Hardware Support to Improve Fuzzing Performance and Precision

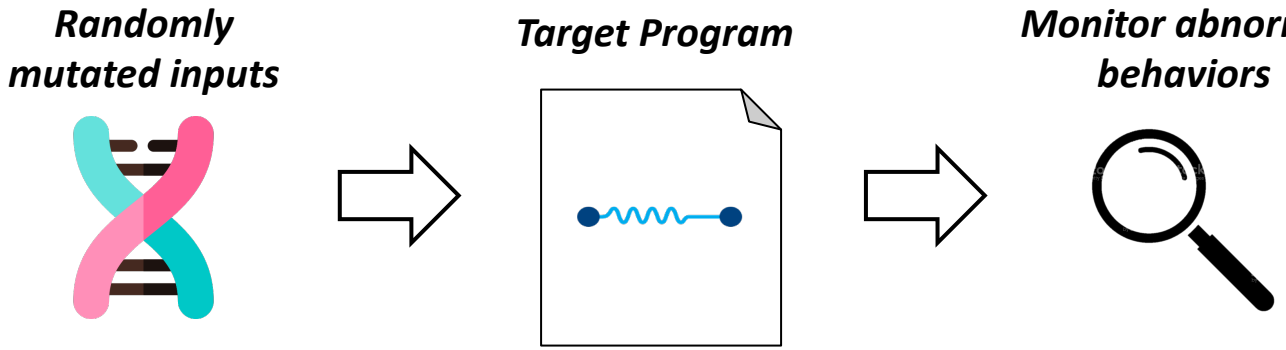
Ren Ding†, **Yonghae Kim**†, Fan Sang, Wen Xu, Gururaj Saileshwar, Taesoo Kim

Georgia Institute of Technology

†Authors contributed equally to this work.

Fuzz Testing

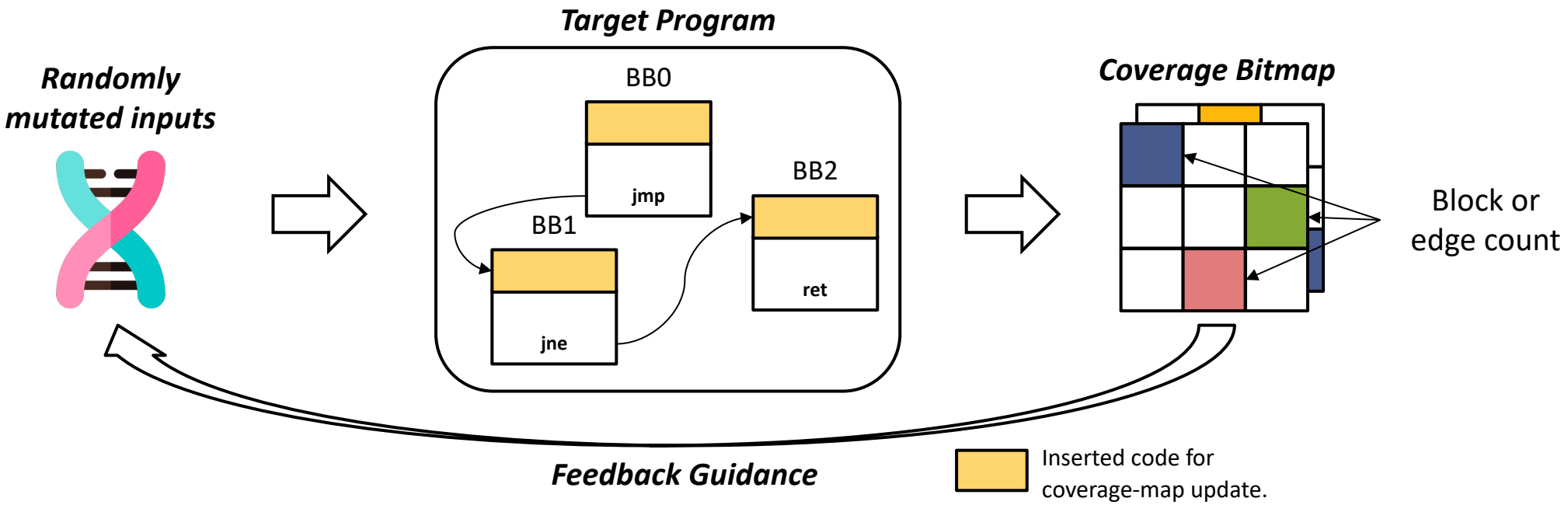
- One of the most effective bug-finding techniques.
  - Minimal manual efforts & pre-knowledge required.
- E.g., ClusterFuzz[1] has uncovered numerous bugs in real-world programs.
  - >25K bugs in Google Chrome.
- Enormous executions require huge computing resources.



[1] Google's in-house fuzzing infrastructure.

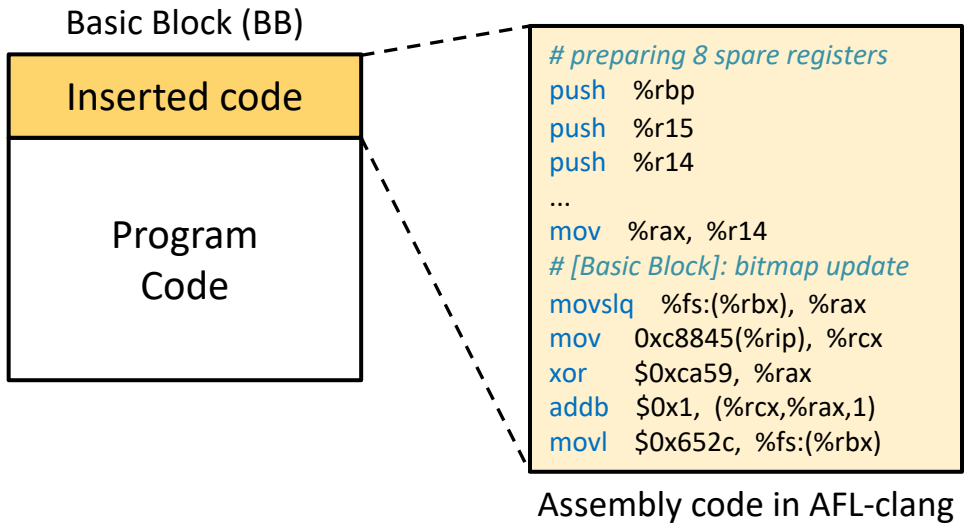
Coverage-Guided Fuzzing

- Inputs reaching more code paths are favored.
  - Thus, feedback guidance depends on *code coverage*.
- To measure code coverage, insert code at each basic block (BB).
- After program execution, count # of reached BBs or BB edges.



AFL's Tracing Overhead

- Software instrumentation still incurs a significant runtime overhead.
  - 60% in AFL-clang and 260% in AFL-QEMU.[2]
    - In AFL-QEMU, binary translation and trap handling overhead further degrades the performance.



1. Instruction overhead.
- Increase program size (2x ↑).[2]

2. Execution overhead.
- Increase cache port contention.
- Reduce cache locality.

[2] The results are measured on an x86 platform across SPEC 2006 benchmarks.

Motivation

- Achieving a low overhead will bring an ***immediate return***.
  - More fuzzing tests in finite time.
  - Substantial computing resource saving.

- Essential task of fuzzing: *Monitor control-flow transfer and manage code-coverage information.*

- In HW, *every control-flow divergence is observed and managed.*
  - There might be enough information to manage code coverage in HW.

Can we design a customized hardware platform for fuzzing?

Overview of SNAP

- A customized hardware platform to enhance fuzzing performance and precision.

Support transparent fuzzing in HW.
- No requirement of source code.

Hardware-based tracing at near-zero cost.
- Reduces tracing slowdown from 600% to 3%.

Expose richer feedback from HW to SW.
- Uses micro-architectural state to improve fuzzing precision.

Generic interfaces supporting variety of fuzzers.
- <100 LoC required in various fuzzers in FuzzBench.

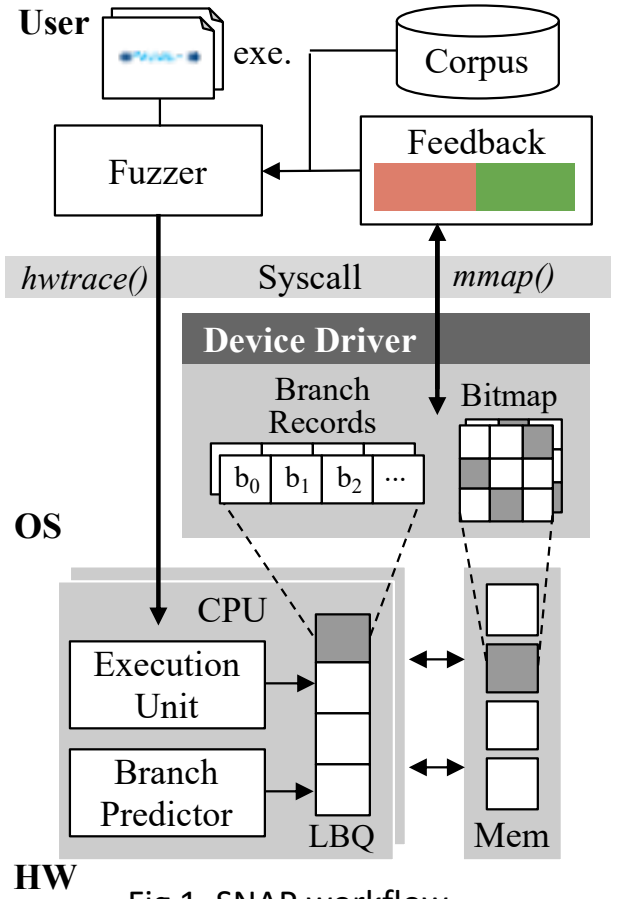


Fig 1. SNAP workflow.

Georgia Tech

SNAP Architecture

- SNAP is prototyped on top of the RISC-V BOOM core.
- Hardware primitives (4.8% area and 6.5% power overhead).
  - **Trace Decision Logic.** ← *Determines which instruction to trace.*
  - **Bitmap Update Queue (BUQ).** ← *Generates bitmap update requests.*
  - **Last Branch Queue (LBQ).** ← *Records last-executed branches.*

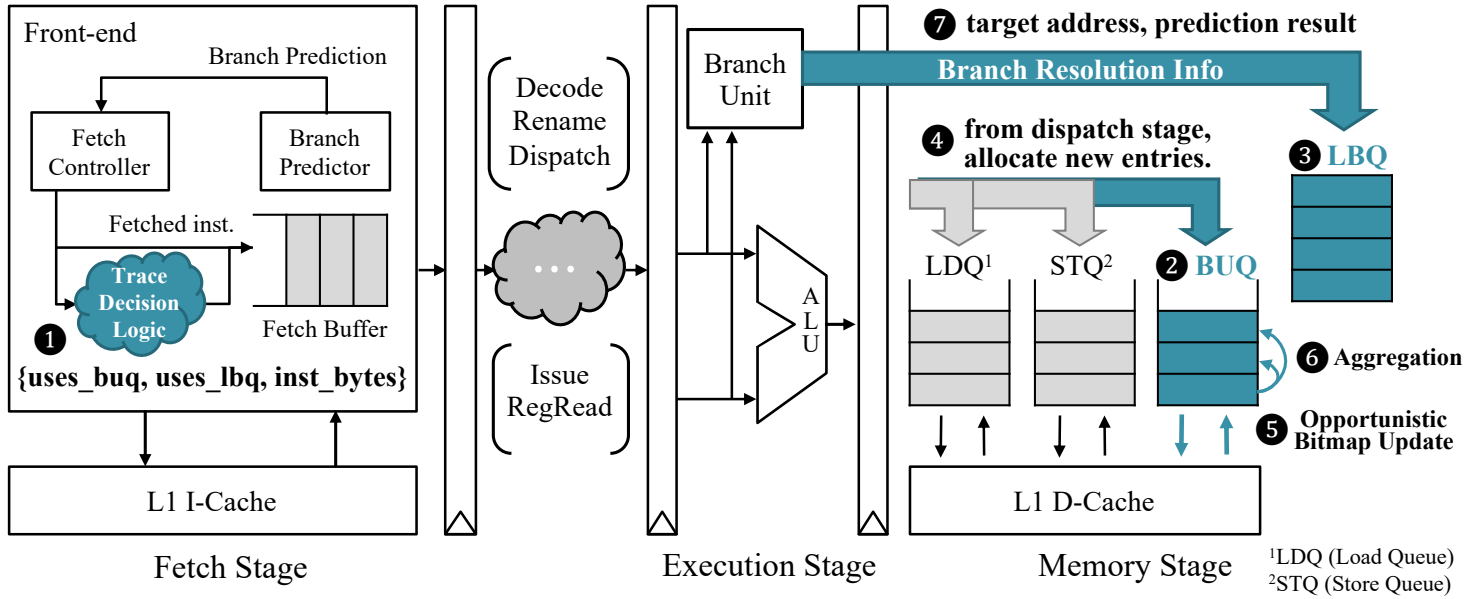


Fig 2. SNAP Architecture.

Trace Decision Logic

- Determines which instruction needs to be traced.
- Tags two bits.
  - **uses_buq**: target instruction of a branch.
  - **uses_lbq**: control-flow instruction.
- Tags instruction bytes (**inst_bytes**).
  - Used in our edge encoding algorithm.
- Lightweight computations.
  - Does not incur clock cycle delays.

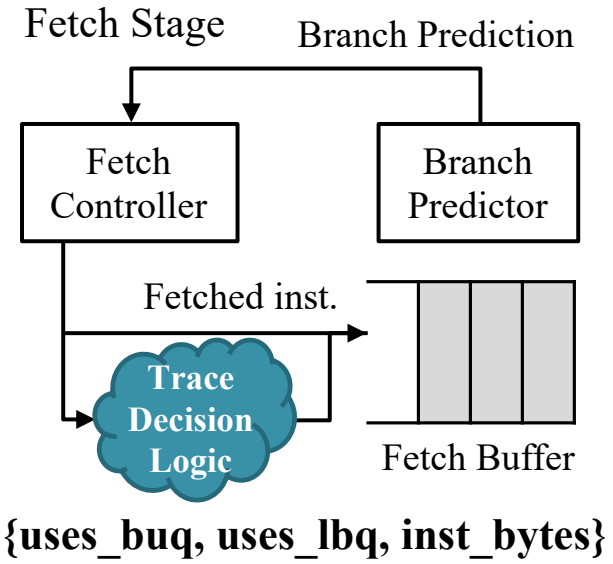


Fig 3. Trace decision logic in fetch stage.

Bitmap Update Queue

- Enqueues instructions tagged with uses_buq.
- Operates through four states:
  - **s_init:** Calculates the bitmap location.
  - **s_load:** Reads edge count from the bitmap location.
  - **s_store:** Writes edge count+1 to the same location.
  - **s_done:** Waits until being deallocated.

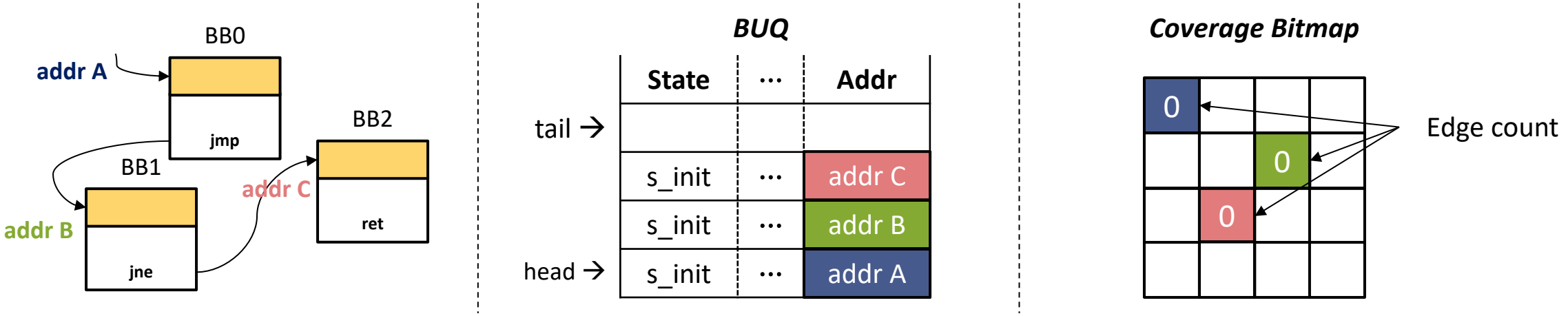


Fig 4. Coverage bitmap update by BUQ.

Bitmap Update Queue

- Enqueues instructions tagged with uses_buq.
- Operates through four states:
  - **s_init:** Calculates the bitmap location.
  - **s_load:** Reads edge count from the bitmap location.
  - **s_store:** Writes edge count+1 to the same location.
  - **s_done:** Waits until being deallocated.

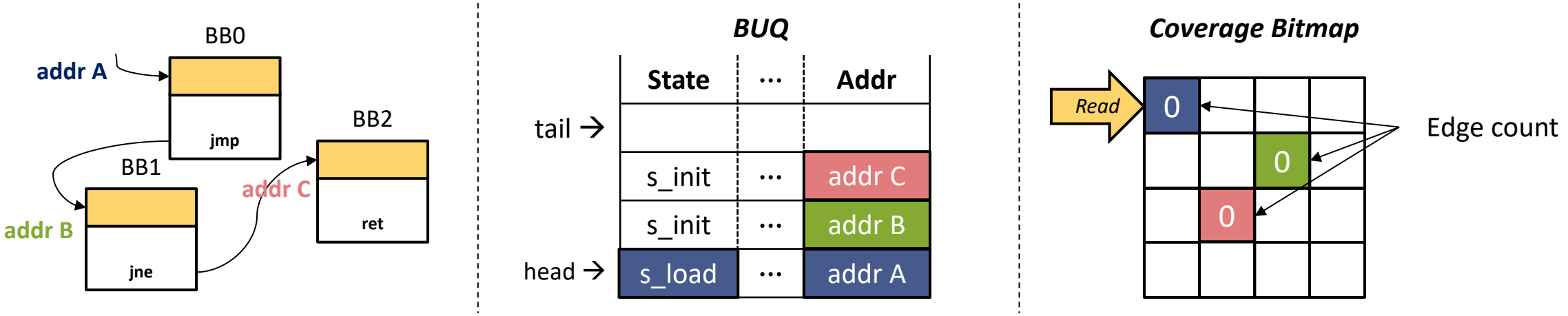


Fig 4. Coverage bitmap update by BUQ.

Bitmap Update Queue

- Enqueues instructions tagged with uses_buq.
- Operates through four states:
  - **s_init:** Calculates the bitmap location.
  - **s_load:** Reads edge count from the bitmap location.
  - **s_store:** Writes edge count+1 to the same location.
  - **s_done:** Waits until being deallocated.

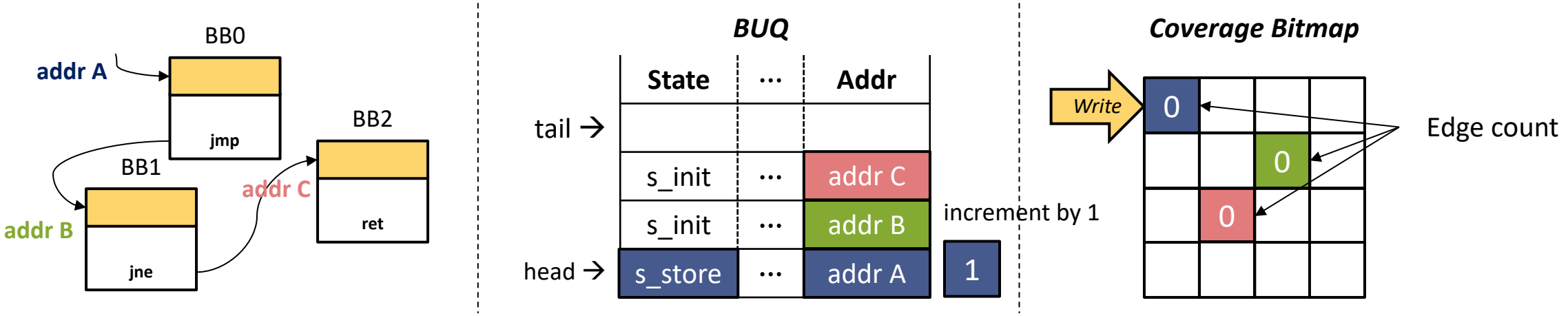


Fig 4. Coverage bitmap update by BUQ.

Bitmap Update Queue

- Enqueues instructions tagged with uses_buq.
- Operates through four states:
  - **s_init:** Calculates the bitmap location.
  - **s_load:** Reads edge count from the bitmap location.
  - **s_store:** Writes edge count+1 to the same location.
  - **s_done:** Waits until being deallocated.

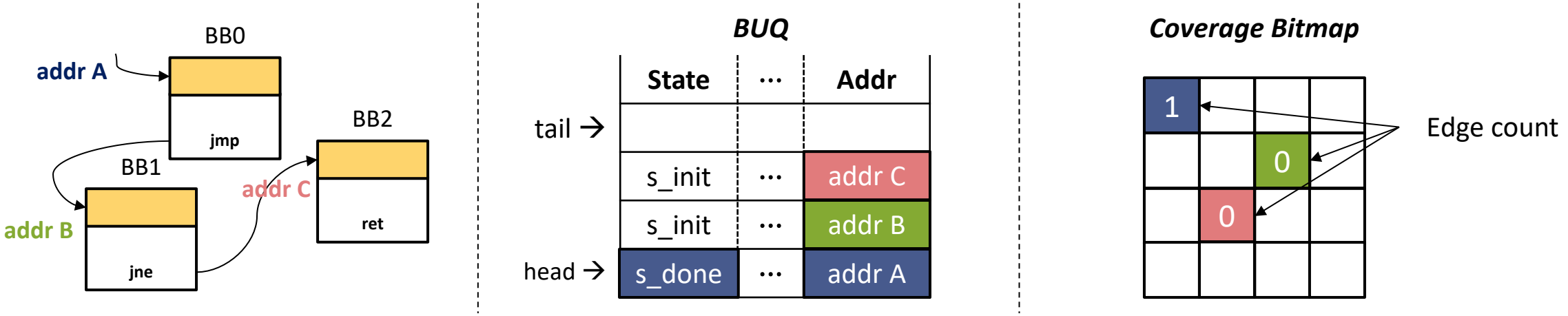


Fig 4. Coverage bitmap update by BUQ.

Edge Encoding in SNAP

- How to calculate the bitmap location?
  - Calculate a **hash** and use it as an **index** to the bitmap.
    → Hash collisions can exist.
- Conventional techniques use:
  - Random basic block ID or memory address of basic block.
- SNAP takes:
  - Target address of a branch (addr).
  - Instruction bytes of a target instruction (inst_bytes).
- Instruction bytes are used to increase the entropy and reduce hash collisions.

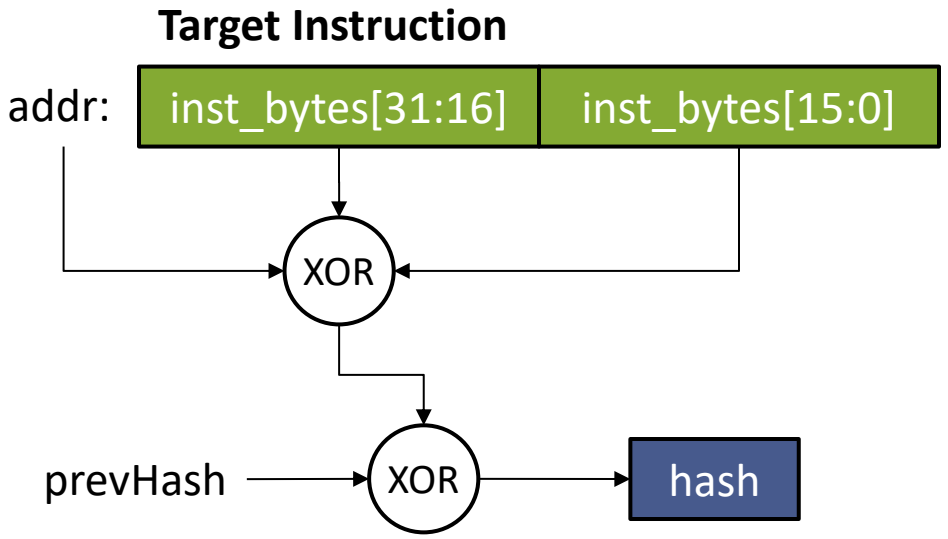


Fig 5. Edge encoding proposed in SNAP.

Last Branch Queue

- Enqueues instructions tagged with uses_lbq.
- Records the information of the last 32 branches.
  - Branch sequence: immediate control-flow context.
  - Prediction Results: approximated data-flow feedback.

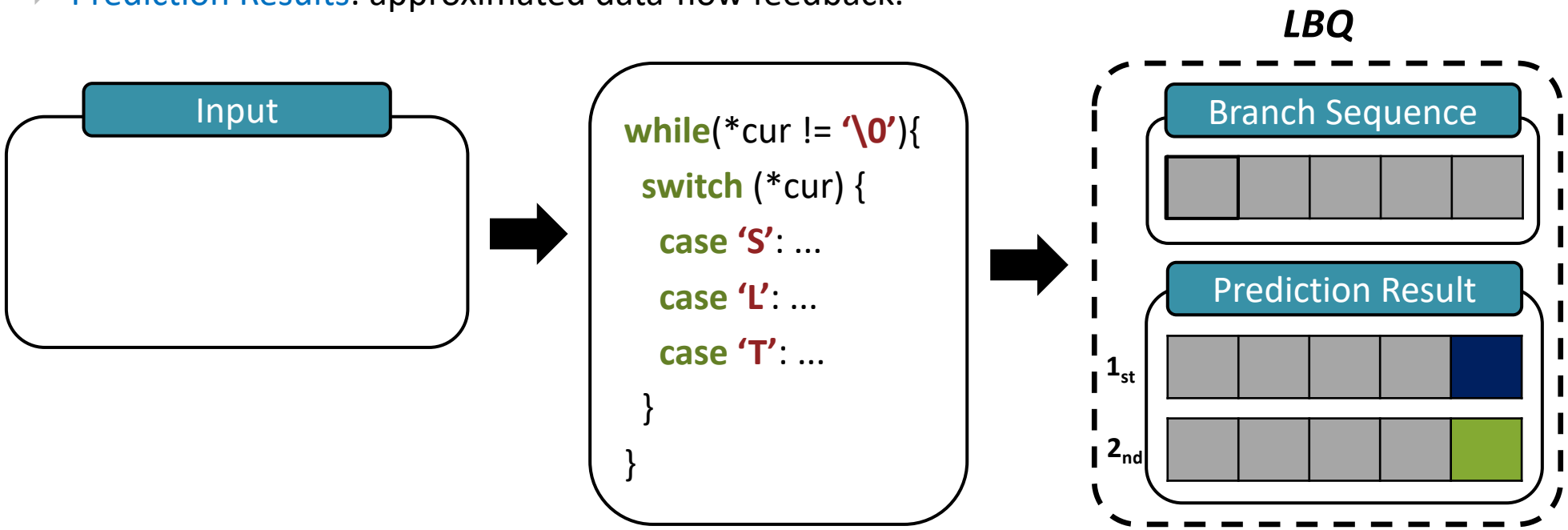


Fig 6. Example of control- and data-flow feedback in LBQ.

Last Branch Queue

- Enqueues instructions tagged with uses_lbq.
- Records the information of the last 32 branches.
  - Branch sequence: immediate control-flow context.
  - Prediction Results: approximated data-flow feedback.

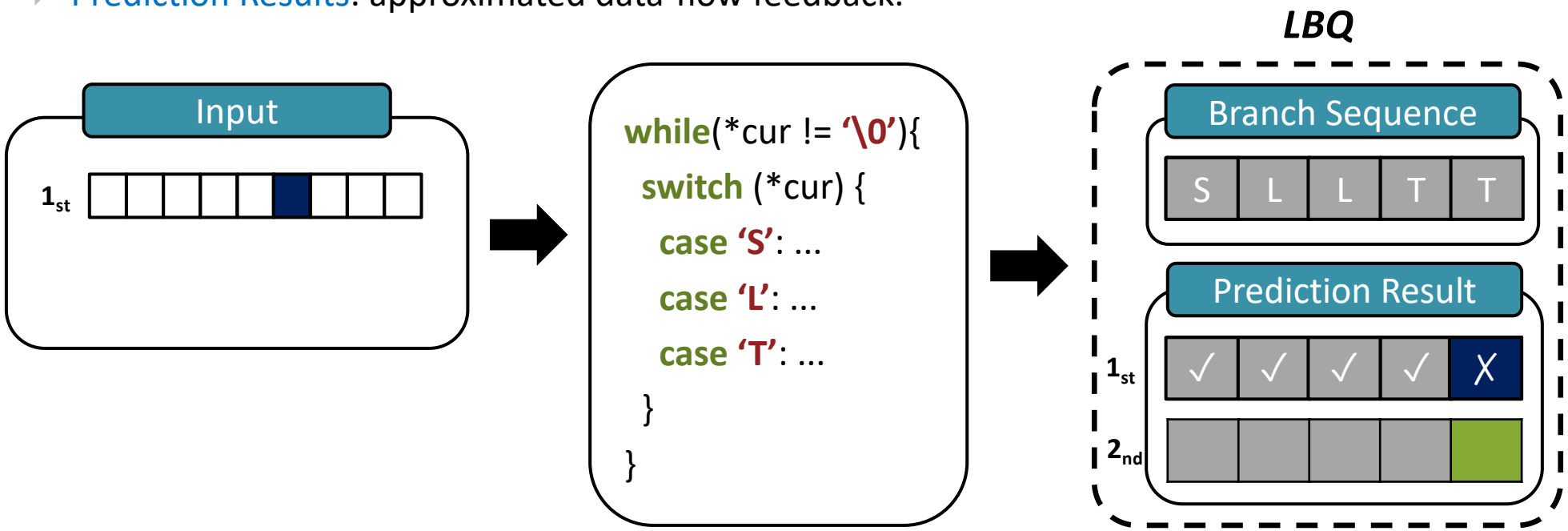


Fig 6. Example of control- and data-flow feedback in LBQ.

Last Branch Queue

- Enqueues instructions tagged with uses_lbq.
- Records the information of the last 32 branches.
  - Branch sequence: immediate control-flow context.
  - Prediction Results: approximated data-flow feedback.

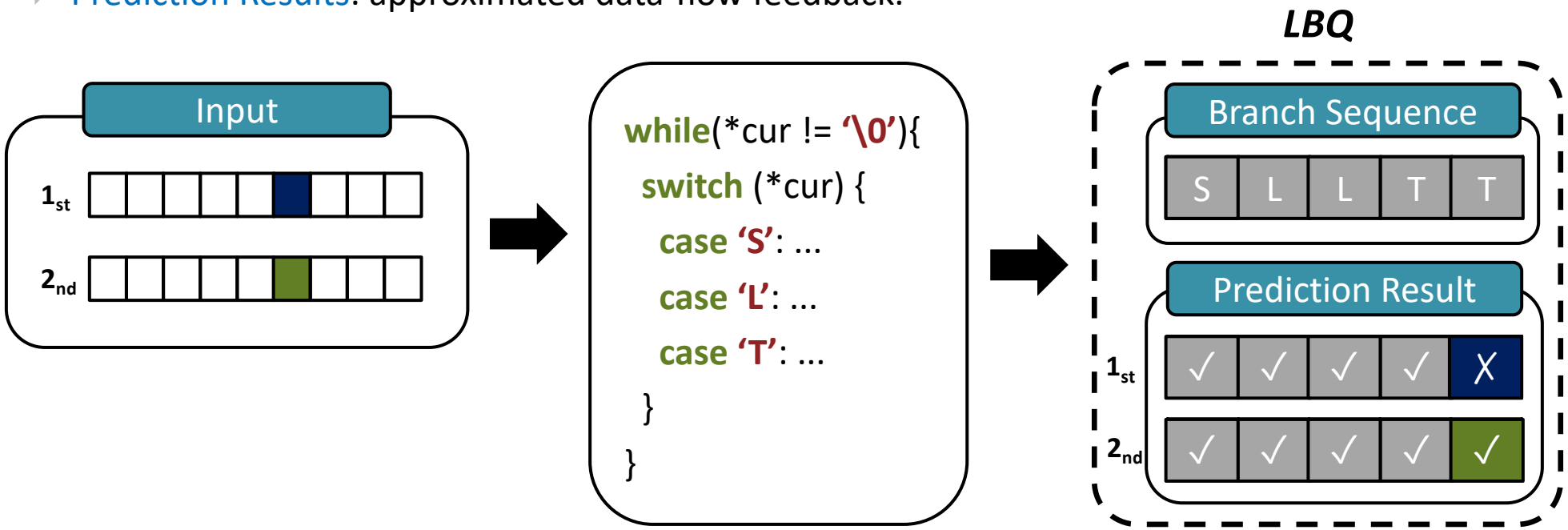


Fig 6. Example of control- and data-flow feedback in LBQ.

Micro-architectural Optimizations

- Bitmap update is not on the critical path of program execution.
- ***Opportunistic bitmap update*.**
  - Send only when unused cache bandwidth is observed or the BUQ is full.
- ***Memory request aggregation*.**
  - Aggregate bitmap update requests to the same bitmap location.

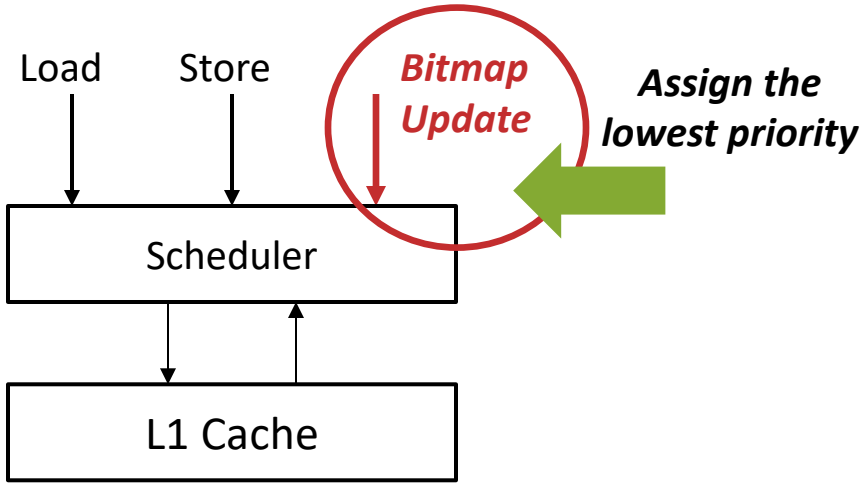


Fig 7. Opportunistic bitmap update.

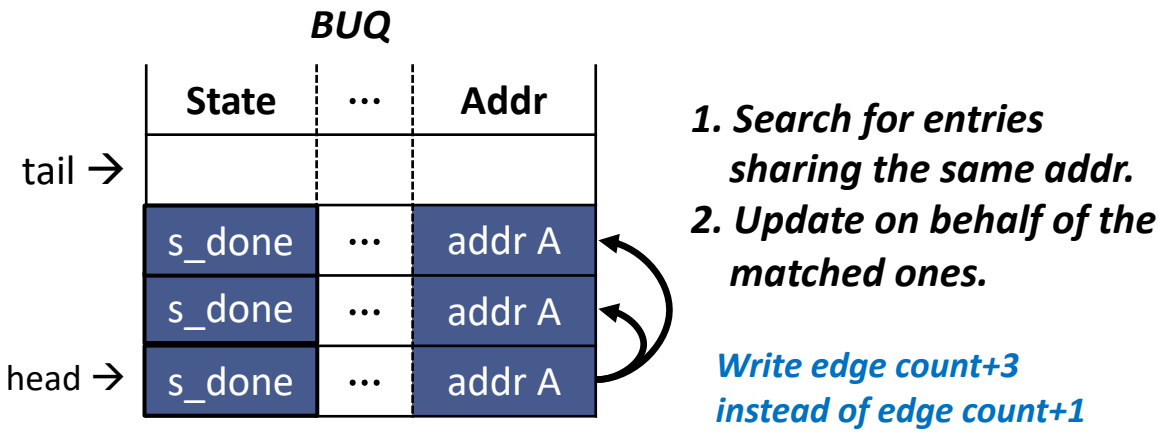


Fig 8. Memory request aggregation.

Experimental Setup

- Prototyped SNAP on top of the RISC-V BOOM core.
- Evaluated on Amazon EC2 F1 controlled by FireSim.
  - An open-source FPGA simulation platform.
- Each FPGA instance runs Linux kernel v5.4.0.
- Enabled user emulation of QEMU v4.1.1 to profile encoding collisions.

- Evaluated benchmark suites:
  - SPEC CPU2006 benchmark suite.
  - Binutils v2.28.

Tracing Overhead

- SNAP incurs a barely 3.14% overhead.
- Significantly outperforms SW solution (AFL-gcc: 599%).

- Main reasons of performance benefits:
  - No extra instructions for tracing/coverage-map update.
  - Reduced memory requests.
    - Request aggregation; 13% on avg. & up to 40%.

- Major cause of overhead: *Cache thrashing*.
  - Memory accesses to the coverage bitmap can evict useful cache lines from caches.

Table 1. Tracing overhead across SPEC 2006 benchmarks

Name	SNAP (%)			AFL-gcc (%)
	32 KB	64 KB	128 KB	
perlbench	7.63	4.28	4.20	690.27
bzip2	2.32	2.21	2.10	657.05
gcc	7.85	5.11	4.97	520.81
mcf	1.75	1.54	1.54	349.83
gobmk	16.92	5.25	4.92	742.98
hmmer	0.72	0.60	0.54	749.56
sjeng	7.29	0.68	0.52	703.44
libquantum	0.80	0.67	0.44	546.67
h264ref	10.37	0.27	0.07	251.56
omnetpp	13.88	5.55	5.37	452.89
astar	0.37	0.30	0.30	422.96
xalancbmk	21.24	11.26	11.11	1109.24
Mean	7.59	**3.14**	3.00	**599.77**

Reduce the overhead to near-zero (3%).

Georgia Tech

Fuzzing Throughput

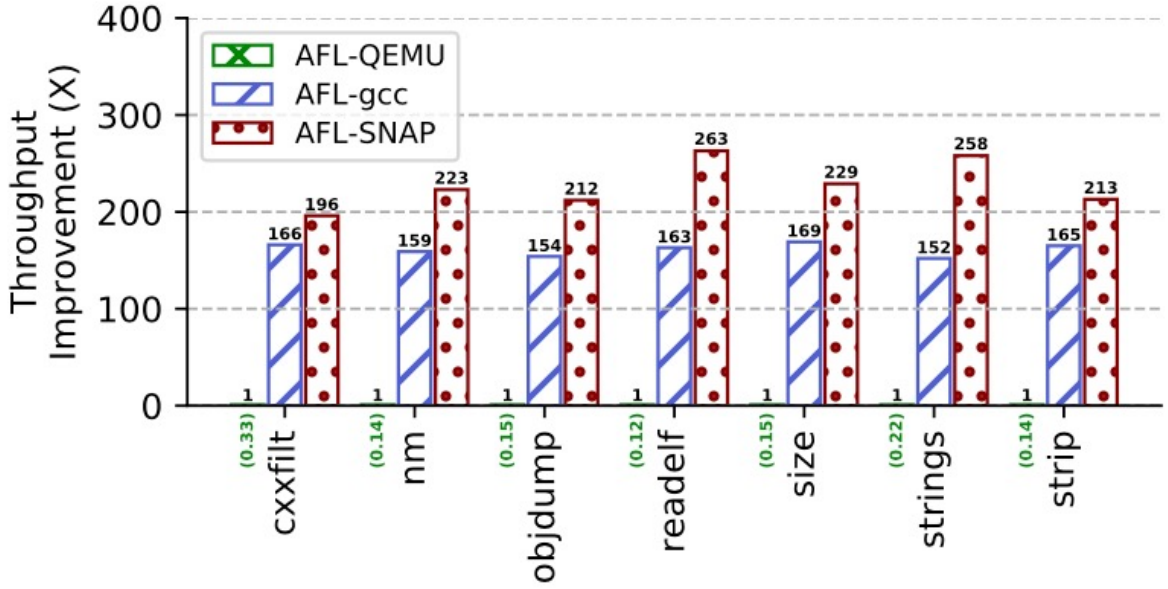


Fig 10. The average execution speed from fuzzing across Binutils v2.28.

- AFL with SNAP (AFL-SNAP) achieves 41% and 228x higher execution speed than AFL-gcc and AFL-QEMU.
- With micro-architectural optimizations, SNAP outperforms the prior work (PHMon) which only achieves a 16x higher speed.

Edge Coverage

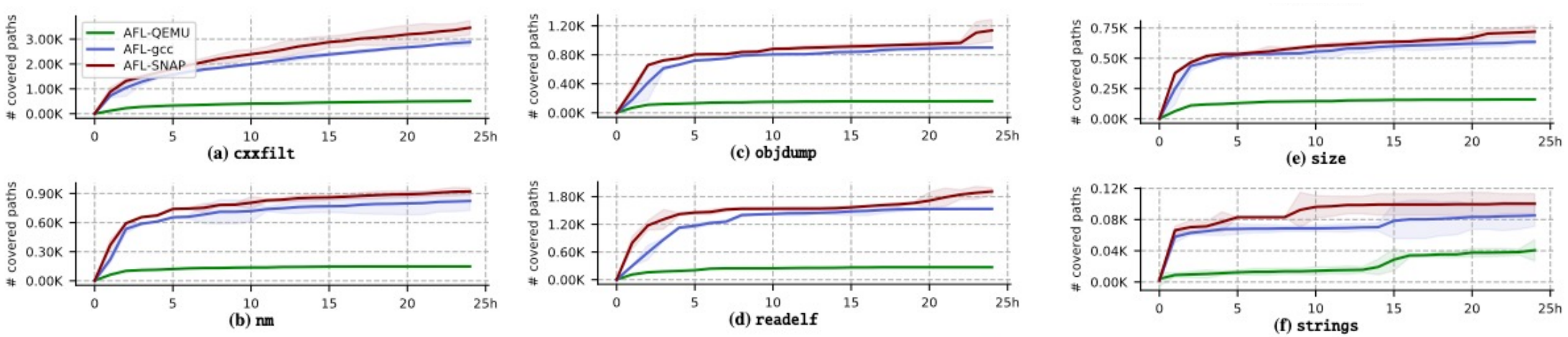


Fig 11. The overall covered paths from fuzzing Binutils v2.28 for 24 hours.

- AFL with SNAP (AFL-SNAP) consistently covers more paths throughout the experiment.
  - AFL-QEMU and AFL-gcc reach 23.26% and 84.59% of the paths discovered by AFL-SNAP, respectively.
- Higher throughput of SNAP is the key contributor to its outperformance.

Discussion

- Usage beyond fuzzing:
  - Efficient coverage estimation for unit testing.
  - Execution fingerprint for logging and forensic purposes.
  - Approximated performance metrics in a specific code region.
    - E.g., branch prediction results.
- Limitations:
  - Kernel coverage filtered by the privilege level.
  - Dynamic code generation with reused code pages.
    - E.g., JIT and library loading/unloading.
  - Dedicated buffer for coverage bitmap storage.
- We hope SNAP motivates future studies and adoption on custom ASICs.

Conclusion

- **SNAP**: a customized hardware platform for better fuzzing performance and precision.
  - We prototyped SNAP in an FPGA evaluation platform at full-system level.

- By leveraging micro-architectural optimizations, SNAP enabled:
  - Transparent hardware-based tracing.
  - Richer feedback on execution states.

  } *at near-zero performance cost.*

- The adopted fuzzer running on SNAP (AFL-SNAP) achieved:
  - 41% and 228x higher fuzzing throughput compared to AFL-gcc and AFL-QEMU.
  - Thus, higher code coverage throughout fuzzing.
  - Dramatically lower cost for fuzzing-as-a-service.

- SNAP is available at https://github.com/sslab-gatech/SNAP.

Thank you!

Questions?