Vector Runahead

Ajeya Naithani (Ghent)
Sam Ainsworth (Edinburgh)
Timothy M. Jones (Cambridge)
Lieven Eeckhout (Ghent)

THE UNIVERSITY *of* EDINBURGH

UNIVERSITY OF CAMBRIDGE

Vector Runahead Summary

1. Targets indirect memory accesses
   -- Complex address calculation

2. Purely microarchitectural and speculative

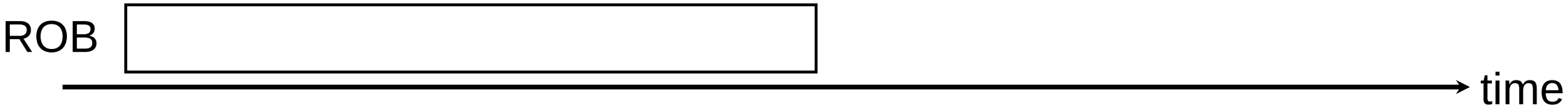
3. Executes multiple loop iterations as a single vector instruction
   -- Workload does not have to be vectorizable

4. Independent of front-end bandwidth

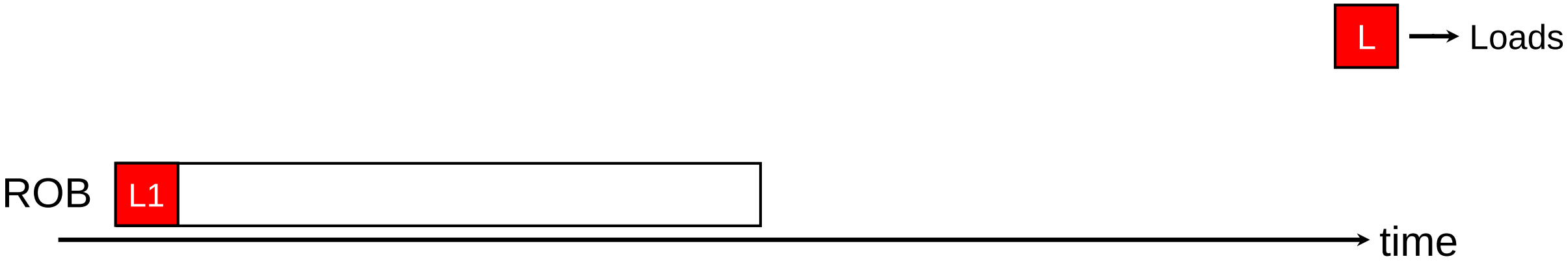
Full-Window Stalls Degrade Performance

Full-Window Stalls Degrade Performance

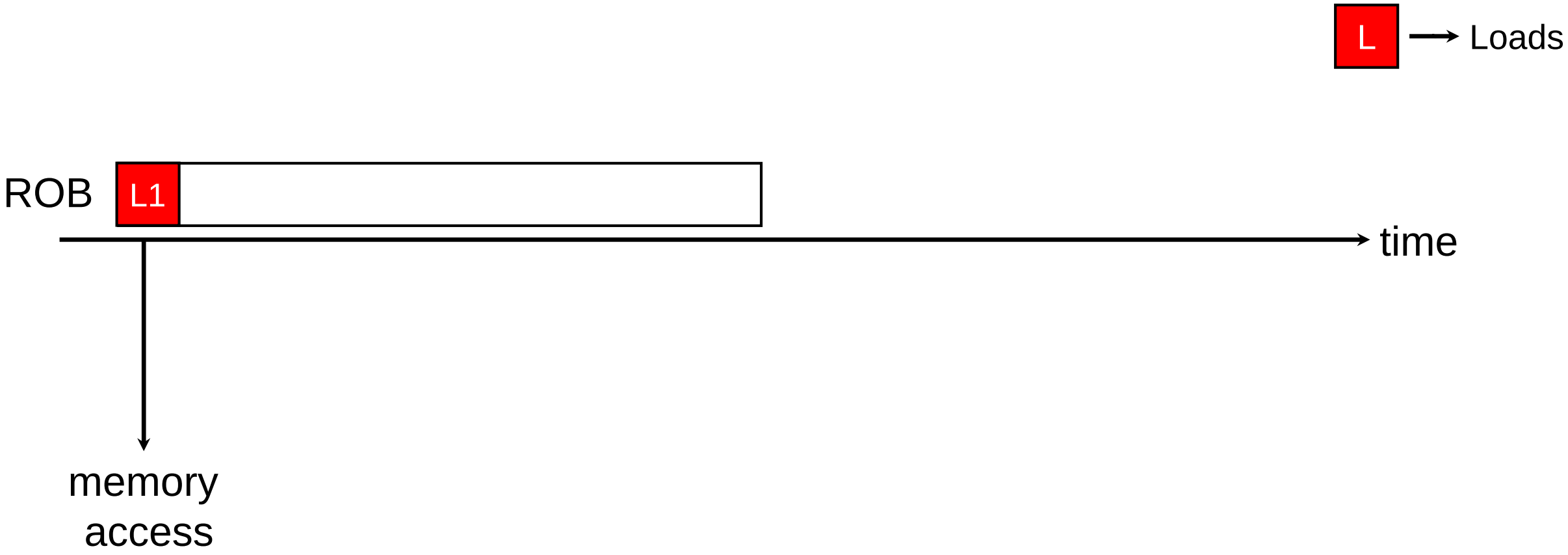
Full-Window Stalls Degrade Performance



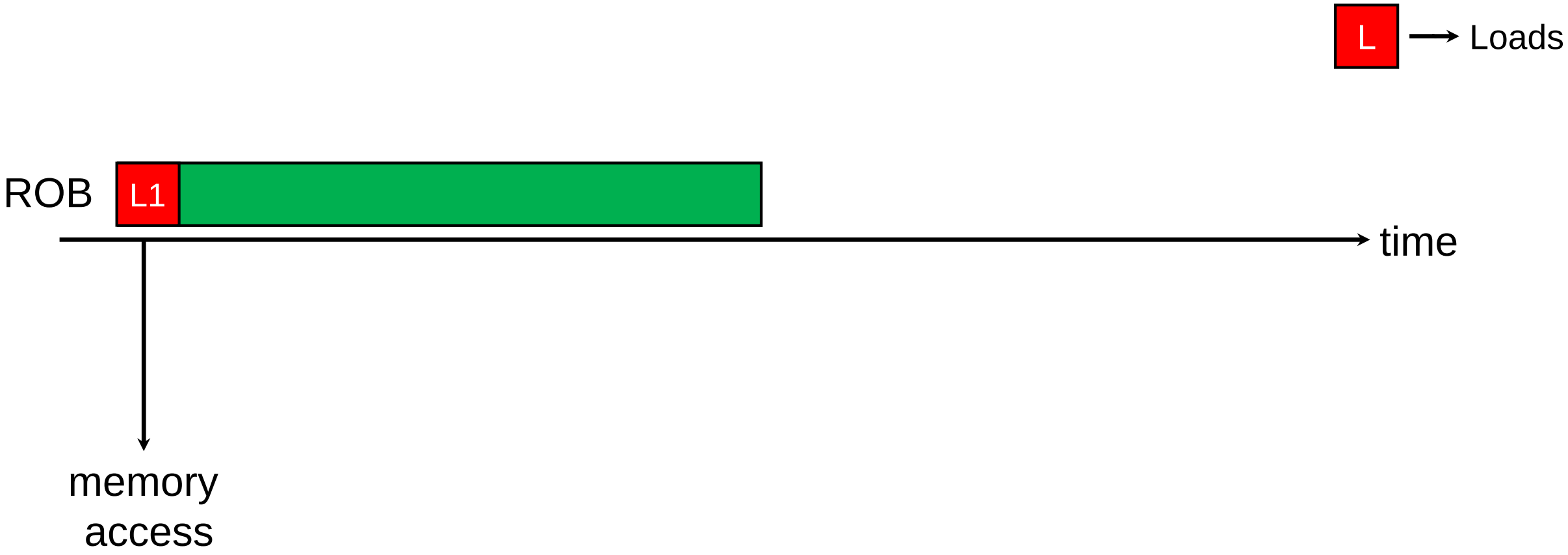
Full-Window Stalls Degrade Performance



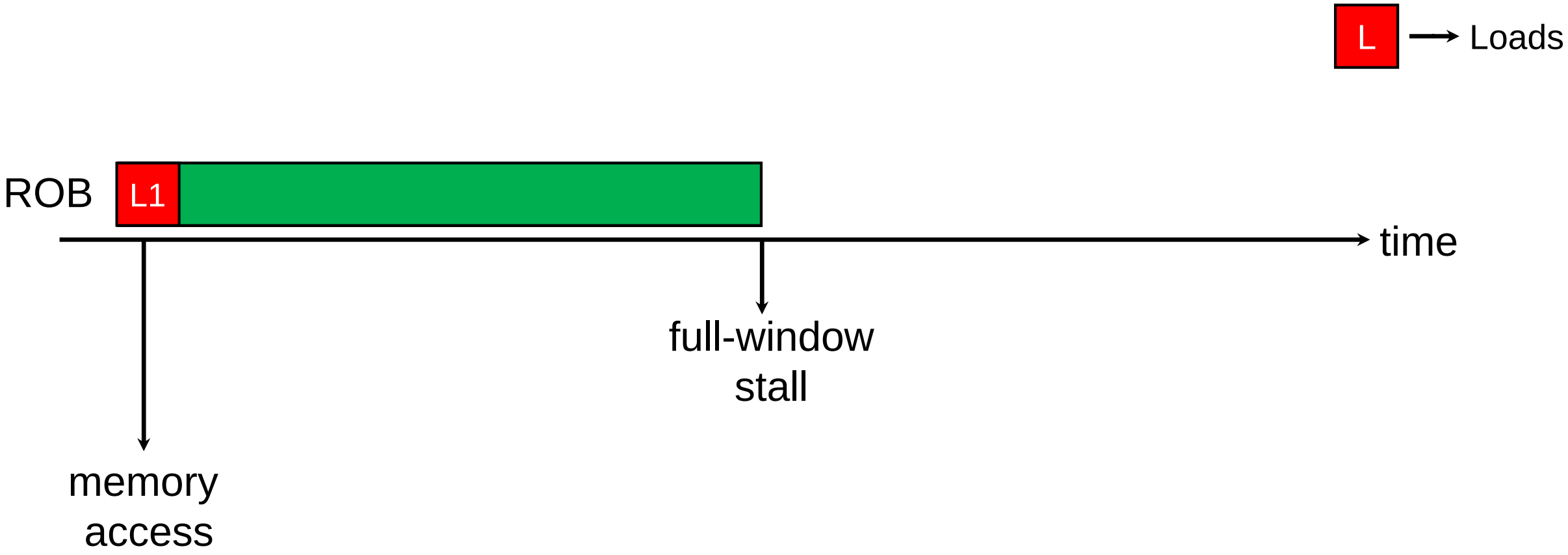
Full-Window Stalls Degrade Performance



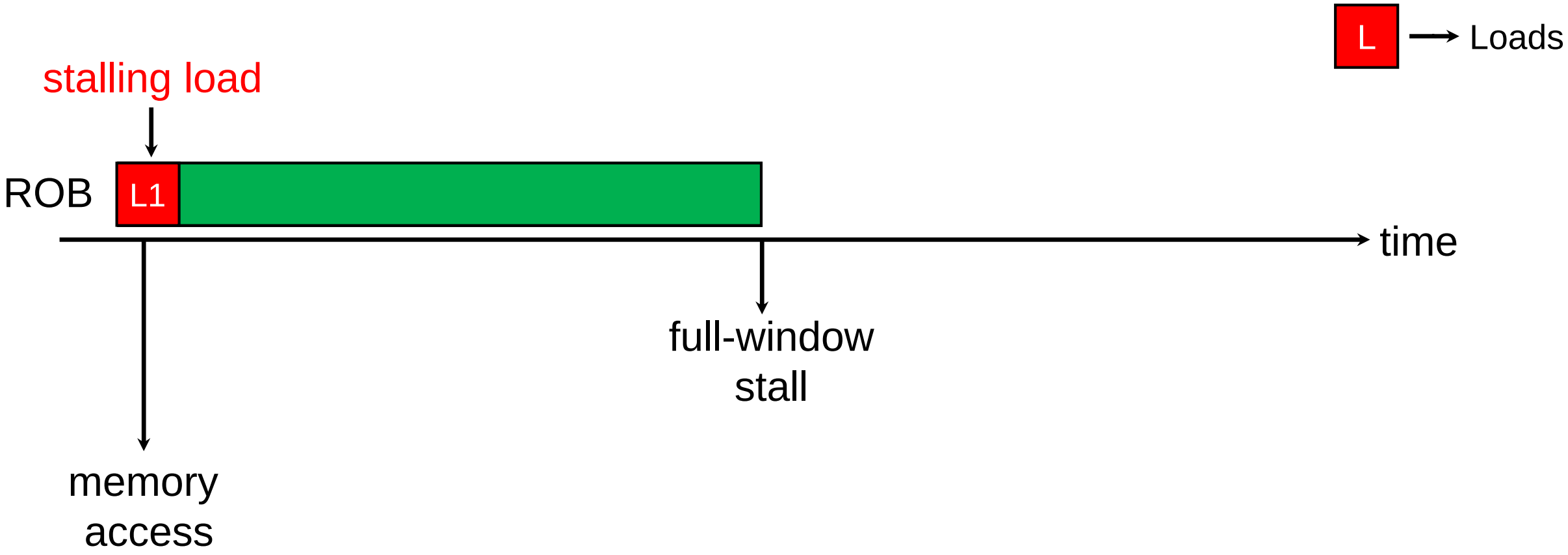
Full-Window Stalls Degrade Performance



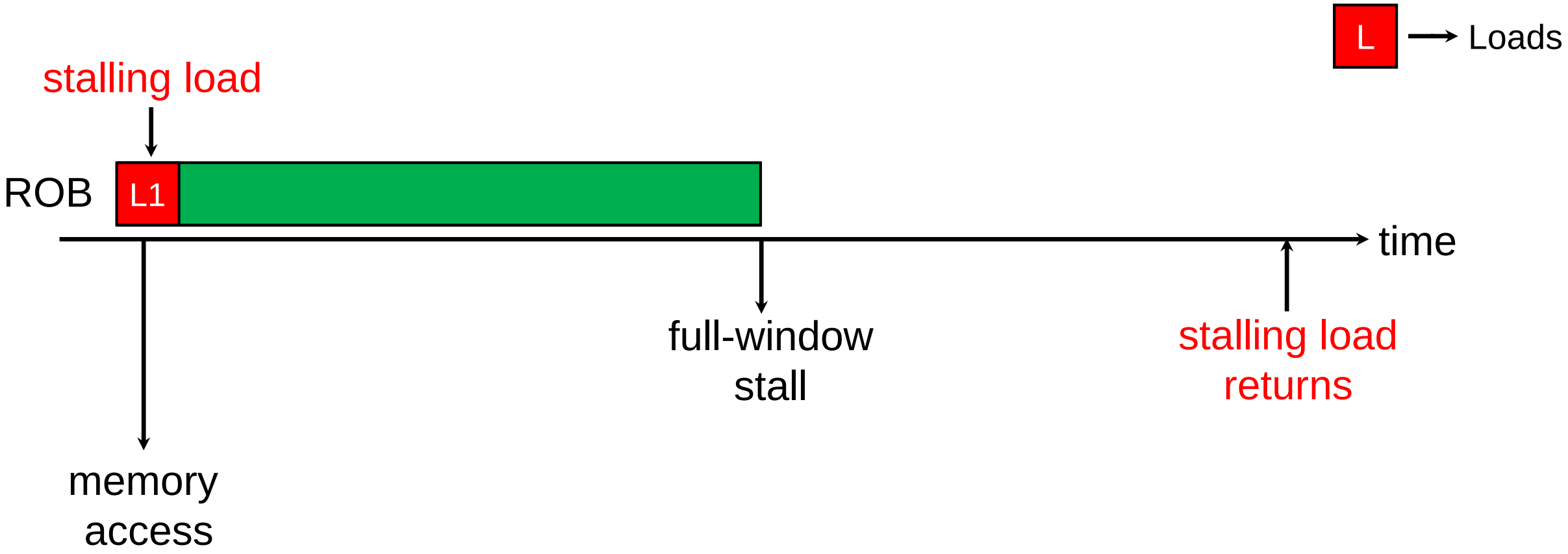
Full-Window Stalls Degrade Performance



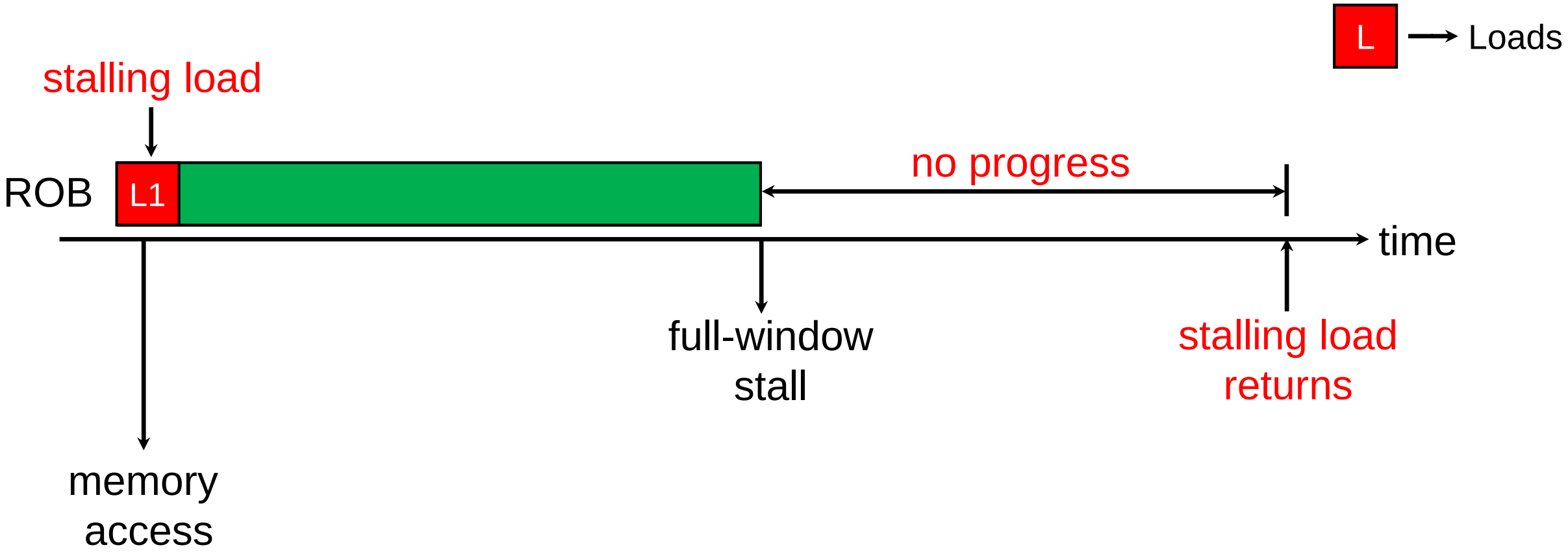
Full-Window Stalls Degrade Performance



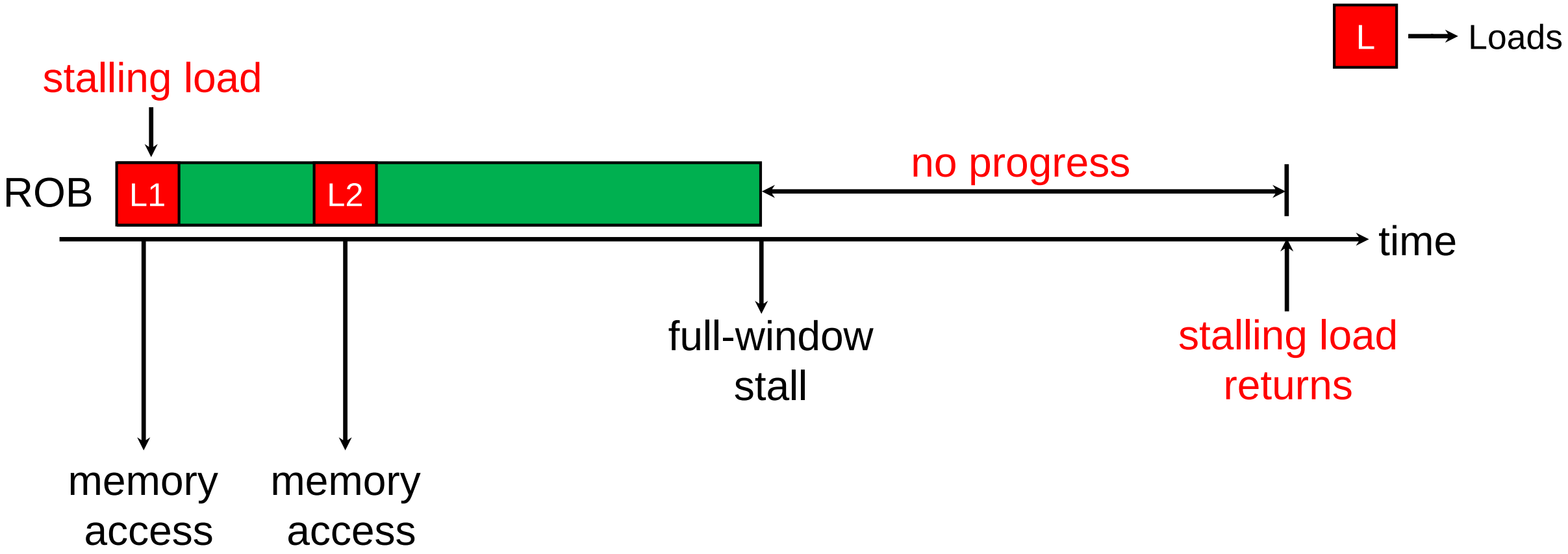
Full-Window Stalls Degrade Performance



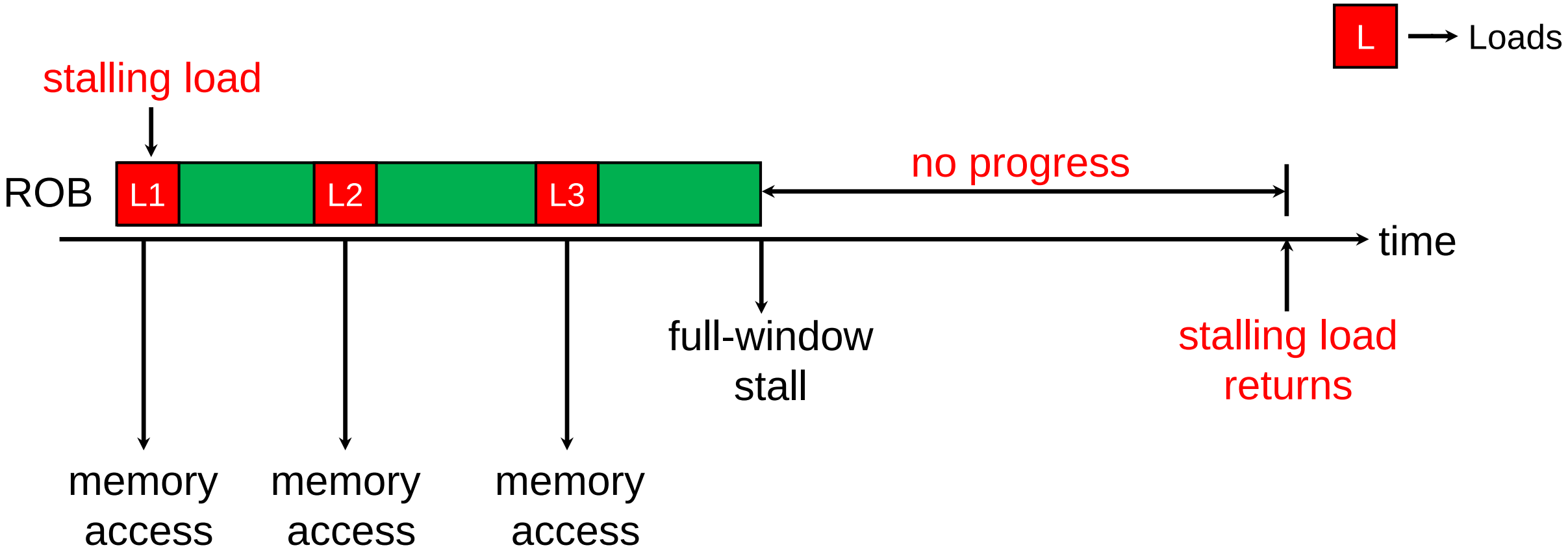
Full-Window Stalls Degrade Performance



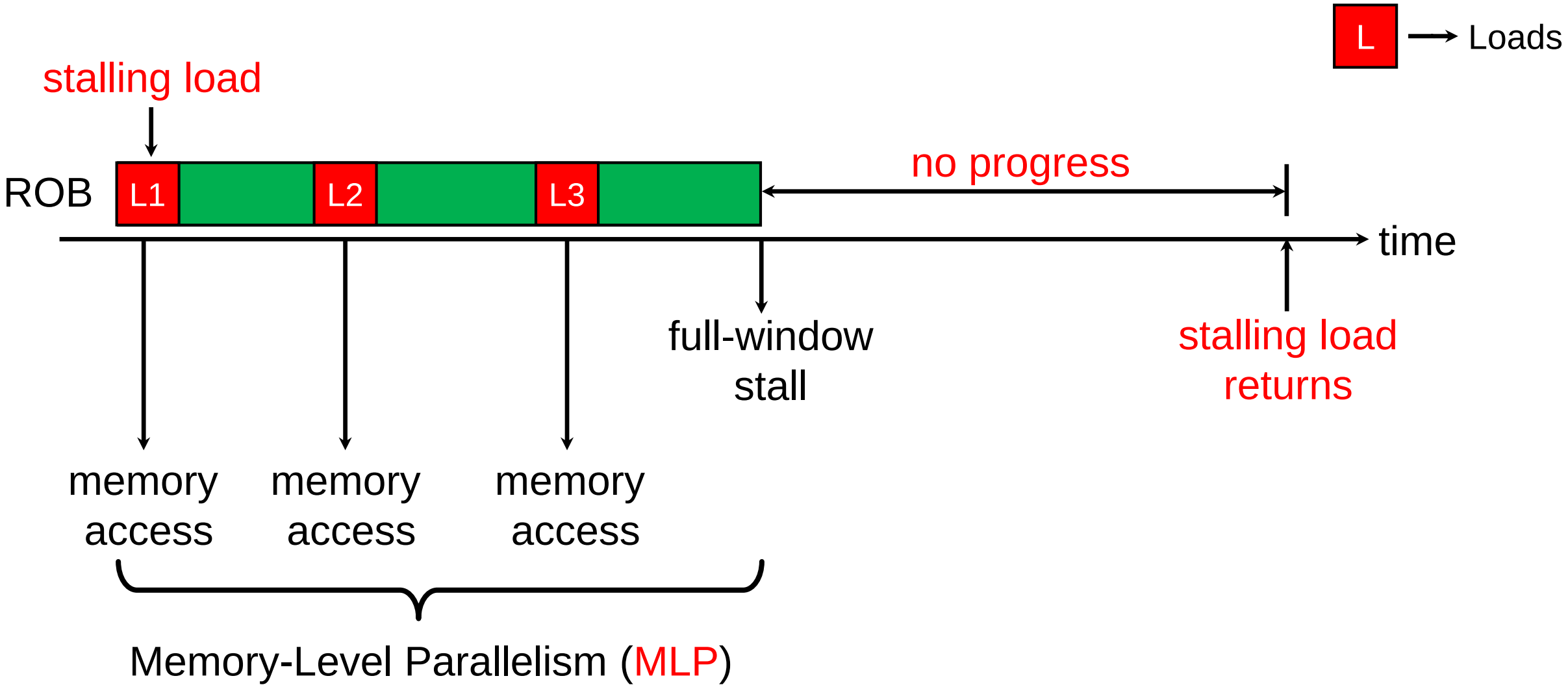
Full-Window Stalls Degrade Performance



Full-Window Stalls Degrade Performance

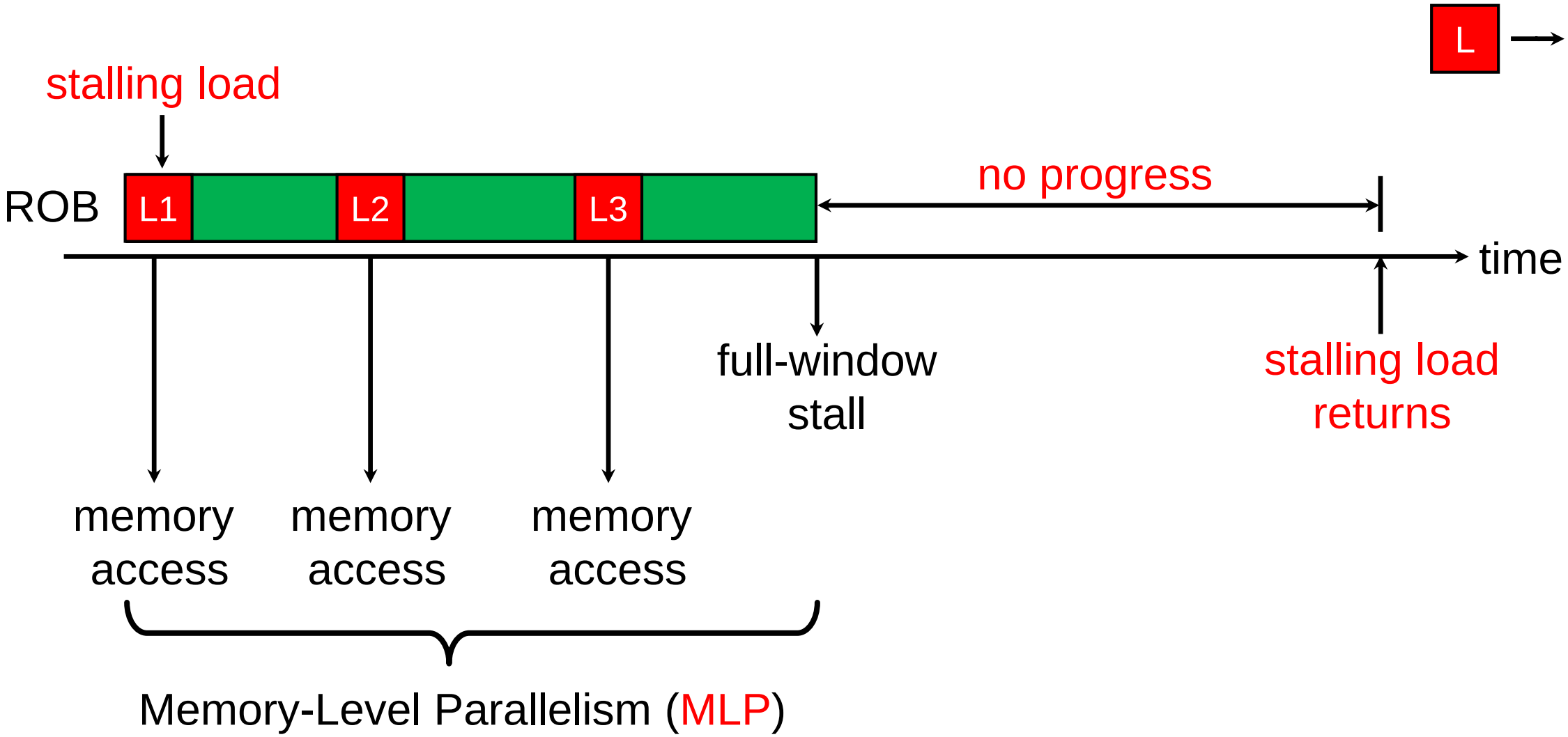


Full-Window Stalls Degrade Performance



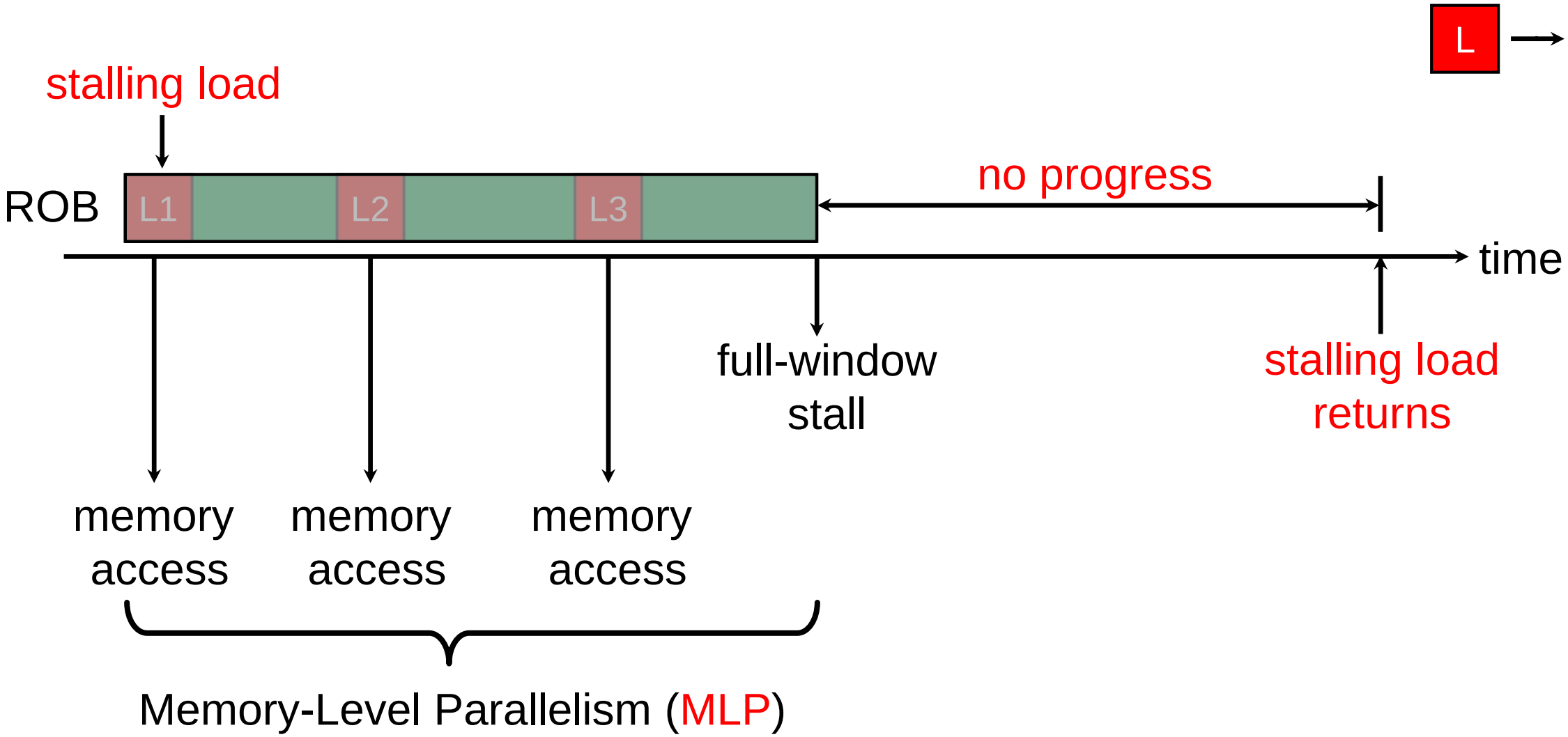
Runahead Execution Prefetches under a Full-Window Stall

ISCA 2005



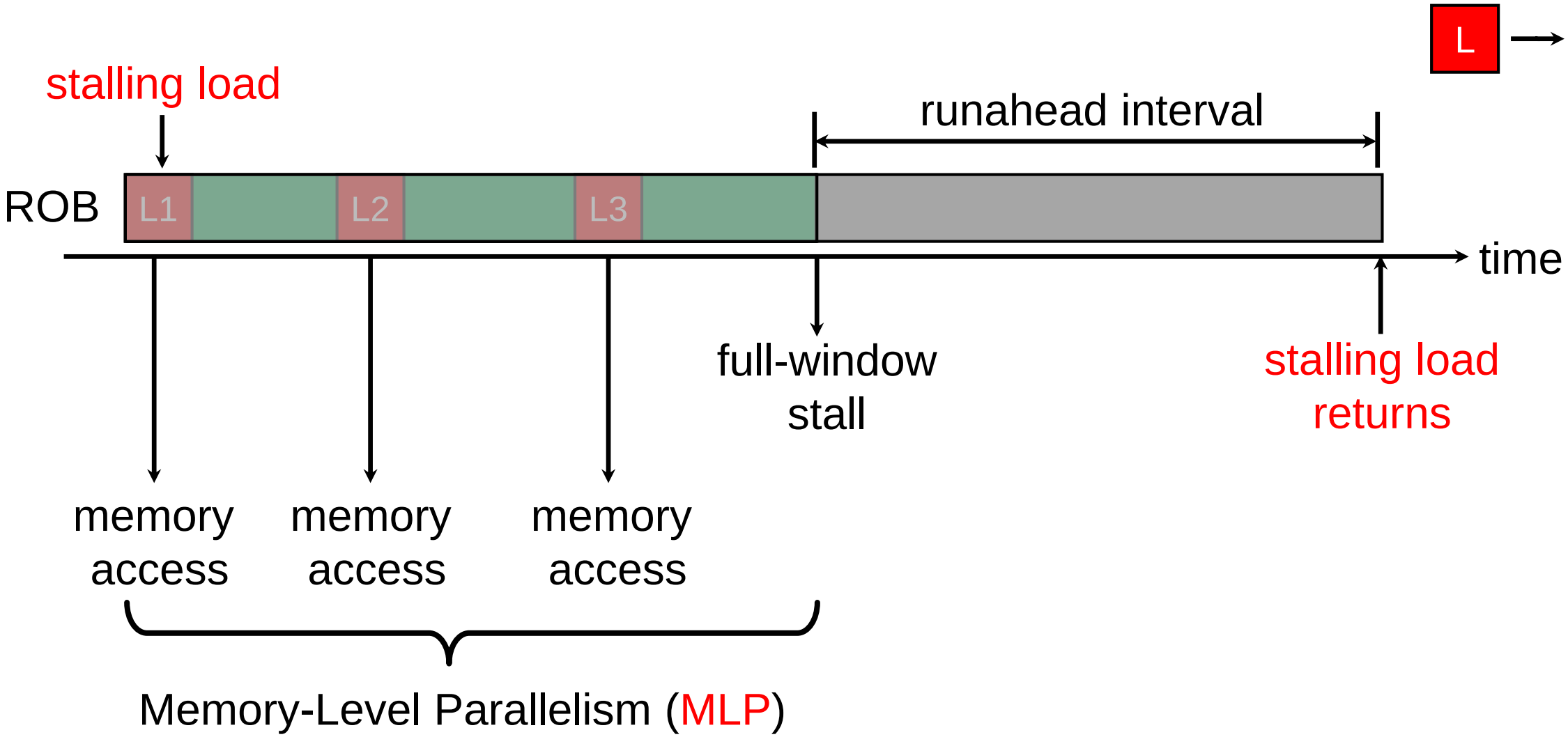
Runahead Execution Prefetches under a Full-Window Stall

ISCA 2005



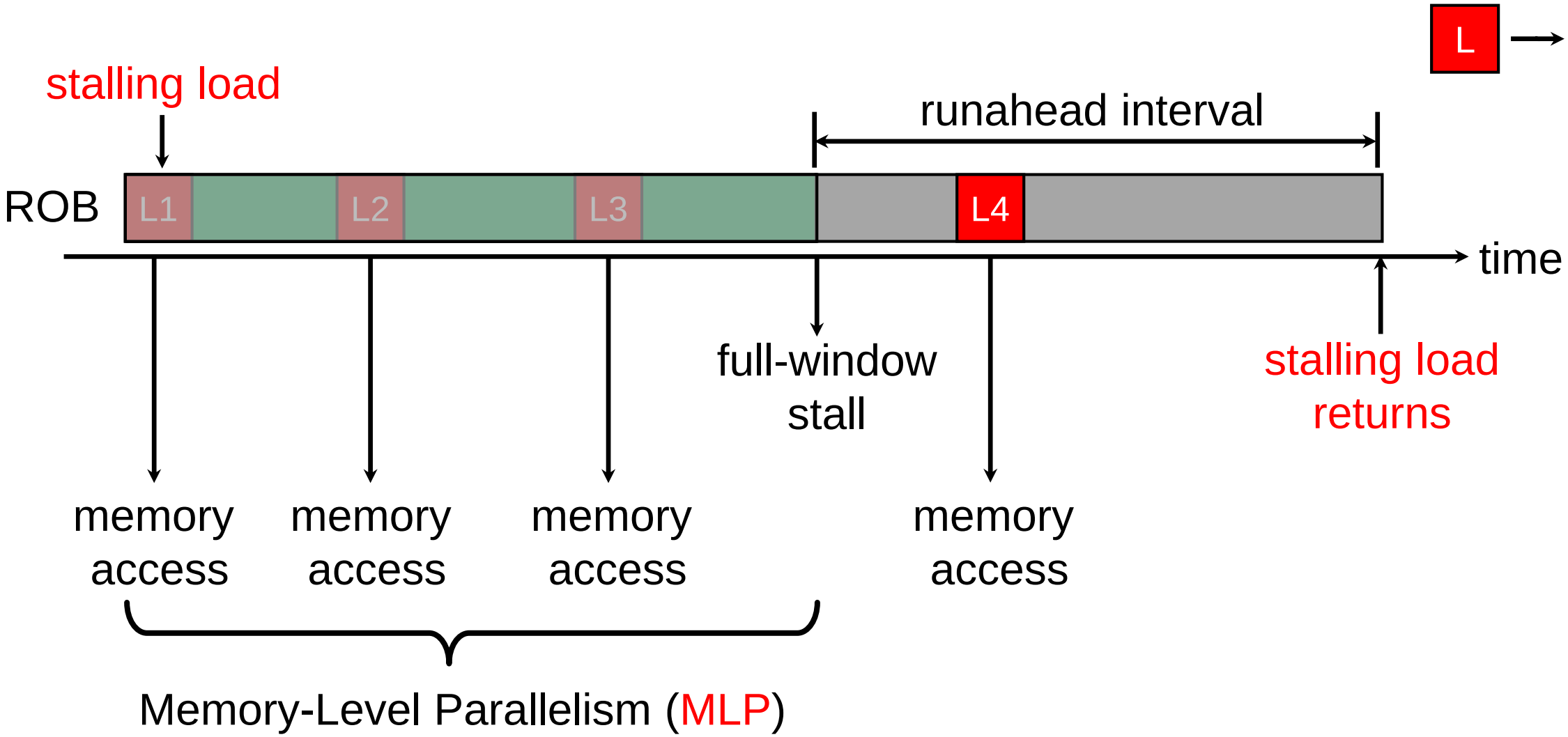
Runahead Execution Prefetches under a Full-Window Stall

ISCA 2005



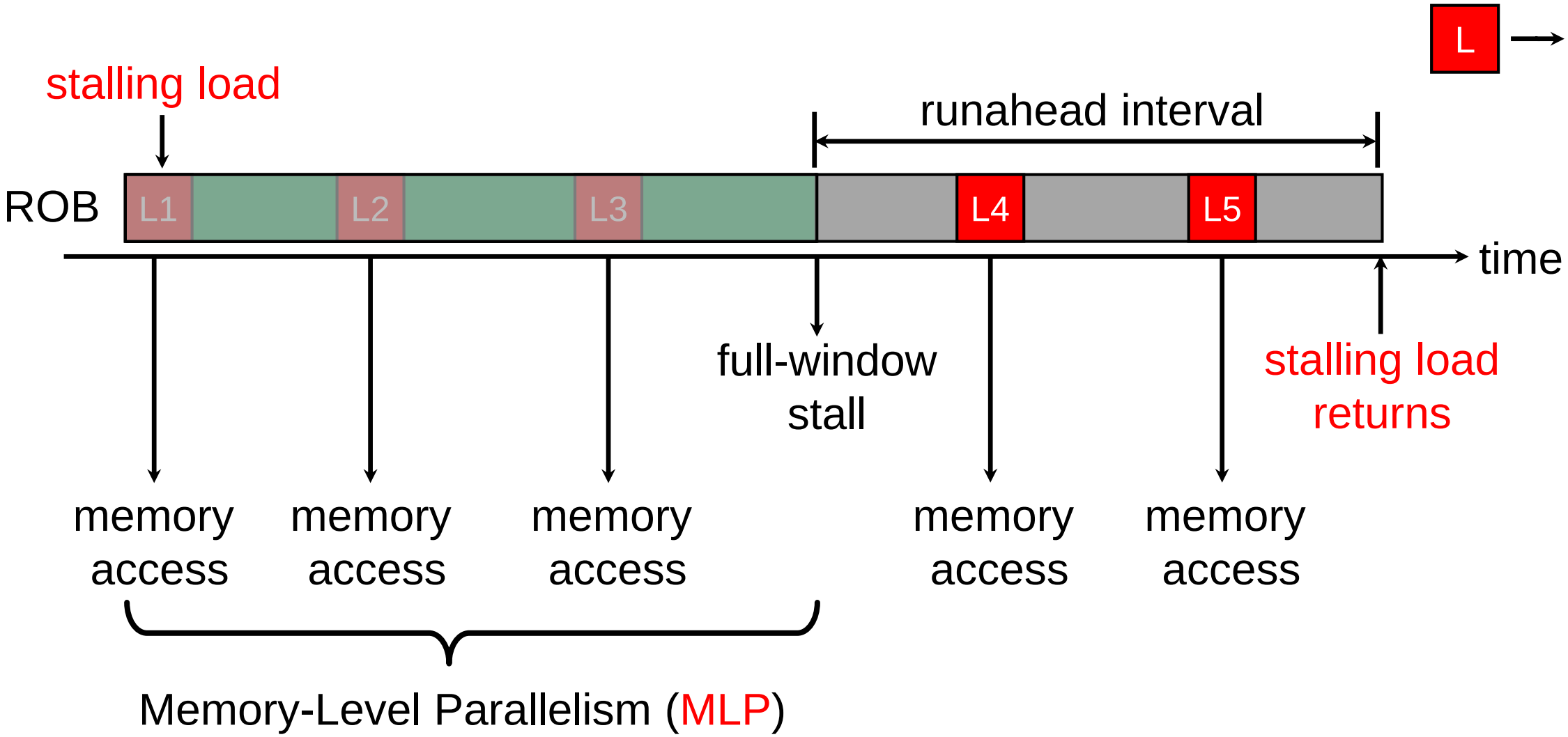
Runahead Execution Prefetches under a Full-Window Stall

ISCA 2005



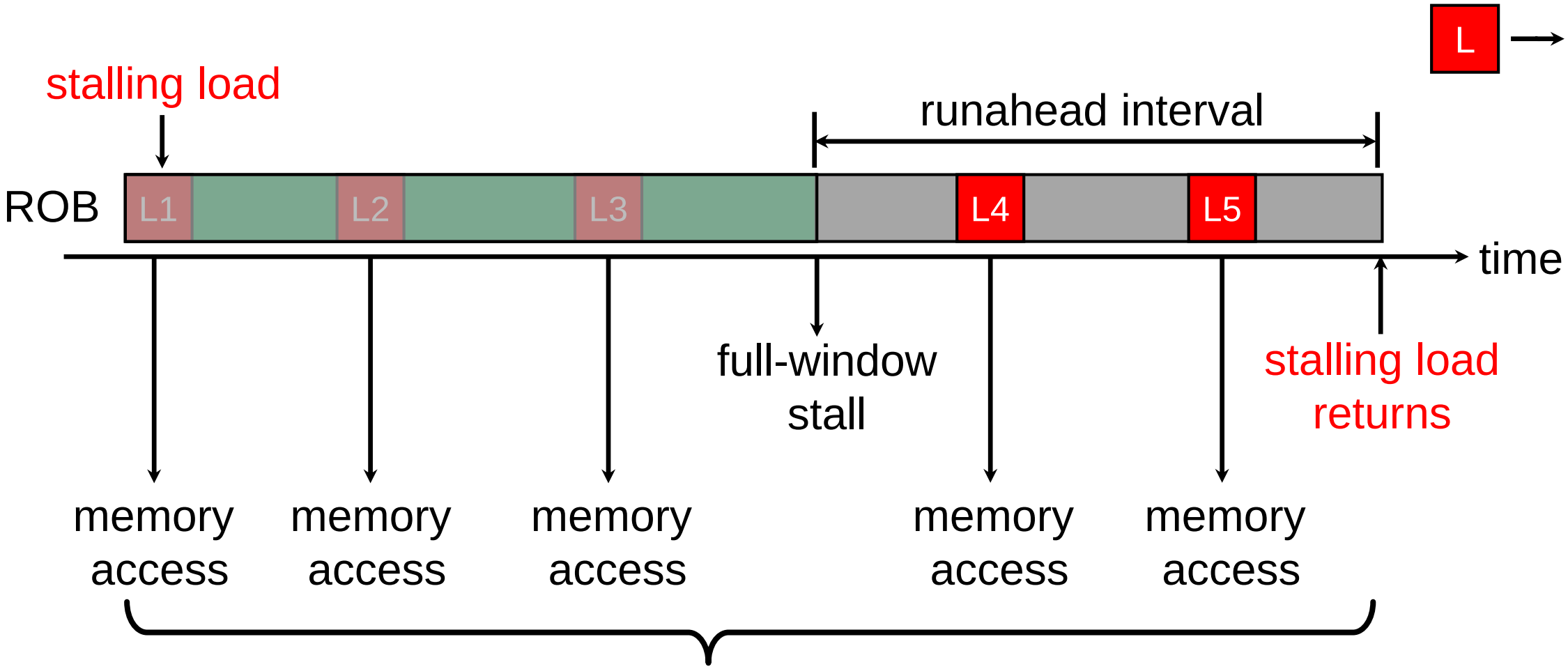
Runahead Execution Prefetches under a Full-Window Stall

ISCA 2005



Runahead Execution Prefetches under a Full-Window Stall

ISCA 2005



Increased Memory-Level Parallelism (MLP)

Precise Runahead Eliminates Needless Work from Runahead

HPCA 2020

Precise Runahead Eliminates Needless Work from Runahead

HPCA 2020

1. Does not flush ROB

2. Executes only useful instructions

3. Runs ahead even for short intervals

Precise Runahead Eliminates Needless Work from Runahead

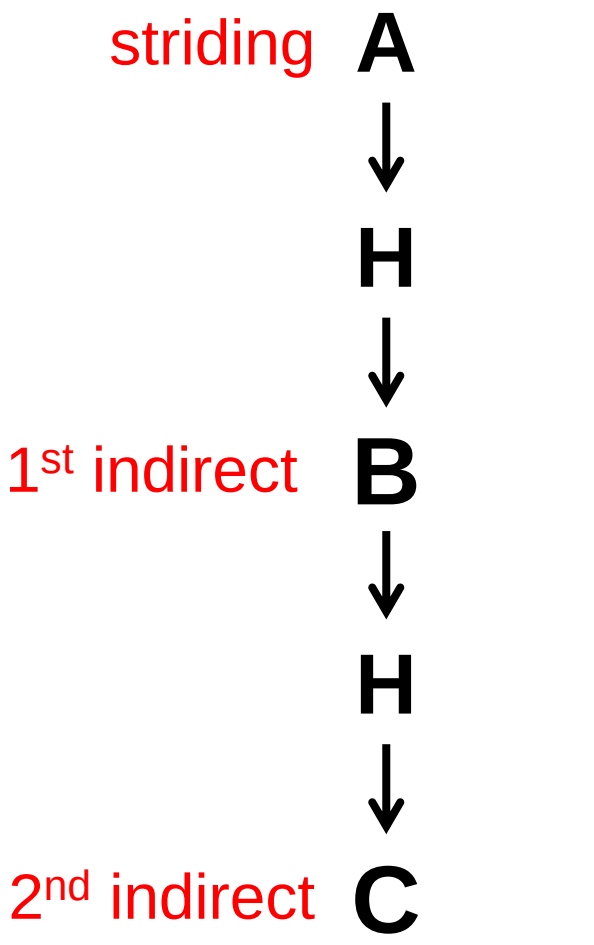
HPCA 2020

1. Does not flush ROB
2. Executes only useful instructions
3. Runs ahead even for short intervals
4. Efficiently manages microarchitectural resources

Runahead for Indirect Memory Accesses

representative example

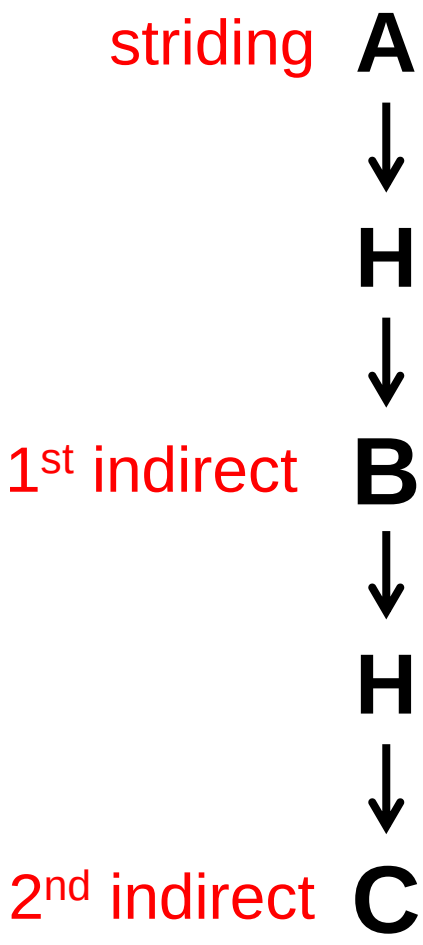
```
for(i = 0; i < NUM_KEYS; i++)
  C[hash(B[hash(A[i])])]++;
```



indirect chain

Runahead for Indirect Memory Accesses

representative example

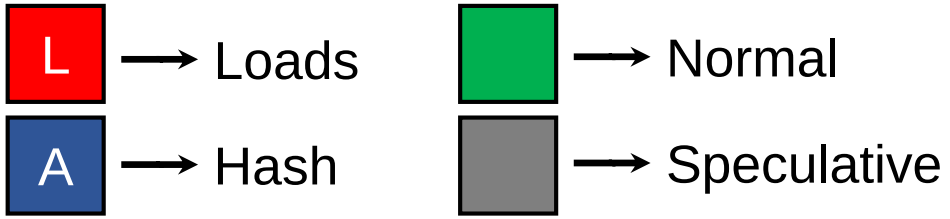



```
for(i = 0; i < NUM_KEYS; i++)
  C[hash(B[hash(A[i])])]++;
```

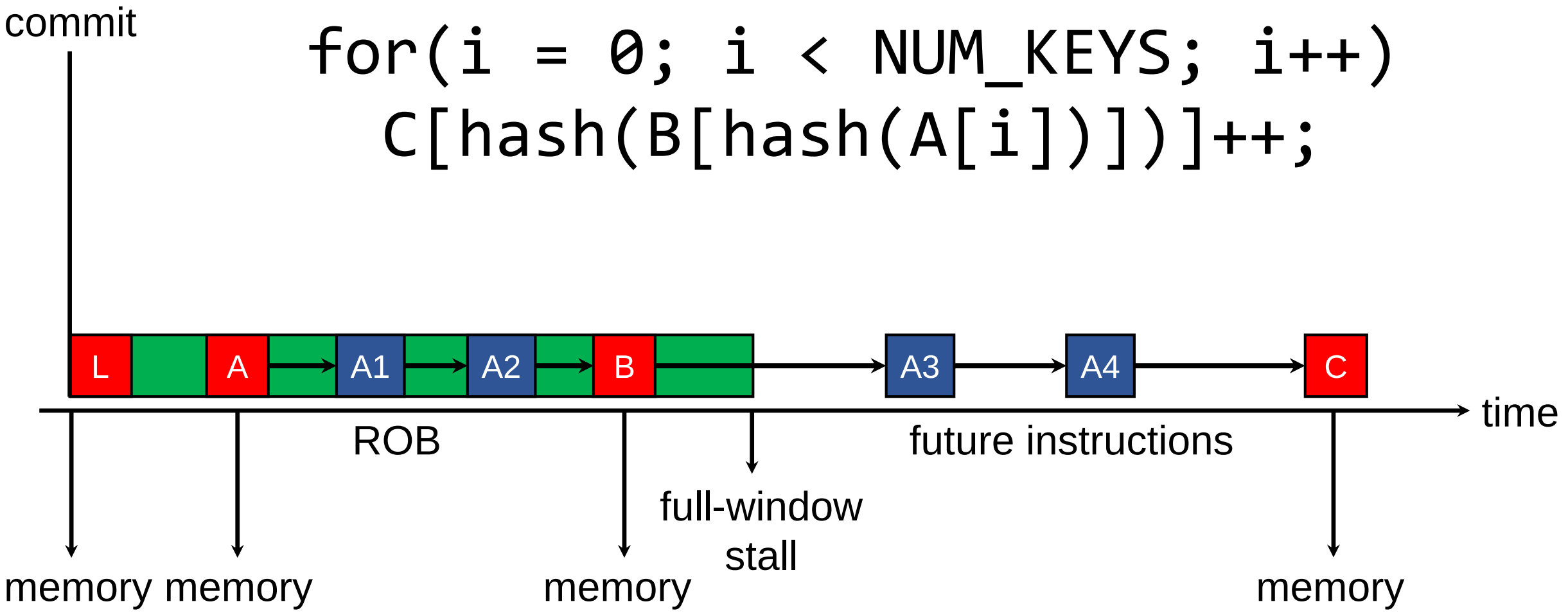
```
unsigned int
hash(unsigned int x) {
  x = ((x >> 16) ^ x) * 0x45d9f3b;
  x = ((x >> 16) ^ x) * 0x45d9f3b;
  x = (x >> 16) ^ x;
return x & (MAX_KEY - 1);
}
```

complex indirect address computation

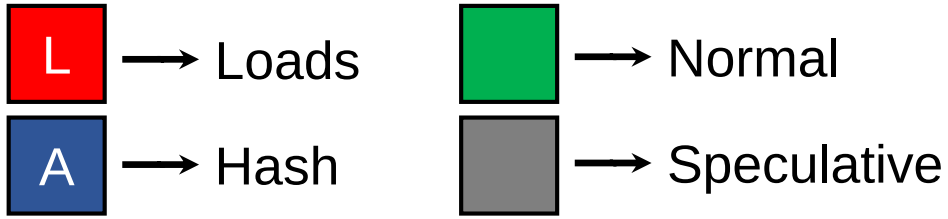
Runahead for Indirect Memory Accesses

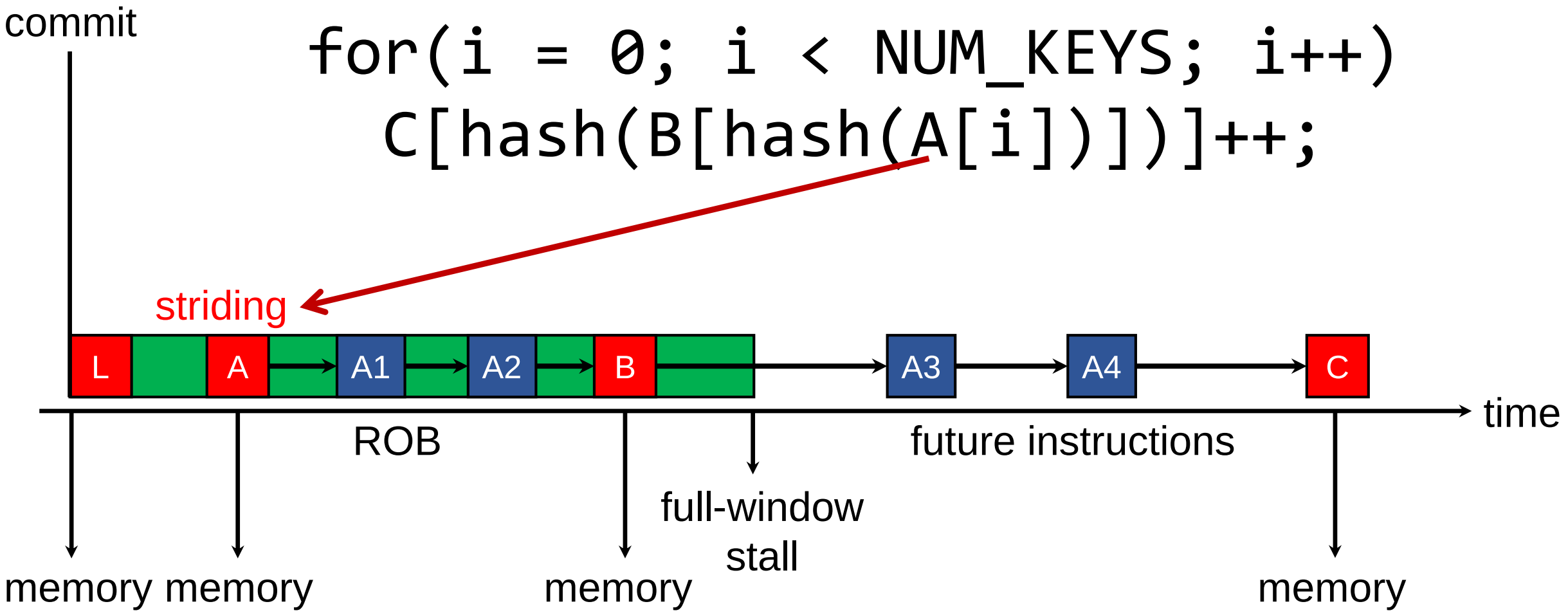



```
for(i = 0; i < NUM_KEYS; i++)
  C[hash(B[hash(A[i])])]++;
```

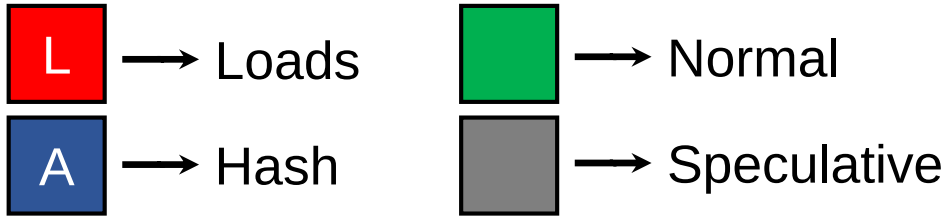


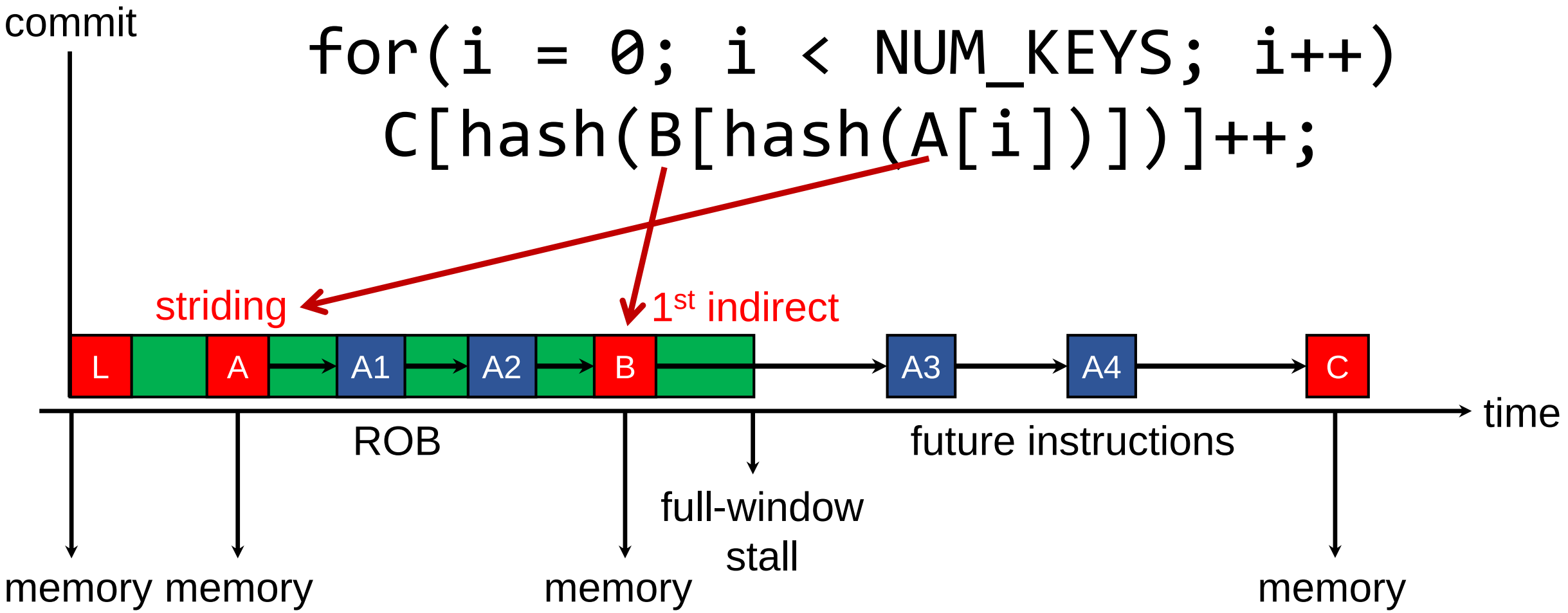
Runahead for Indirect Memory Accesses



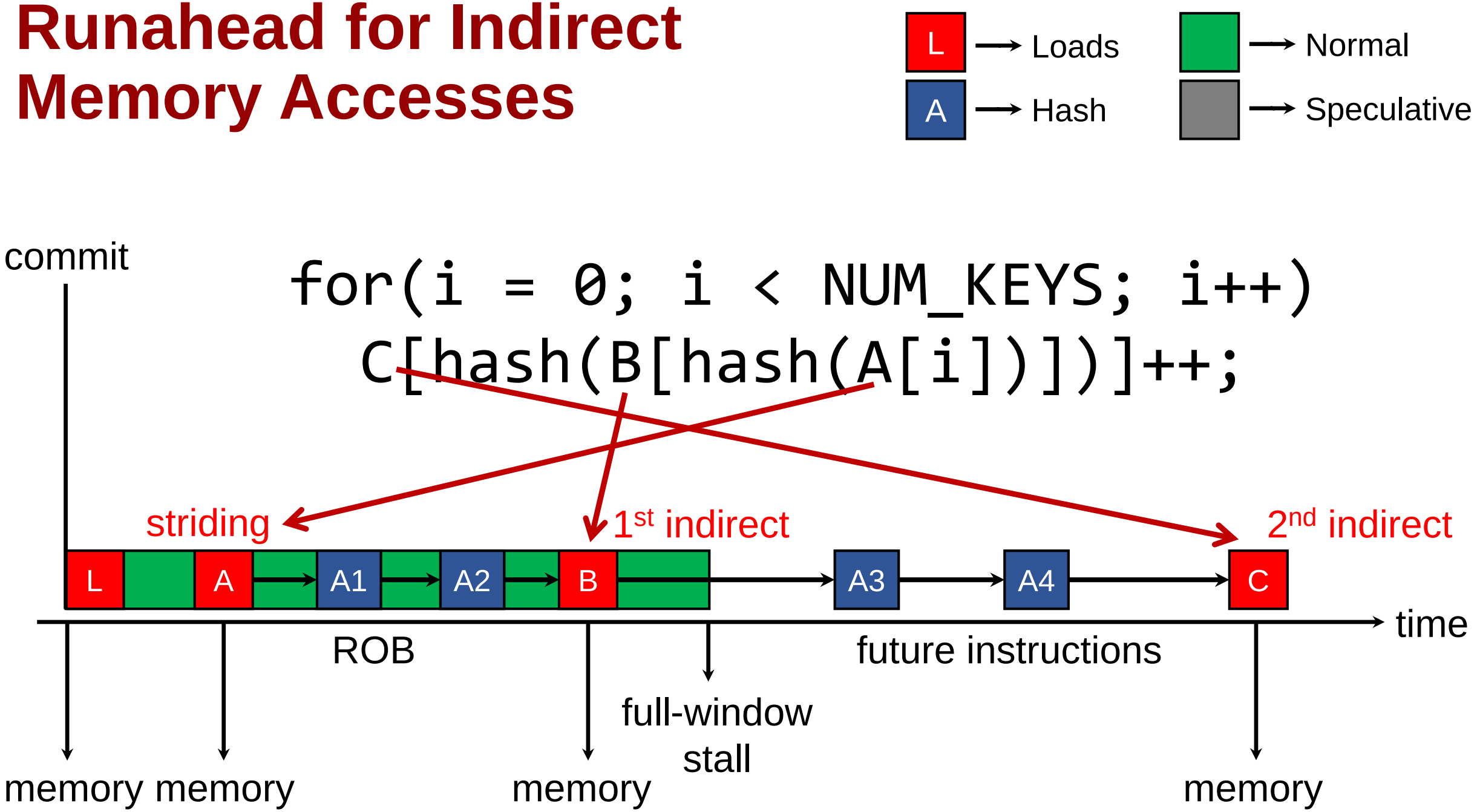


Runahead for Indirect Memory Accesses

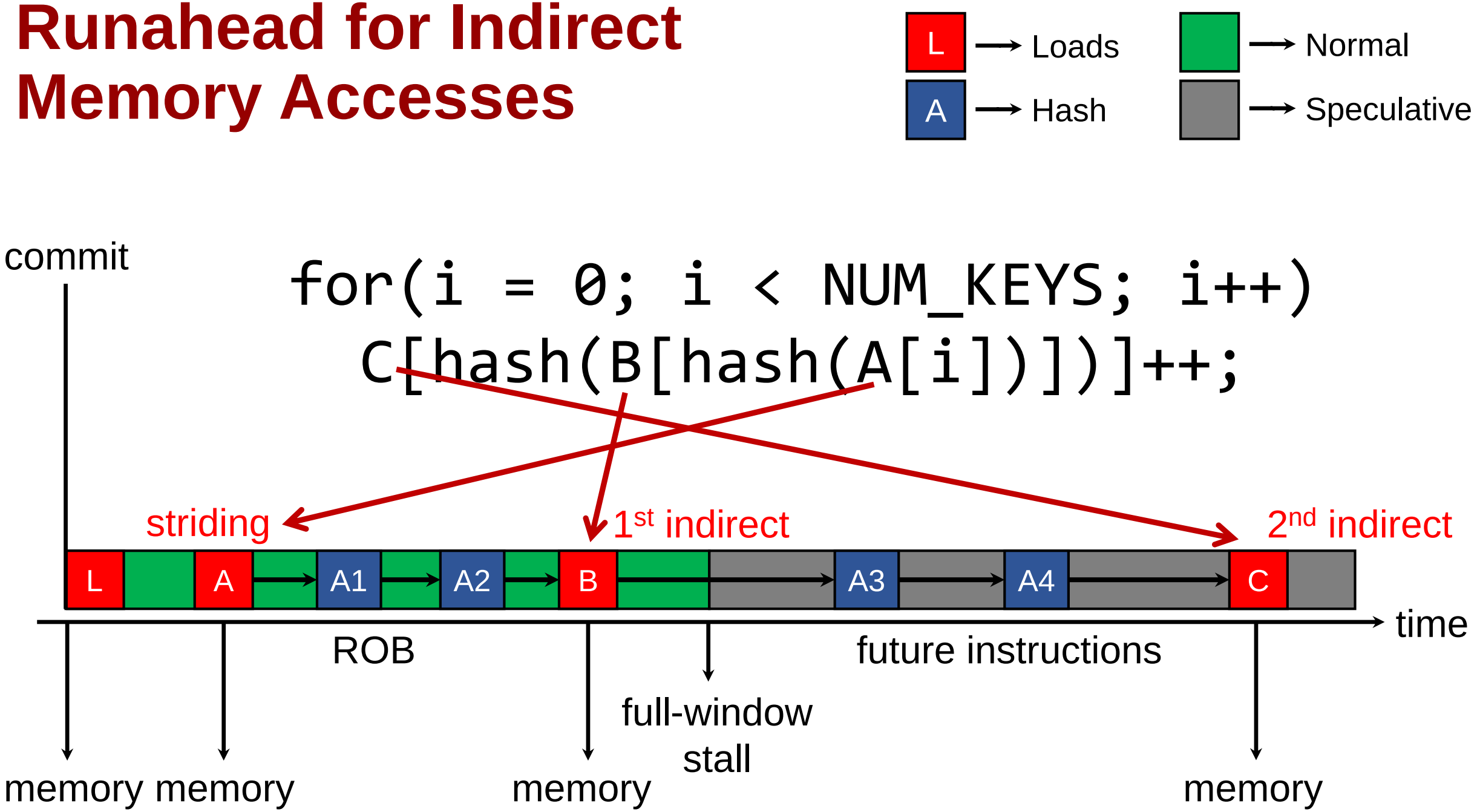




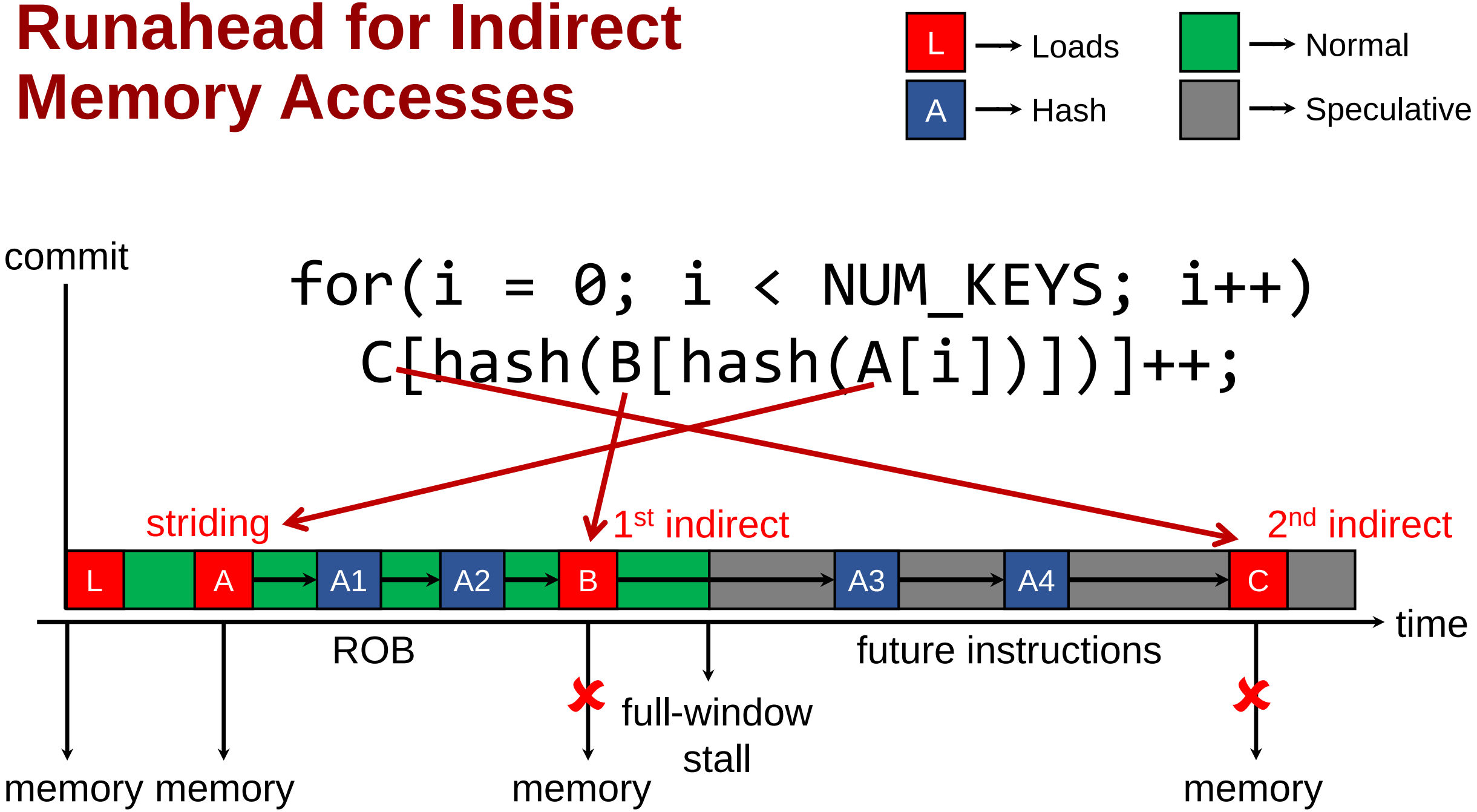
Runahead for Indirect Memory Accesses



Runahead for Indirect Memory Accesses



Runahead for Indirect Memory Accesses



Runahead Cannot Prefetch Indirect Memory Accesses

Runahead Cannot Prefetch Indirect Memory Accesses

1. Core cannot issue indirect loads

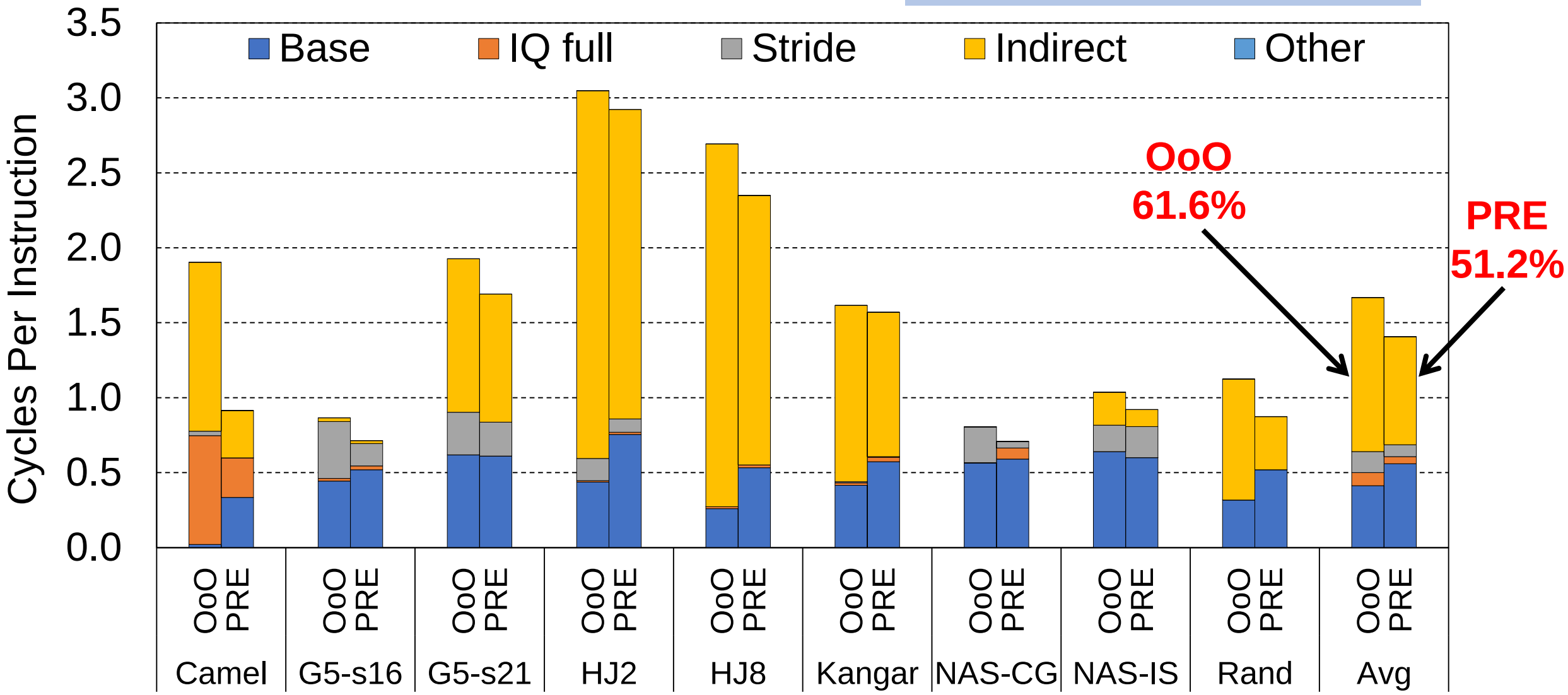
Runahead Cannot Prefetch Indirect Memory Accesses

1. Core cannot issue indirect loads

2. Limited by the front-end bandwidth

A New Microarchitectural Technique is Required

with stride prefetcher



Vector Runahead for Indirect Memory Accesses

Vector Runahead for Indirect Memory Accesses

1. Accesses to array A are striding

Vector Runahead for Indirect Memory Accesses

1. Accesses to array A are striding

2. Executing the indirect chain generates addresses for B and C

Vector Runahead for Indirect Memory Accesses

1. Accesses to array A are striding

2. Executing the indirect chain generates addresses for B and C

i = 0 **A[0] → H_0 → B[H_0] → H_1 → C[H_1]**

Vector Runahead for Indirect Memory Accesses

1. Accesses to array A are striding

2. Executing the indirect chain generates addresses for B and C

i = 0	**A[0] → H_0 → B[H_0] → H_1 → C[H_1]**
i = 1	**A[1] → H_2 → B[H_2] → H_3 → C[H_3]**

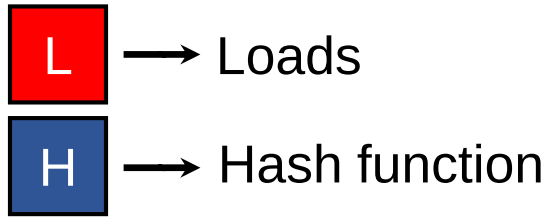
Vector Runahead for Indirect Memory Accesses

1. Accesses to array A are striding

2. Executing the indirect chain generates addresses for B and C

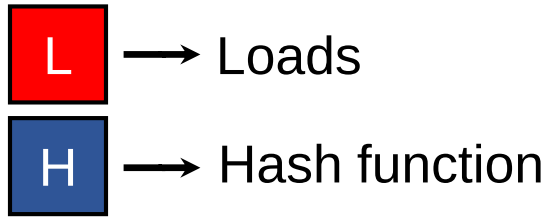
i = 0	A[0] → H_0 → B[H_0] → H_1 → C[H_1]
i = 1	A[1] → H_2 → B[H_2] → H_3 → C[H_3]
i = 2	A[2] → H_4 → B[H_4] → H_5 → C[H_5]

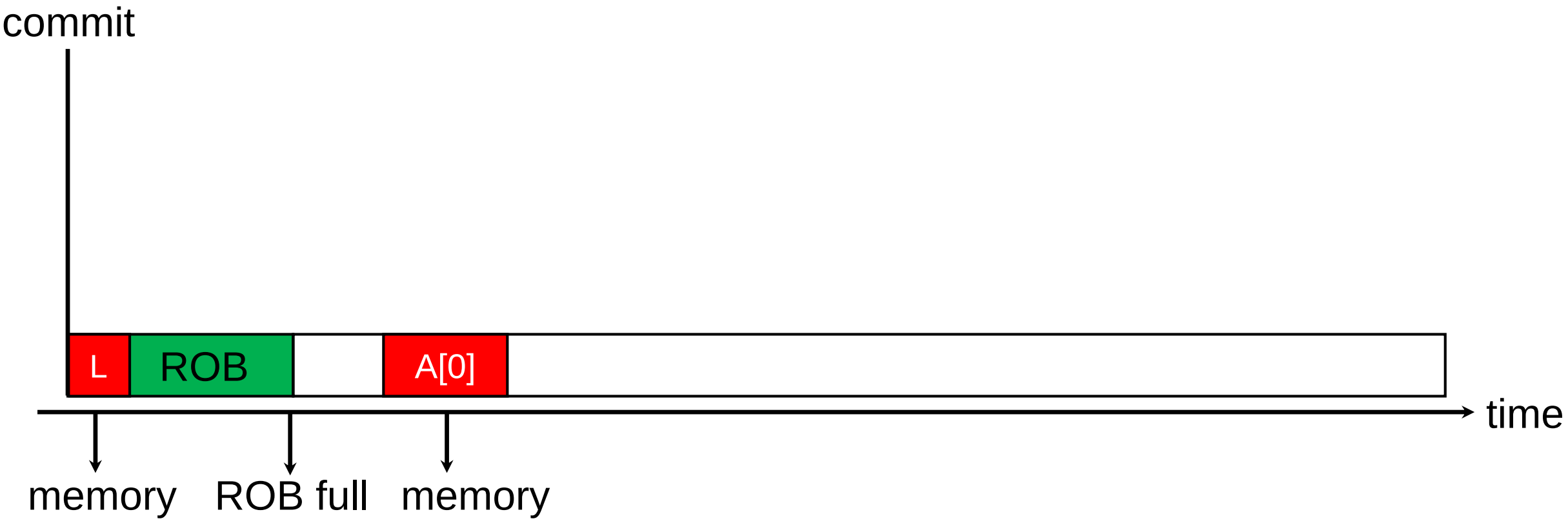
Generating Future Indirect Memory Addresses



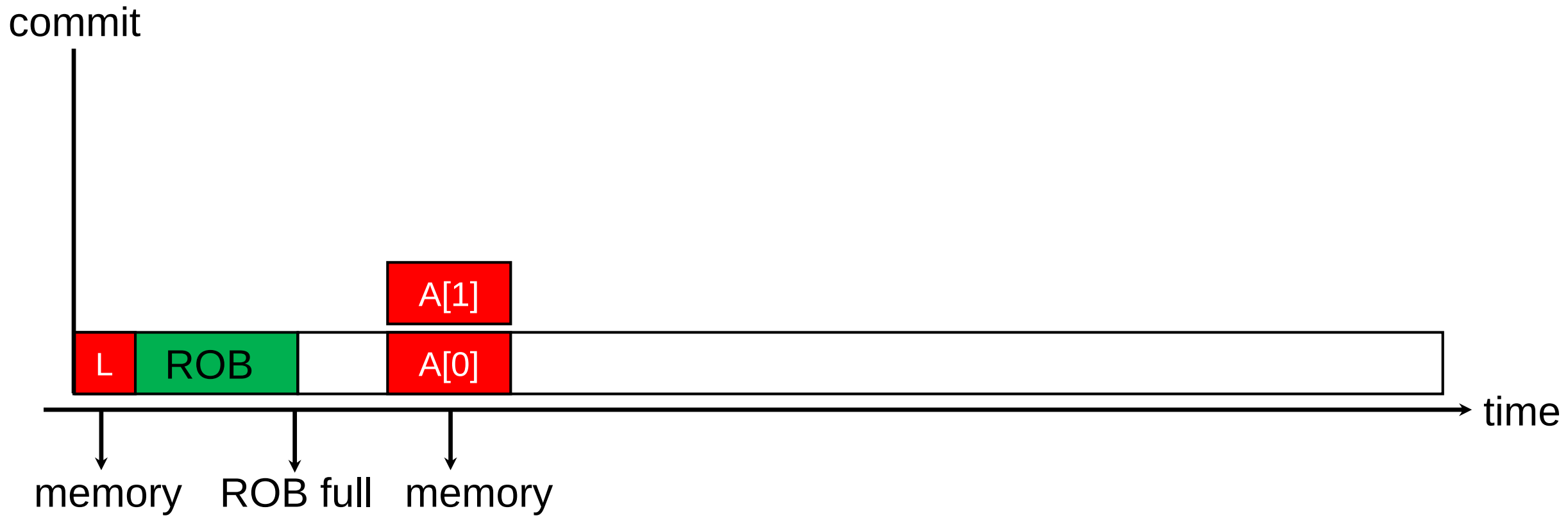
Generating Future Indirect Memory Addresses

Generating Future Indirect Memory Addresses

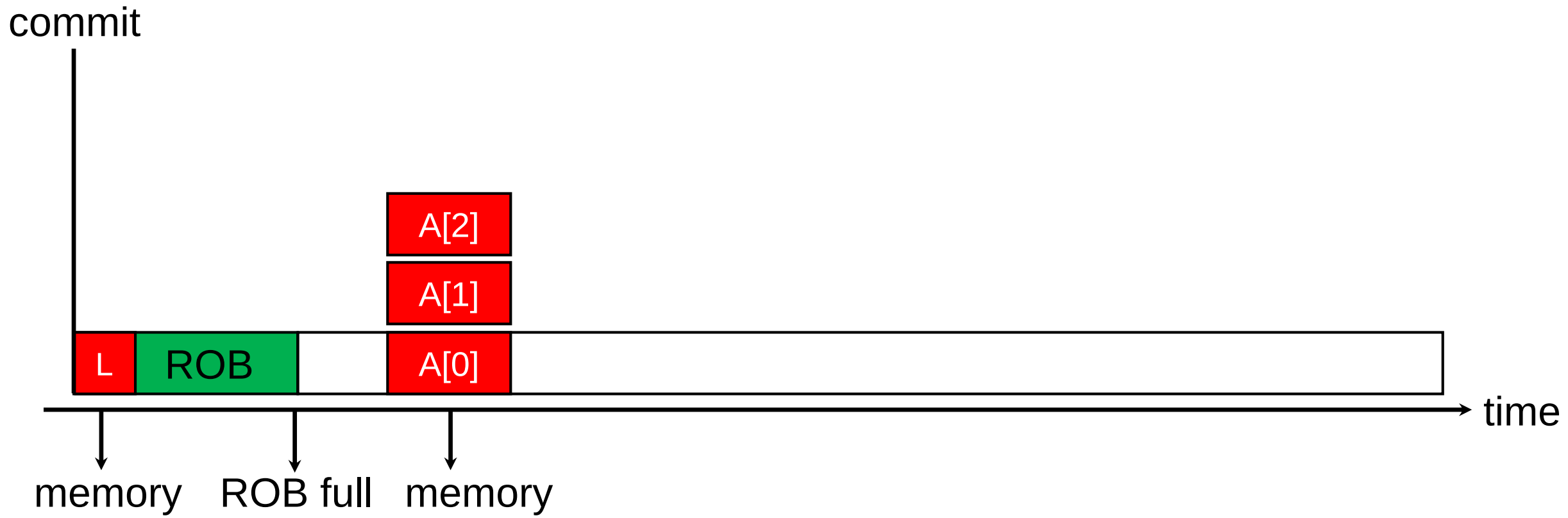




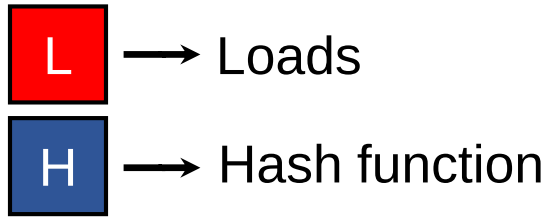
Generating Future Indirect Memory Addresses

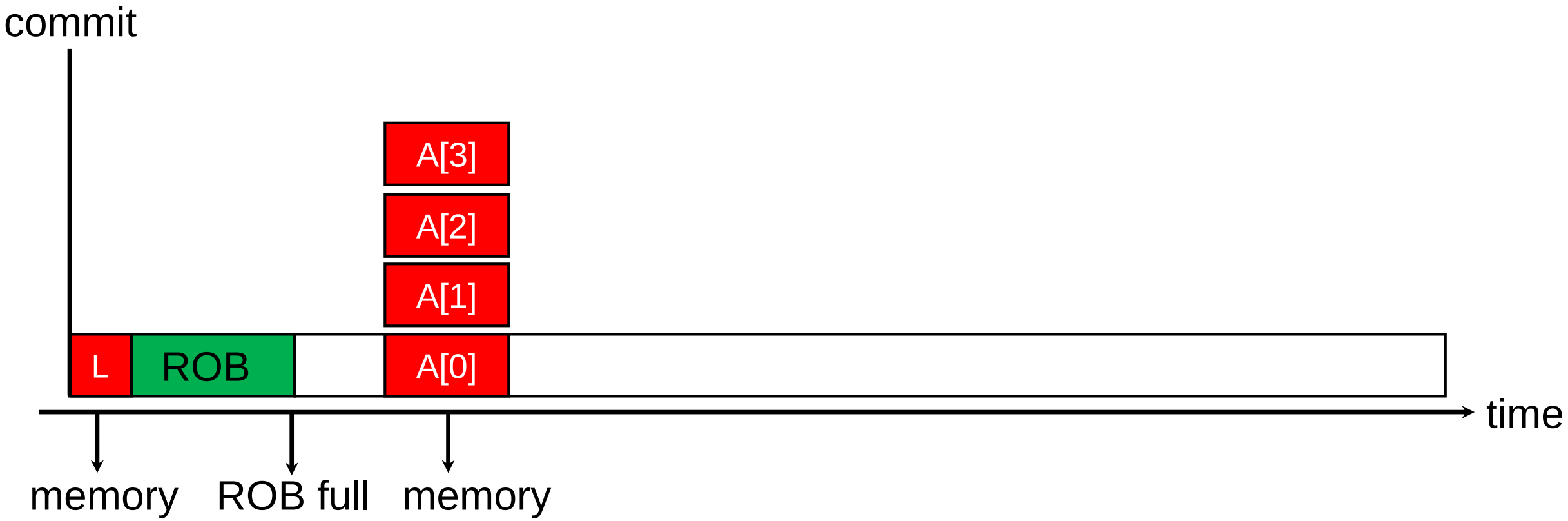


Generating Future Indirect Memory Addresses

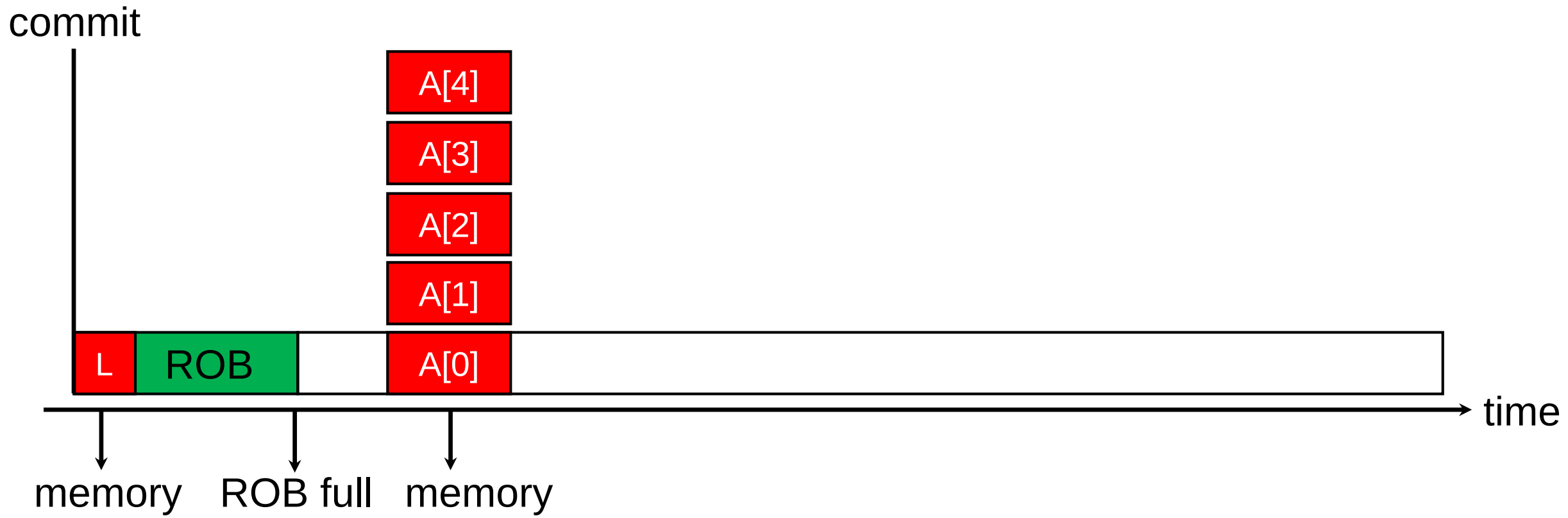


Generating Future Indirect Memory Addresses

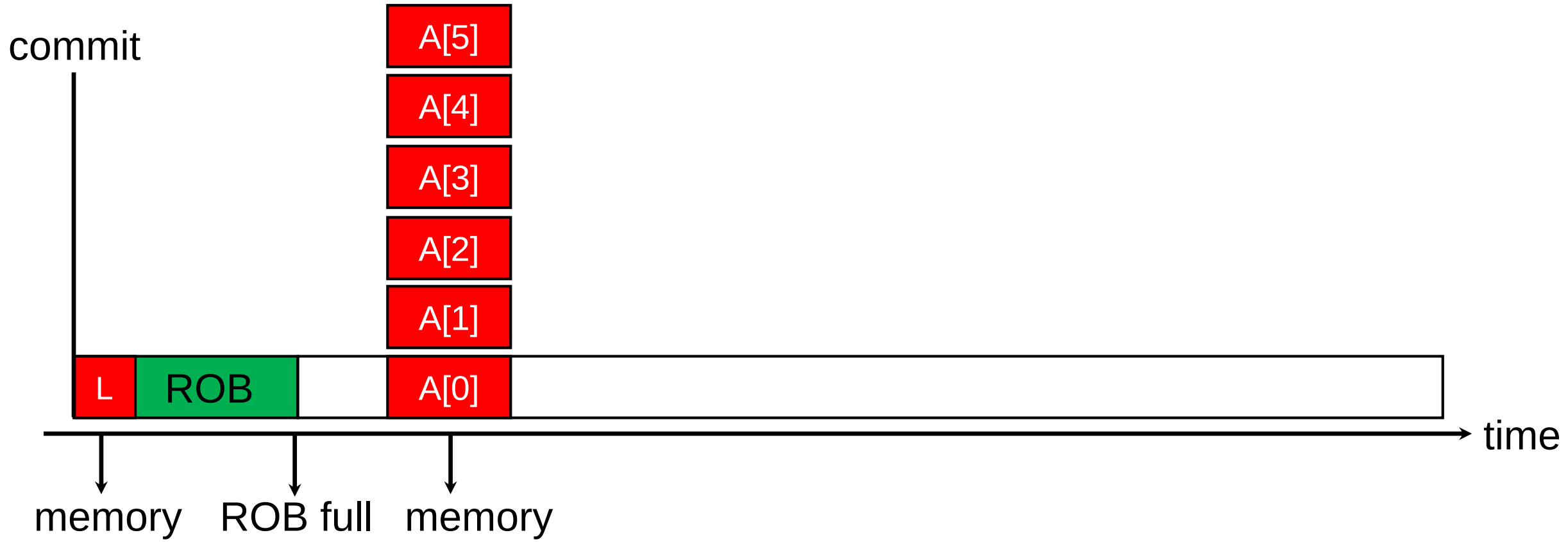




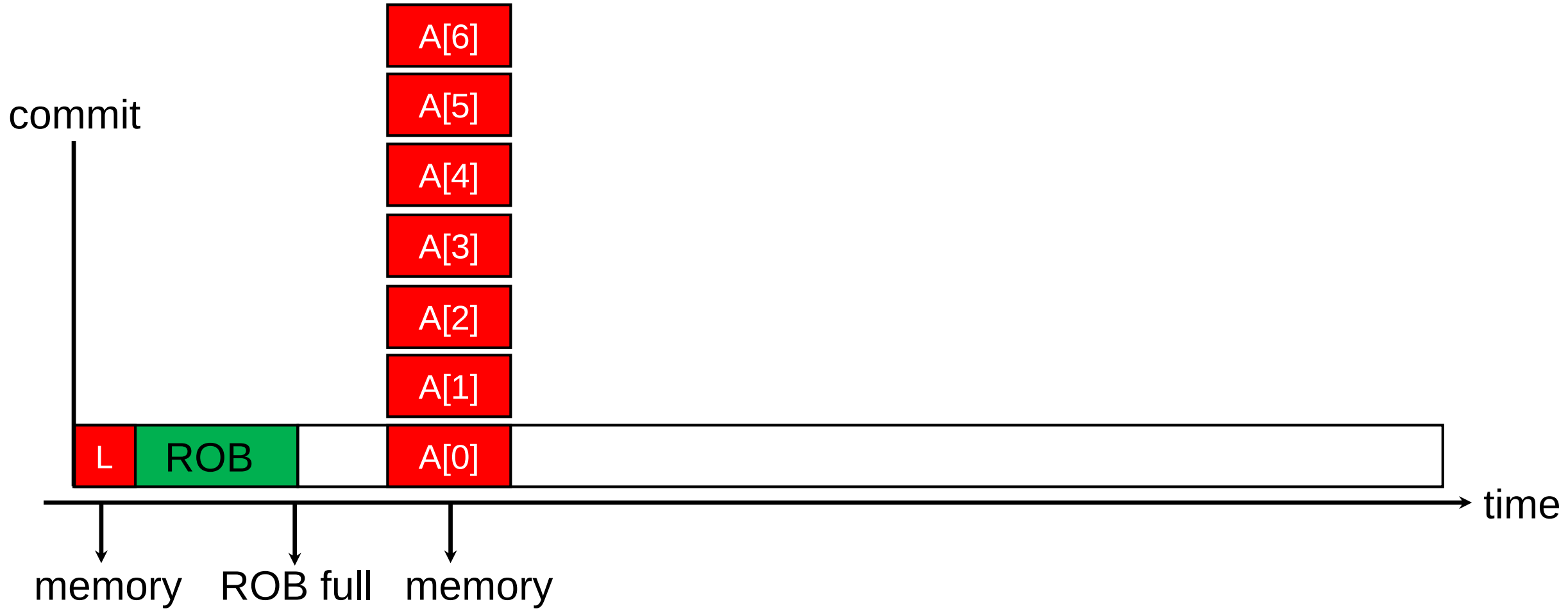
Generating Future Indirect Memory Addresses



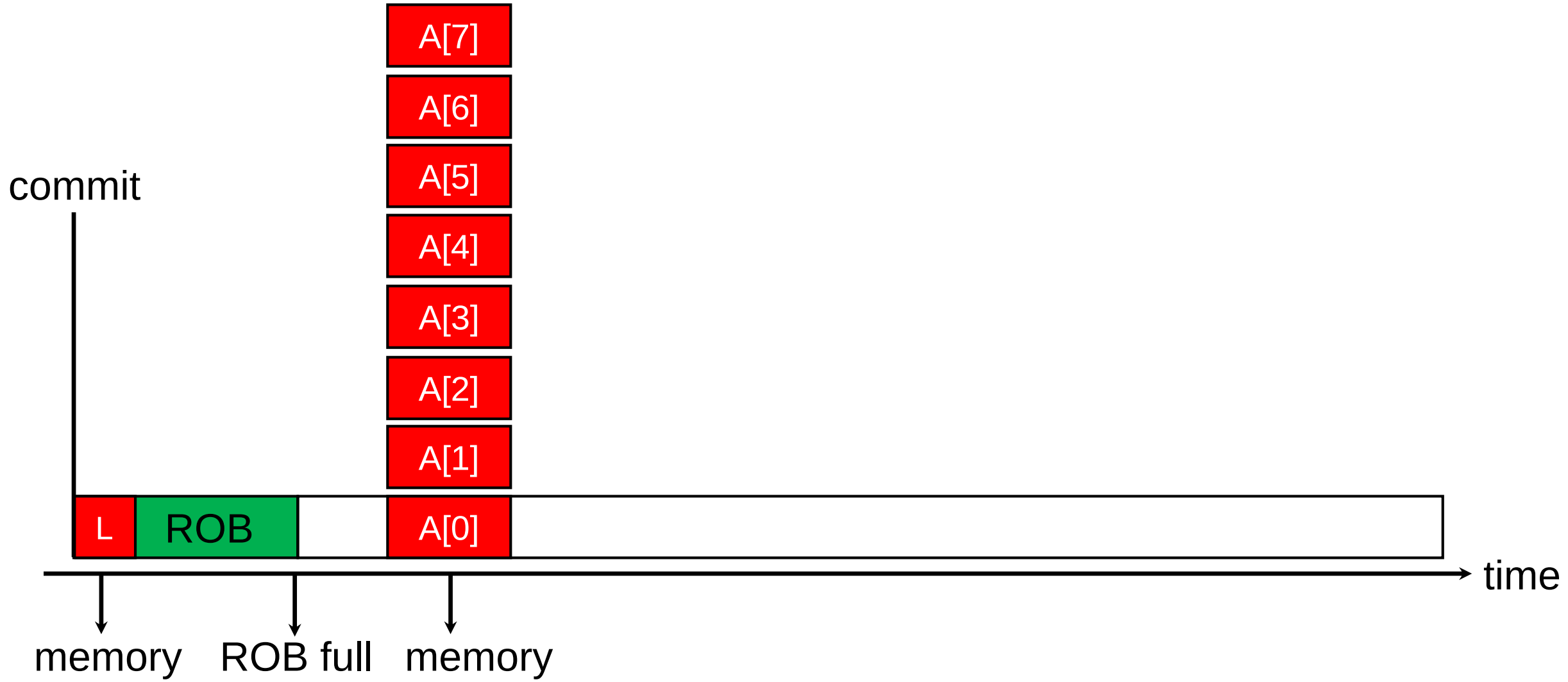
Generating Future Indirect Memory Addresses



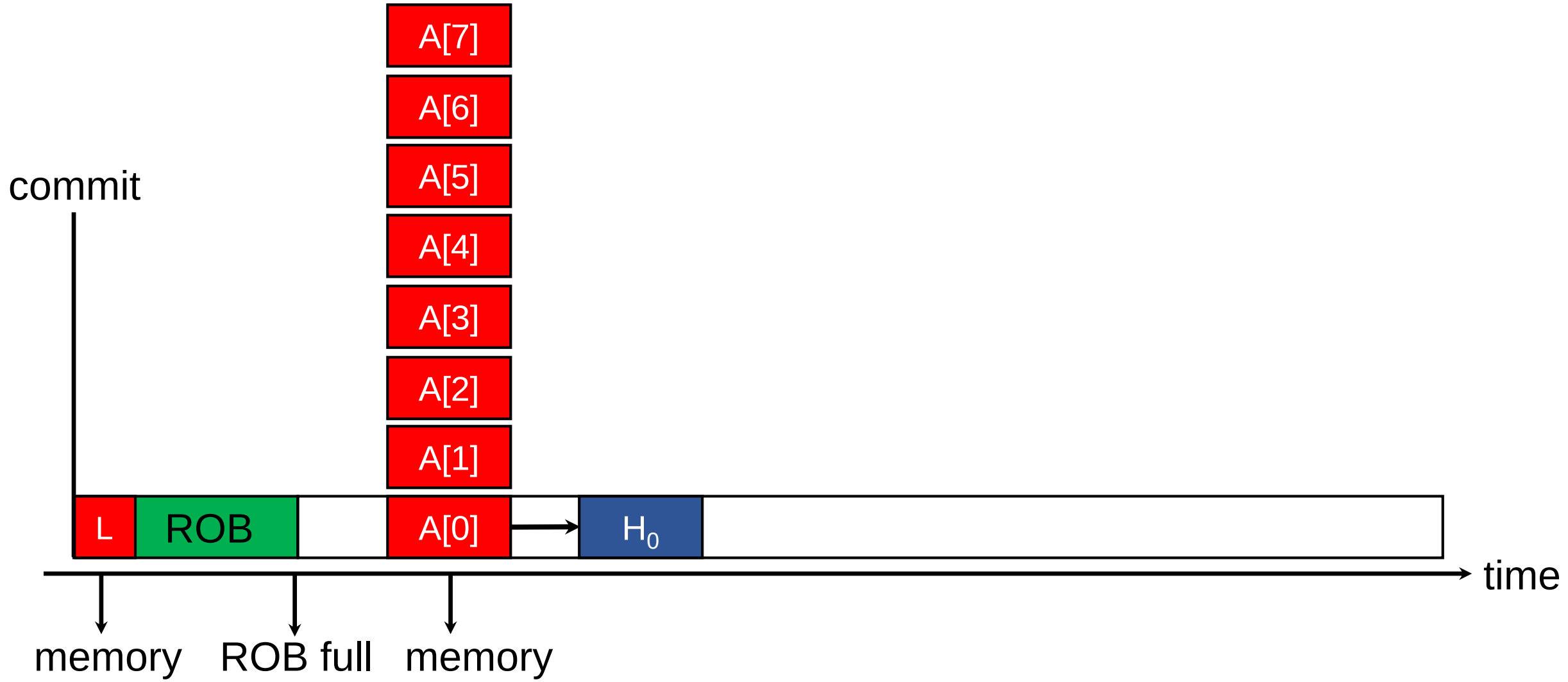
Generating Future Indirect Memory Addresses



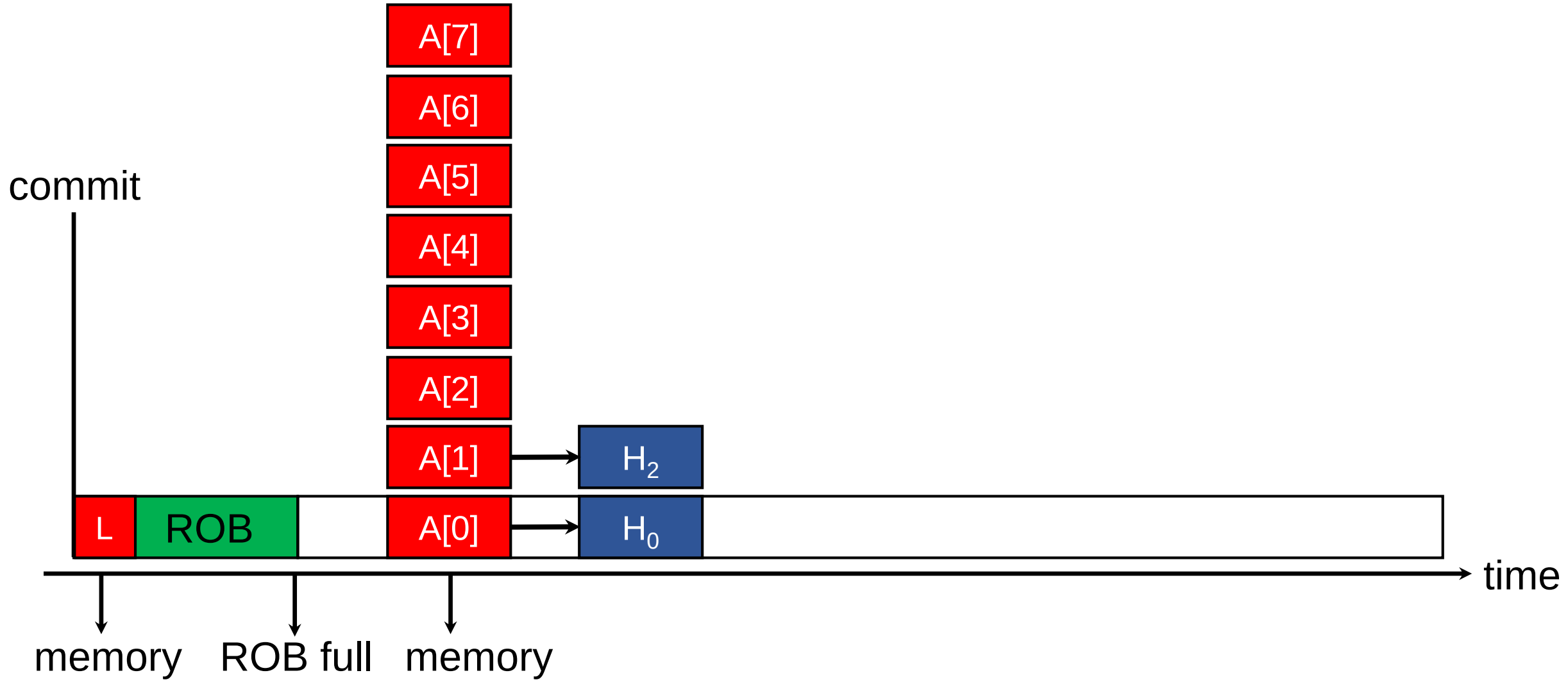
Generating Future Indirect Memory Addresses



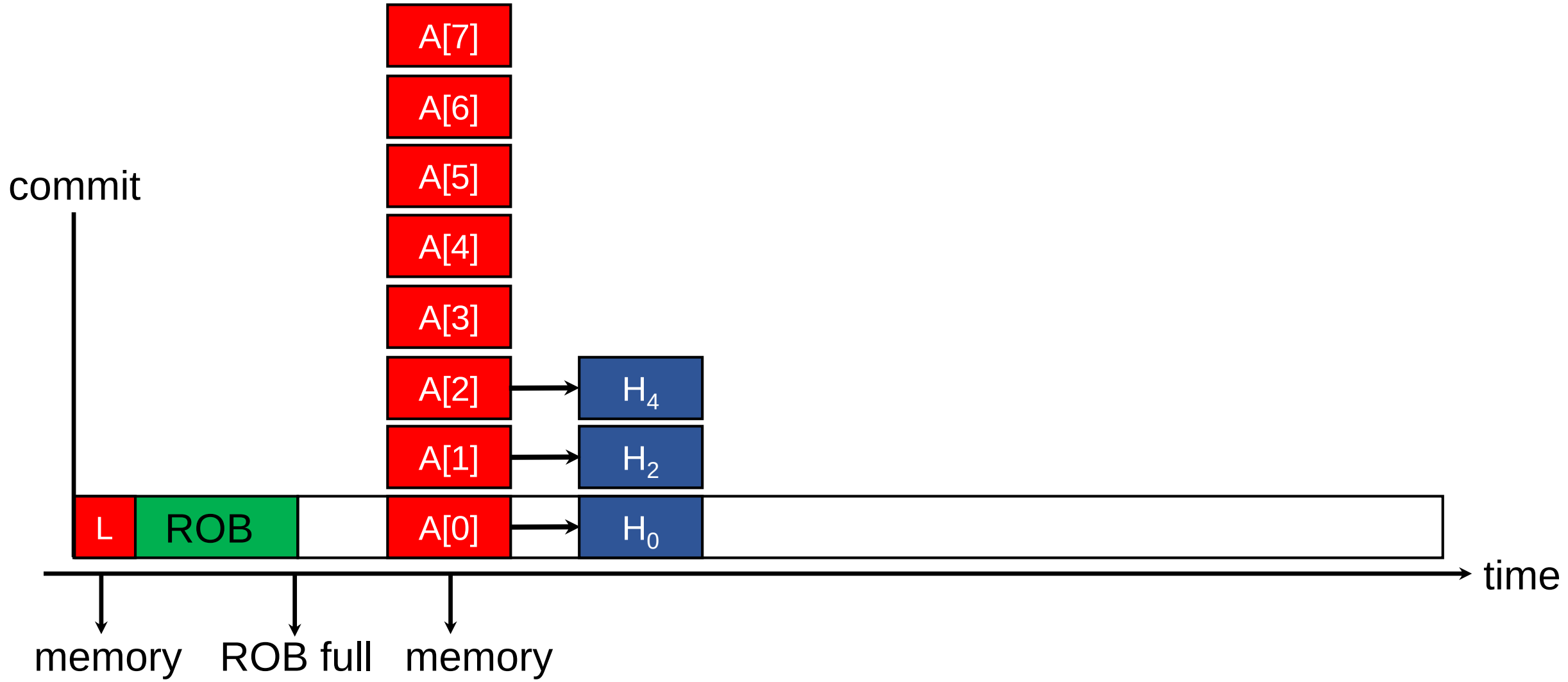
Generating Future Indirect Memory Addresses



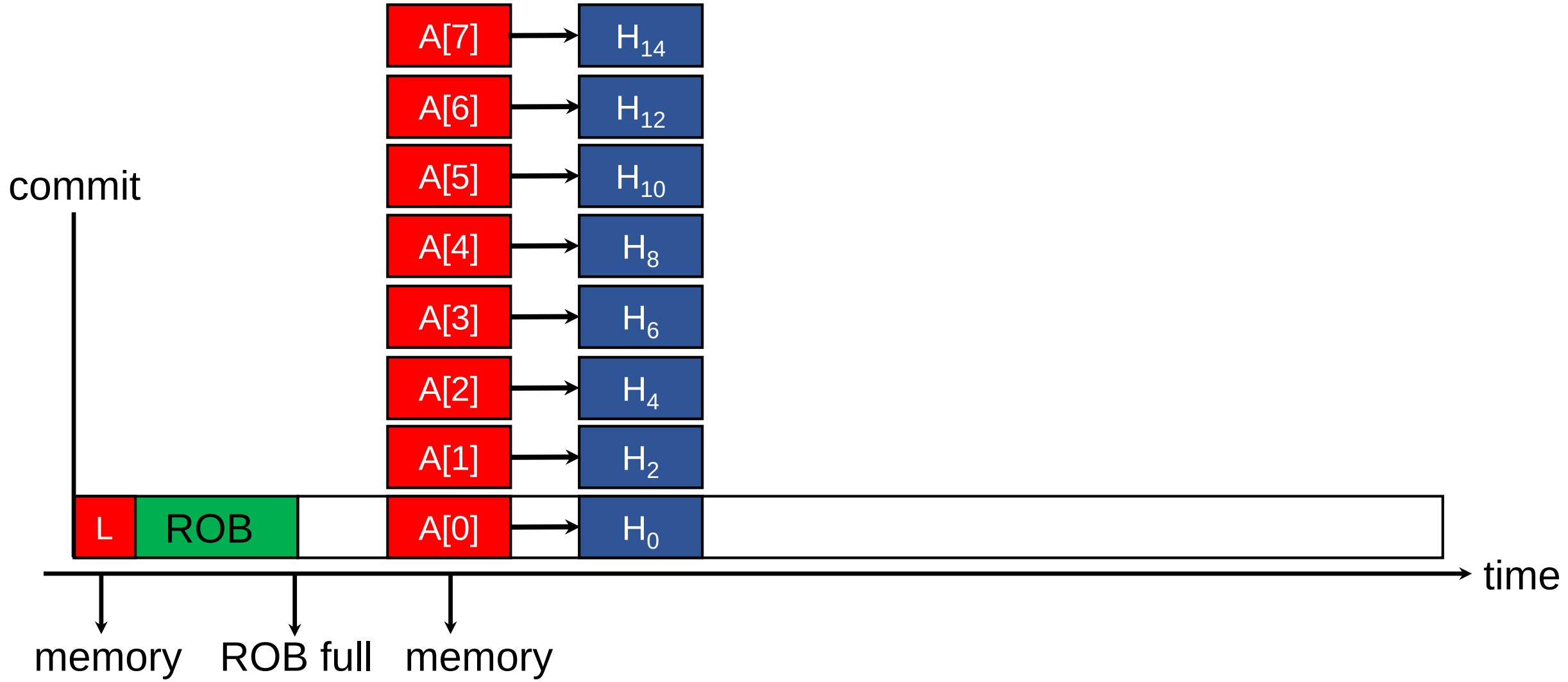
Generating Future Indirect Memory Addresses



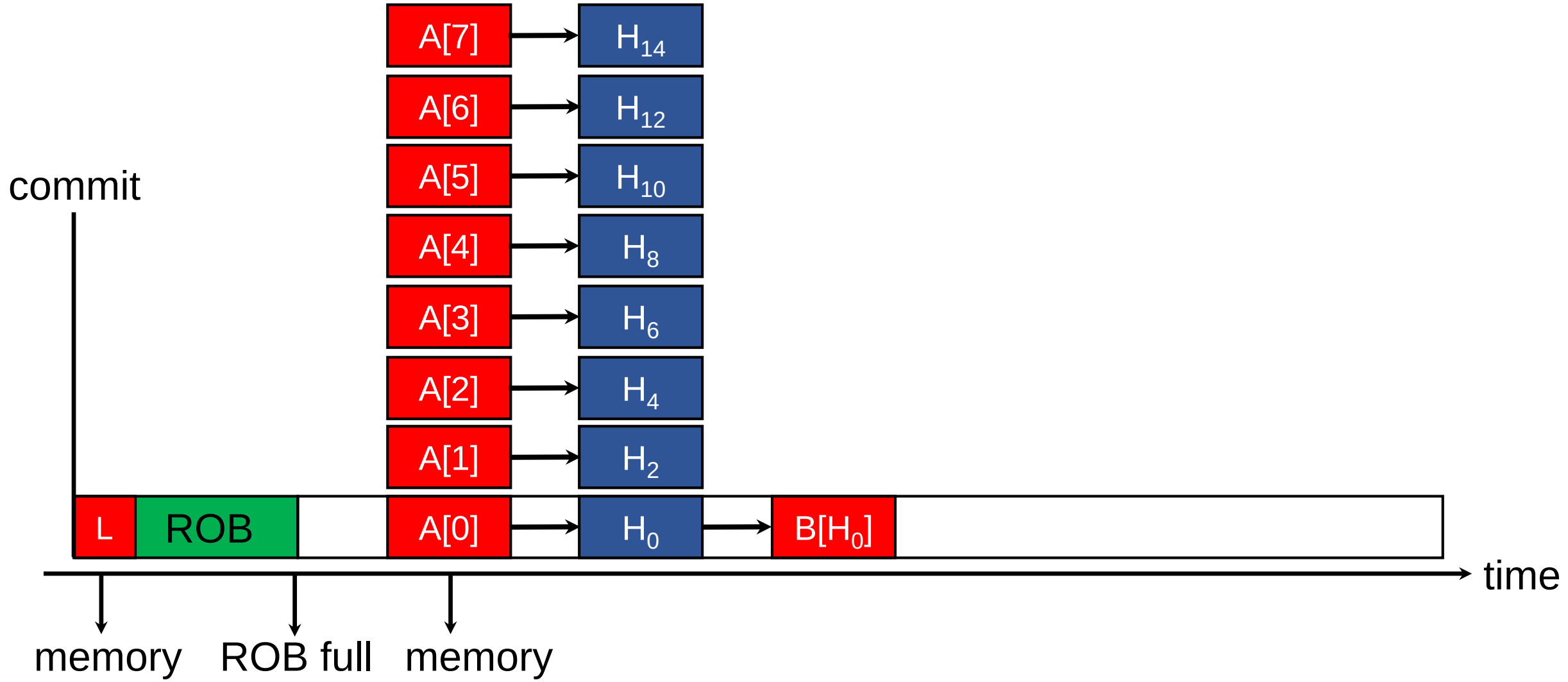
Generating Future Indirect Memory Addresses



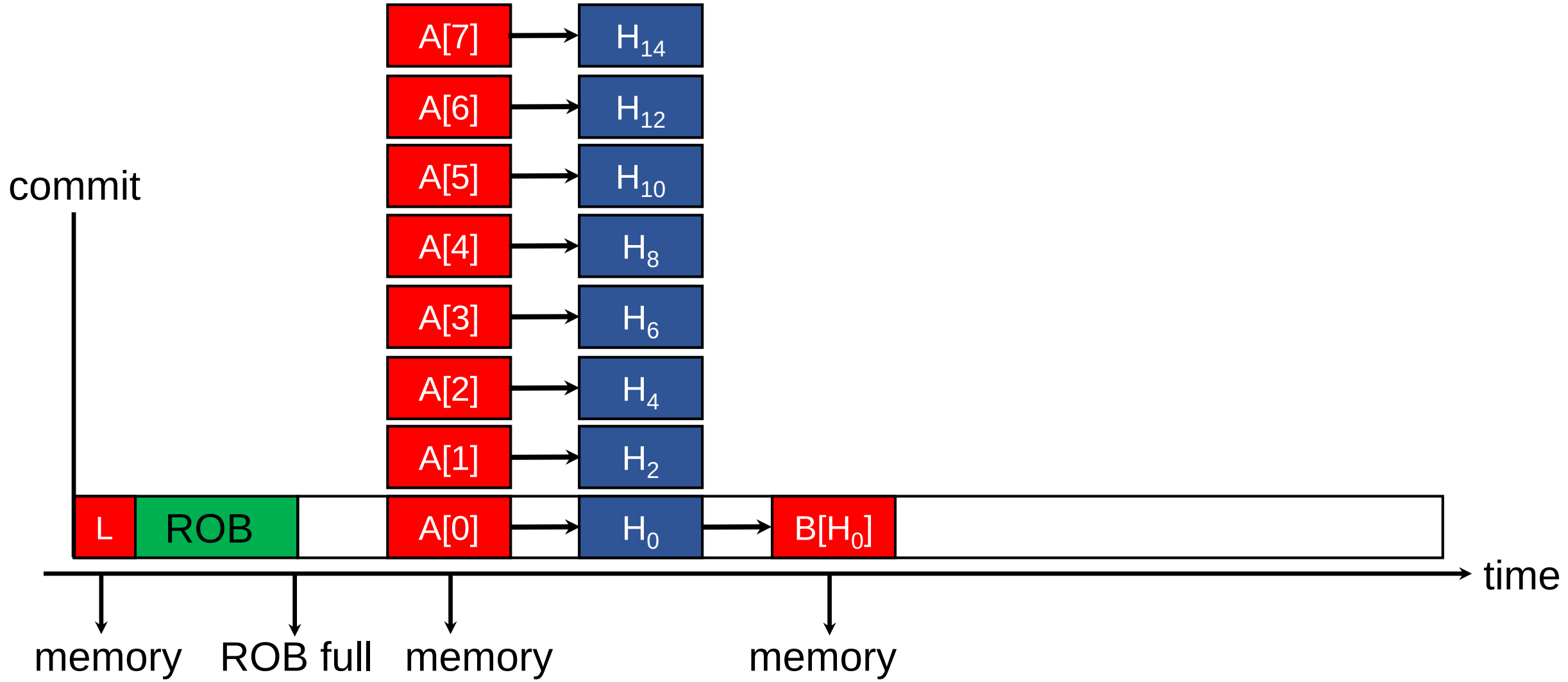
Generating Future Indirect Memory Addresses



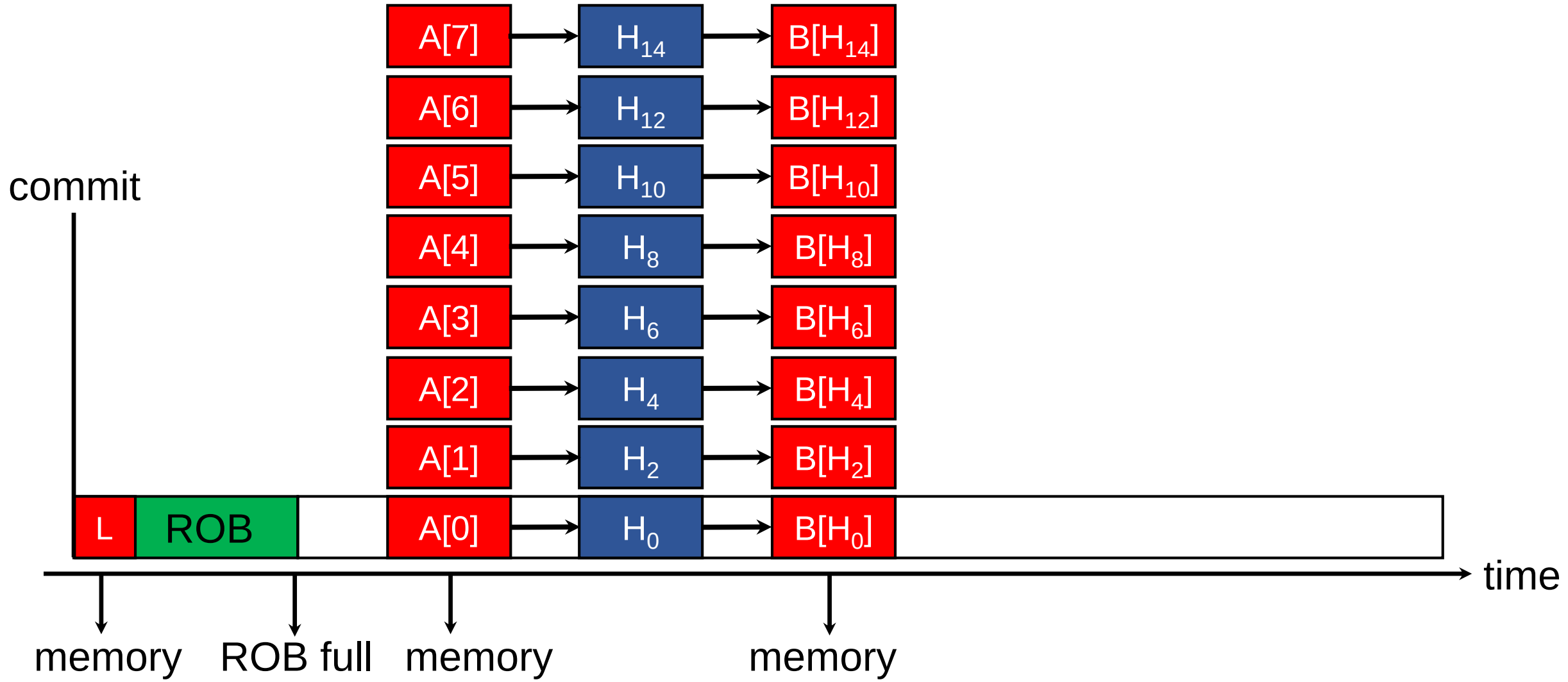
Generating Future Indirect Memory Addresses



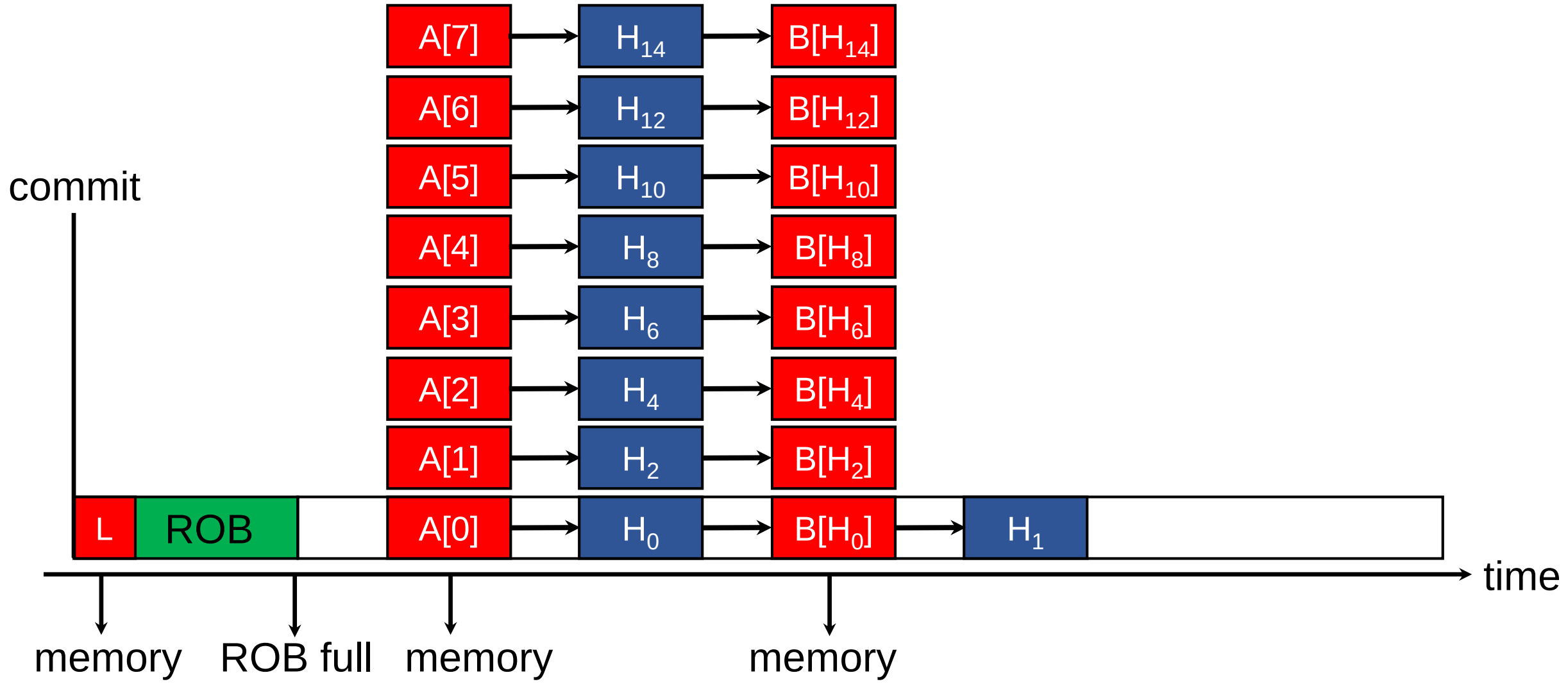
Generating Future Indirect Memory Addresses



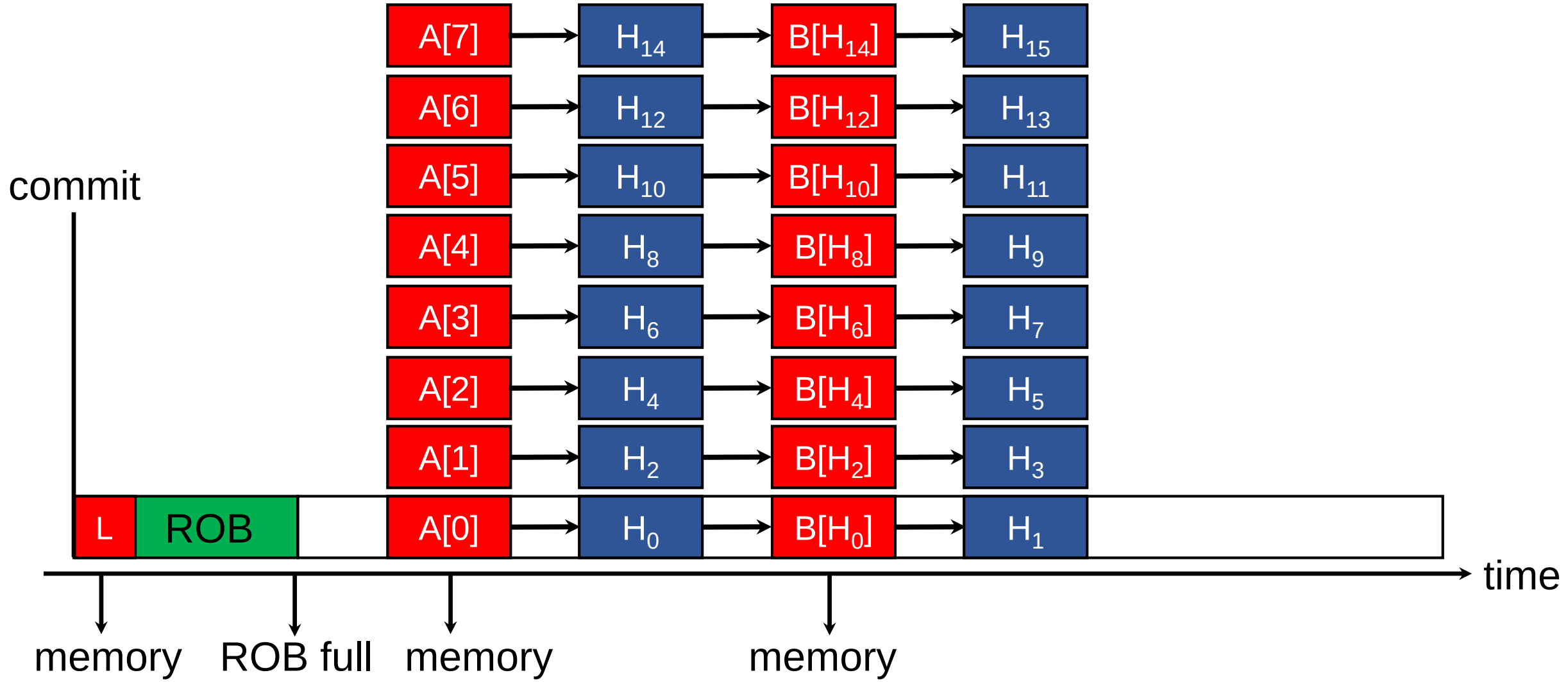
Generating Future Indirect Memory Addresses



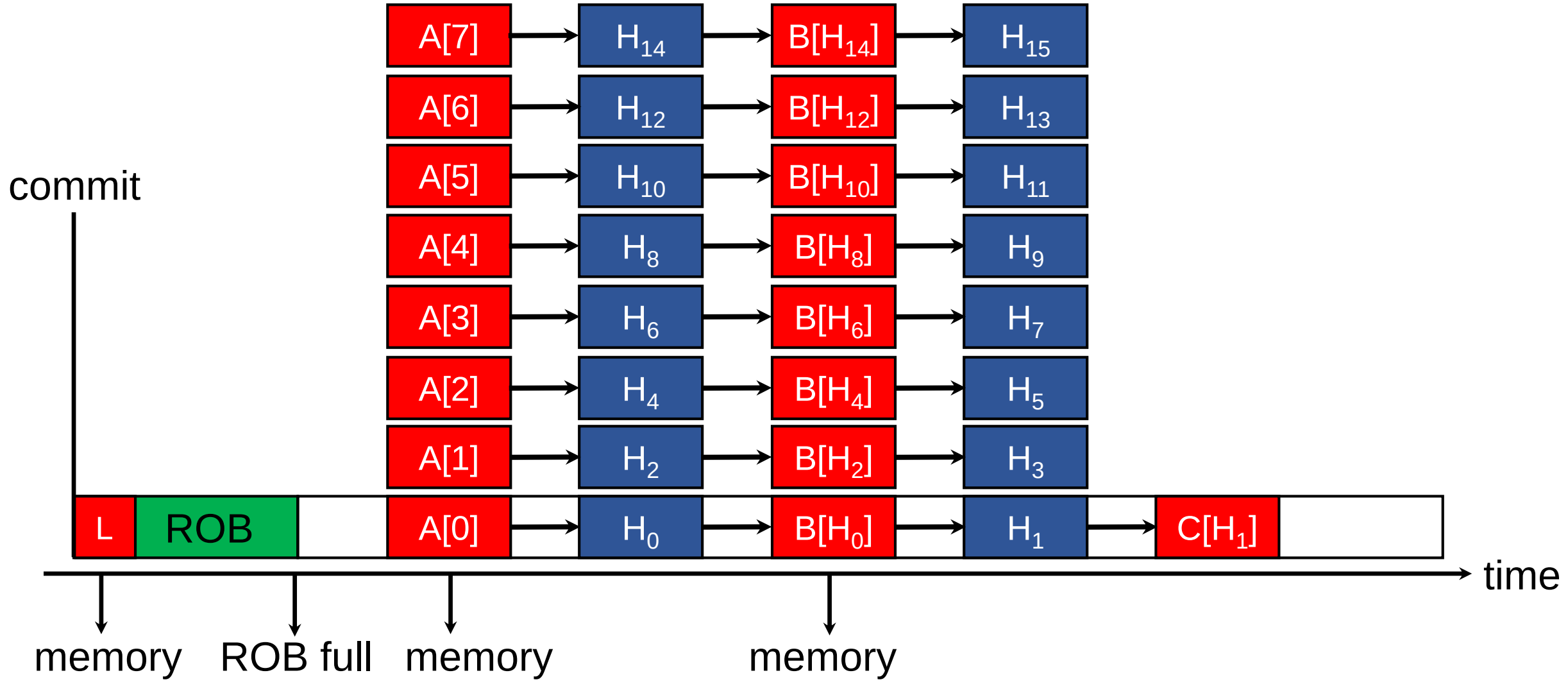
Generating Future Indirect Memory Addresses



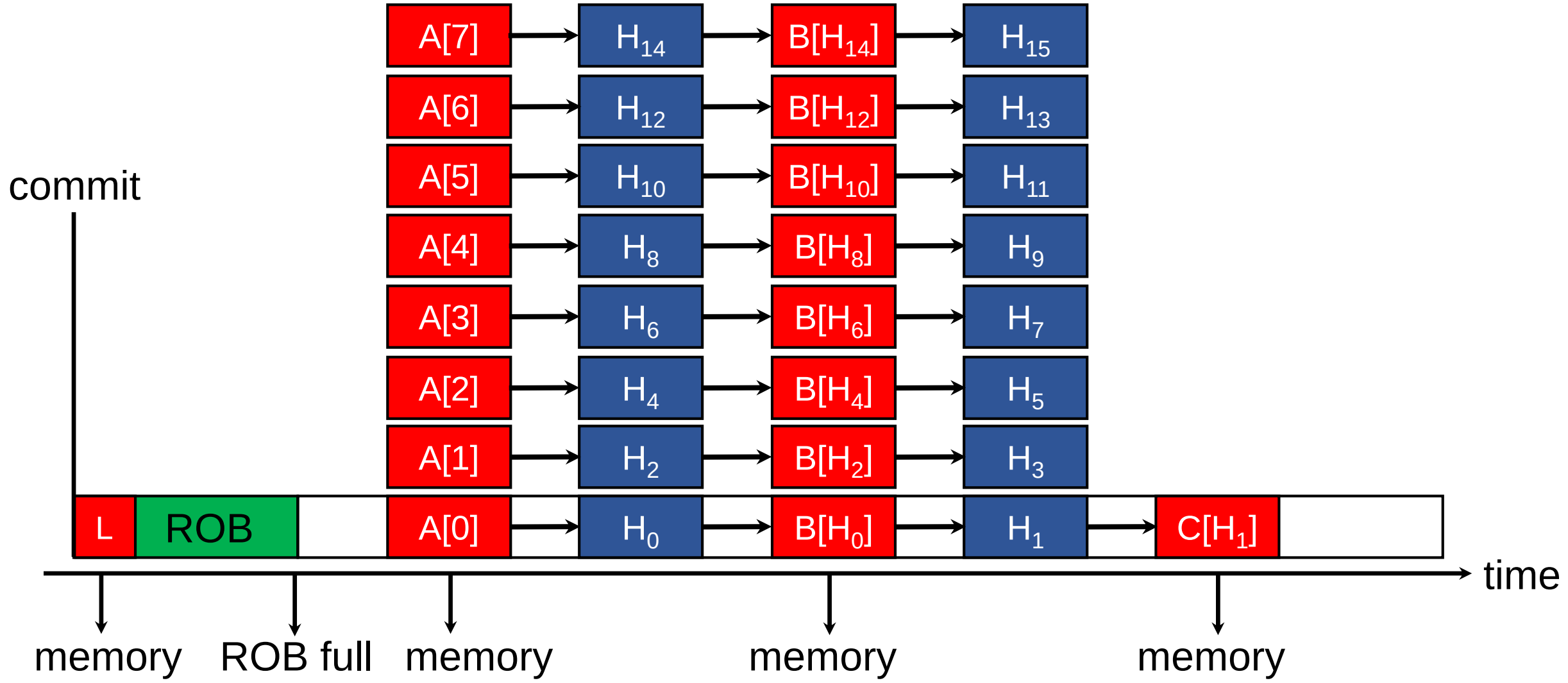
Generating Future Indirect Memory Addresses



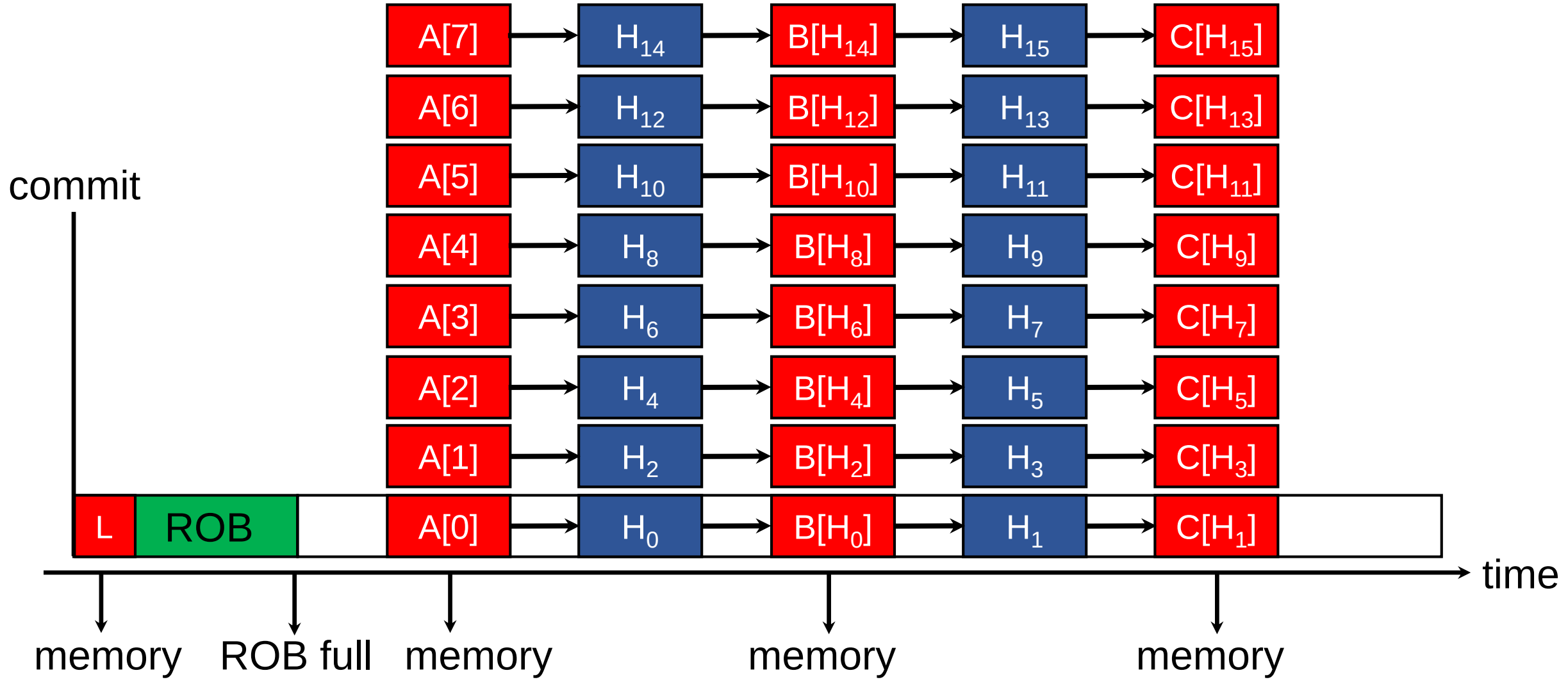
Generating Future Indirect Memory Addresses



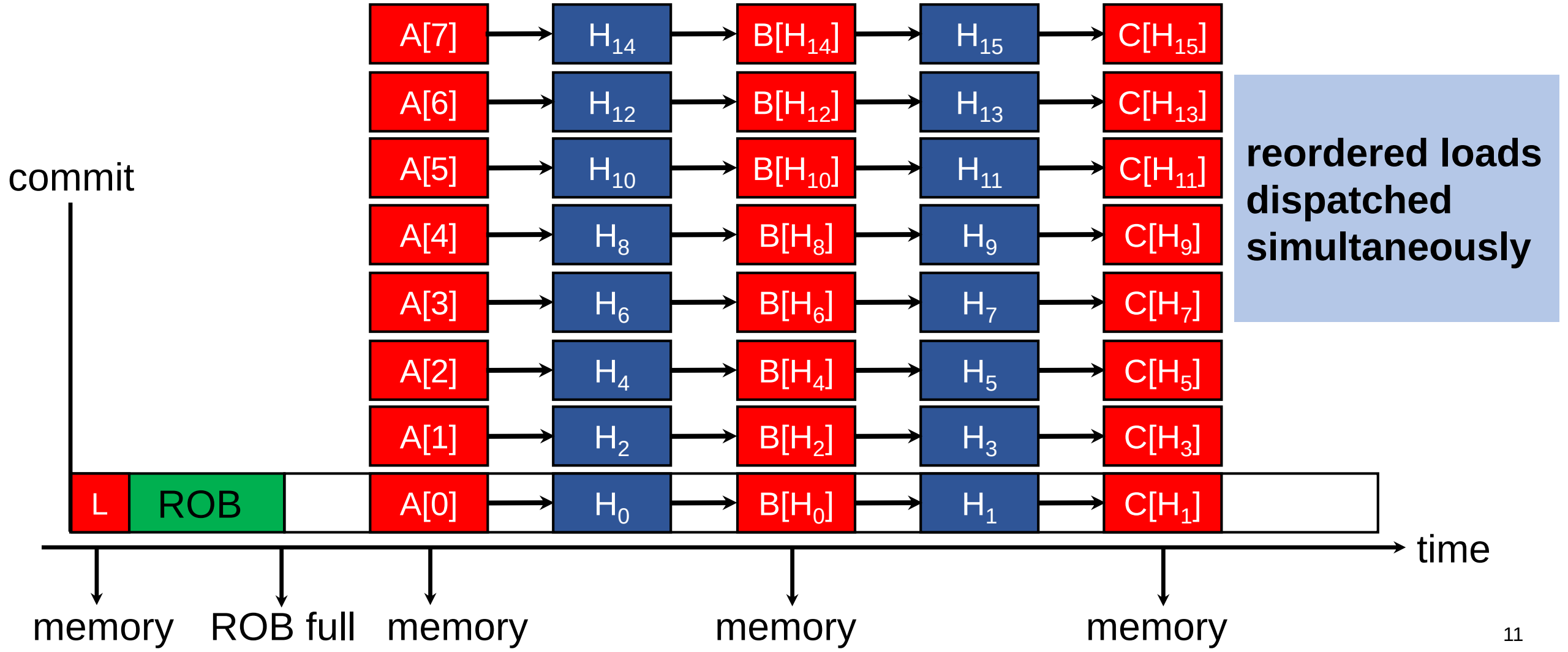
Generating Future Indirect Memory Addresses



Generating Future Indirect Memory Addresses

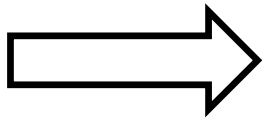


Generating Future Indirect Memory Addresses



Vectorization for Increasing Back-End Throughput

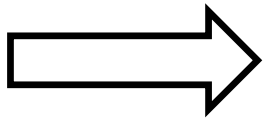
1. Back-end stalls due to large number of scalar instructions



issue queue
functional units
register file

Vectorization for Increasing Back-End Throughput

1. Back-end stalls due to large number of scalar instructions

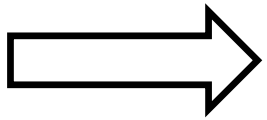


issue queue
functional units
register file

2. Speculatively vectorize indirect chains
   - Back-end stalls reduce by **~8x**
   - Use vector (physical) register file instead

Vectorization for Increasing Back-End Throughput

1. Back-end stalls due to large number of scalar instructions

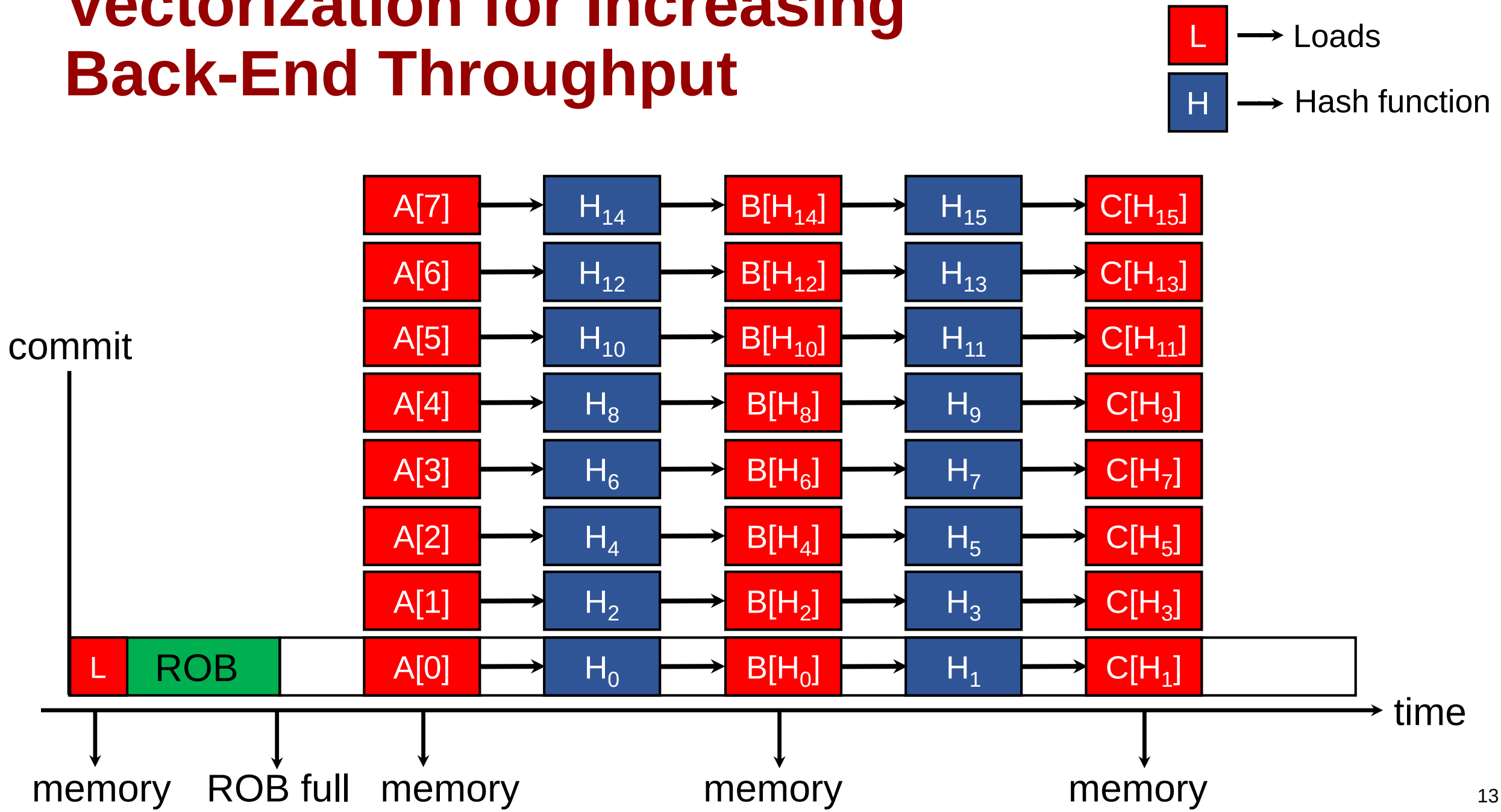


issue queue
functional units
register file

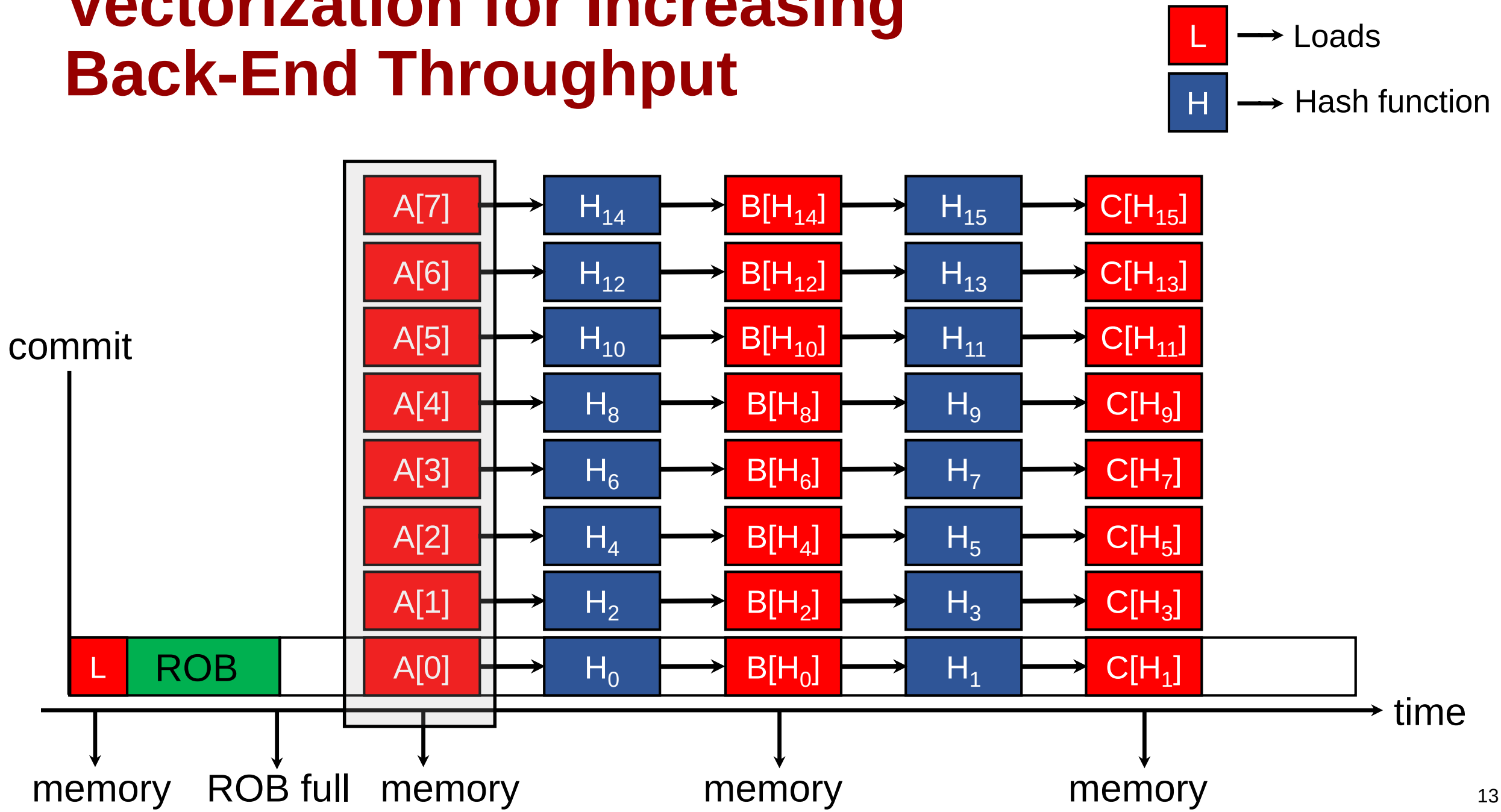
2. Speculatively vectorize indirect chains
   - Back-end stalls reduce by **~8x**
   - Use vector (physical) register file instead

3. Workloads do not have to be vectorizable

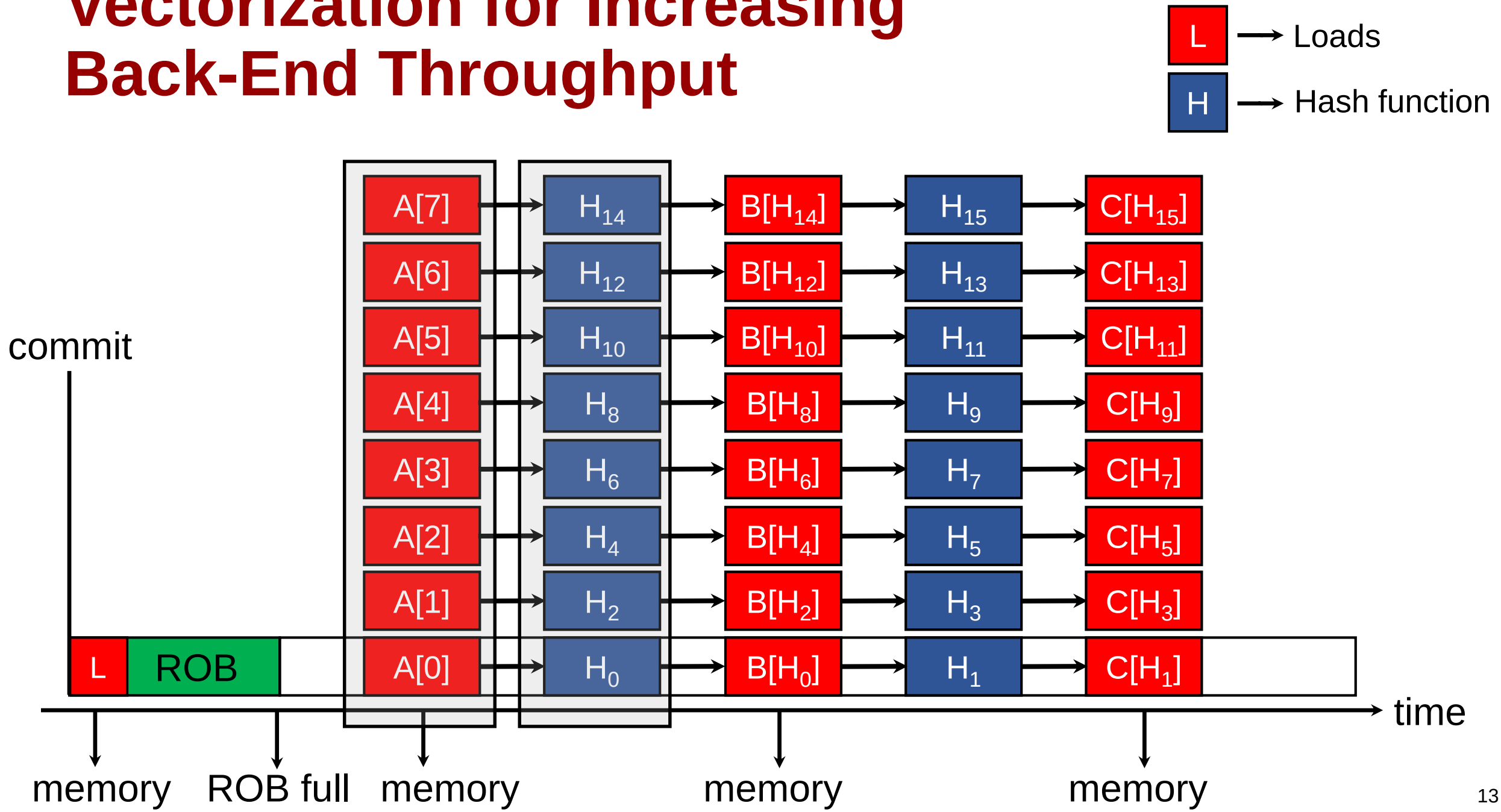
Vectorization for Increasing Back-End Throughput



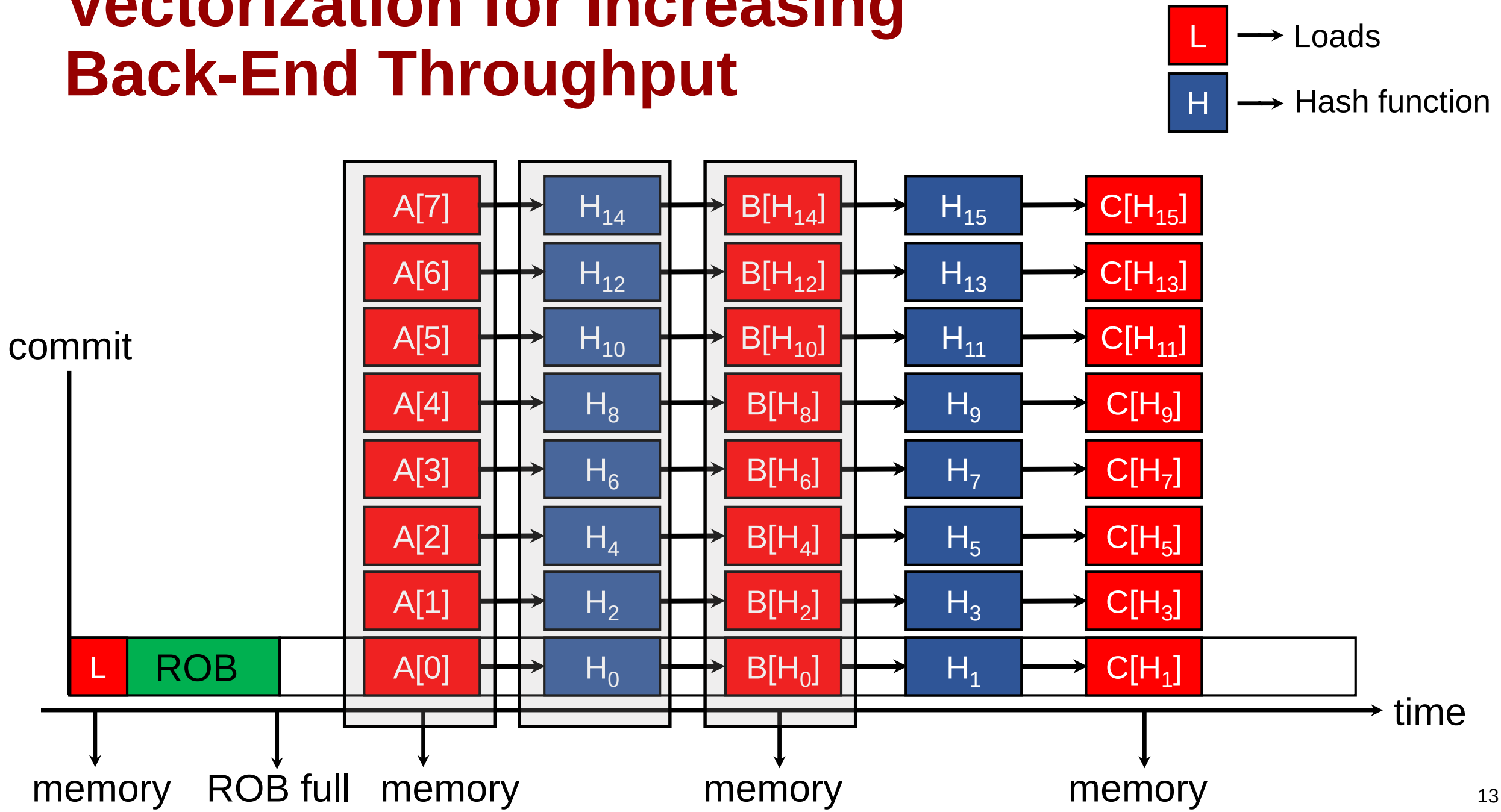
Vectorization for Increasing Back-End Throughput



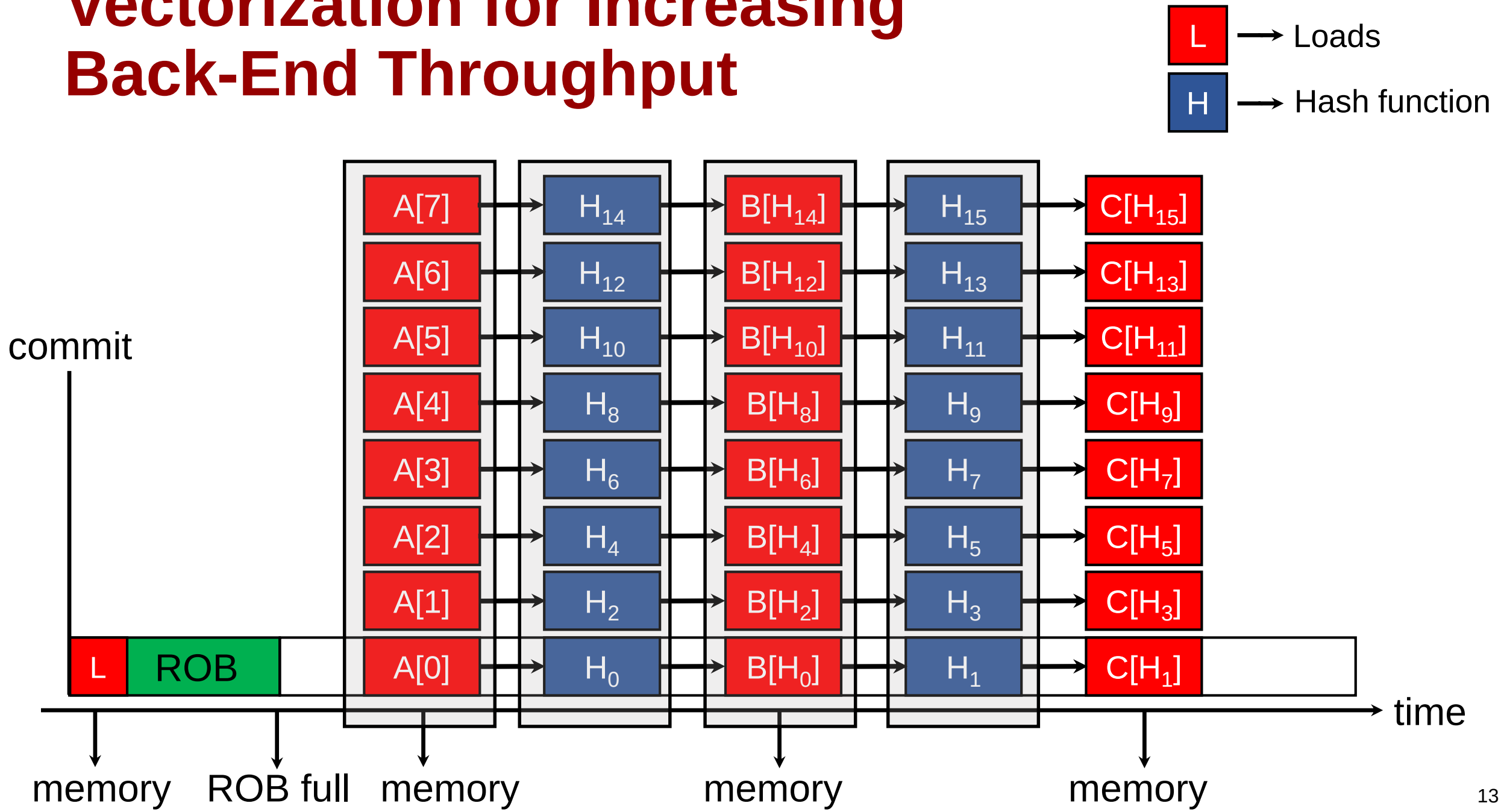
Vectorization for Increasing Back-End Throughput



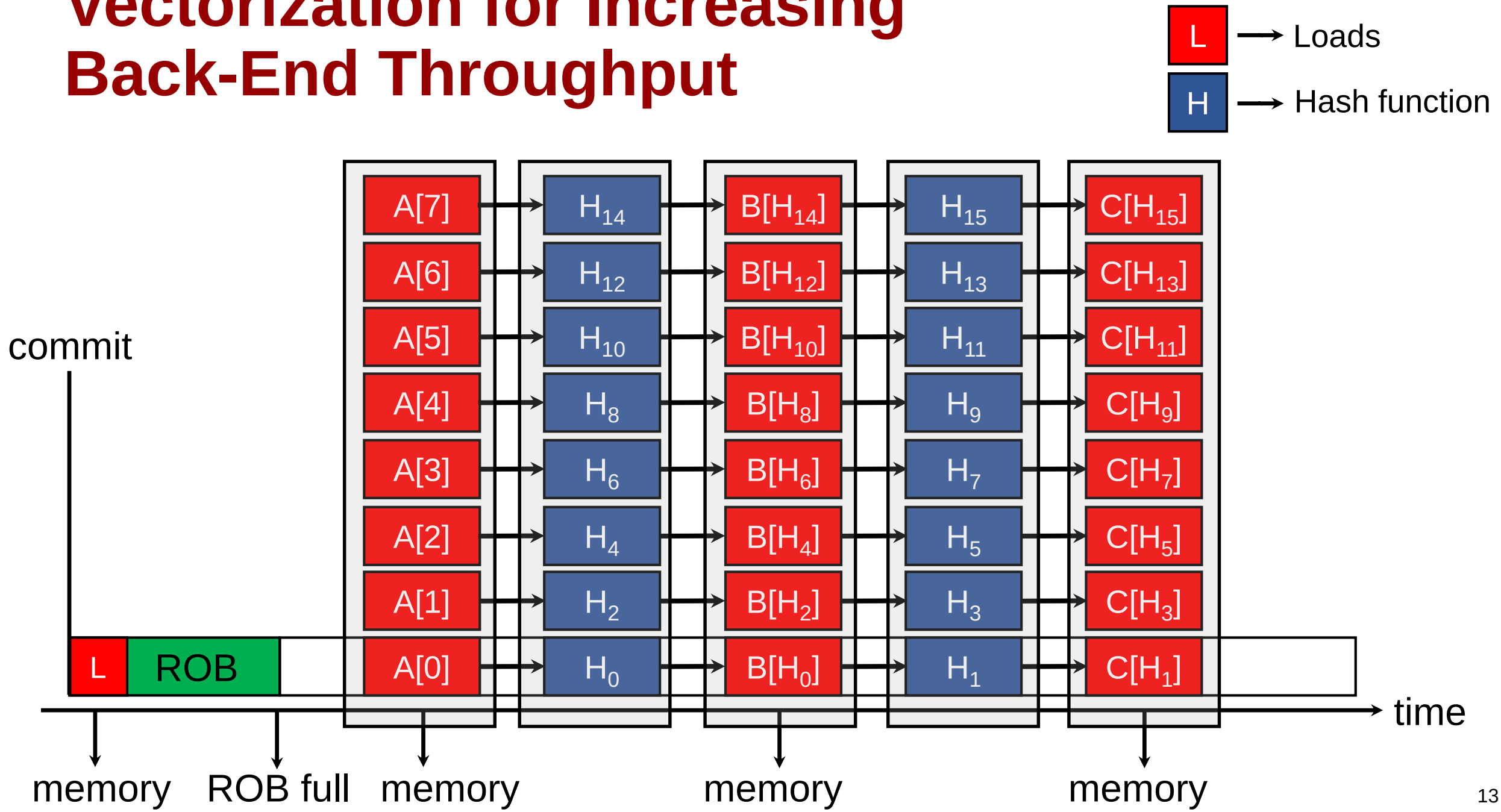
Vectorization for Increasing Back-End Throughput



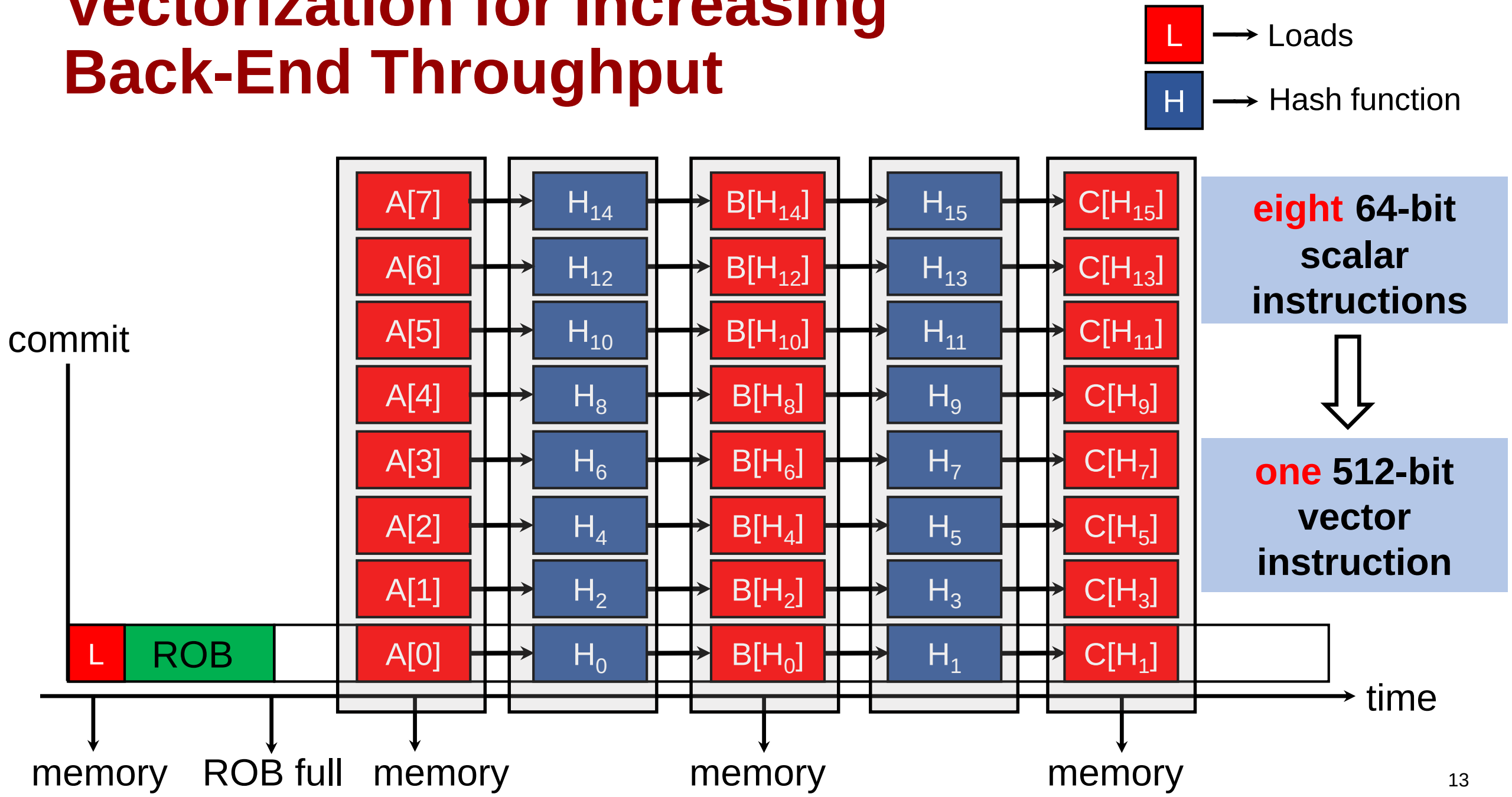
Vectorization for Increasing Back-End Throughput



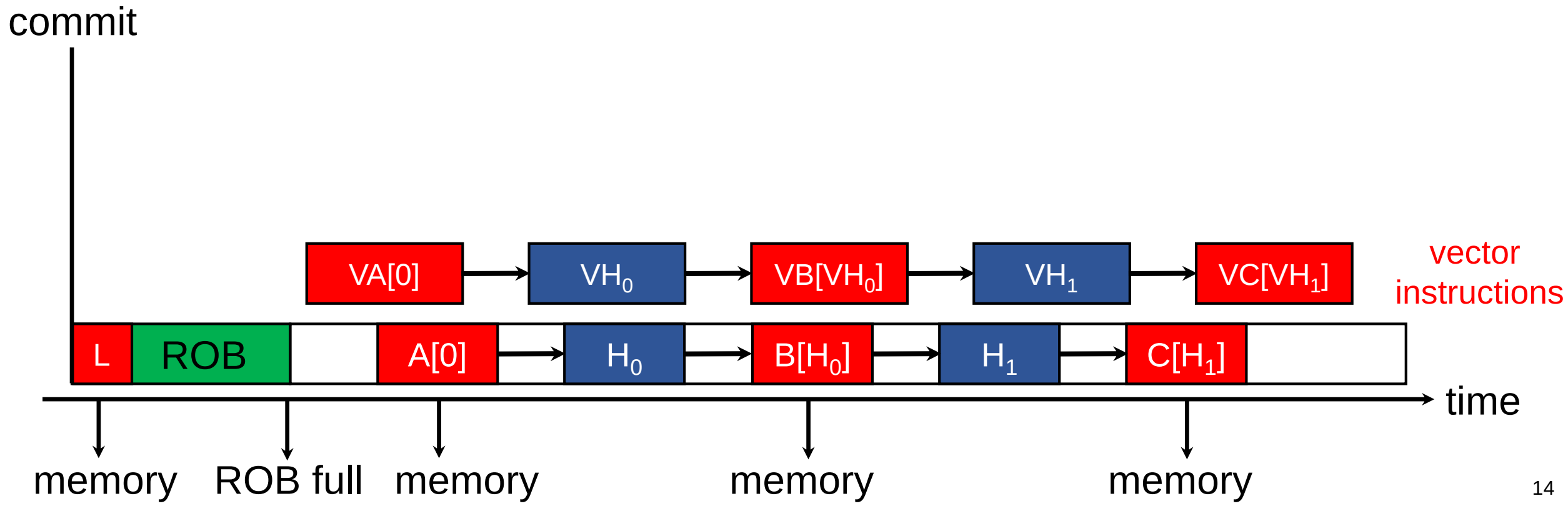
Vectorization for Increasing Back-End Throughput



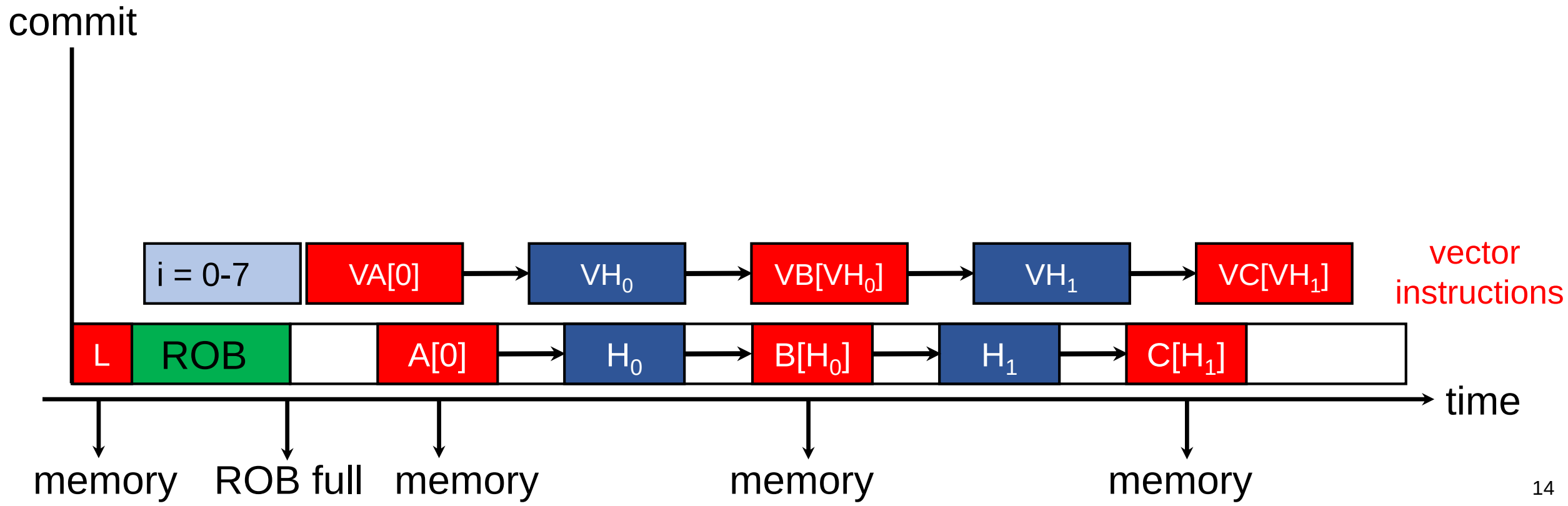
Vectorization for Increasing Back-End Throughput



Vectorization for Increasing Back-End Throughput



Vectorization for Increasing Back-End Throughput



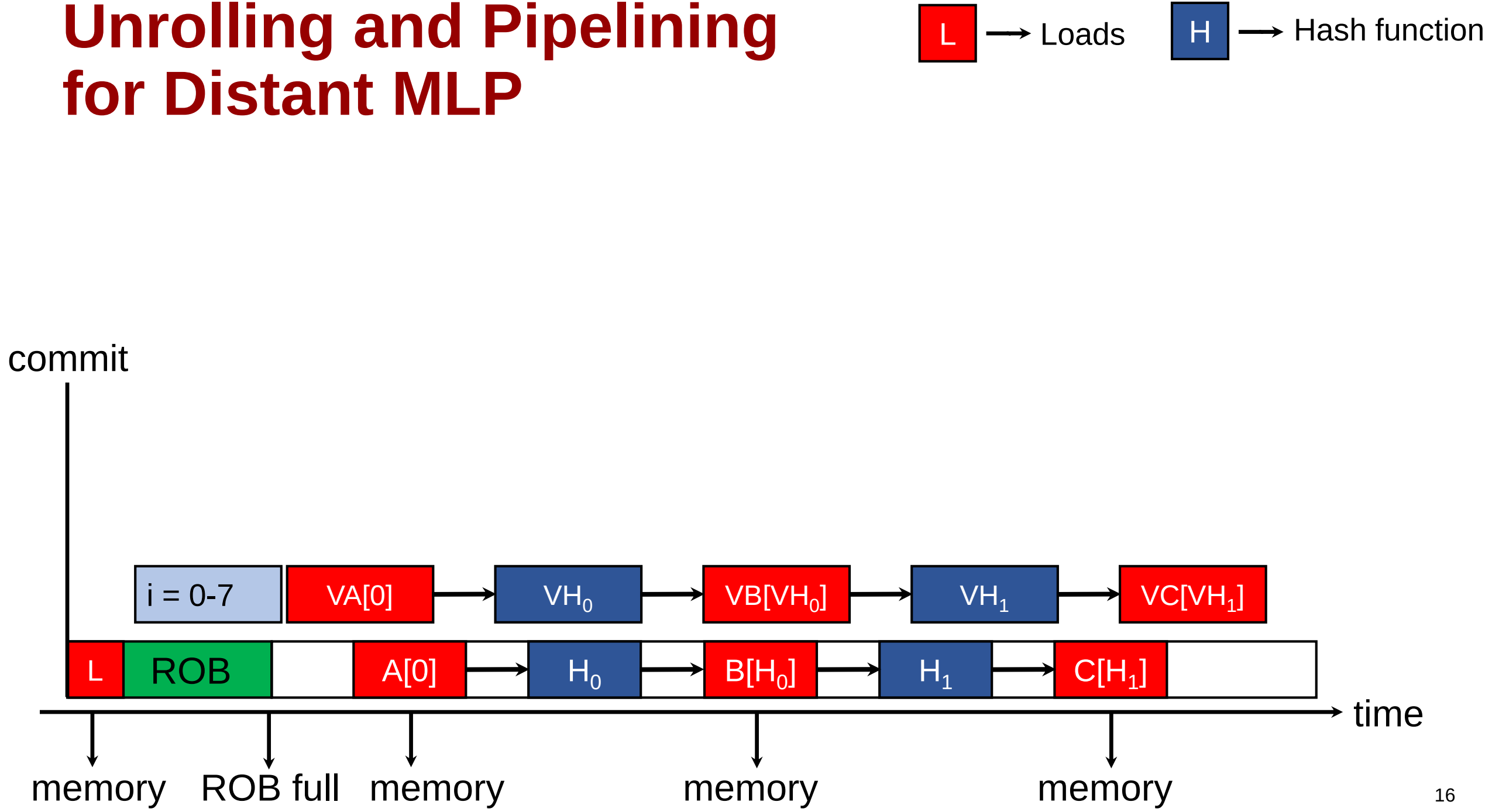
Unrolling and Pipelining for Distant MLP

1. Future instructions are generated automatically
   -- Independent of the front-end bandwidth

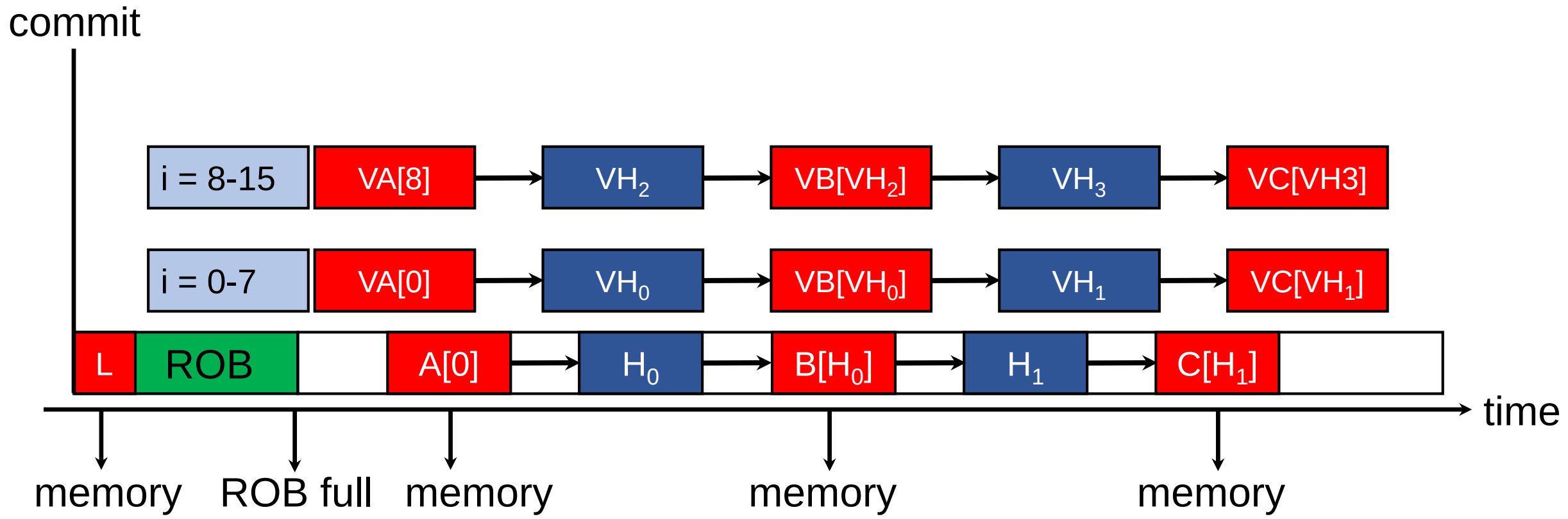
2. Issue more than one vector instructions for each scalar instruction
   -- Unrolling: Dispatch next round of vector instructions after issuing the first

   -- Pipelining: Dispatch multiple vector instructions for each scalar instruction

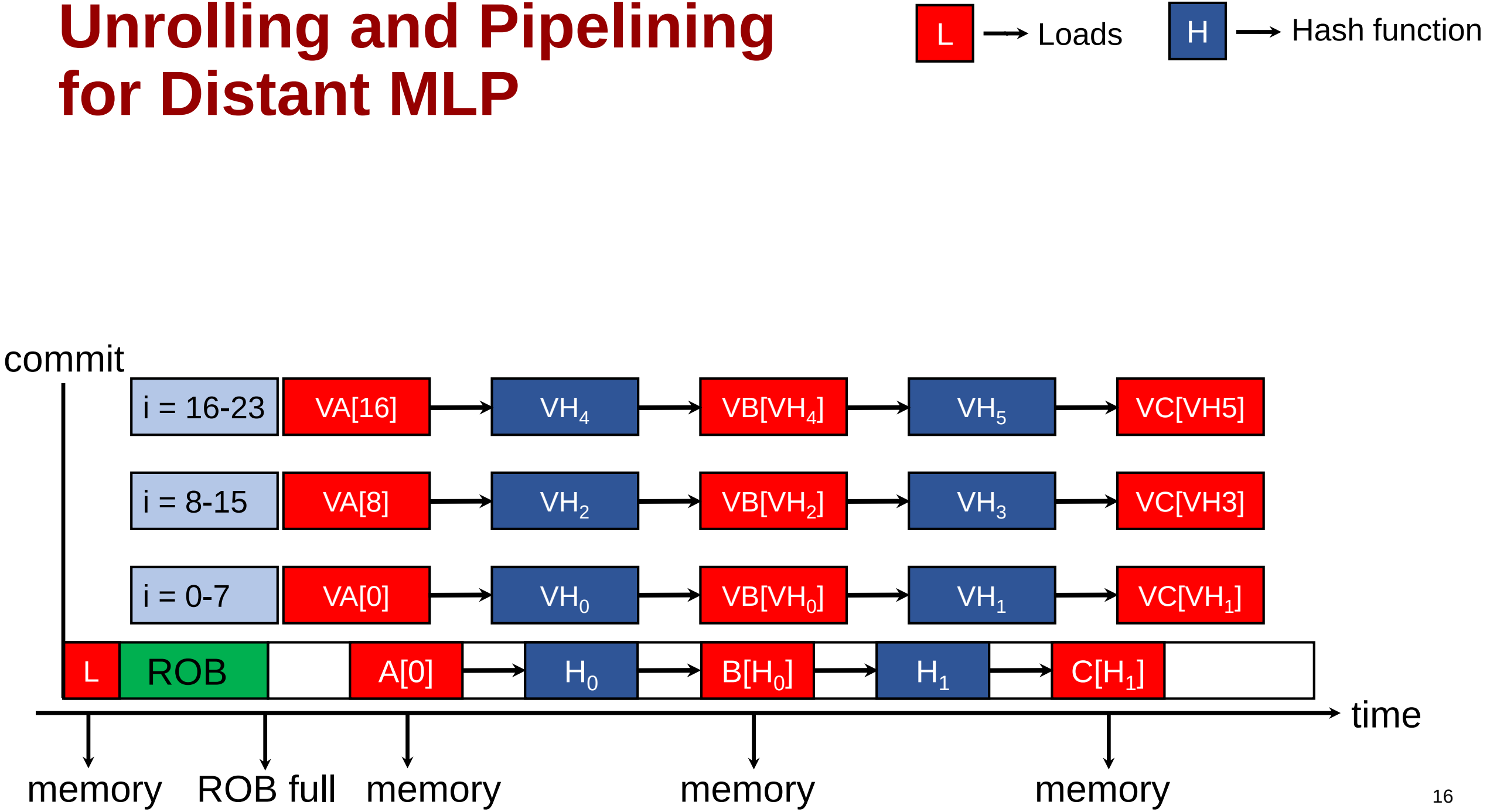
Unrolling and Pipelining for Distant MLP



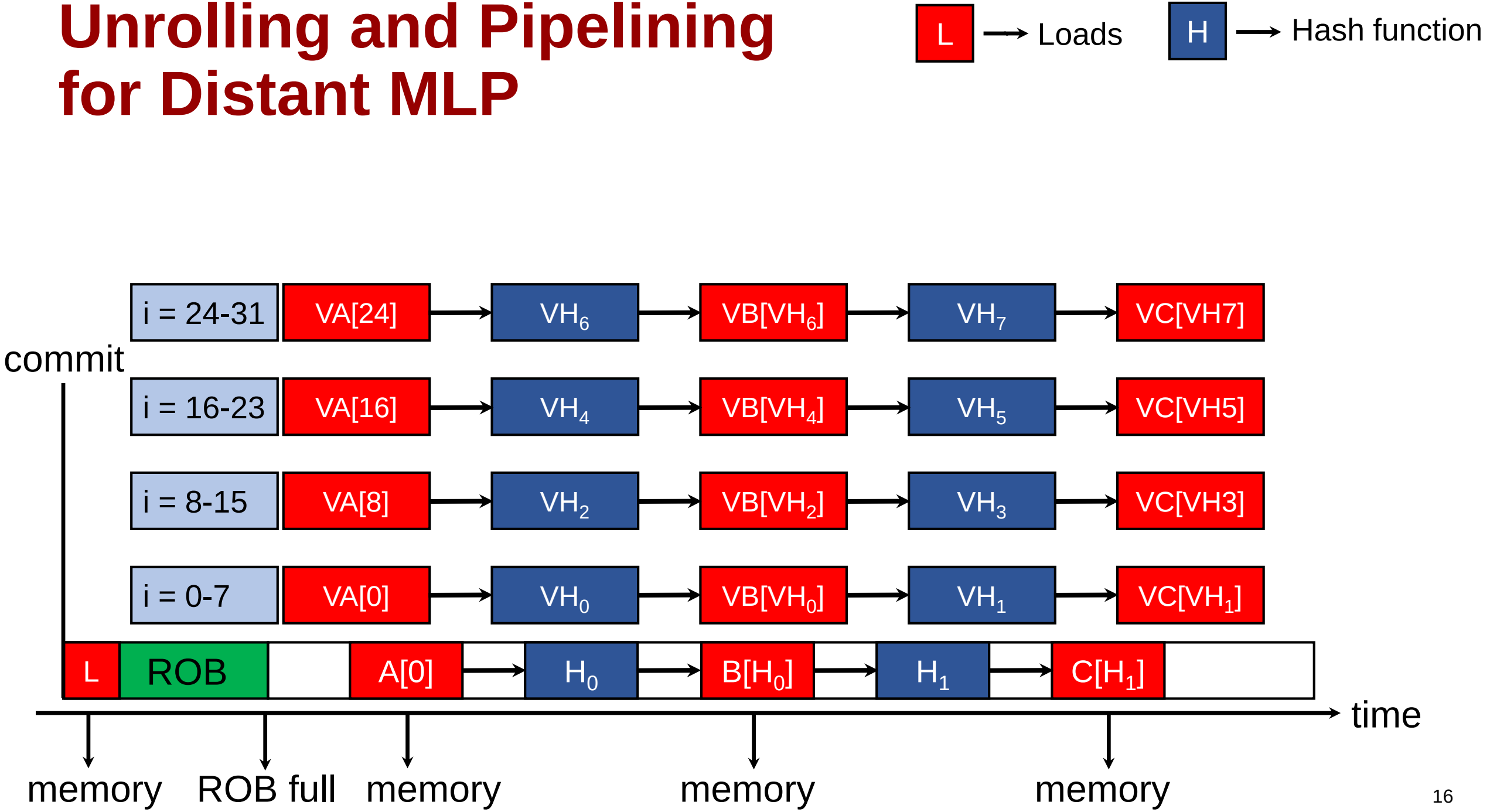
Unrolling and Pipelining for Distant MLP



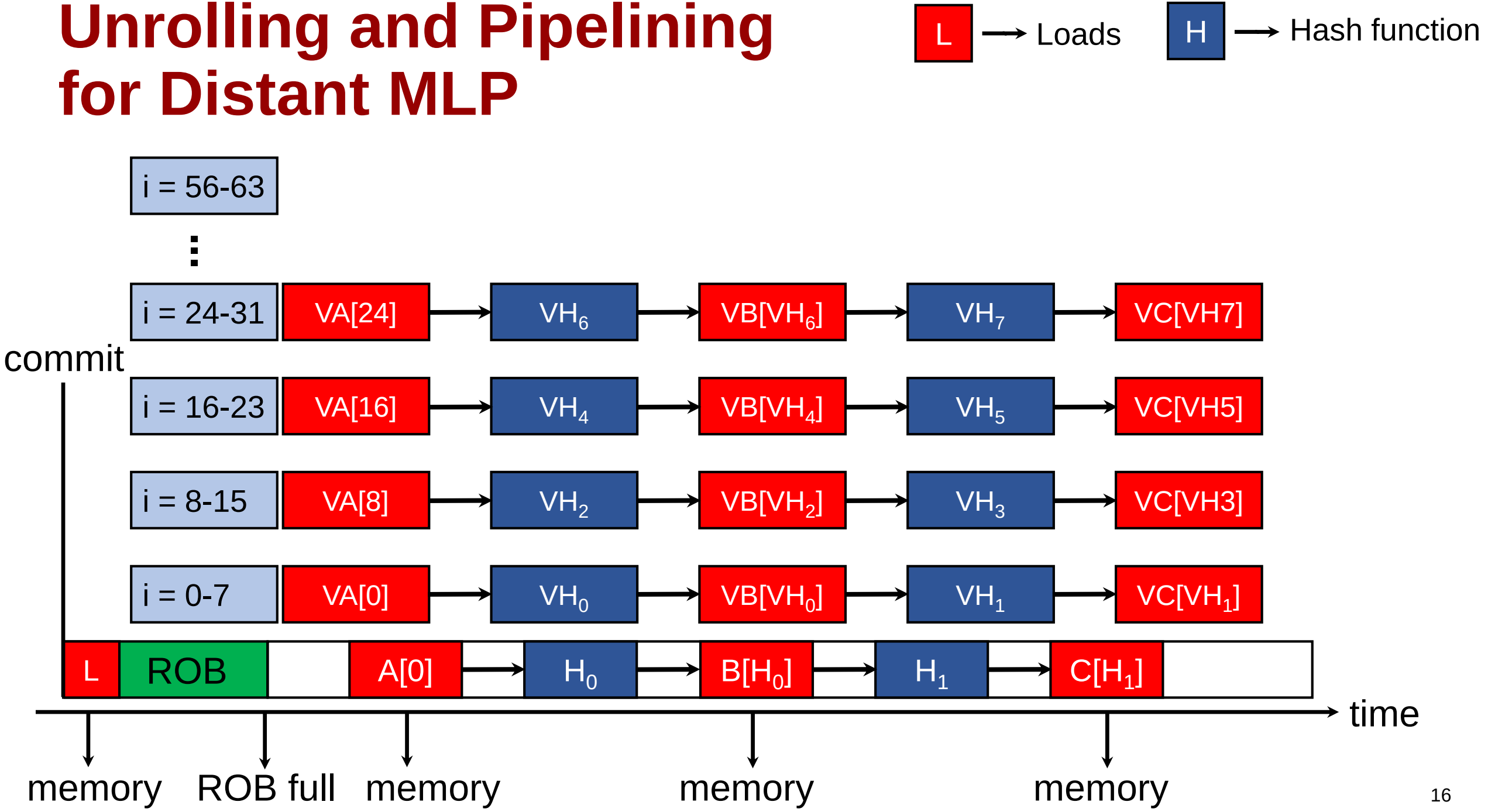
Unrolling and Pipelining for Distant MLP



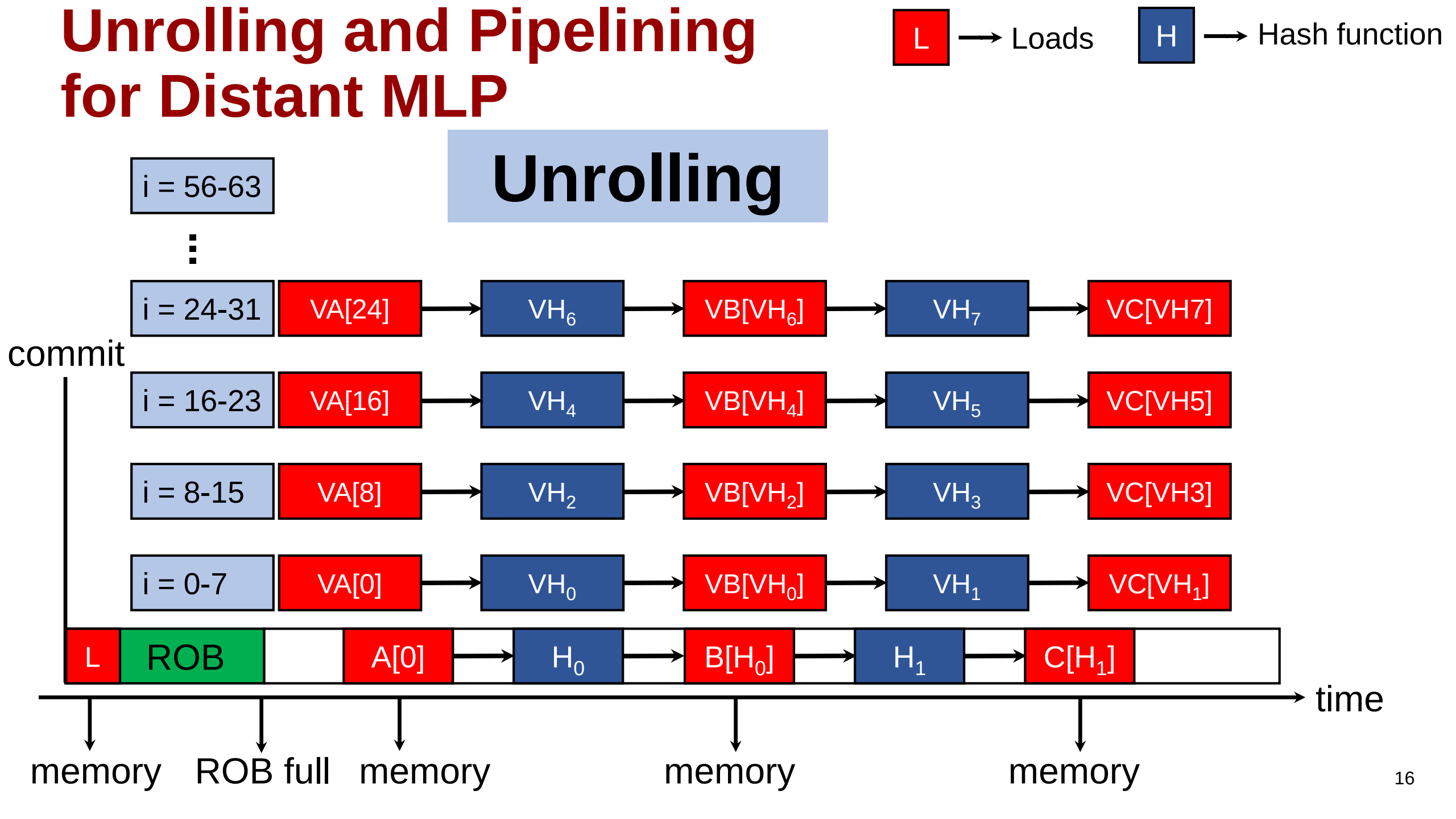
Unrolling and Pipelining for Distant MLP



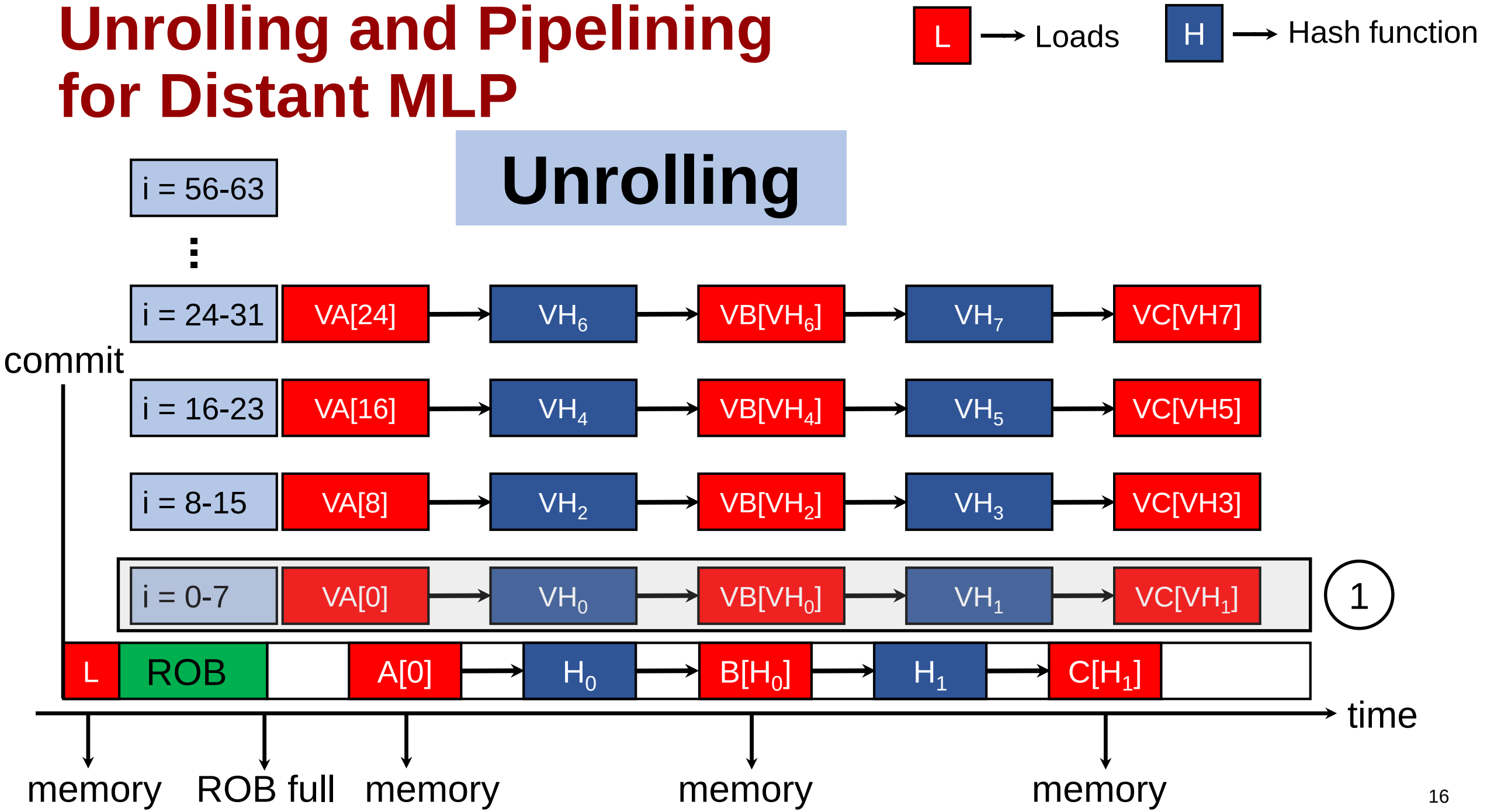
Unrolling and Pipelining for Distant MLP



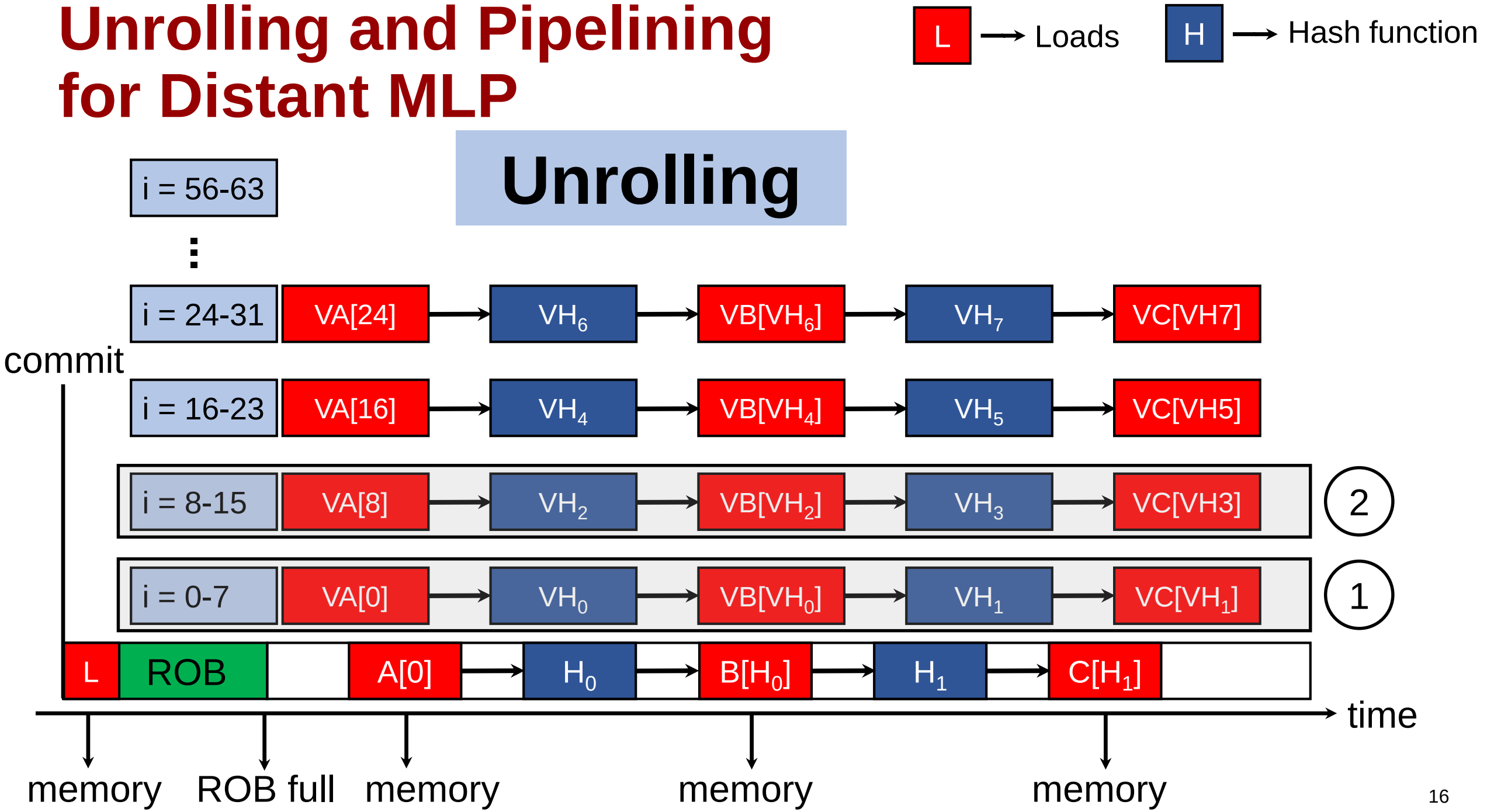
Unrolling and Pipelining for Distant MLP



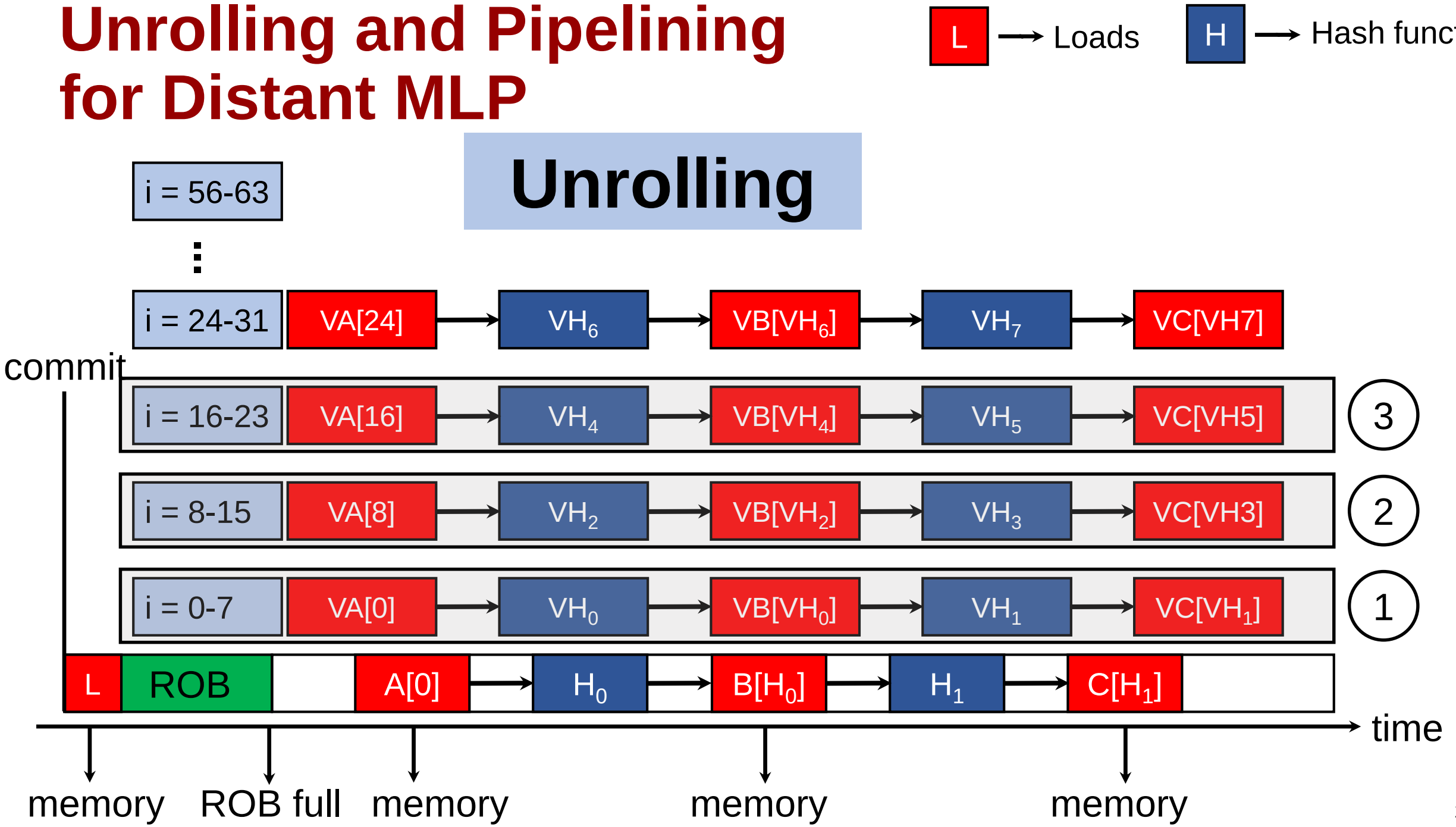
Unrolling and Pipelining for Distant MLP



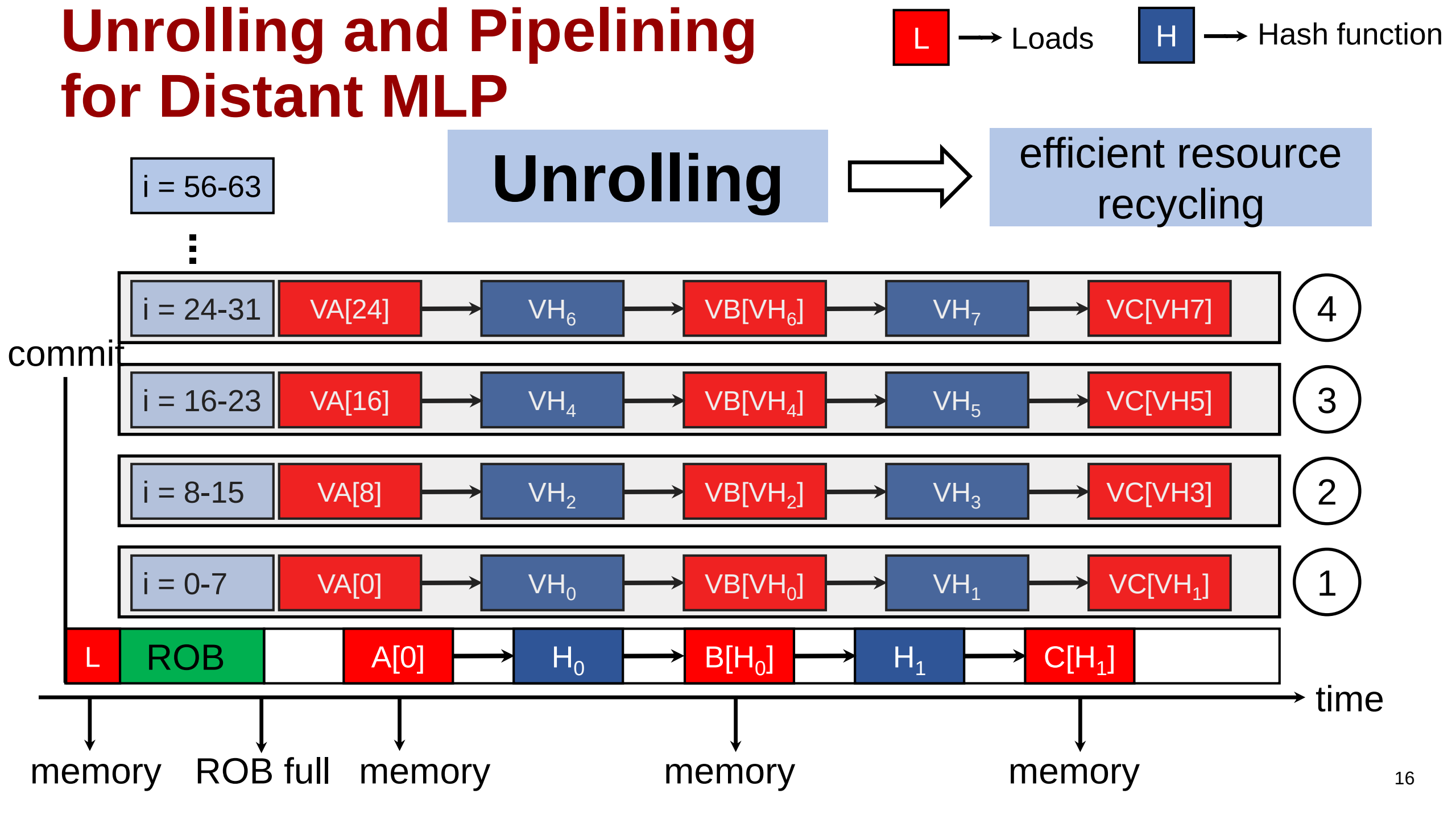
Unrolling and Pipelining for Distant MLP



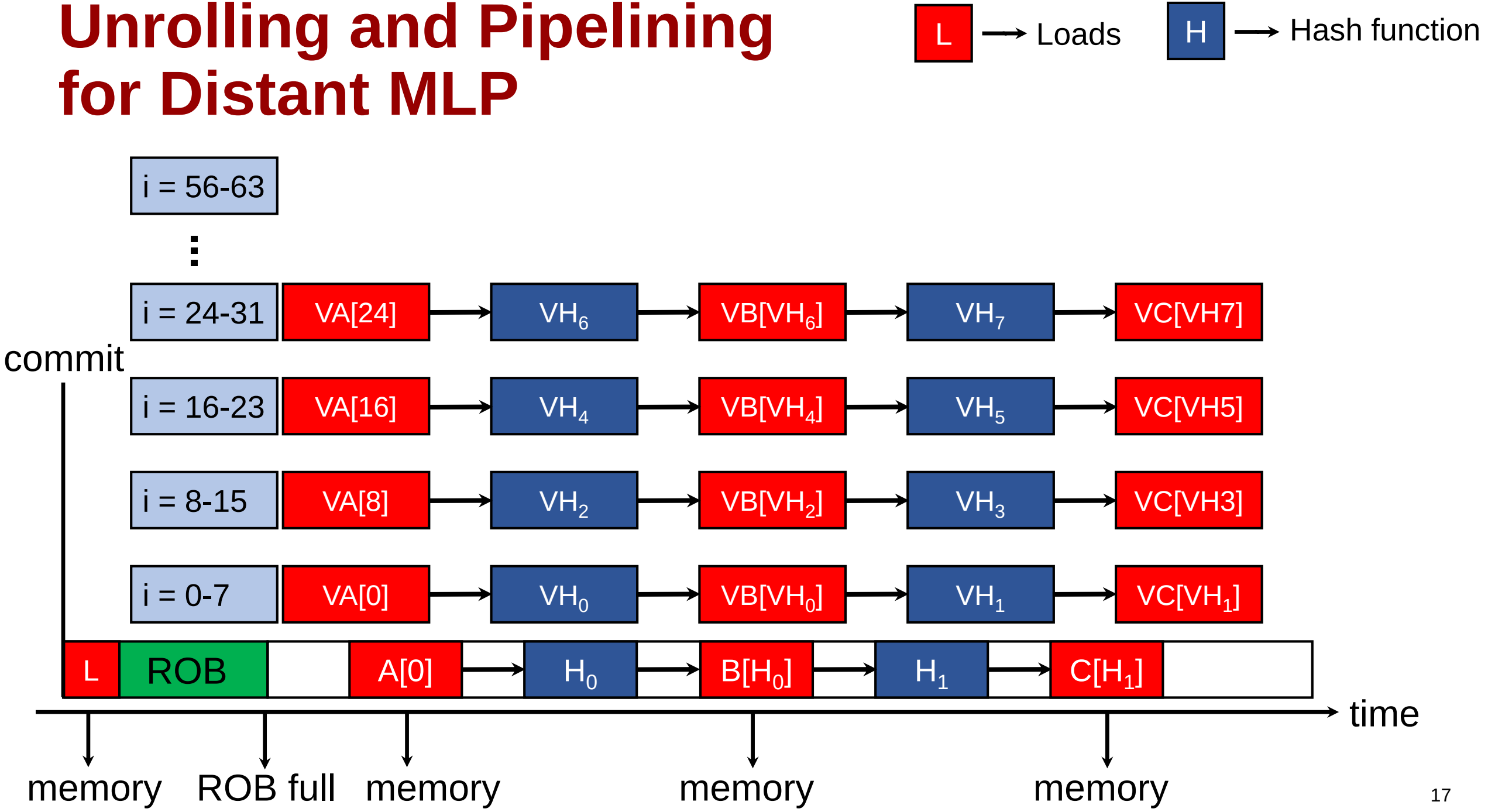
Unrolling and Pipelining for Distant MLP



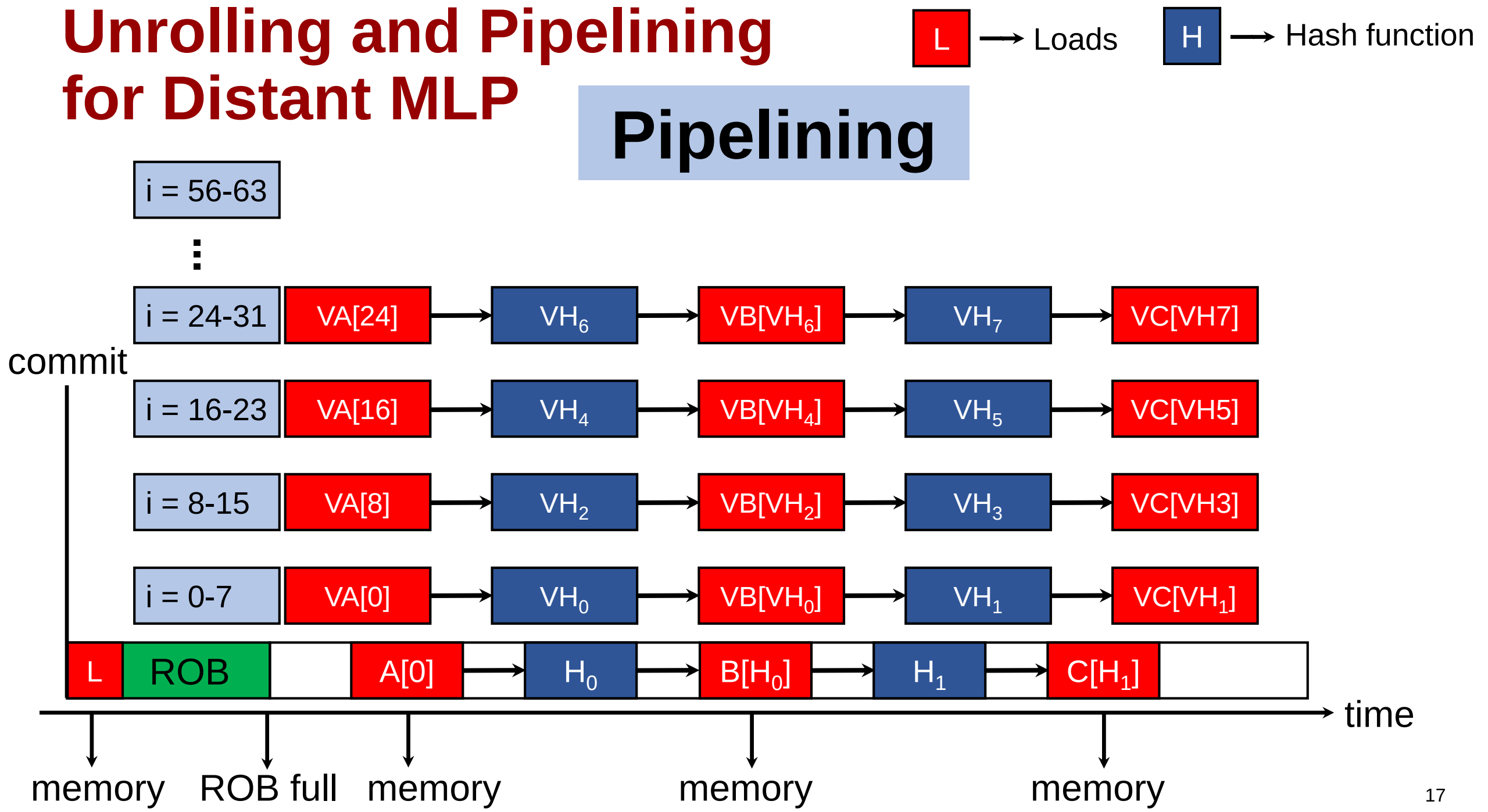
Unrolling and Pipelining for Distant MLP



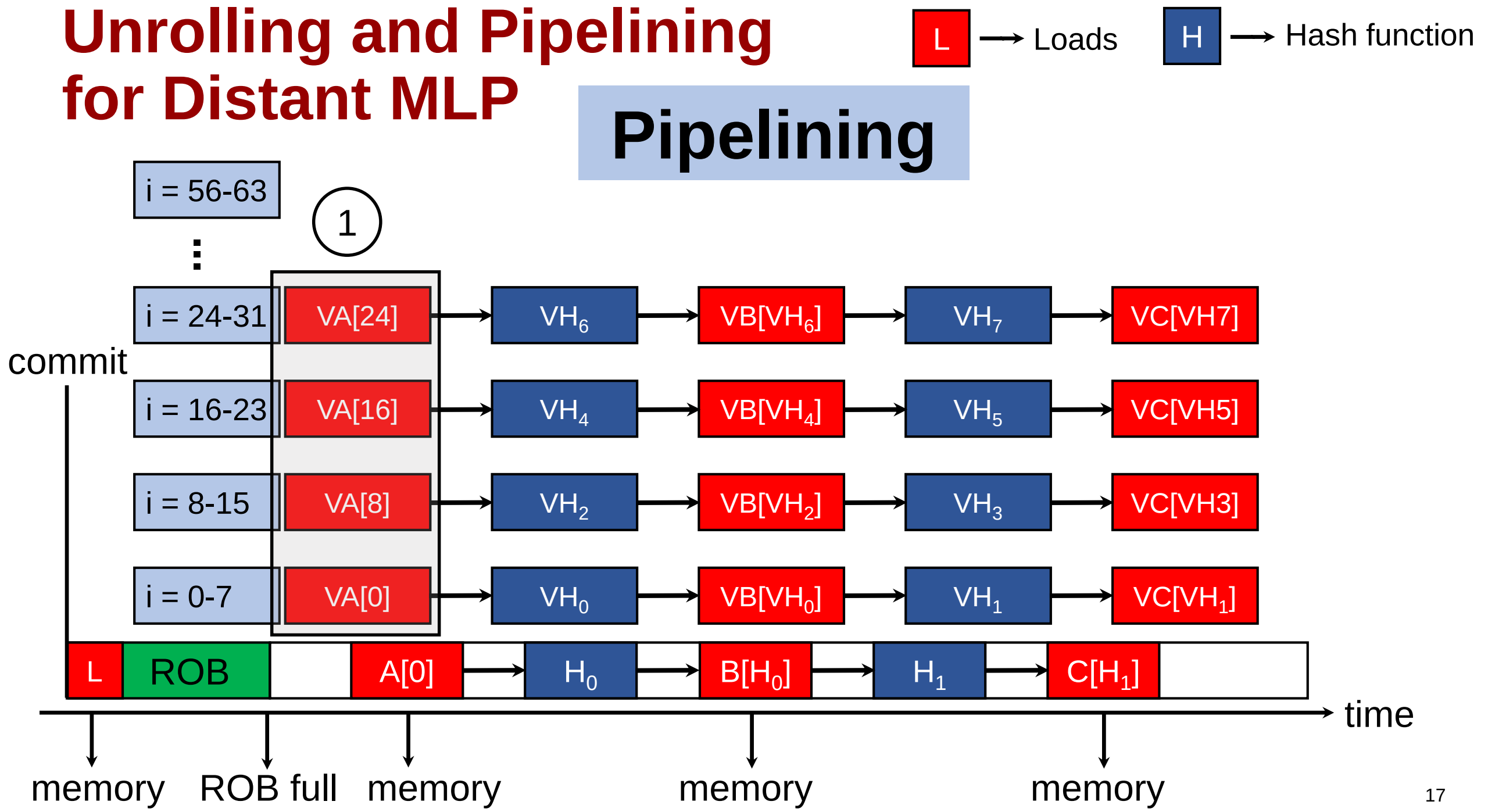
Unrolling and Pipelining for Distant MLP



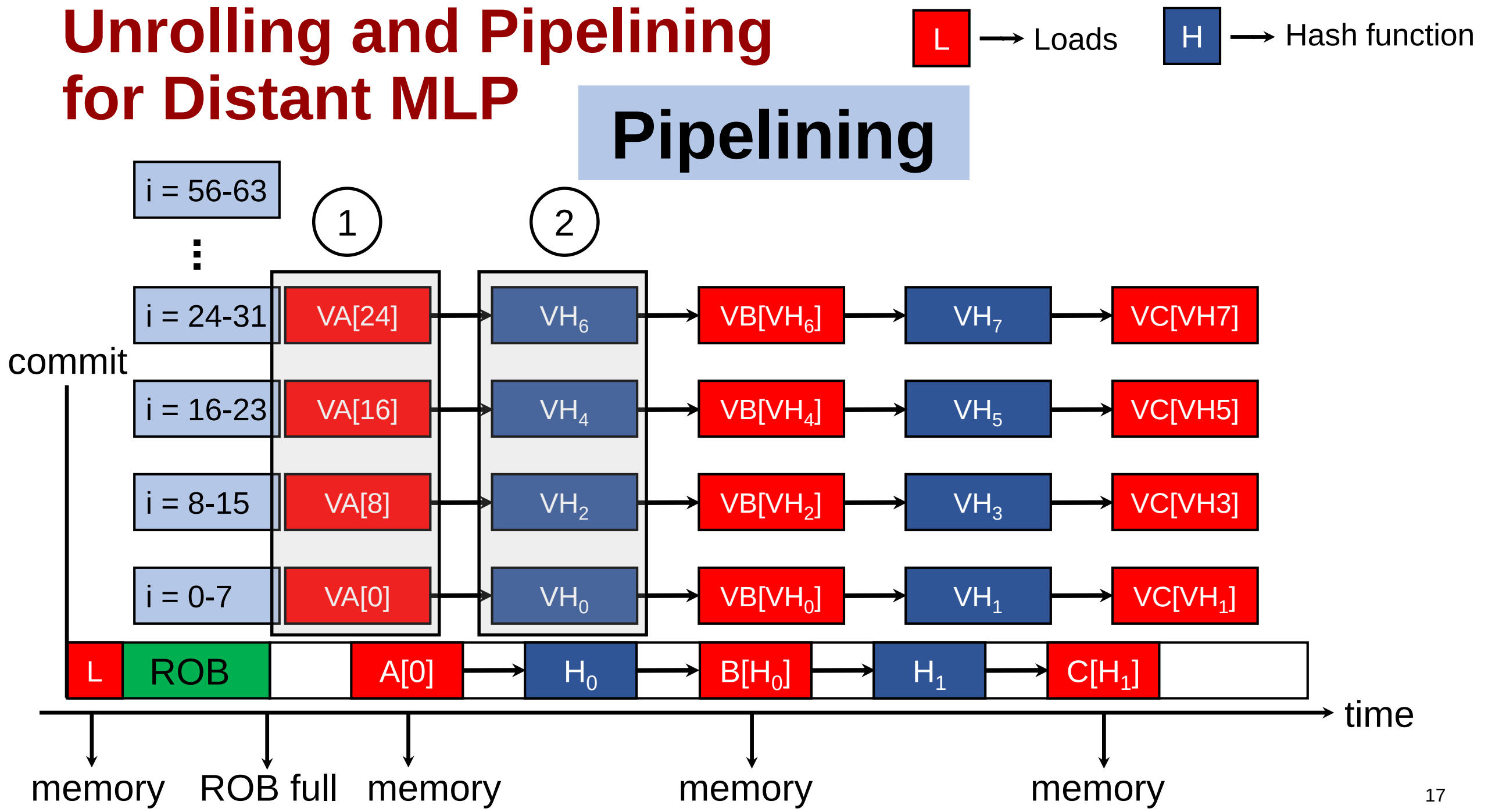
Unrolling and Pipelining for Distant MLP



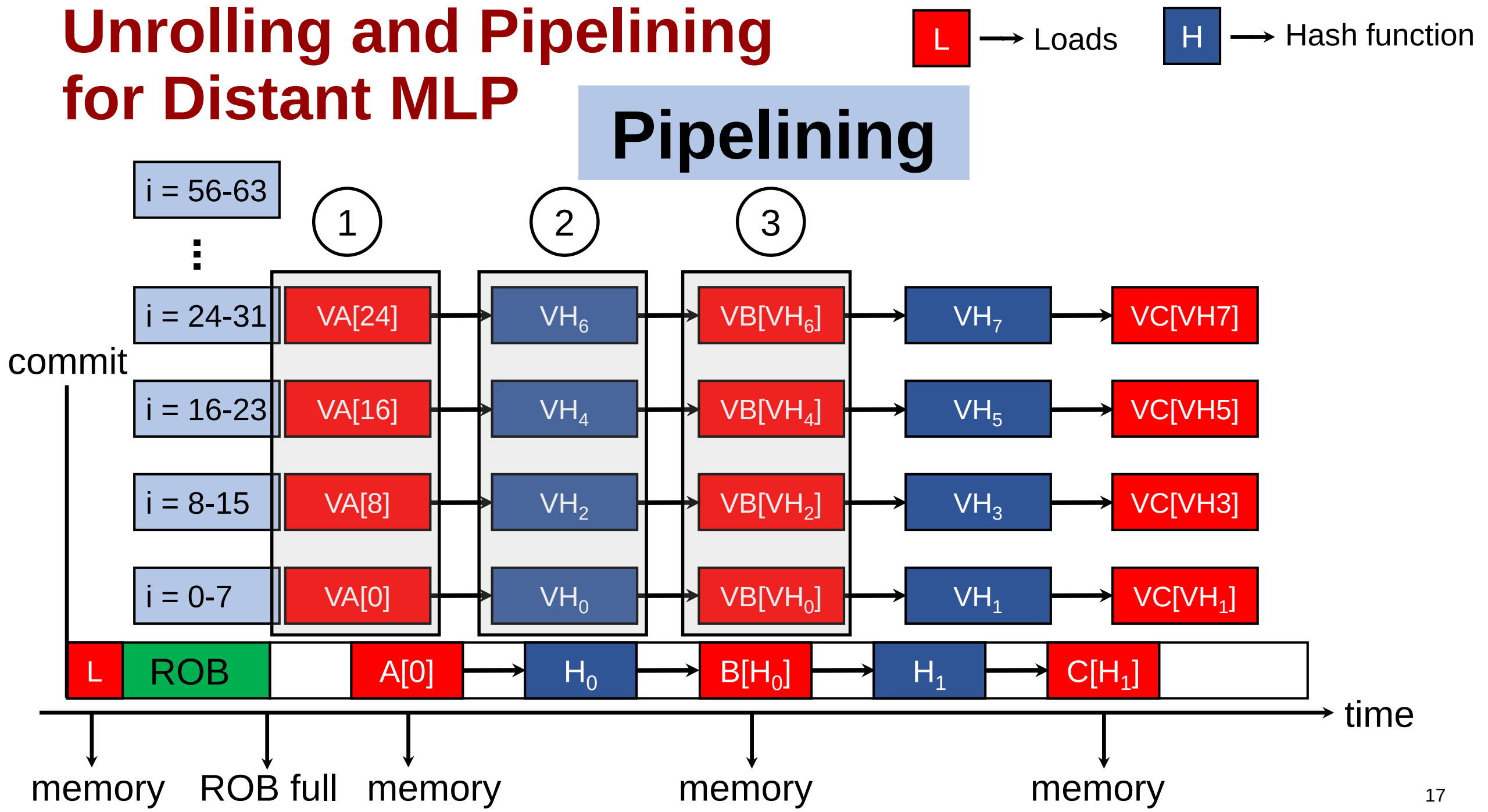
Unrolling and Pipelining for Distant MLP



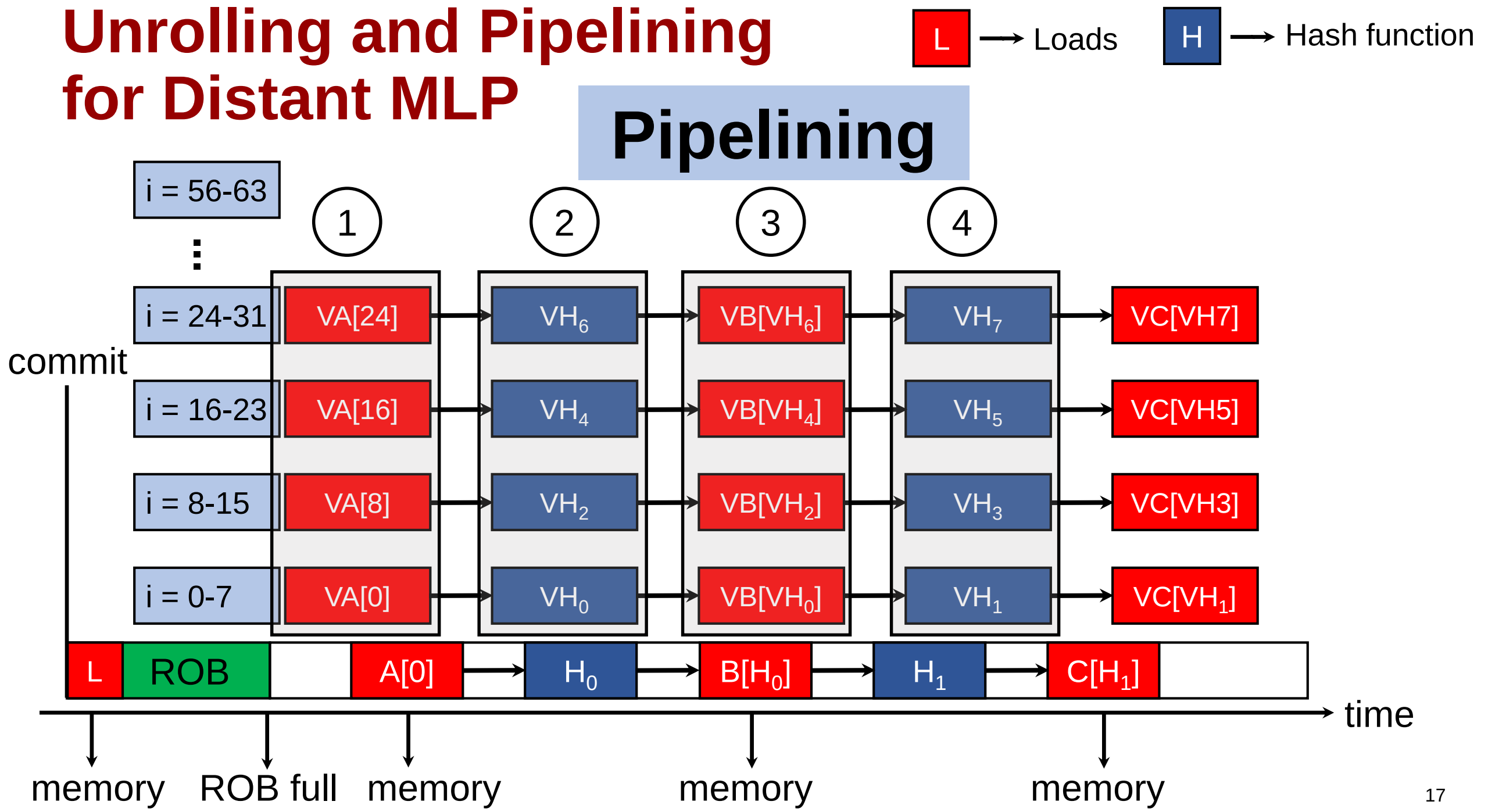
Unrolling and Pipelining for Distant MLP



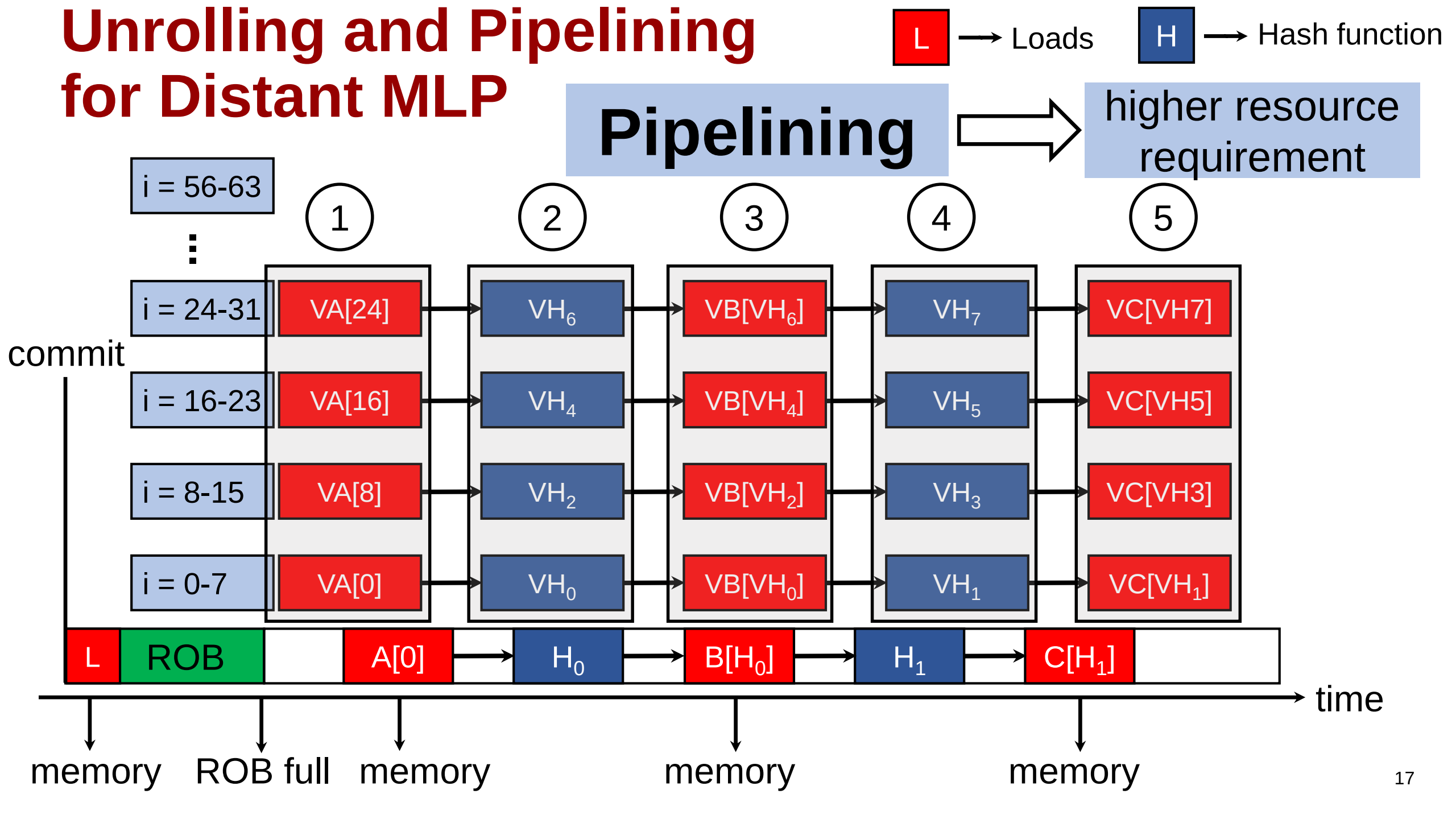
Unrolling and Pipelining for Distant MLP



Unrolling and Pipelining for Distant MLP

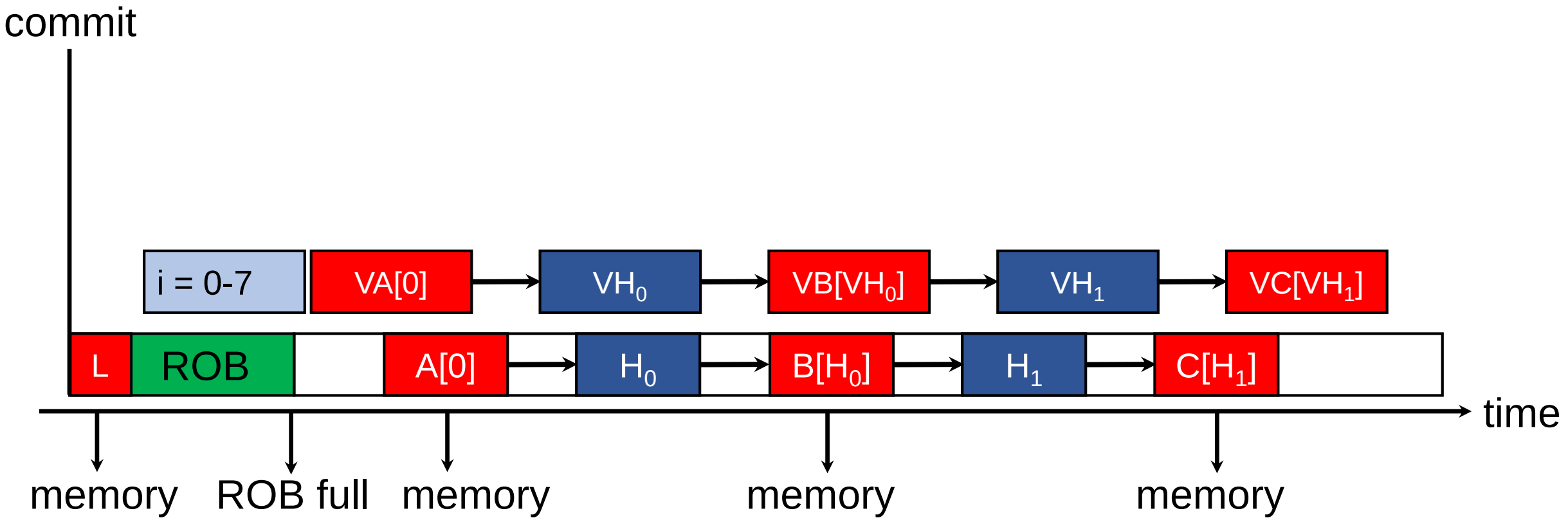


Unrolling and Pipelining for Distant MLP



Delayed Termination in Vector Runahead

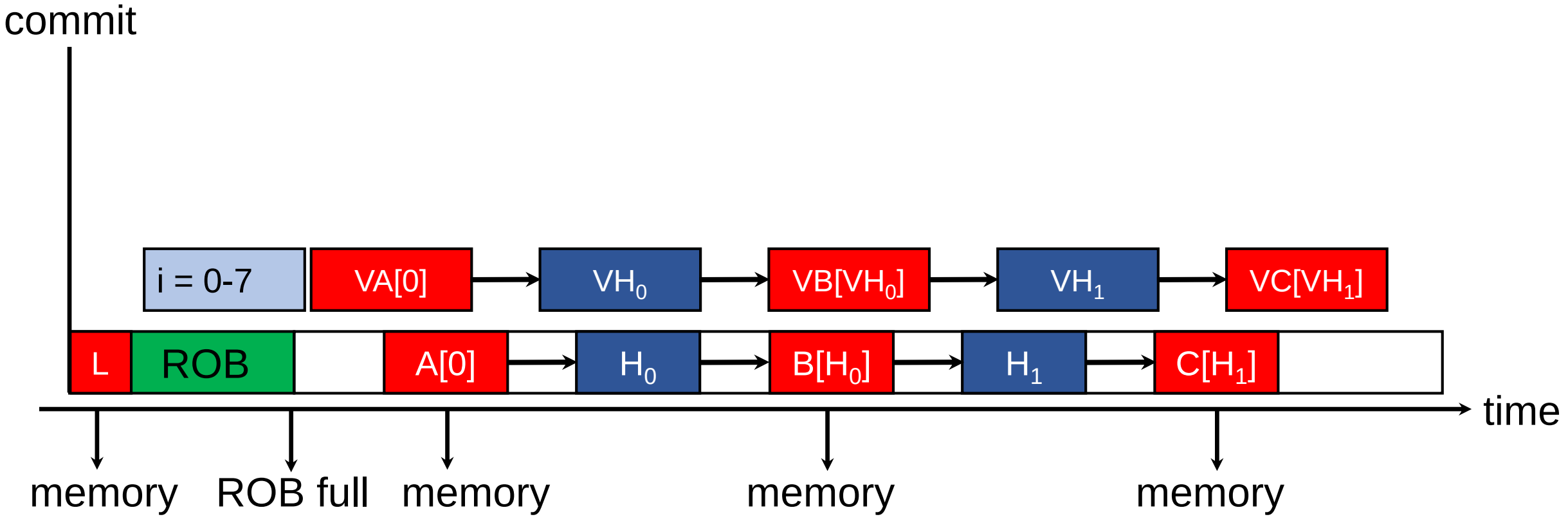
When to terminate runahead mode?



Delayed Termination in Vector Runahead

When to terminate runahead mode?

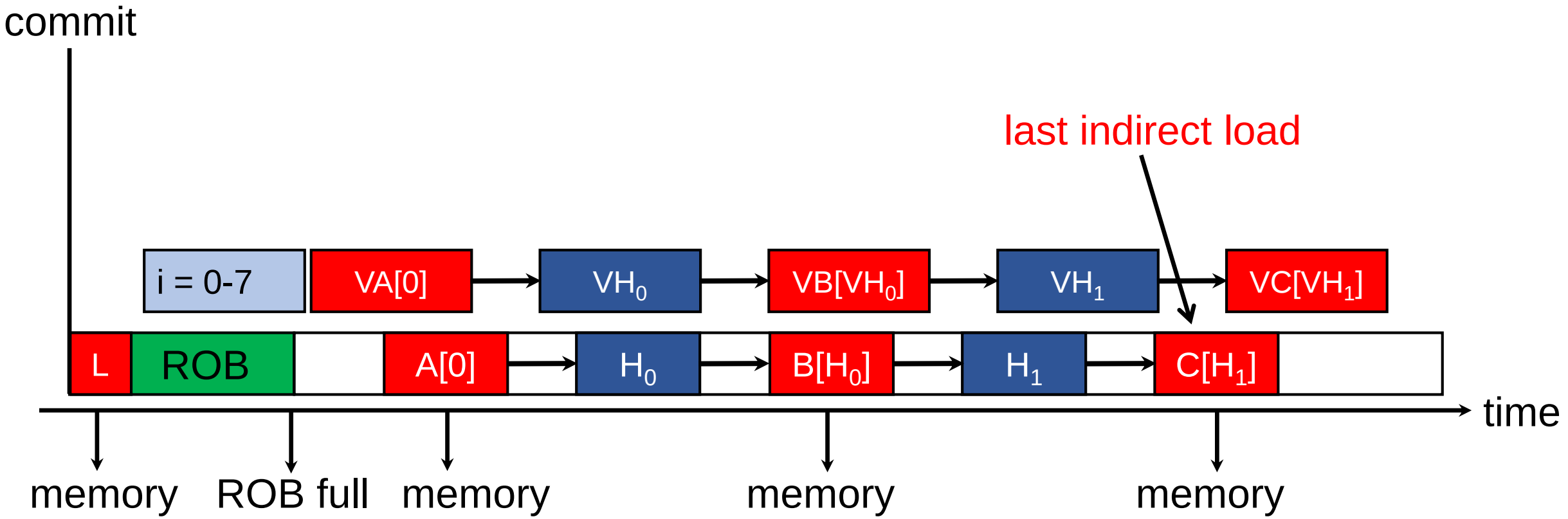
Last indirect load: Identified with vector taint tracking



Delayed Termination in Vector Runahead

When to terminate runahead mode?

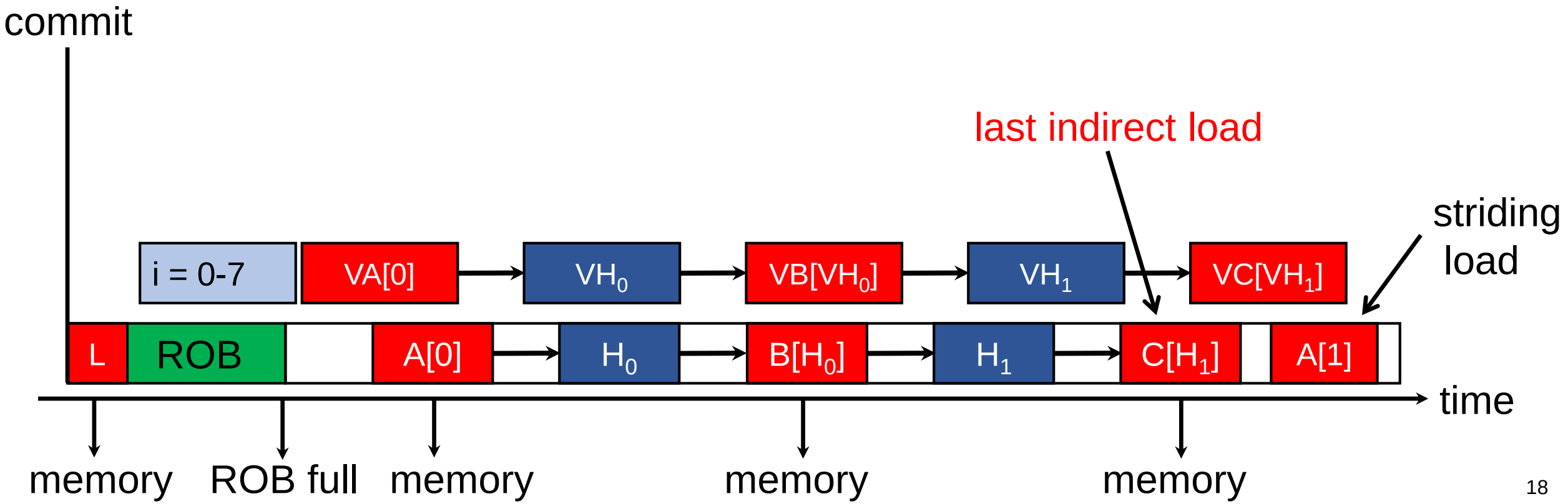
Last indirect load: Identified with vector taint tracking



Delayed Termination in Vector Runahead

When to terminate runahead mode?

Last indirect load: Identified with vector taint tracking

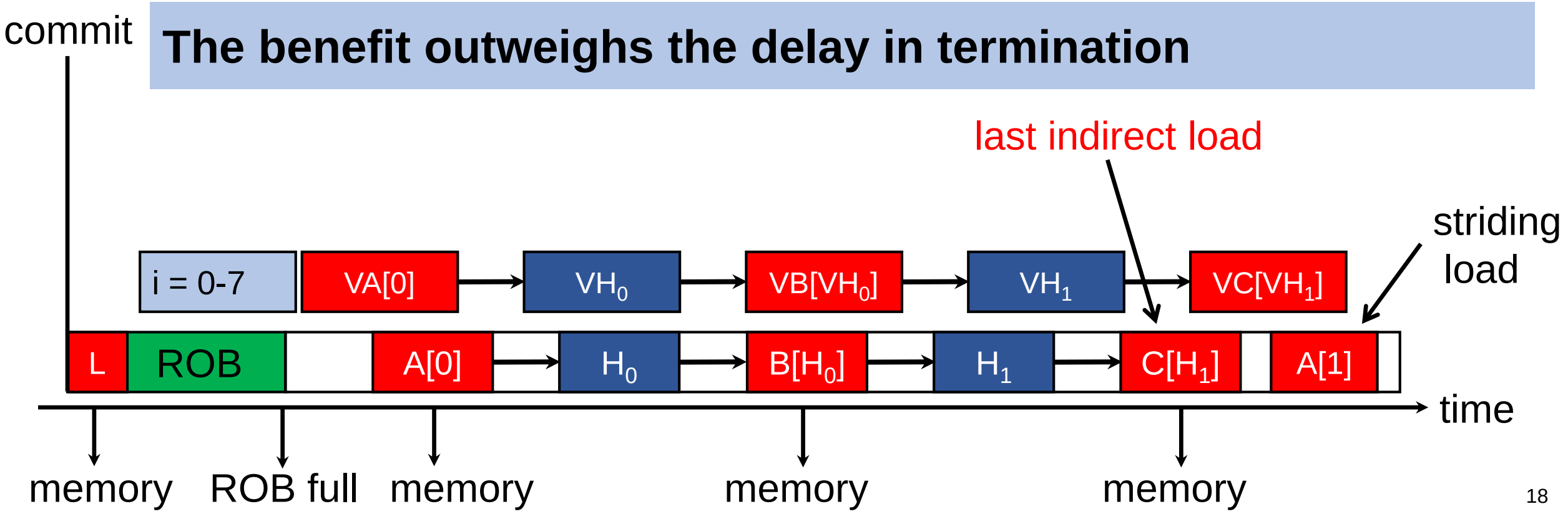


Delayed Termination in Vector Runahead

When to terminate runahead mode?

Last indirect load: Identified with vector taint tracking

The benefit outweighs the delay in termination



Register Renaming in Vector Runahead

Register Renaming in Vector Runahead

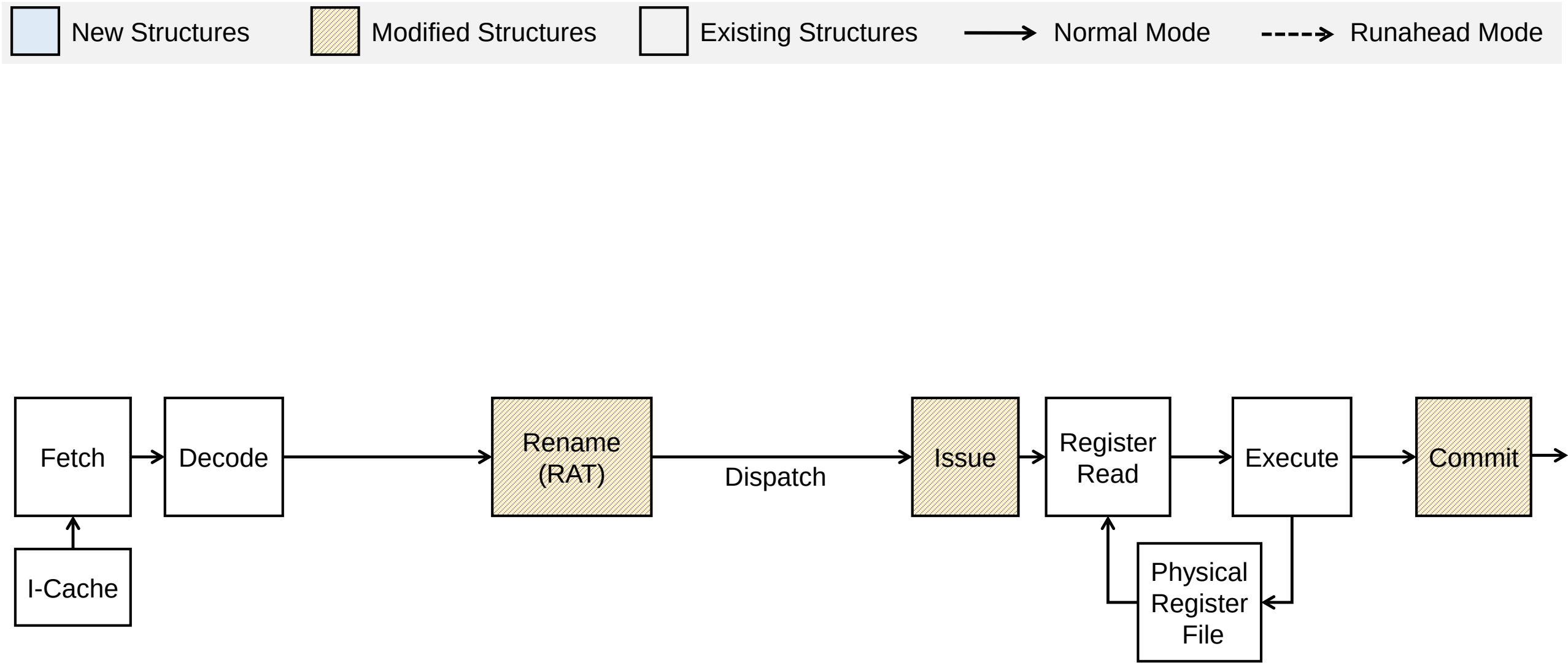
1. **Allocation:** Scalar architectural registers are renamed to vector physical registers using Vector Register Allocation Table (VRAT)

Register Renaming in Vector Runahead

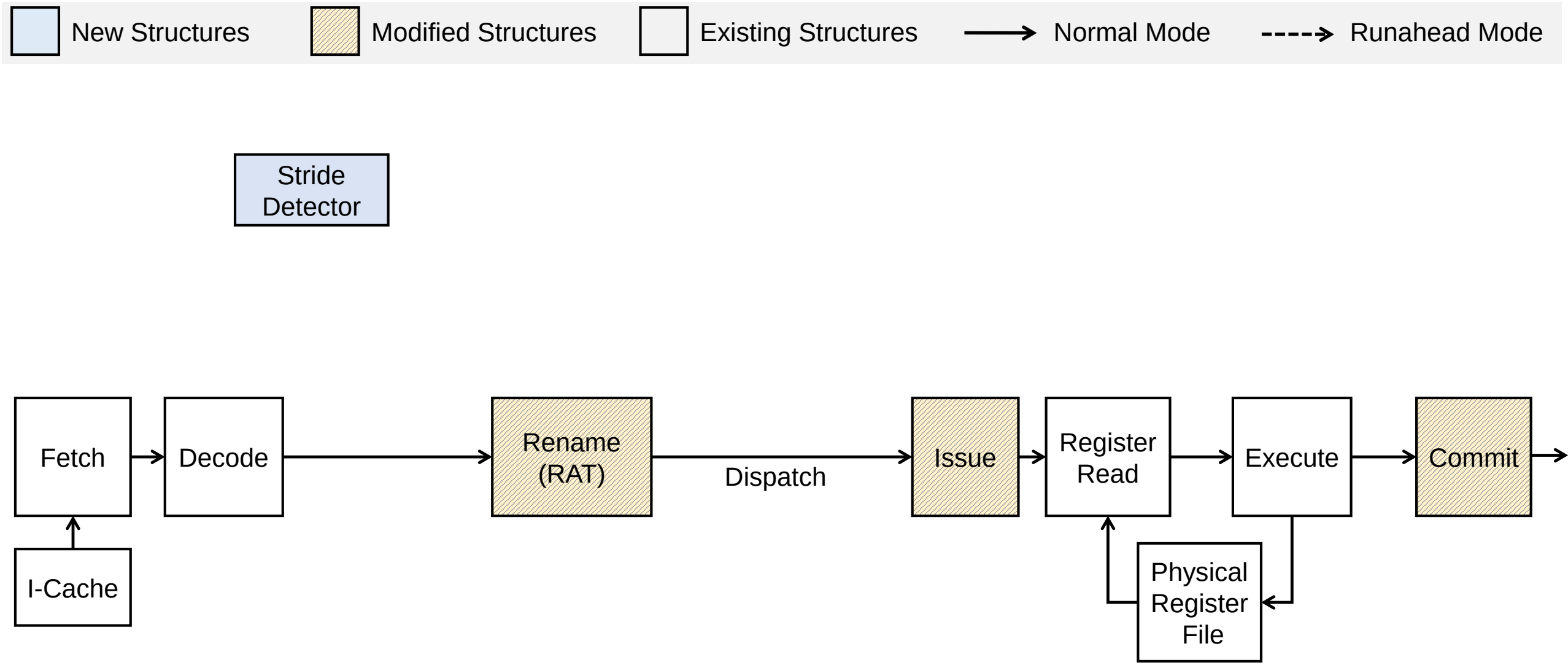
1. **Allocation:** Scalar architectural registers are renamed to vector physical registers using Vector Register Allocation Table (VRAT)

2. **Deallocation:** Like Precise Runahead Using Register Deallocation Queue (RDQ)

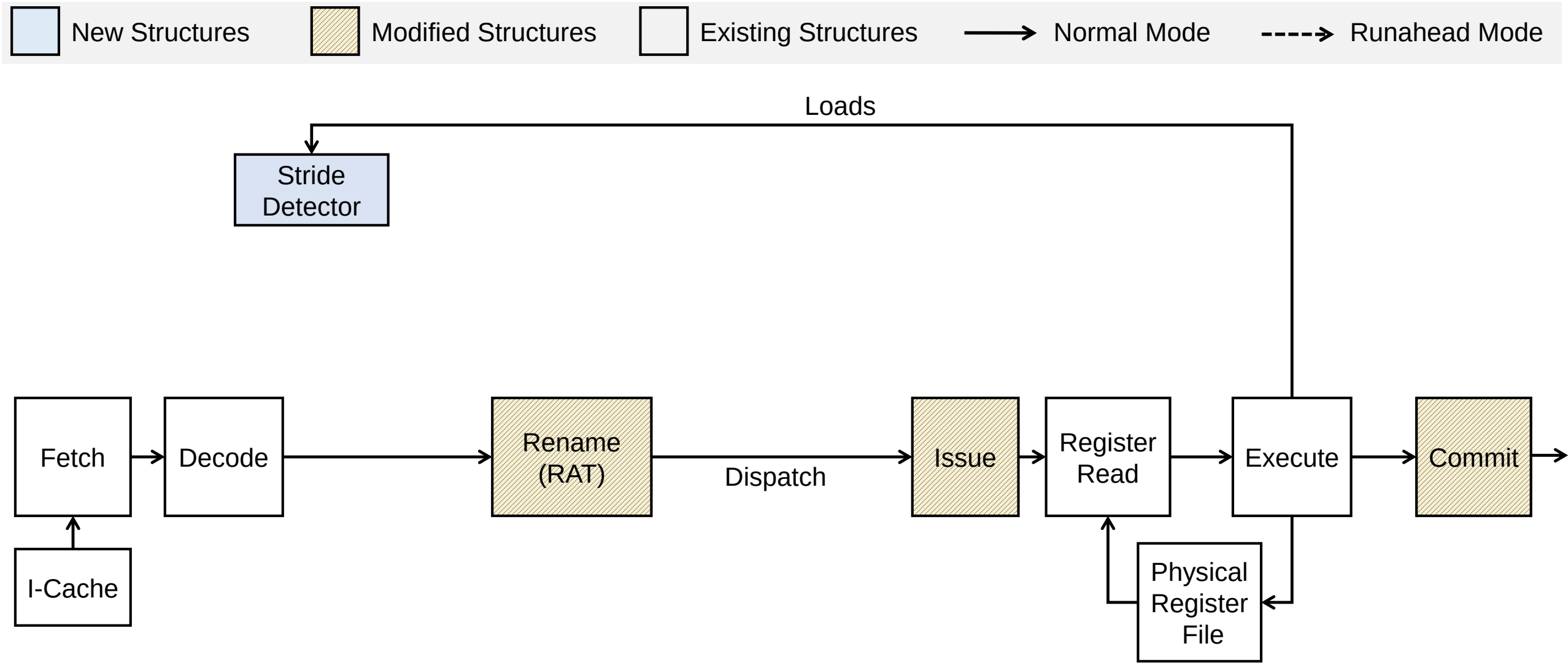
Vector Runahead Microarchitecture



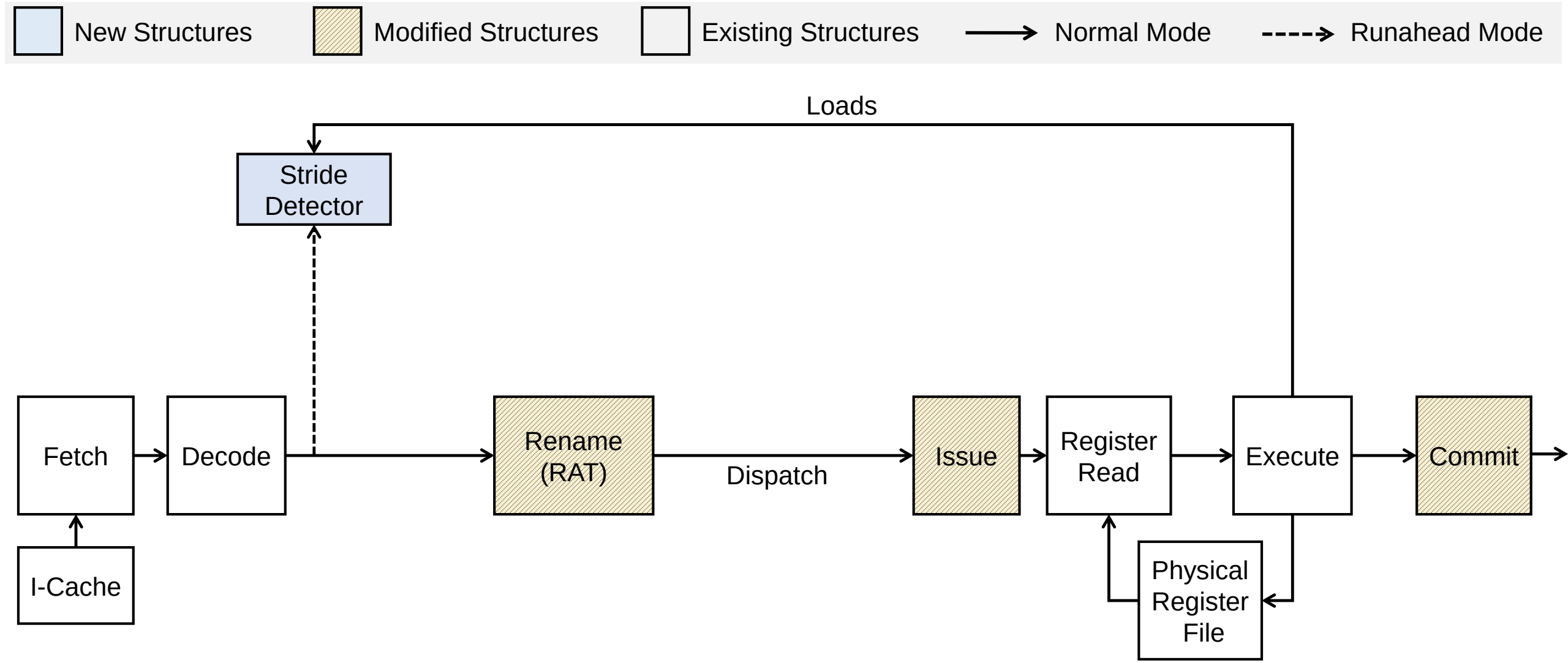
Vector Runahead Microarchitecture



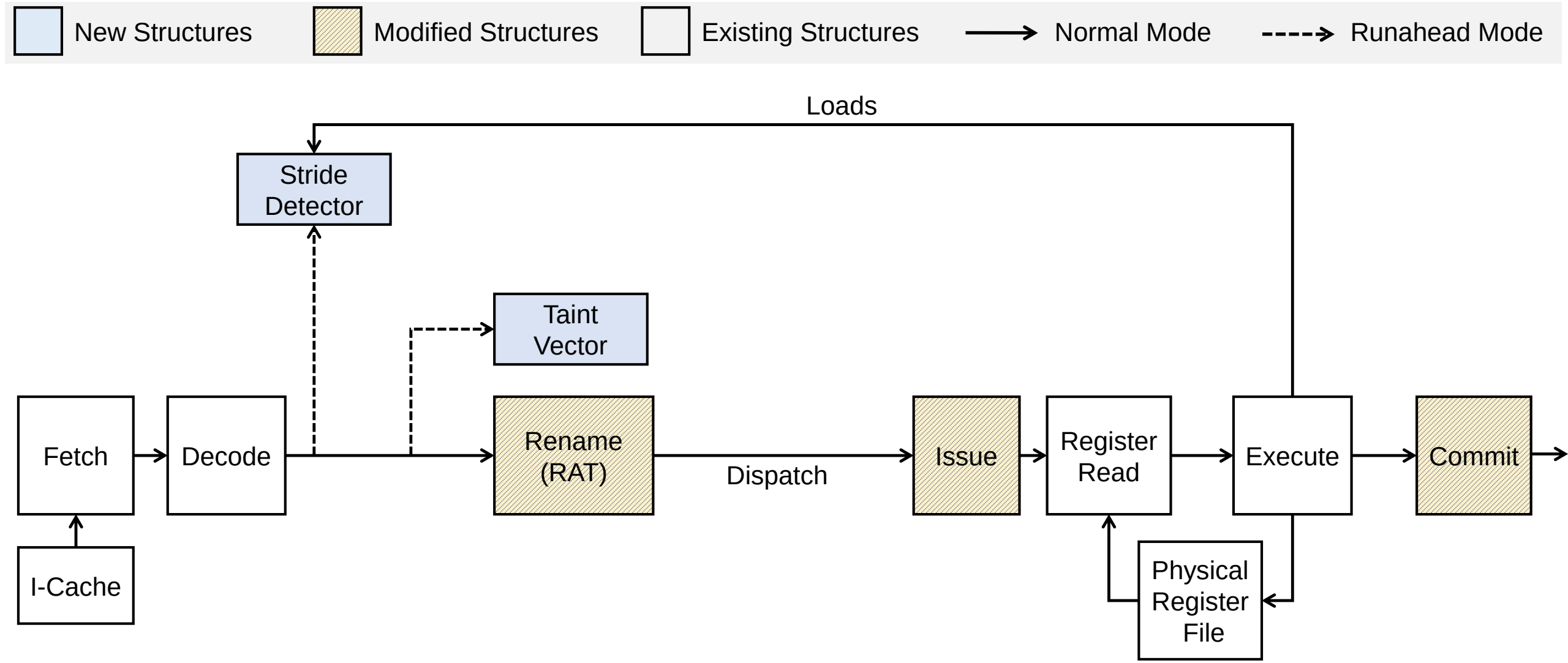
Vector Runahead Microarchitecture



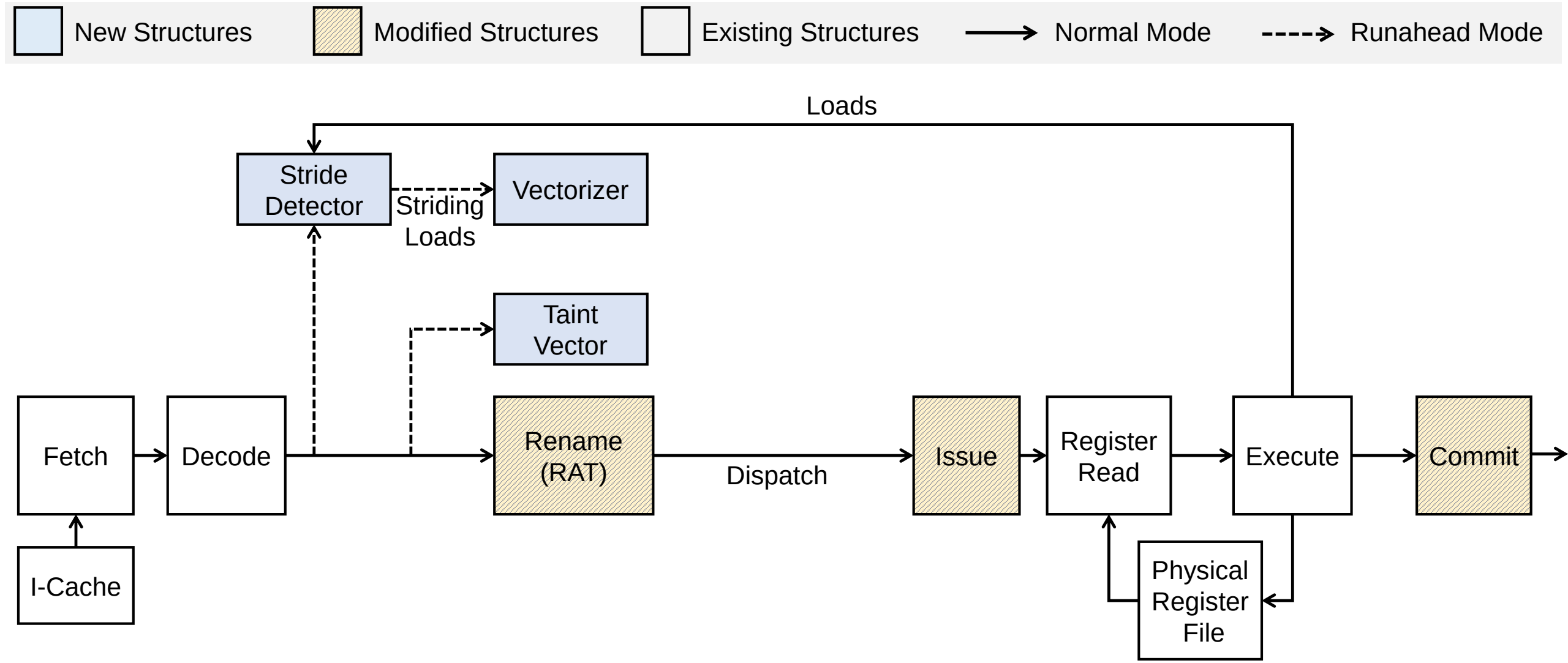
Vector Runahead Microarchitecture



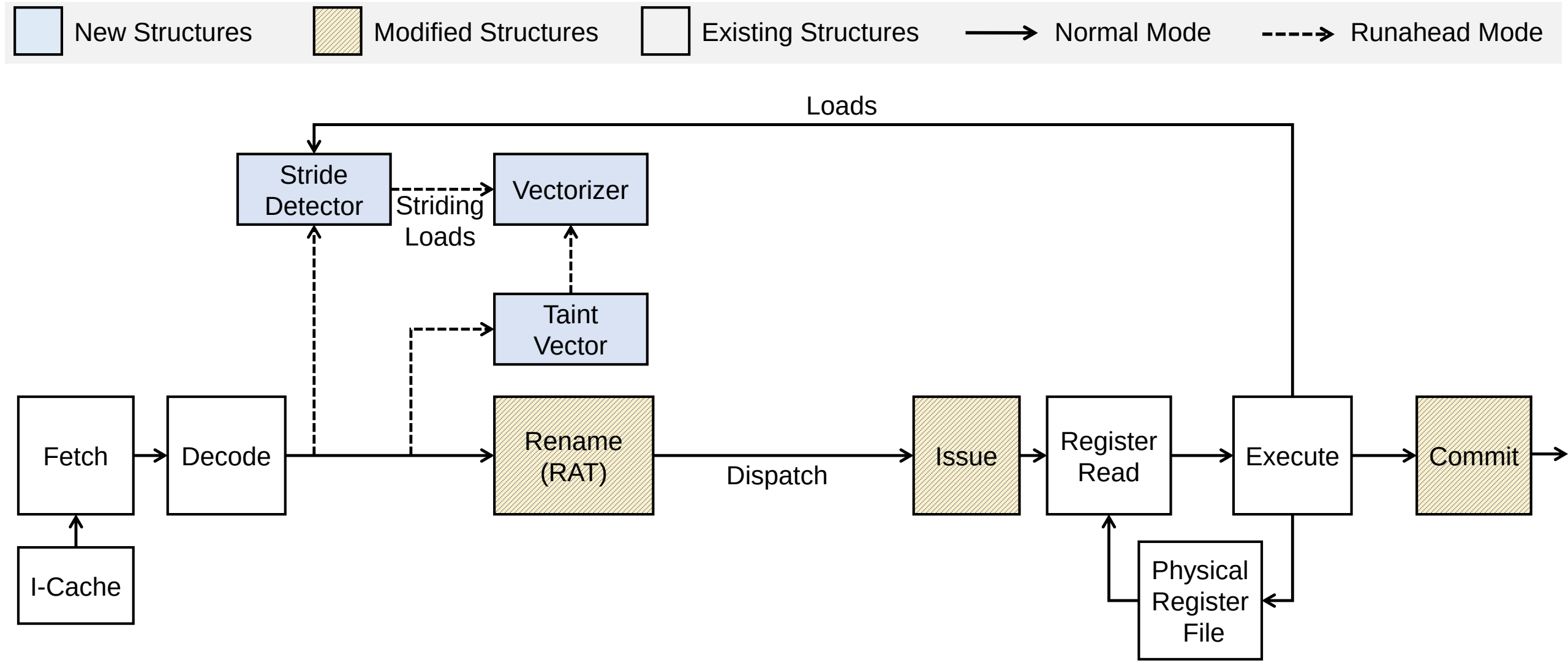
Vector Runahead Microarchitecture



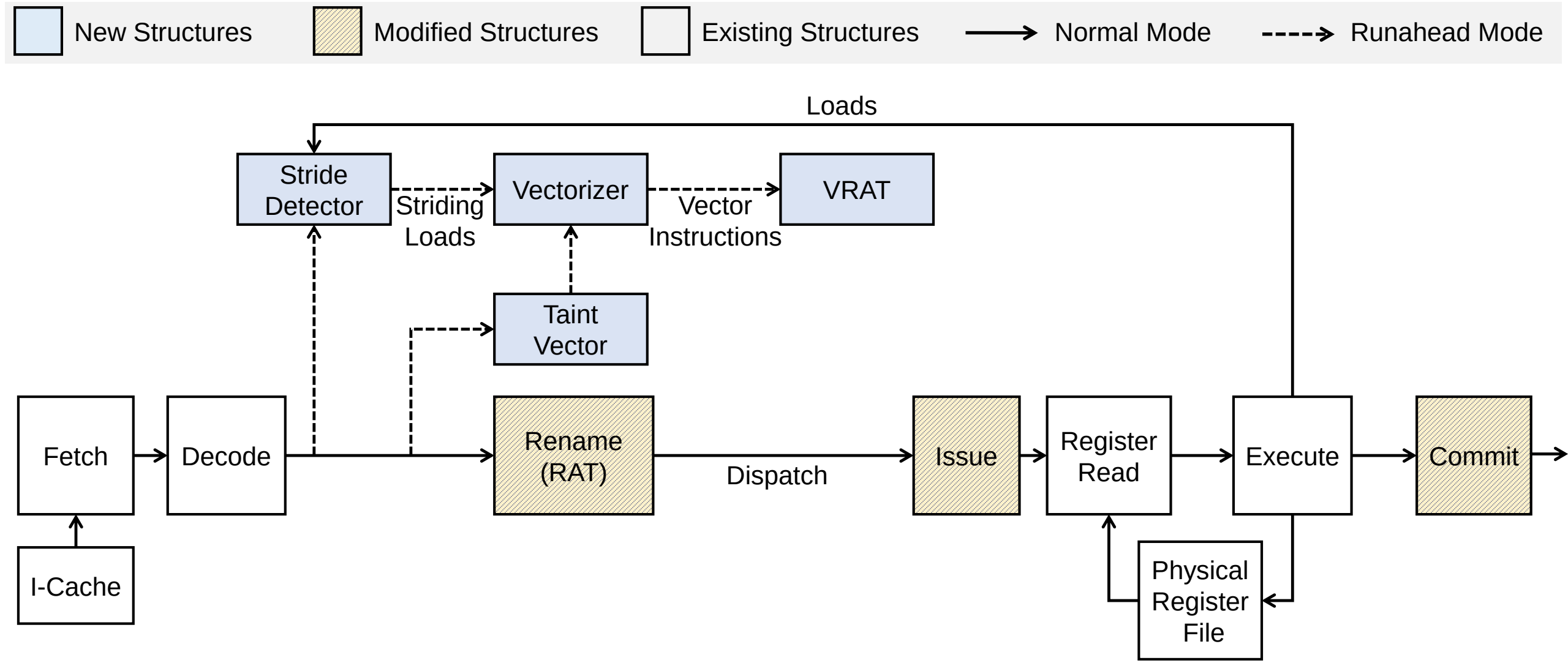
Vector Runahead Microarchitecture



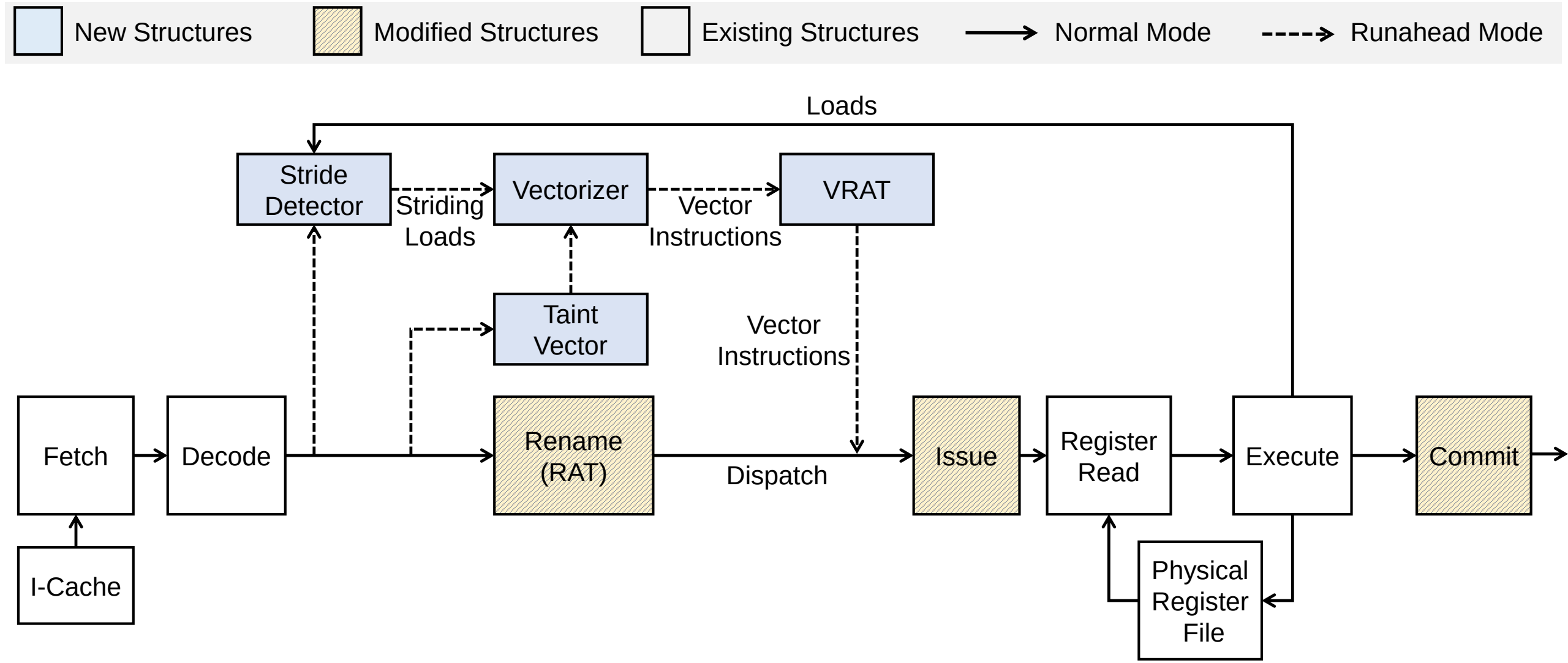
Vector Runahead Microarchitecture



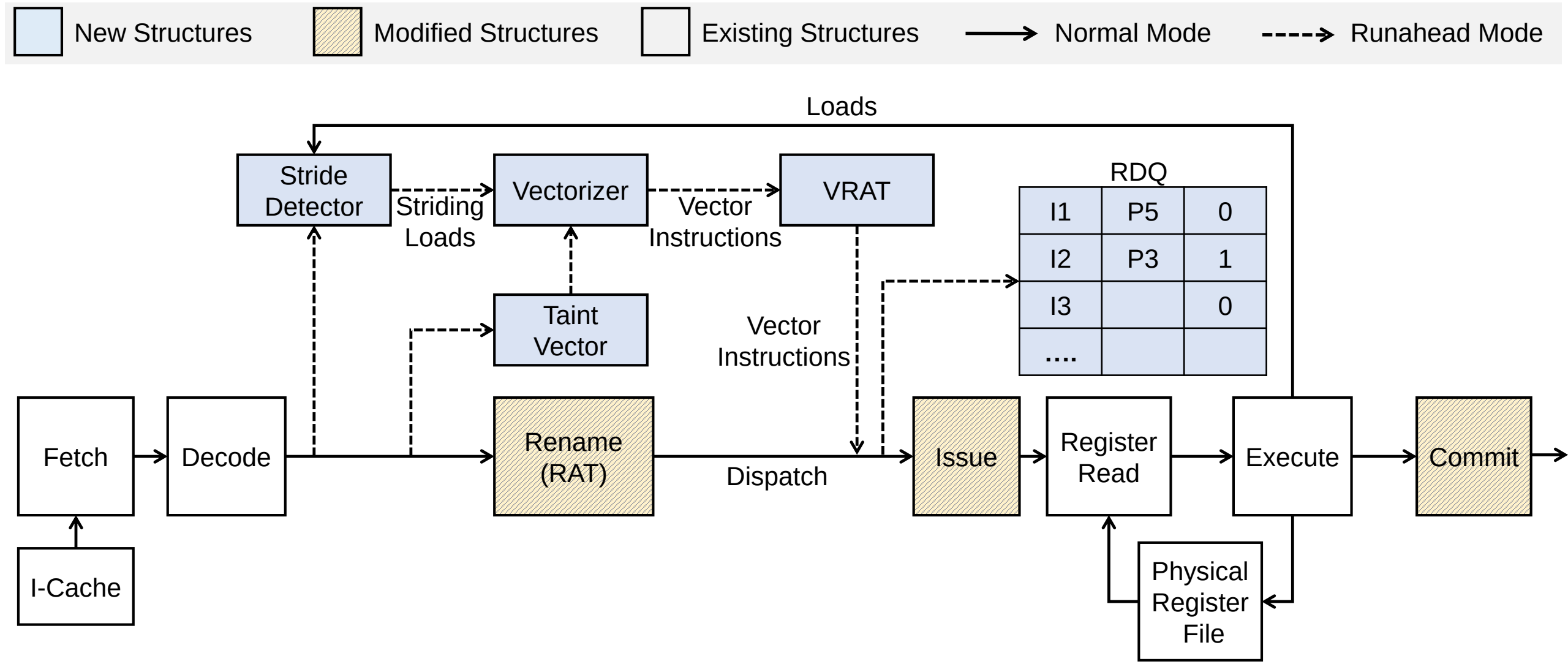
Vector Runahead Microarchitecture



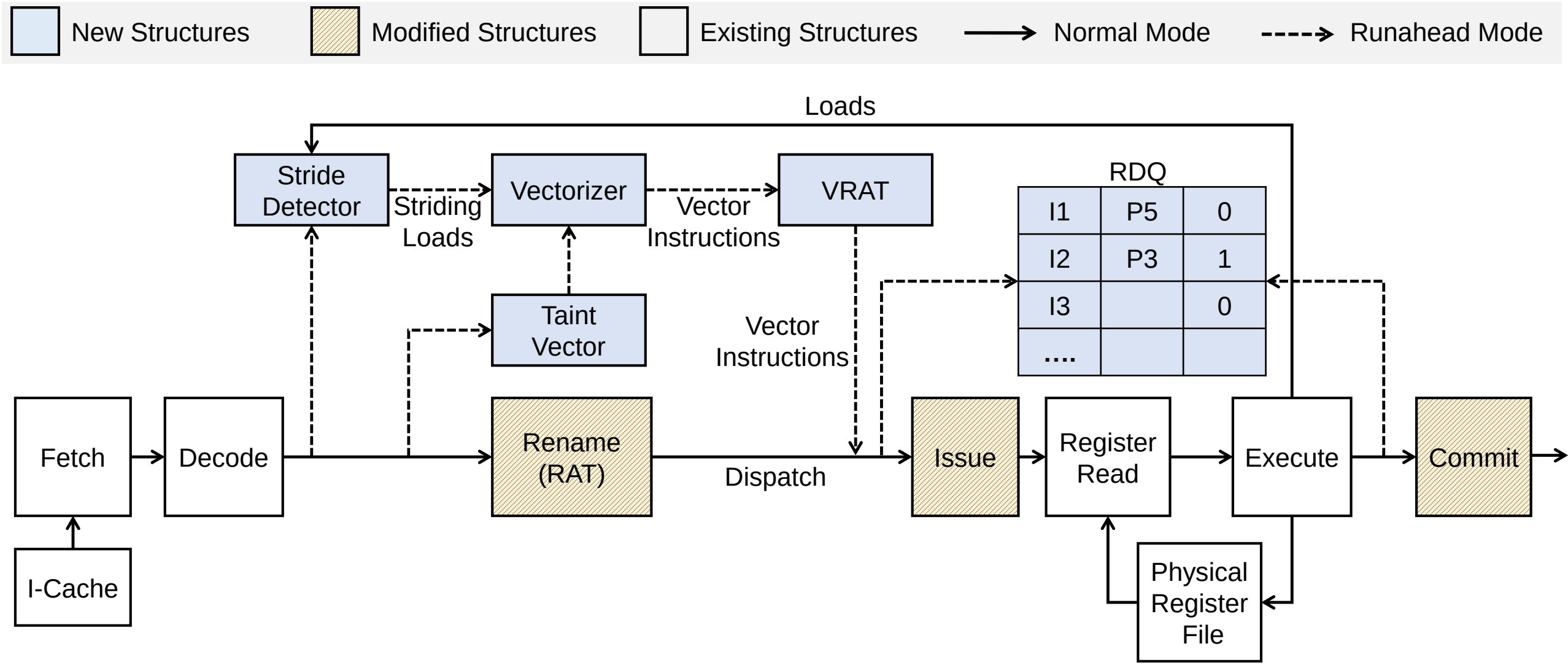
Vector Runahead Microarchitecture



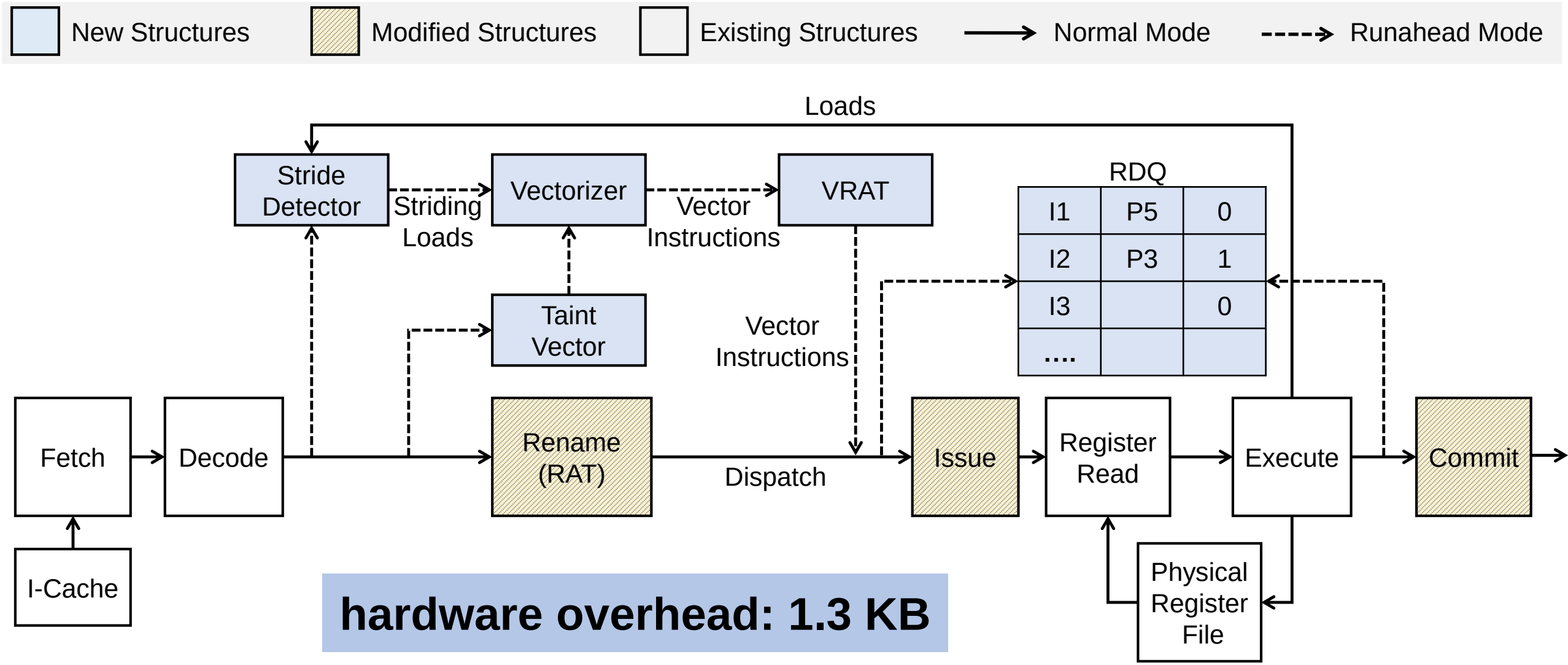
Vector Runahead Microarchitecture



Vector Runahead Microarchitecture



Vector Runahead Microarchitecture



Evaluation

Simulator: Sniper 6.0

Baseline: 3.2 GHz OoO, ROB=224, issue queue=97,
load queue=64, store queue=60, register file: 180 int, 96 vector

Workloads: Benchmarks with complex memory indirection patterns from HPC, graph analytics, and database domains

Aggressive stride prefetcher with 16 streams, MSHR=24

Evaluation

OoO: Baseline out-of-order core

Evaluation

OoO: Baseline out-of-order core

PRE: Precise runahead execution*

-- Ideal stalling slice table

*[Naithani et al. HPCA'20]

Evaluation

OoO: Baseline out-of-order core

PRE: Precise runahead execution*

-- Ideal stalling slice table

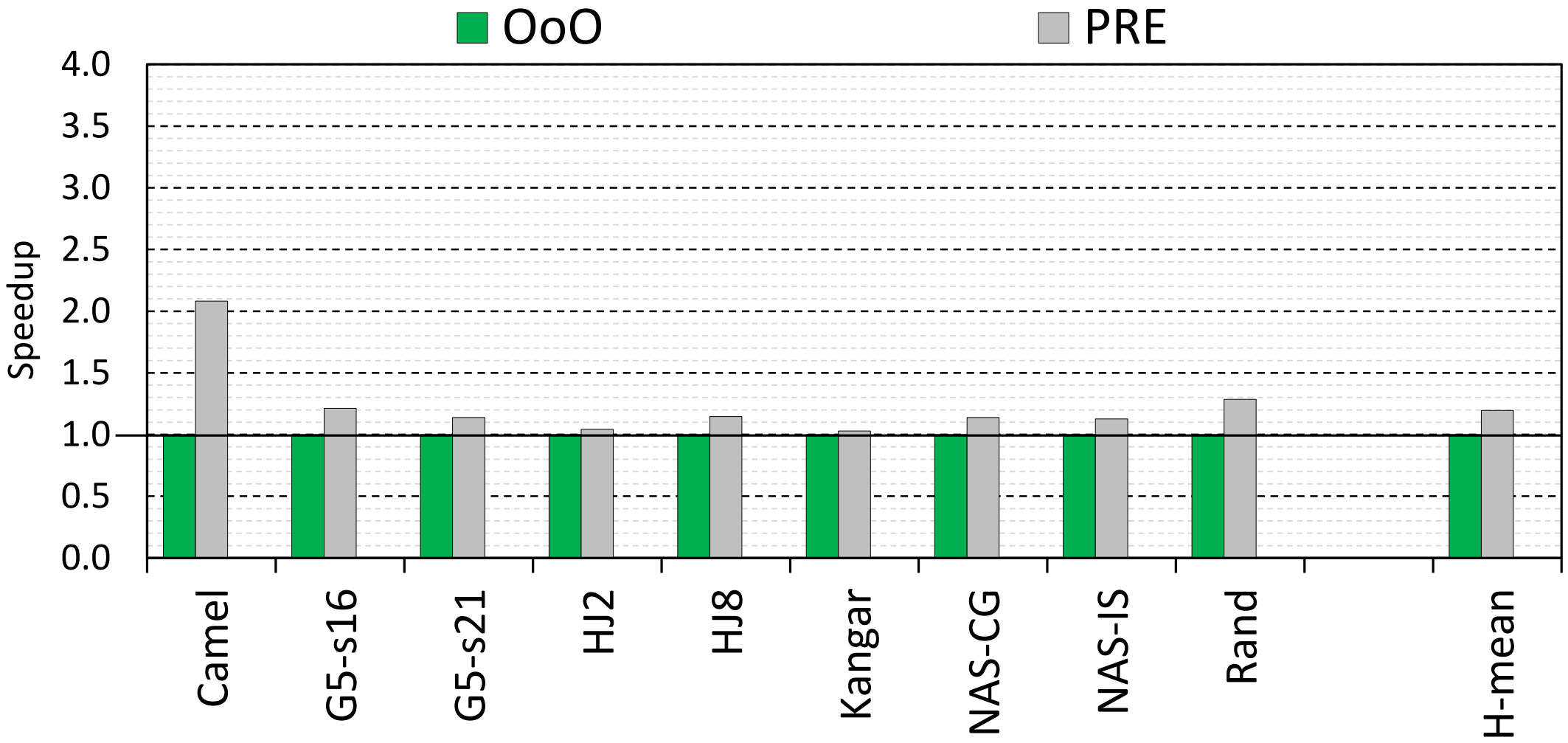
VR: Vector Runahead**

-- Pipelined version: Simultaneously issue eight vector instructions for each scalar instruction

*[Naithani et al. HPCA'20]

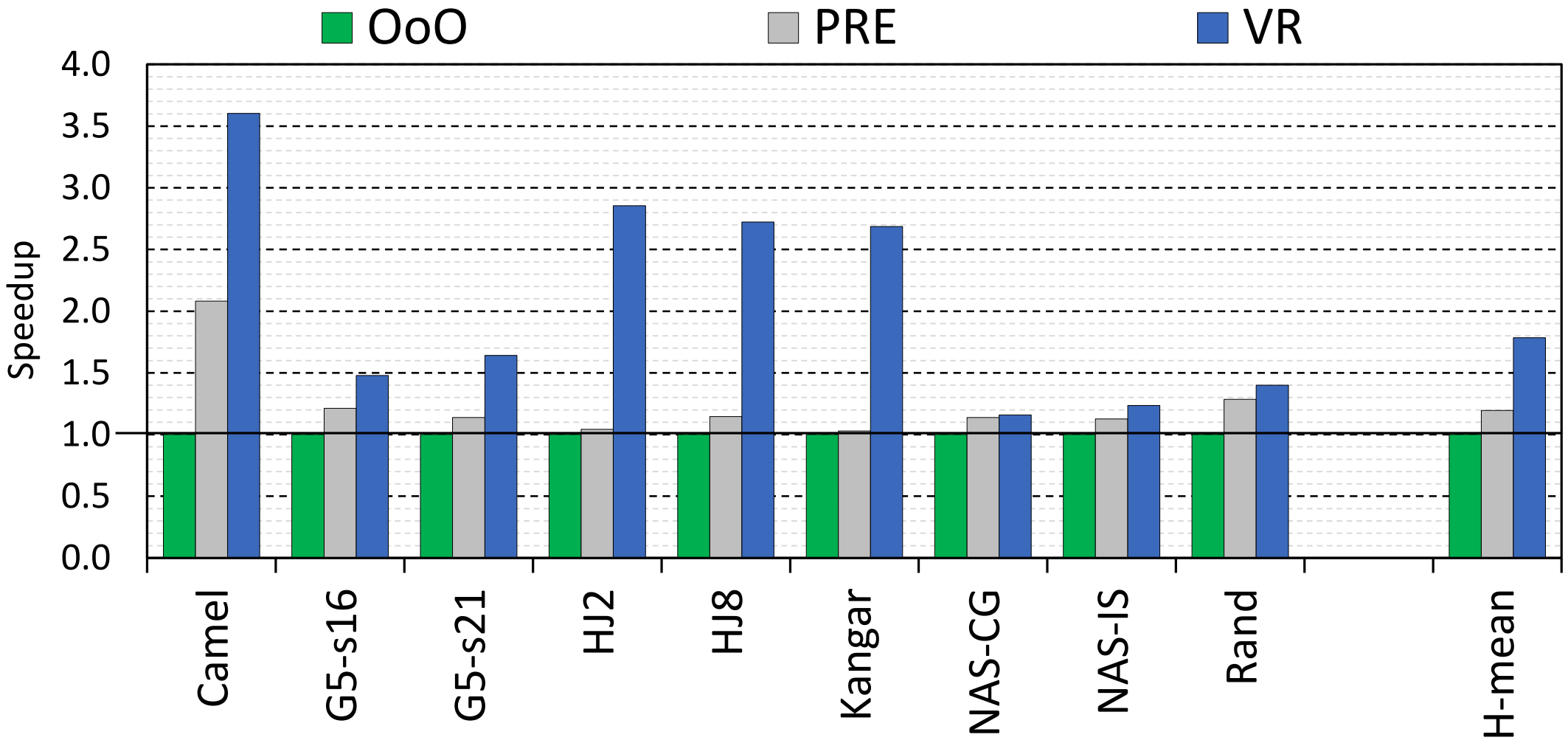
**[Naithani et al. ISCA'21]

Evaluation – Performance



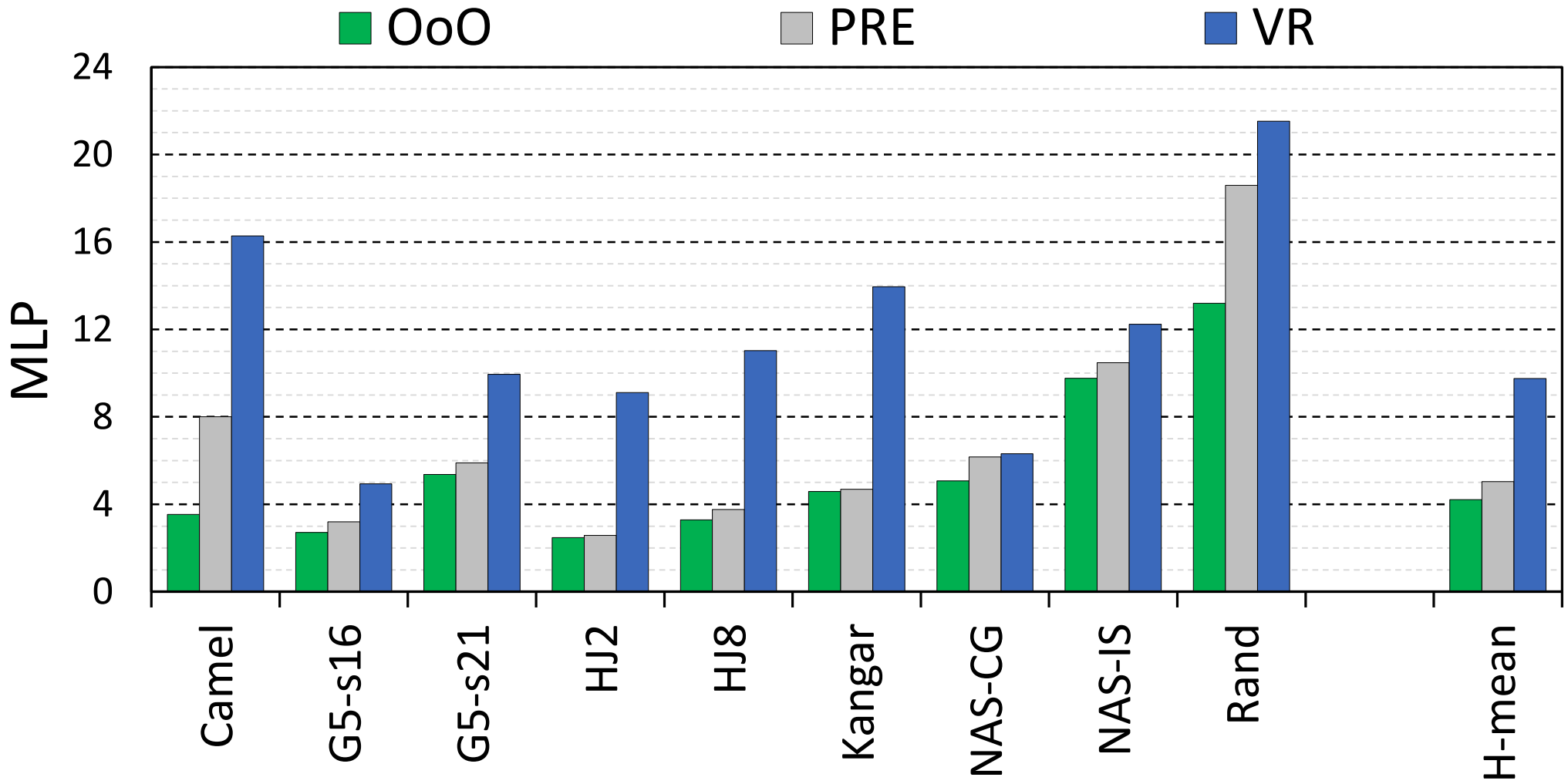
PRE: 1.20x

Evaluation – Performance



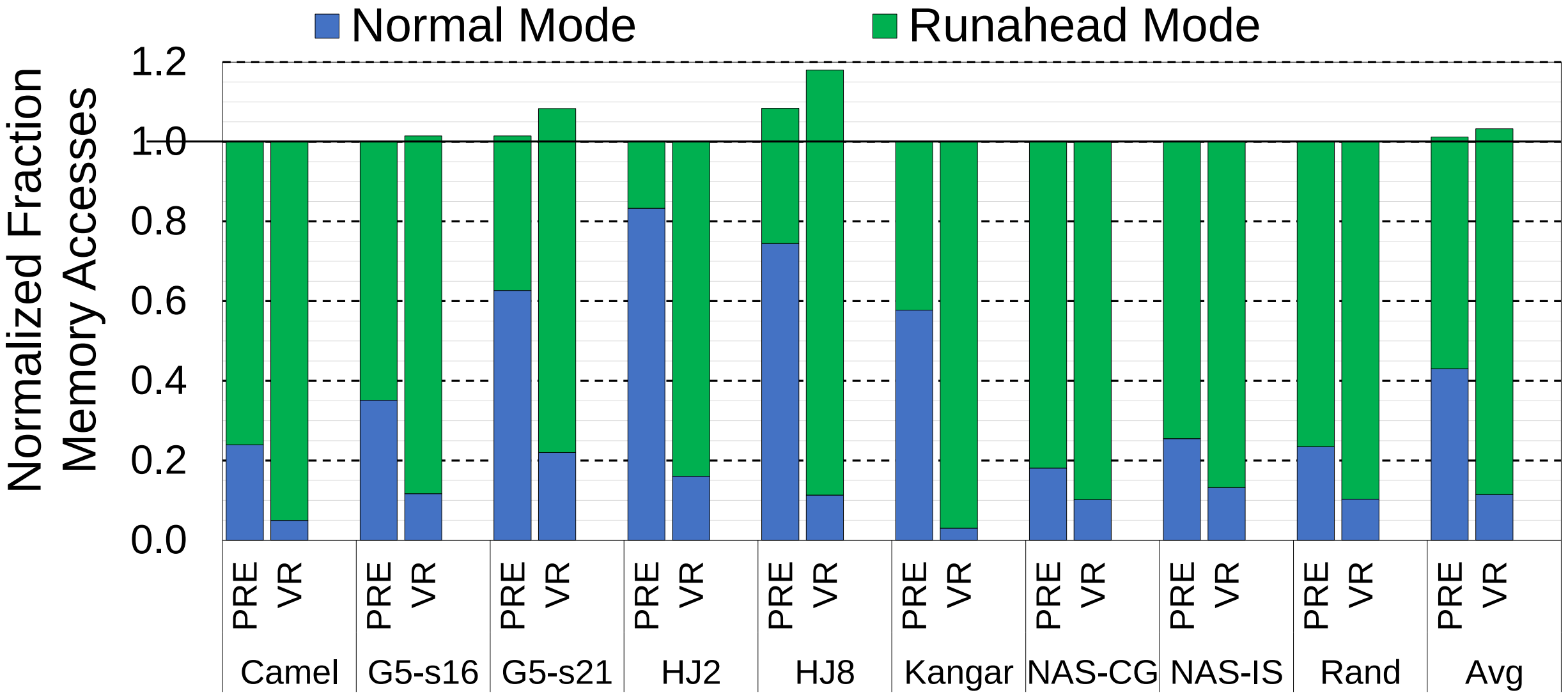
PRE: 1.20x **VR: 1.79x**

Evaluation – Memory-Level Parallelism



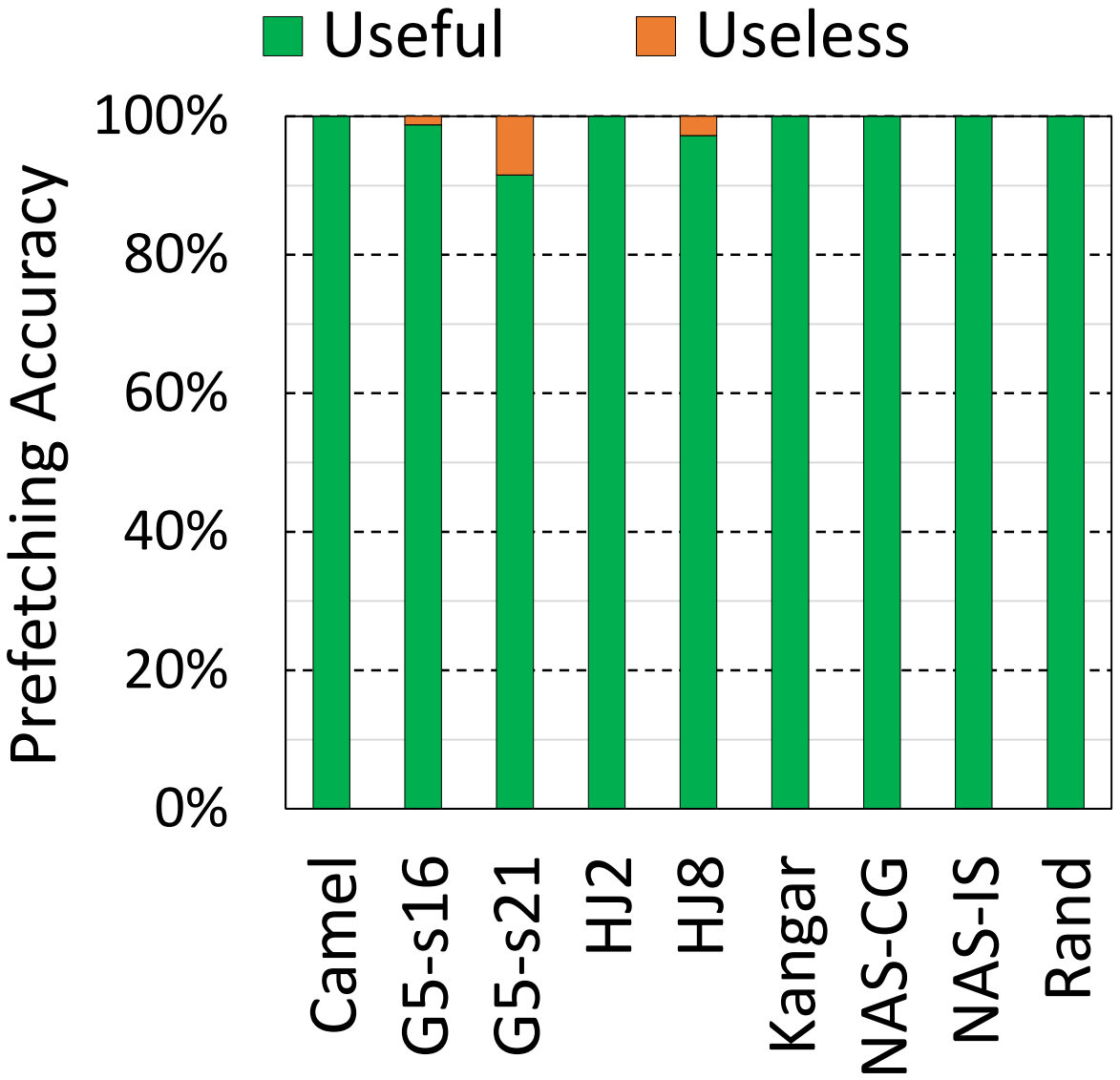
OoO: 4.2 **PRE: 5** **VR: 9.8**

Prefetching Coverage

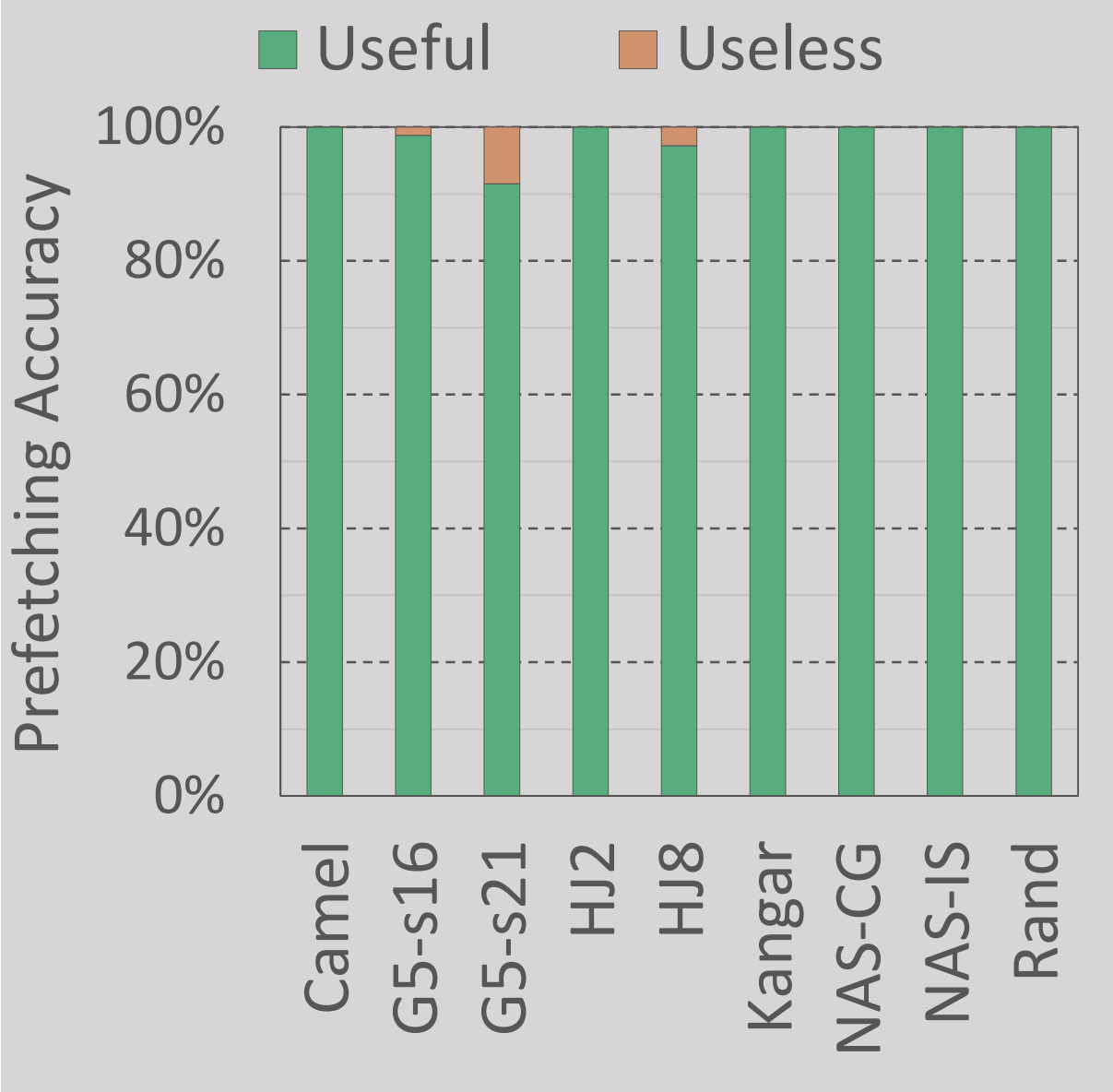


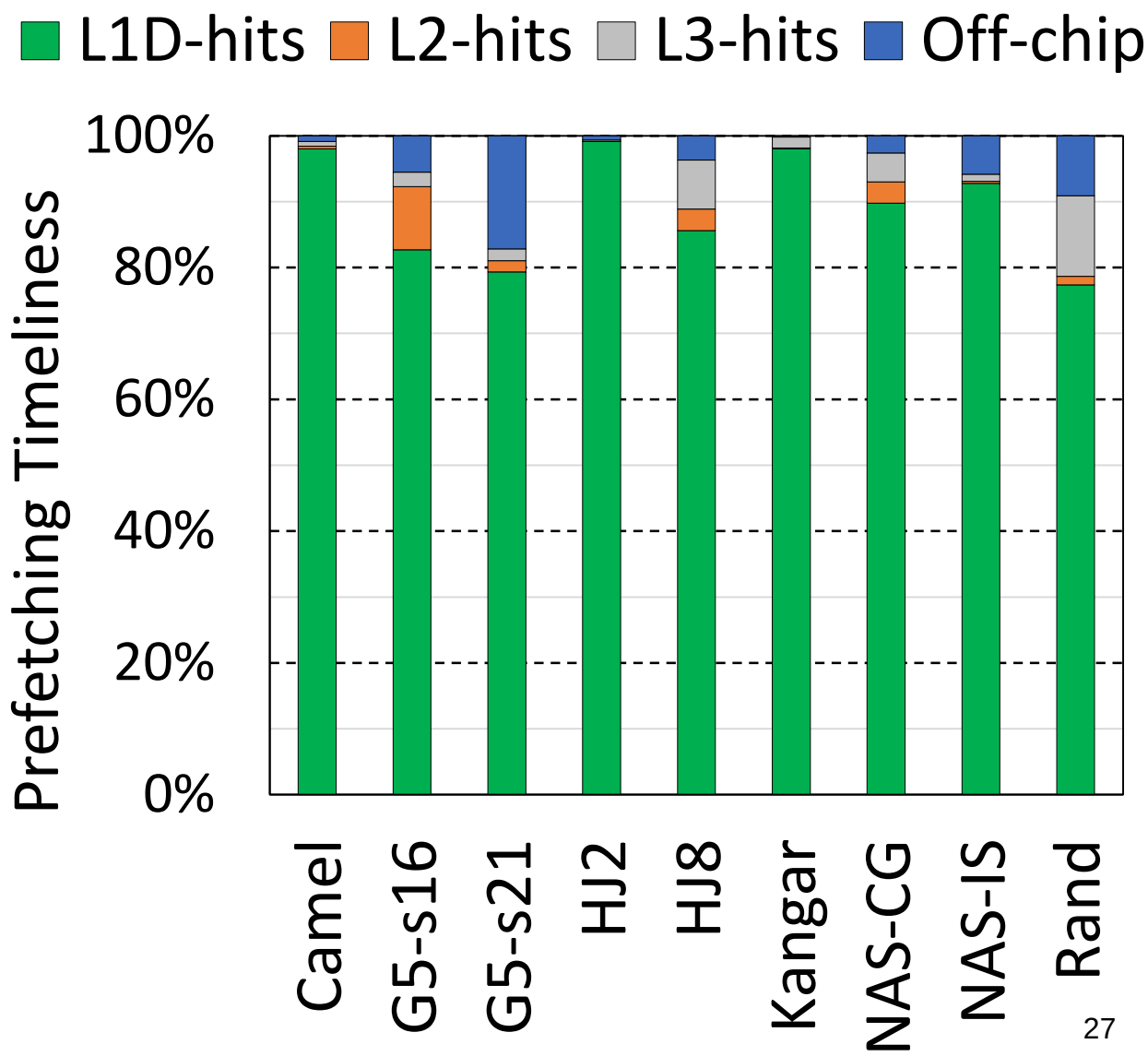
Prefetching Accuracy and Timeliness

Prefetching Accuracy and Timeliness



Prefetching Accuracy and Timeliness





Evaluation – In the Paper

Evaluation – In the Paper

1. Indirect Memory Prefetcher
   -- IMP 1.19x versus Vector Runahead: 1.79x
   -- Works only for simple single-level indirects

MICRO 2015

Evaluation – In the Paper

1. Indirect Memory Prefetcher
   -- IMP 1.19x versus Vector Runahead: 1.79x
   -- Works only for simple single-level indirects

2. Unrolling and pipelining

MICRO 2015

Evaluation – In the Paper

1. Indirect Memory Prefetcher
   -- IMP 1.19x versus Vector Runahead: 1.79x
   -- Works only for simple single-level indirects

MICRO 2015

2. Unrolling and pipelining

3. Varying LLC size: Vector Runahead equally effective

Evaluation – In the Paper

1. Indirect Memory Prefetcher
   -- IMP 1.19x versus Vector Runahead: 1.79x
   -- Works only for simple single-level indirects

MICRO 2015

2. Unrolling and pipelining

3. Varying LLC size: Vector Runahead equally effective

4. Varying MSHR entries
   -- Sufficient MSHRs needed for VR's extreme level of MLP

Conclusions

Conclusions

1. Achieves high degree of MLP for indirect accesses with complex address calculation

Conclusions

1. Achieves high degree of MLP for indirect accesses with complex address calculation

2. Vector Runahead = Reordered loads + speculative vectorization + delayed termination

Conclusions

1. Achieves high degree of MLP for indirect accesses with complex address calculation

2. Vector Runahead = Reordered loads + speculative vectorization + delayed termination

3. Increases effective fetch/decode bandwidth

Conclusions

1. Achieves high degree of MLP for indirect accesses with complex address calculation

2. Vector Runahead = Reordered loads + speculative vectorization + delayed termination

3. Increases effective fetch/decode bandwidth

1.79x higher performance by 2.3x higher MLP

Vector Runahead

Ajeya Naithani (Ghent)
Sam Ainsworth (Edinburgh)
Timothy M. Jones (Cambridge)
Lieven Eeckhout (Ghent)

THE UNIVERSITY *of* EDINBURGH

UNIVERSITY OF CAMBRIDGE