vPIM: Efficient Virtual Address Translation for Scalable Processing-in-Memory Architectures

Amel Fatima, Sihang Liu, Korakit Seemakhupt, Rachata Ausavarungnirun, Samira Khan

Summary

- Multiple PIM stacks provide increase capacity for emerging data-intensive workloads

Problem: Address translation incurs significant overhead in accessing multiple PIM stacks

- ***Radix-based translation:*** Multiple sequential cross-stack page table walks
- ***Cuckoo hash-based translation:*** Multiple parallel accesses across memory stacks

Goal: *Design an efficient address translation scheme for multi-stack PIM systems*

Observations and Key Ideas:

- *Observation: High intra-stack parallelism with multiple vaults in each stack*
- Key Idea: Network-contention-aware hash
  - Change cuckoo hash layout to generate parallel accesses within one stack across multiple vaults
  - Send only one request towards the stack and generate parallel accesses within the stack
- *Observation: High parallelism enables minimal performance loss with fewer cores*
- Key Idea: Pre-translate addresses
  - Use a few cores to run ahead future iterations of the workload
  - Pre-translate cross-stack addresses and bring them inside the same stack for future use

Result: vPIM provide 4.4x and 1.7x speedup compared to radix and cuckoo hash

Outline

- Processing-in-Memory Introduction
- Address Translation in PIM Architecture
- Challenges and Key Ideas
- Evaluation
- Conclusion

Outline

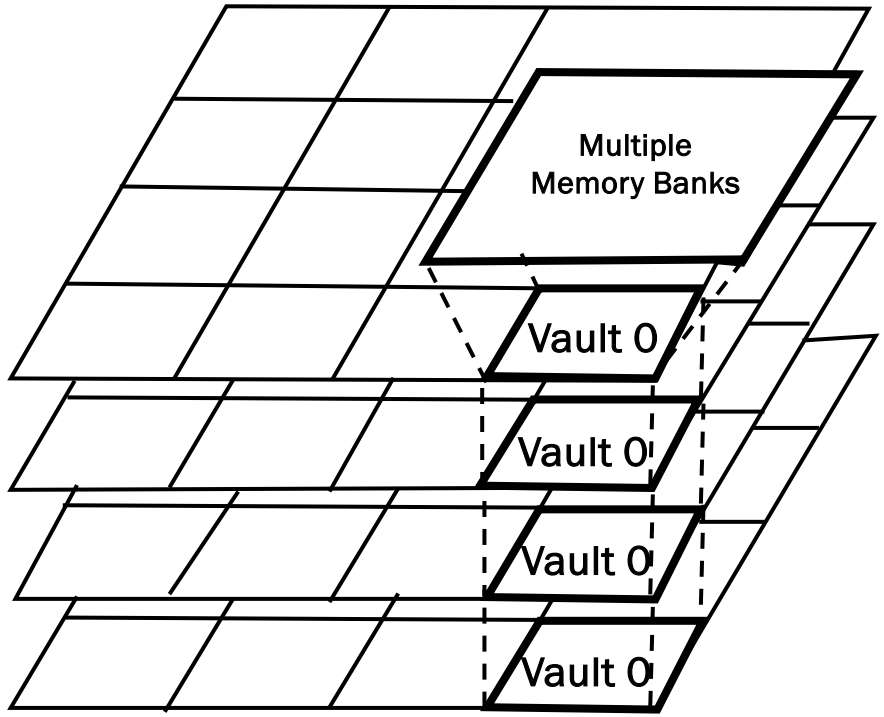
- Processing-in-Memory Introduction
- Address Translation in PIM Architecture
- Challenges and Key Ideas
- Evaluation
- Conclusion

Processing in Memory (PIM)

- PIM is a new technology where computation is placed closer to the memory

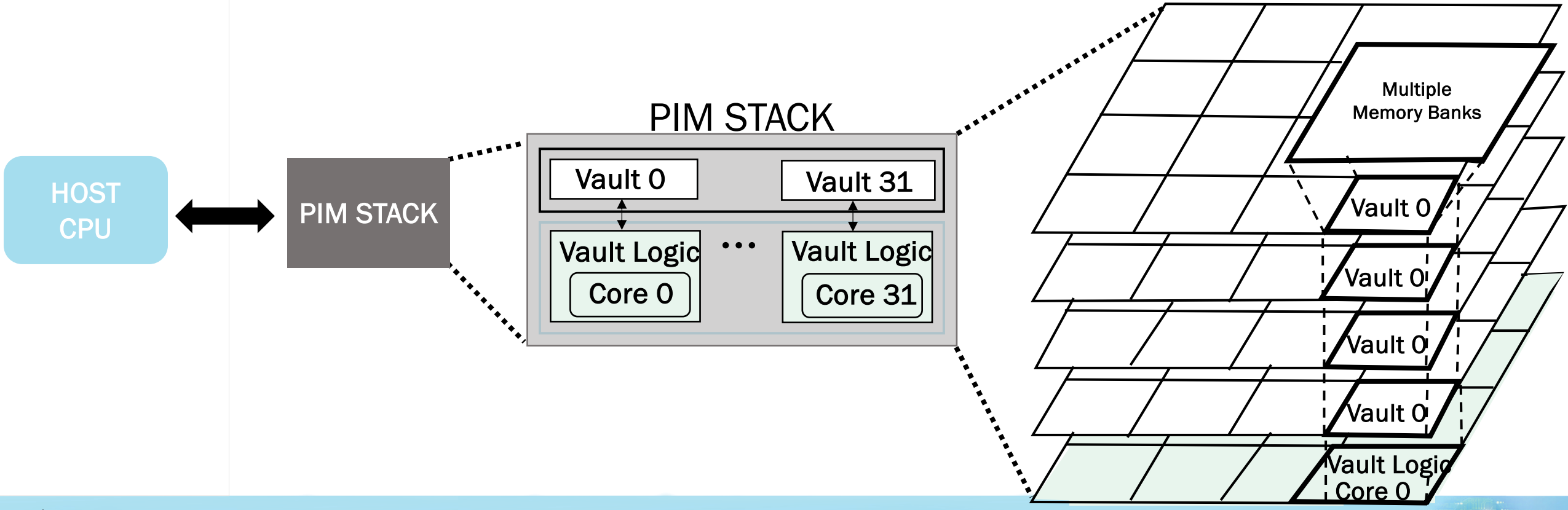
Processing in Memory (PIM)

- PIM is a new technology where computation is placed closer to the memory



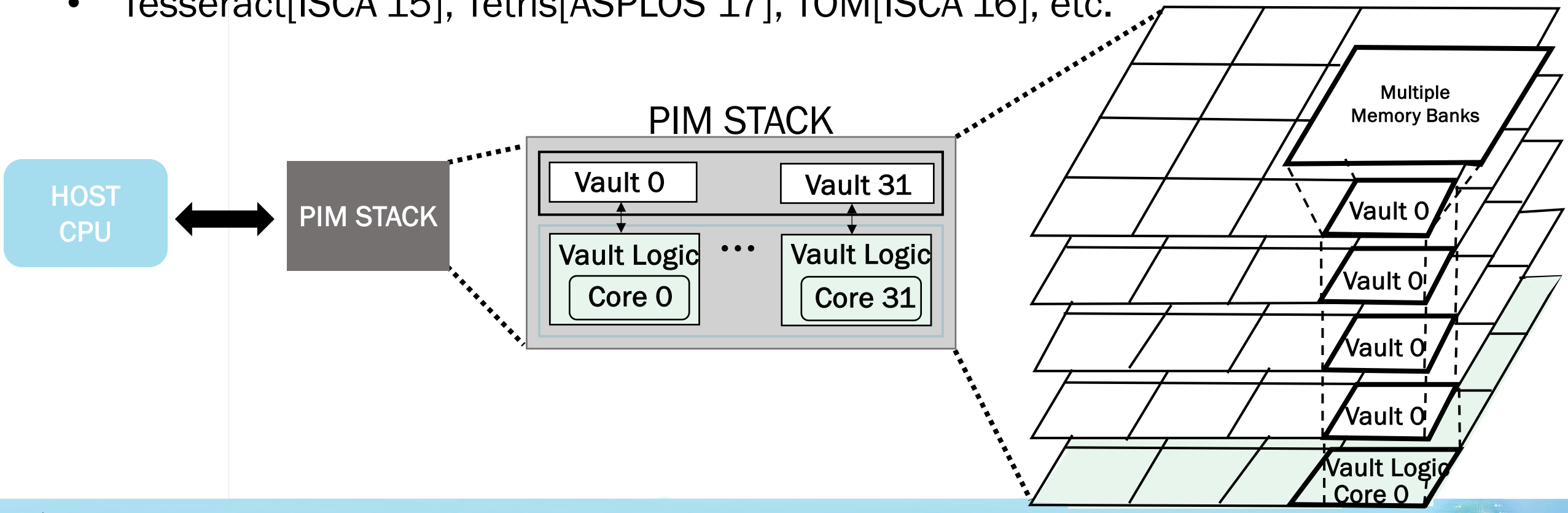
Processing in Memory (PIM)

- PIM is a new technology where computation is placed closer to the memory



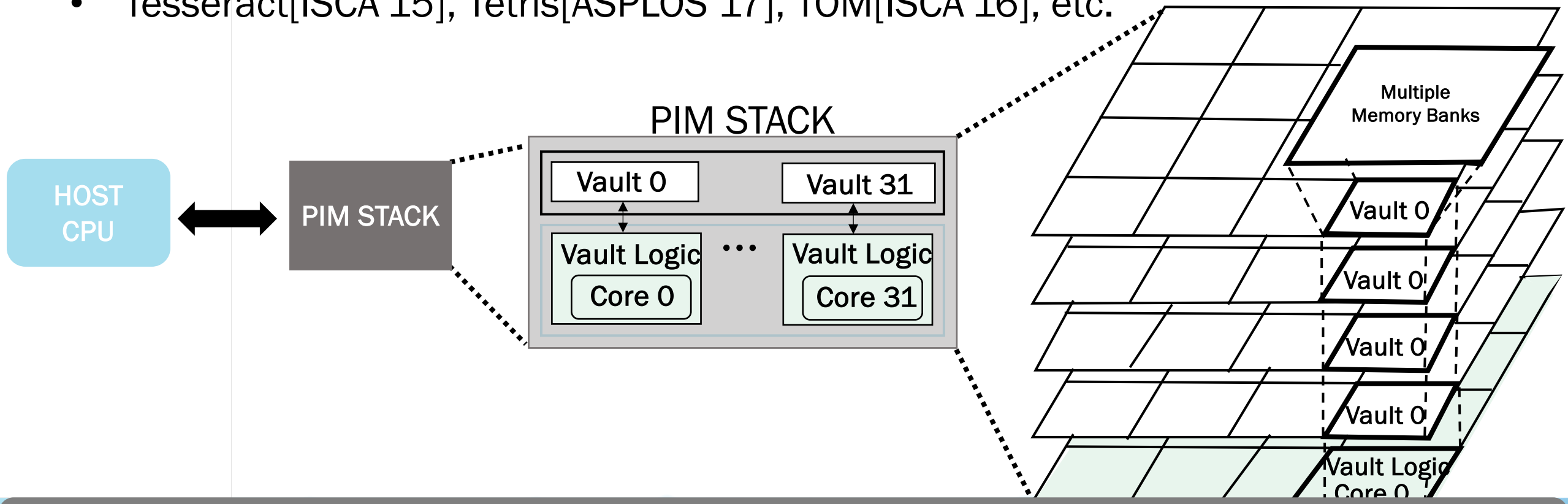
Processing in Memory (PIM)

- PIM is a new technology where computation is placed closer to the memory
- Prior works used PIM to reduce data movement between CPU and memory
  - Tesseract[ISCA'15], Tetris[ASPLOS'17], TOM[ISCA'16], etc.



Processing in Memory (PIM)

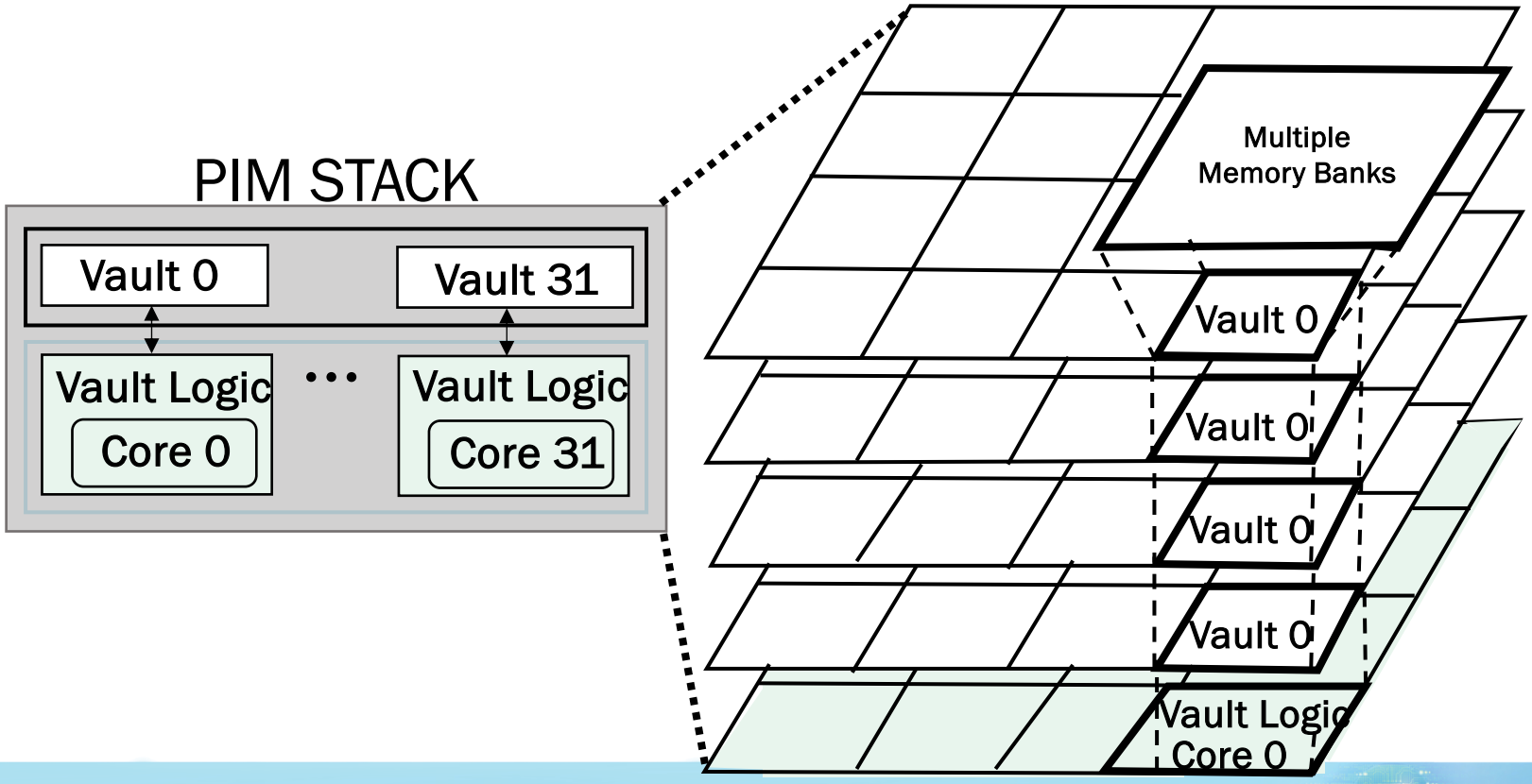
- PIM is a new technology where computation is placed closer to the memory
- Prior works used PIM to reduce data movement between CPU and memory
  - Tesseract[ISCA'15], Tetris[ASPLOS'17], TOM[ISCA'16], etc.



Placing computation closer to memory improves performance

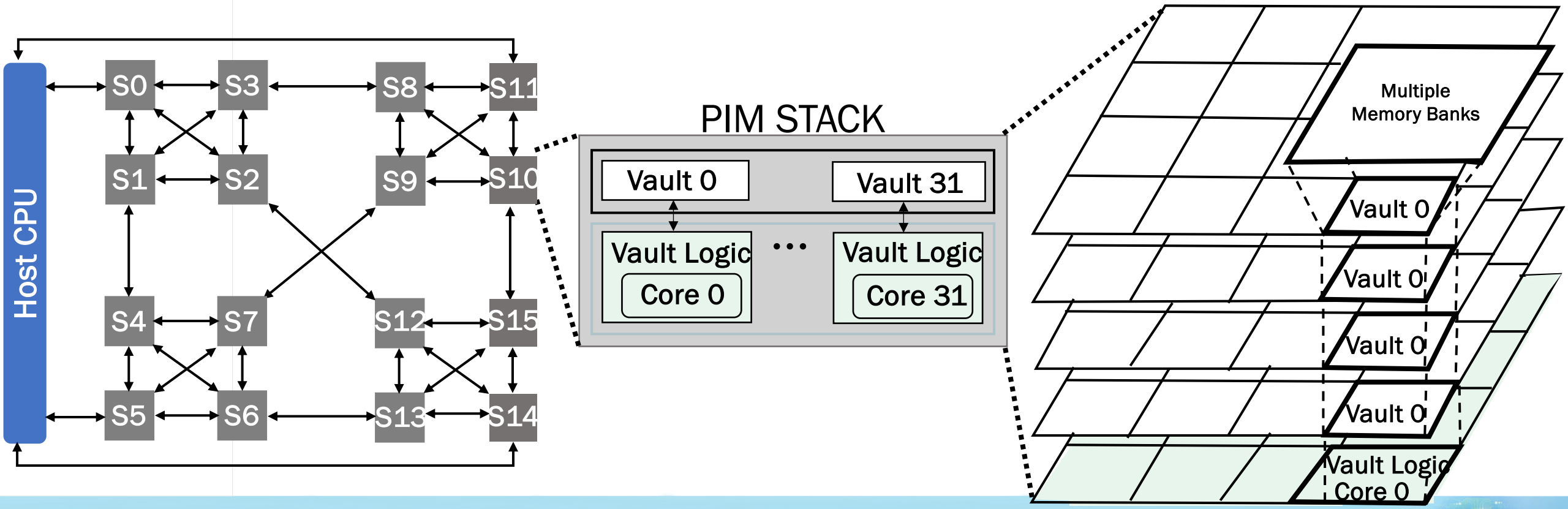
Processing in Memory (PIM)

- Data-intensive applications demand more memory



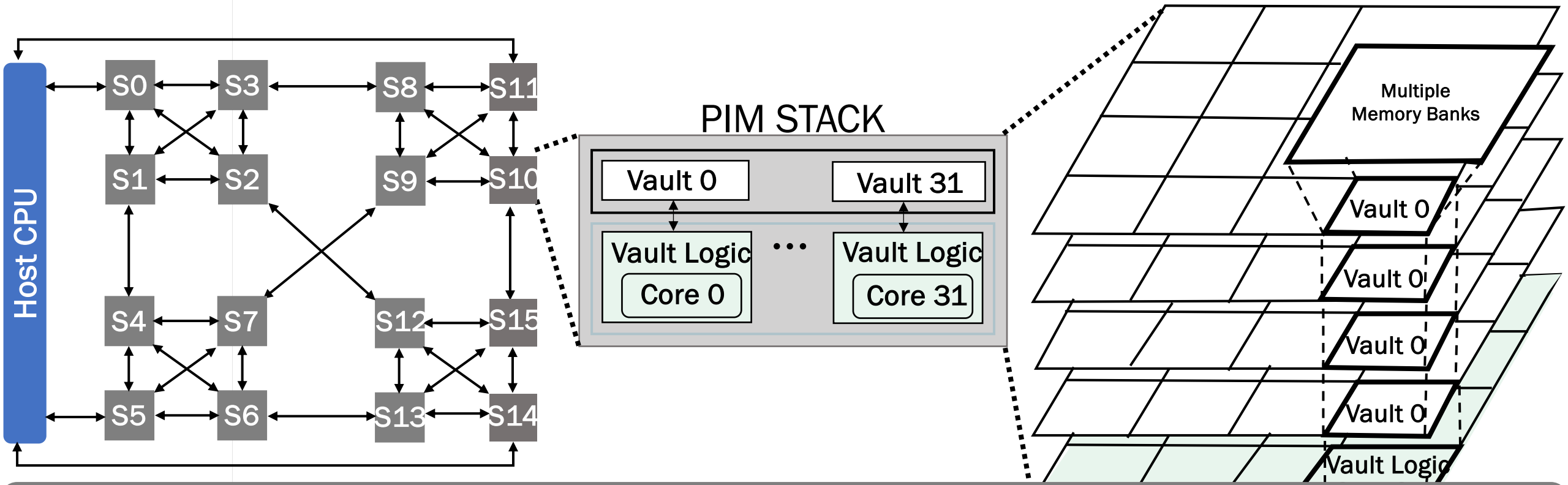
Processing in Memory (PIM)

- Data-intensive applications demand more memory
- Multiple stacks are connected over a memory network to scale up to higher capacity



Processing in Memory (PIM)

- Data-intensive applications demand more memory
- Multiple stacks are connected over a memory network to scale up to higher capacity

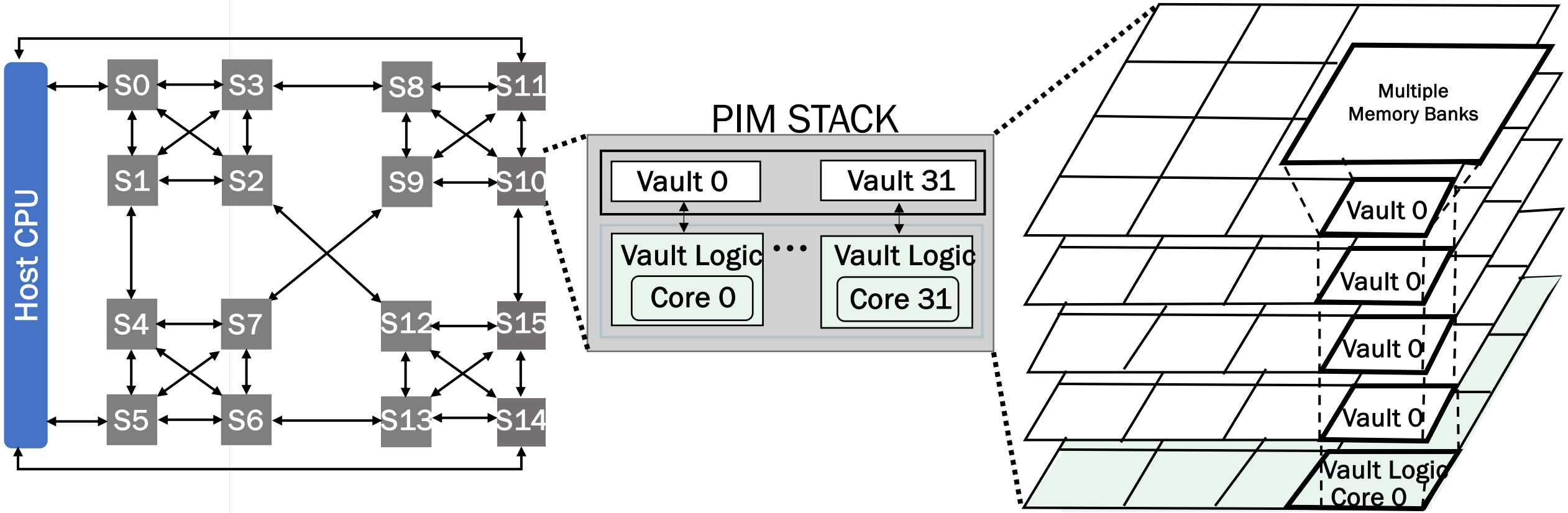


A multi-stacked PIM system provides increased capacity for emerging data-intensive applications

Outline

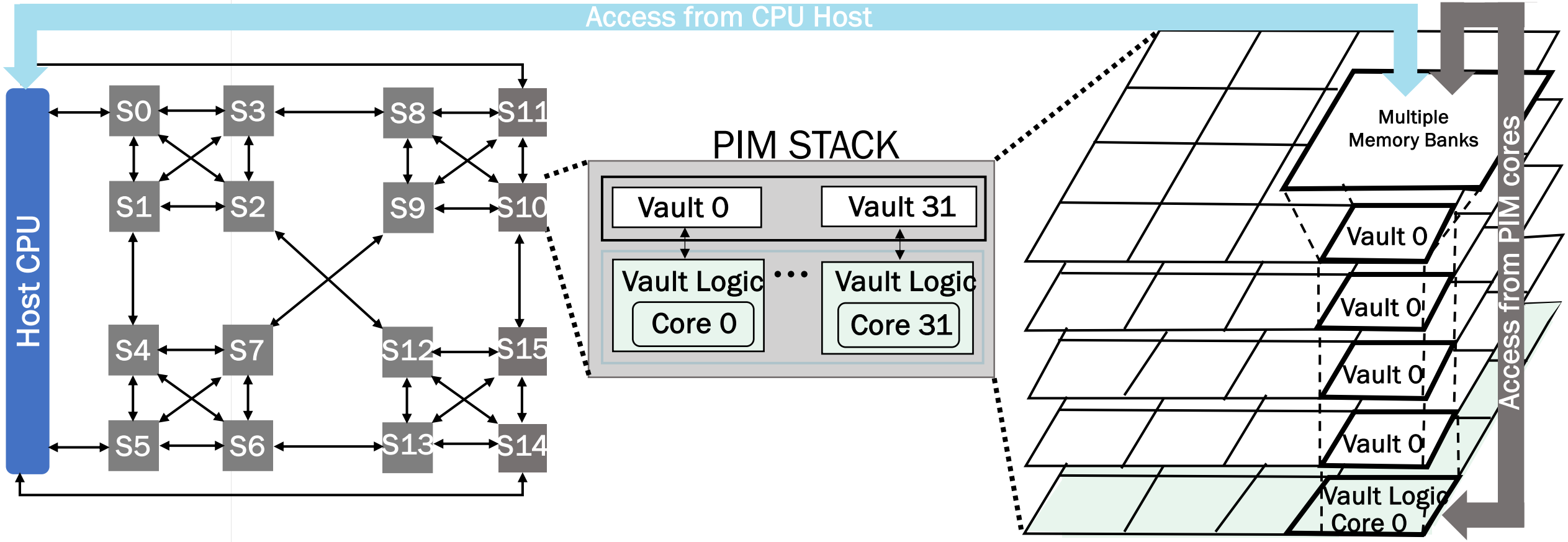
- Processing-in-Memory Introduction
- Address Translation in PIM Architecture
- Challenges and Key Ideas
- Evaluation
- Conclusion

Memory Management in PIM



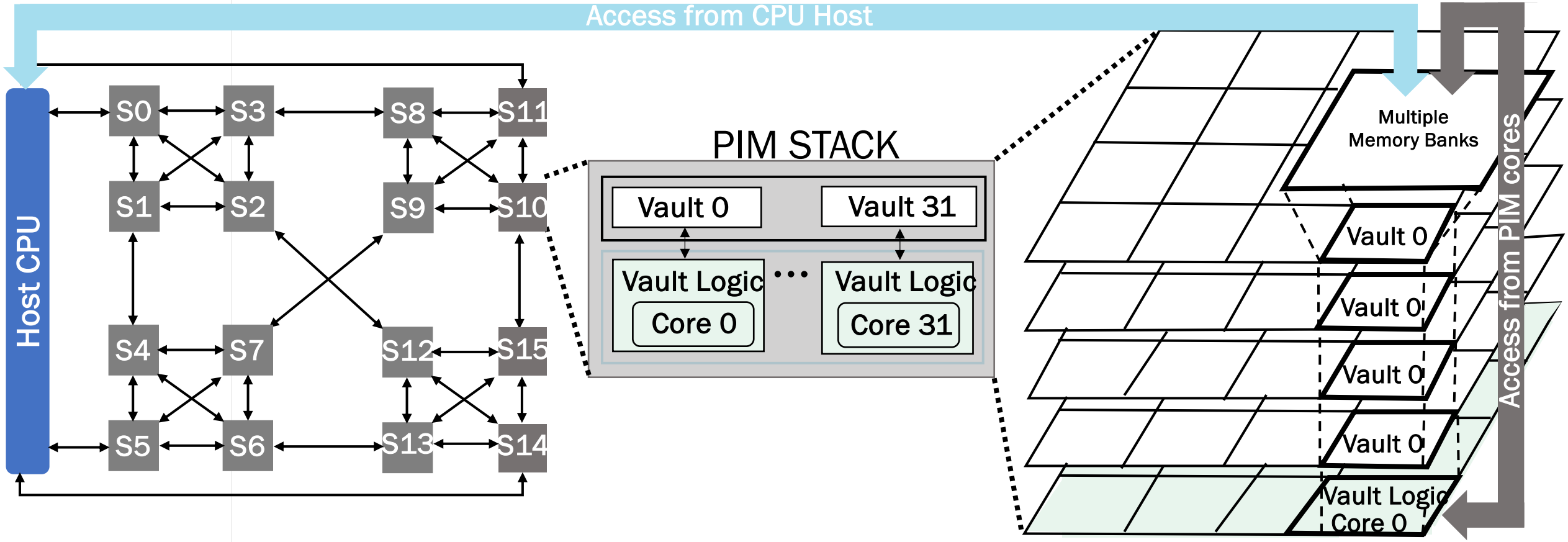
Memory Management in PIM

- Both CPU and PIM cores are accessing the same memory



Memory Management in PIM

- Both CPU and PIM cores are accessing the same memory

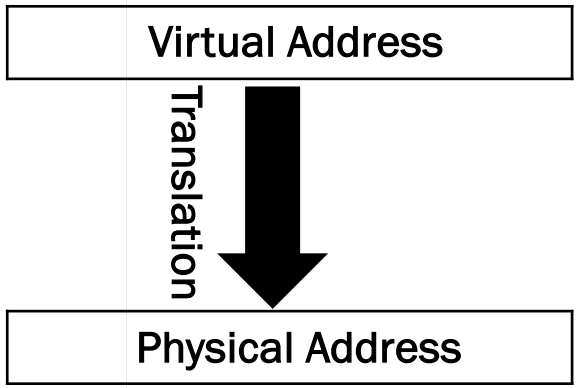


PIM provides an opportunity for a unified virtual address space

Address Translation

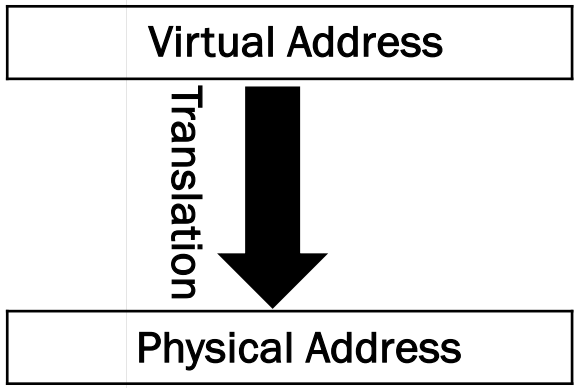
Address Translation

- Unified virtual memory requires translating virtual addresses to physical addresses



Address Translation

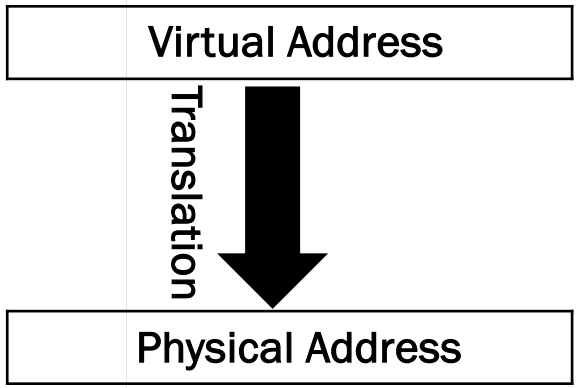
- Unified virtual memory requires translating virtual addresses to physical addresses



- Address translation requires multiple accesses to the page table

Address Translation

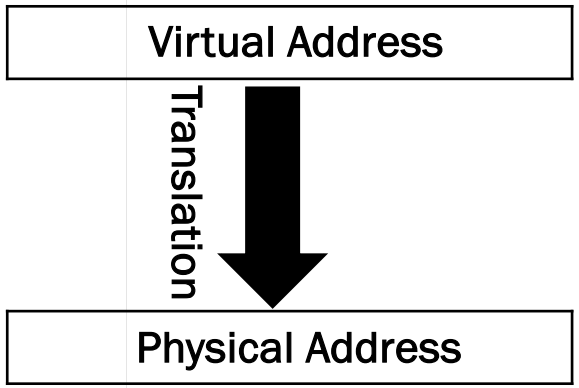
- Unified virtual memory requires translating virtual addresses to physical addresses



- Address translation requires multiple accesses to the page table
- It introduces significant overhead in conventional systems which is even worse in PIM

Address Translation

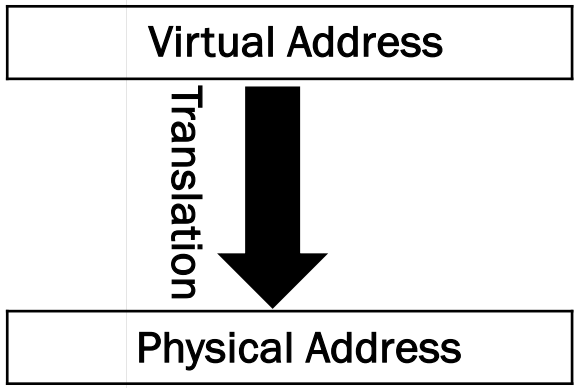
- Unified virtual memory requires translating virtual addresses to physical addresses



- Address translation requires multiple accesses to the page table
- It introduces significant overhead in conventional systems which is even worse in PIM
- There are two different commonly used approaches for address translation:

Address Translation

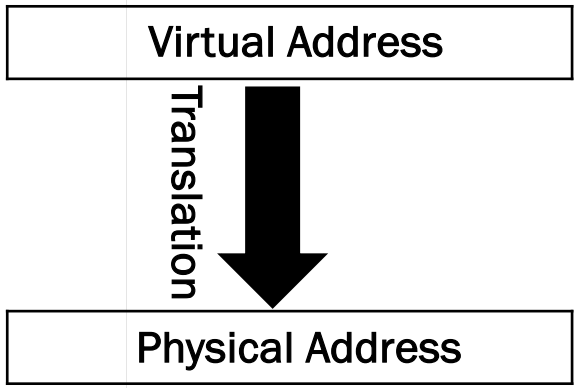
- Unified virtual memory requires translating virtual addresses to physical addresses



- Address translation requires multiple accesses to the page table
- It introduces significant overhead in conventional systems which is even worse in PIM
- There are two different commonly used approaches for address translation:
  - Radix Page Table

Address Translation

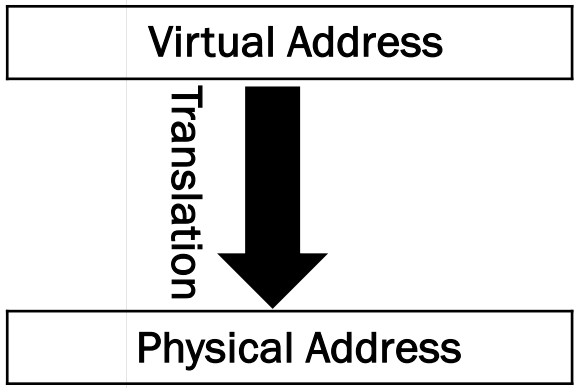
- Unified virtual memory requires translating virtual addresses to physical addresses



- Address translation requires multiple accesses to the page table
- It introduces significant overhead in conventional systems which is even worse in PIM
- There are two different commonly used approaches for address translation:
  - Radix Page Table
  - Hash Page Table

Address Translation

- Unified virtual memory requires translating virtual addresses to physical addresses

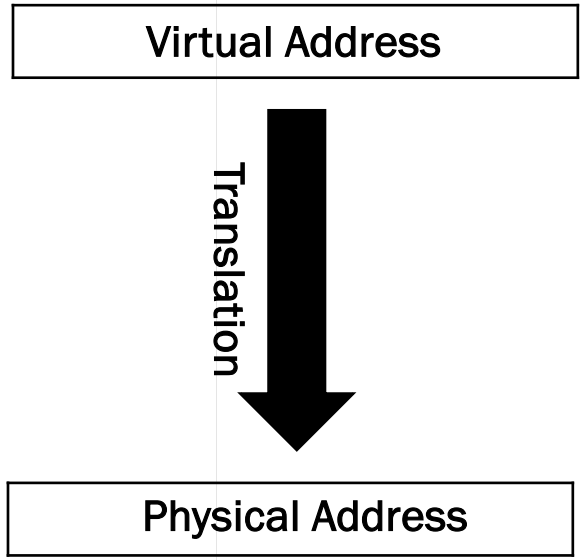


- Address translation requires multiple accesses to the page table
- It introduces significant overhead in conventional systems which is even worse in PIM
- There are two different commonly used approaches for address translation:
  - Radix Page Table
  - Hash Page Table

Both radix and hash page table introduce significant overhead in multi-stacked PIM system

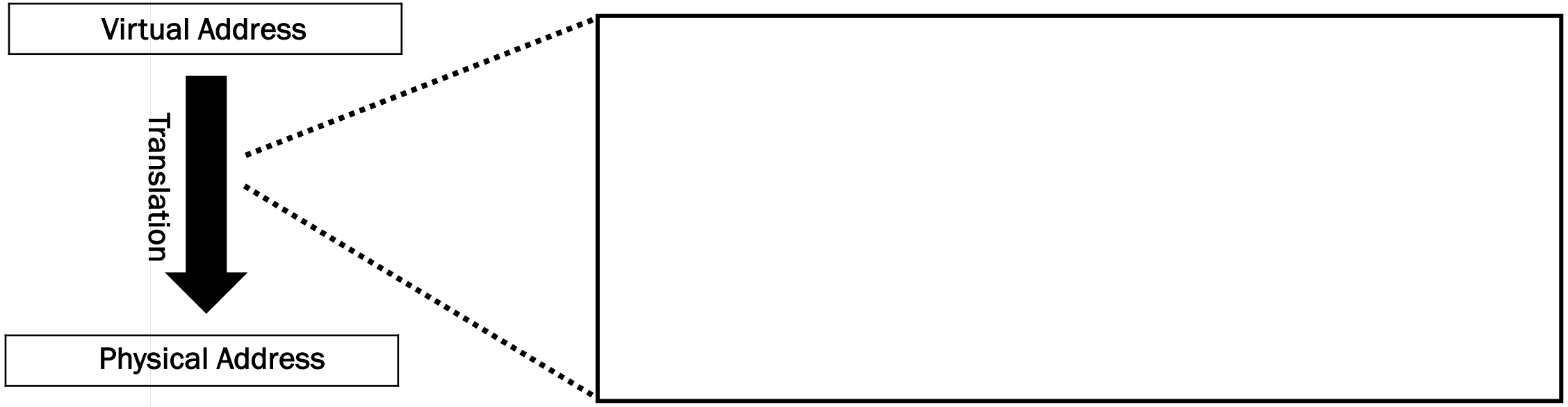
Radix Based Address Translation

- A virtual address is translated to a physical address by walking 4 levels of page table



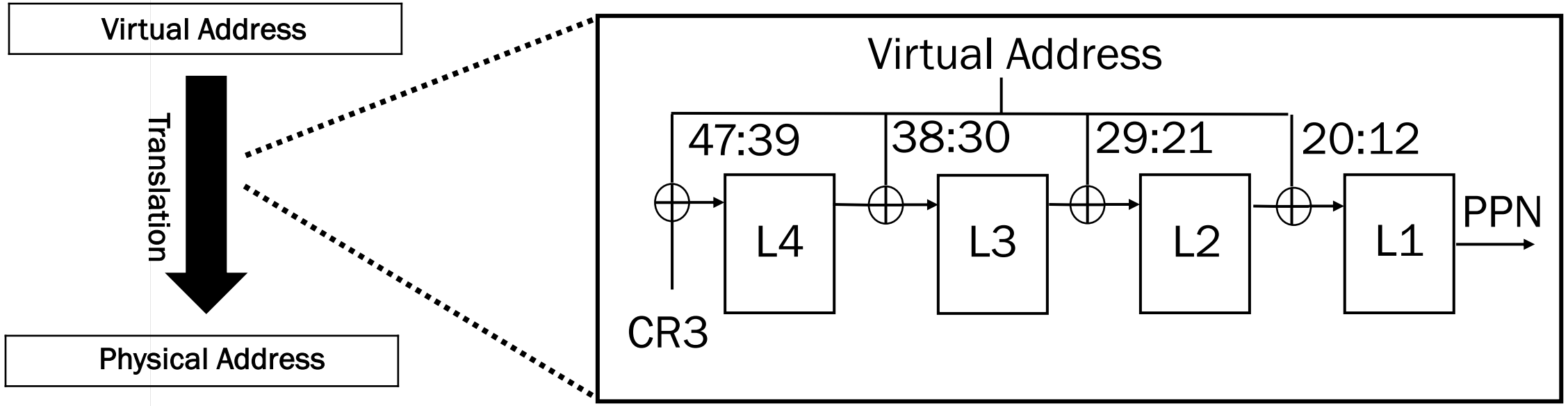
Radix Based Address Translation

- A virtual address is translated to a physical address by walking 4 levels of page table



Radix Based Address Translation

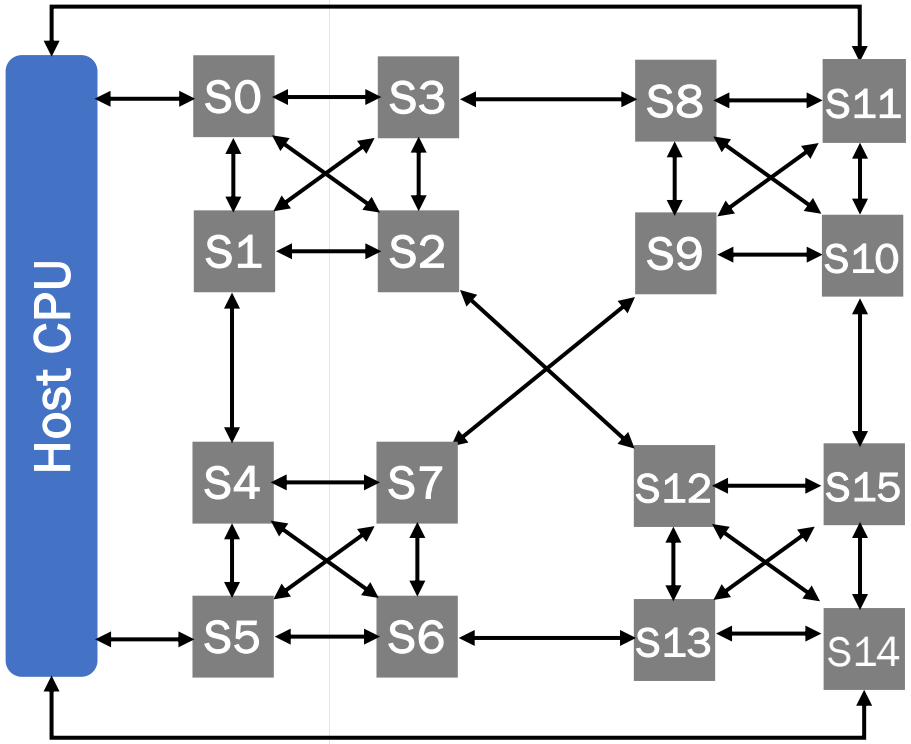
- A virtual address is translated to a physical address by walking 4 levels of page table



Radix Based Translation in PIM

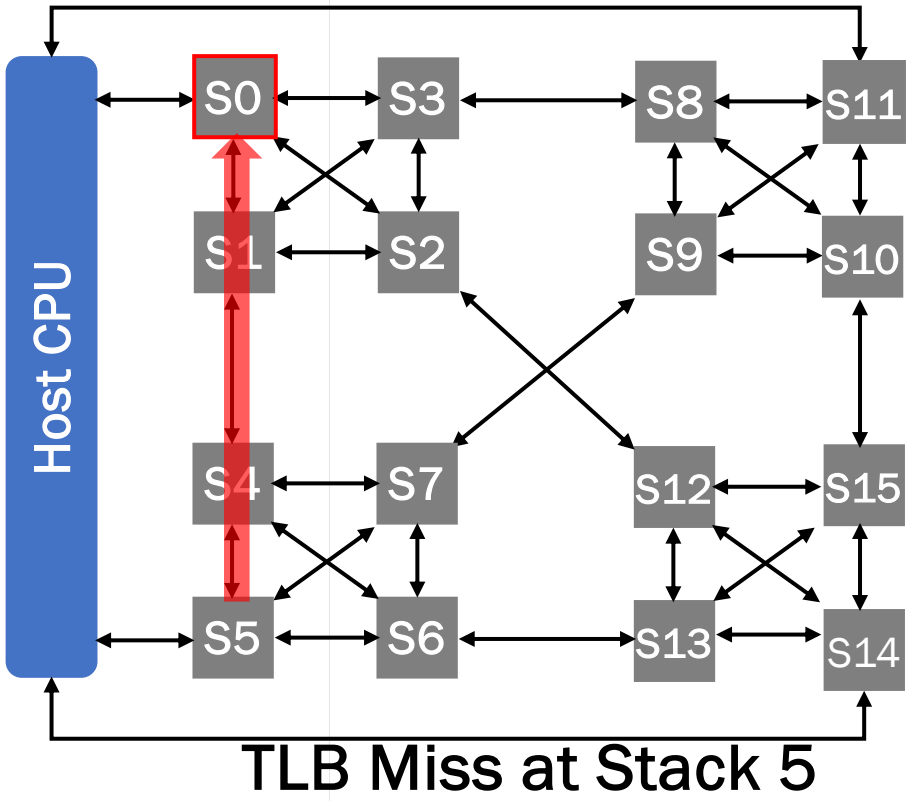
Radix Based Translation in PIM

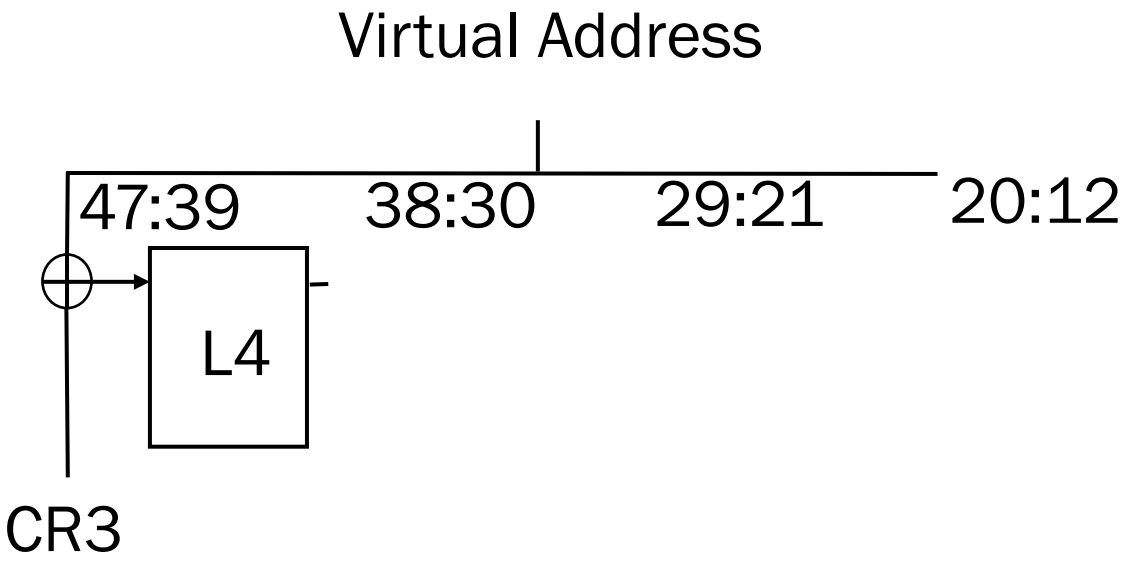
- Page Tables are distributed across the PIM stacks



Radix Based Translation in PIM

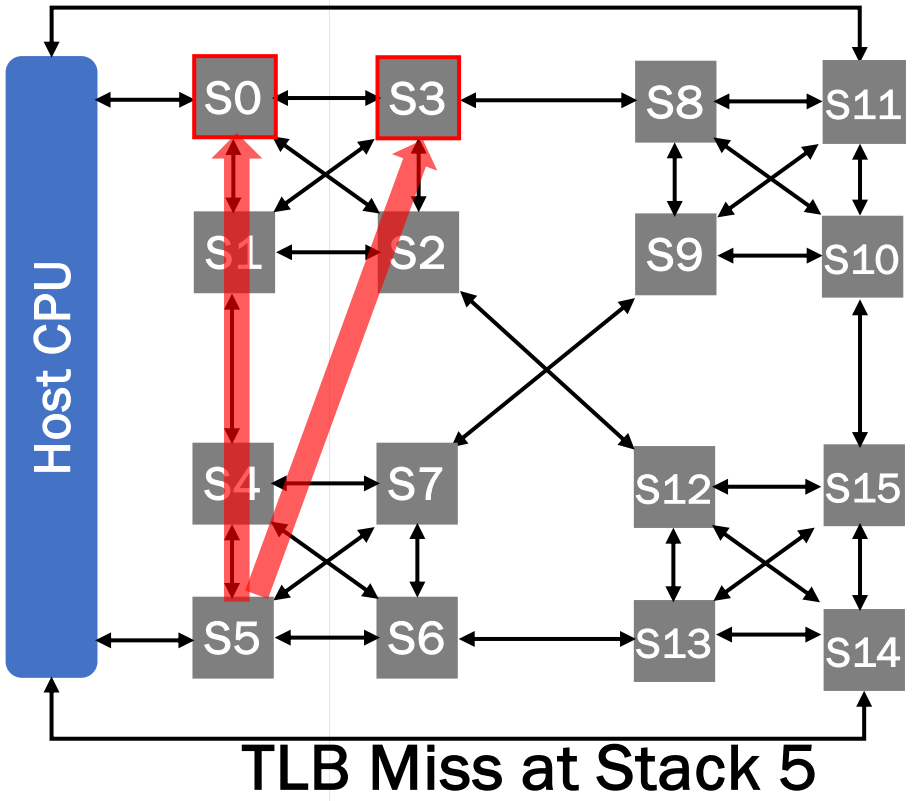
- Page Tables are distributed across the PIM stacks
- Each translation can require up to four accesses to different PIM stacks

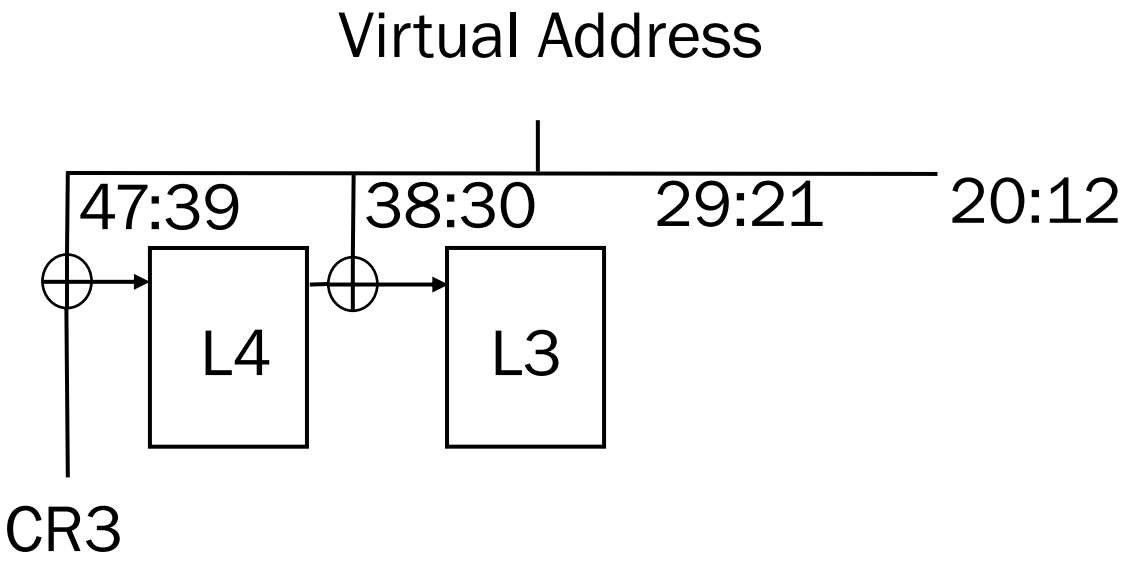




Radix Based Translation in PIM

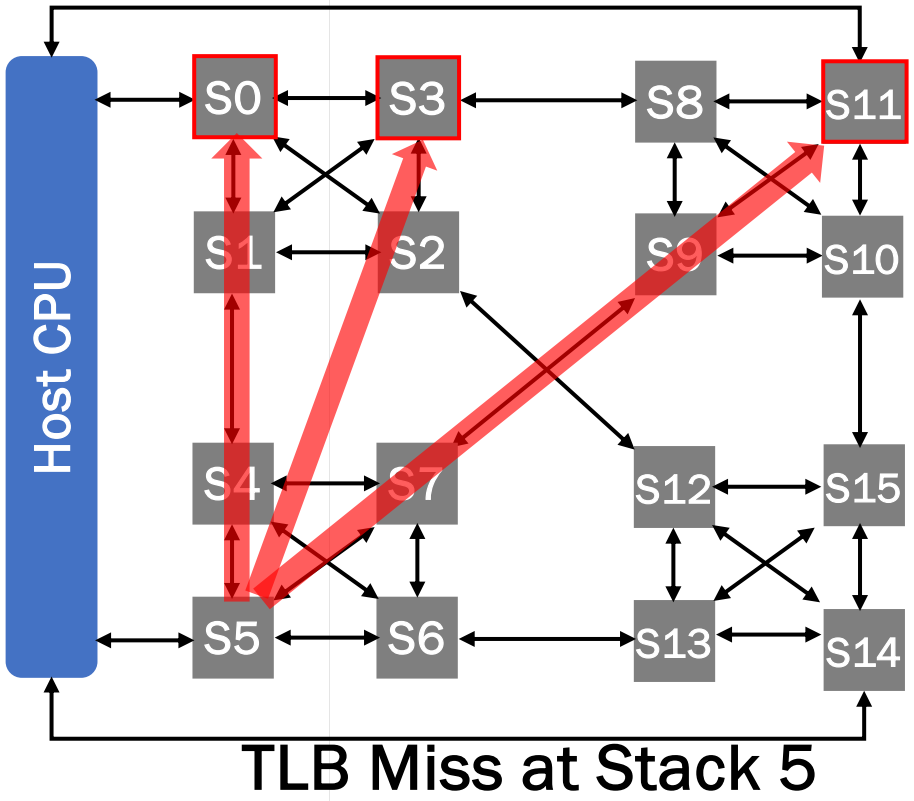
- Page Tables are distributed across the PIM stacks
- Each translation can require up to four accesses to different PIM stacks

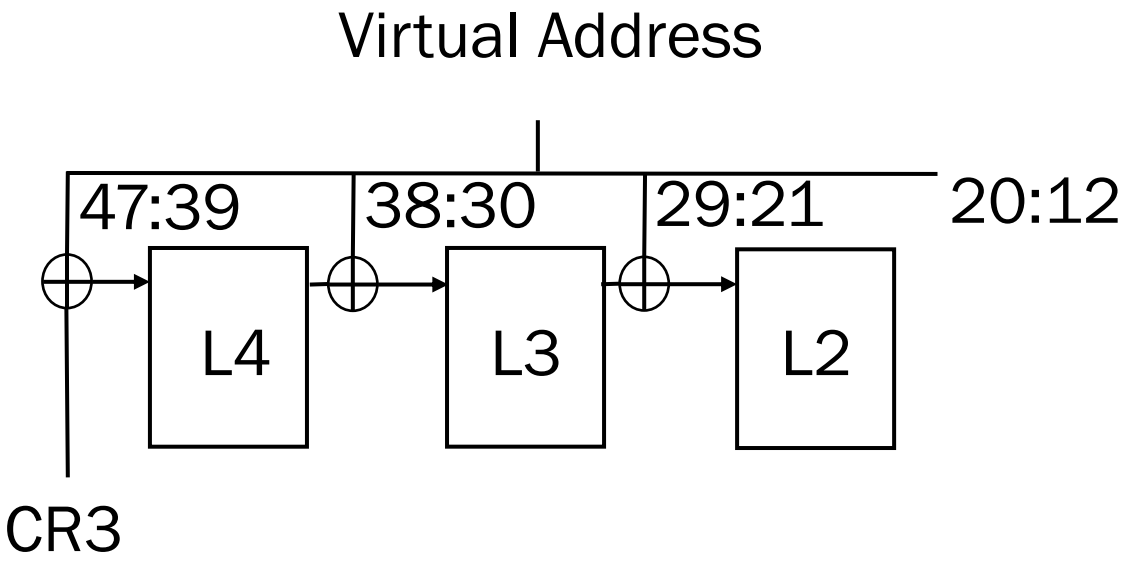




Radix Based Translation in PIM

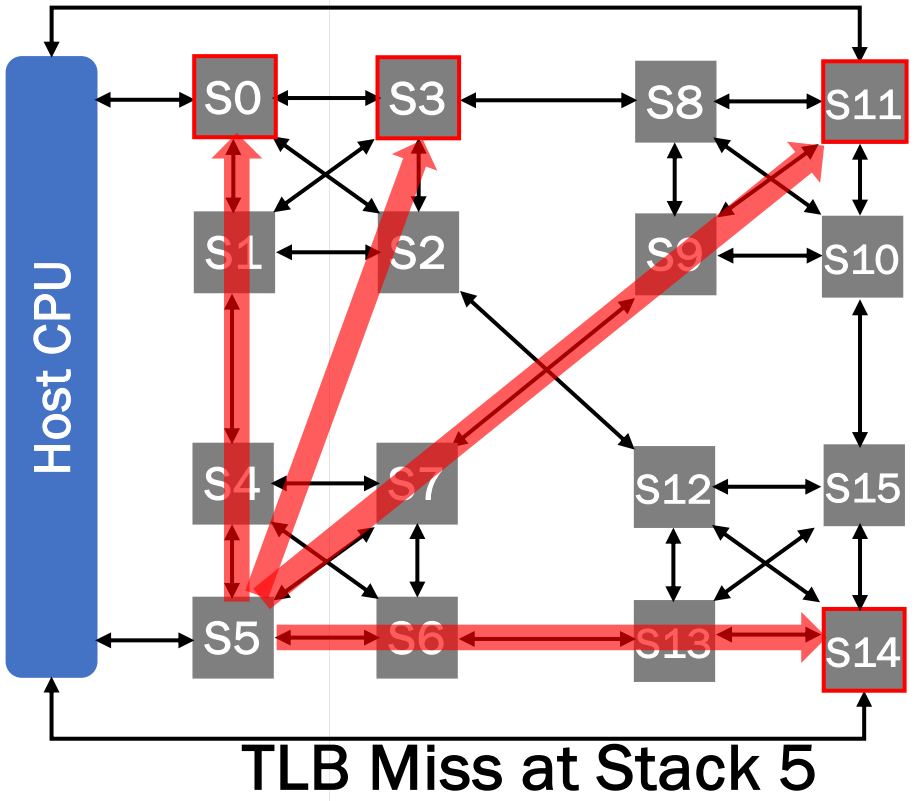
- Page Tables are distributed across the PIM stacks
- Each translation can require up to four accesses to different PIM stacks

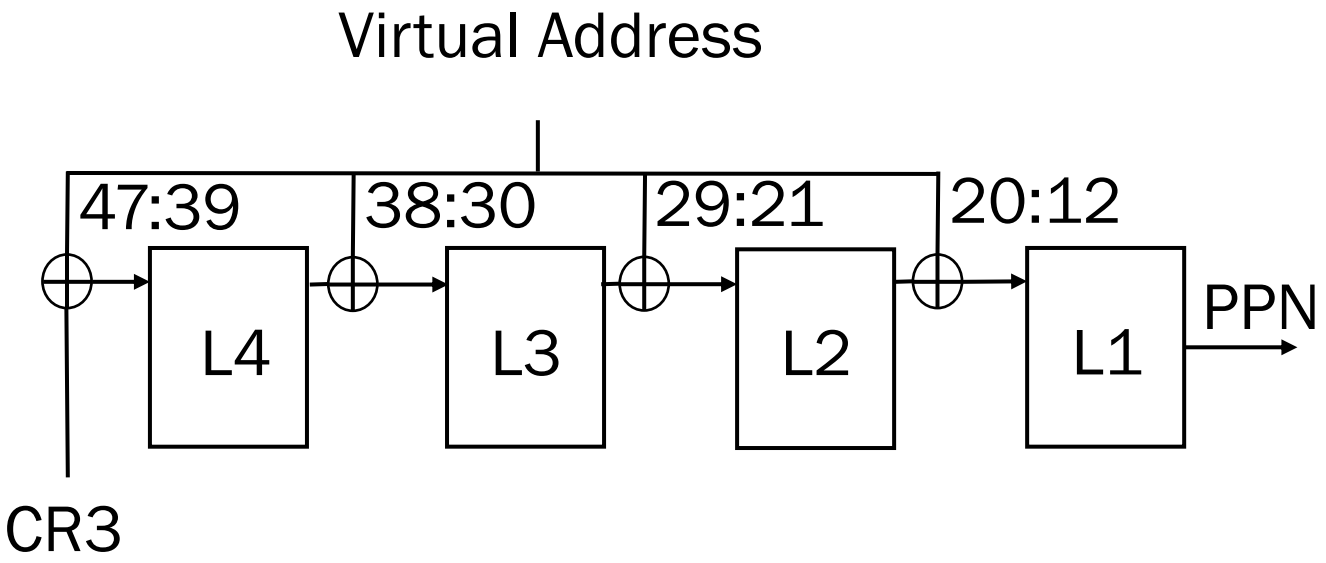




Radix Based Translation in PIM

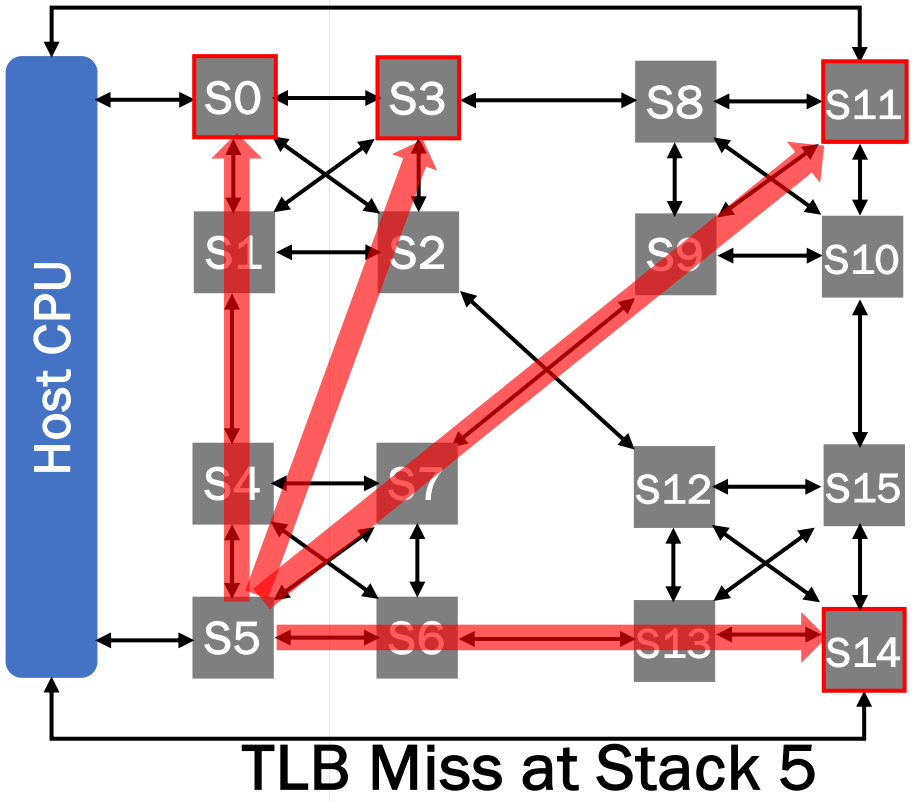
- Page Tables are distributed across the PIM stacks
- Each translation can require up to four accesses to different PIM stacks

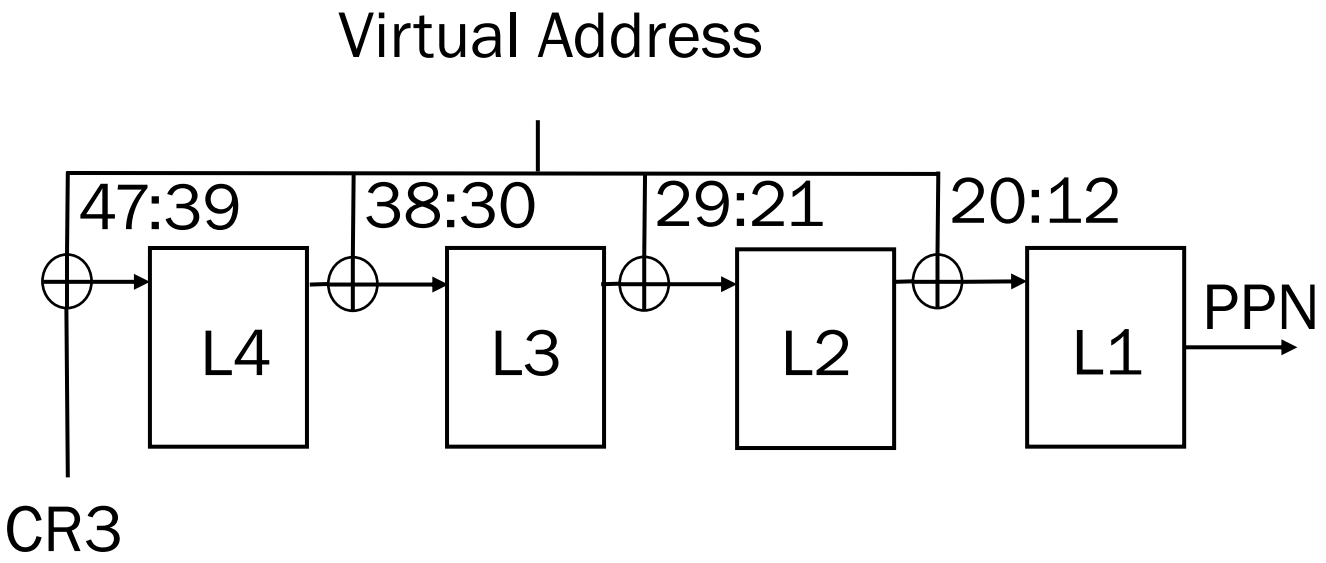




Radix Based Translation in PIM

- Page Tables are distributed across the PIM stacks
- Each translation can require up to four accesses to different PIM stacks

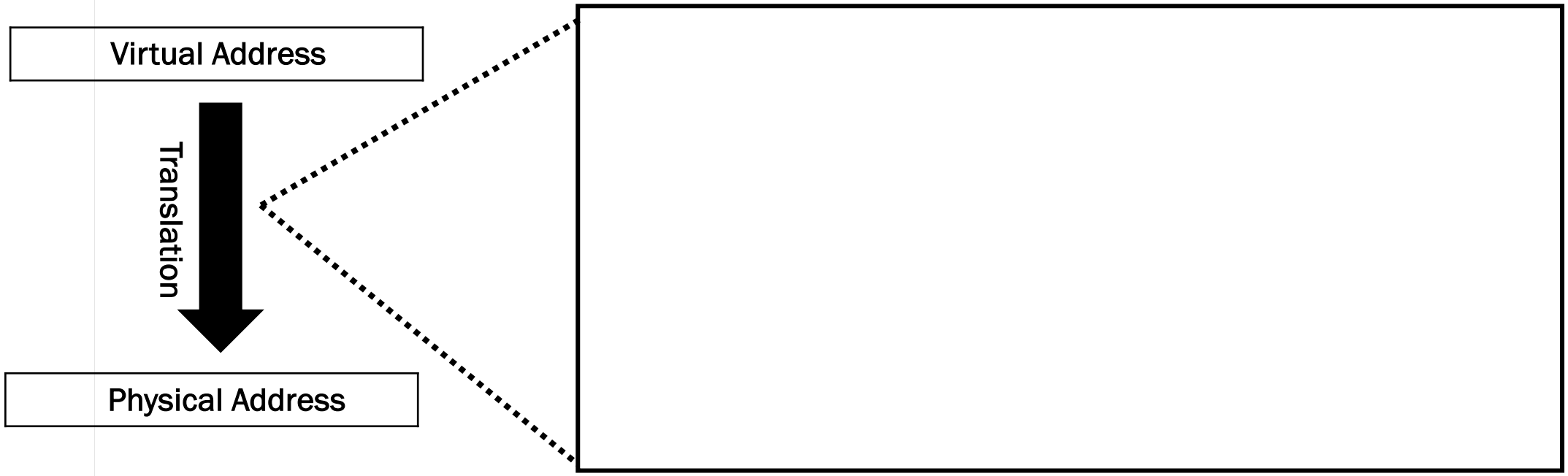




A single radix page walk can introduce sequential accesses to multiple memory stacks

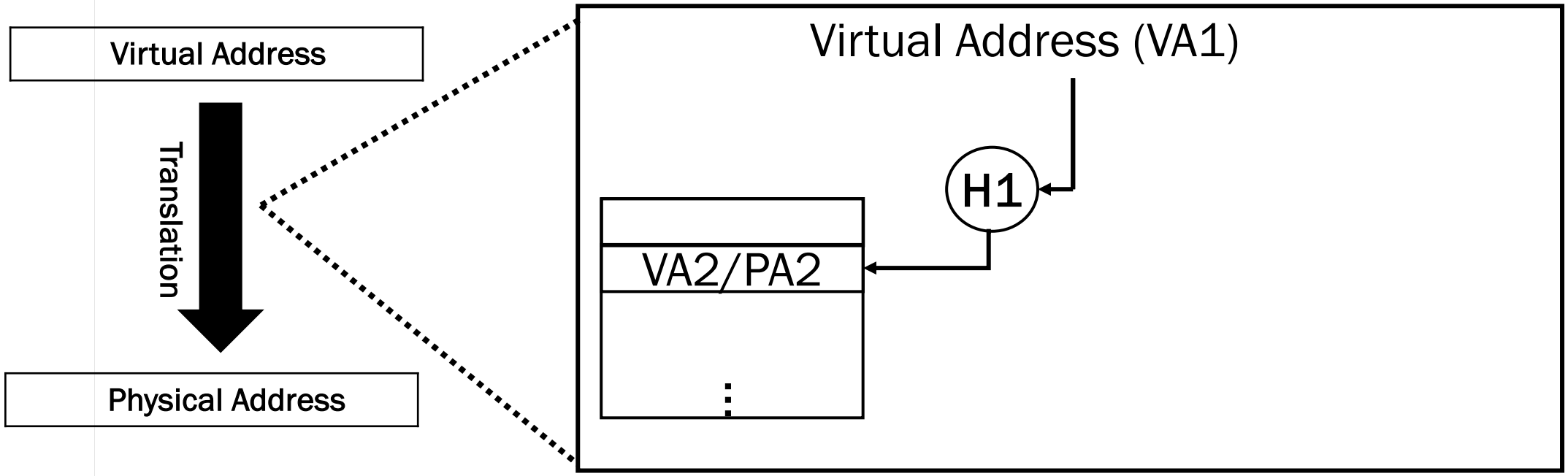
Hash Based Address Translation

- Hash based translation use hash table to lookup a physical address



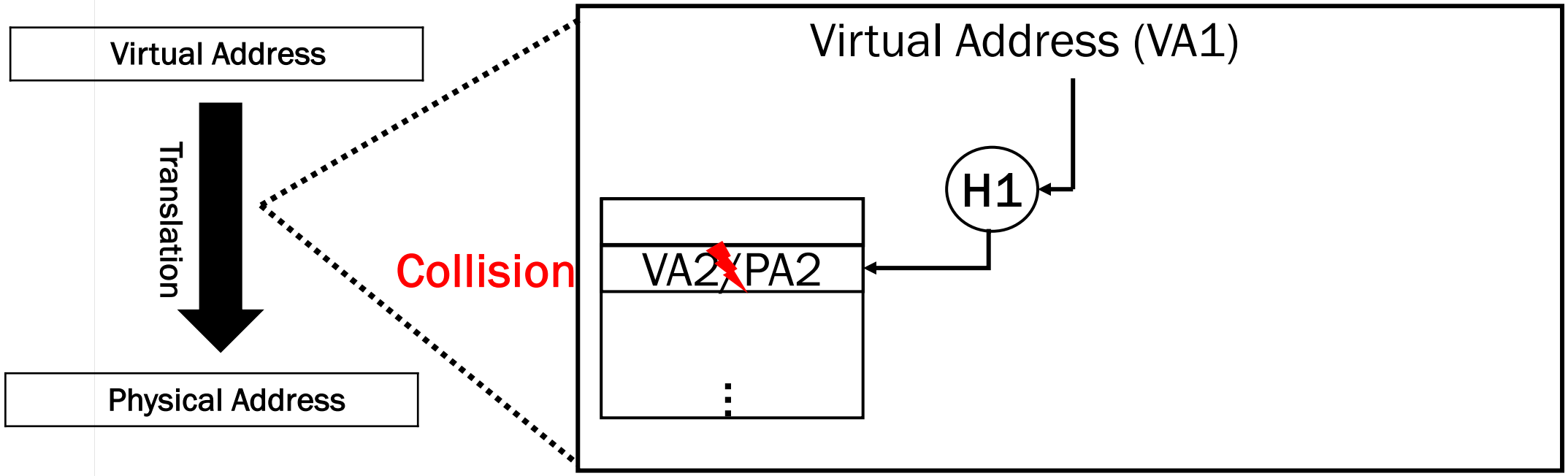
Hash Based Address Translation

- Hash based translation use hash table to lookup a physical address



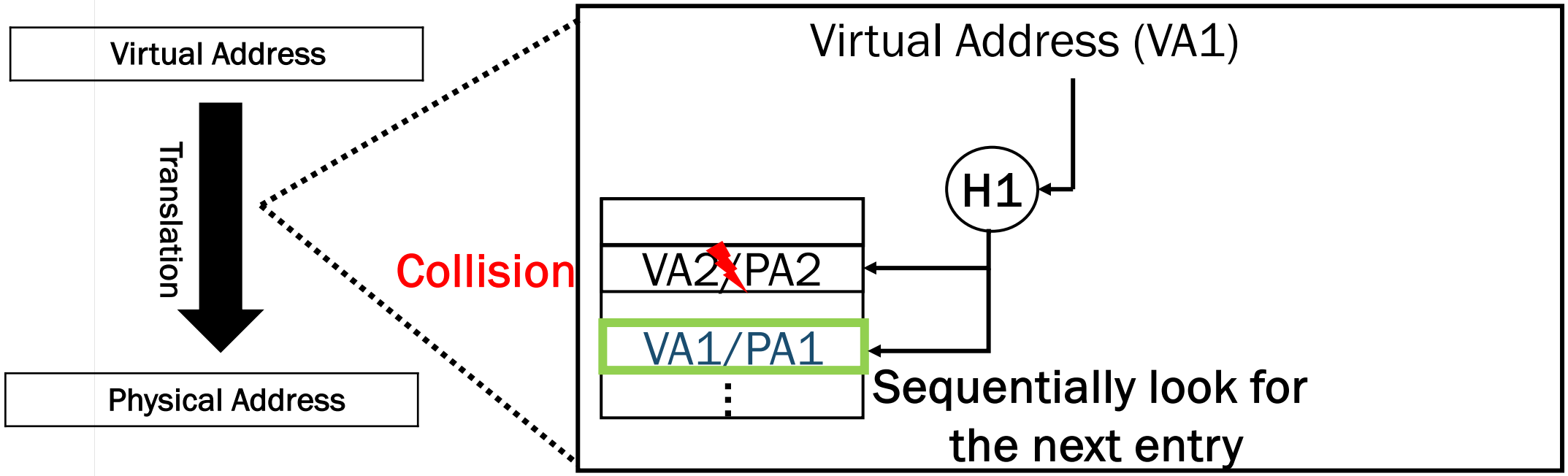
Hash Based Address Translation

- Hash based translation use hash table to lookup a physical address
- Hash page table suffers from collision overhead



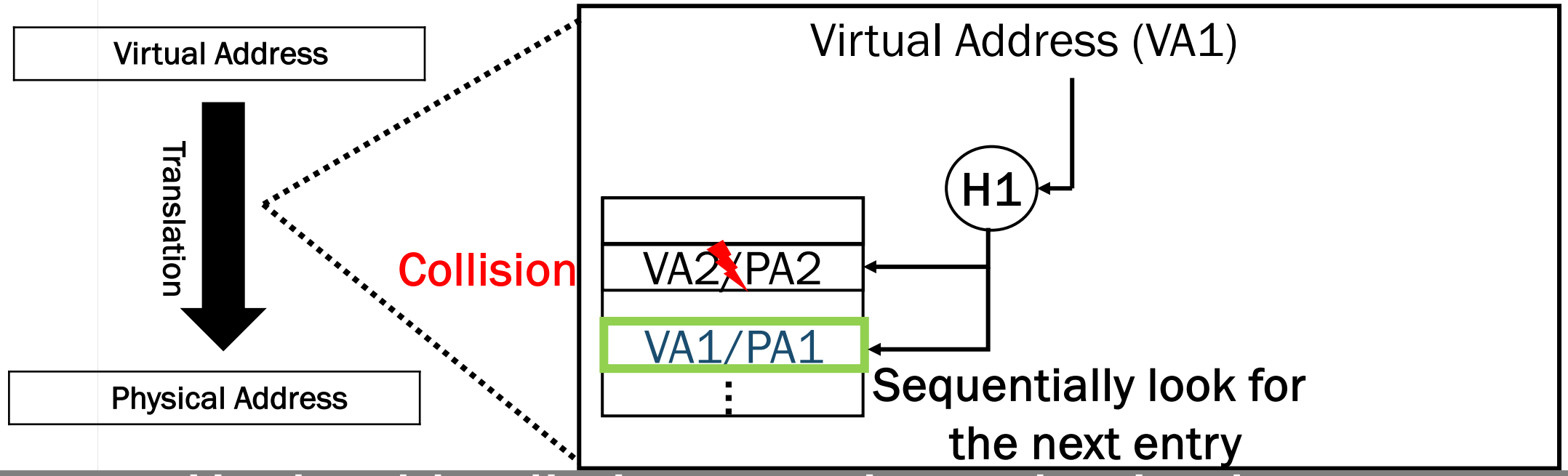
Hash Based Address Translation

- Hash based translation use hash table to lookup a physical address
- Hash page table suffers from collision overhead
- Collisions lead to multiple sequential accesses in a hash table



Hash Based Address Translation

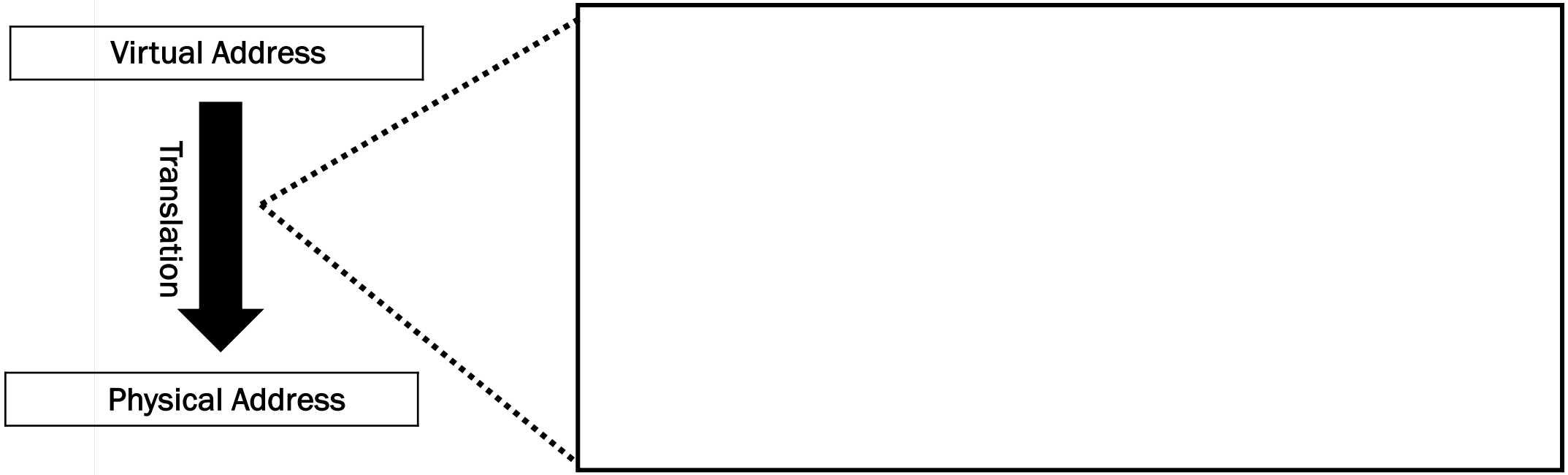
- Hash based translation use hash table to lookup a physical address
- Hash page table suffers from collision overhead
- Collisions lead to multiple sequential accesses in a hash table



Hash table eliminates pointer chasing but introduces sequential accesses due to collisions.

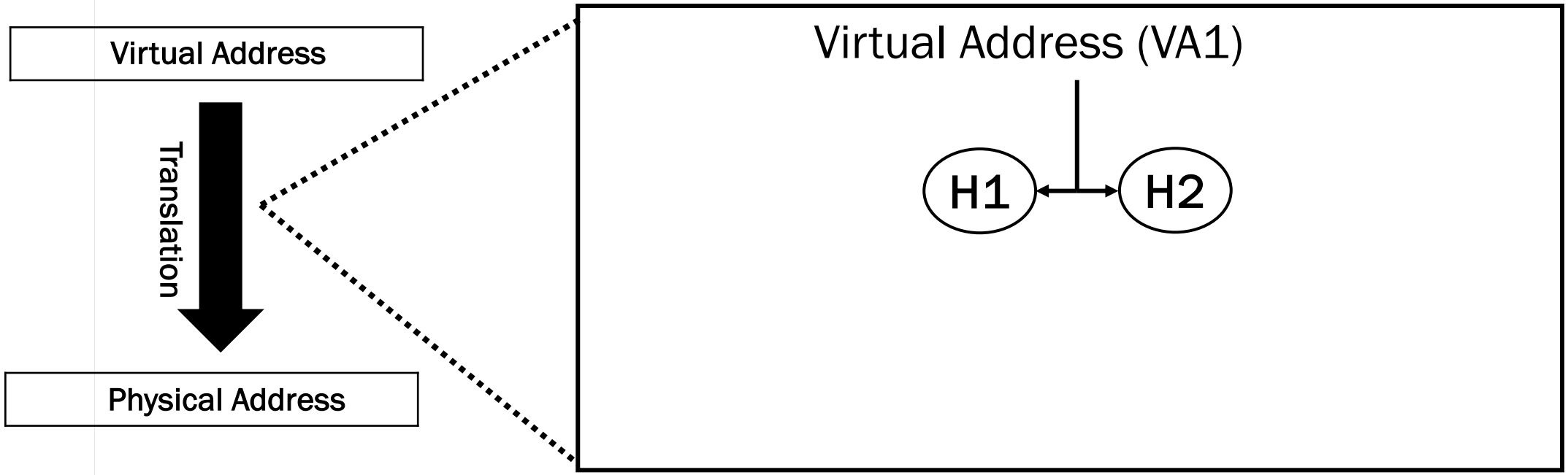
Alternative: Cuckoo Hash Table

- Multiple parallel accesses are issued in hash table to mask the overhead of collisions



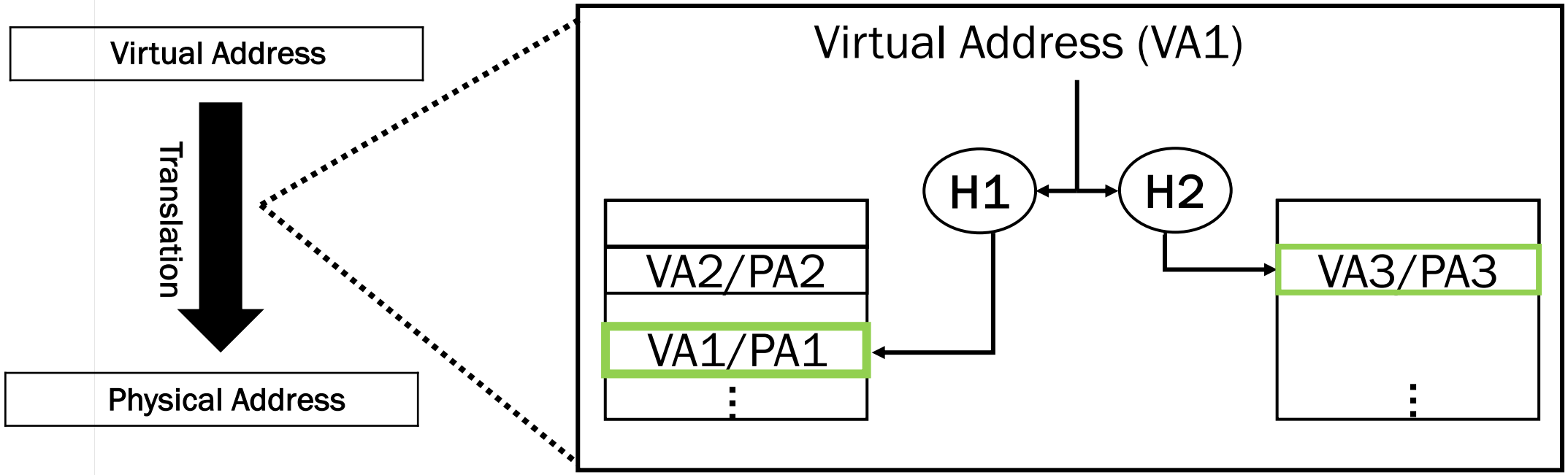
Alternative: Cuckoo Hash Table

- Multiple parallel accesses are issued in hash table to mask the overhead of collisions



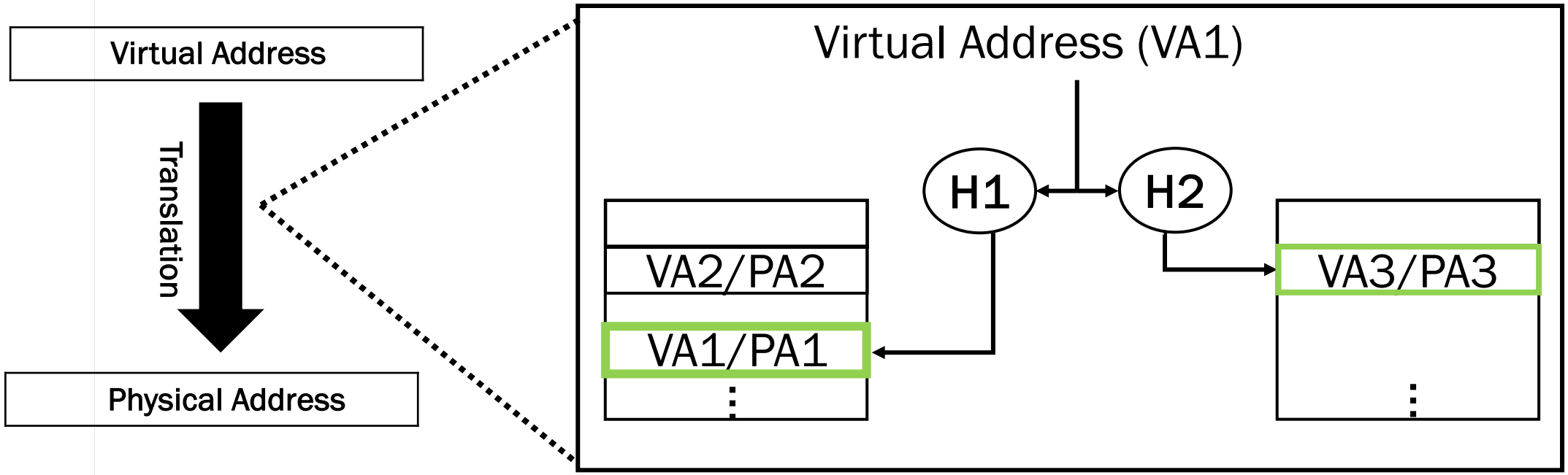
Alternative: Cuckoo Hash Table

- Multiple parallel accesses are issued in hash table to mask the overhead of collisions



Alternative: Cuckoo Hash Table

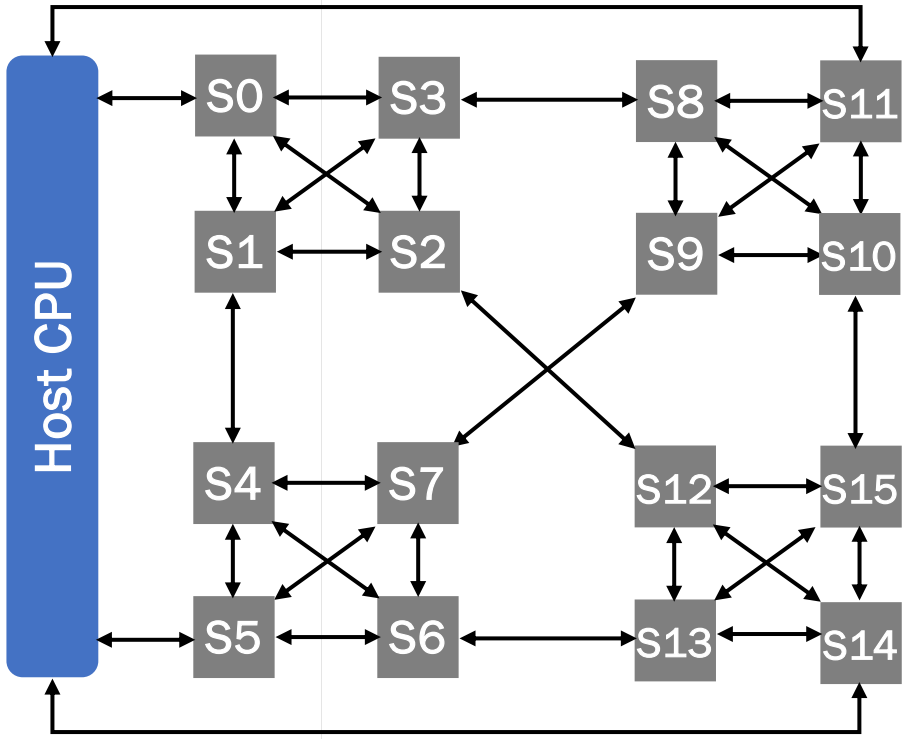
- Multiple parallel accesses are issued in hash table to mask the overhead of collisions



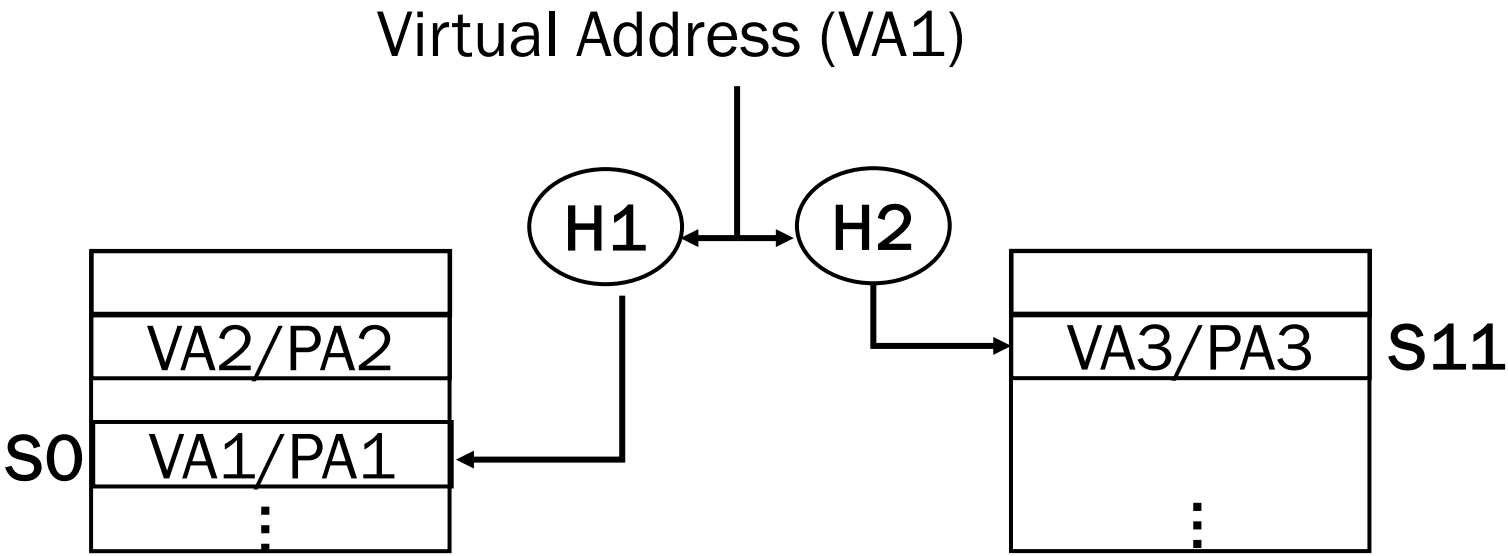
Cuckoo hash table enables parallel lookups upon collisions

Alternative: Cuckoo Hash Table in PIM

- Parallel accesses are issued across different stacks in PIM

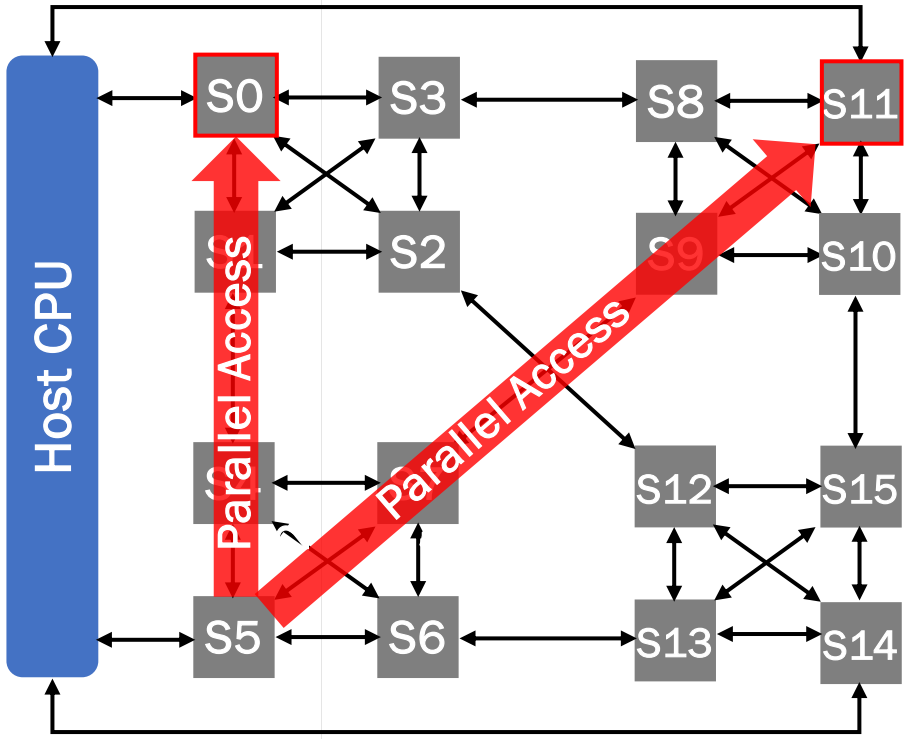


TLB Miss at Stack 5

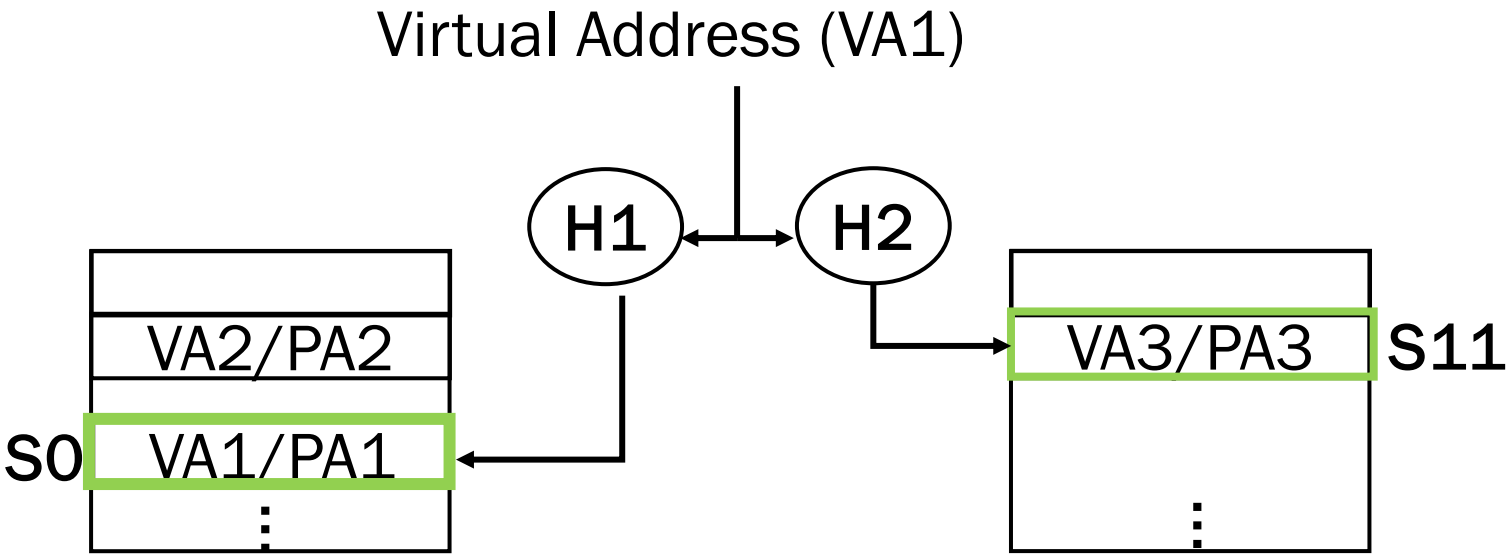


Alternative: Cuckoo Hash Table in PIM

- Parallel accesses are issued across different stacks in PIM

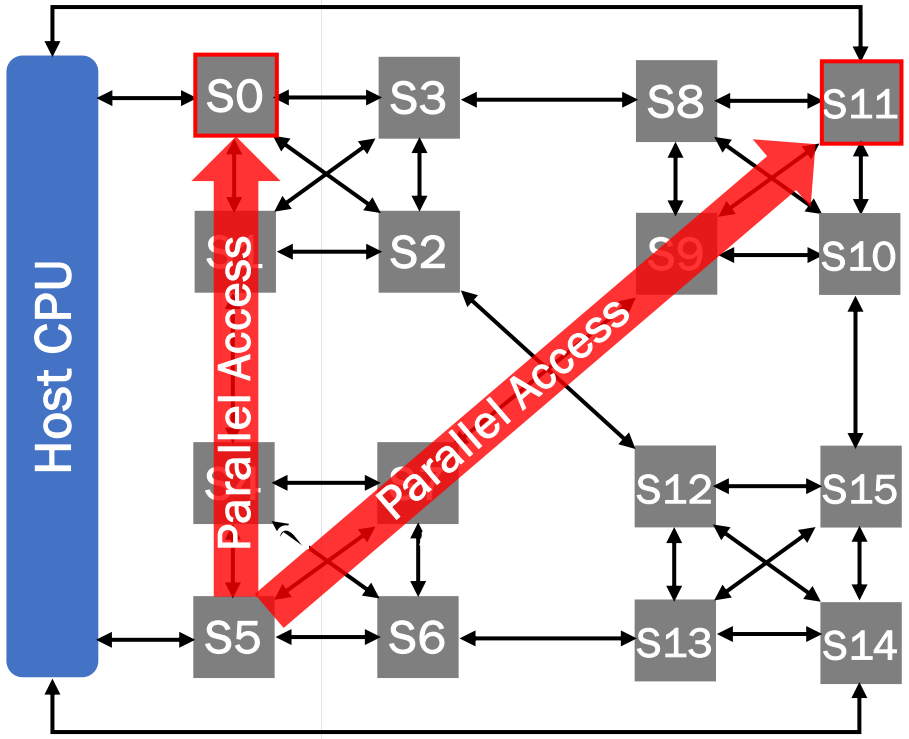


TLB Miss at Stack 5

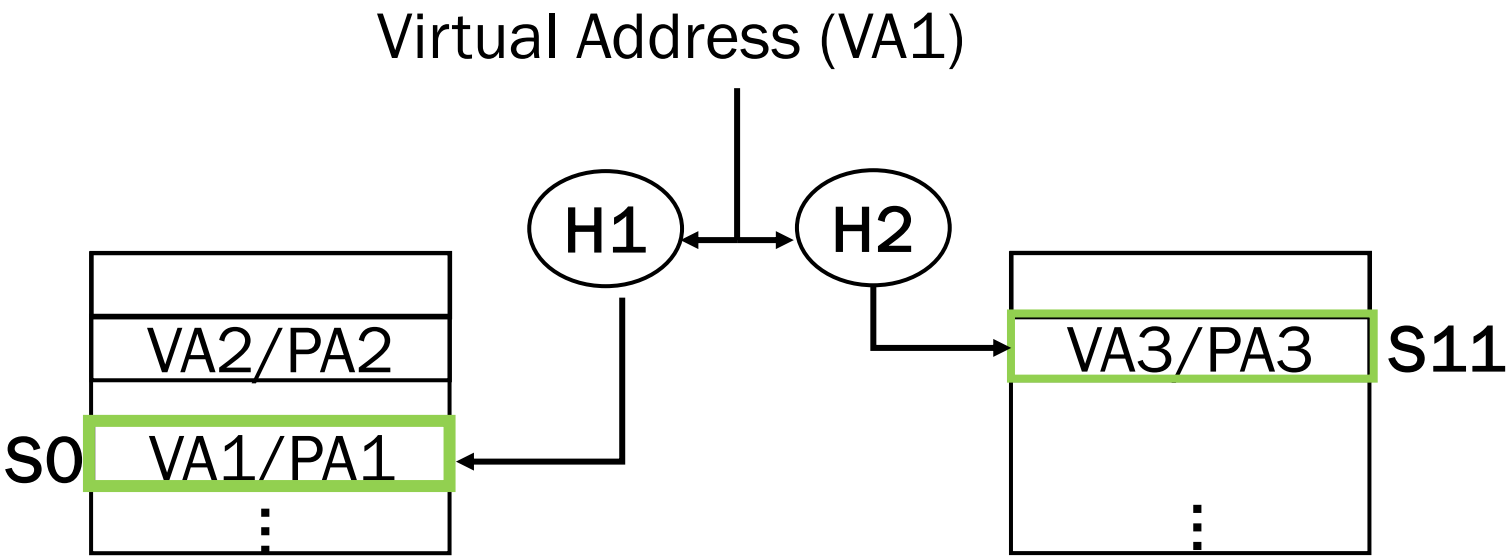


Alternative: Cuckoo Hash Table in PIM

- Parallel accesses are issued across different stacks in PIM



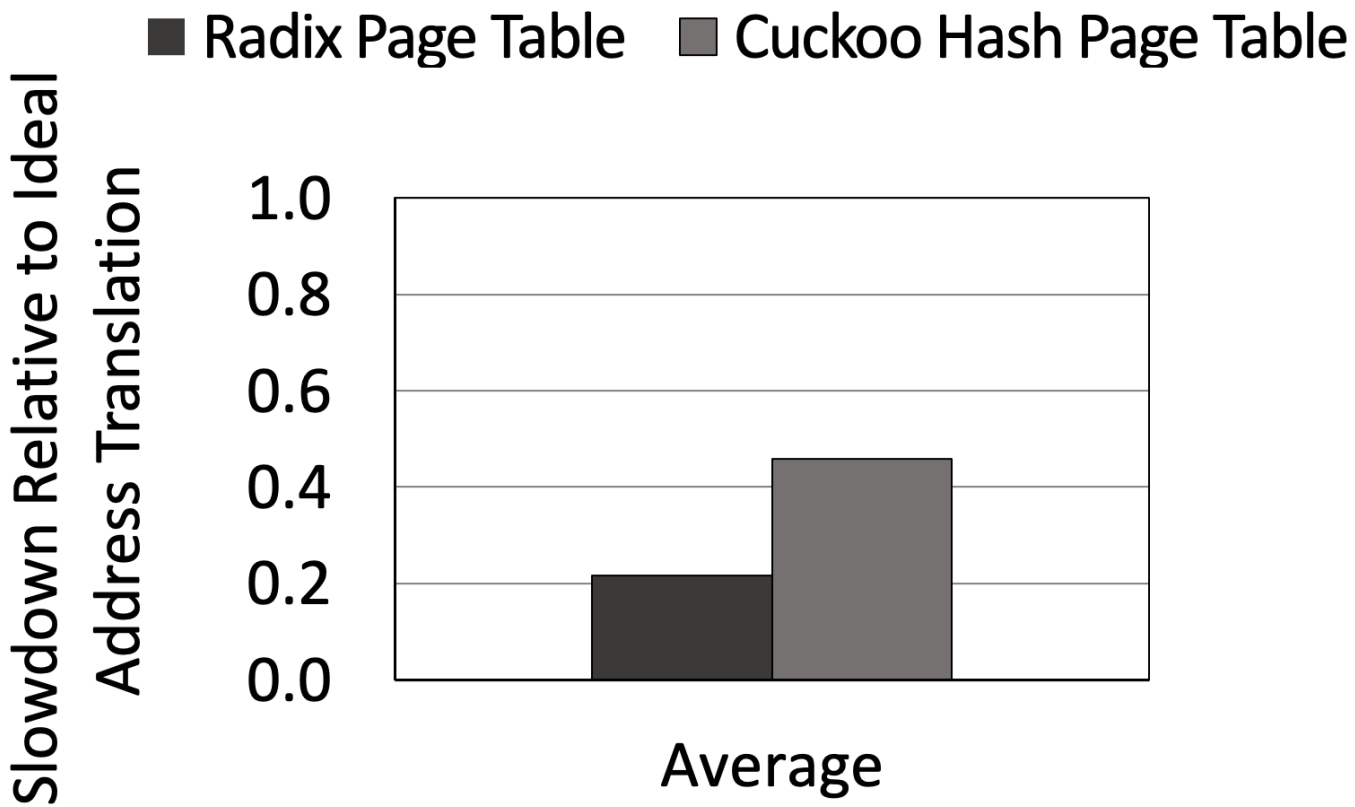
TLB Miss at Stack 5



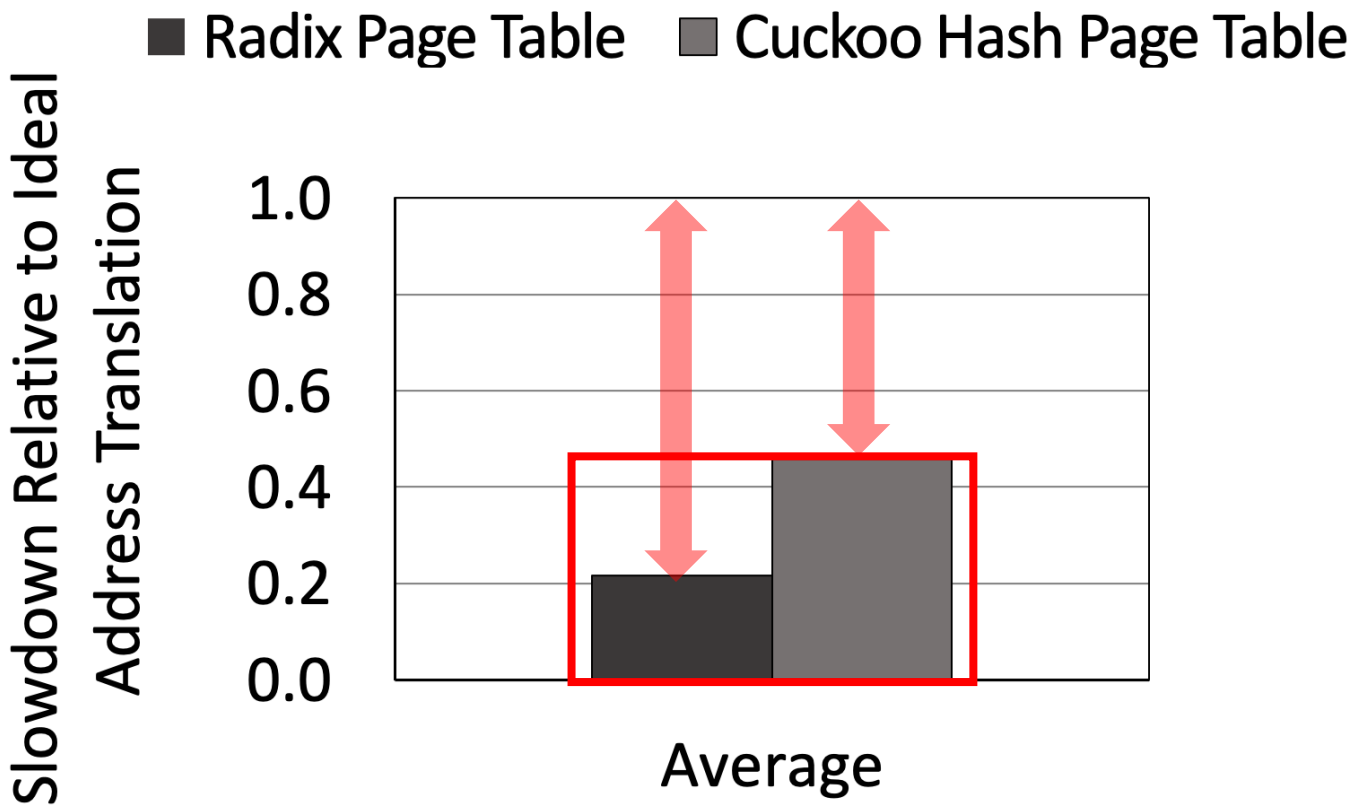
Parallel cross-stack accesses increase network contention and thus overall memory access time

Overhead of Hash and Radix Translation in PIM

Overhead of Hash and Radix Translation in PIM



Overhead of Hash and Radix Translation in PIM



Naïve integration of radix and cuckoo hash page table cause significant slowdown in a PIM based system

Goal

Goal

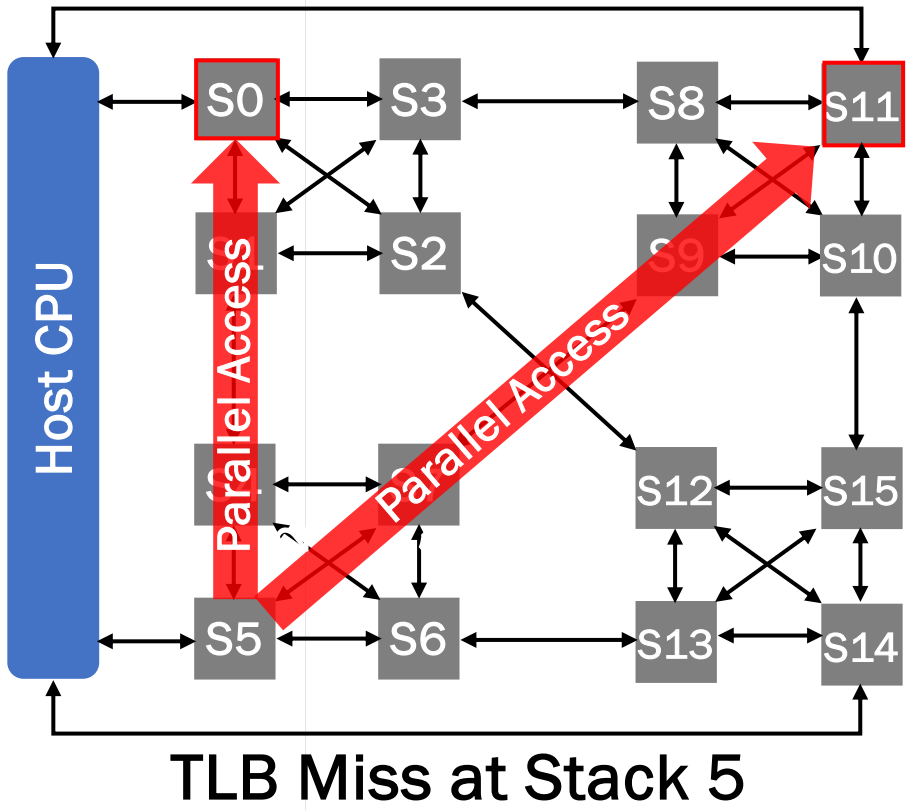
Design an efficient address translation scheme for multi-stack PIM systems

Outline

- Processing-in-Memory Introduction
- Address Translation in PIM Architecture
- Challenges and Key Ideas
- Evaluation
- Conclusion

Challenge: Parallel lookups increase contention

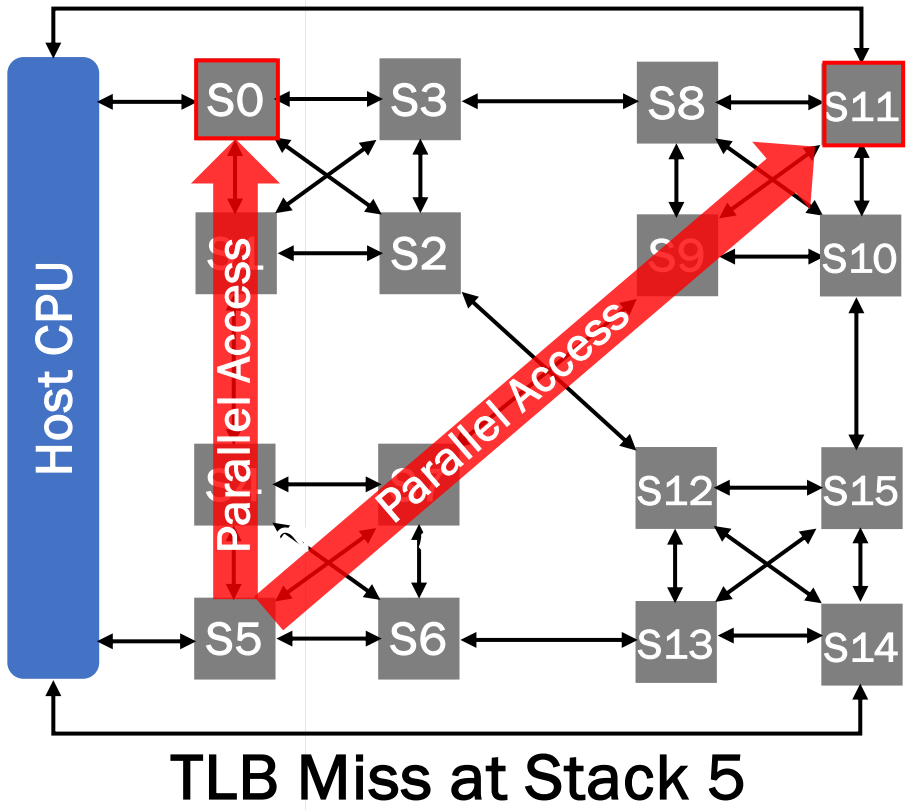
- Parallel lookups in cuckoo hash lead to increased memory network contention



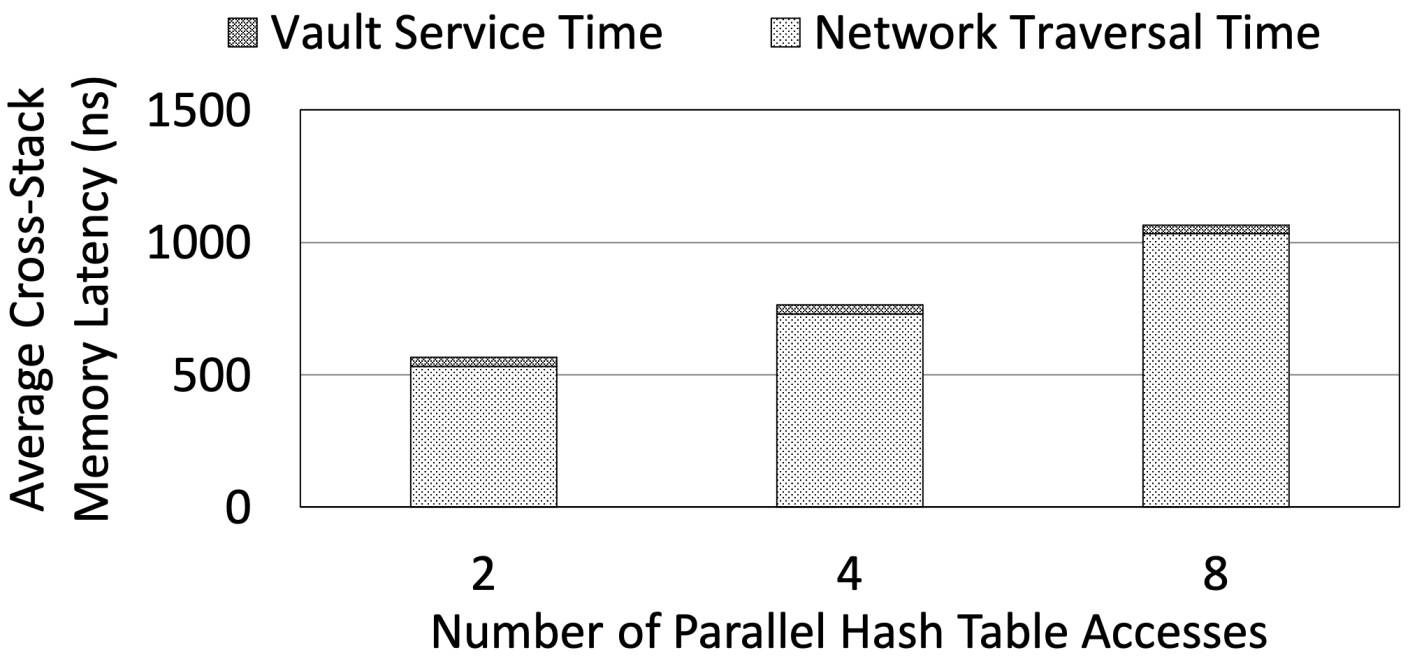
TLB Miss at Stack 5

Challenge: Parallel lookups increase contention

- Parallel lookups in cuckoo hash lead to increased memory network contention

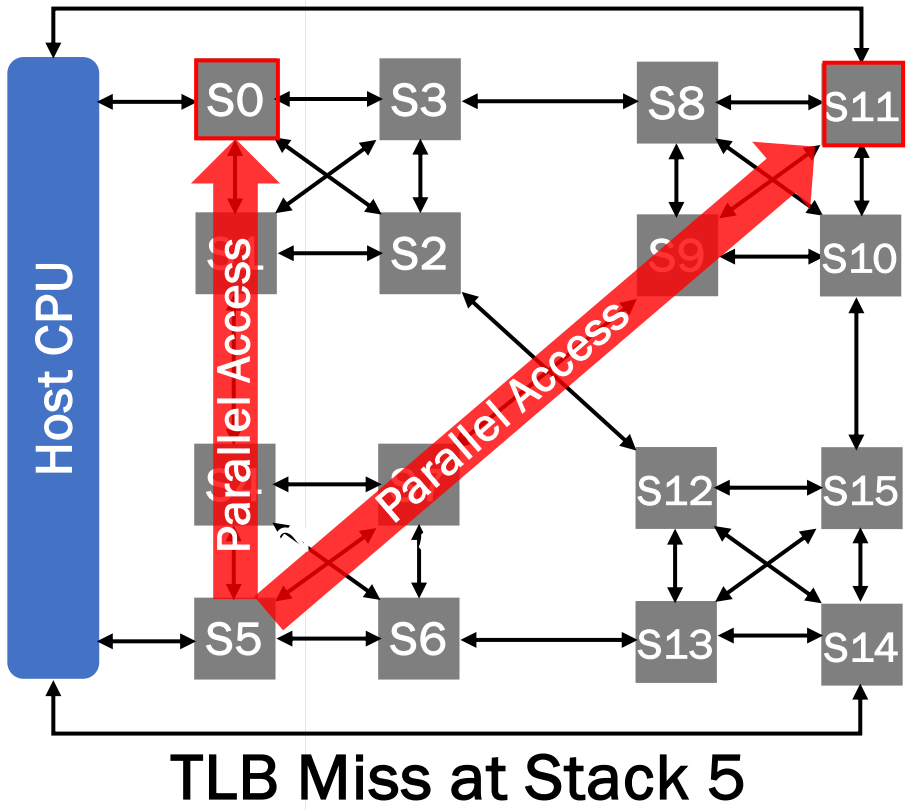


TLB Miss at Stack 5

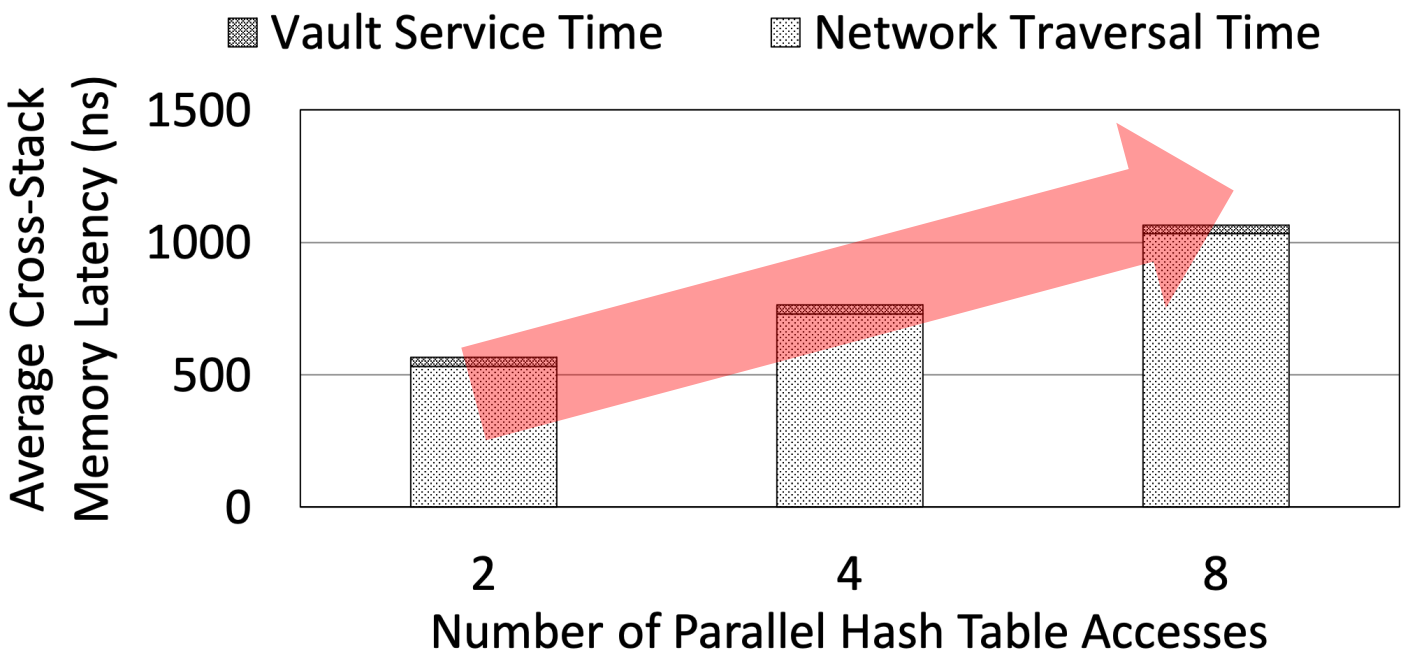


Challenge: Parallel lookups increase contention

- Parallel lookups in cuckoo hash lead to increased memory network contention

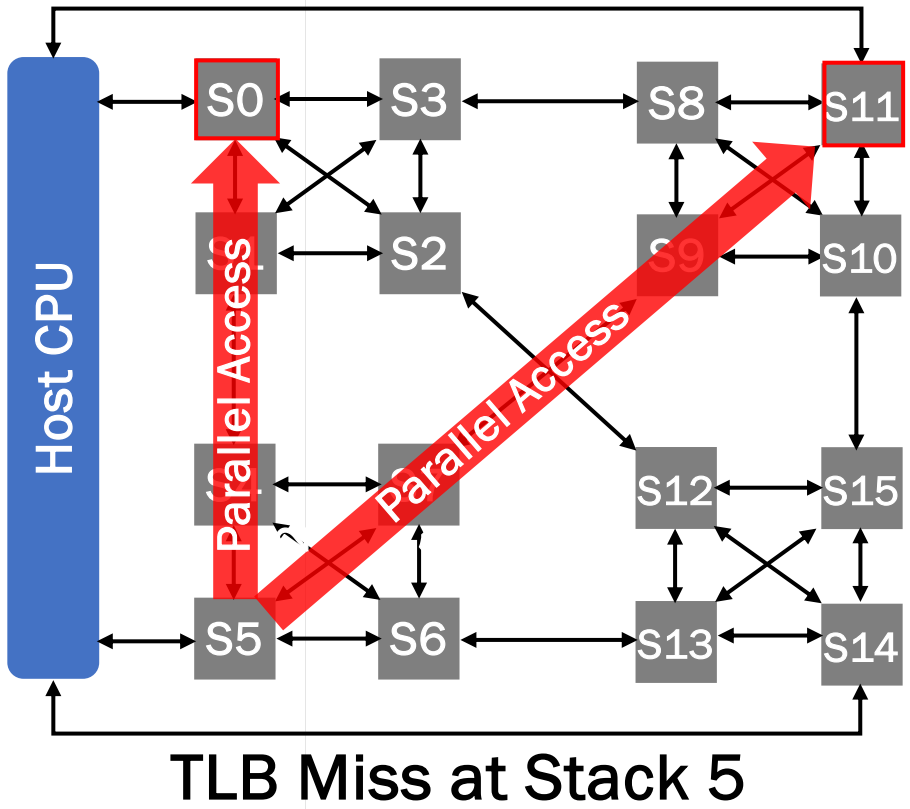


TLB Miss at Stack 5

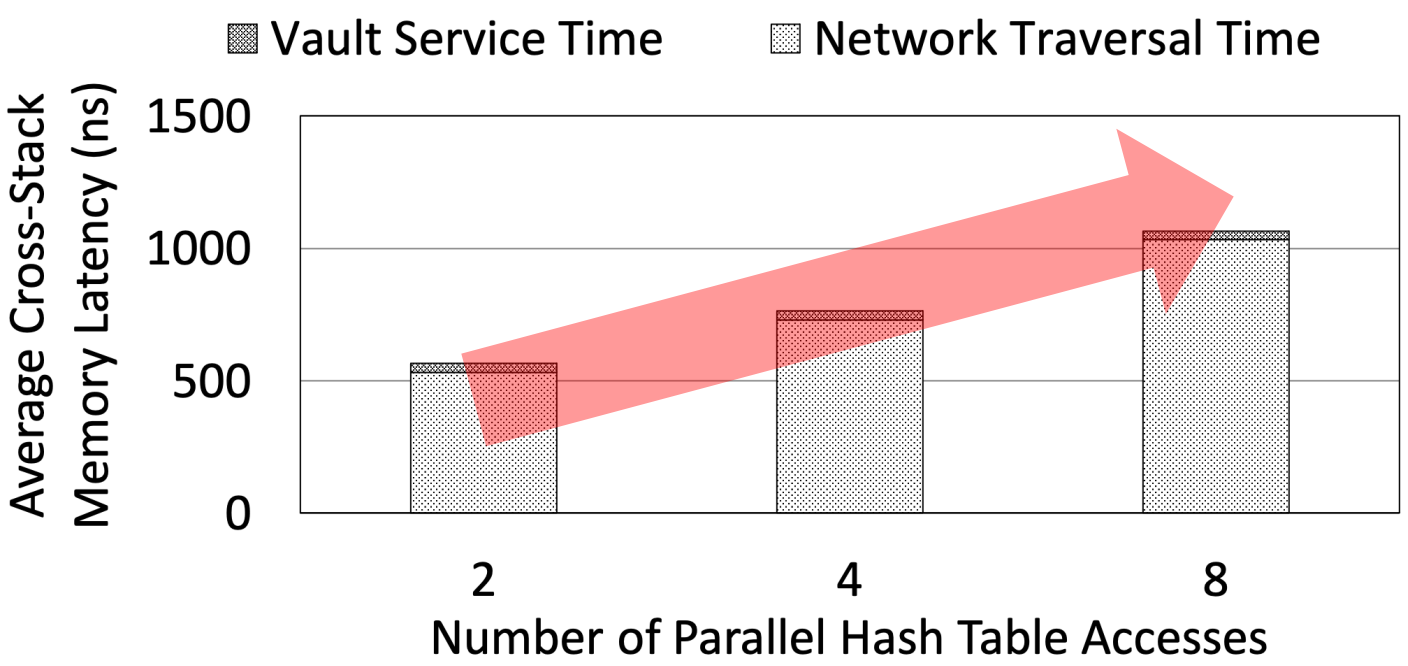


Challenge: Parallel lookups increase contention

- Parallel lookups in cuckoo hash lead to increased memory network contention

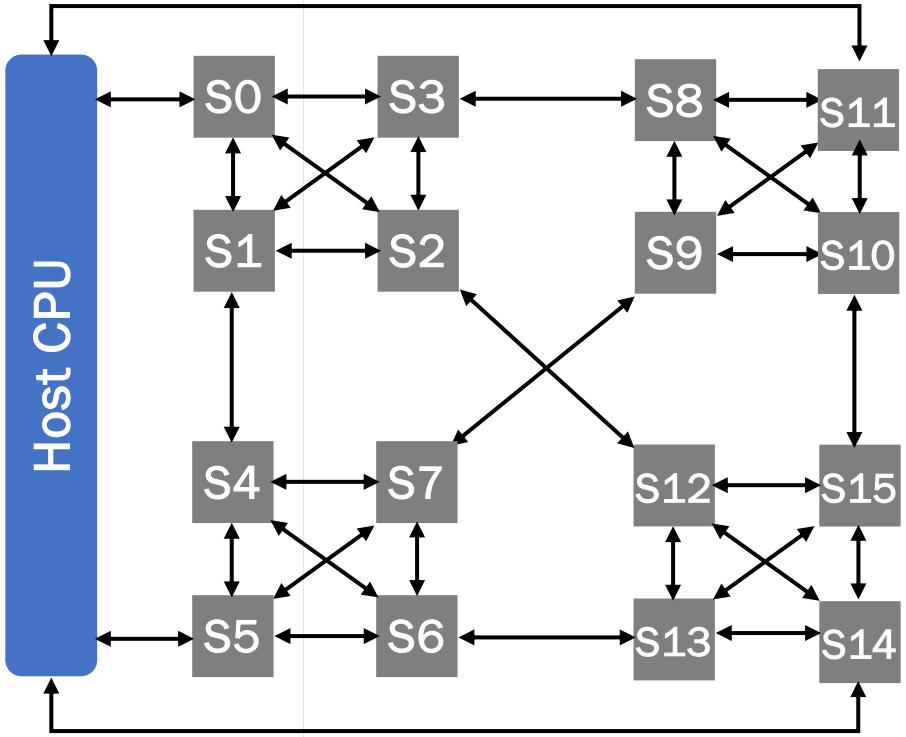


TLB Miss at Stack 5



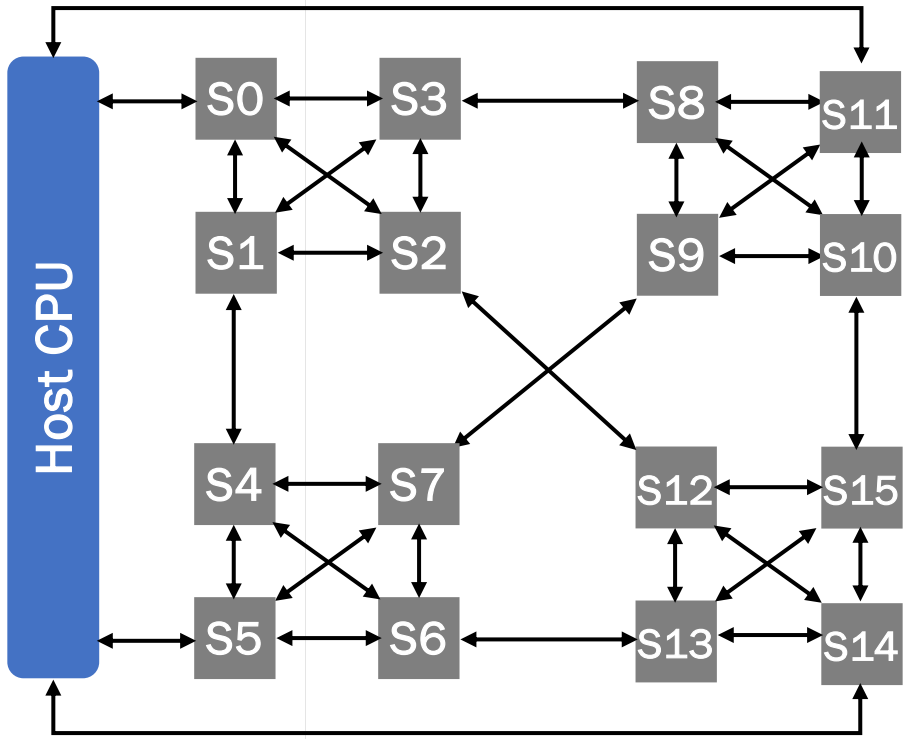
Cross-stack memory access latency increase with parallel accesses

Observation: Massive parallelism inside stack



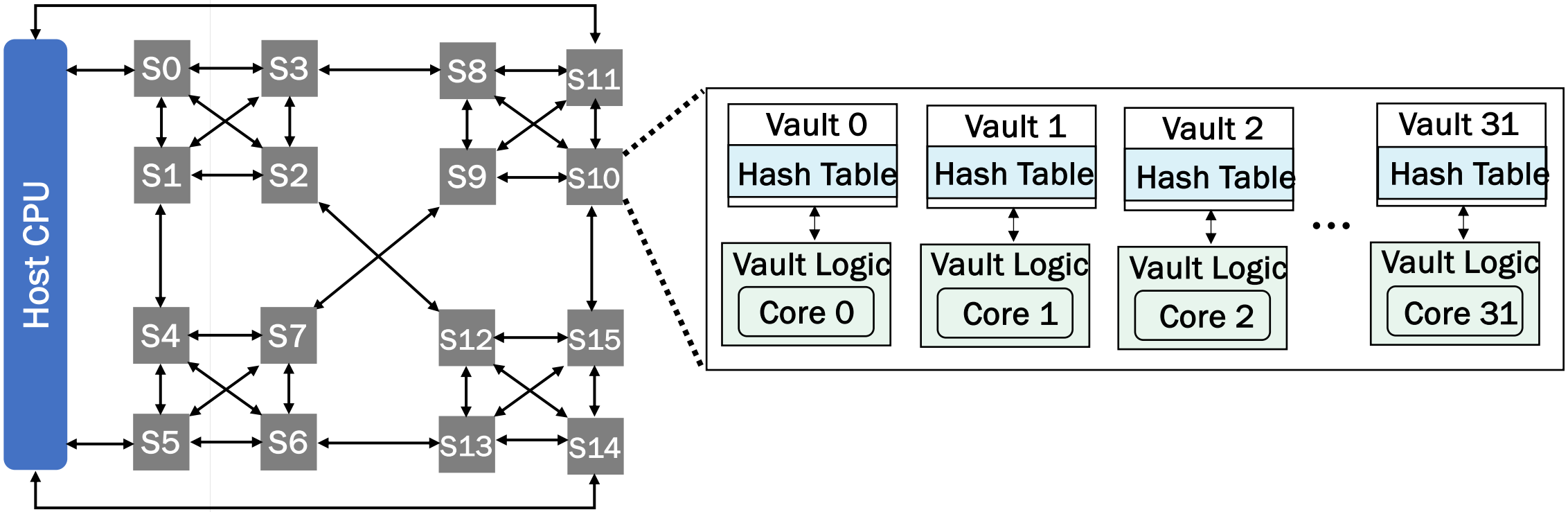
Observation: Massive parallelism inside stack

- We observe immense intra-stack parallelism for parallel accesses



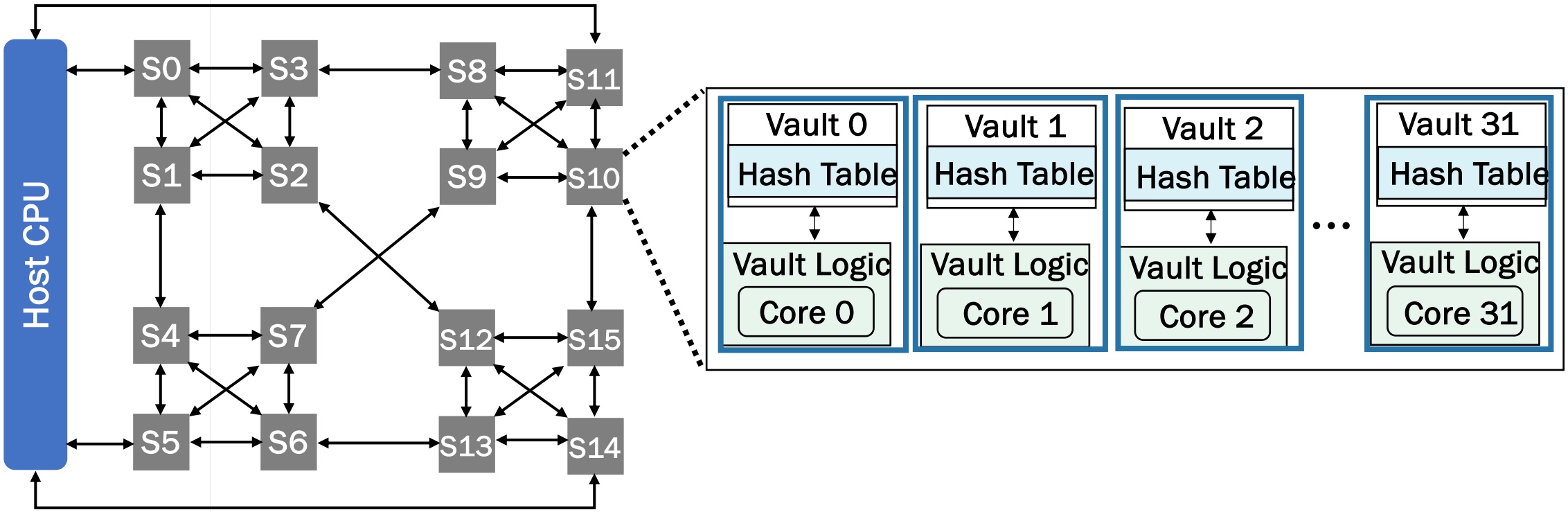
Observation: Massive parallelism inside stack

- We observe immense intra-stack parallelism for parallel accesses



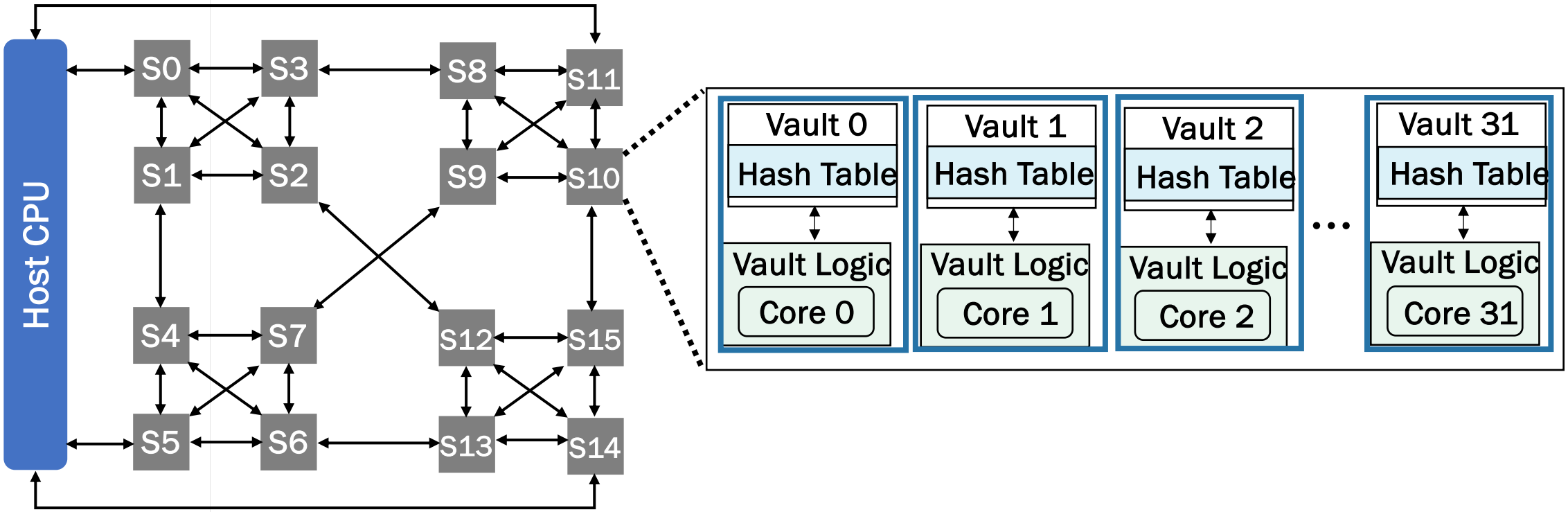
Observation: Massive parallelism inside stack

- We observe immense intra-stack parallelism for parallel accesses



Observation: Massive parallelism inside stack

- We observe immense intra-stack parallelism for parallel accesses



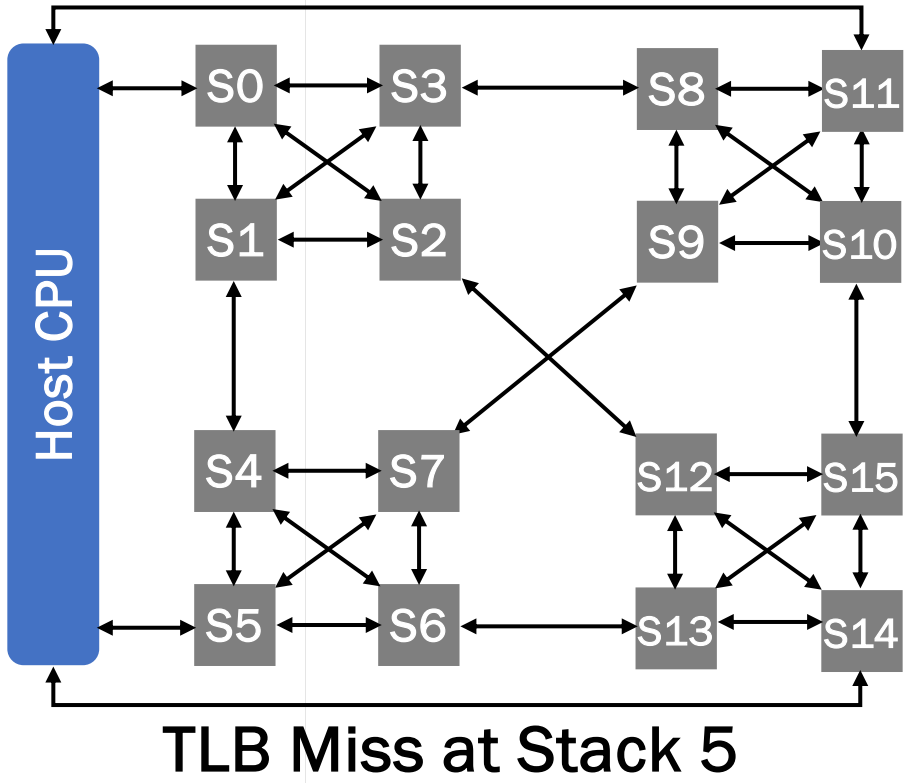
Network traversal can be avoided if parallel accesses are generated within the same stack and across different vaults

Key Idea: Network-contention-aware hash

- Parallel lookups generated within the stack avoid costly cross-stack page table walks

Key Idea: Network-contention-aware hash

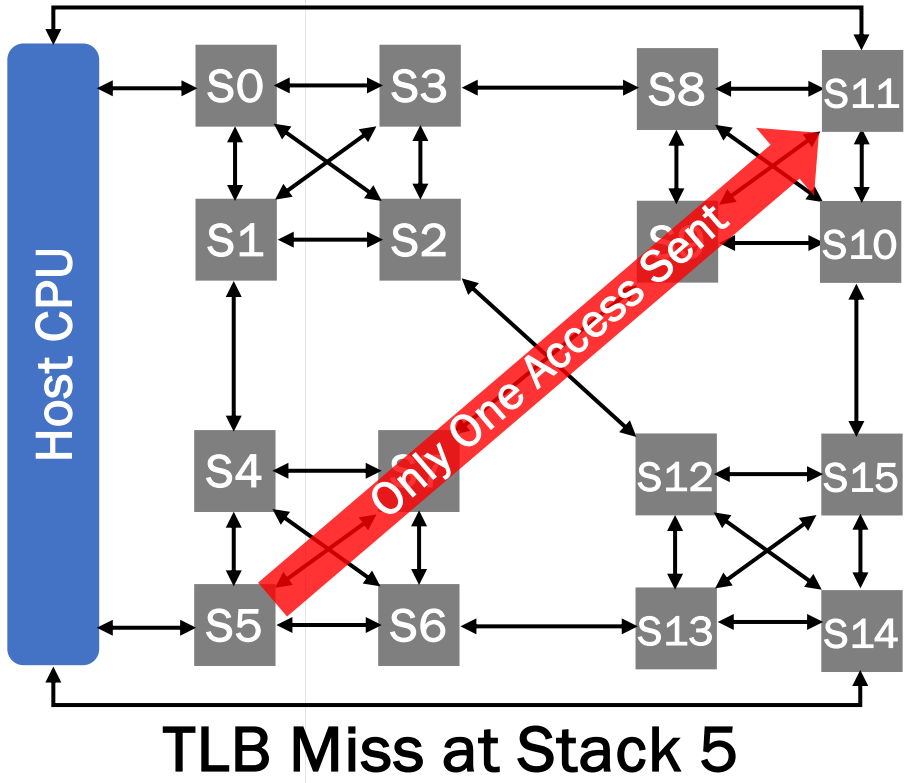
- Parallel lookups generated within the stack avoid costly cross-stack page table walks



TLB Miss at Stack 5

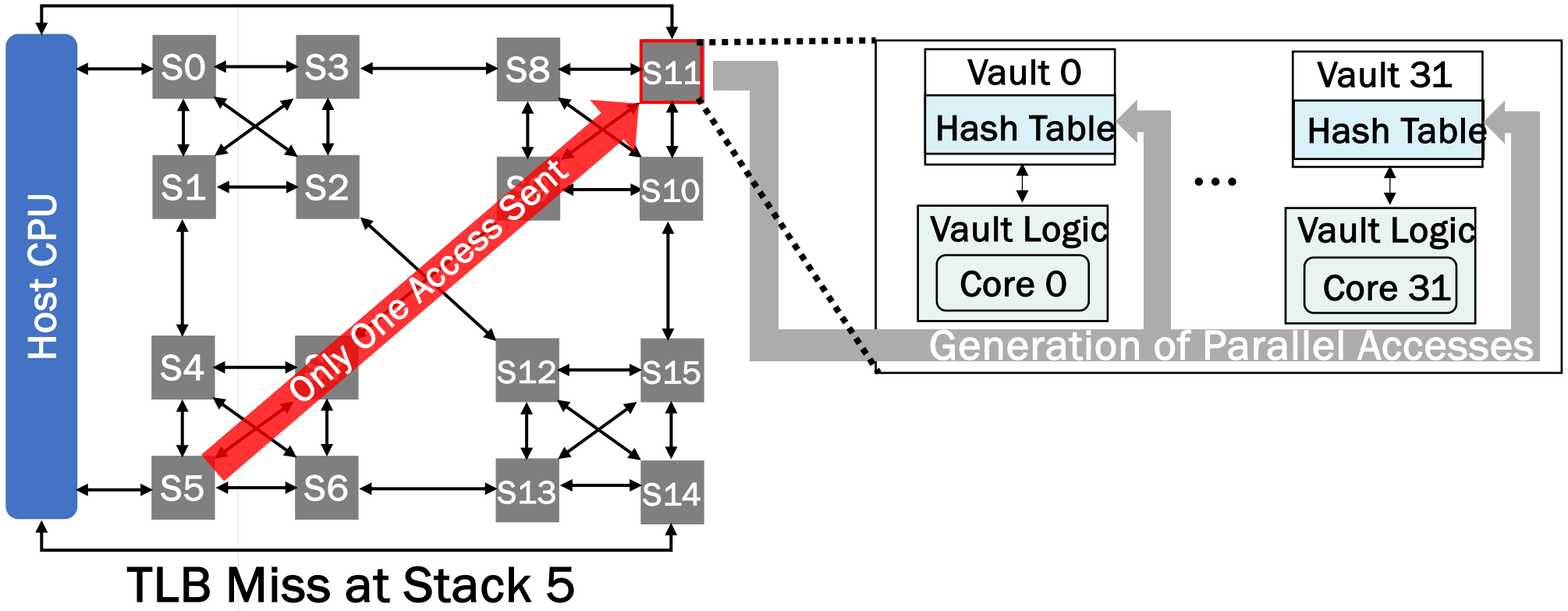
Key Idea: Network-contention-aware hash

- Parallel lookups generated within the stack avoid costly cross-stack page table walks



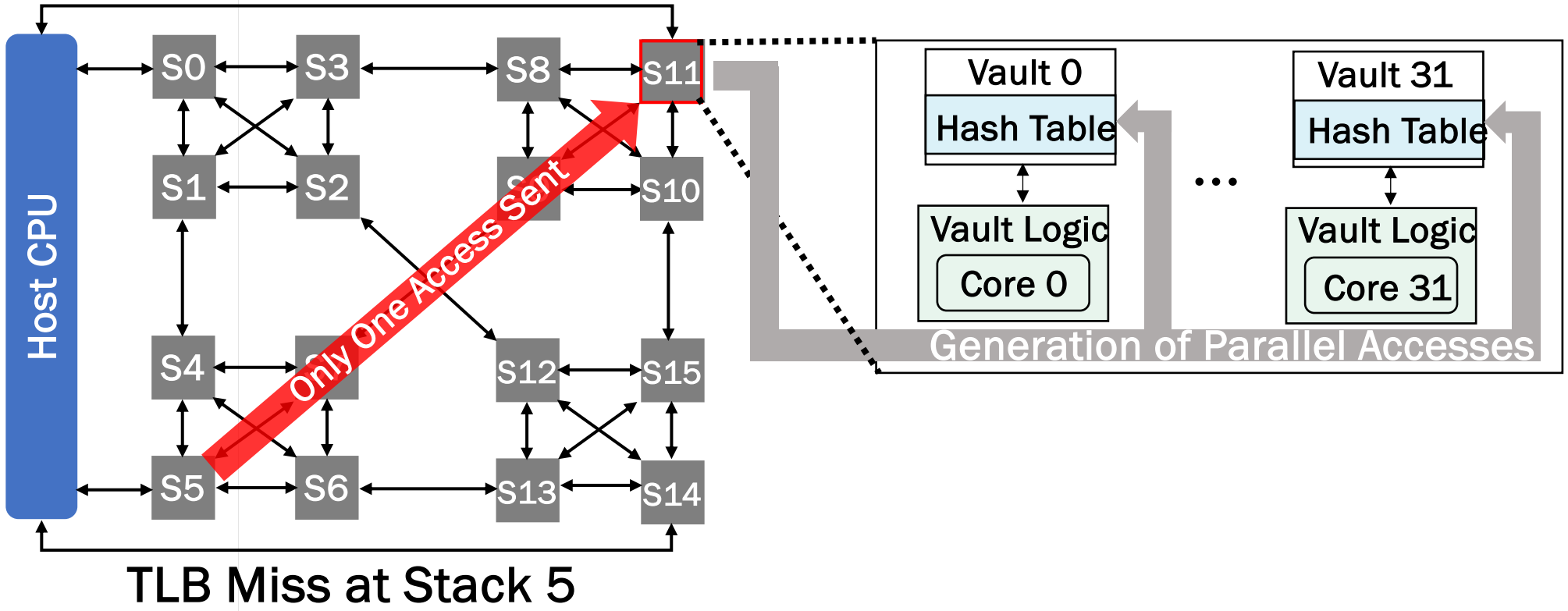
Key Idea: Network-contention-aware hash

- Parallel lookups generated within the stack avoid costly cross-stack page table walks



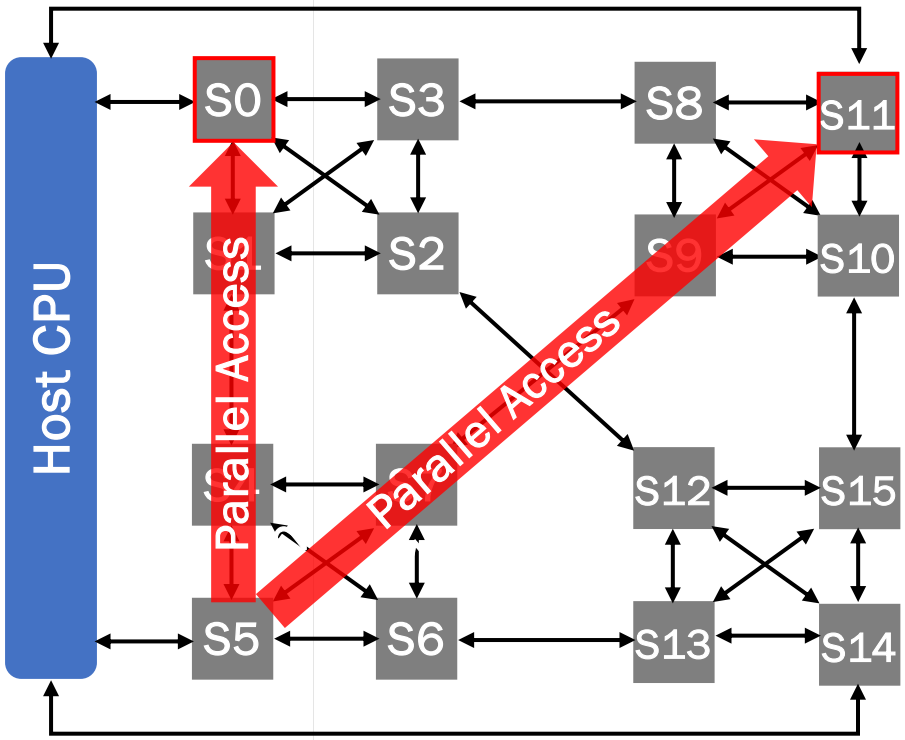
Key Idea: Network-contention-aware hash

- Parallel lookups generated within the stack avoid costly cross-stack page table walks

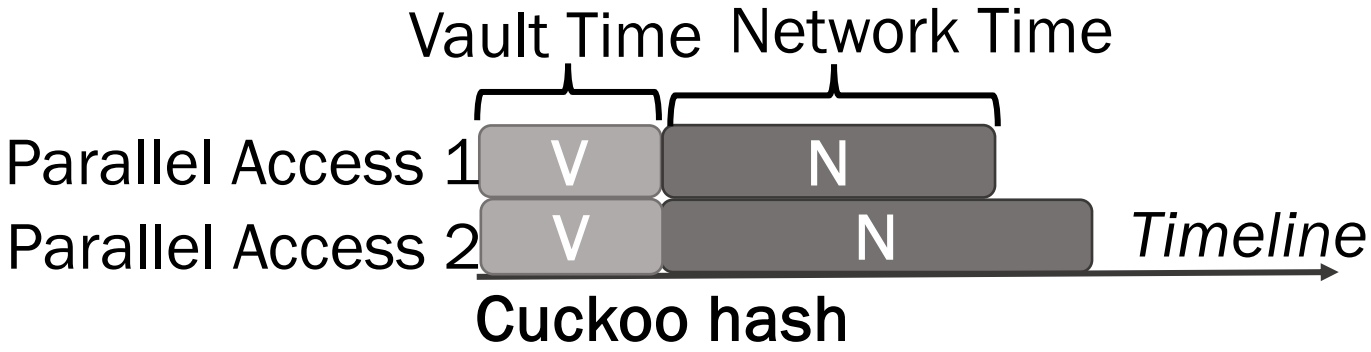


Network-contention-aware hash spawn parallel accesses within the same stack to avoid network traversal

Benefit and Opportunity

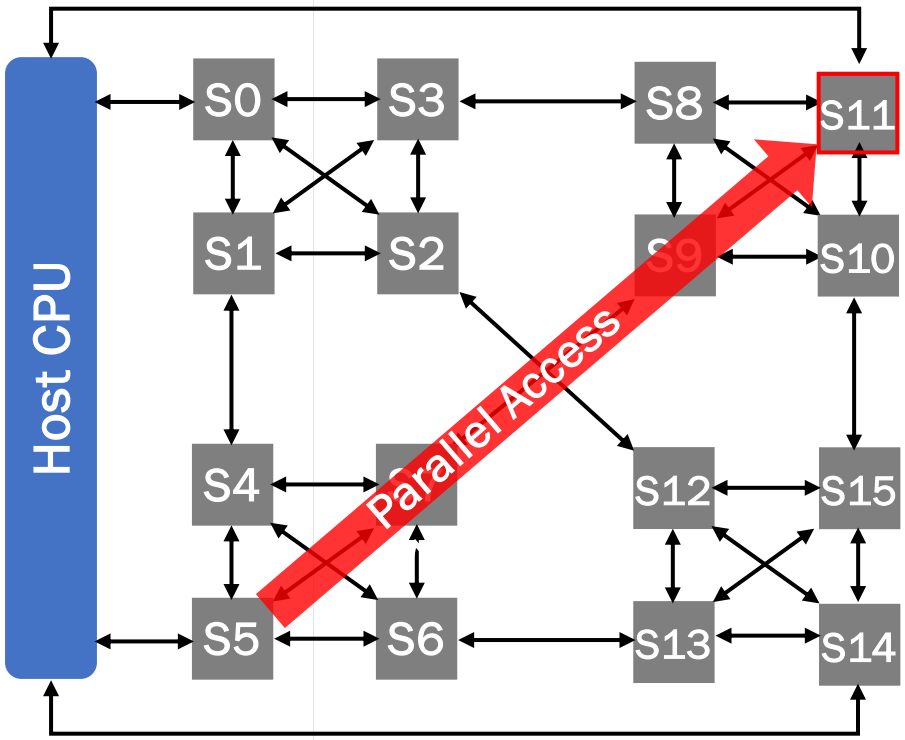


TLB Miss at Stack 5

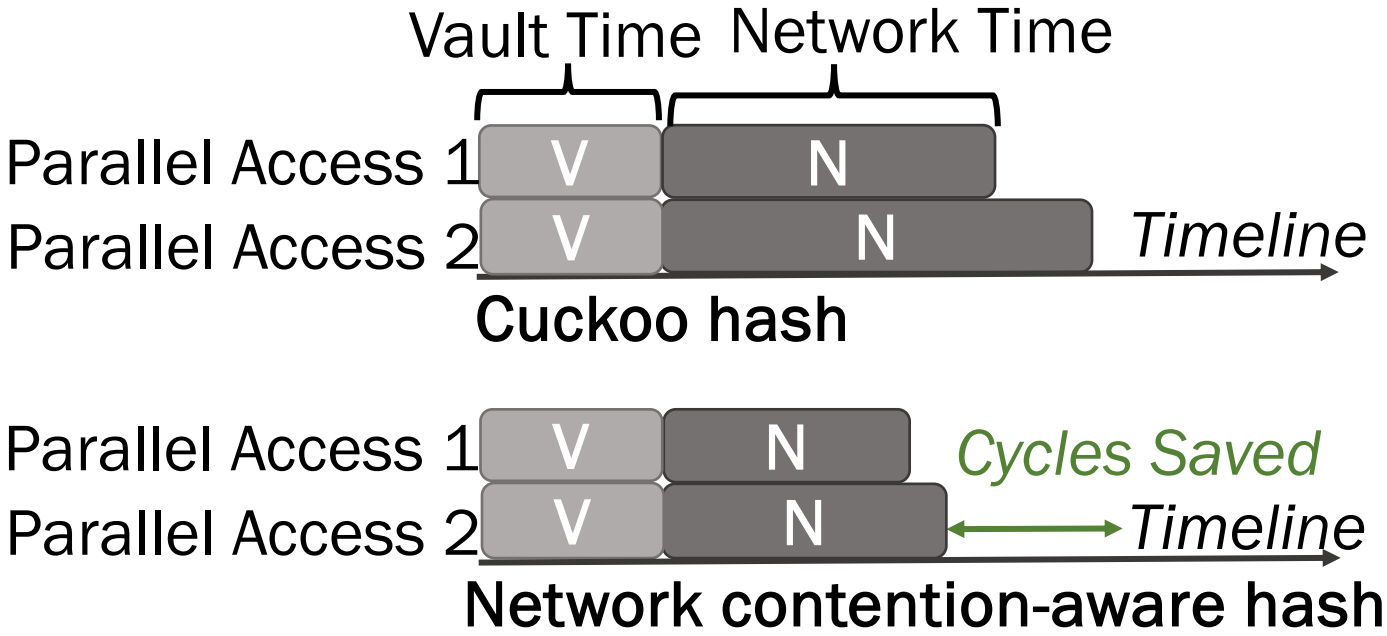


Cuckoo hash

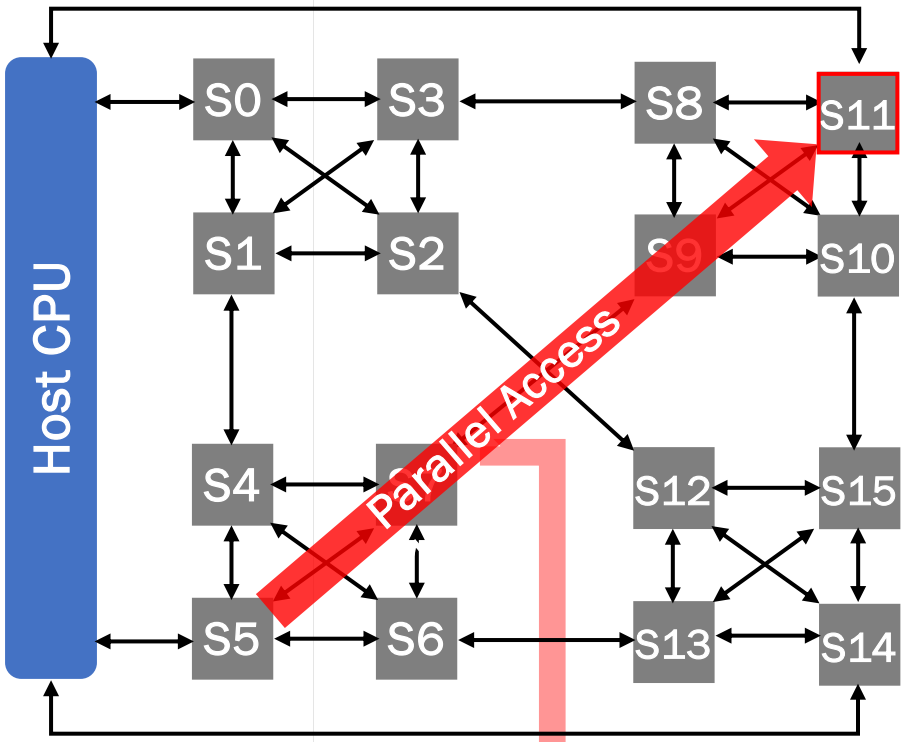
Benefit and Opportunity



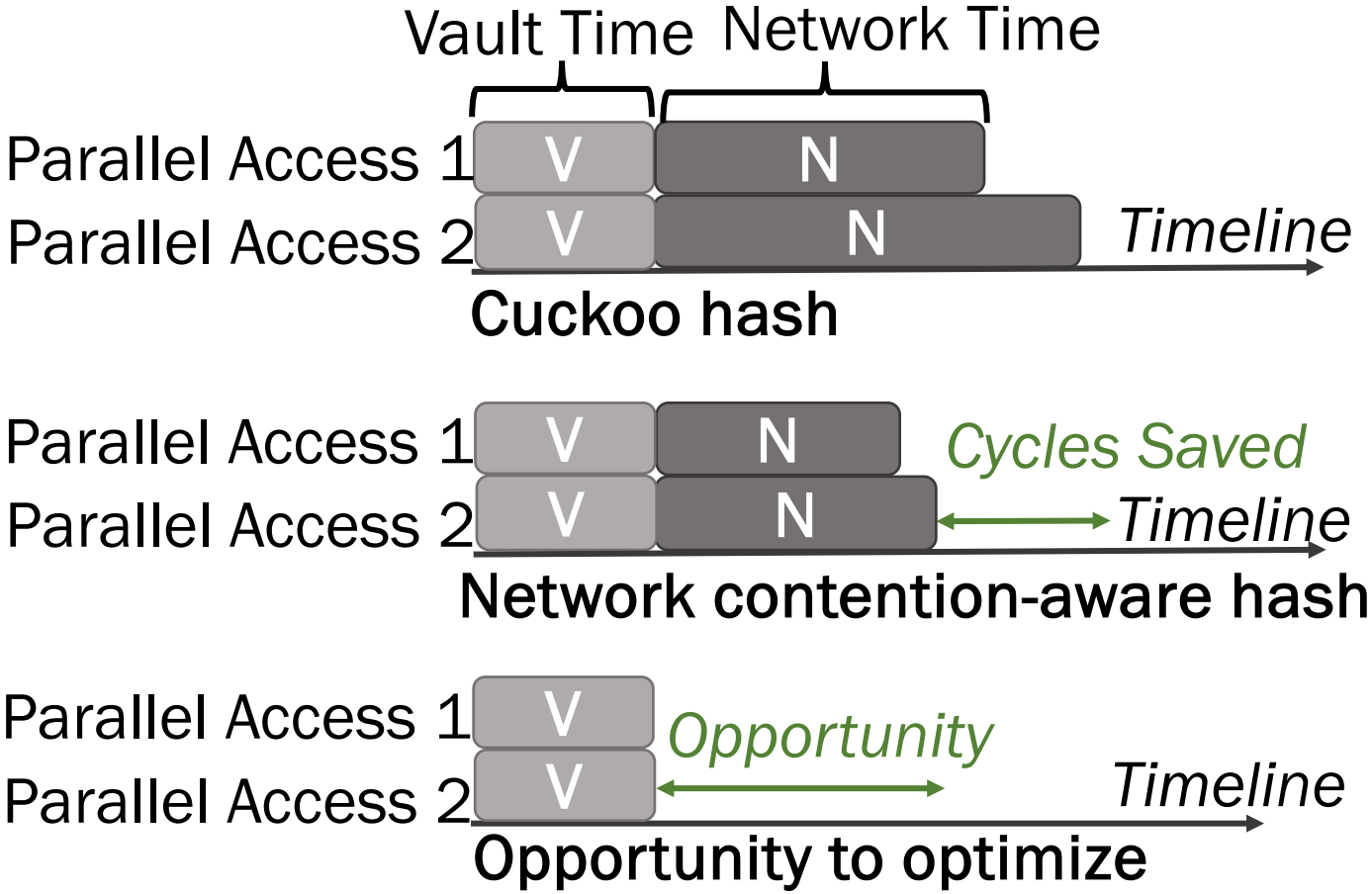
TLB Miss at Stack 5



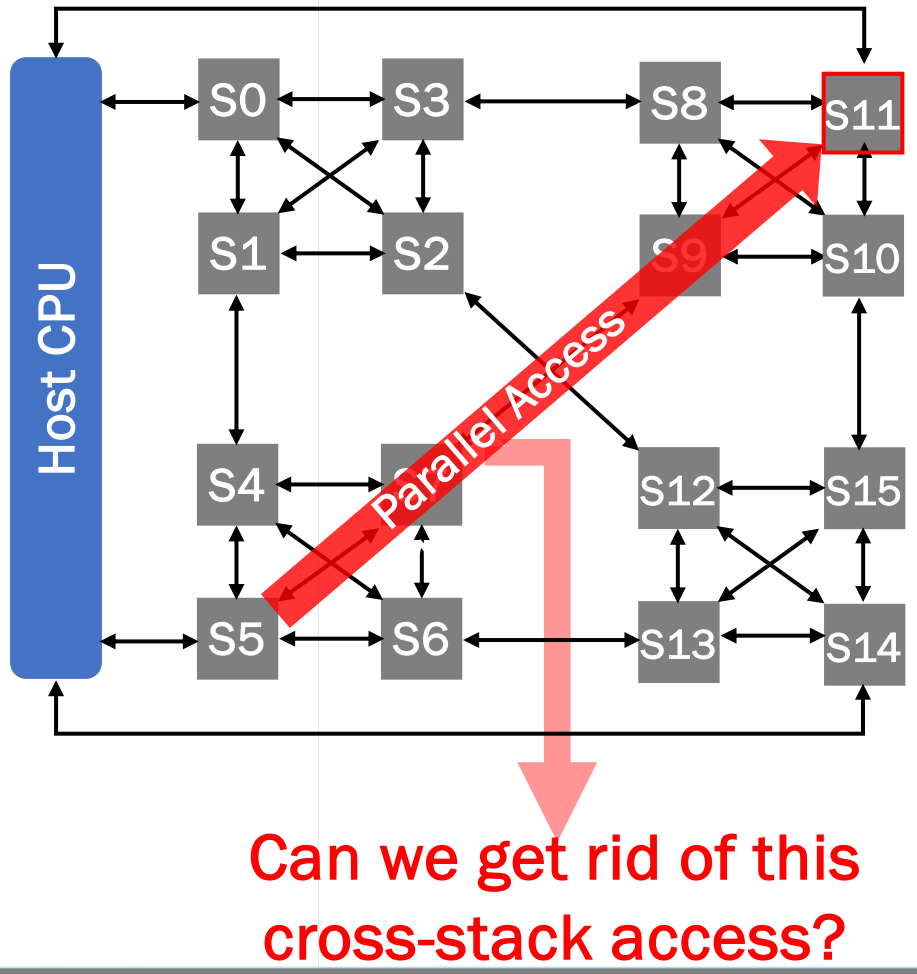
Benefit and Opportunity

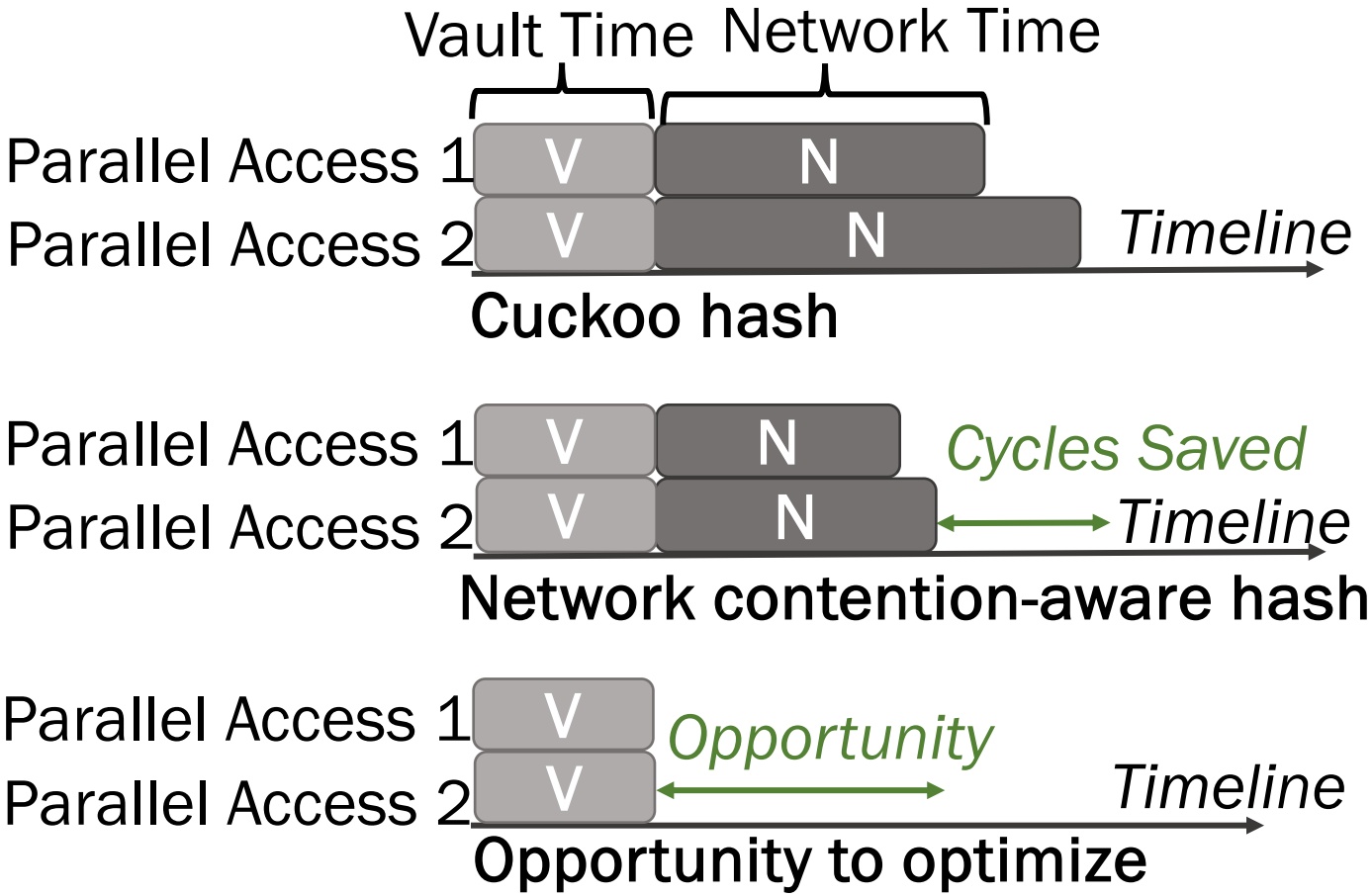


Can we get rid of this cross-stack access?



Benefit and Opportunity



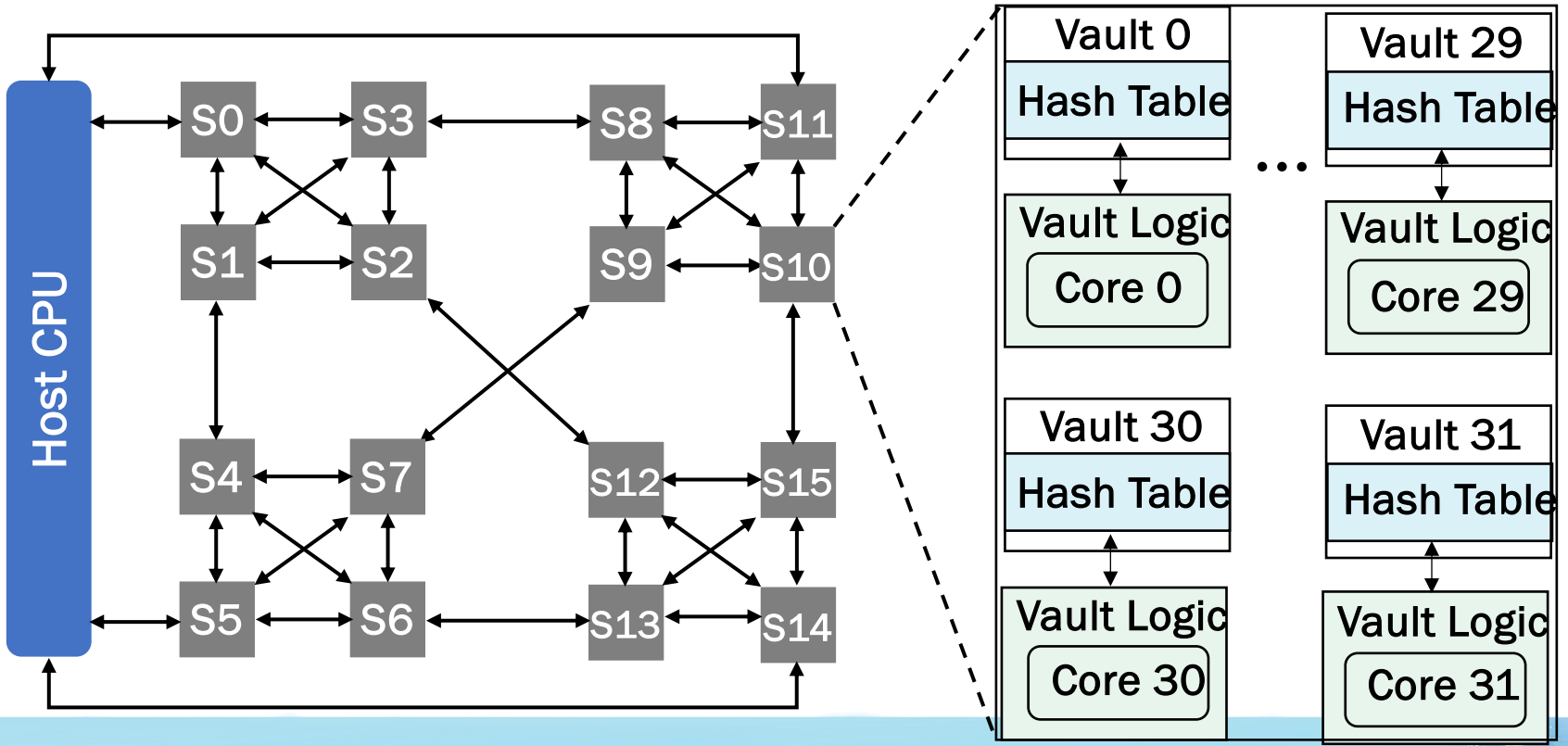


Is there any way to eliminate the overhead of cross-stack access?

Observation: PIM programs feature high parallelism

- PIM suitable programs manifest as independent threads that can be assigned to different cores and execute in parallel

```
#pragma omp parallel for
for(...)
{

        ...

}
```

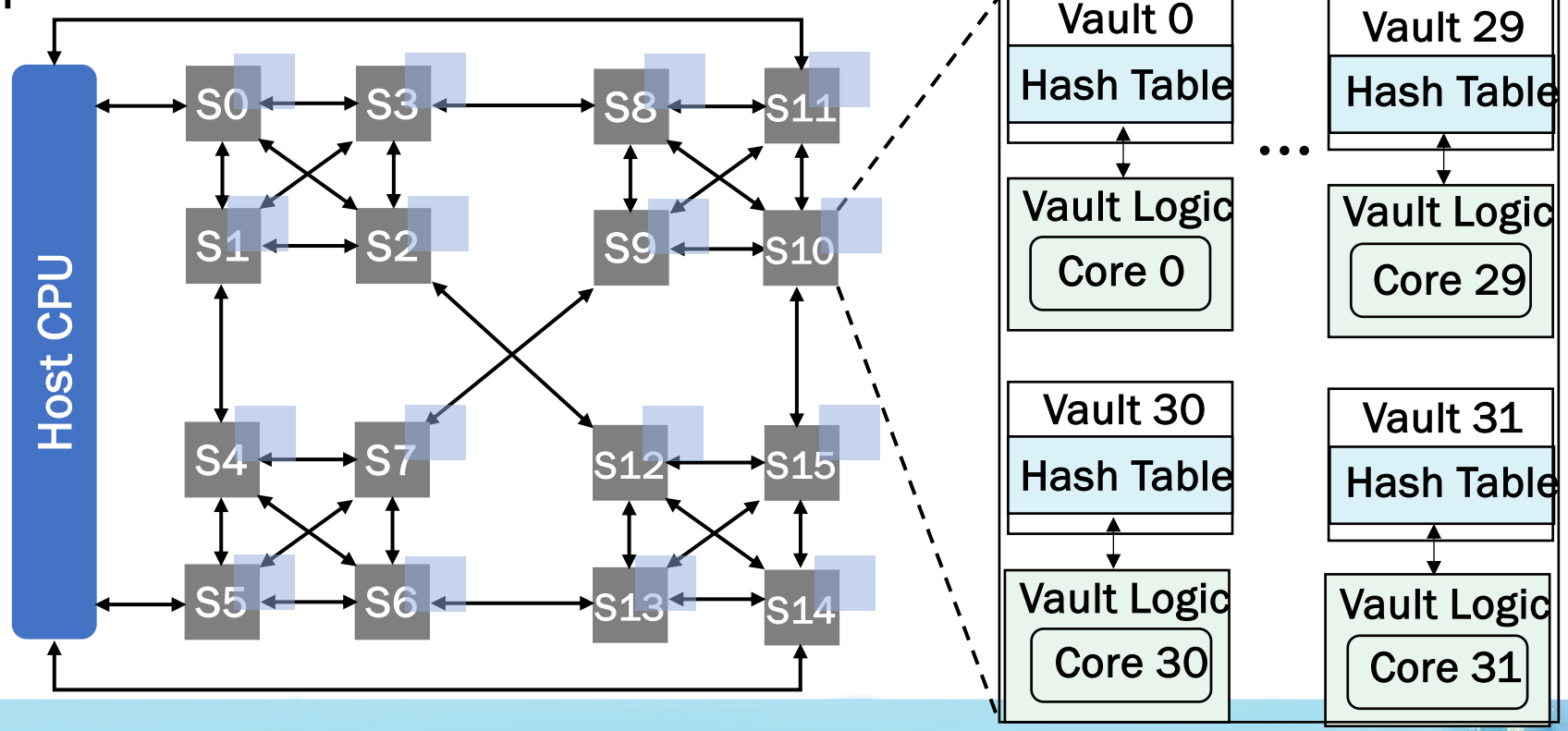


Observation: PIM programs feature high parallelism

- PIM suitable programs manifest as independent threads that can be assigned to different cores and execute in parallel

Iterations are evenly distributed across the PIM stacks for computation

```
#pragma omp parallel for
for(...)
{

    ...

}
```



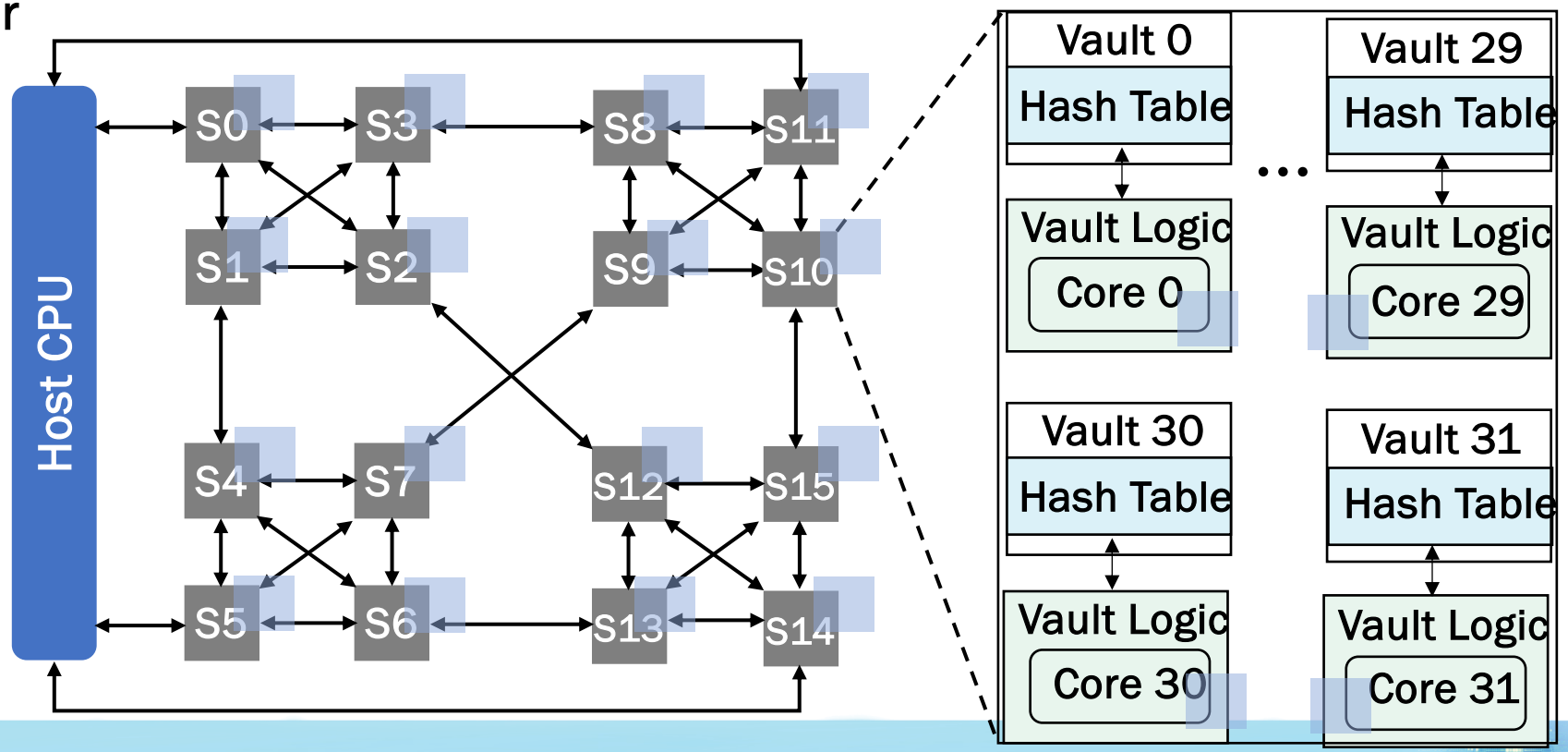
Observation: PIM programs feature high parallelism

- PIM suitable programs manifest as independent threads that can be assigned to different cores and execute in parallel

Iterations are evenly distributed across the PIM stacks for computation

```
#pragma omp parallel for
for(...)
{

    ...

}
```

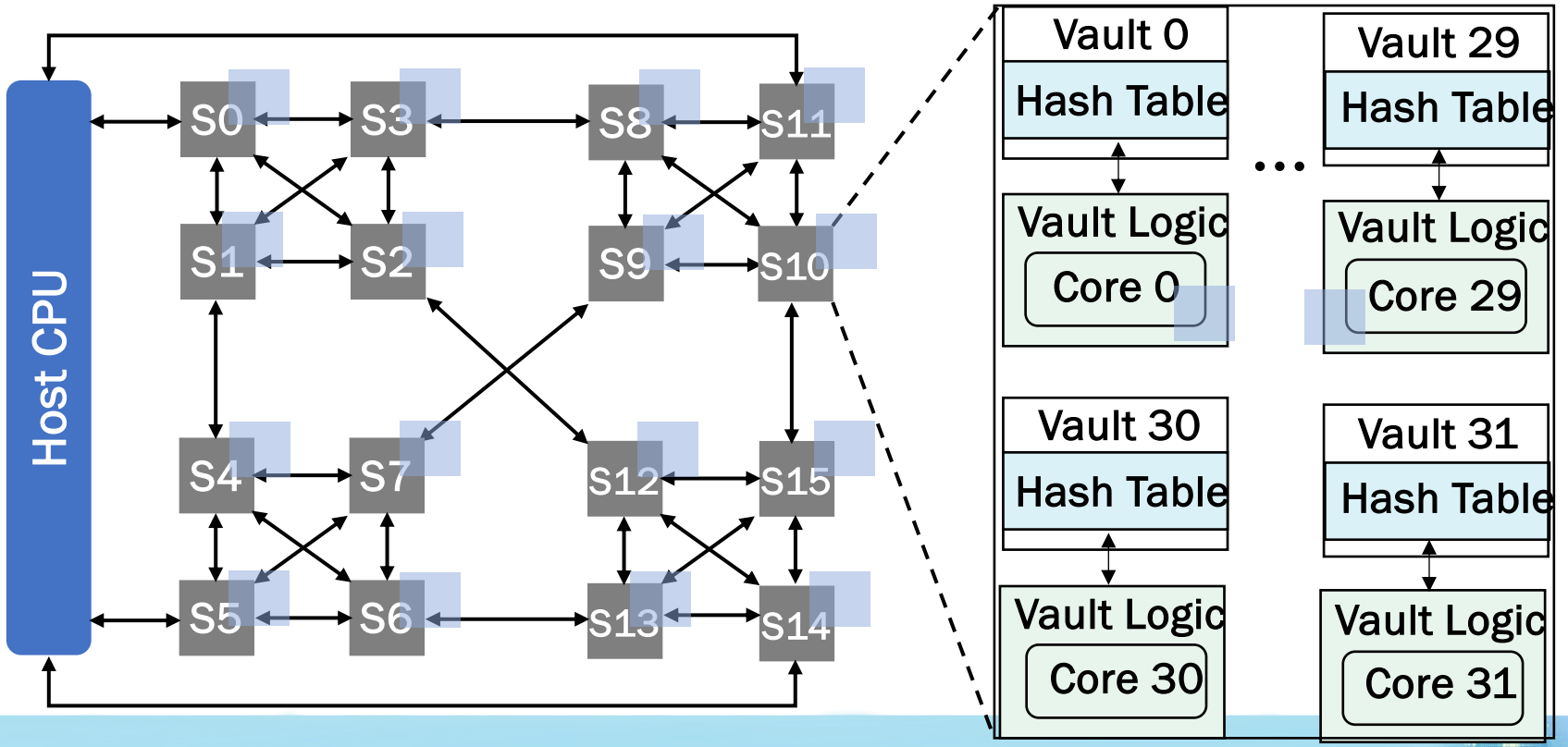
Iterations are further distributed across the vaults within a stack



Observation: Fewer cores cause minimal slowdown

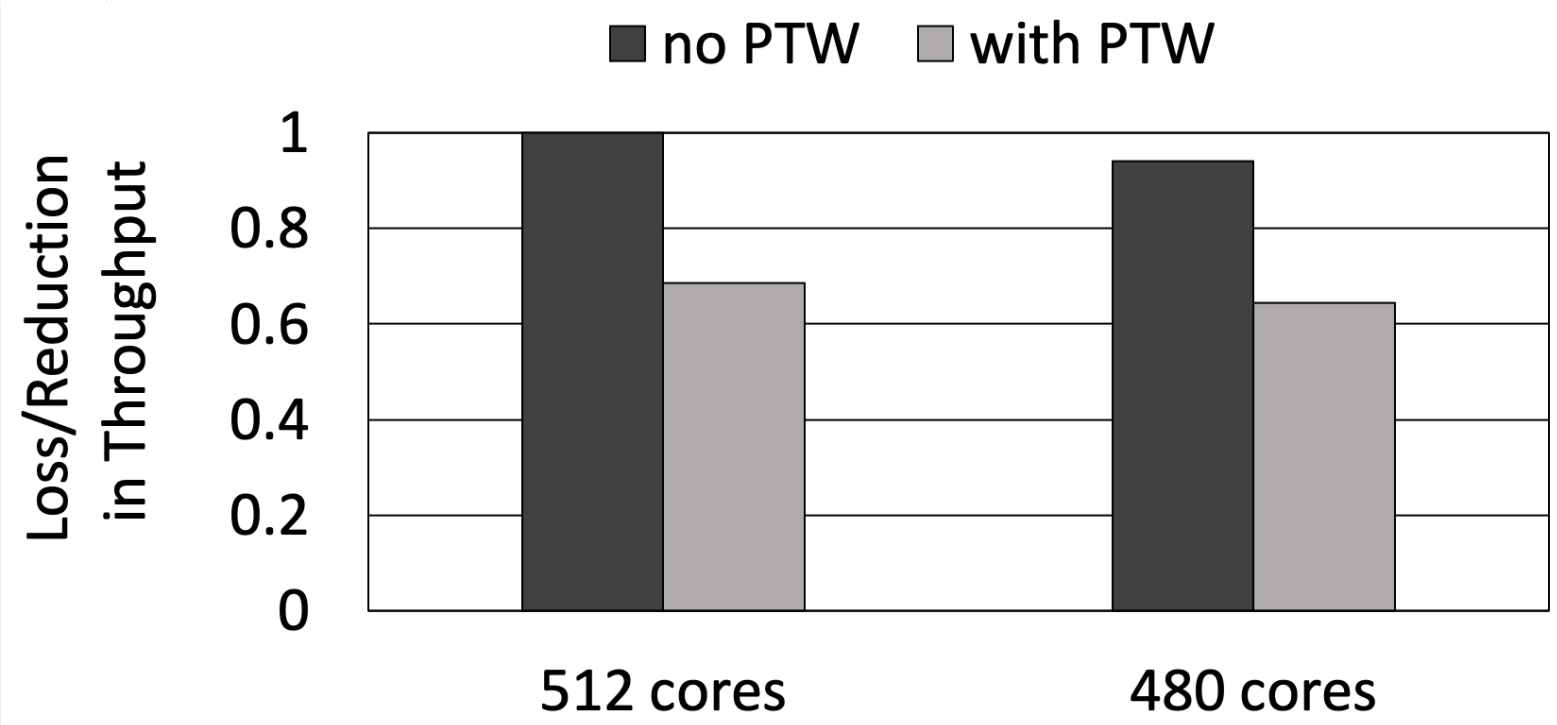
- Reduction in throughput has minimal impact on performance

```
#pragma omp parallel for
for(...)
{

      ...

}
```



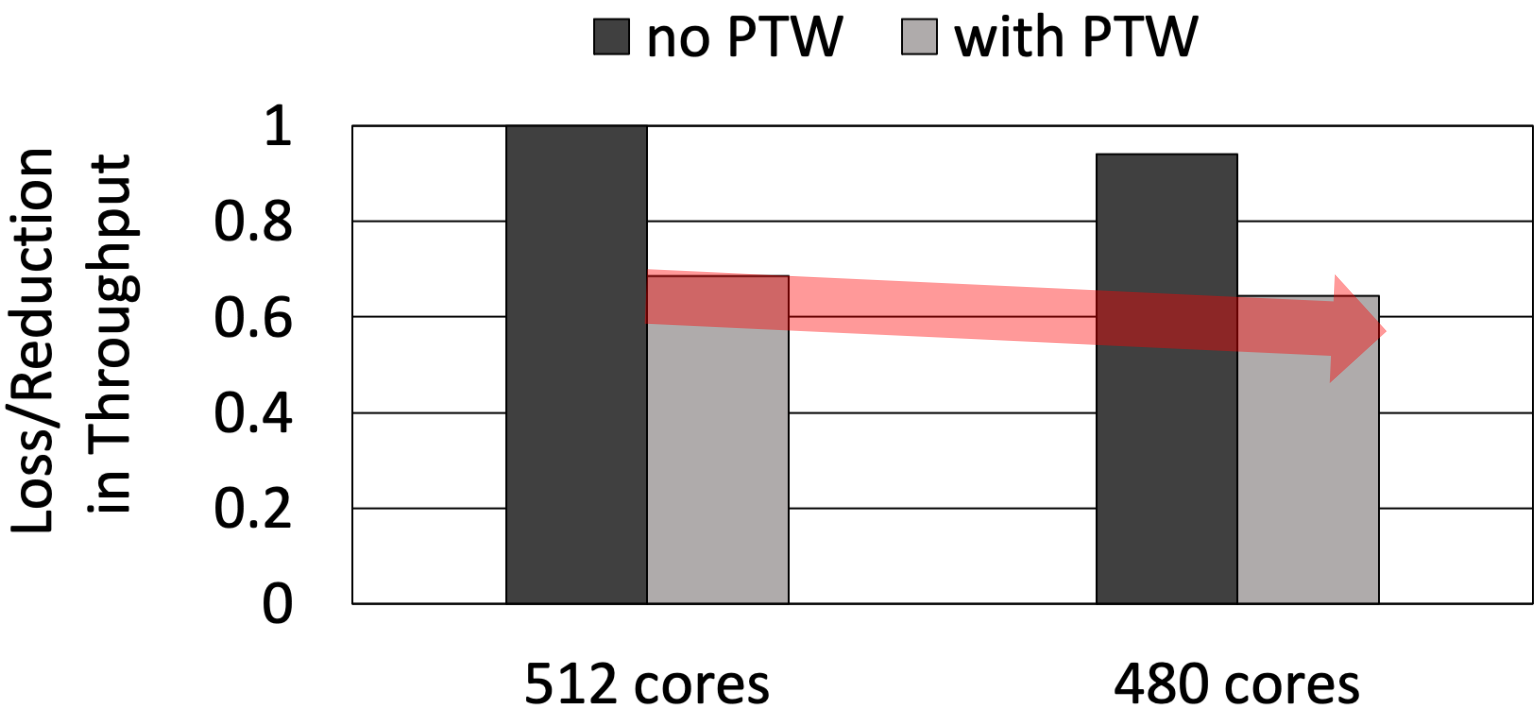
Page table walk overhead

- Observation: PIM suitable programs feature high parallelism and manifest as independent loops that can be assigned to different cores and be executed in parallel
- Observation: Reduction in throughput has minimal impact on performance



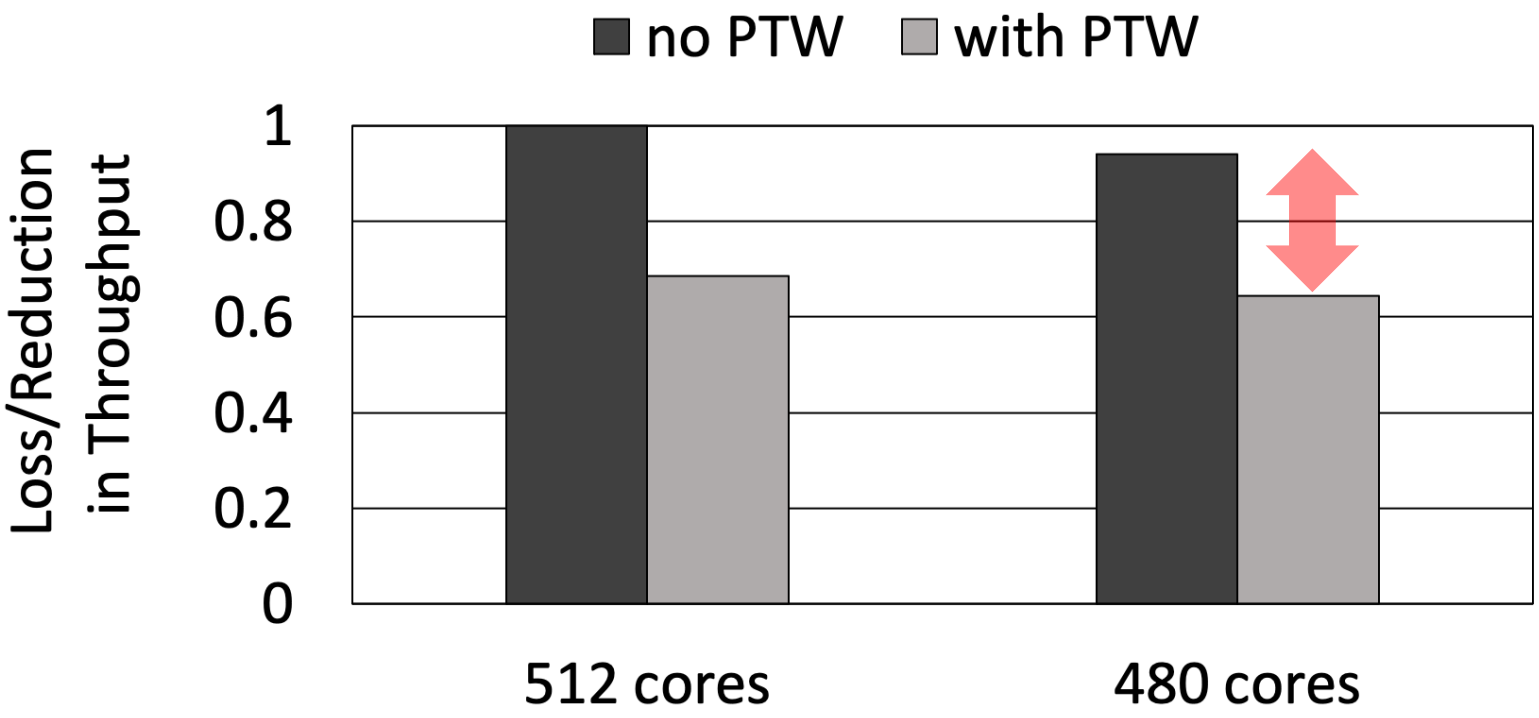
Page table walk overhead

- Observation: PIM suitable programs feature high parallelism and manifest as independent loops that can be assigned to different cores and be executed in parallel
- Observation: Reduction in throughput has minimal impact on performance



Page table walk overhead

- Observation: PIM suitable programs feature high parallelism and manifest as independent loops that can be assigned to different cores and be executed in parallel
- Observation: Reduction in throughput has minimal impact on performance

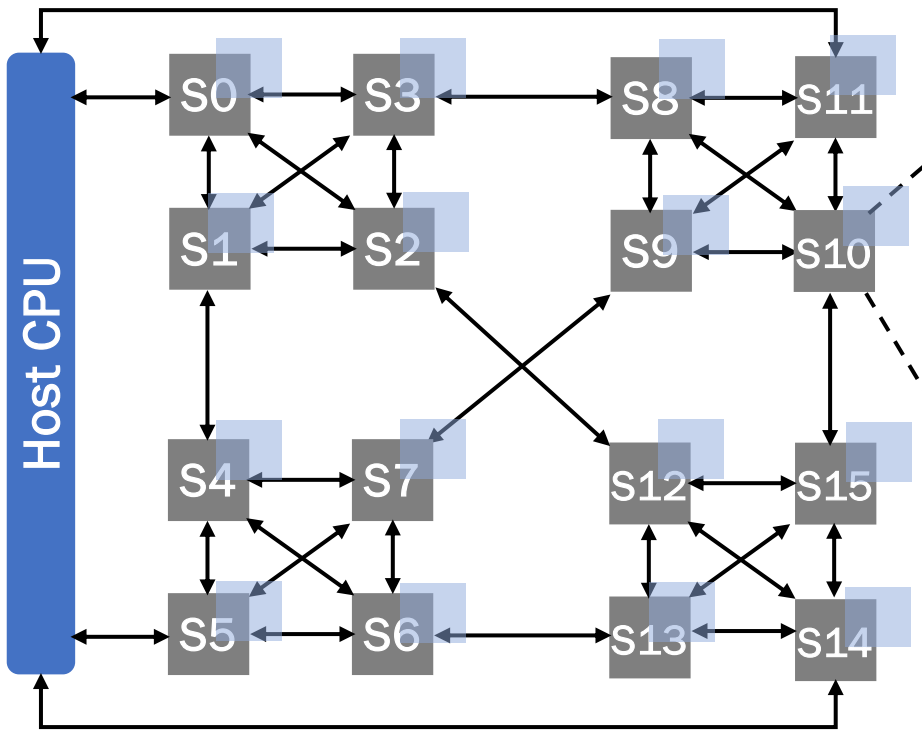


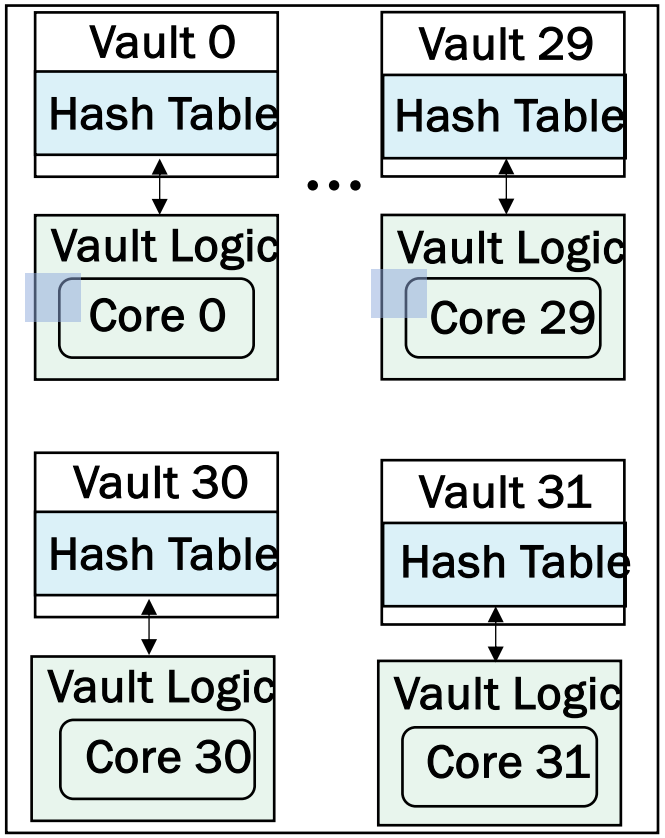
Performance is bottlenecked by the page table walk overhead

Key Idea: Pre-translation of page table walks

Utilize some throughput to pre-translate page table walks

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x

}
for(...)
{
  Code for pre-translation cores
    x=array1[i]
    sum+=x
}
```

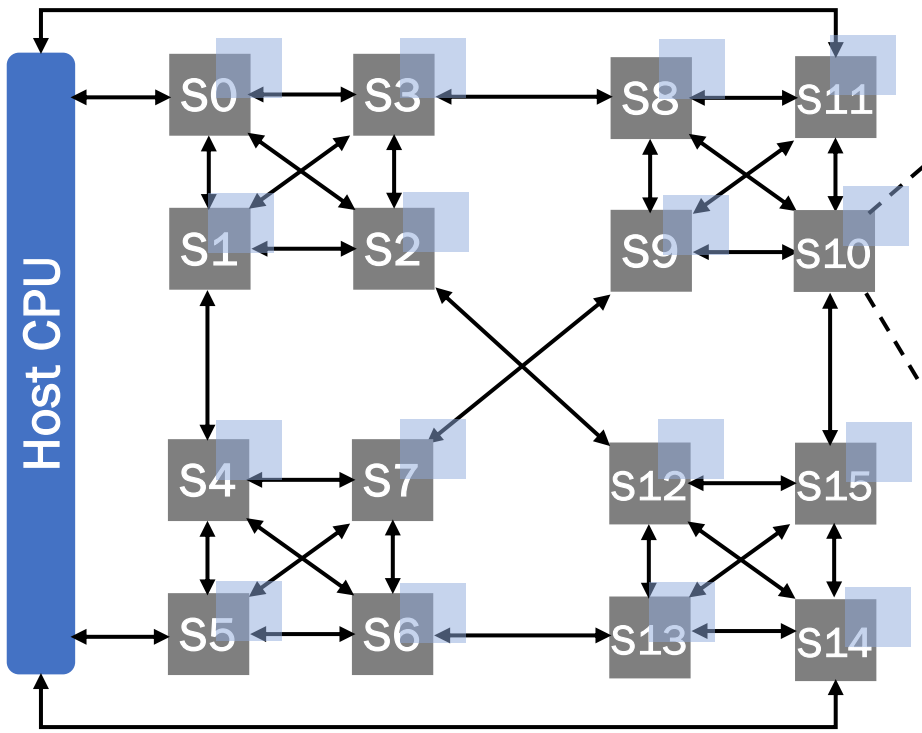


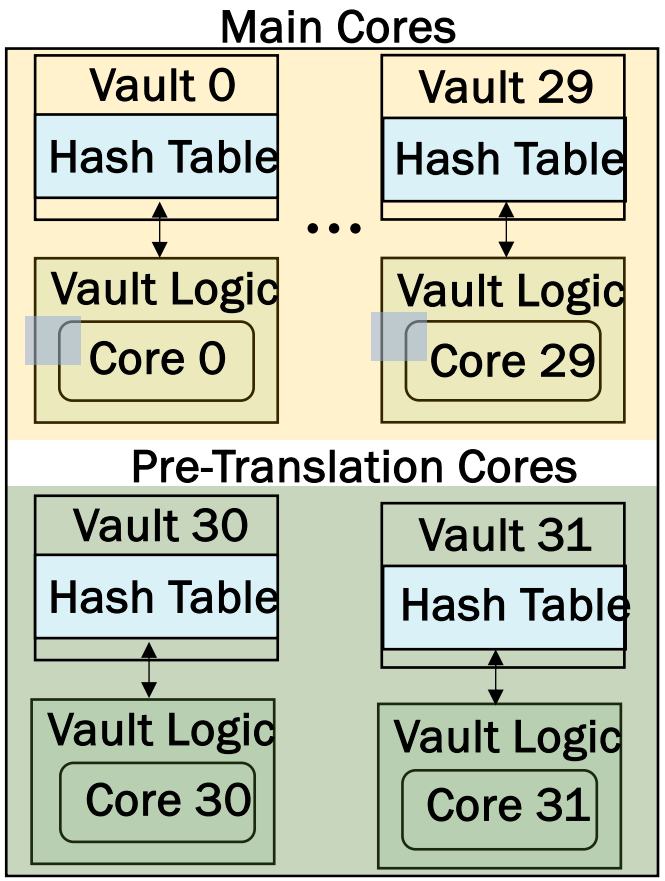


Key Idea: Pre-translation of page table walks

Utilize some throughput to pre-translate page table walks

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...)
{
  Code for pre-translation cores
    x=array1[i]
    sum+=x
}
```

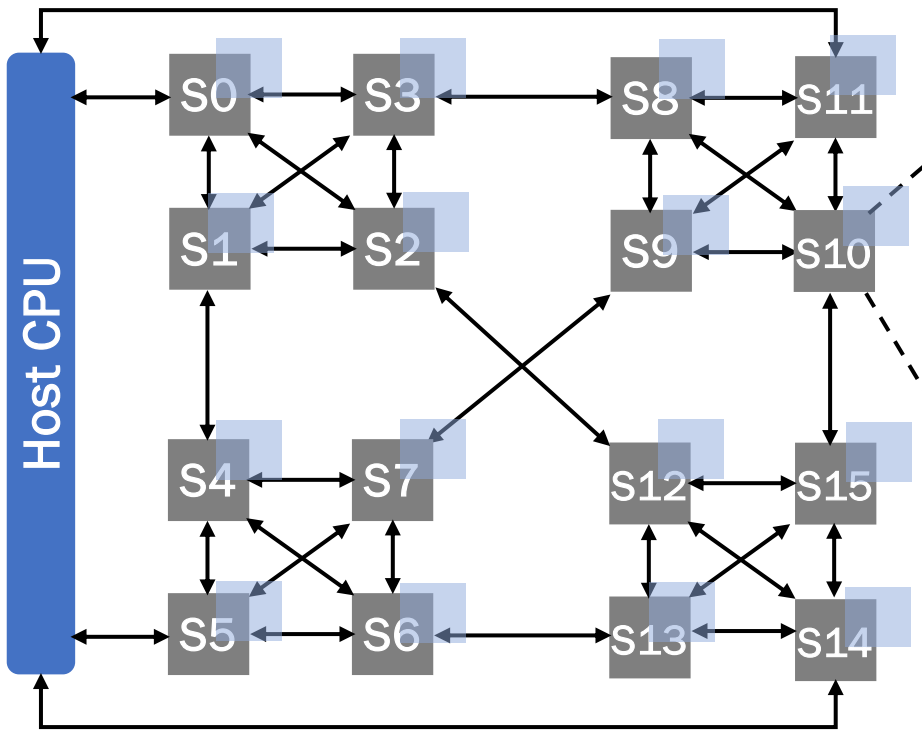


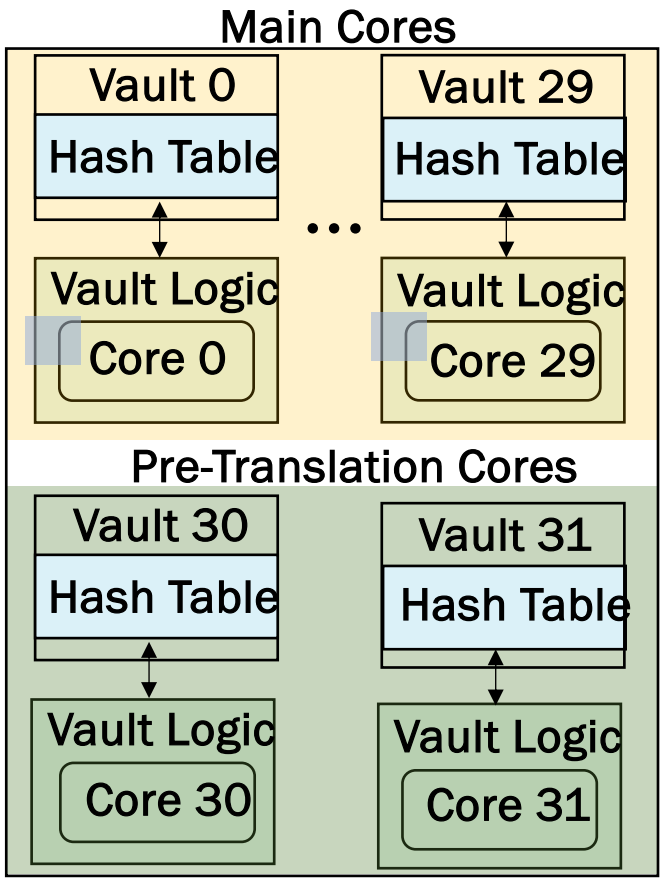


Key Idea: Pre-translation of page table walks

Utilize some throughput to pre-translate page table walks

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...)
{
  Code for pre-translation cores
    x=array1[i]
    sum+=x
}
```



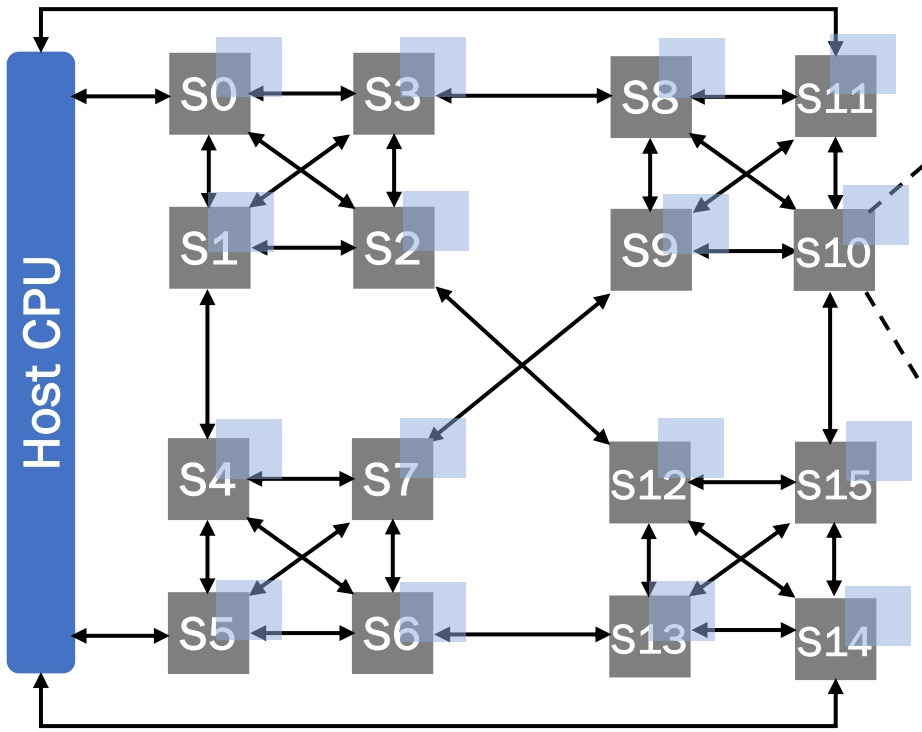


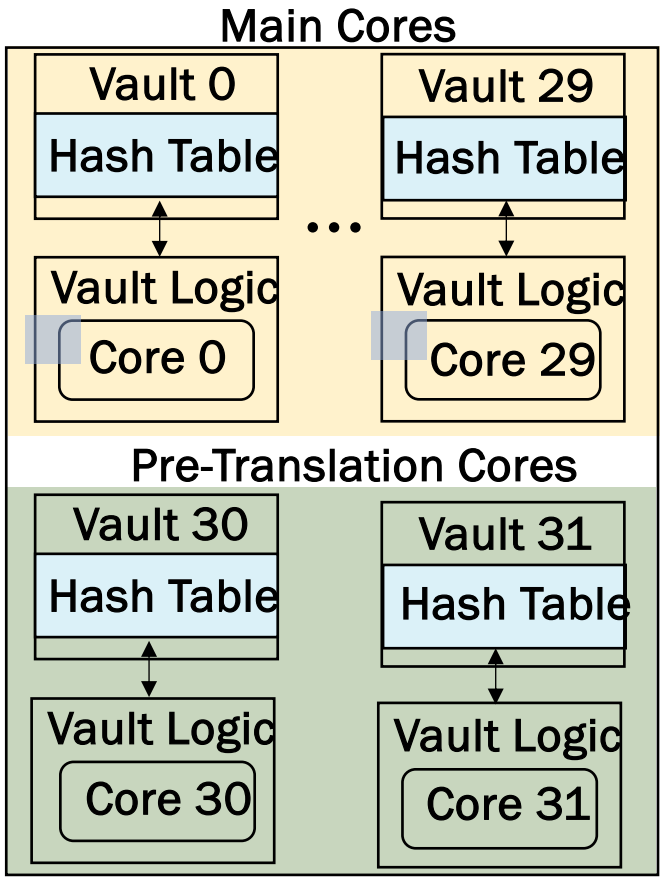
Pre-translation cores run ahead iterations for the main cores

Key Idea: Pre-translation of page table walks

Filter the pre-translation code to remove non-memory accesses and accesses to small data structures that are less likely to trigger PTW

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...)
{
  Code for pre-translation cores
    x=array1[i]
    sum+=x
}
```

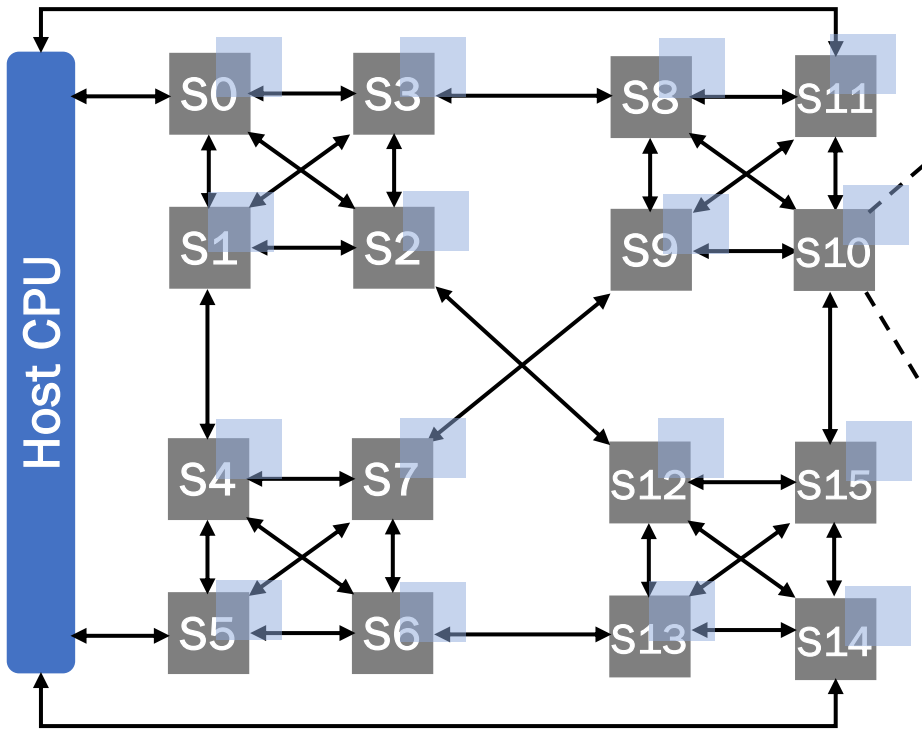


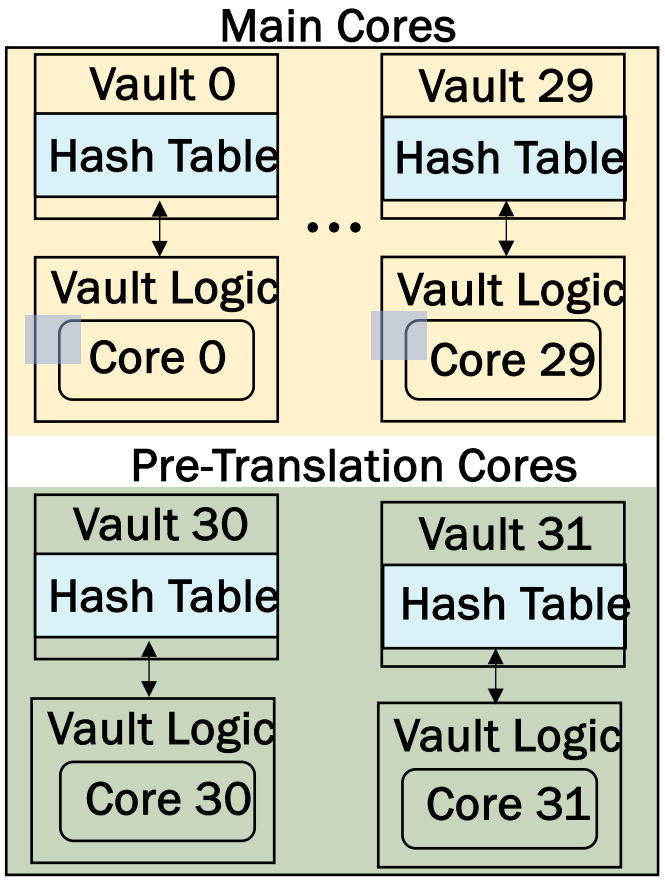


Key Idea: Pre-translation of page table walks

Filter the pre-translation code to remove non-memory accesses and accesses to small data structures that are less likely to trigger PTW

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...)
{
  Code for pre-translation cores
    x=array1[i]
    [redacted]
}
```

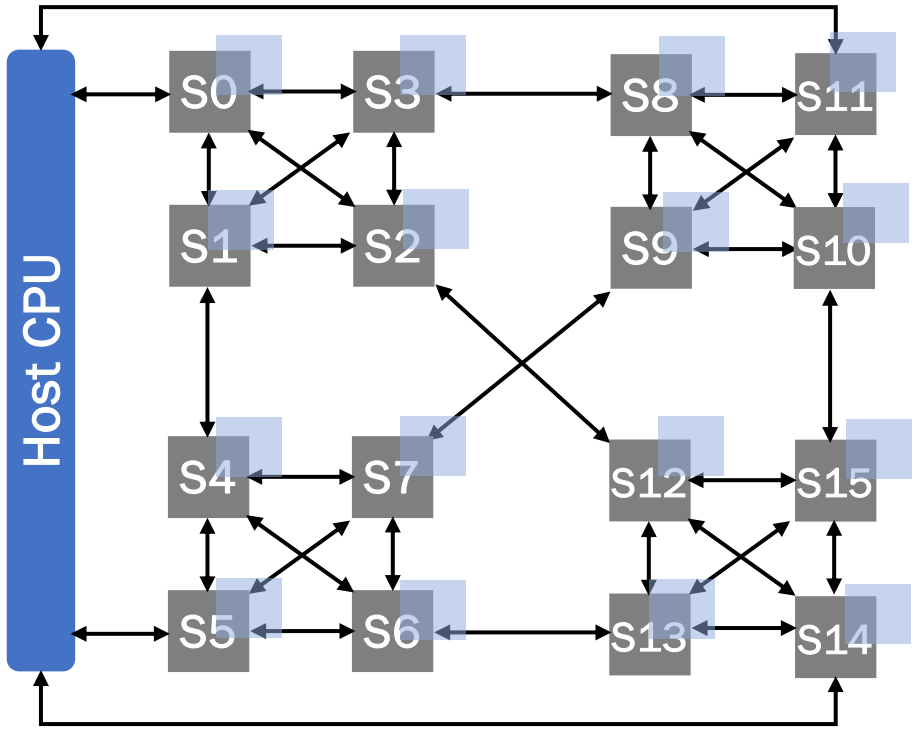


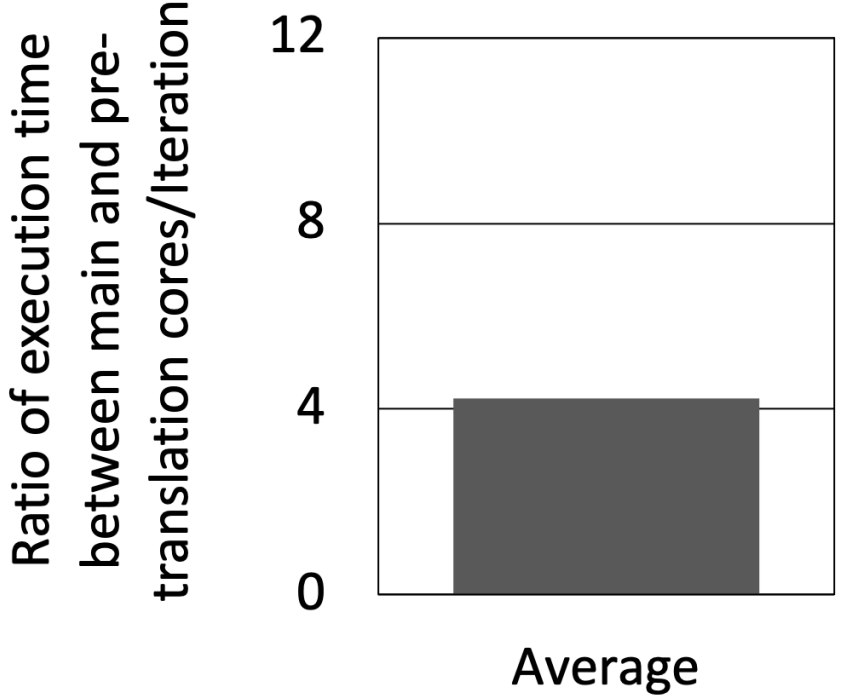


Key Idea: Pre-translation of page table walks

Filter the pre-translation code to remove non-memory accesses and accesses to small data structures that are less likely to trigger PTW

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...)
{
  Code for pre-translation cores
    x=array1[i]
}
```

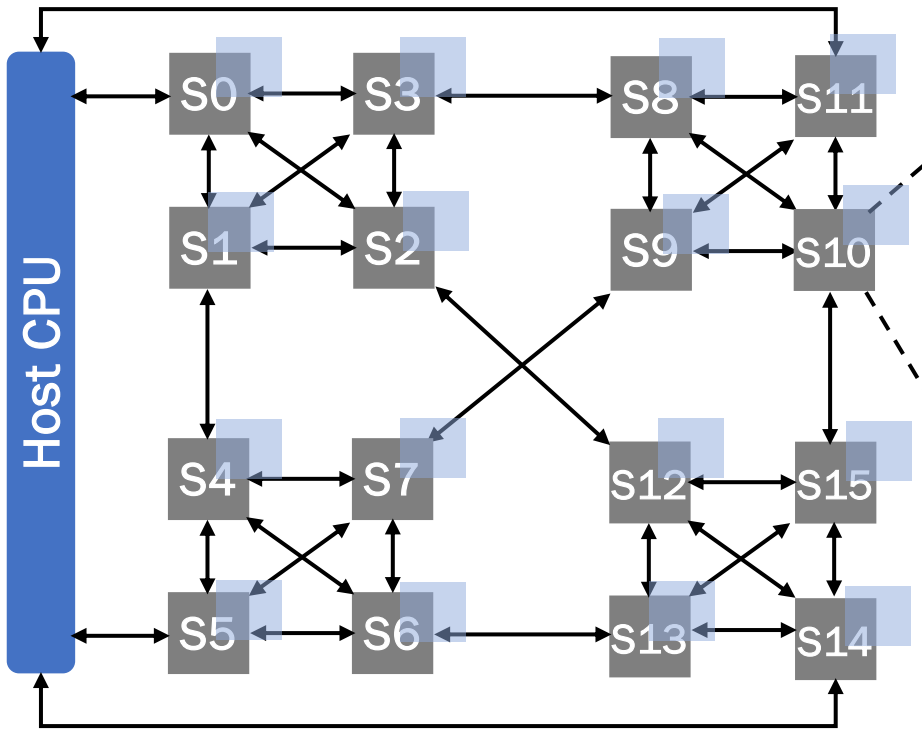


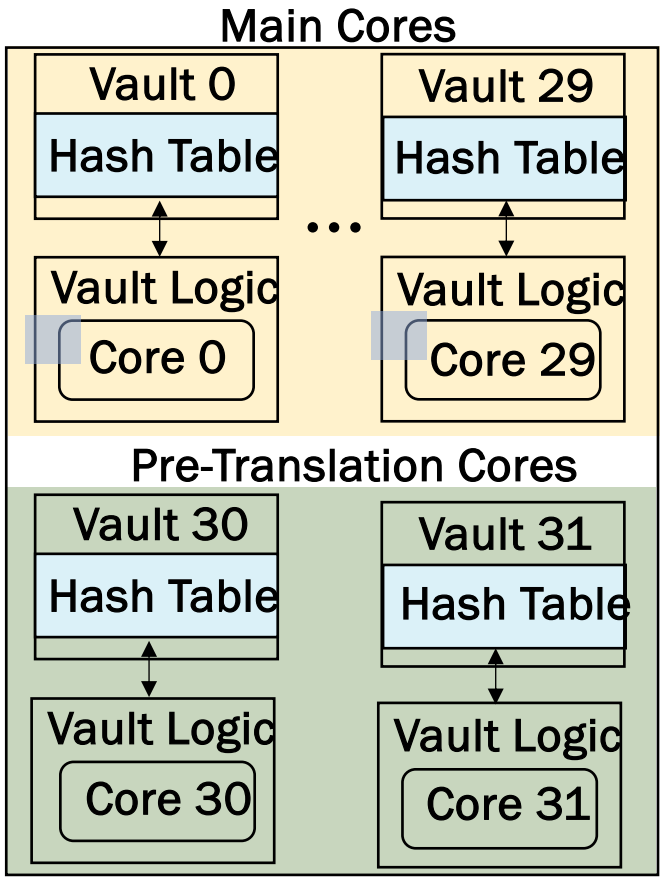


Pre-translation cores run faster than the main cores and are able to help a larger fraction of the main cores

Key Idea: Pre-translation of page table walks

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}

for(...)
{
  Code for pre-translation cores
    x=array1[i]

}
```



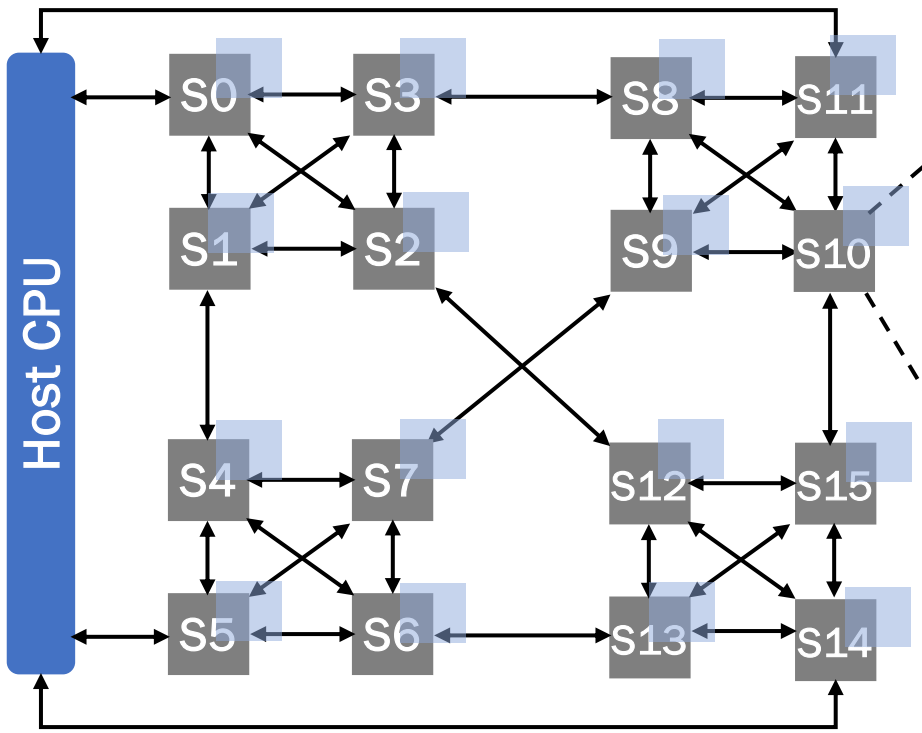


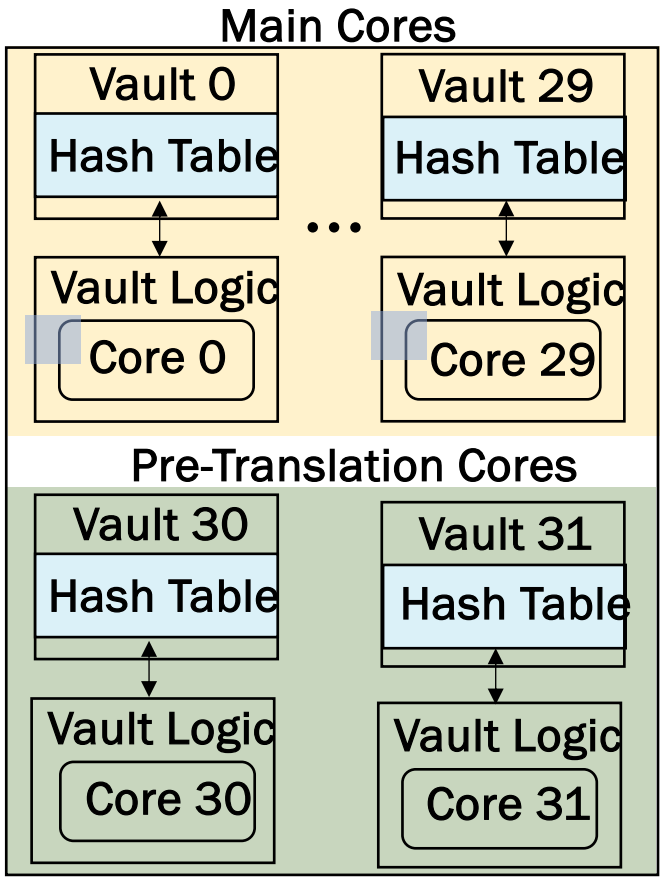
Key Idea: Pre-translation of page table walks

Step 1

Timeline

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}

for(...)
{
  Code for pre-translation cores
    x=array1[i]

}
```



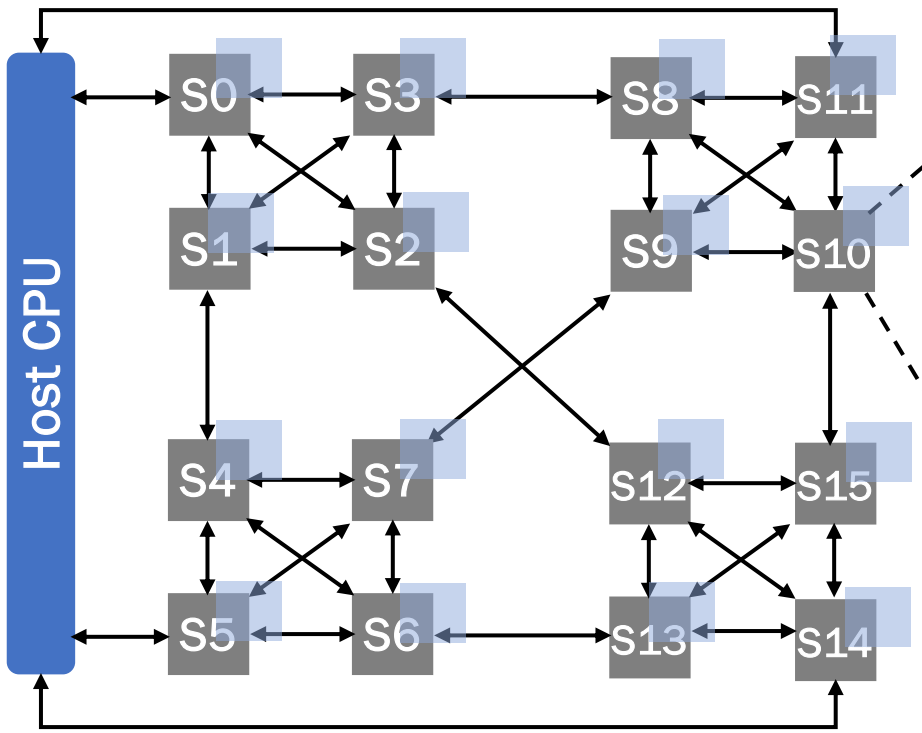


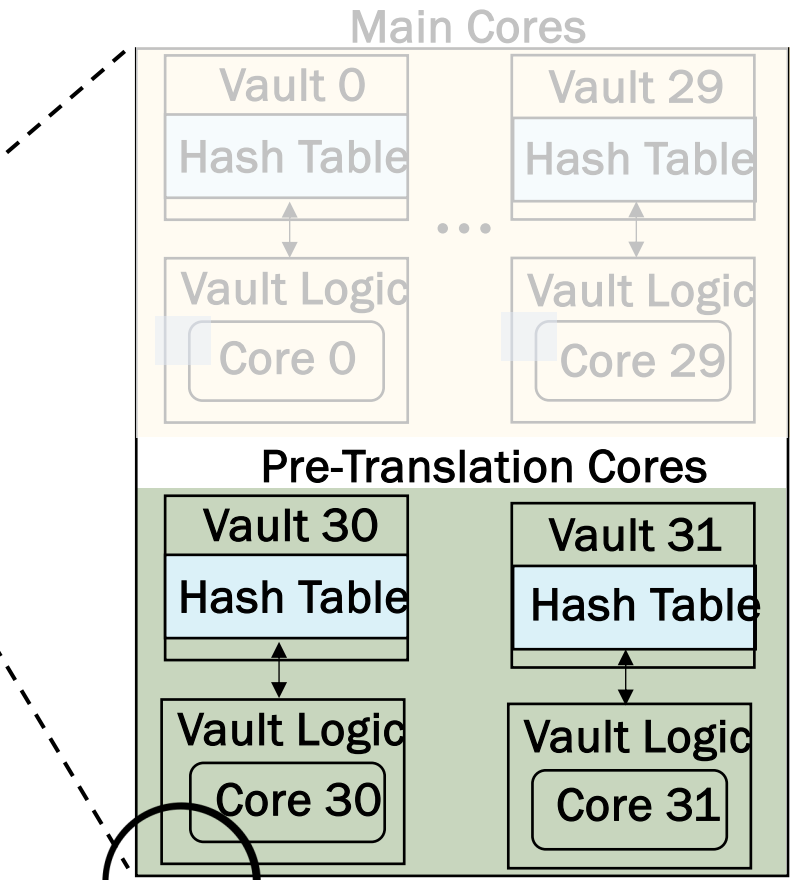
Key Idea: Pre-translation of page table walks

Step 1

Timeline

```
#pragma omp parallel for
for(...) // ITERATION u =0
{
    Code for main cores
    x=array1[i]
    sum+=x
}

for(...) // ITERATION u=10
{
  Code for pre-translation cores
    x=array1[i]

}
```



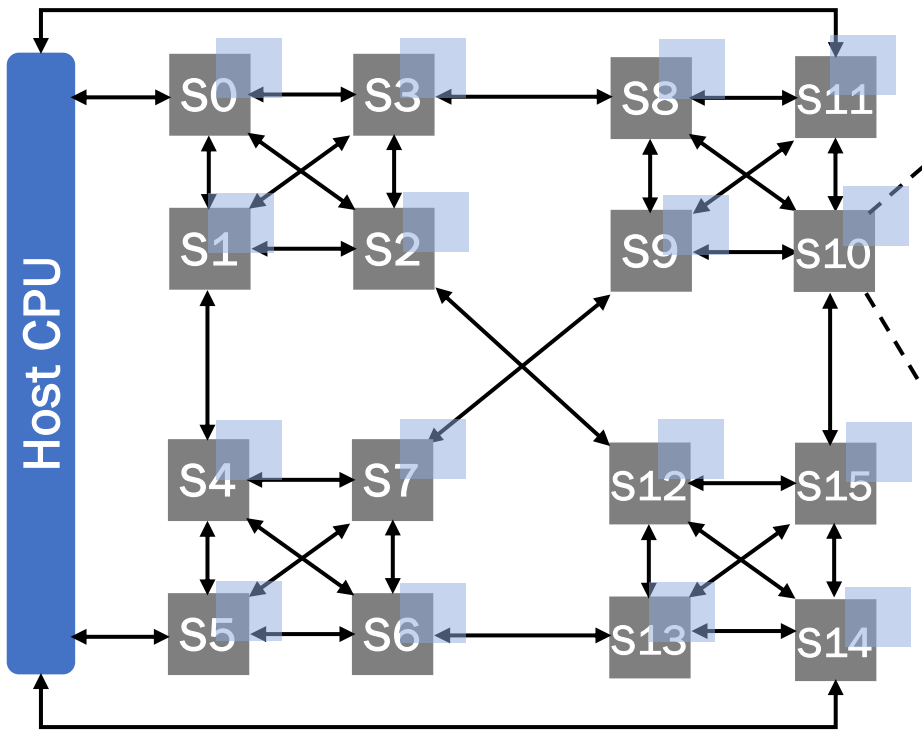


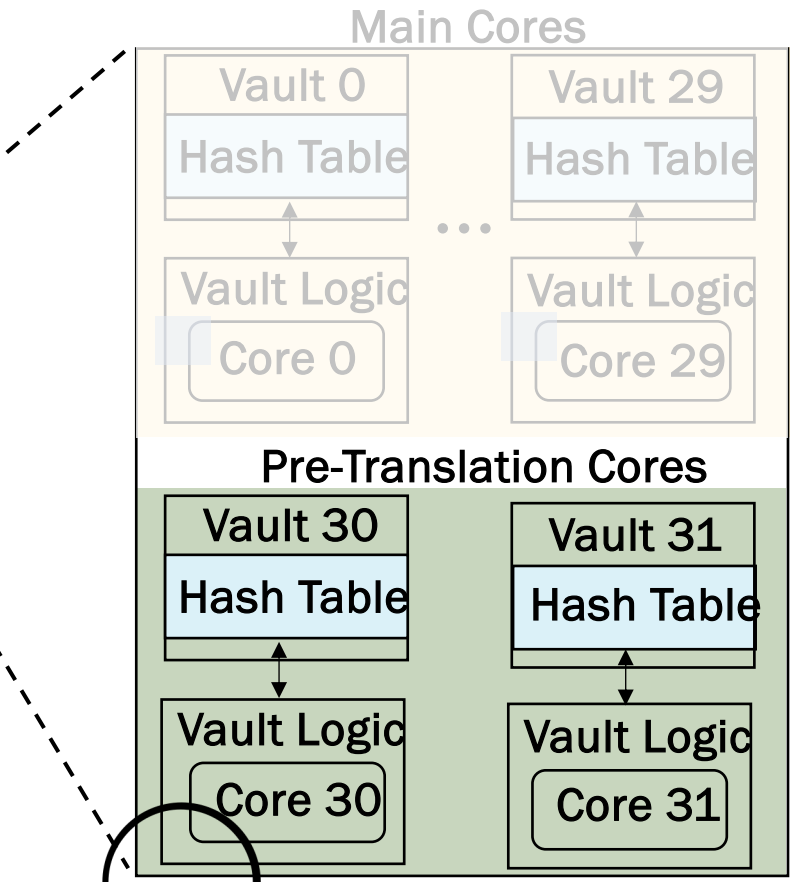
Key Idea: Pre-translation of page table walks

Step 1

Timeline

```
#pragma omp parallel for
for(...) // ITERATION u =0
{
    Code for main cores
    x=array1[i]
    sum+=x
}

for(...) // ITERATION u=10
{
  Code for pre-translation cores
    x=array1[i]   TLB Miss Generated
}
```



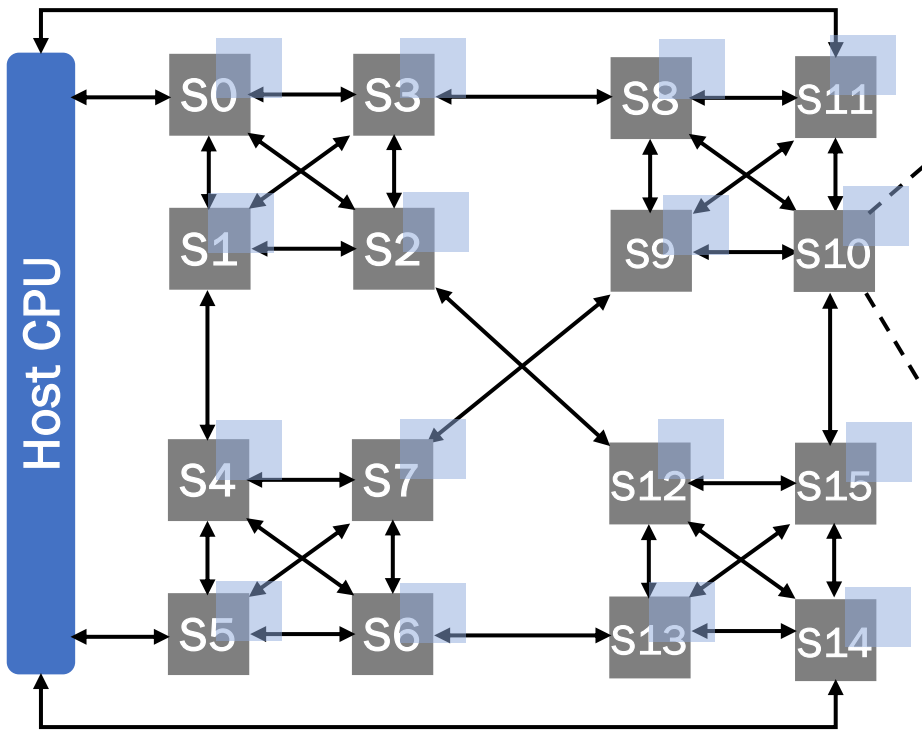


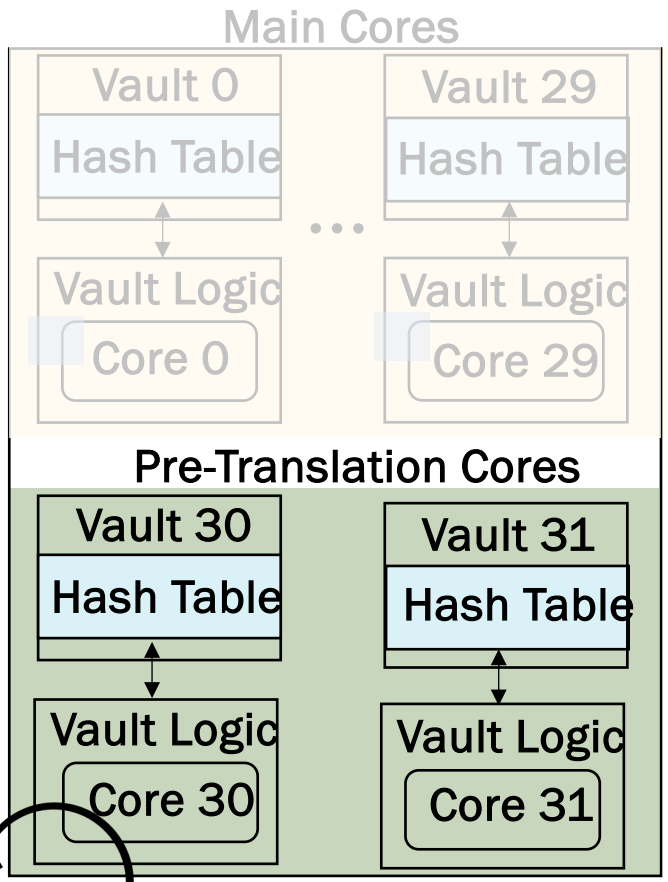
Key Idea: Pre-translation of page table walks

Step 1

Timeline

```
#pragma omp parallel for
for(...) // ITERATION u =0
{
    Code for main cores
    x=array1[i]
    sum+=x
}

for(...) // ITERATION u=10
{
  Code for pre-translation cores
    x=array1[i]   TLB Miss Generated
}
```





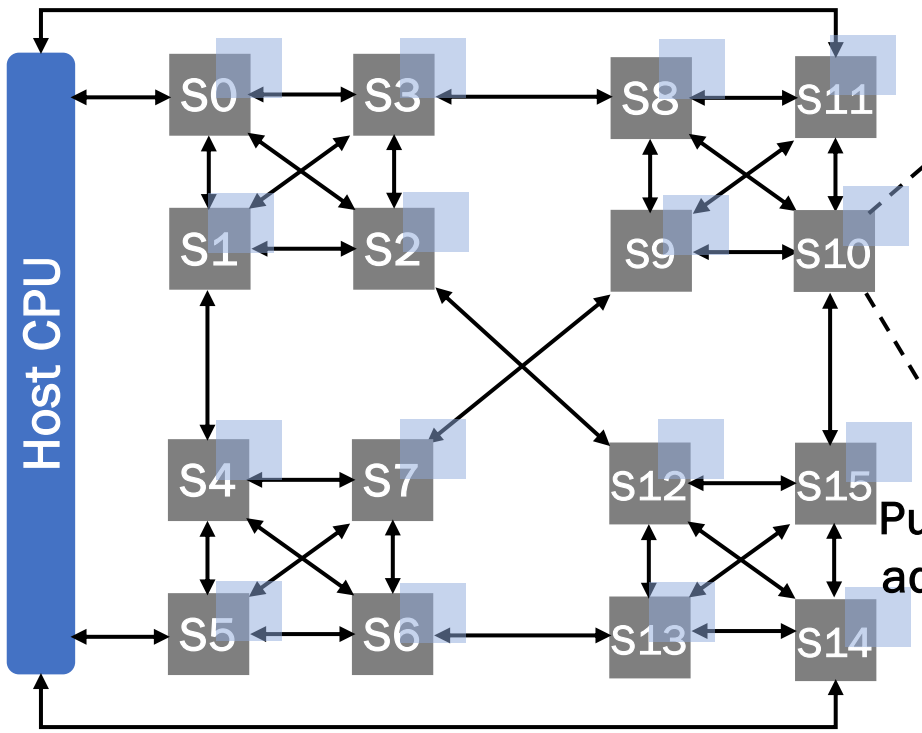
Perform PTW u=10

Key Idea: Pre-translation of page table walks

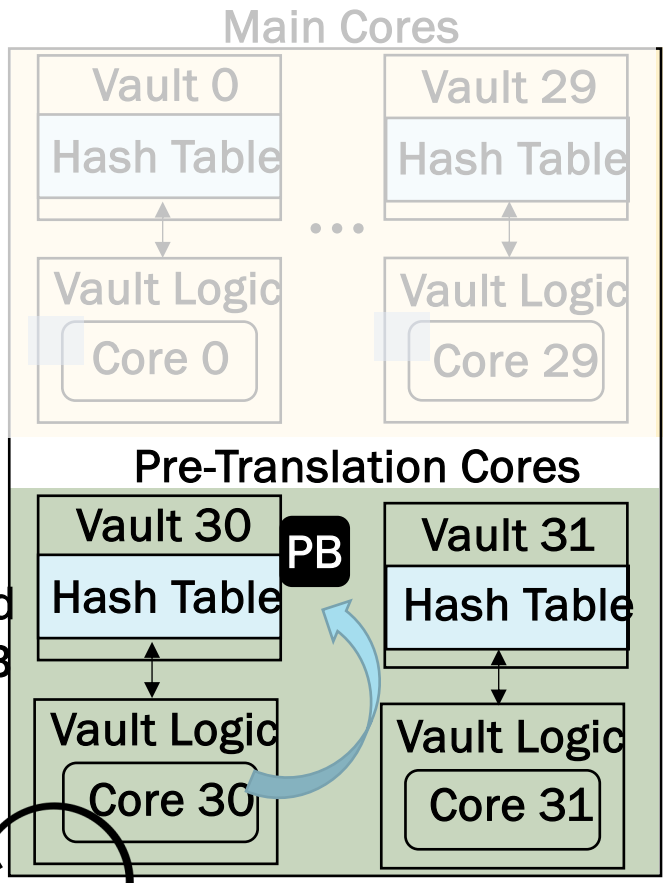
Step 1

Timeline

```
#pragma omp parallel for
for(...) // ITERATION u =0
{
    Code for main cores
    x=array1[i]
    sum+=x
}

for(...) // ITERATION u=10
{
  Code for pre-translation cores
    x=array1[i]   TLB Miss Generated
}
```



Put translated address in PB



Perform PTW u=10

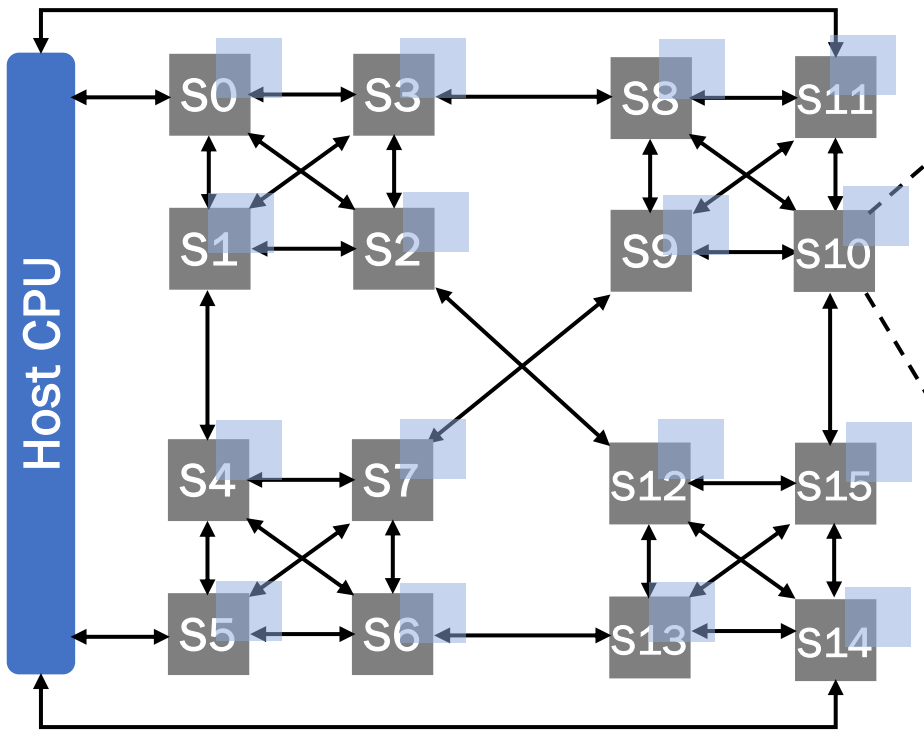
PB Pre-Translation Buffer

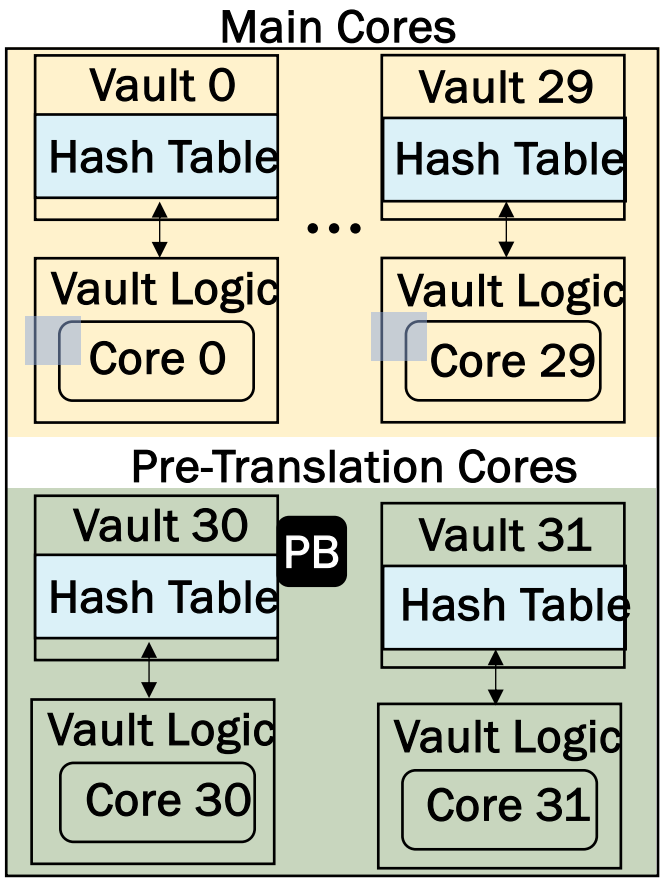
Key Idea: Pre-translation of page table walks

Step 1 | Step 2

Timeline

```
#pragma omp parallel for
for(...)
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...)
{
    Code for pre-translation cores
    x=array1[i]
}
```





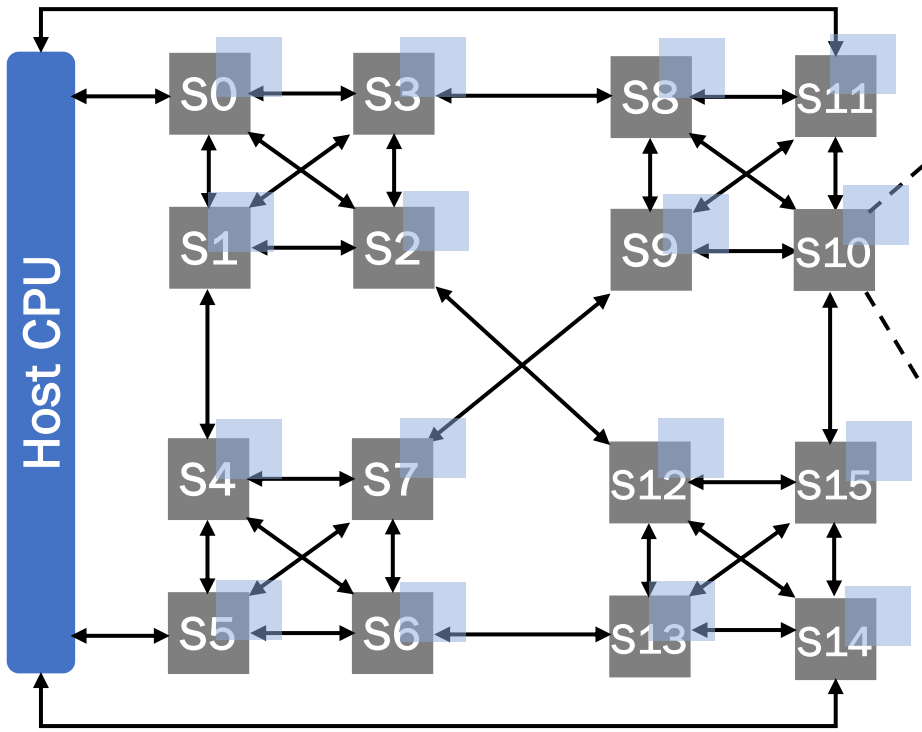
PB Pre-Translation Buffer

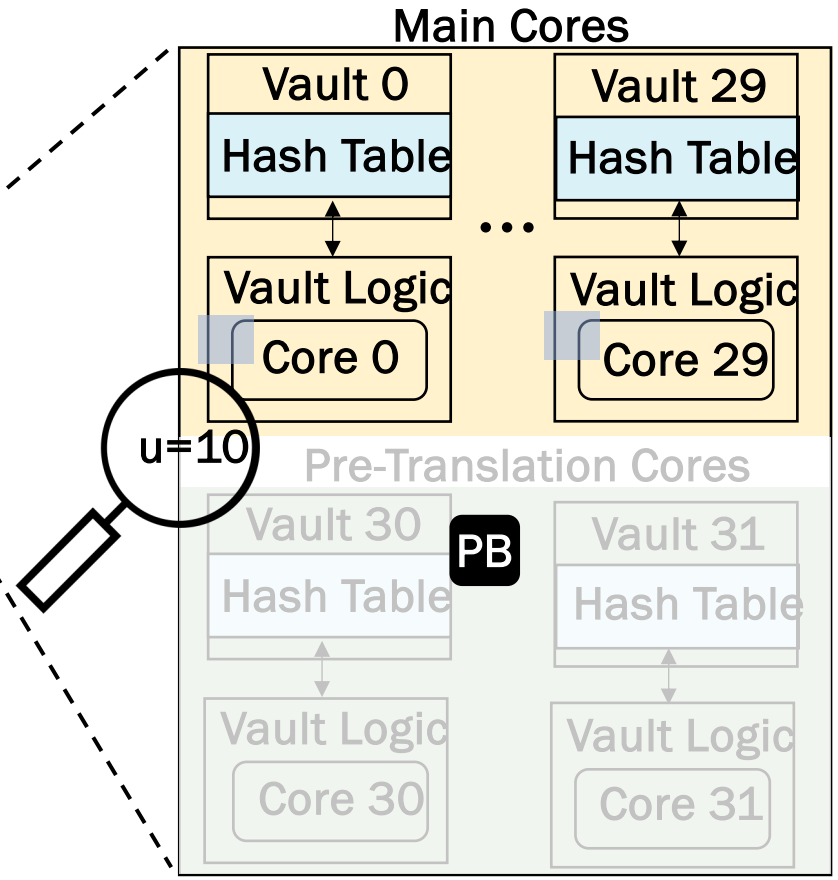
Key Idea: Pre-translation of page table walks

Step 1 | Step 2

Timeline

```
#pragma omp parallel for
for(...) // ITERATION u =10
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...) // ITERATION u=20
{
    Code for pre-translation cores
    x=array1[i]
}
```





PB Pre-Translation Buffer

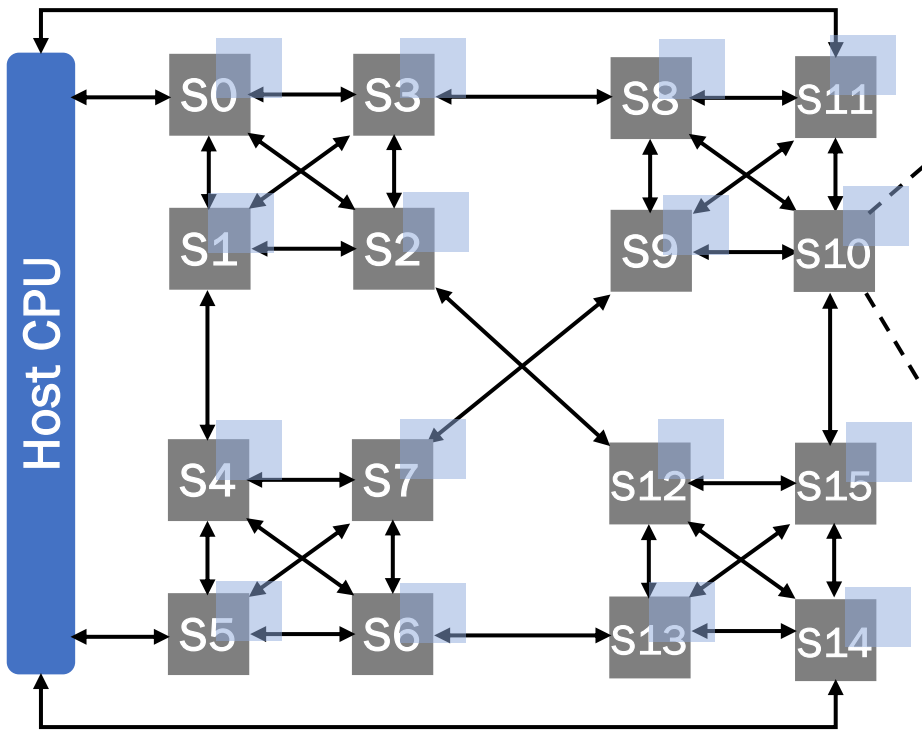
Key Idea: Pre-translation of page table walks

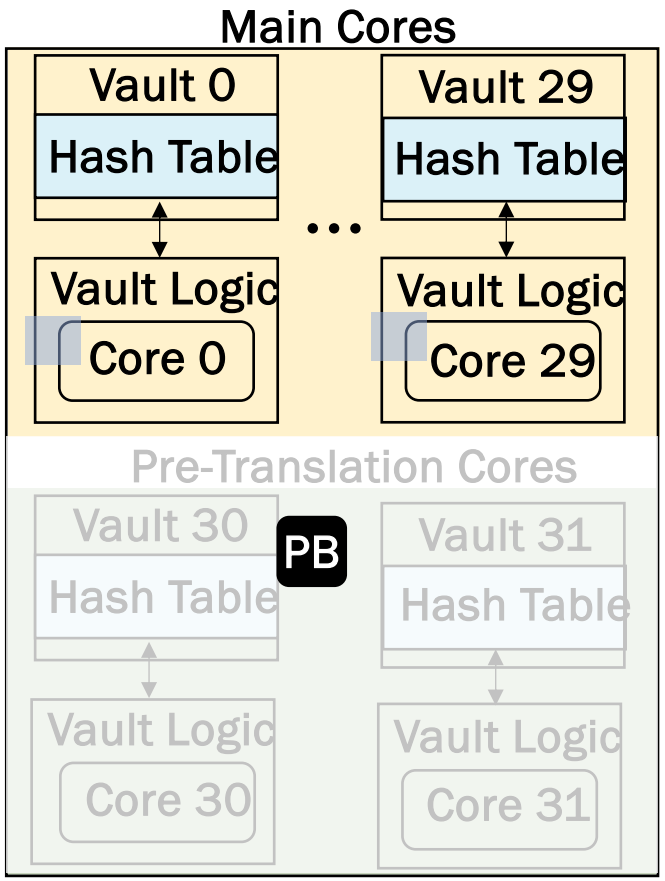
Step 1 | Step 2

Timeline

```
#pragma omp parallel for
for(...) // ITERATION u =10
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...) // ITERATION u=20
{
    Code for pre-translation cores
    x=array1[i]
}
```

TLB Miss Generated





PB Pre-Translation Buffer

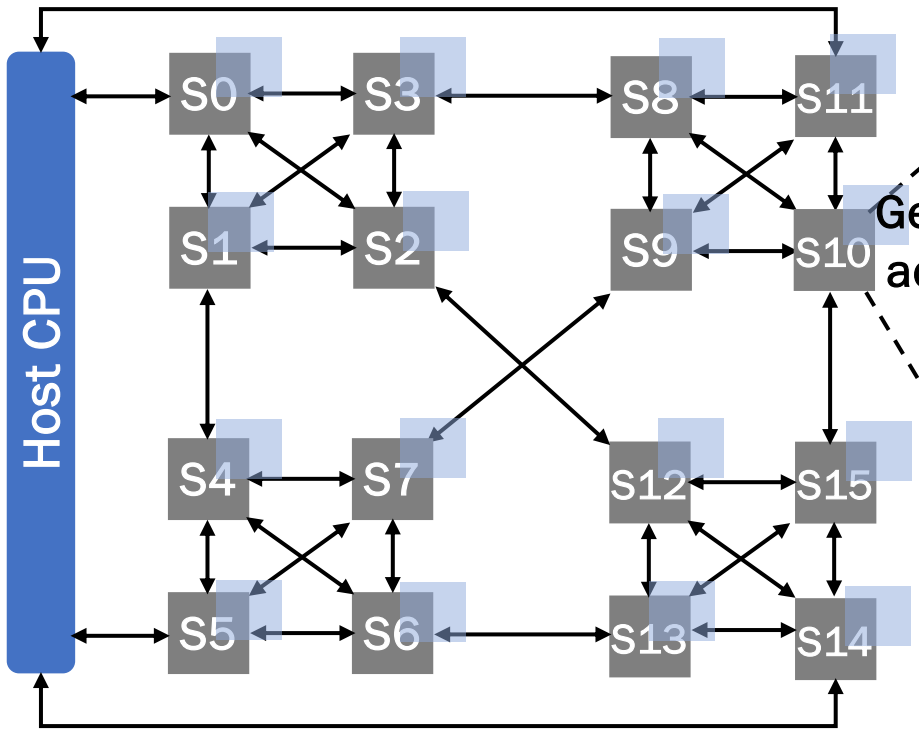
Key Idea: Pre-translation of page table walks

Step 1 | Step 2

Timeline

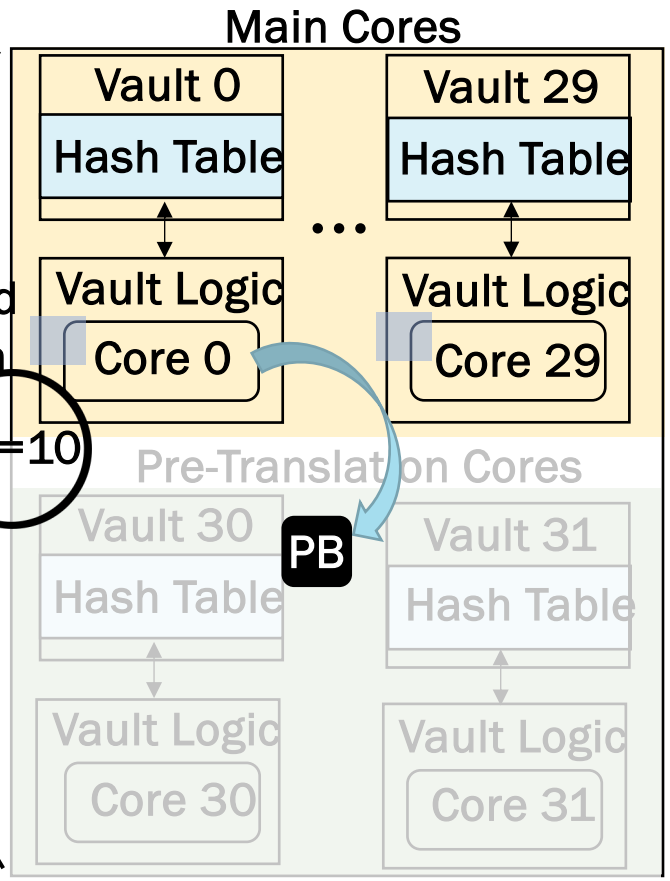
```
#pragma omp parallel for
for(...) // ITERATION u =10
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...) // ITERATION u=20
{
    Code for pre-translation cores
    x=array1[i]
}
```

TLB Miss Generated



Get translated address from PB

u=10



PB Pre-Translation Buffer

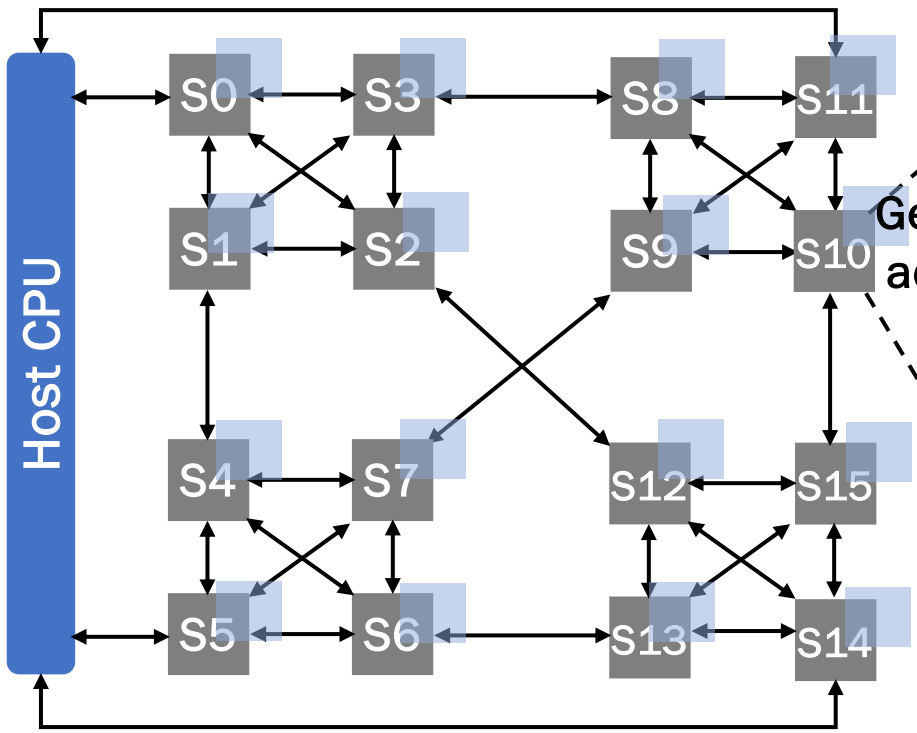
Key Idea: Pre-translation of page table walks

Step 1 | Step 2

Timeline

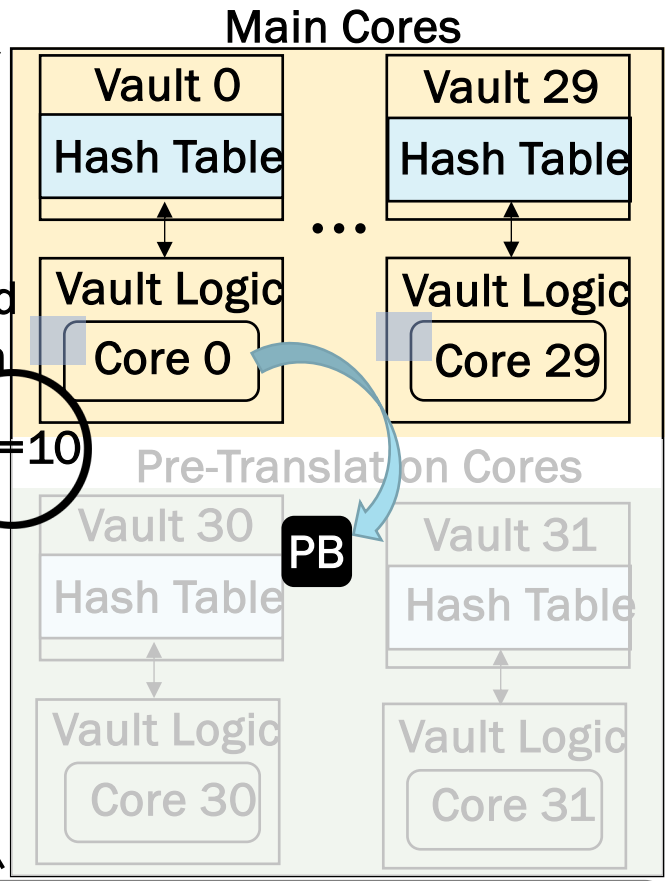
```
#pragma omp parallel for
for(...) // ITERATION u =10
{
    Code for main cores
    x=array1[i]
    sum+=x
}
for(...) // ITERATION u=20
{
    Code for pre-translation cores
    x=array1[i]
}
```

TLB Miss Generated



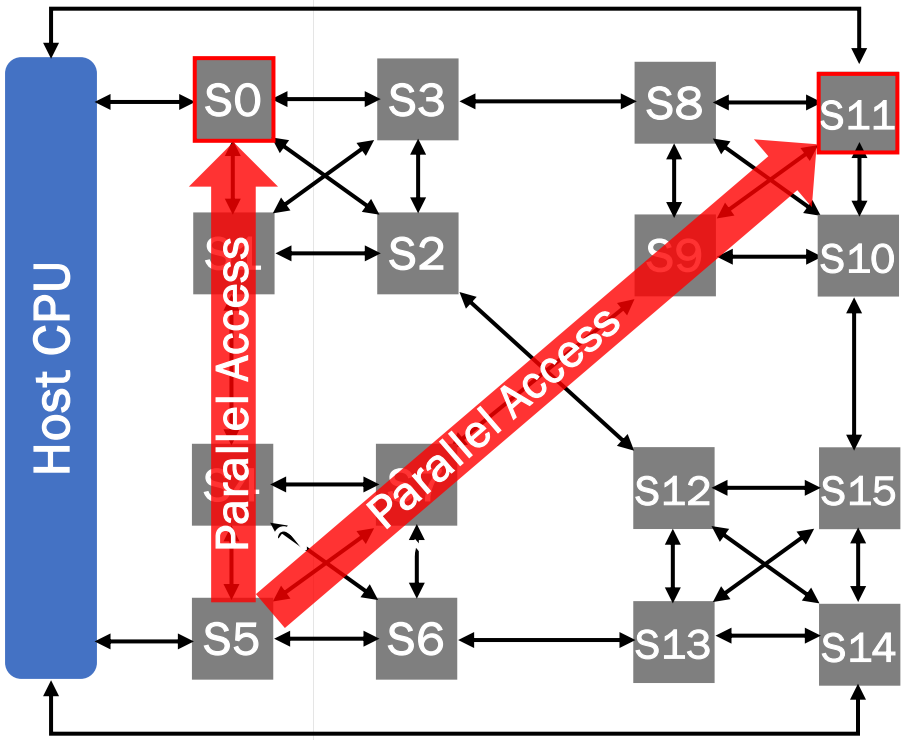
Get translated address from PB

u=10

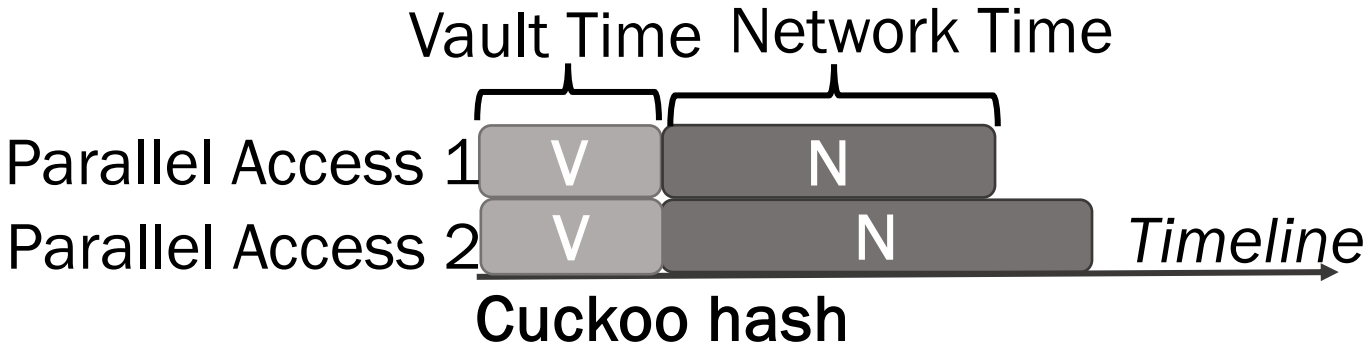


Pre-translation moves the page table walk latency off the critical path of execution

Summary of our approaches

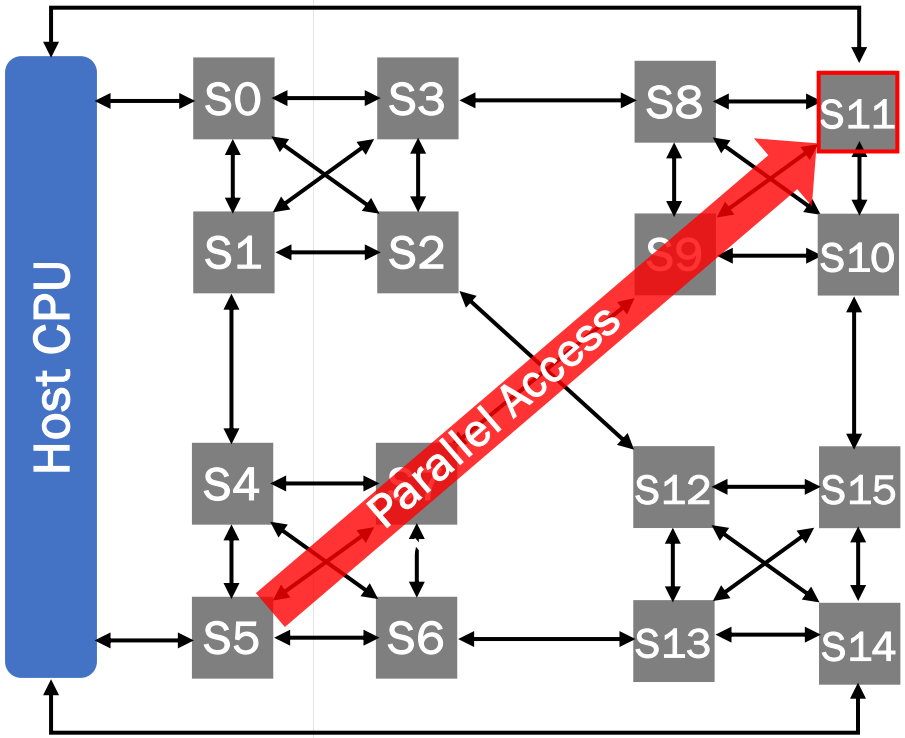


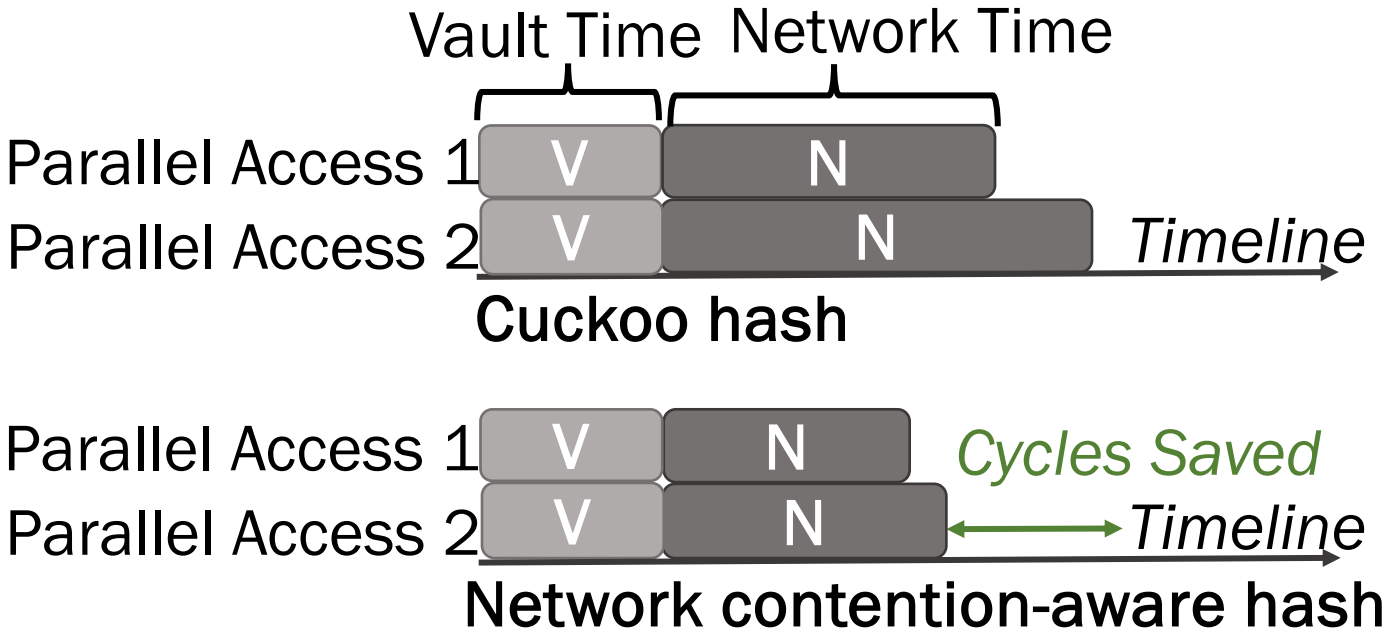
TLB Miss at Stack 5



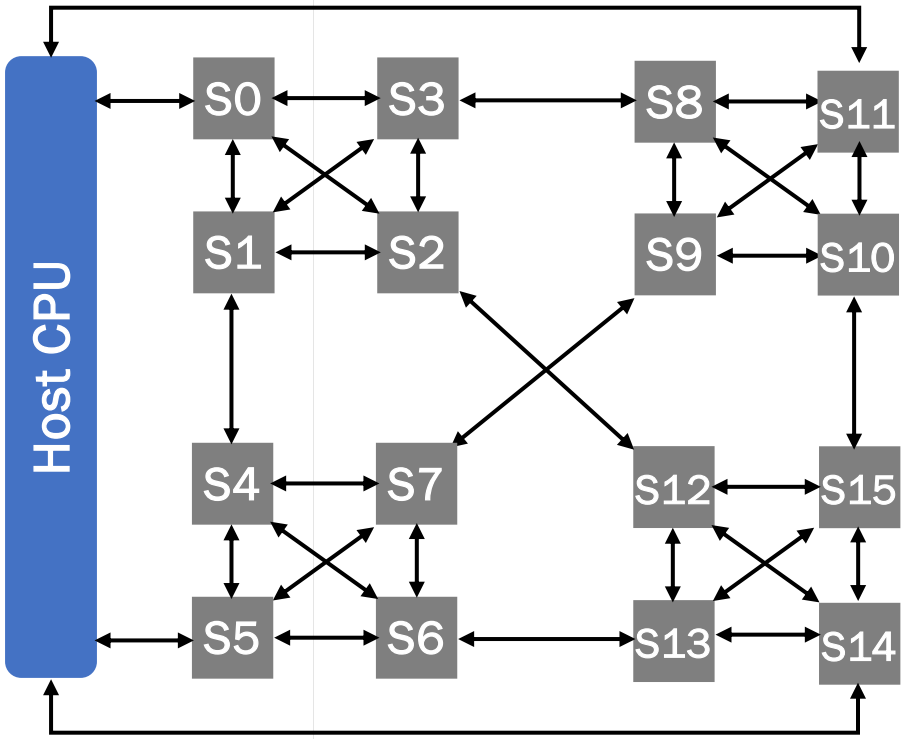
Cuckoo hash

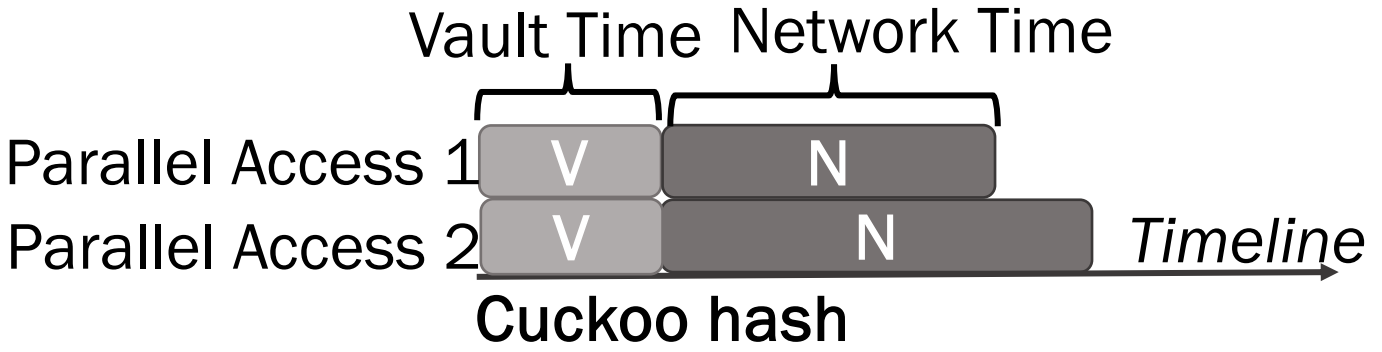
Summary of our approaches

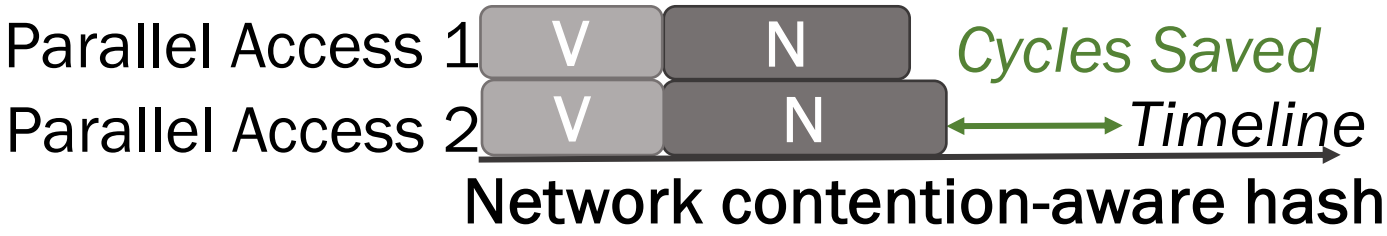


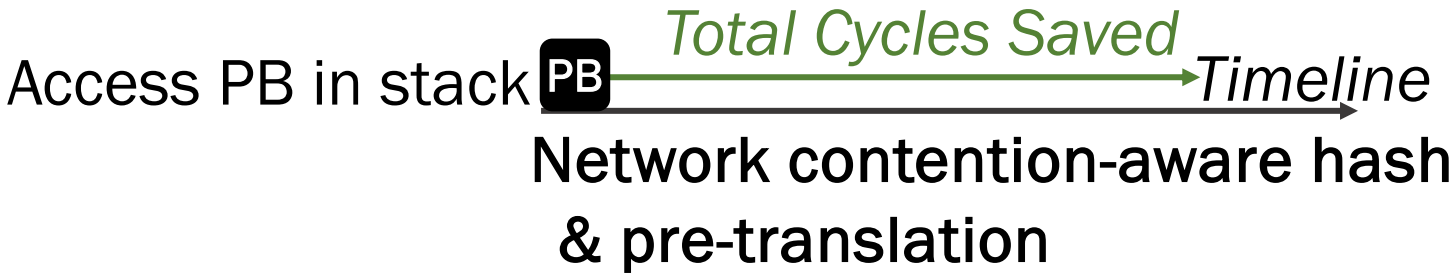


Summary of our approaches









Outline

- Processing-in-Memory Introduction
- Address Translation in PIM Architecture
- Challenges and Key Ideas
- Evaluation
- Conclusion

Methodology

Simulation Infrastructure (MultiPIM)

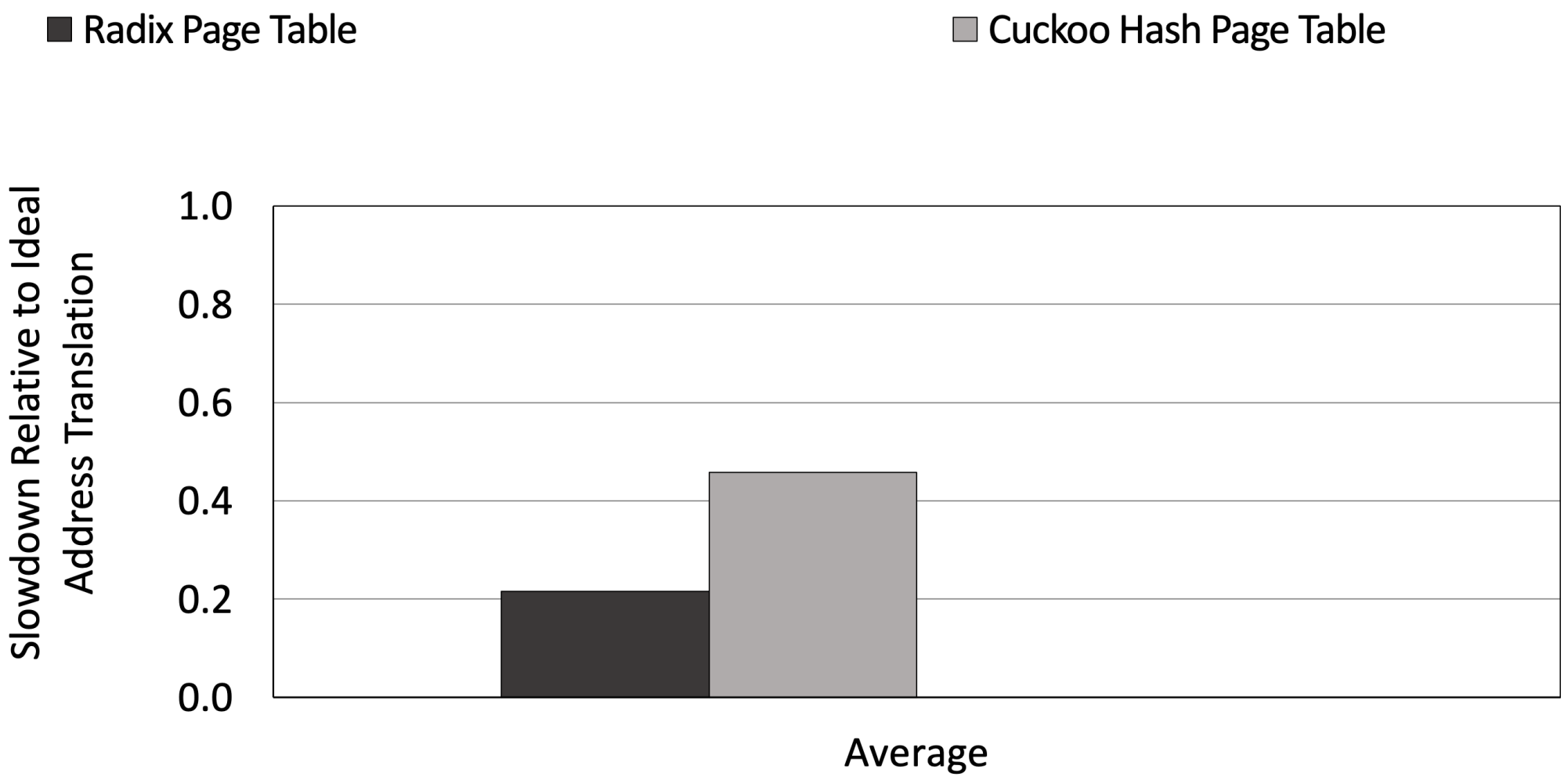
Configuration for Simulated PIM stacks		
Memory	Total number	16
	Memory interconnection	Dragonfly
	Capacity per memory	4GB, 4 layers, 8 banks per vault
	Vaults per memory	32, 32 TSV per vault
	DRAM	FR-FCFS, tCL=tRCD=tRP=17ns, tCWL=13ns
PIM Core	Type	In-order, 1 issue @ 2000MHZ
	L1I/D-Cache	Split 32KB, 4ways, 64B cacheline, LRU, write-back, MESI
	I/D-TLB	Split 64 entries, LRU
	Page table walker	1
	Pre-Translation Buffer Size	1024 entries per stack

Methodology

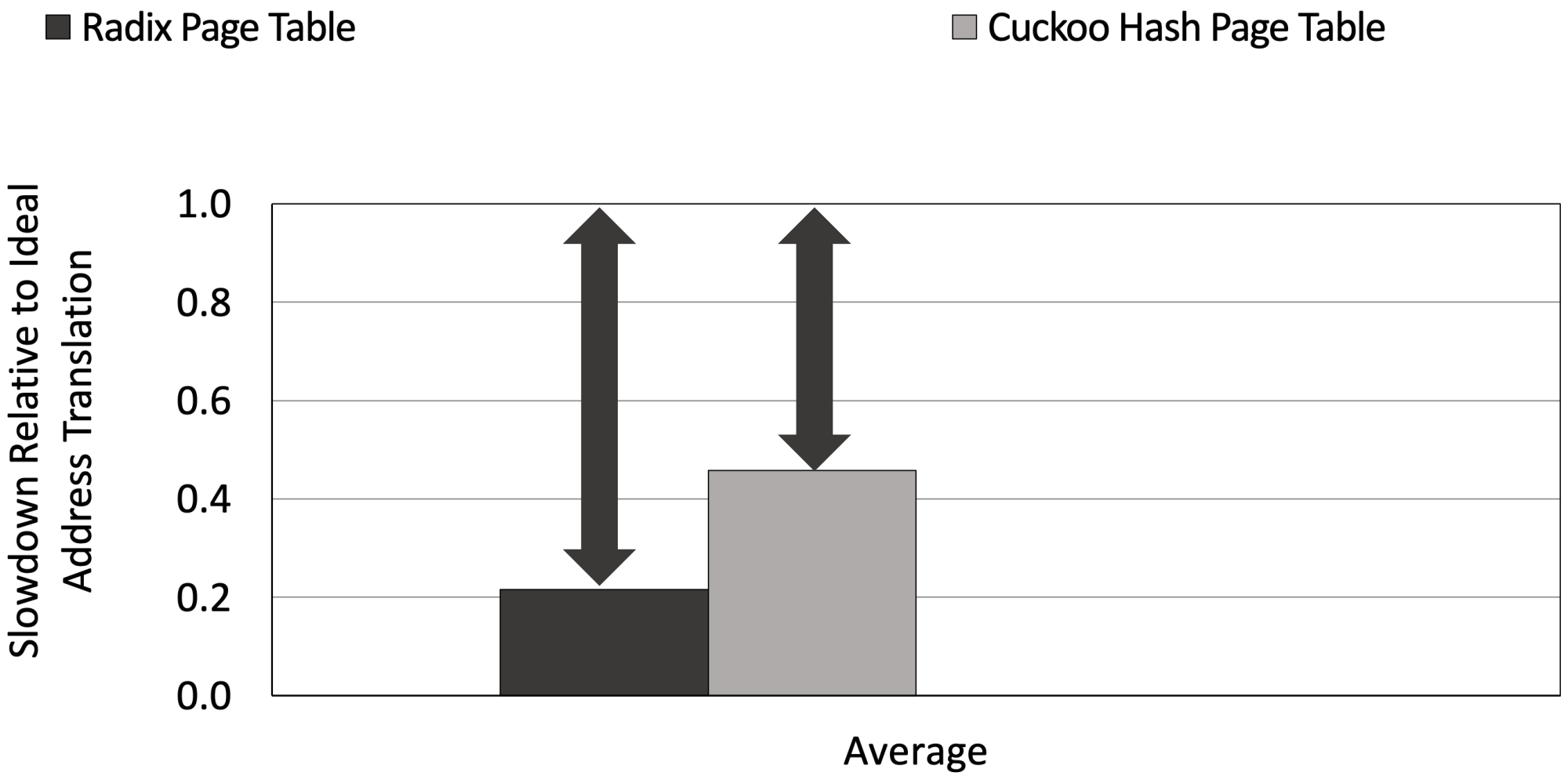
Workloads:

- Gapbs benchmark suite with workloads:
  - bfs
  - tc
  - cc
  - cc_sv
  - sssp
  - pr
- Parboil benchmark suite using the largest available dataset
  - spmv
  - sgemm
  - stencil
  - mri-grid
  - mri-q
  - histo
  - tpacf
  - cutcp
  - lbm

Performance Improvement

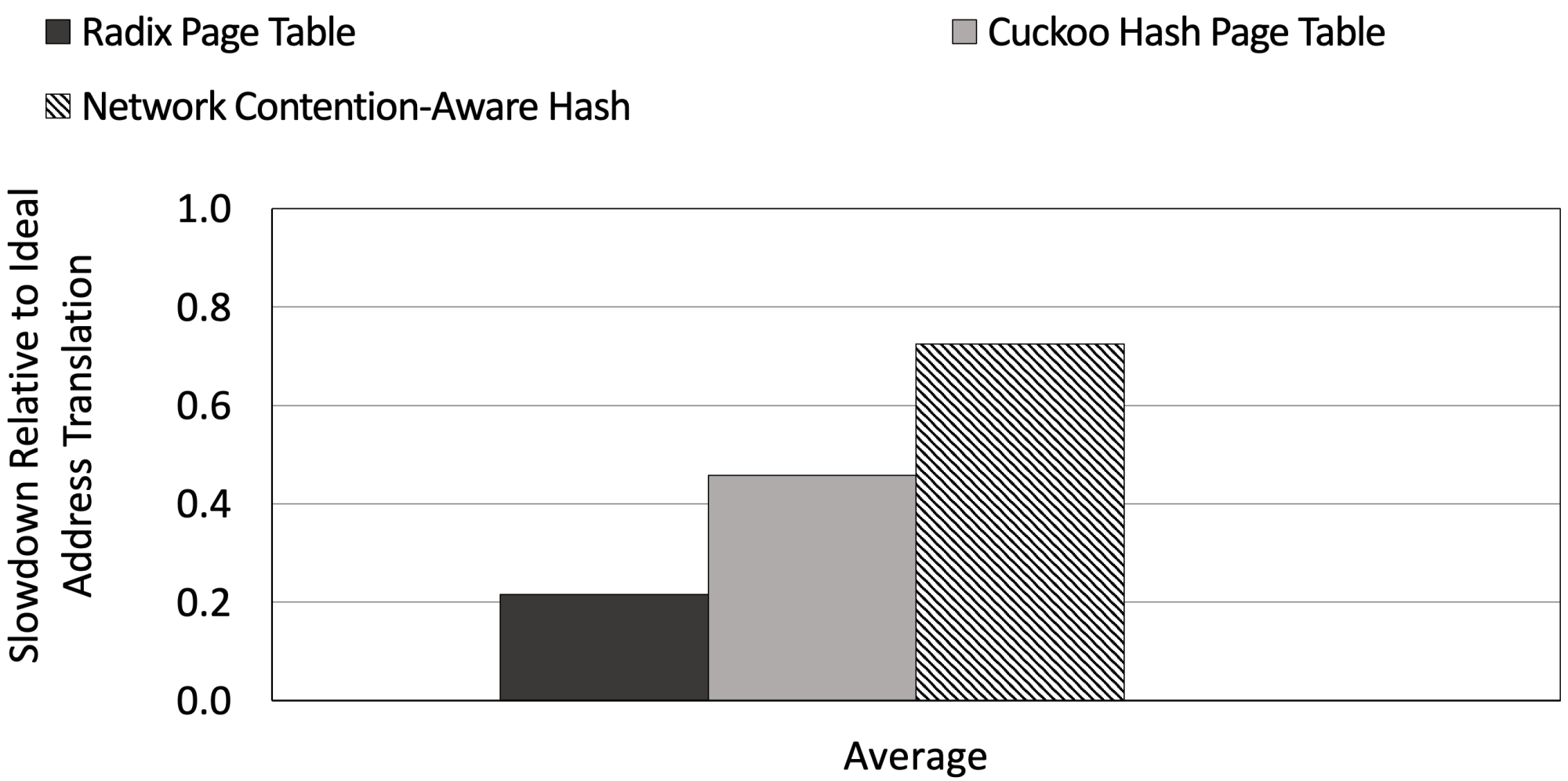


Performance Improvement

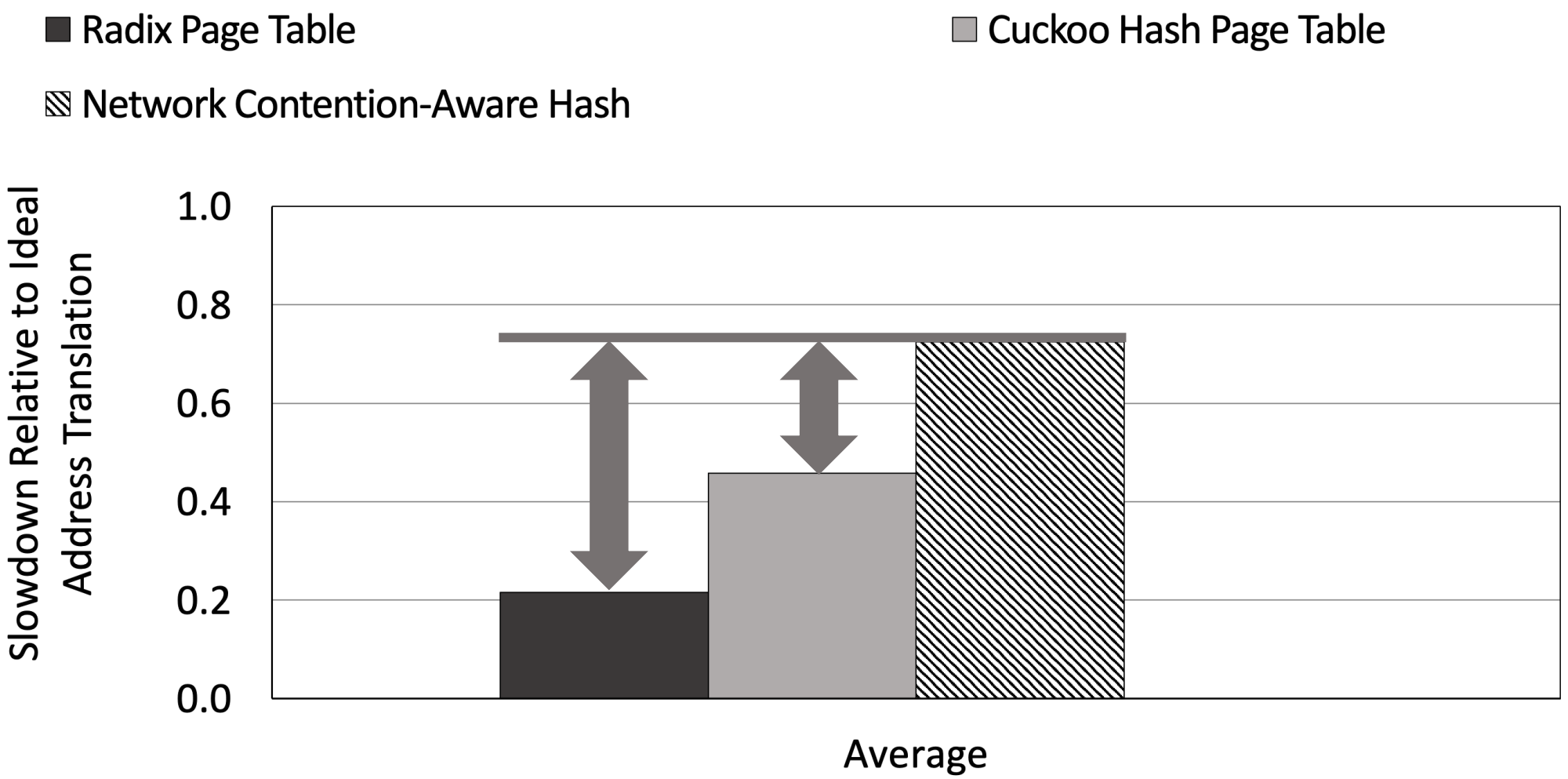


Huge performance overhead with radix and hash page table in PIM

Performance Improvement

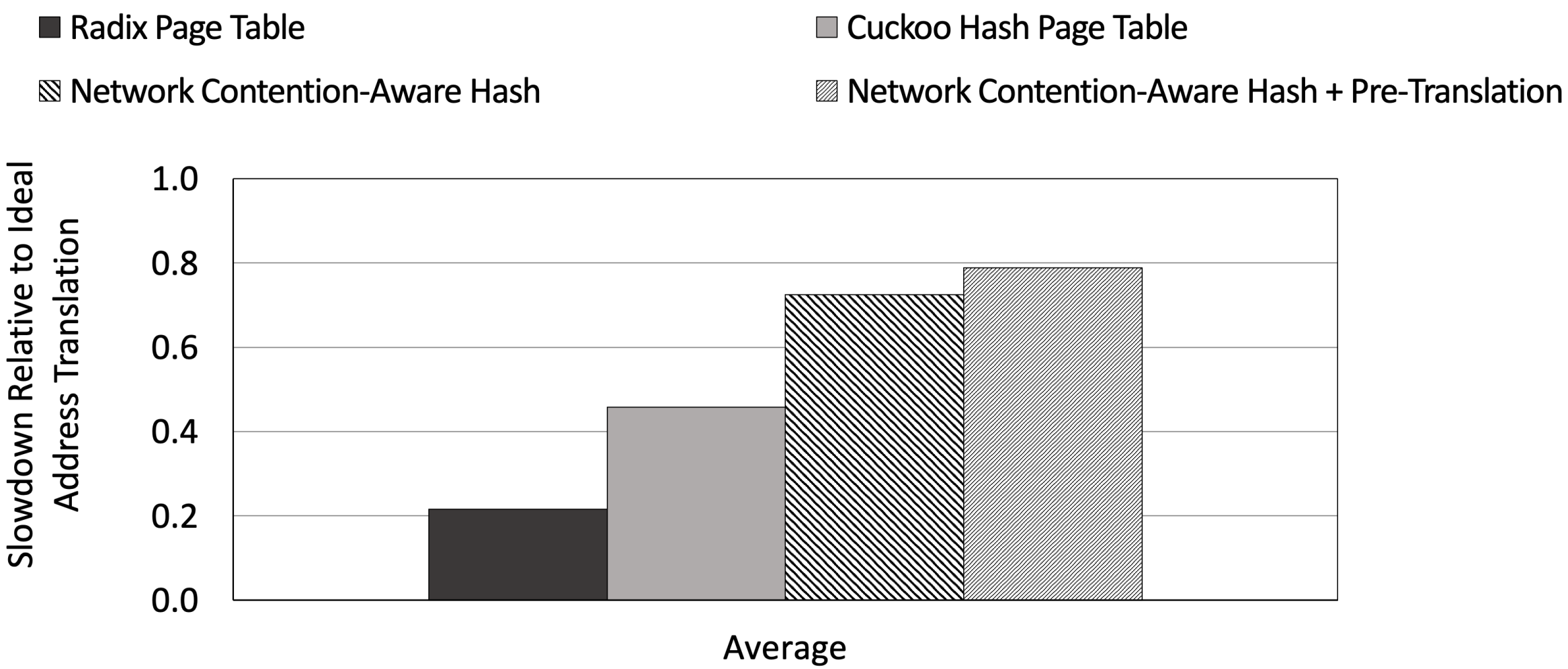


Performance Improvement

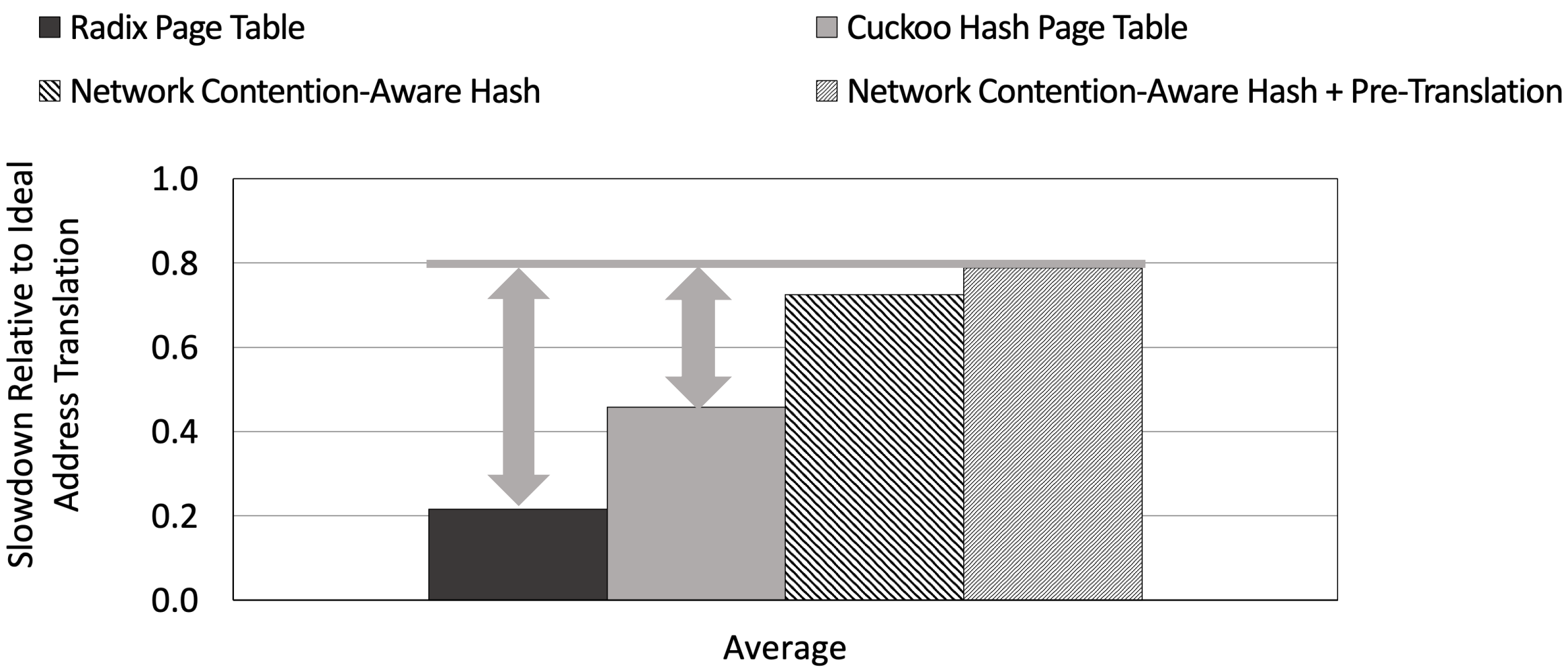


Network contention-aware hash achieves a speedup of 3.97× and 1.6× compared to radix and cuckoo hash page table

Performance Improvement

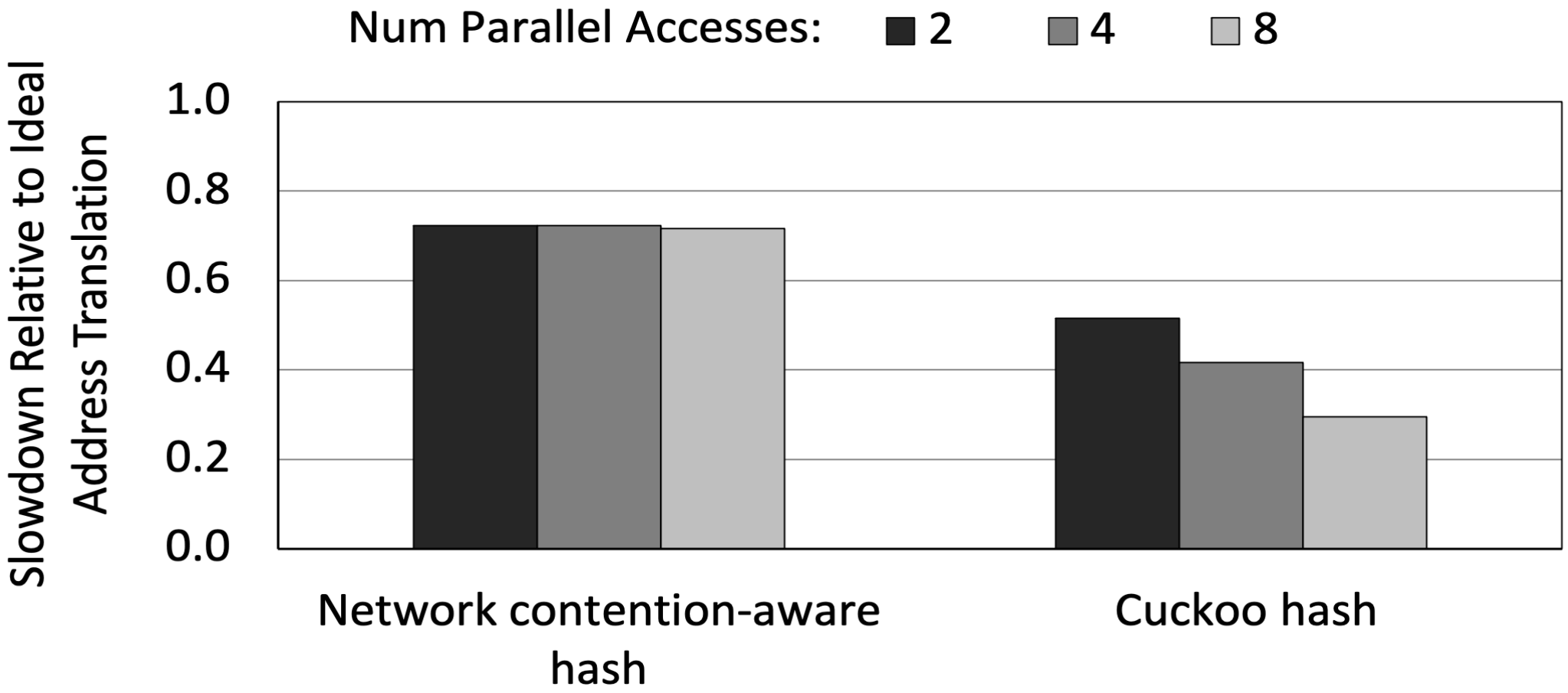


Performance Improvement

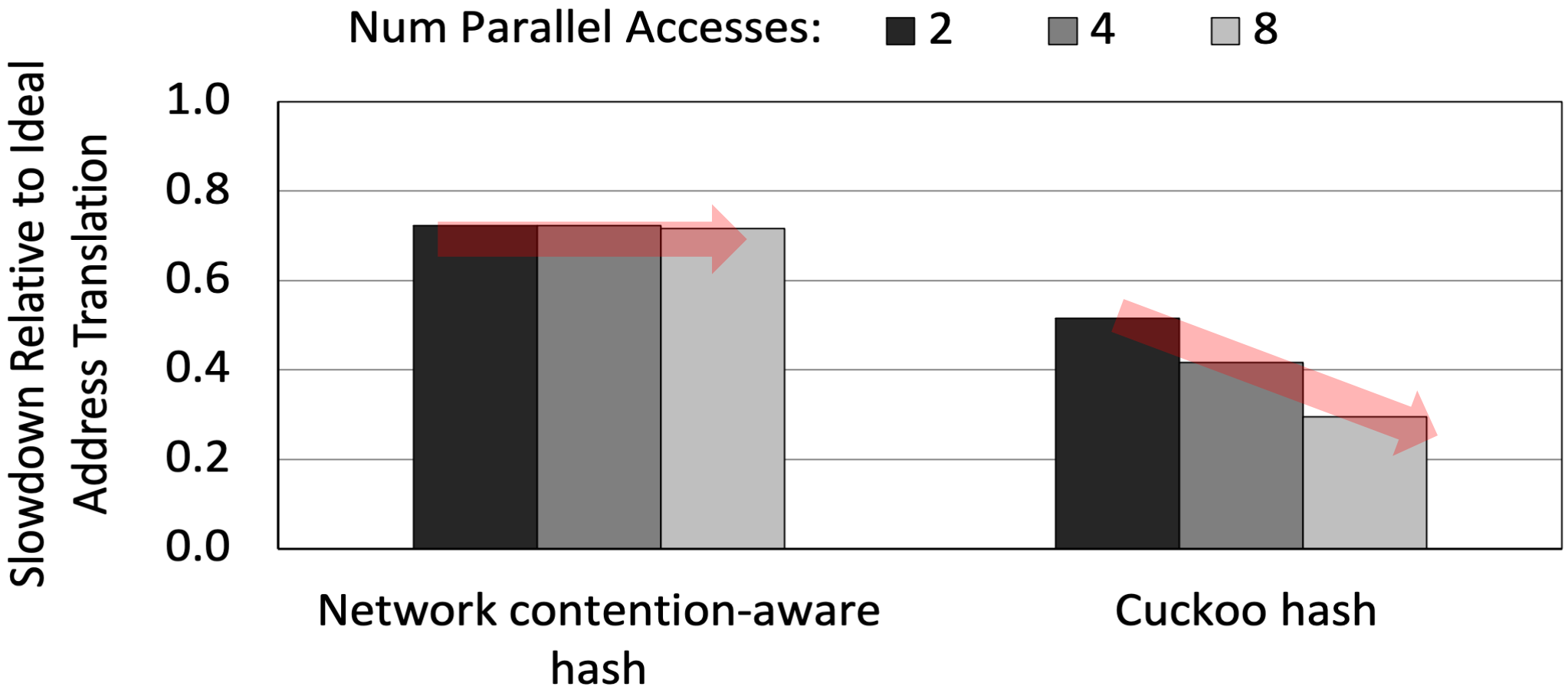


A total speedup of 4.4× and 1.7× is achieved compared to radix and cuckoo hash page table

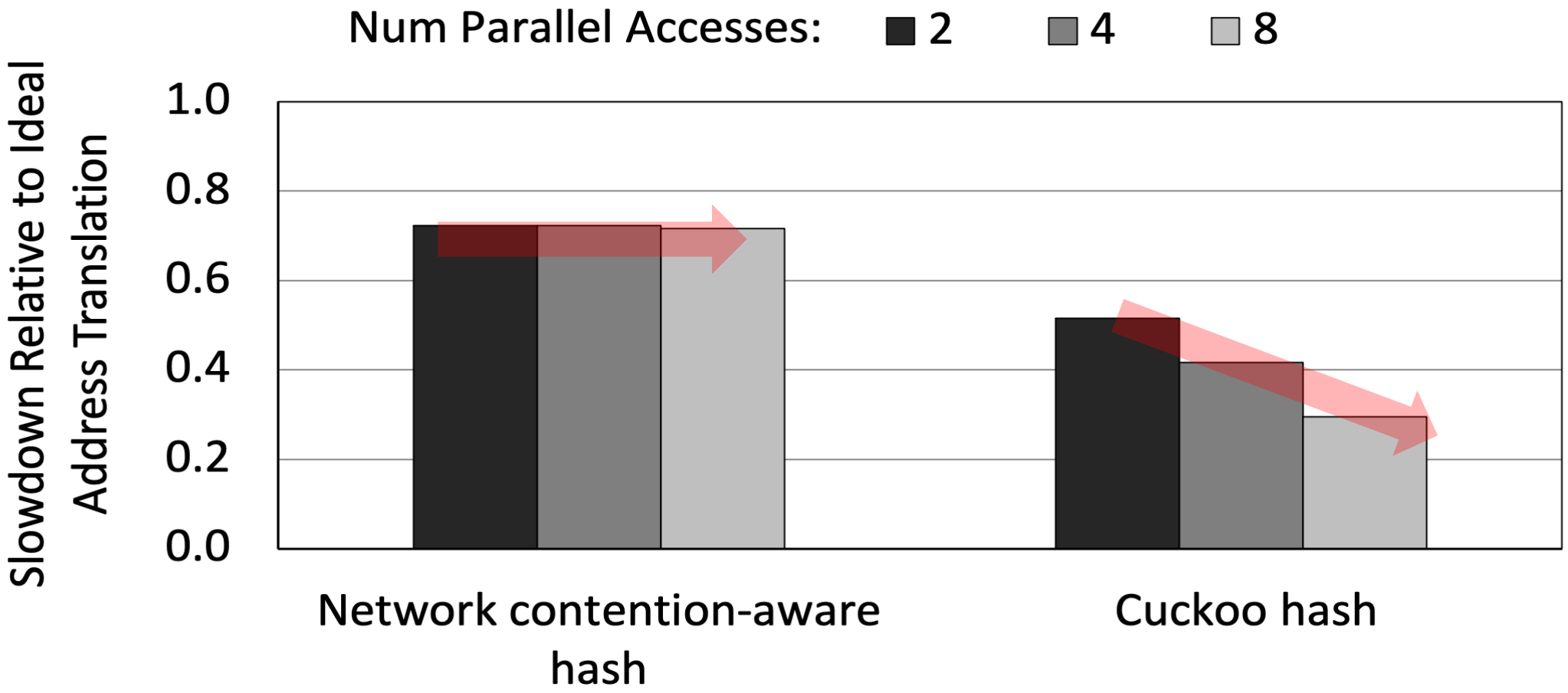
Sensitivity Results



Sensitivity Results

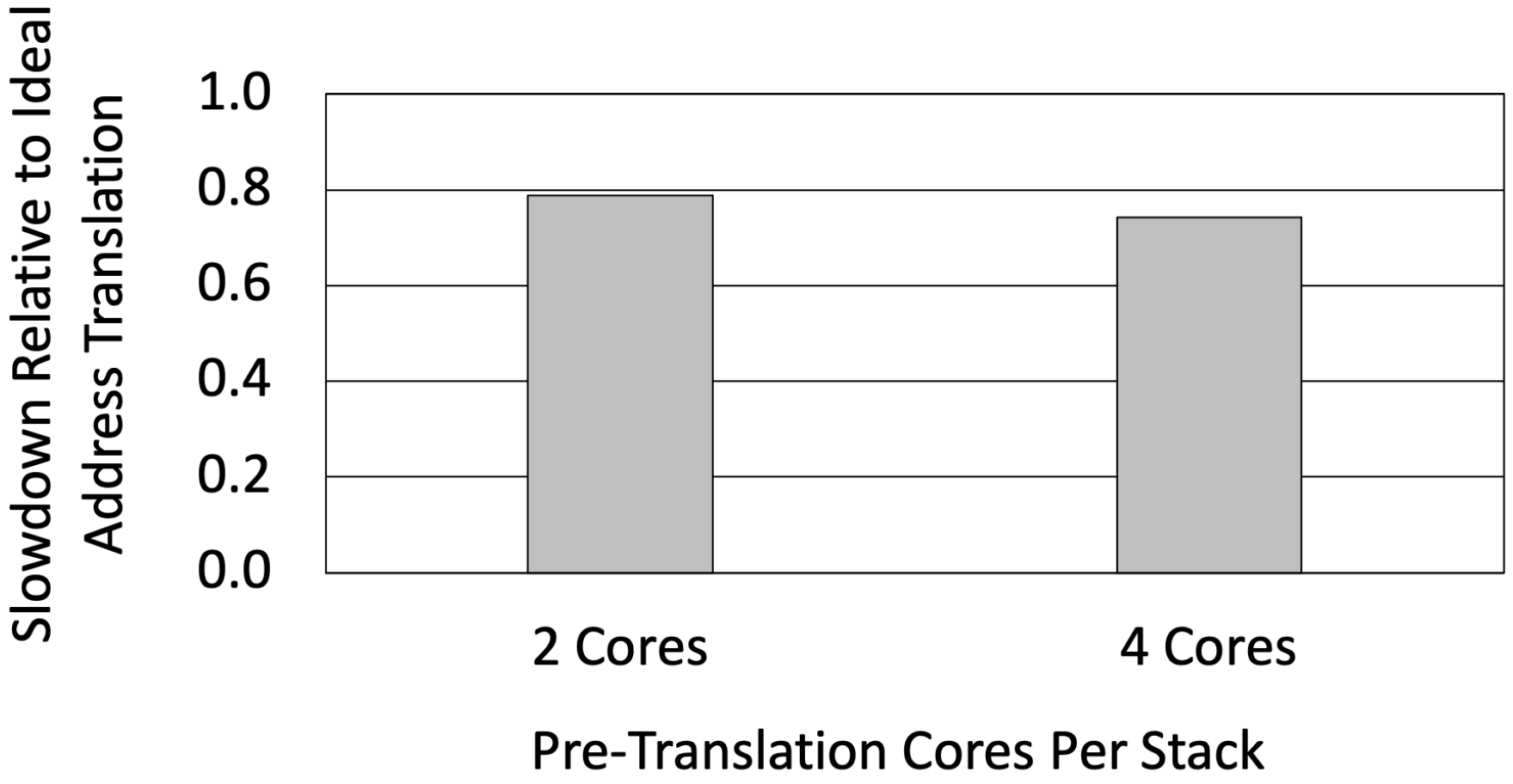


Sensitivity Results

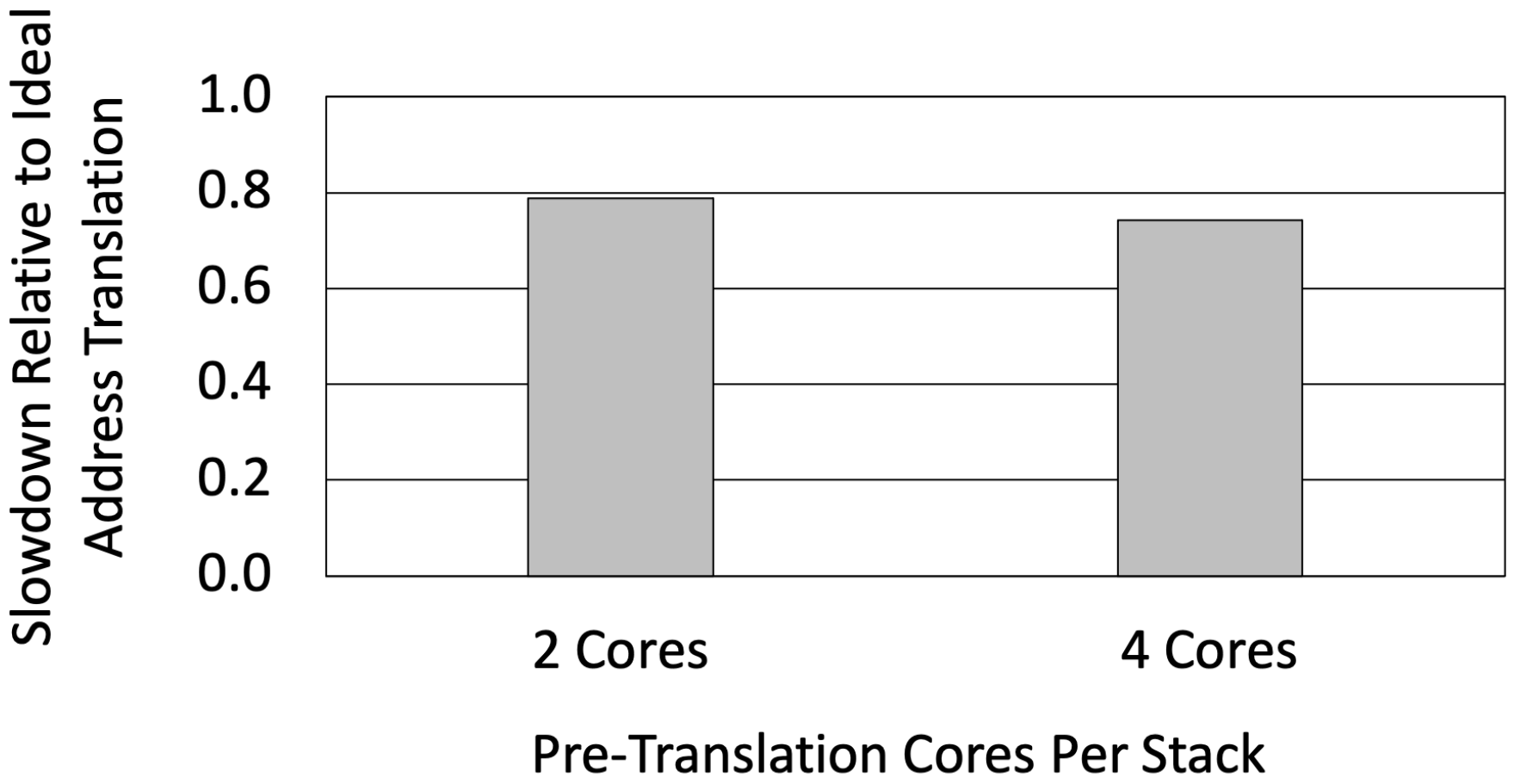


Network contention-aware hash scales well with minor performance degradation

Sensitivity Results



Sensitivity Results



2 cores is a good tradeoff between performance loss and pre-translation

Outline

- Processing-in-Memory Introduction
- Address Translation in PIM Architecture
- Challenges and Key Ideas
- Evaluation
- Conclusion

Conclusion

Multiple (PIM) stacks provide increased capacity for emerging data-intensive workloads

Running generalized applications divided between CPU and PIM is highly important

We propose address translation for multi-stack PIM running same application in CPU & PIM

Key Ideas:

- Network-contention-aware hash: Adapts cuckoo hash layout for multi-stack PIM system
- Pre-translation: Pre-translates addresses to avoid cross-stack page table walk accesses

Result: Provides 4.4x and 1.7x speedup compared to radix and cuckoo hash