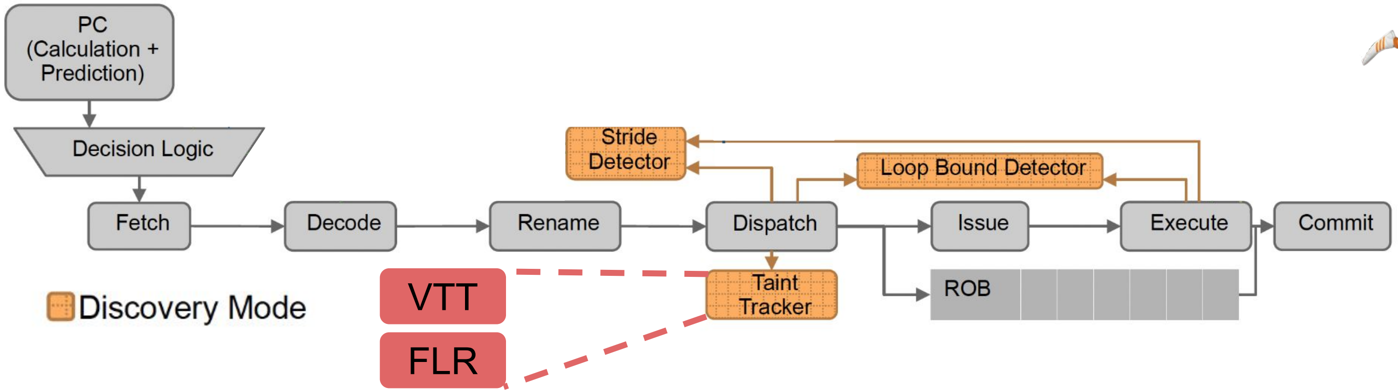
Discovery Mode

DVR::Innermost Striding Load Detection



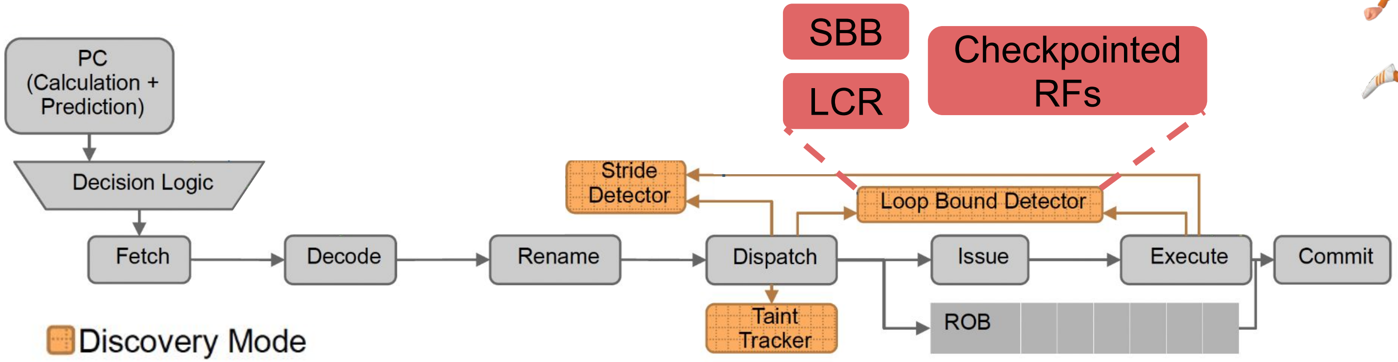
★ Uses the Reference Prediction Table (RPT) to detect striding load.
★ Prefers *innermost striding load* to be the trigger for entering Discovery Mode.

DVR::Dependent-Load Checking



- ★ Vector Taint Tracker (VTT) identifies instructions that will later be vectorized.
- ★ Final-Load Register (FLR) is updated with the PC of a load if its input is tainted.
- ★ All tainted instructions in the dependence chain, **starting from the striding load that triggers the DVR to the last dependent load in FLR.**

DVR::Loop Bound Inference



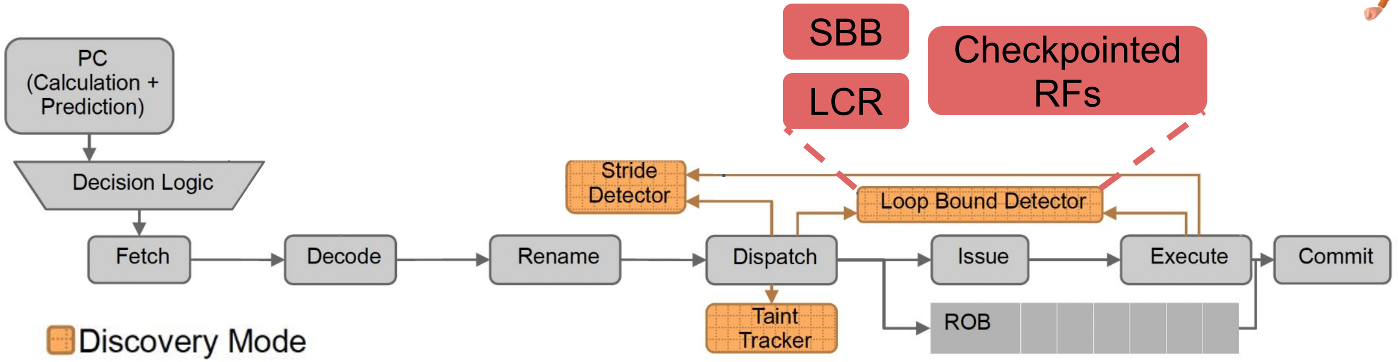
- ★ First branch with a backward edge -> a loop.
- ★ Seen-Branch Bit (SBB) gets set when a backward branch is encountered.
- ★ As long as SBB is not set:
  - ○ Last-Compare Register (LCR) gets updated with the Src/Dst register IDs when encountering a compare instruction as long as SBB is not set.
  - ○ If LCR's tracked Dst register matches a branch's Src register AND the branch-taken destination PC < striding load PC, set SBB.

DVR::Loop Bound Inference

```
1. ADDI r1, 1, r1      ; r1 = r1 + 1
2. LOAD r1, r2         ; r2 = Mem[r1]  ← striding load
3. CMPI r1, r10, r3    ; r3 = (r1 < r10) ? 1 : 0, assume r10 = 10
4. CMP r1, r2, r4      ; r4 = (r1 < r2) ? 1 : 0
5. BNEZ r3, #8         ; Branch to line 7 if r3 is not zero
6. BNEZ r5, #?         ; Branch to somewhere if r5 is not zero
7. BNEZ r3, #1         ; Branch to line 1 if r3 is not zero
```

- ★ First branch with a backward edge -> a loop.
- ★ Seen-Branch Bit (SBB) gets set when a backward branch is encountered.
- ★ As long as SBB is not set:
  - ○ Last-Compare Register (LCR) tracks the Src/Dst register IDs when encountering a compare instruction as long as SBB is not set.
  - ○ If LCR's tracked Dst register matches a branch's Src register AND the branch-taken destination PC < striding load PC, set SBB.

DVR::Loop Bound Inference

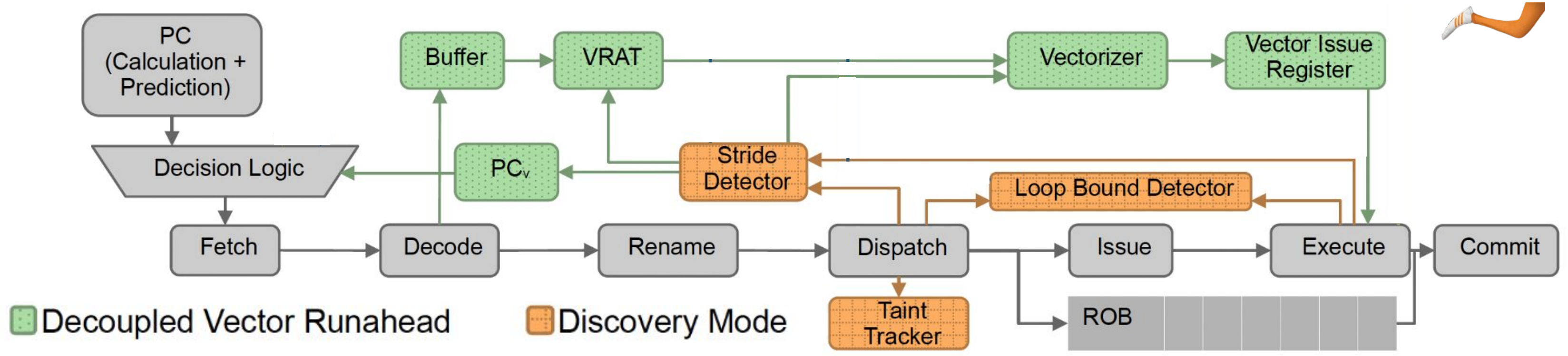


★ Two checkpoints of the architectural register file are taken when entering and exiting Discovery Mode.

★ LCR contains the register mappings of the inputs to the compare instruction used in the loop guard.

★ # loop iterations left can be derived from the difference between the register values.

DVR::Vector Runahead Subthread

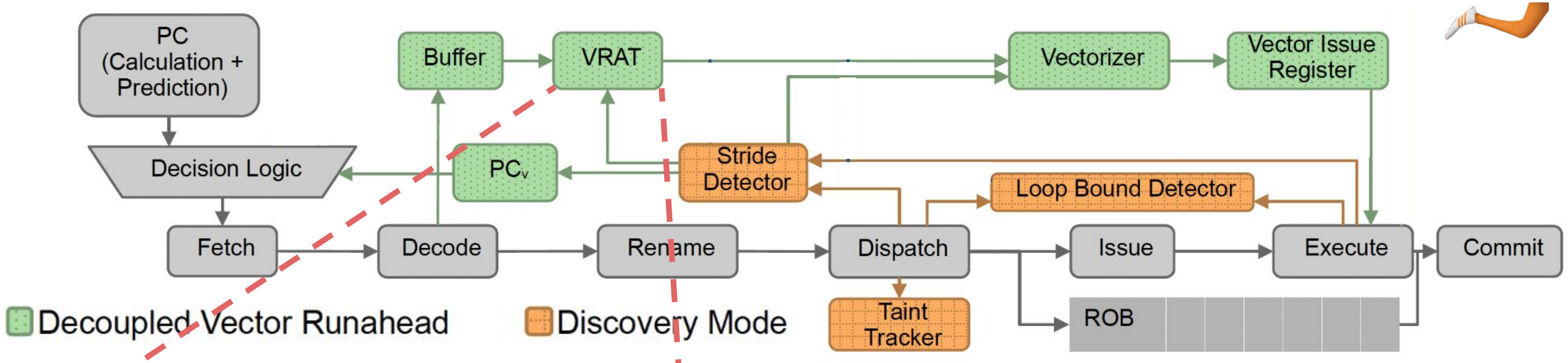


★ Discover Mode identifies: a striding load, its stride, its dependence chain, and remaining iterations of the inner loop.

★ DVR spawns a vector-runahead subthread once the main thread reaches the striding load in the second time.

★ Subthread instructions are generated from the front-end micro-op buffer, which decouples the fetch stage from the rest of the pipeline.

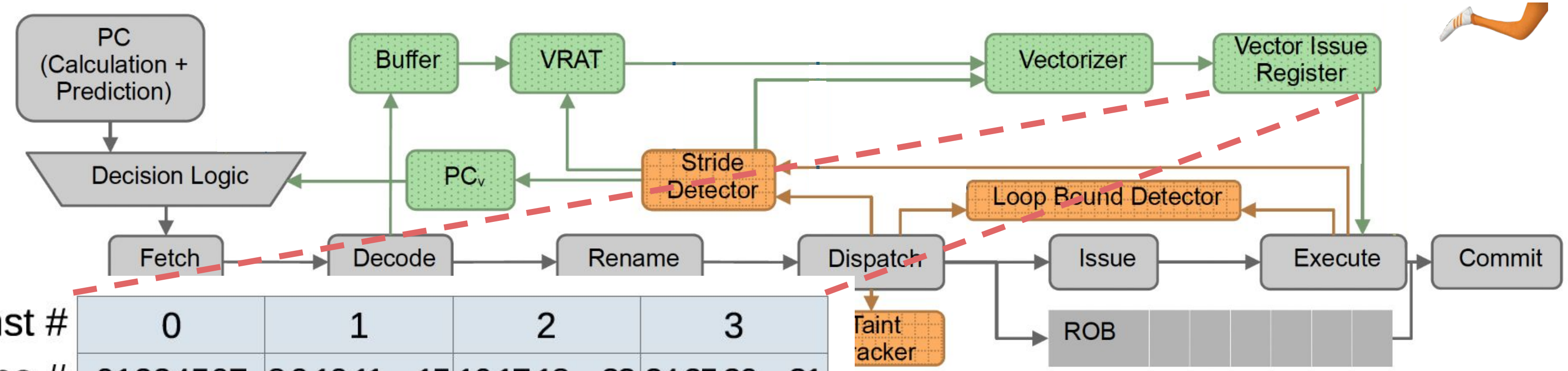
DVR::Vector Register Allocation Table

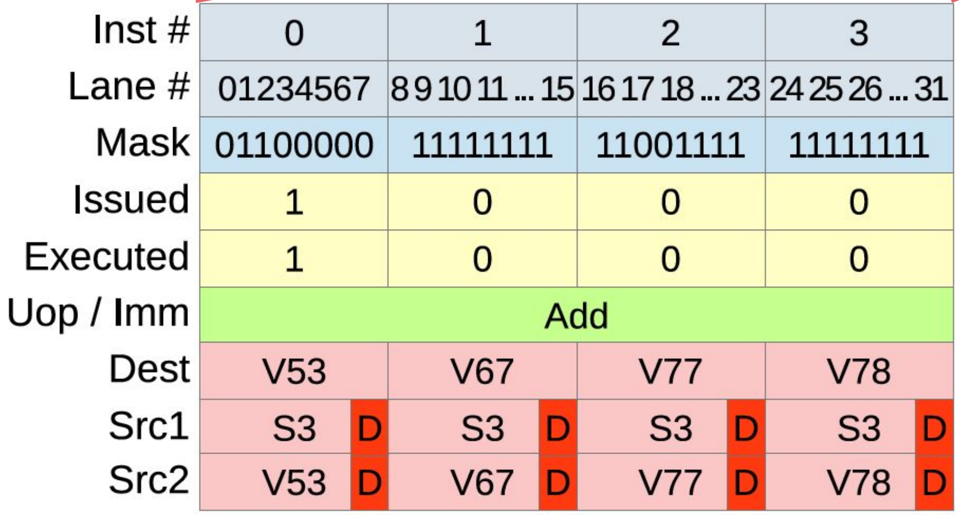


Vreg	Preg(s)							
R1	S45	S45	S45	S45	S45	S45	S45	S45
R2	V34	V35	V36	V37	V38	V39	V68	V69

Figure 4: An example VRAT allocation considering 8 physical registers (one per vector lane) for brevity rather than 16 as in our setup. *Architectural register R1 points to the same scalar physical register (S45) for all lanes. Architectural register R2 has been vectorized to 8 different vector physical registers, because either one of its sources was tainted, or control-flow divergence occurred.*

DVR::Vector Issue Register

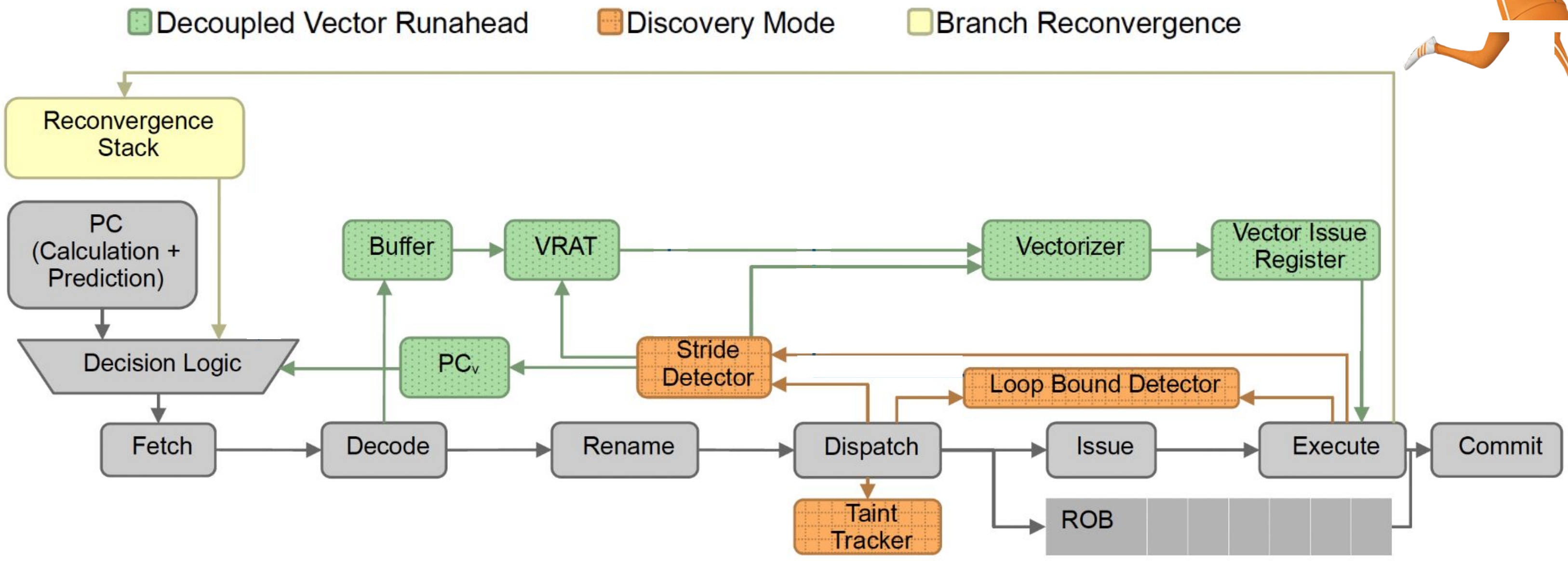




Inst #	0	1	2	3
Lane #	01234567	8 9 10 11 ... 15	16 17 18 ... 23	24 25 26 ... 31
Mask	01100000	11111111	11001111	11111111
Issued	1	0	0	0
Executed	1	0	0	0
Uop / Imm	Add			
Dest	V53	V67	V77	V78
Src1	S3 D	S3 D	S3 D	S3 D
Src2	V53 D	V67 D	V77 D	V78 D

Figure 5: The Vector Issue Register showing 4 AVX-512 vector instructions (instead of 16 as in our setup for brevity). *Fine-grained masking has turned some scalar-equivalent lanes in AVX-512 instructions 0 and 2 into no-ops. The first AVX-512 instruction has been issued and executed, and the last three have neither been issued nor executed. Source register src1 is scalar register S3 and is shared among all lanes (none of which have diverged), which may be for example the base address of an array, whereas source register src2 has been vectorized (for example the index into the array). The destination registers are also vectorized, to the same location as src2 as they were the same architectural scalar register.*

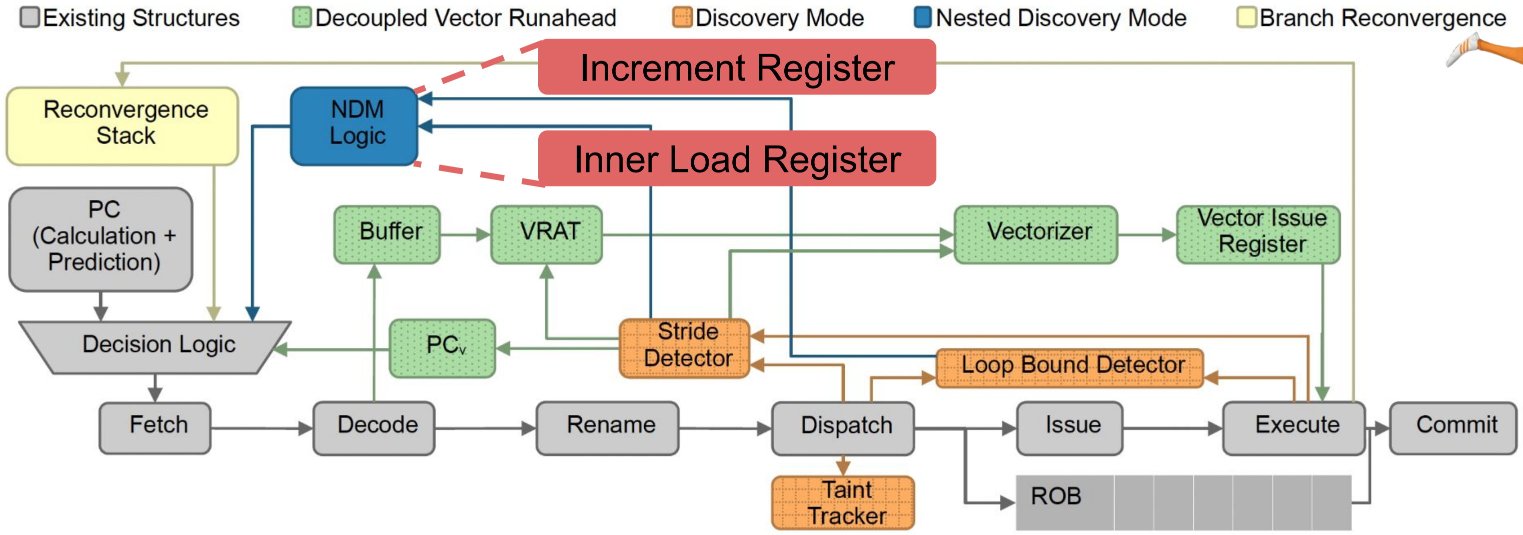
DVR::Branch Reconvergence



PC (48 bits)	Mask (128 bits)
0x1234	111111100000
0x12a0	000000011111

Figure 6: An example reconvergence stack. *The top of the stack stores the current PC and mask. Once the reconvergence point is reached, the stack head is popped and execution proceeds with the next PC and mask.*

DVR::Nested Vector Runahead



★ Nested Discovery Mode (NDM) vectorizes the chain of instructions from the outer striding load to the inner striding load, and discovers loop bounds and data inputs to multiple invocations of the inner loop.

★ Upon reaching the inner striding loop, Nested Vector Runahead (NVR) expands vectorization further to cover the inner loop as well.

What did the paper get **right**?

DVR::Performance

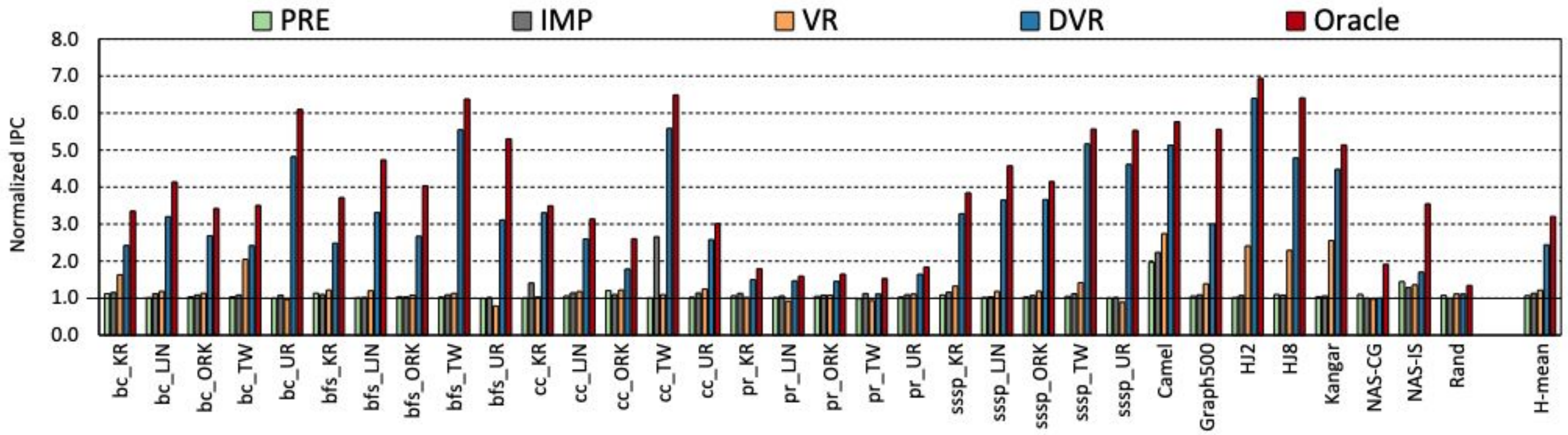


Figure 7: Performance for PRE, VR, DVR and Oracle normalized to a baseline OoO core. *DVR achieves 2.4× higher performance (and up to 6.4×) compared to a baseline OoO core.*

DVR::Performance vs. Growing ROBs

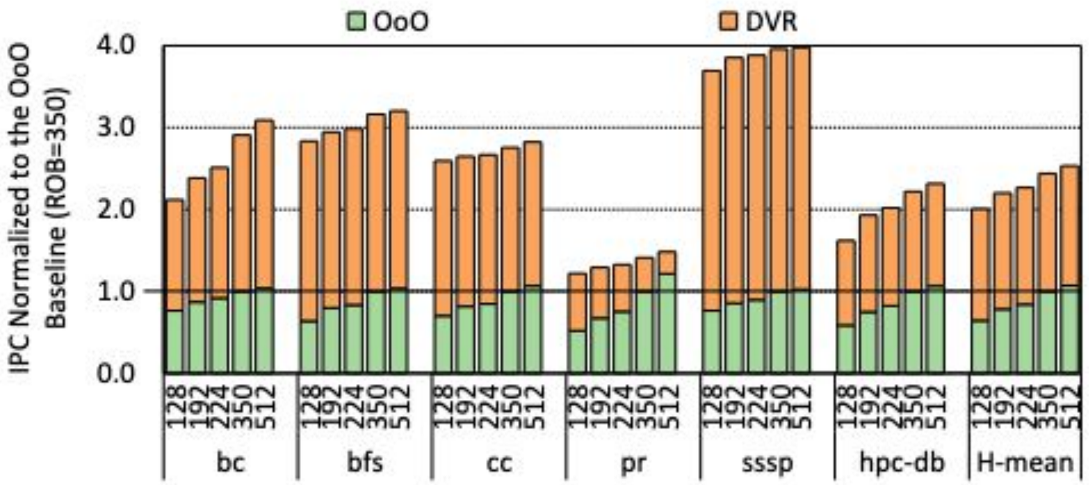


Figure 12: Performance of DVR with increasing ROB size, relative to our baseline OoO core with 350-entry ROB. *The performance gains delivered by DVR continue to increase despite the large size of the ROB.*

Recall…

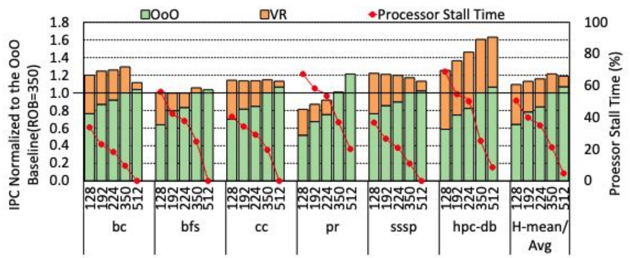


Figure 2: Performance of an OoO core and VR, normalized to a baseline 350-entry ROB OoO core (left axis), and processor stall time due to a full ROB (right axis), as a function of ROB size. *The performance gain of VR diminishes with increasing ROB size, and for some benchmarks overall performance even decreases.*

DVR::Memory Level Parallelism

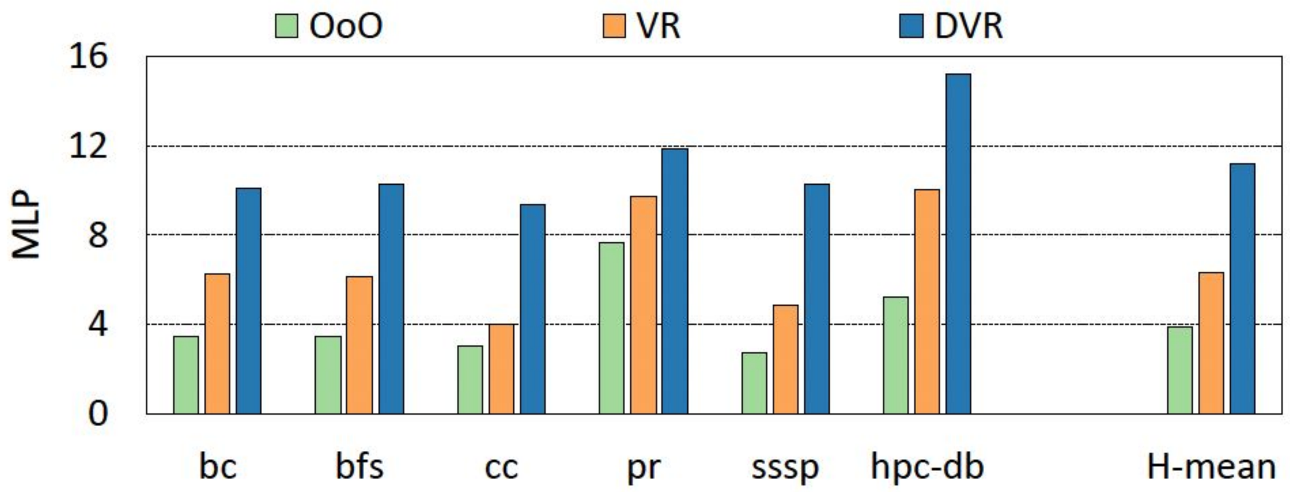


Figure 9: Memory-level parallelism, in terms of MSHRs used per cycle on average, for DVR and VR compared to the base-line OoO core. *DVR generates significantly more parallel outstanding memory accesses.*

DVR::Accuracy and Coverage

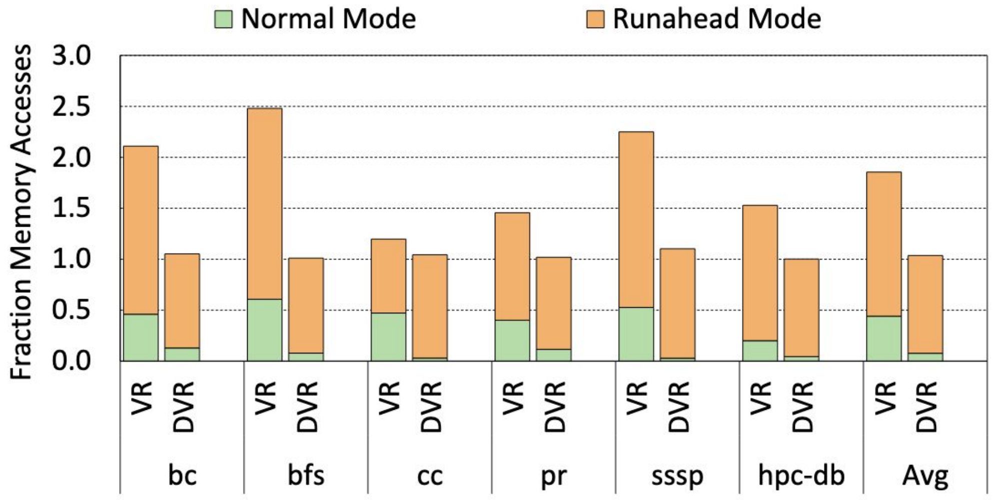


Figure 10: Accuracy and Coverage: number of off-chip memory accesses for VR and DVR normalized to OoO, and fraction of memory accesses in normal versus runahead mode. *DVR successfully prefetches DRAM accesses, converting them into on-chip cache hits when the program subsequently accesses them in normal mode.*

DVR::Timeliness

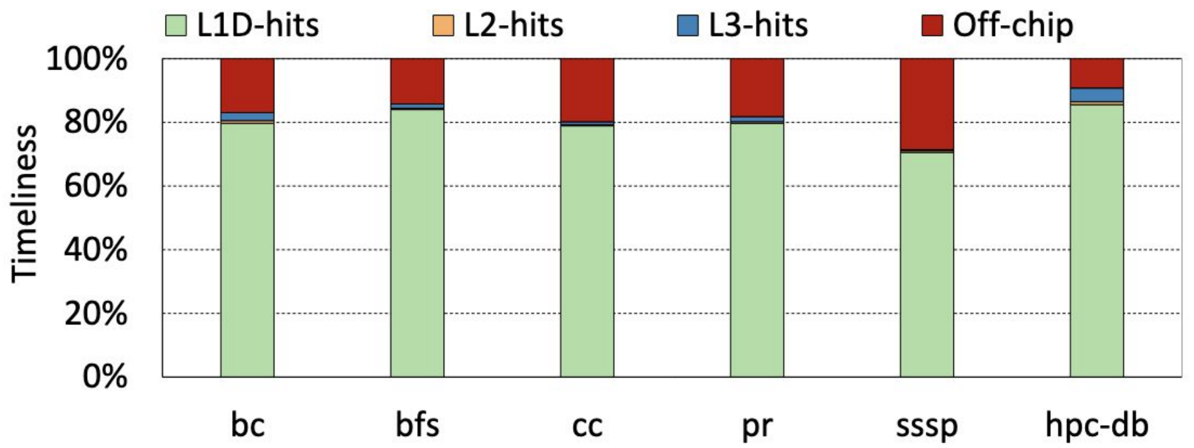


Figure 11: Timeliness: fraction of total prefetched cachelines in runahead mode for which the data is present in the L1-D, L2 and L3 caches during normal mode; 'Off-chip' represents either the cachelines prefetched incorrectly or the cache lines for which the data is still being transferred from memory.

What did the paper get **wrong**?

Remaining Question

- No discussion of area
  - ~1.1k bytes of added storage
  - How much area of logic?

- No discussion of power

What are your **takeaways**?

Takeaways

- ★ The helper thread strikes back!
  - Less switching overhead than traditional runahead
- ★ No need for fancy OoO!
  - Lots of MLP already exposed
  - Mostly RAW dependencies left behind

References

★ O. Mutlu, J. Stark, C. Wilkerson and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, Anaheim, CA, USA, 2003, pp. 129-140, doi: 10.1109/HPCA.2003.1183532.

★ A. Naithani, J. Feliu, A. Adileh and L. Eeckhout, "Precise Runahead Execution," *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, San Diego, CA, USA, 2020, pp. 397-410, doi: 10.1109/HPCA47549.2020.00040.

★ A. Naithani, S. Ainsworth, T. M. Jones and L. Eeckhout, "Vector Runahead," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Valencia, Spain, 2021, pp. 195-208, doi: 10.1109/ISCA52012.2021.00024.

★ Ajeya Naithani, Jaime Roelandts, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. 2023. Decoupled Vector Runahead. In Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23). Association for Computing Machinery, New York, NY, USA, 17–31. https://doi.org/10.1145/3613424.3614255