Adaptive GPU Cache Bypassing

Yingying Tian* , Sooraj Puthoor†, Joseph L. Greathouse†,
Bradford M. Beckmann†, Daniel A. Jiménez*

Texas A&M University*, AMD Research†

Outline

- Background and Motivation
- Adaptive GPU Cache bypassing
- Methodology
- Evaluation Results
- Case Study of Programmability
- Conclusion

Graphics Processing Units (GPUs)

- Tremendous throughput
- High performance computing
- Target for general-purpose GPU computing
  - Programming model: CUDA, OpenCL
  - Hardware support: cache hierarchies
  - AMD GCN: 16KB L1, NVIDIA Fermi: 16KB/48KB configurable

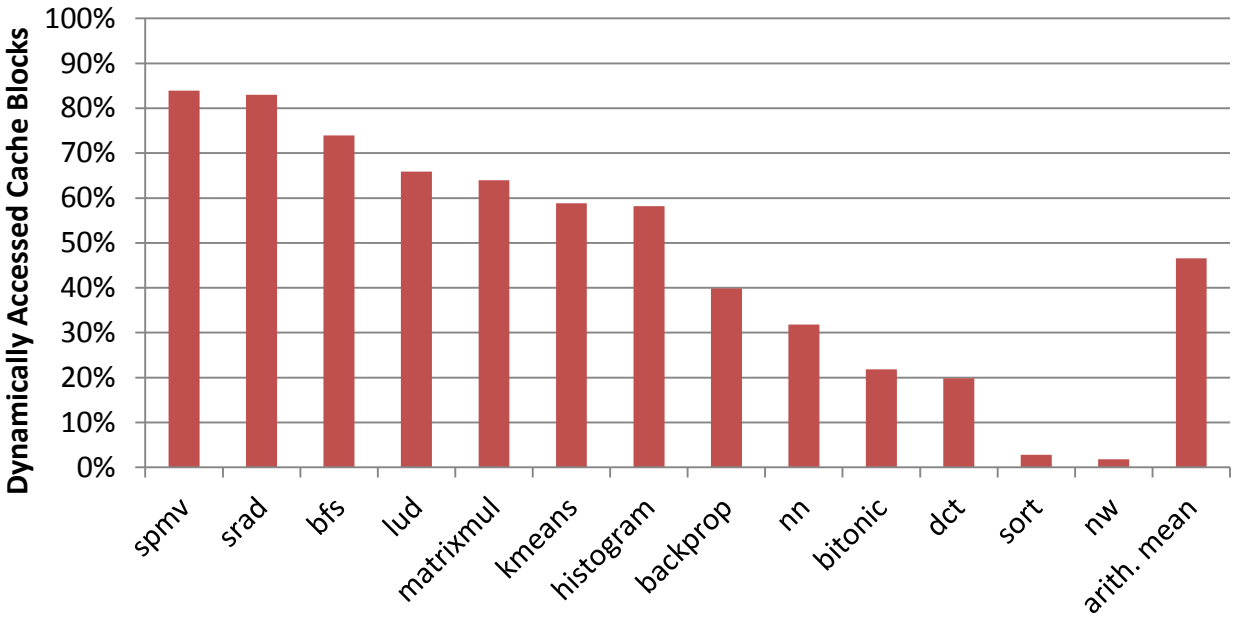
AMD

Inefficiency of GPU Caches

- Thousands of concurrent threads
  - Low per-thread cache capacity
- Characteristics of GPU workloads
  - Large data structures
  - Useless insertion consumes energy
  - Reducing the performance by replacing useful blocks
- Scratchpad memories filter temporal locality
  - Less reuse caught in caches
  - Limit programmability

Memory Characteristics

- Zero-reuse blocks: inserted into caches without being accessed again until evicted



- 46% (max. 84%) of L1 cache blocks are only accessed once before eviction
- Consume energy; pollute cache; cause more replacement

Motivation

- GPU caches are inefficient
- Increasing cache sizes is impractical
- Inserting zero-reuse blocks wastes power without performance gain
- Inserting zero-reuse blocks causes useful blocks being replaced to reduce performance

Motivation

- GPU caches are inefficient
- Increasing cache sizes is impractical
- Inserting zero-reuse blocks wastes power without performance gain
- Inserting zero-reuse blocks causes useful blocks being replaced to reduce performance

Objective: Bypass zero-reuse blocks

Outline

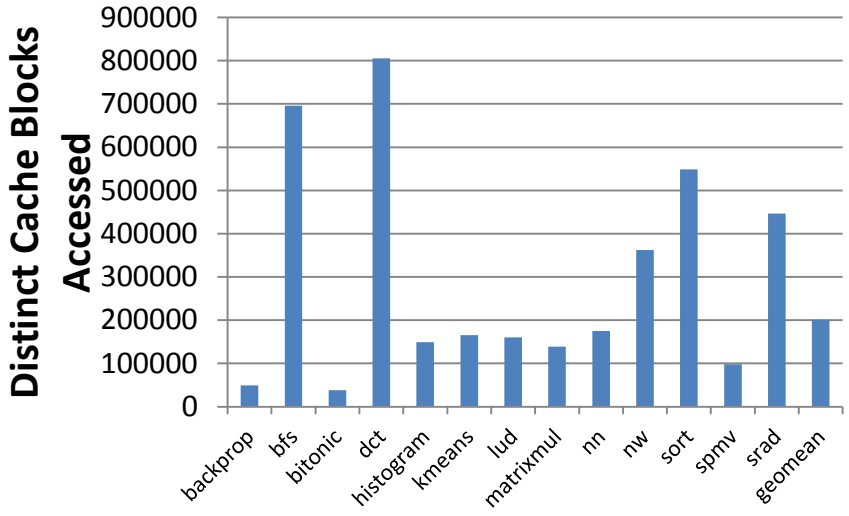
- Background and Motivation
- Adaptive GPU Cache bypassing
- Methodology
- Evaluation Results
- Case Study of Programmability
- Conclusion

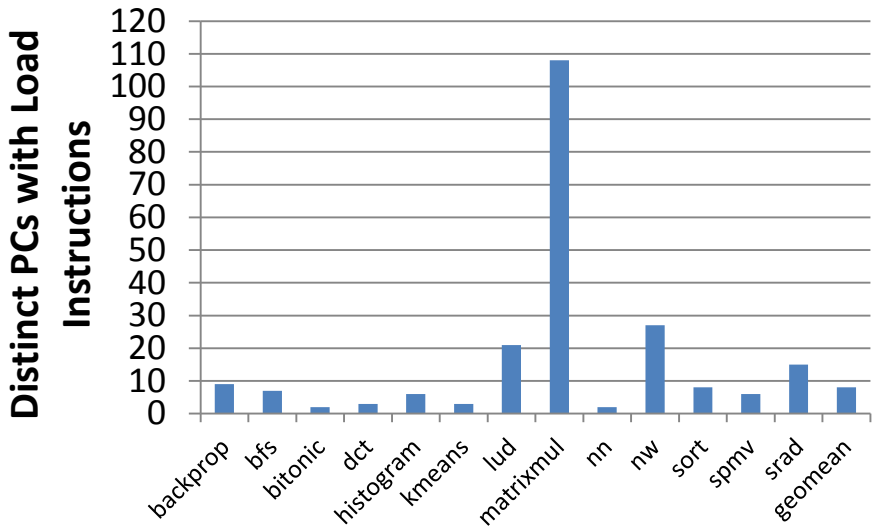
Bypass Zero-Reuse Blocks

- Static bypassing on resource limit [Jia *et. al* 2014]
  - Degrade the performance

- Dynamic bypassing
  - Adaptively bypass zero-reuse blocks
  - Make prediction on cache misses
  - What information shall we use to make prediction?
    - Memory addresses
    - Memory instructions

Memory Address vs. PC





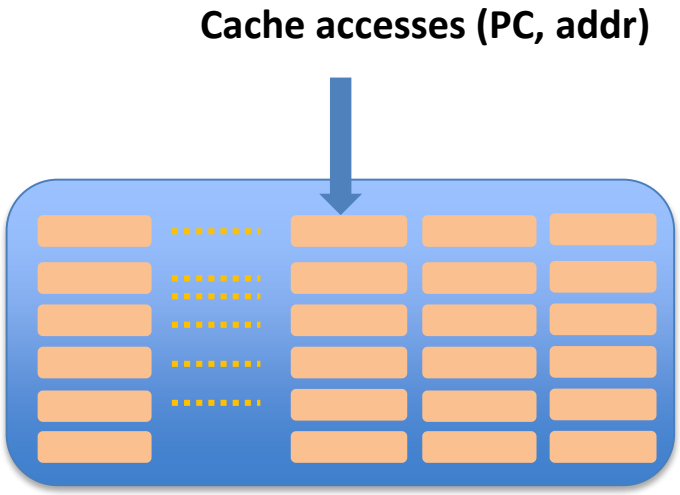
- Using memory addresses is impractical due to storage
- Using memory instructions consumes far less overhead
- PCs generalize the behaviors of SIMD with high accuracy

Adaptive GPU Cache Bypassing

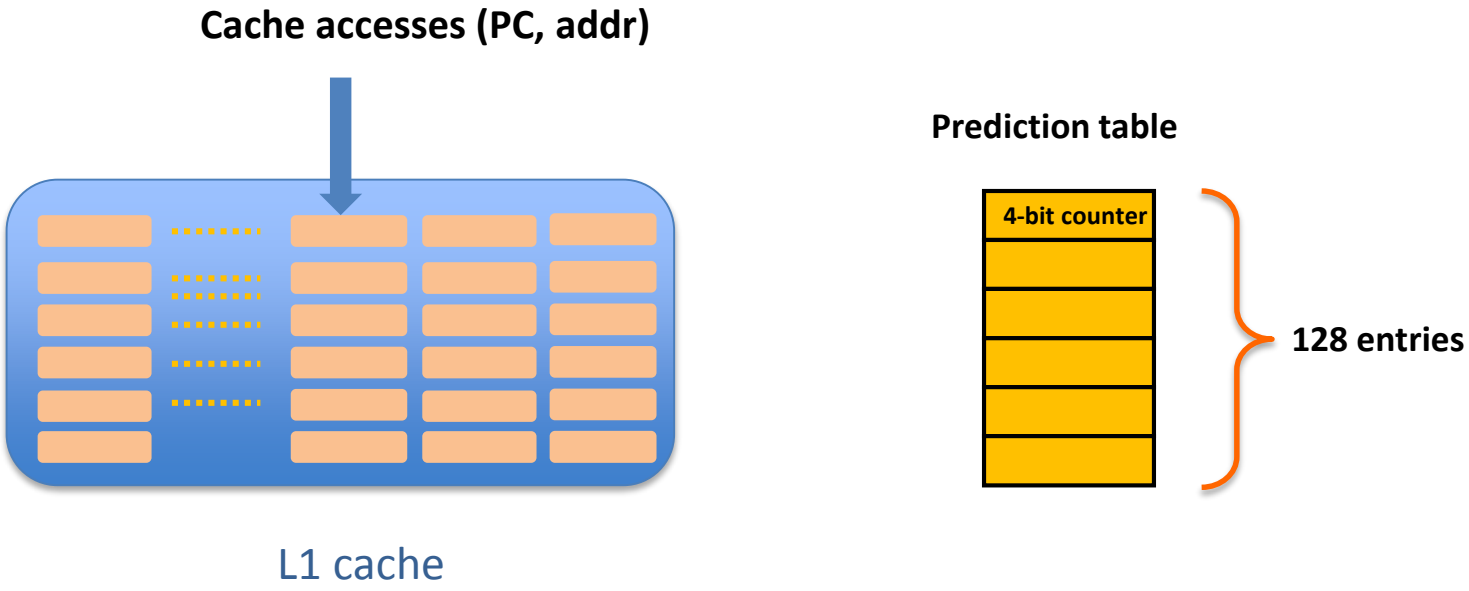
PC-based dynamic Bypass Predictor

- Inspired by dead block prediction in CPU caches [Lai *et. al* 2001][Khan *et. al* 2010]
  - If a PC leads to the last access to one block, then the same PC will lead to the last access to other blocks
- On cache misses, predict if it is zero-reuse
  - If yes, requested block will bypass the cache
- On cache hits, verify previous prediction and make new prediction

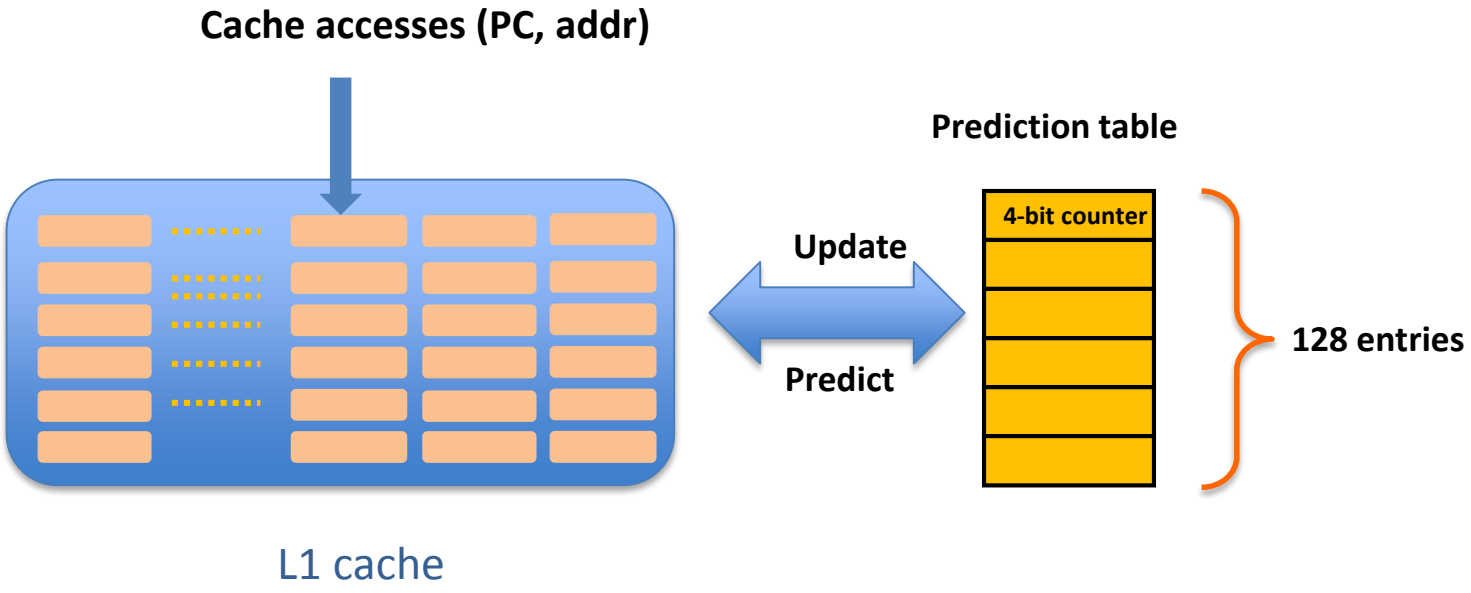
Structure of PC-based Bypass Predictor



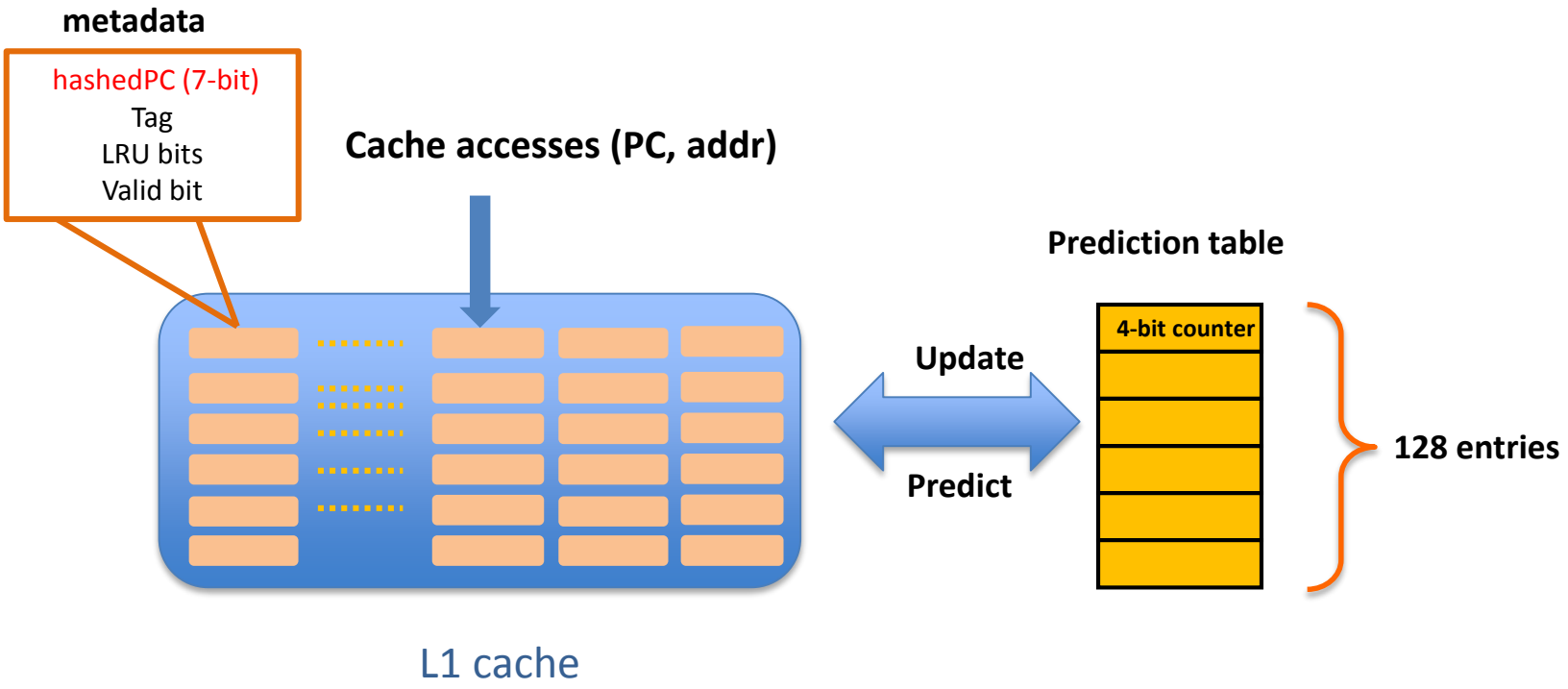
Structure of PC-based Bypass Predictor



Structure of PC-based Bypass Predictor

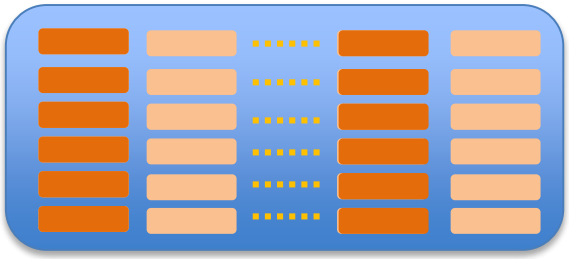


Structure of PC-based Bypass Predictor



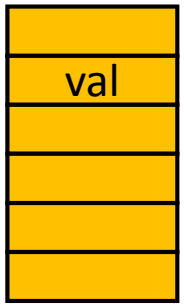
HashedPC: the hash value of the PC of the last instruction that accessed this block

Example

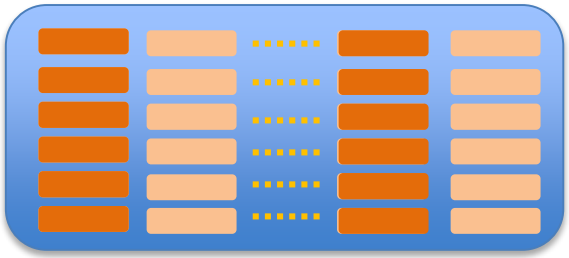


L1 cache

Prediction table

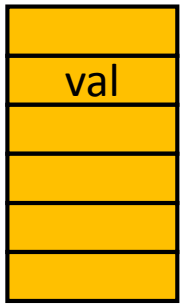


Example

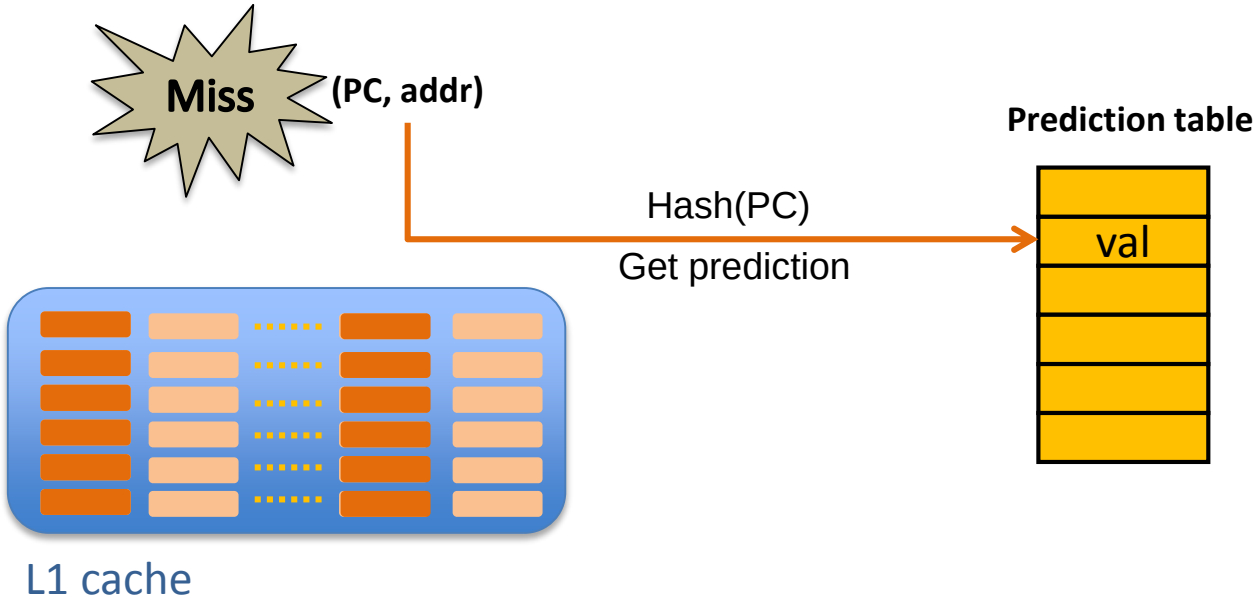


L1 cache

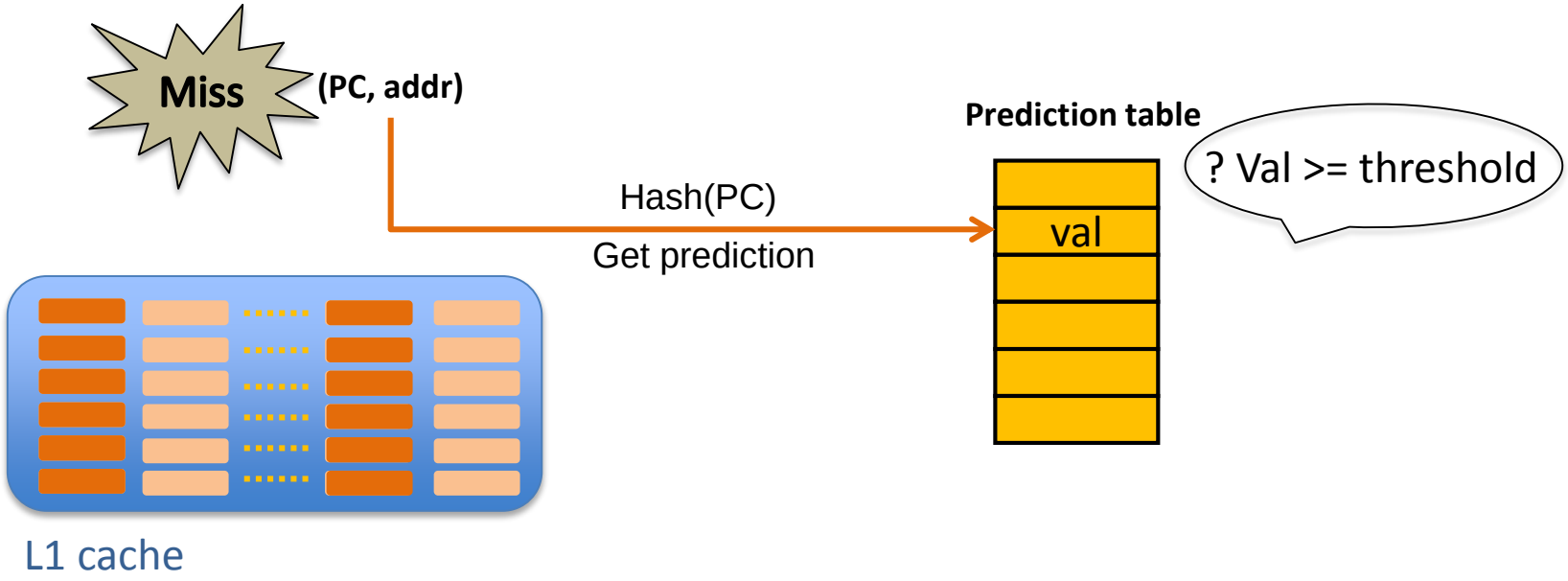
Prediction table



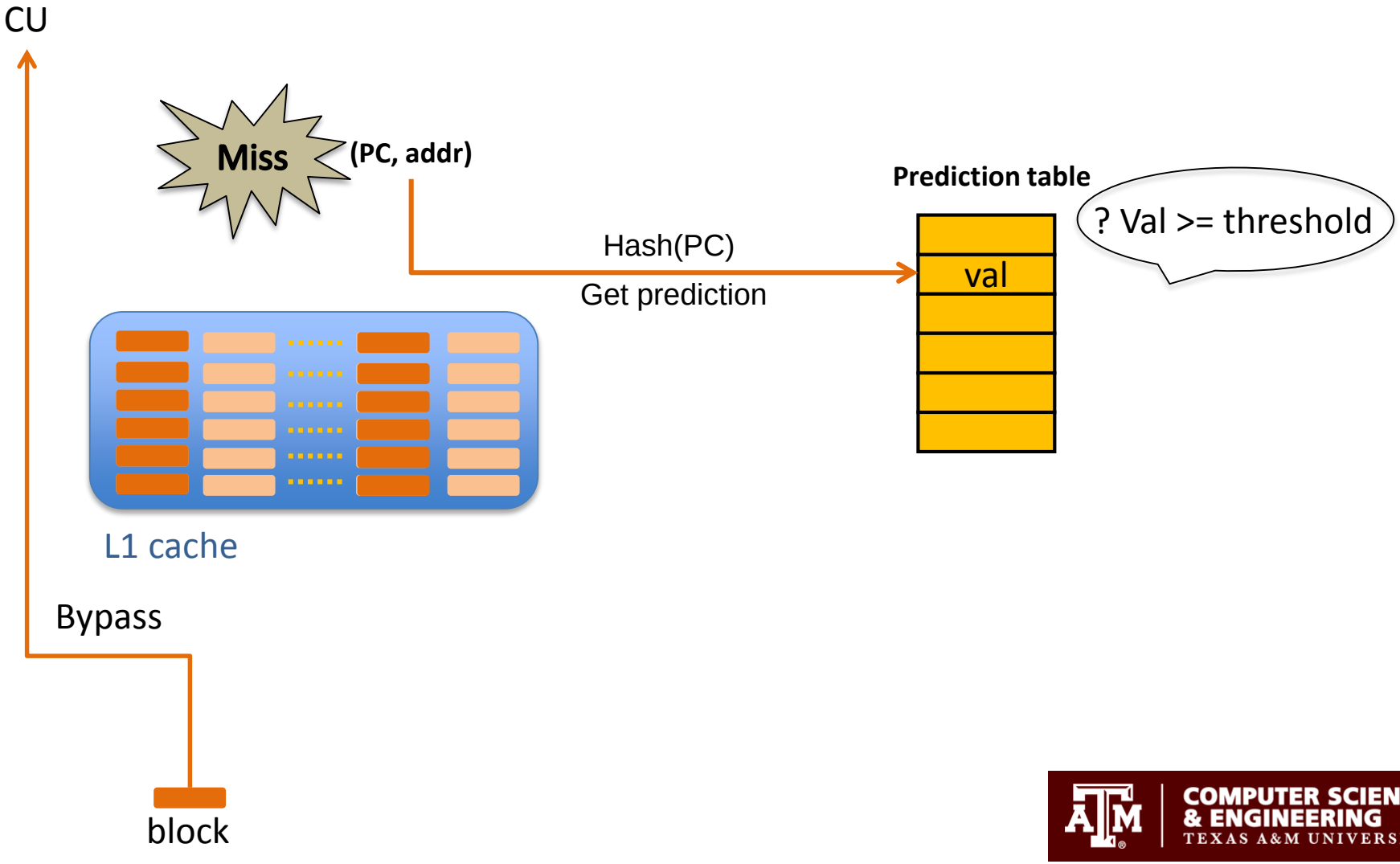
Example



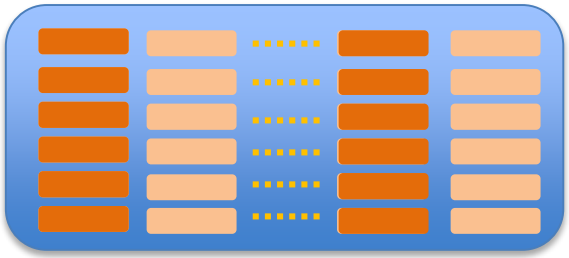
Example



Example

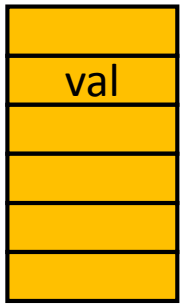


Example

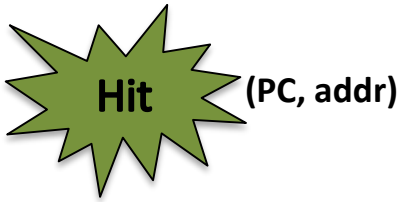


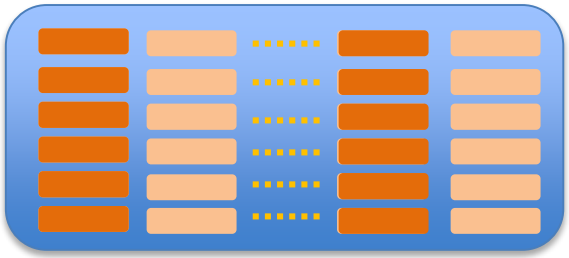
L1 cache

Prediction table



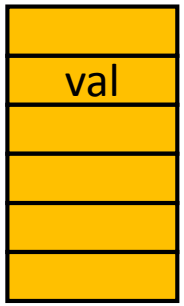
Example



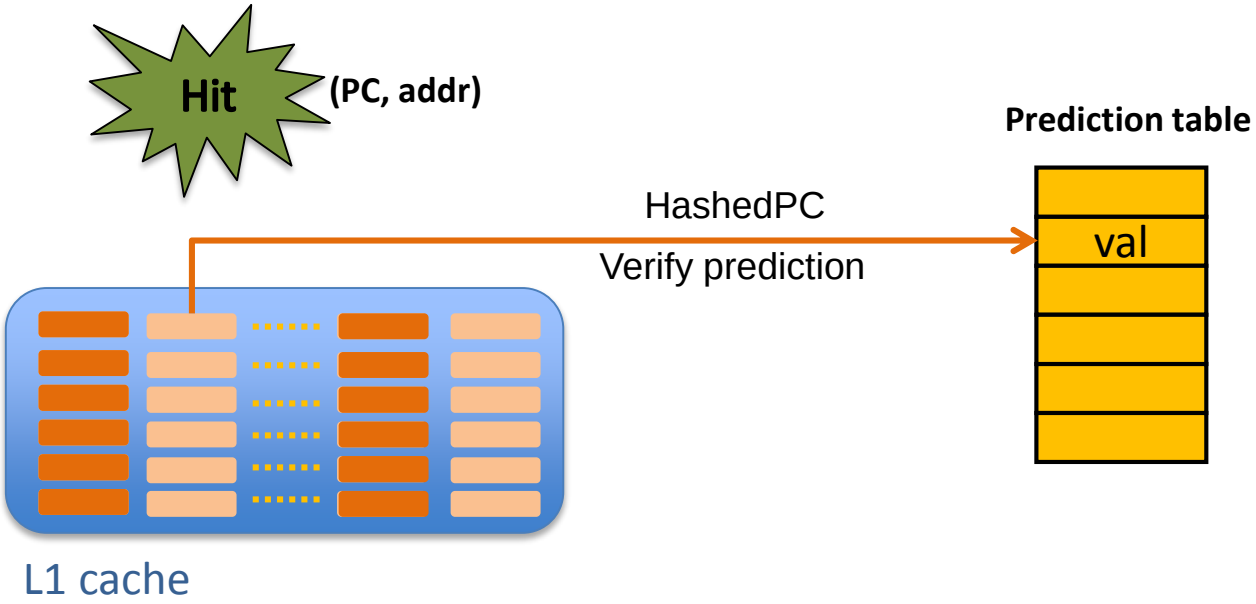


L1 cache

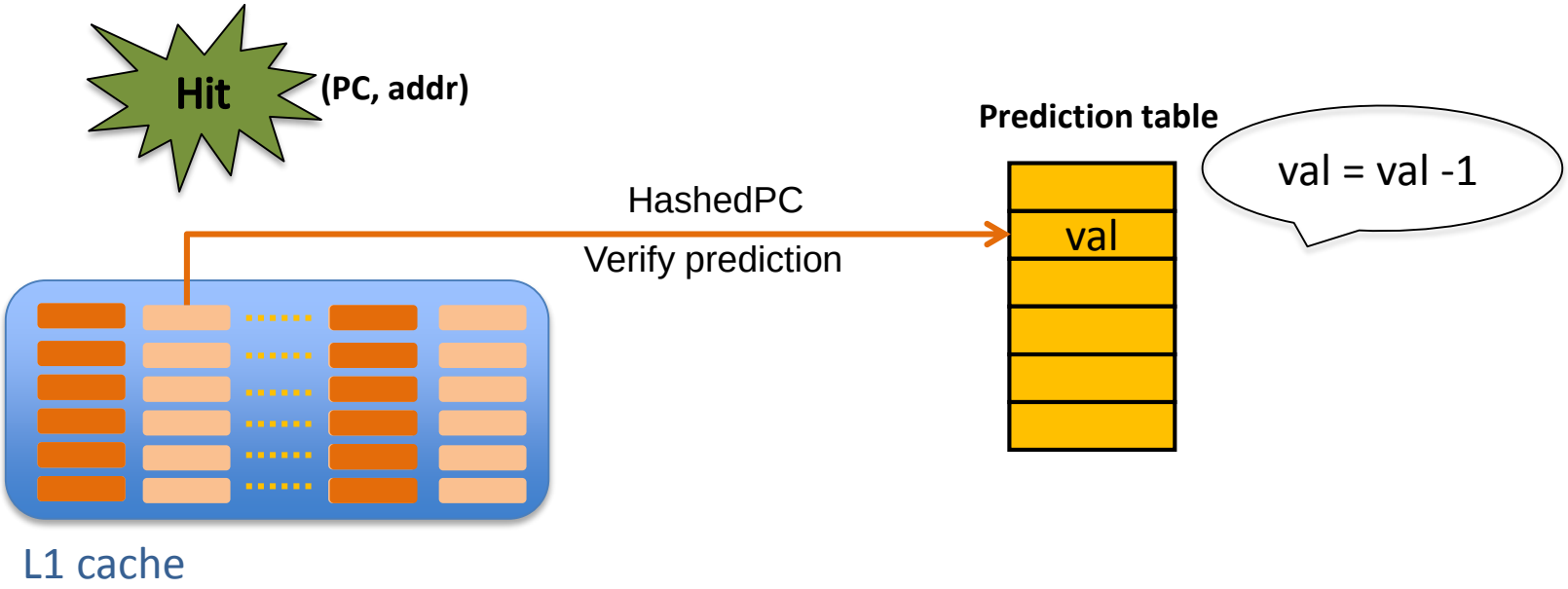
Prediction table



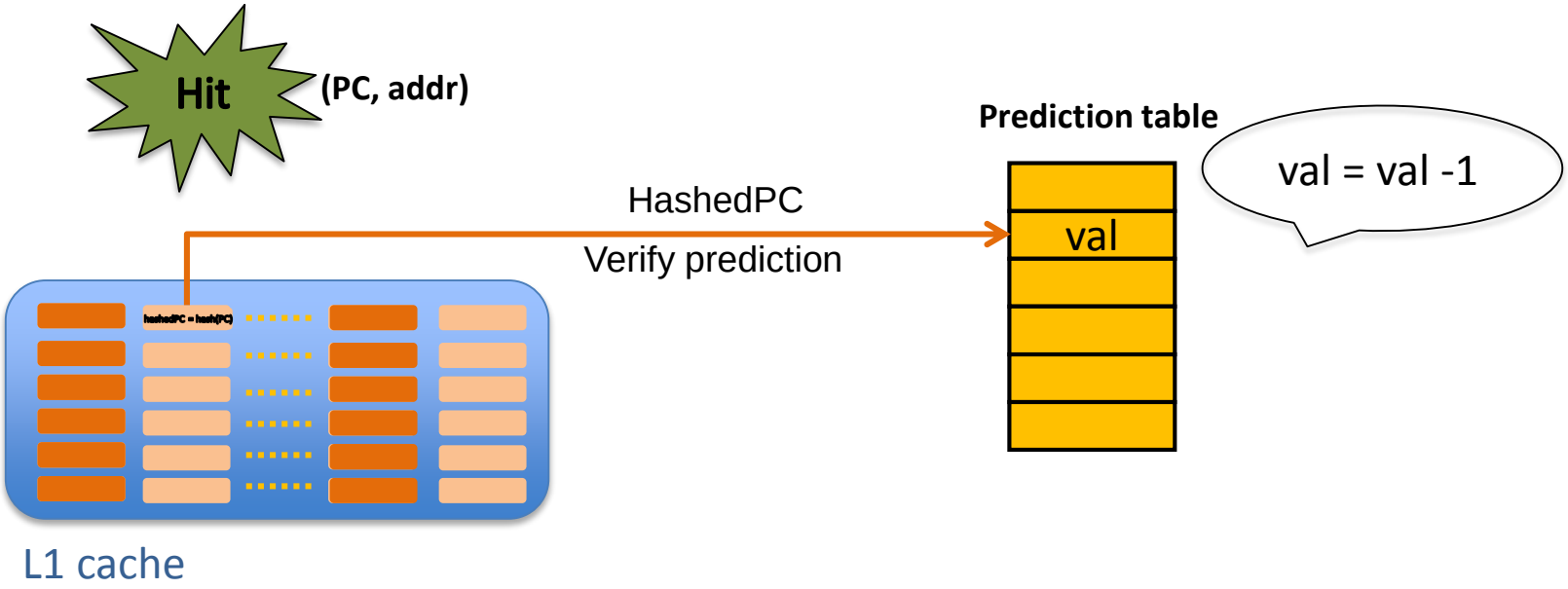
Example



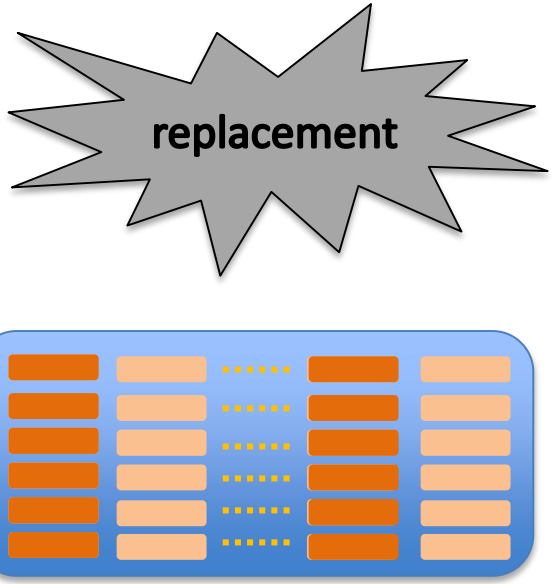
Example

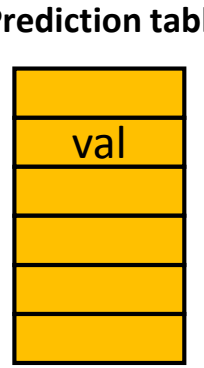


Example

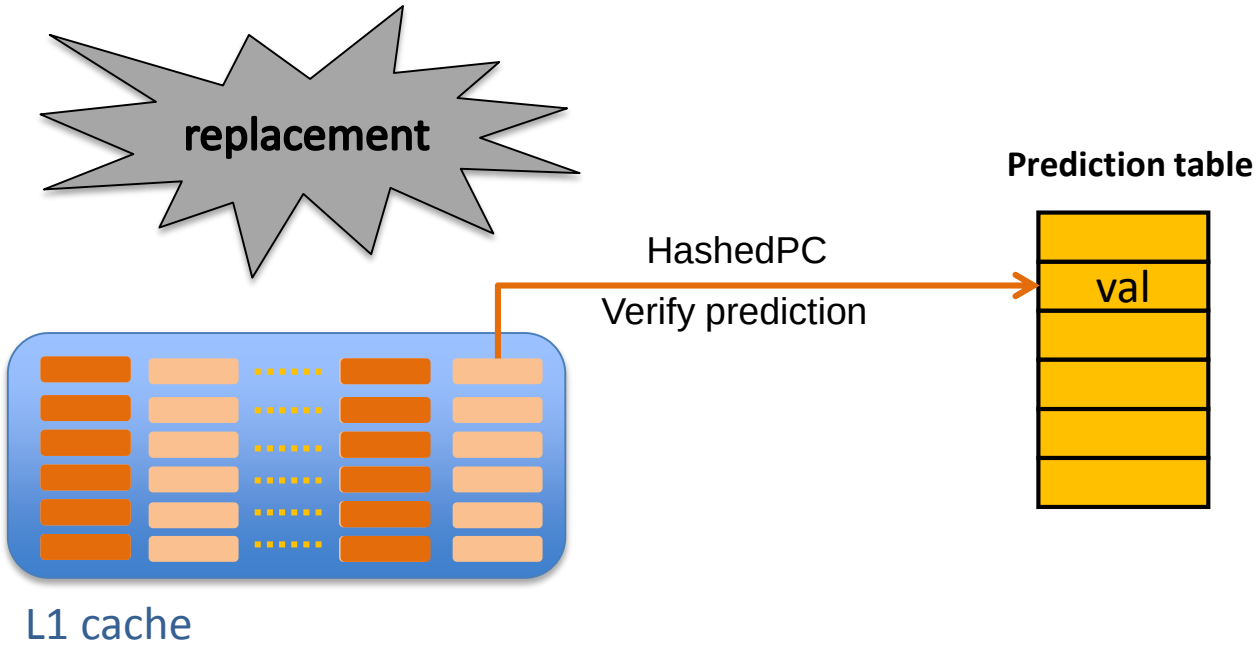


Example

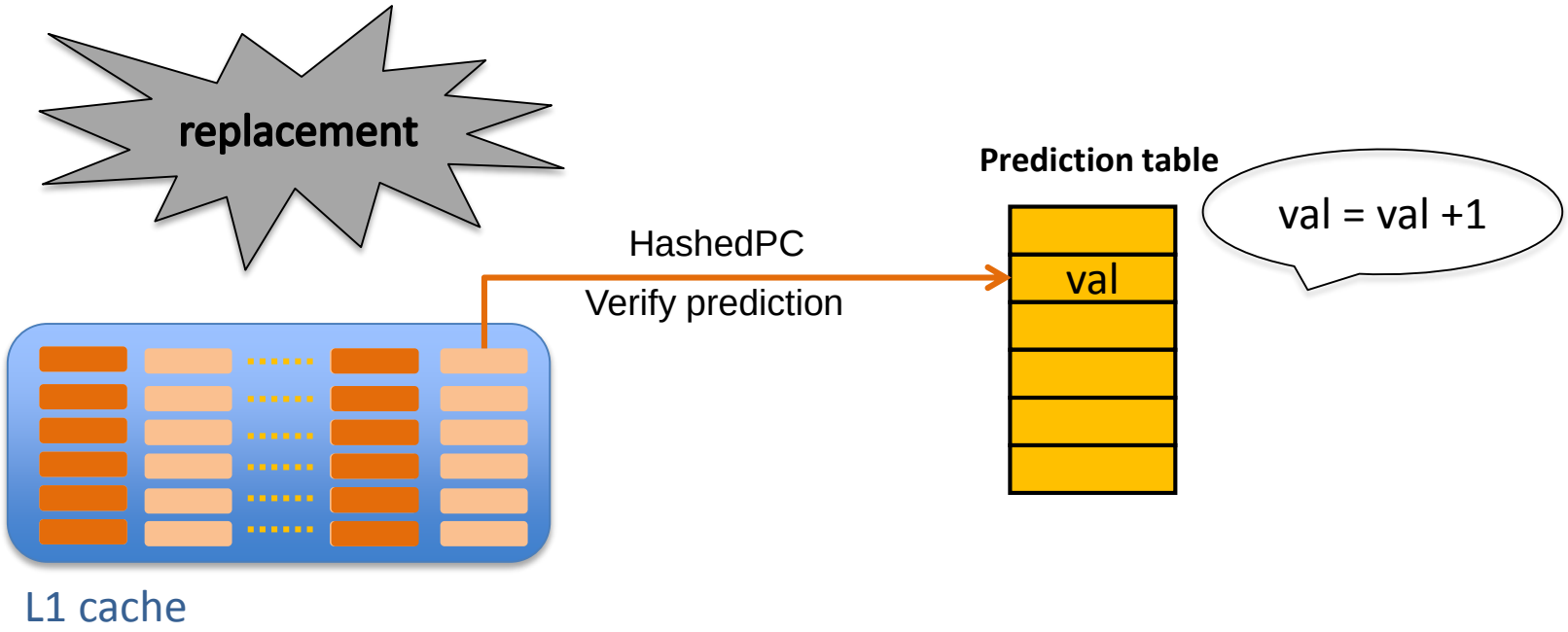




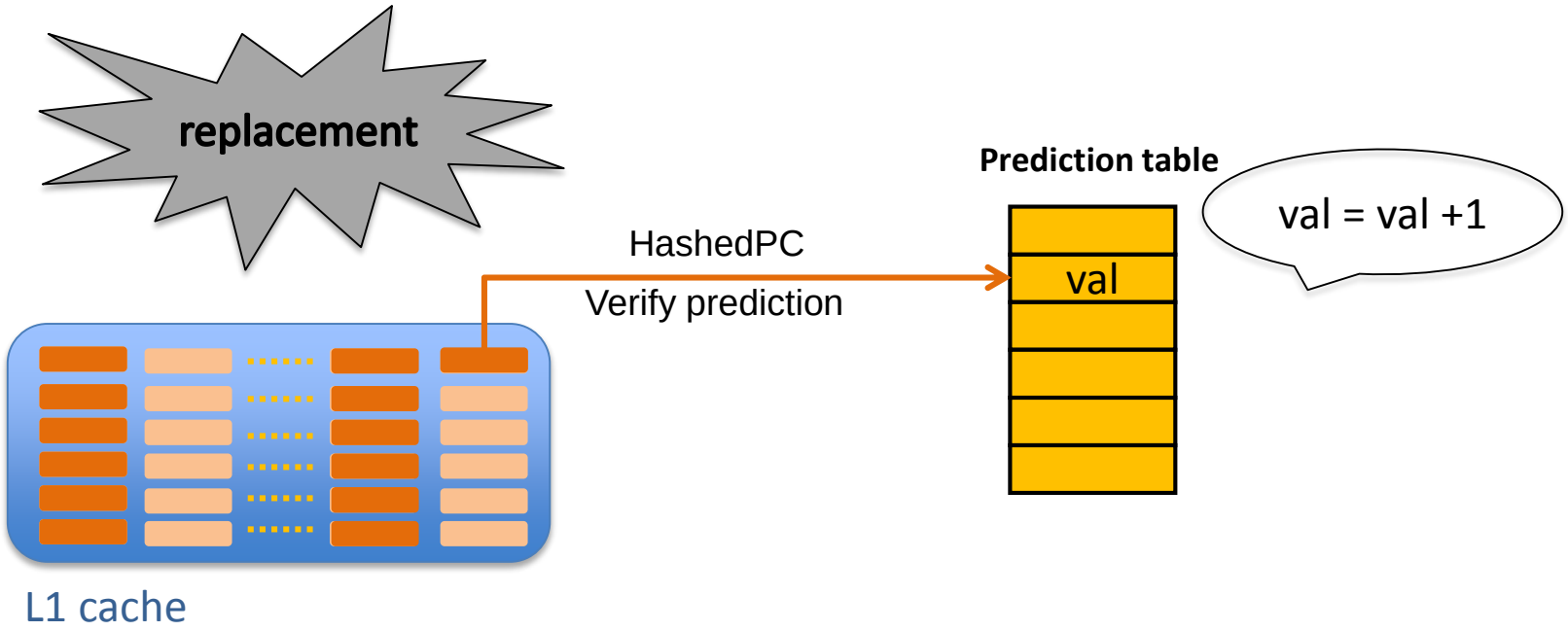
Example



Example


replacement
HashedPC
Verify prediction
Prediction table
val
val = val +1
L1 cache

Example



Misprediction Correction

- Bypass misprediction is irreversible
- Cause additional penalties to access lower level caches
- Utilize this procedure to help verify misprediction
- Each L2 entry contains an extra bit: *BypassBit*
  - BypassBit == 1 if the requested block will bypass L1
  - Intuition: if it is a bypass misprediction, this block should be re-referenced soon, and will be hit in the L2 cache
- Miscellaneous
  - Set dueling
  - Sampling

Outline

- Background and Motivation
- Adaptive GPU Cache bypassing
- Methodology
- Evaluation Results
- Case Study of Programmability
- Conclusion

Methodology

- In-house APU simulator that extends Gem5
  - Similar to AMD Graphics Core Next architecture
- Benchmarks: Rodinia, AMD APP SDK, OpenDwarfs

GPU cache configuration	
CUs	8, 1GHz, 64 scalar units by 4 SIMDs
L1 cache	8-way, 16KB, 1-cycle tag, 4-cycle data access
Shared L2 cache	16-way, 256KB, 4-cycle tag, 16-cycle data access
Shared L3 cache	16-way, 4MB, 15-cycle tag, 30-cycle data access

Storage Overhead

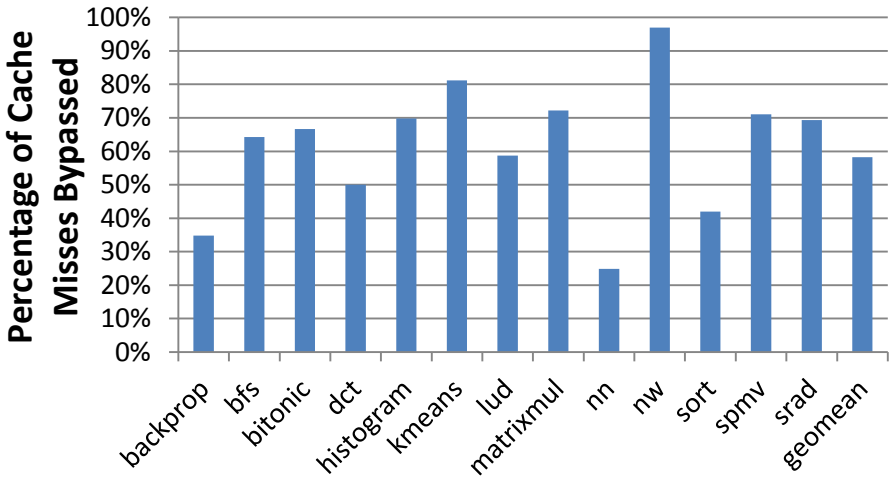
- Evaluated Techniques

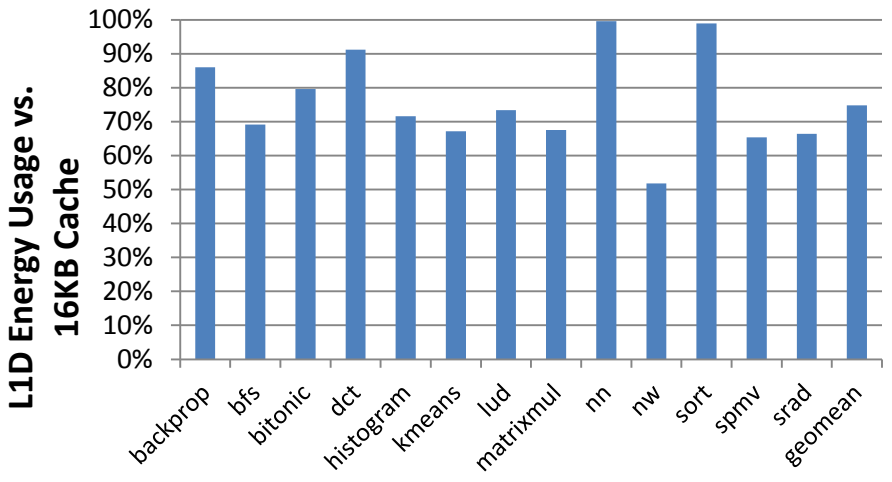
PC-based bypass predictor	
Prediction table/L1 cache	4-bit * 128 entries
Metadata/L1 block	7 bits
Metadata/L2 block	1 bit
Total cost	224 bytes out of 16KB L1 (1.5%) 0.5KB out of 256KB L2 (0.2%)
Counter-based bypass predictor	
Prediction table/L1 cache	128* 128 entries, 5-bits
Metadata/L1 block	20 bits (hashedPC, counters, prediction)
Total cost	10.625KB out of 16KB L1 (66.4%)

Outline

- Background and Motivation
- Adaptive GPU Cache bypassing
- Methodology
- Evaluation Results
- Case Study of Programmability
- Conclusion

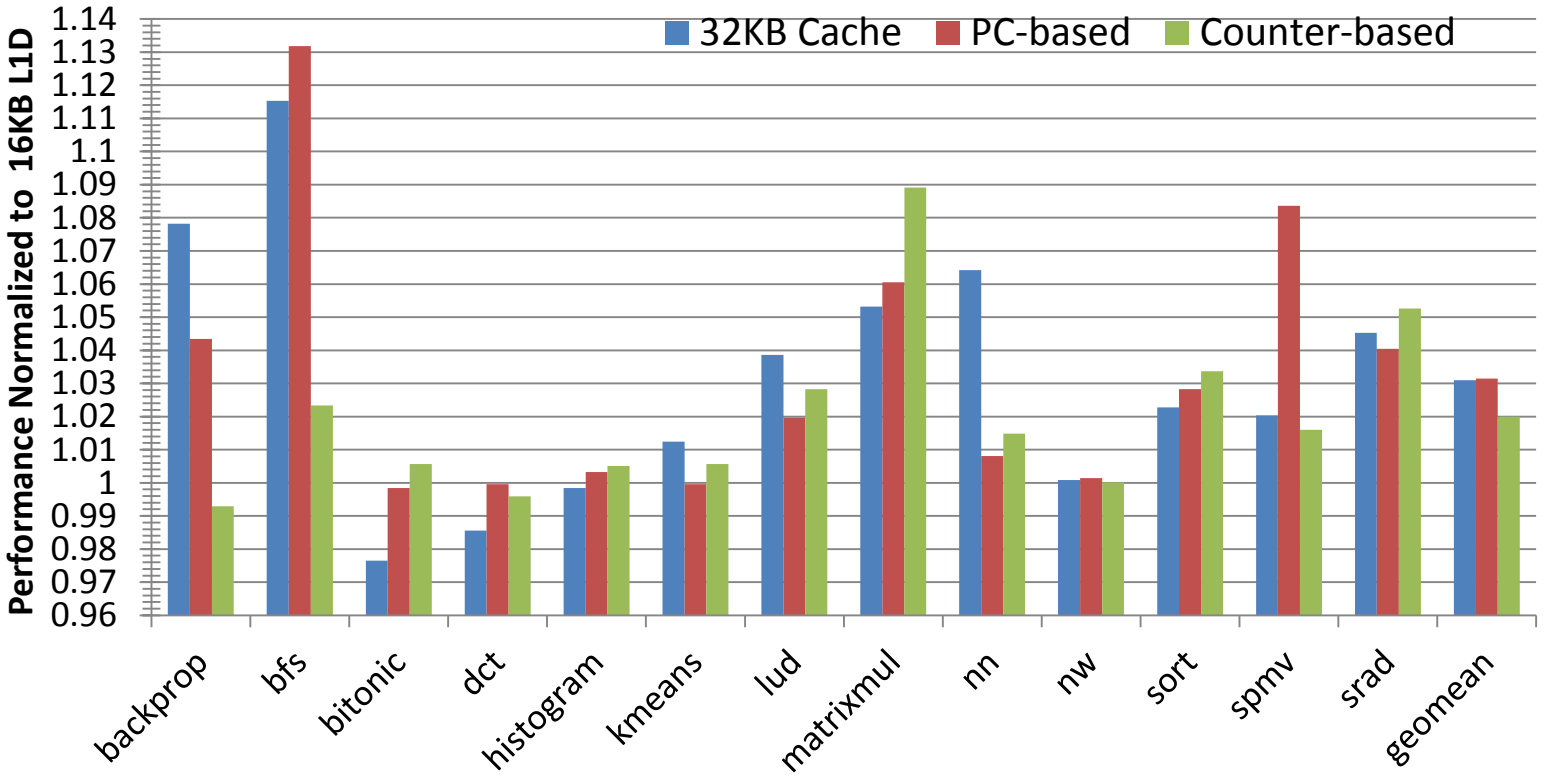
Energy Savings



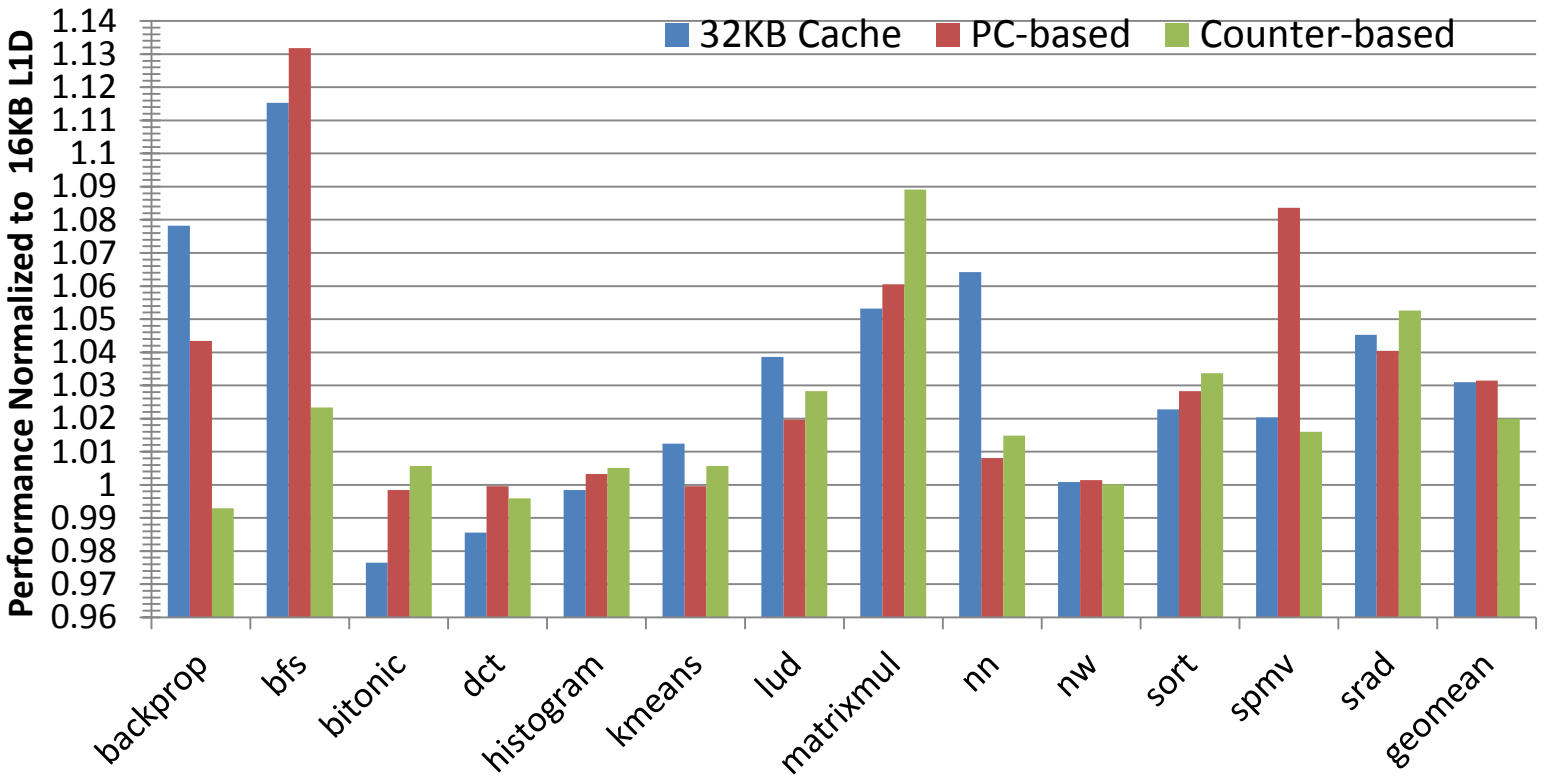


- 58% of cache fills are prevented with cache bypassing
- The energy cost of L1 cache is reduced by up to 49%, on average by 25%
- Reduces dynamic power by 18% , increases the leakage power by only 2.5%

Performance Improvement



Performance Improvement

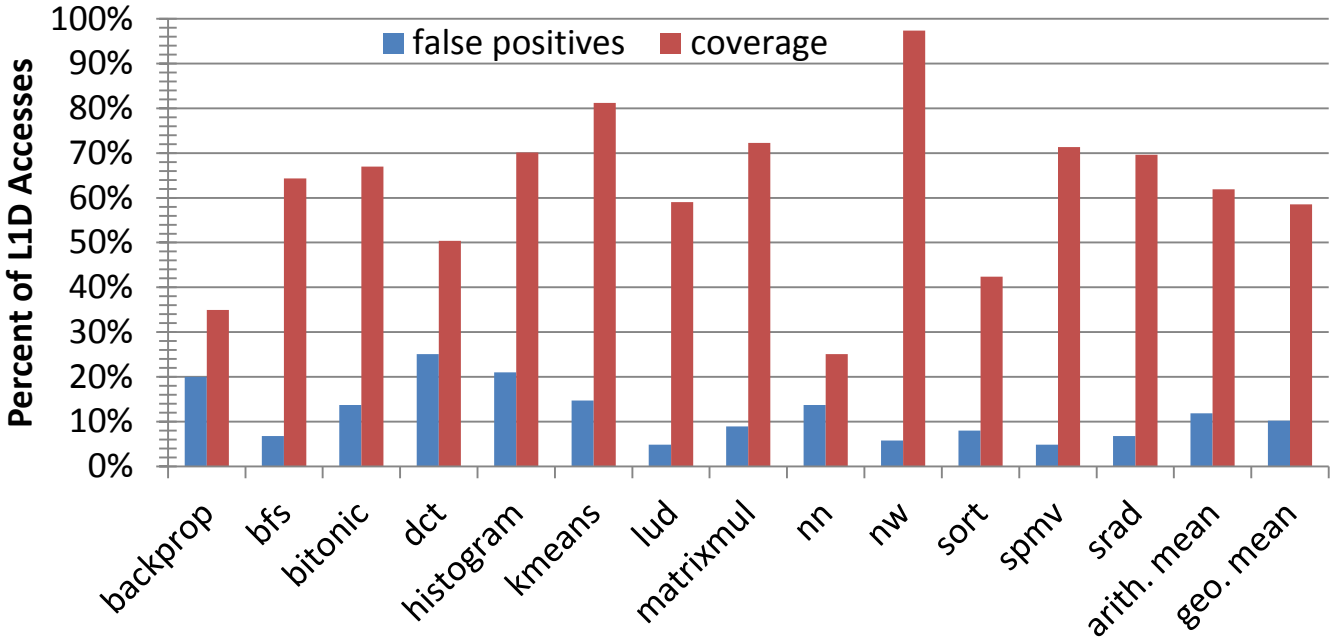


Outperforms counter-based predictor

Achieves performance of 32KB upper-bound

Prediction Accuracy

- False positives: incorrectly bypassed to-be-reused blocks
- Coverage: the ratio of bypass prediction to all prediction



Coverage ratio is 58.6%, false positive rate is 12%

Outline

- Background and Motivation
- Adaptive GPU Cache bypassing
- Methodology
- Evaluation Results
- Case Study of Programmability
- Conclusion

Scratchpad Memories vs.Caches

- Store reused data shared within a compute unit
- Programmer-managed vs. hardware-controlled
- Scratchpad memories filter out temporal locality
- Limited programmability
  - Explicitly remapping from memory address space to the scratchpad address space

AMD

Scratchpad Memories vs.Caches

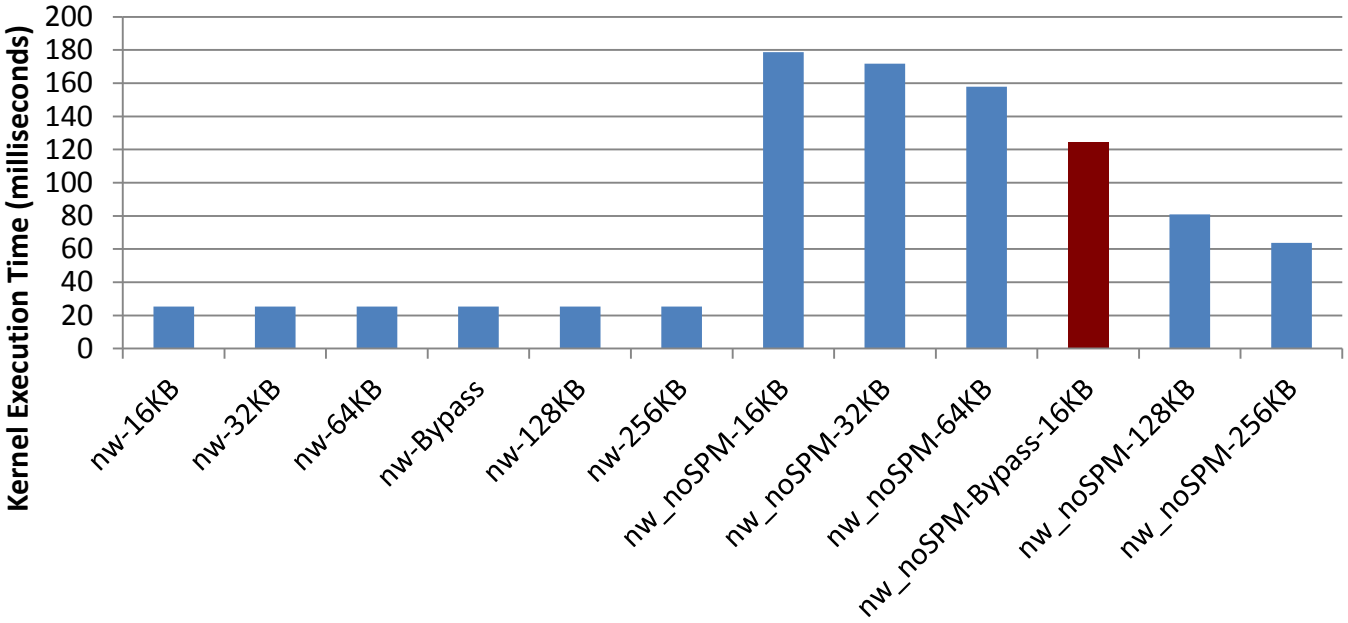
- Store reused data shared within a compute unit
- Programmer-managed vs. hardware-controlled
- Scratchpad memories filter out temporal locality
- Limited programmability
  - Explicitly remapping from memory address space to the scratchpad address space

To what extent can L1 cache bypassing make up for the performance loss caused by removing scratchpad memories?

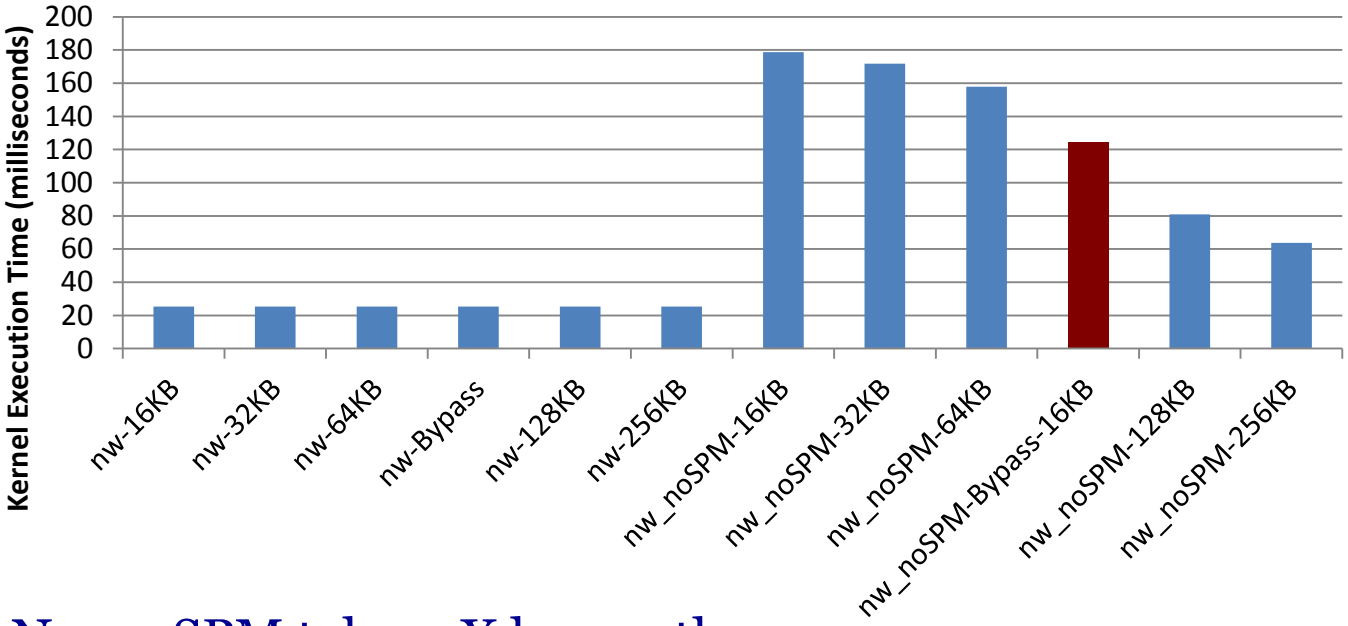
Case Study: Needleman-Wunsch (nw)

- nw:
  - Global optimization algorithm
  - Compute-sensitive
  - Very little reuse observed in L1 caches
- Rewrote nw to remove the use of scratchpad memories: nw-noSPM
  - Not simply replace *_local_* functions with *_global_* functions
  - Best-effort re-written version

Case Study (cont.)

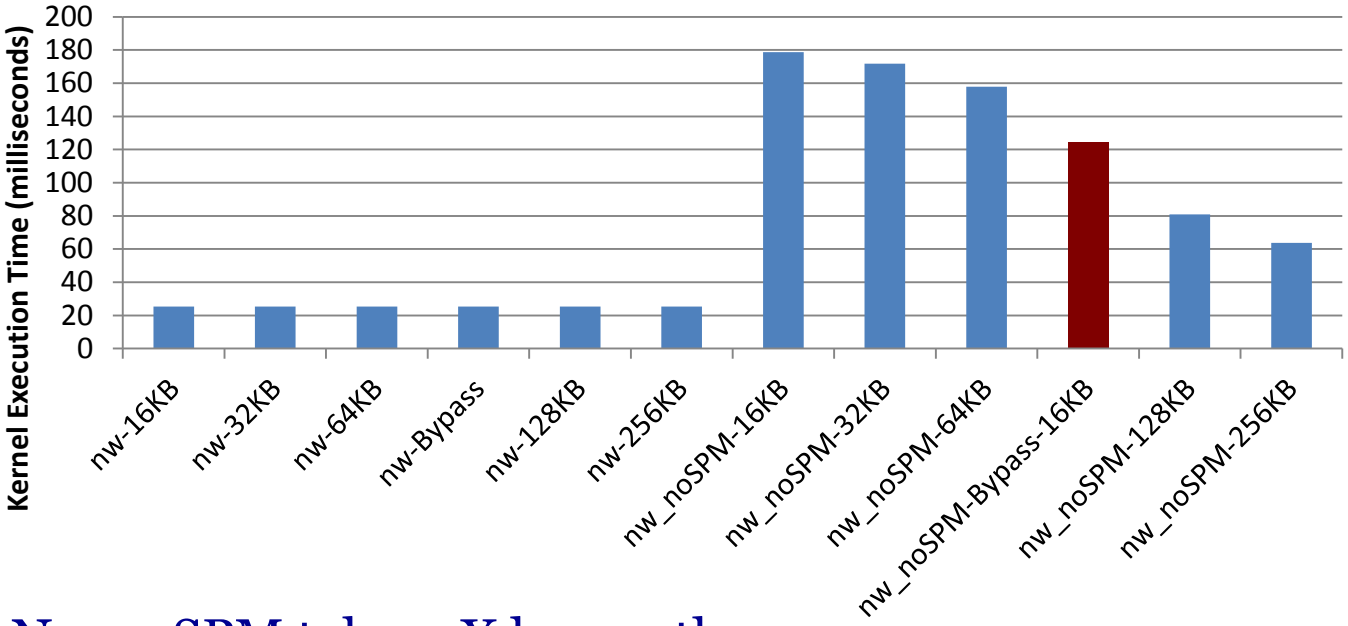


Case Study (cont.)



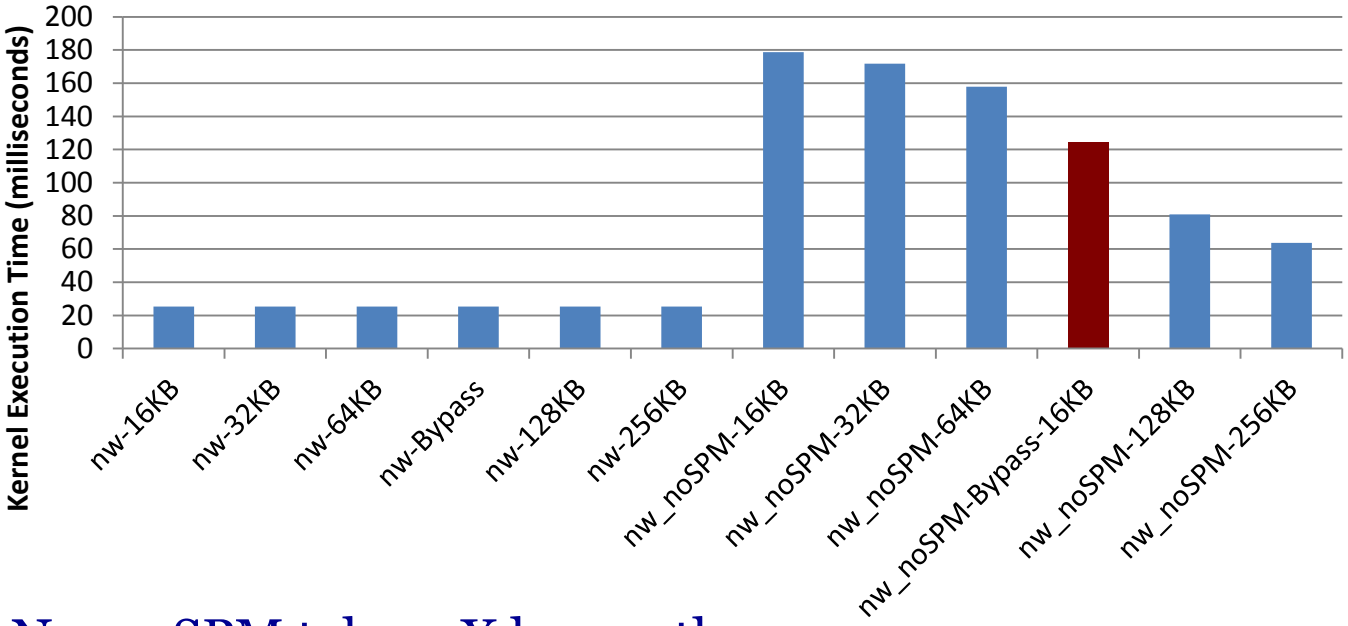
- Nw-noSPM takes 7X longer than nw

Case Study (cont.)



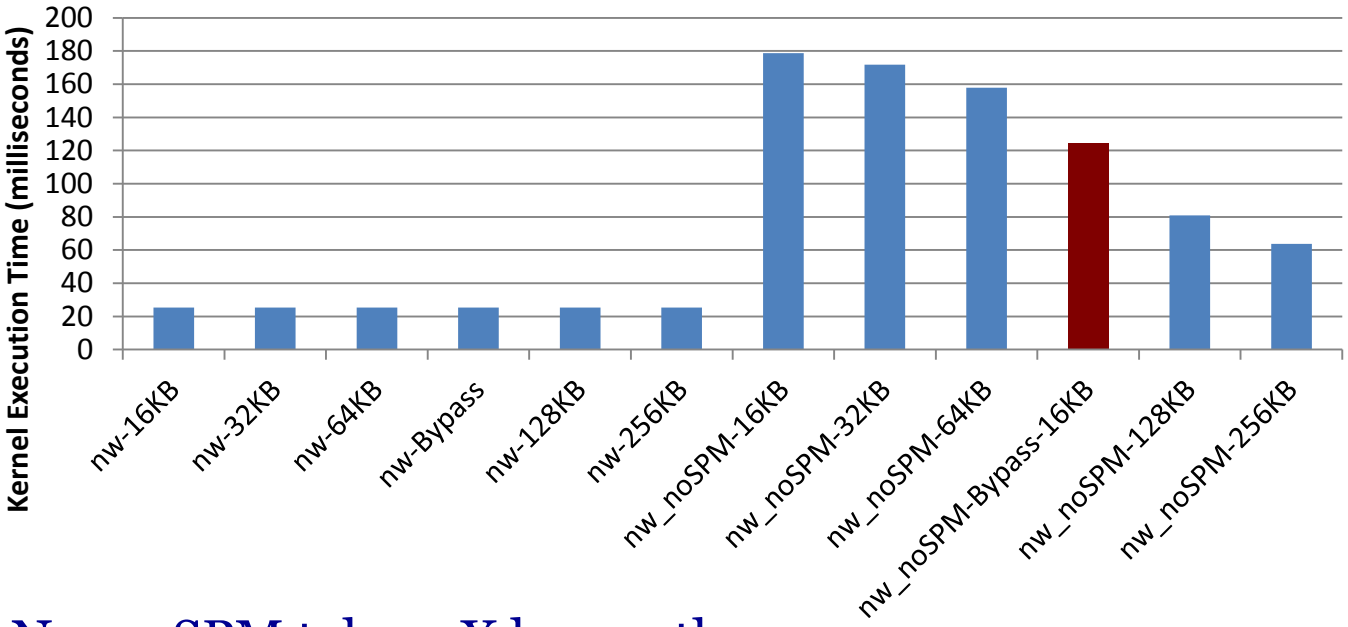
- Nw-noSPM takes 7X longer than nw
- With bypassing, the gap is reduced by 30%

Case Study (cont.)



- Nw-noSPM takes 7X longer than nw
- With bypassing, the gap is reduced by 30%
- 16KB L1 cache + bypassing outperforms 64KB L1 cache

Case Study (cont.)



- Nw-noSPM takes 7X longer than nw
- With bypassing, the gap is reduced by 30%
- 16KB L1 cache + bypassing outperforms 64KB L1 cache
- Note that scratchpad memory is 64KB while L1 is only 16KB

AMD

Conclusion

- GPU caches are inefficient
- We propose a simple but effective GPU cache bypassing technique
- Improves GPU cache efficiency
- Reduces energy overhead
- Requires far less storage overhead
- Bypassing allows us to move towards more programmable GPUs

COMPUTER SCIENCE & ENGINEERING
TEXAS A&M UNIVERSITY

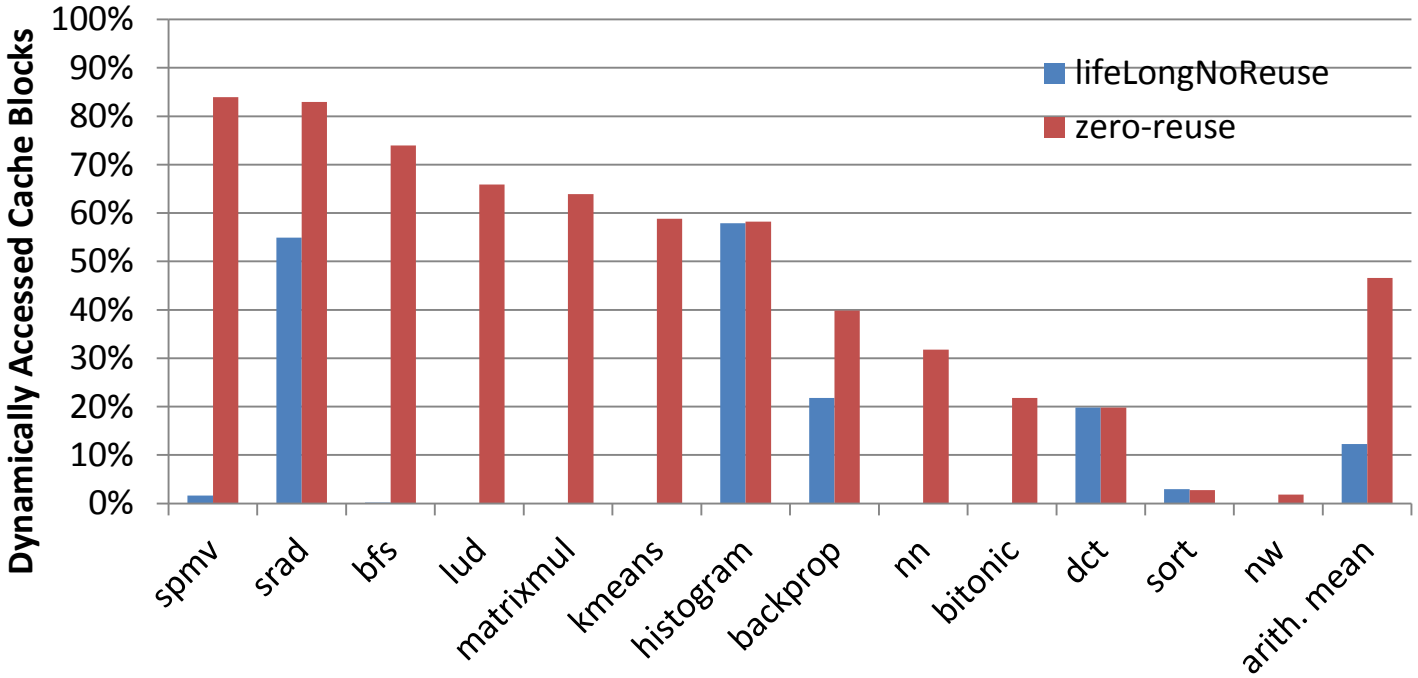
AMD

Thank you!

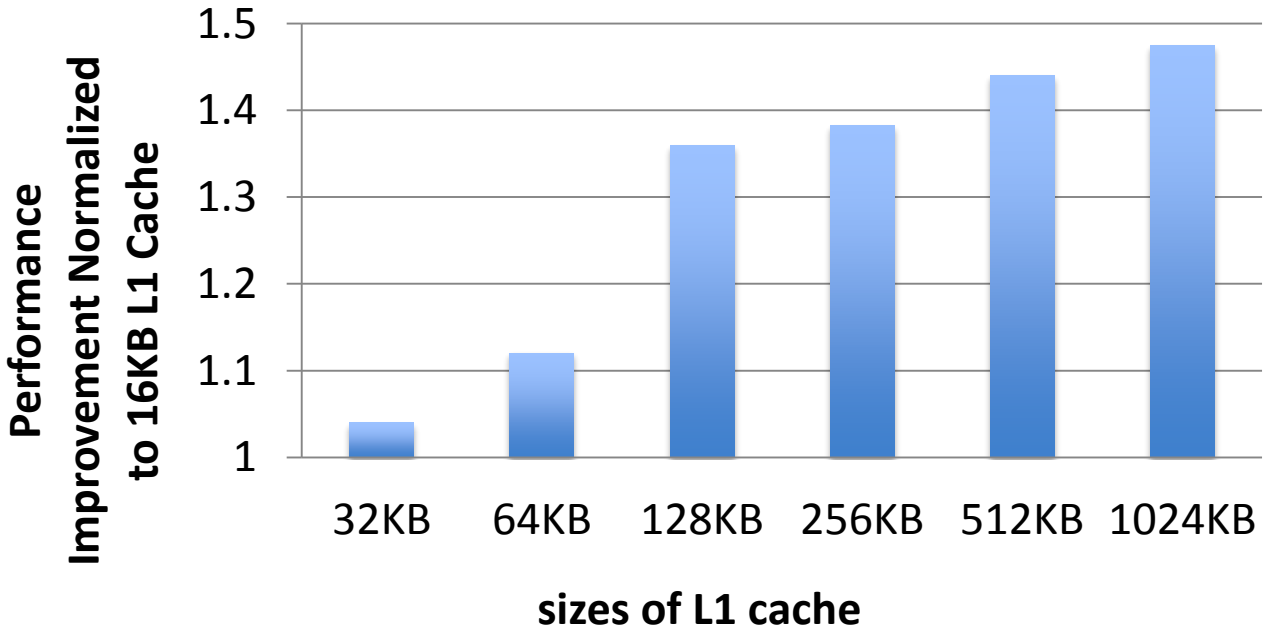
Questions?

Backup

- Do all zero-reuse actually streaming?



GPU Cache Capacity



- Caches are useful, but current cache size are too small to gain benefit
- Performance benefit by increasing cache sizes << extra area taken up
- Adding more computational resources
- Leading to lower per-thread cache capacity

Power Overhead

Energy (nJ)	16KB baseline	bypassing
per tag access	0.00134096	0.0017867
per data access	0.106434	0.106434
per prediction table access	N/A	0.000126232
Dynamic Power (mW)	44.2935	36.1491
Static Power (mW)	7.538627	7.72904

Compare with SDBP

- SDBP is designed for LLCs, where much of the temporal locality has been filtered
- our technique is designed for GPU L1 caches, where temporal information is complete. We use PCs because of the observation of characteristics of GPGPU memory accesses.
- GPU kernels are small and frequently launched, the interleaving changes frequently.