# Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

Rahul Bera, Adithya Ranganathan, Joydeep Rakshit, Sujit Mahto, Anant V. Nori, Jayesh Gaur, Ataberk Olgun, Konstantinos Kanellopoulos, Mohammad Sadrosadati, Sreenivas Subramoney, Onur Mutlu

51st International Symposium on Computer Architecture (ISCA)

## Problem : Load Instruction

- Load instructions cause:
  - Data dependence: results of instructions need to be consumed by the other.
  - Resource dependence: multiple instructions contend for the same hardware resource.

- Thus, load instructions are major source of ILP limitation.

## Problem : Load Instruction in Detail

- Load instruction takes longer latency than non-memory instructions.

- Why? Load instructions need two operations:
  1. Compute the load address
  2. Fetch data by accessing memory hierarchy

-> Power & Performance Overhead

# Prior Works : LVP & MRN

- Load Value Prediction
  - LVP predicts the value of a load instruction.
  - Speculatively executes load-data-dependent instructions.

- Memory Renaming
  - MRN learns dependence relationship between a store-load instruction pair.
  - Then, speculatively executes load-dependent instruction by forwarding the data directly from the associated store instruction.

## Prior Works : Limitations

1. Predicted or forwarded value has to be verified by actually executing the load.

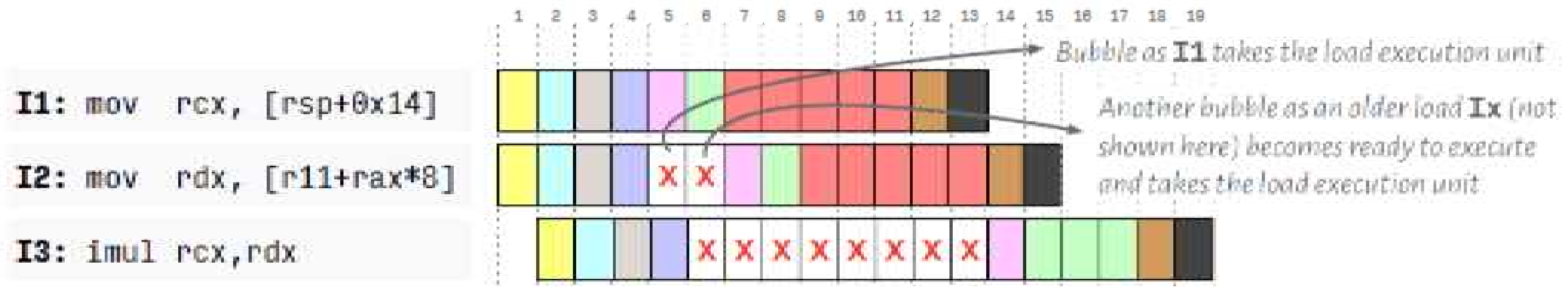
2. Incorrect prediction or forwarding: re-execution of load-dependent execution which incurs performance & power overhead.

-> LVP & MRN could mitigate data dependence but not resource dependence.

## Motivation

- Assume OOO processor with fetch, issue and retire bandwidth of two instructions and one load execution unit and perfect LVP.
- There are three instructions and two of them are load instructions.
- Comparison processor configuration without LVP, with LVP and LVP + load elimination.

# Motivation : Without LVP

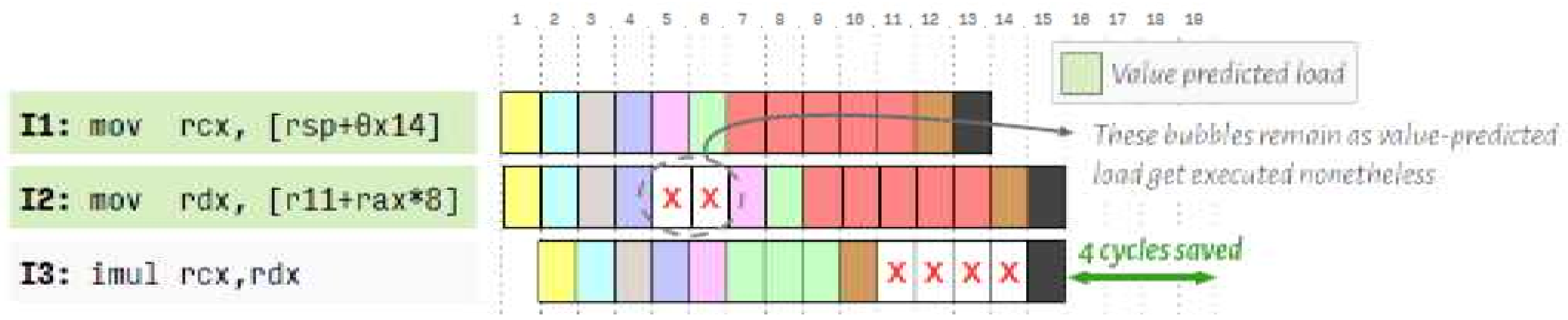

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Bubble as I1 takes the load execution unit
Another bubble as an older load Ix (not shown here) becomes ready to execute and takes the load execution unit
I1: mov rcx, [rsp+0x14]
I2: mov rdx, [r11+rax*8]
I3: imul rcx,rdx
X

- $I_1$ issued in cycle 5, thus $I_2$ stalled.

- $I_3$ has data-dependence on rdx and rcx, so it must be stalled.

- This requires 19 cycles in total.

# Motivation : With LVP

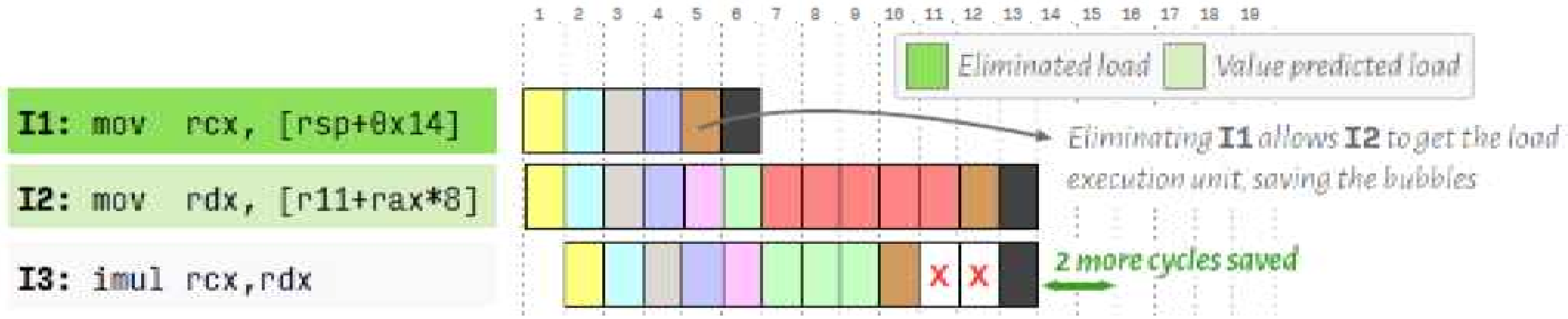


- I1 and I2 are value-predicted. Then, I3 retires 4 cycles earlier.

- But predicted values need to be verified, I2 still experiences stalls in cycle 5 and 6.

- This requires 15 cycles in total.

# Motivation : With LVP + Load Elimination



- If the execution of I1 could be safely eliminated, 2 more cycles can be saved.

- This requires 13 cycles in total.

## Constable

- As we saw above, safe elimination of load can mitigate data dependence but also the resource dependence.
- This paper proposes Constable which safely eliminates the entire execution of a load instruction.

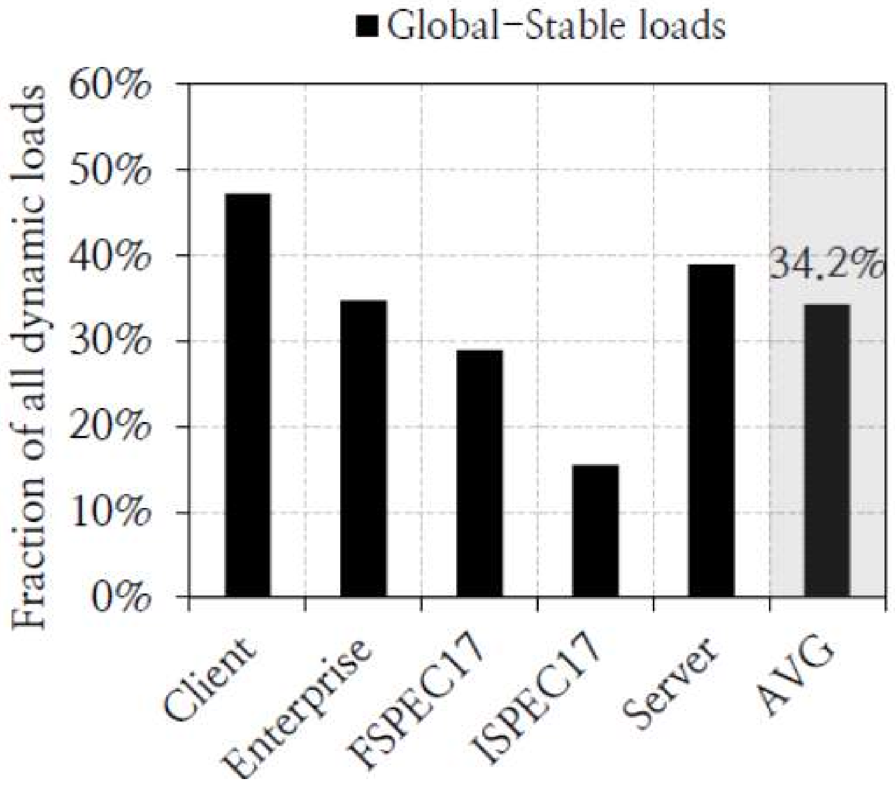
# Key Observations : Global Stable Loads

- Static load instructions repeatedly fetch the same value from the same load address across entire workload trace.

- It is called global-stable. This is prime candidate for elimination.

- Since address generation & data fetch produce same results across all dynamic instances.

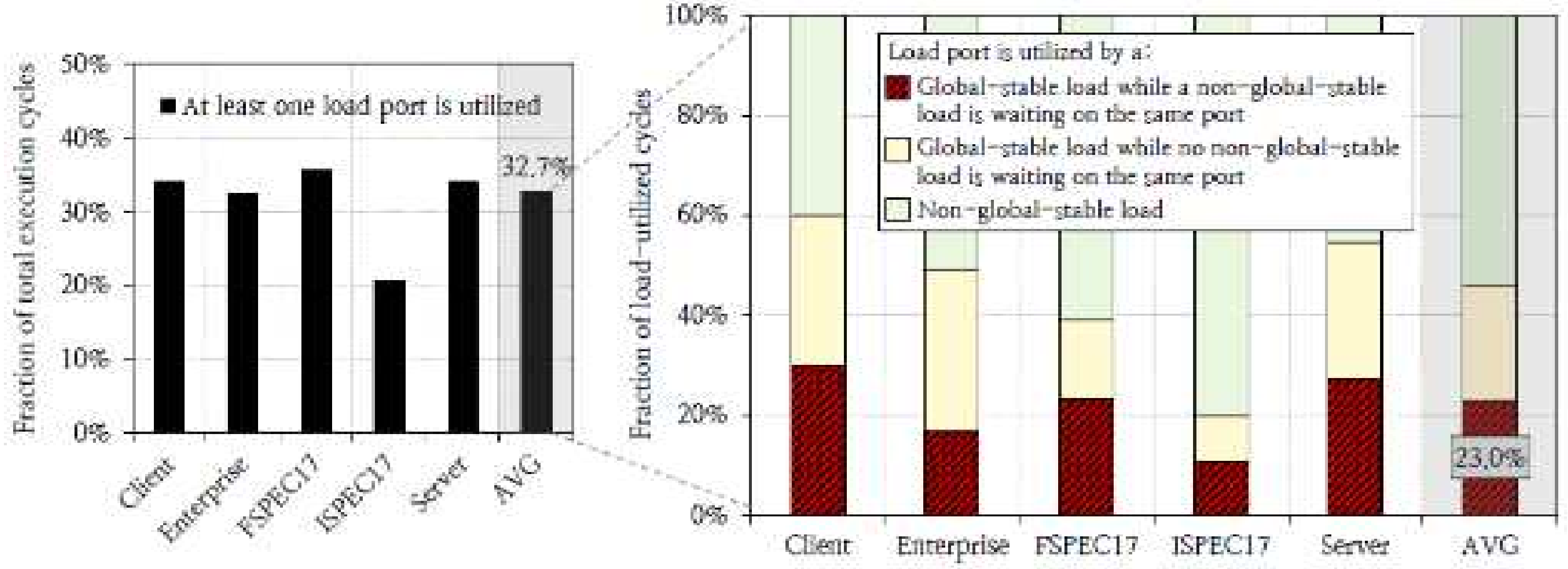
## Key Observations : Global Stable Loads



Global stable loads on real workloads

- 34.2% of all dynamic loads are global-stable.
- Significant fraction of load instructions in real workloads are global-stable.

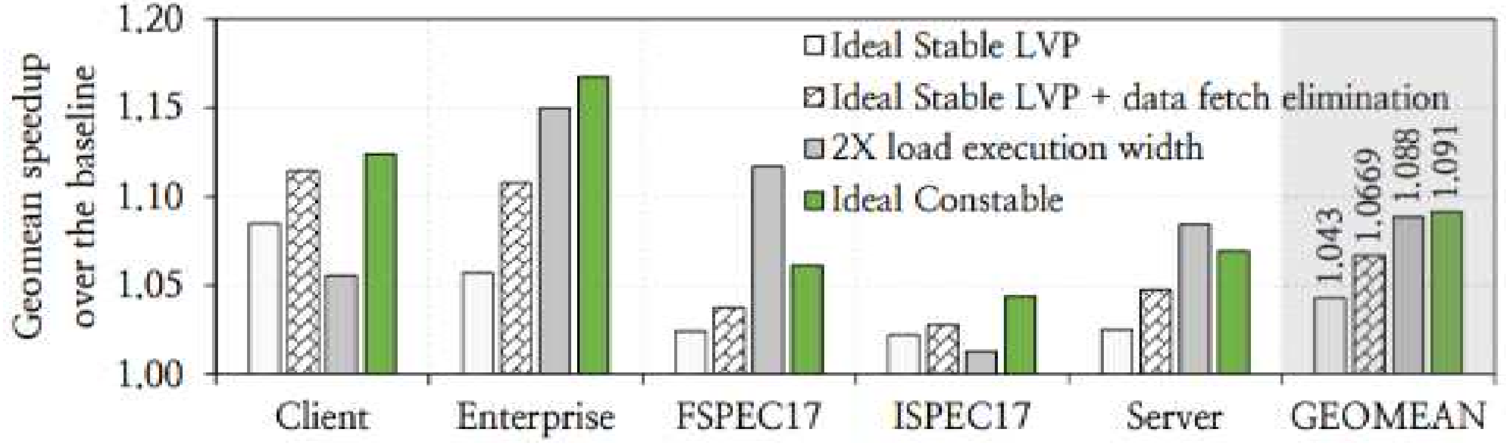
# Key Observations : Resource Dependence between Load Instructions



- Analysis of the utilization of load ports.
- Global-stable & non-global-stable are waiting to be scheduled on the same port.
- If global-stable loads are eliminated, non-global-stable could be scheduled faster.

-> Global-stable load instructions cause significant resource dependence!

# Key Observations : Performance Headroom



- Figure shows that eliminating the execution of global-stable loads has high performance headroom.

- Mitigating both data and resource dependence has higher performance potential than only mitigating data dependence.

# Constable : Key Insight

## Condition 1

- None of the source registers of I has been written between the occurrences of $I_1$ and $I_2$.

-> Address computation of $I_2$ can be safely eliminated.

## Condition 2

- No store or snoop request has arrived to the memory address of $I_1$ between the occurrences of $I_1$ and $I_2$.

-> Data fetch operation of $I_2$ can be safely eliminated.

## Constable : Key Steps

- Identifying load instructions
  - Constable dynamically identifies load instructions that have repeatedly fetched the same value from the same load address
  - This is called likely-stable.

- Tracking for modification
  - When Constable gains enough confidence that the load instruction is likely-stable,
  - Constable tracks modifications to the registers of load instruction.

# Constable : Key Steps

- Elimination of load instruction
  - Constable eliminates the execution of all future instances of the likely-stable load
  - Breaks the load data dependence using last fetched value
  - Until there is a write to the source registers.

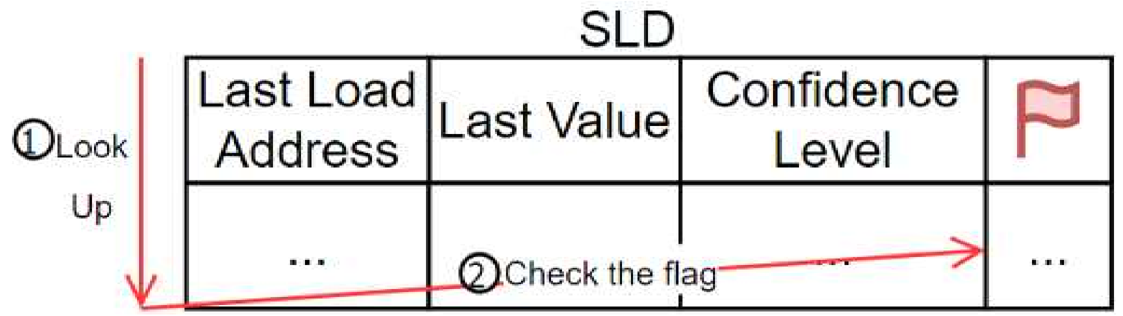
## Constable : Main Hardware

- Stable Load Detector (SLD)
  - PC-indexed table
  - Stores key information of last load instruction.

- Register Monitor Table (RMT)
  - Architectural register-indexed table to monitor modification to architectural registers.

- Address Monitor Table (AMT)
  - Physical-address-indexed table to monitor modifications in memory.
  - Also avoids eliminating a load instruction that is modified.

# Step 1 : Check SLD



① Checks the SLD using load PC during rename stage.

② Check can_eliminate flag

1) If it's set -> eliminate load execution.
2) If it's not set -> just execute and re-check the SLD at the writeback stage.

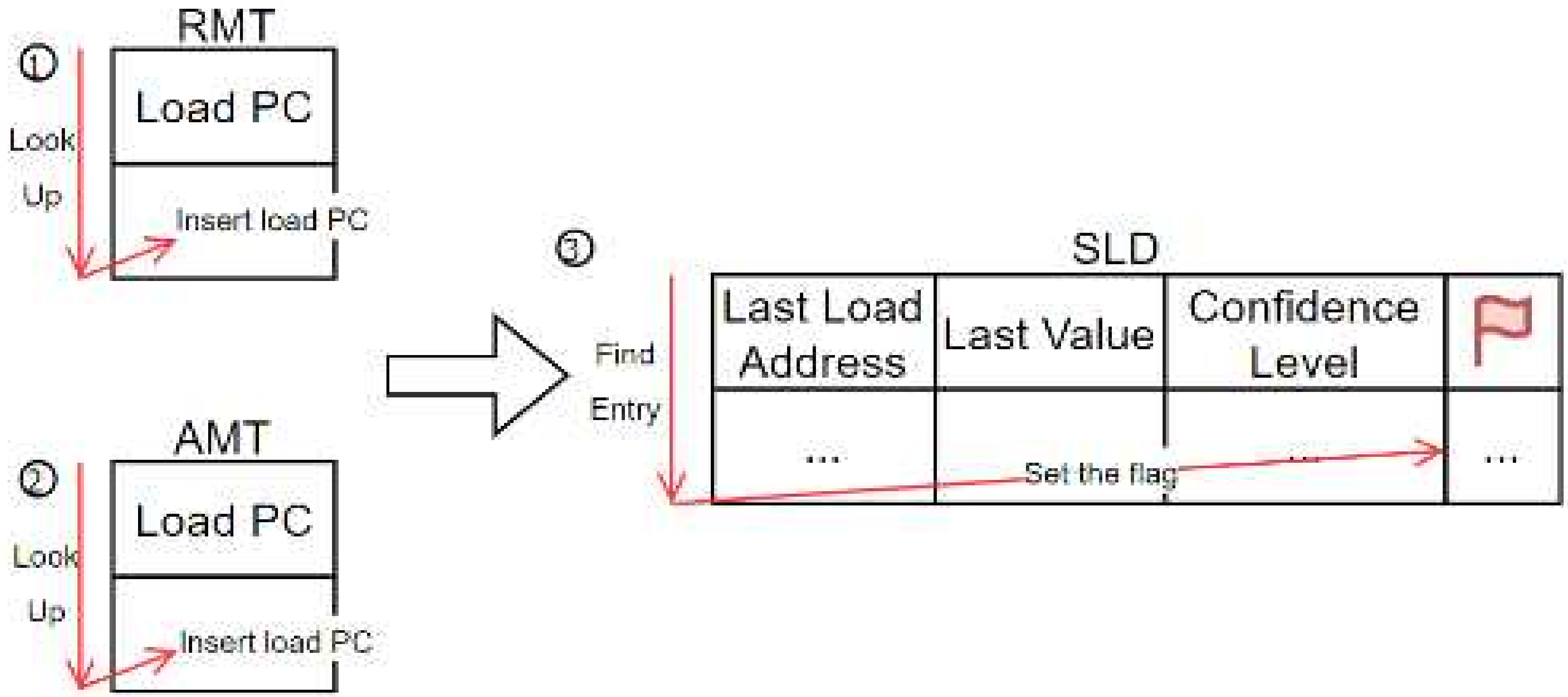
## Step 2 : Update Likely-Stable but Non-Eliminated Load

① Update RMT
  1) Looks up RMT with source register.
  2) Inserts load PC to RMT entry.

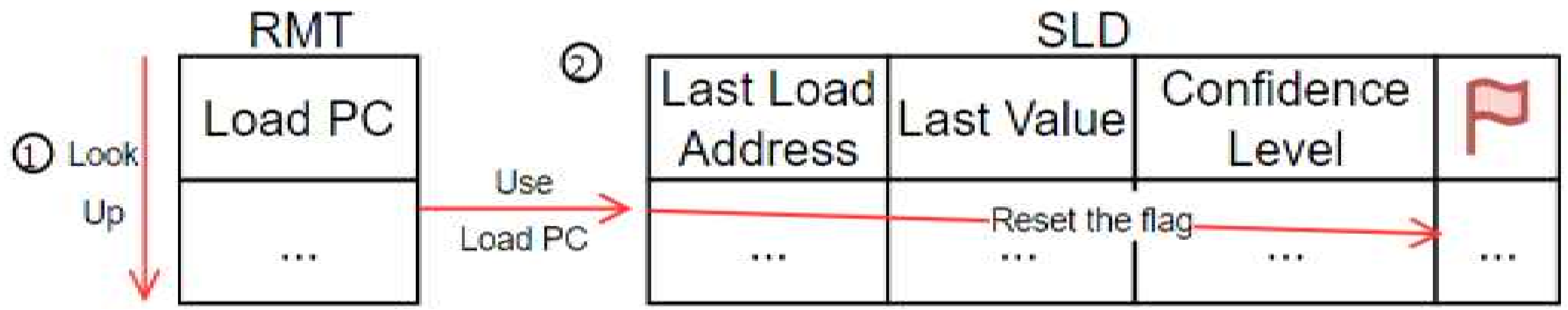
② Update AMT
  1) Looks up AMT with physical address of load instrunction.
  2) If it's found -> insert load PC into AMT entry.
  3) If it's not found -> inserts load PC into new AMT entry.

③ Set can_eliminate flag in SLD.

# Step 2 : Update Likely-Stable but Non-Eliminated Load



# Step 3 : Avoid Wrong Elimination

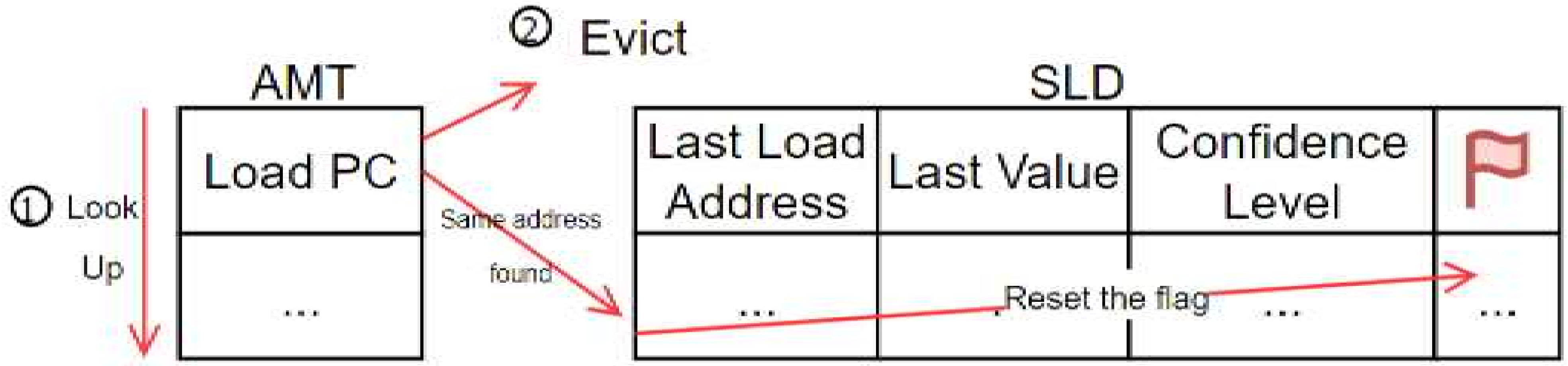


① Looks up RMT with destination register of every instruction during rename stage.

② If any load PC in the RMT entry,

1) Looks up SLD using load PC.
2) Resets the can_eliminate flag since it's going to be modified.

# Step 4 : Handling Store Instruction

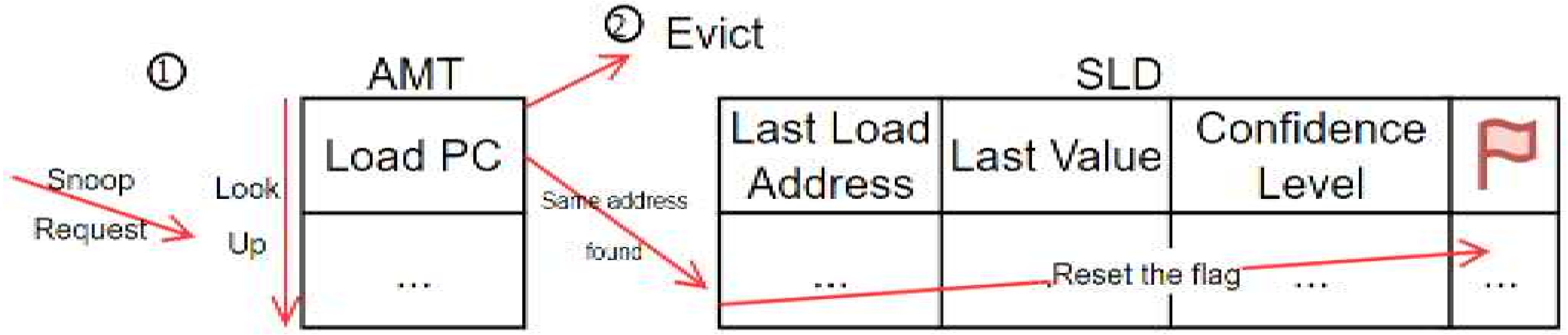


① Looks up AMT with physical address of store instruction and if the same address exists,

1) Go SLD.
2) Reset can_eliminate flag in that entry.

② Evicts the AMT entry.

# Step 5 : Handling Snoop Request



① When snoop request arrives,

1) Looks up AMT using the snoop address.
2) If the address exists, go SLD and reset can_eliminate flag.

② Evict the AMT entry as same as store instruction.

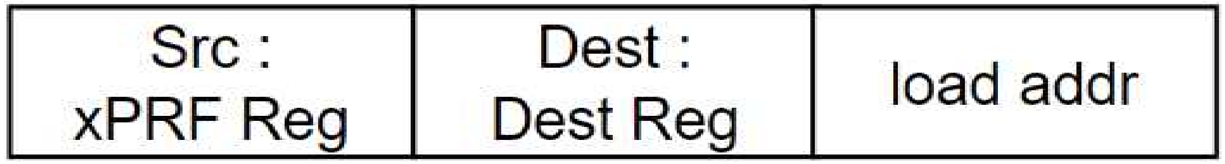
# Considerations & Issues

- In-flight load dependents
- In-flight stores
- Cache coherence in multi-core
- SLD Design
- Wrong path execution
- Changes in physical address mapping

## In-flight Load Dependent Issue

- Constable needs to supply the load value to all dependent in-flight instructions.
- Thus small extra register file called xPRF can be used.
- xPRF holds the values of eliminated load instructions.

# In-flight Load Dependent Issue : Rename Stage



Converted Instruction

- Load instruction is converted into three-operand register move instruction.

- The converted instruction maps its dest reg to src xPRF reg and complete its execution.

- Therefore, dependents of converted instruction just need to read xPRF register.

## In-flight Load Dependent Issue : Allocation & Retirement Stage

- Converted instruction gets ROB entry and LB entry.
- Address field in LB entry gets updated the last-computed load address, the third operand.
- This is required for correctly disambiguate the eliminated load from the in-flight stores.
- Converted instruction has already been completed before, so it can directly retire.

## In-flight Store Issue

- There may be eliminated loads that are younger than the store instruction whose addresses match with the store address.
- Their elimination might not correct.
- Constable solve this issue with existing memory disambiguation logic.
- If this issue happens, flushes the pipeline and re-execute all the younger instructions.

# Cache Coherence Issue

- What is the problem of snoop requests?

1. Loss of elimination opportunity due to clean eviction at the core-private cache.

2. Tracking snoop requests at cacheline address granularity.

## Cache Coherence Issue : Evicted Cacheline

- Load cannot be eliminated if it accesses the evicted cacheline whose core-valid bit was reset.

- Drawbacks
  1. If the evicted cacheline is clean, Constable loses elimination opportunity.
  2. Design complexity : Constable needs to look up and invalidate AMT entry for every cacheline eviction.

## Cache Coherence Issue : Evicted Cacheline

- Constable pins the CV-bit of an eliminated load's cacheline.

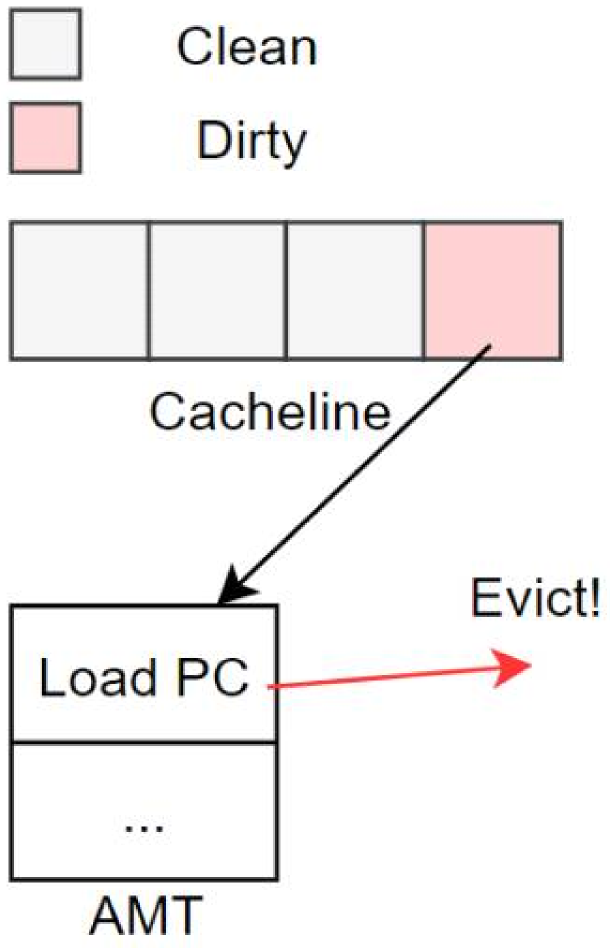
- Pinning CV-bit ensures that
  1. Even if the cacheline gets evicted from core-private cache, snoop request gets delivered.
  2. Constable does not need to look up AMT on every core-private cache eviction.

## Cache Coherence Issue : Address Granularity

- Snoop request contains a cacheline address, not full memory address.
- Constable indexes AMT using physical address at cacheline granularity.
- This might incur false address collisions.

## How could the collision occur?

- One cacheline might have multiple data.
- A data in cacheline was modified by a store instruction.
- Other data are not modified, but are regarded as modified.



## Cache Coherence Issue : Address Granularity

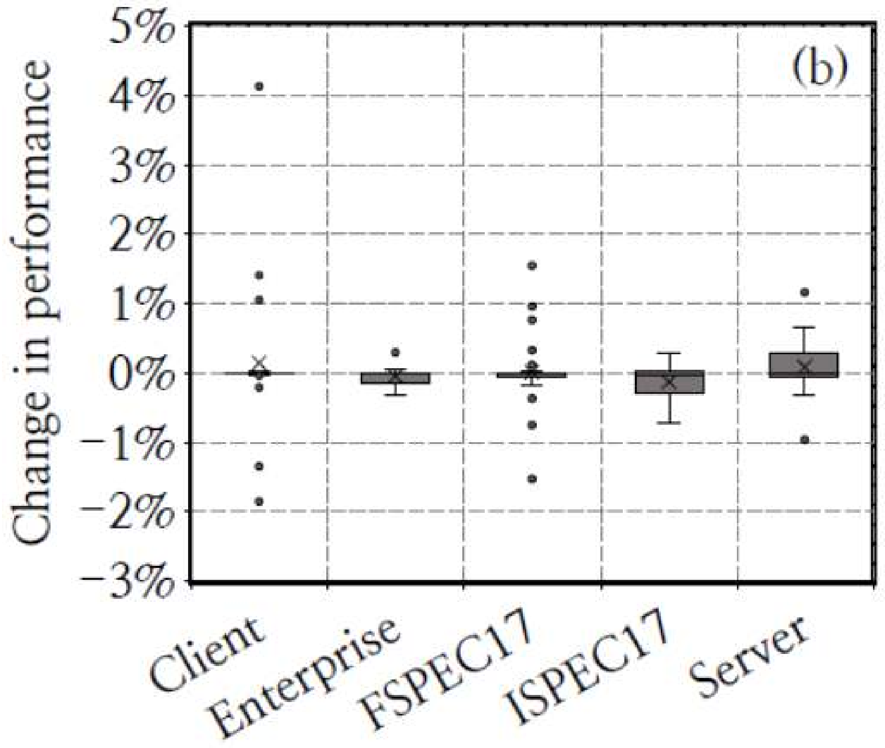
- However, this problem is negligible.
- Constable with cacheline-address-indexed AMT has only 0.4% fewer performance than Constable with fully-address-indexed AMT.
- Why? Compiler tends to lay out likely-stable load instructions together.

## Designing SLD

- Every cycle, SLD needs to support enough reads and writes in the rename stage.
- 98.3% of all workload have less than or equal to two loads on each cycle -> Three read ports.
- 98.23% of all cycles on average across all workloads have two or fewer SLD updates -> Two write ports.
- If there are more than three reads or two writes, then stall the rename stage.

# Wrong Path Execution

- Constable's structures may get updated by wrong branch prediction.
- This could be restored by branch misprediction recovery.
- But, researchers observed that it has negligible performance degradation.
- Thus, Constable doesn't have its own recovery  mechansim for branch misprediction



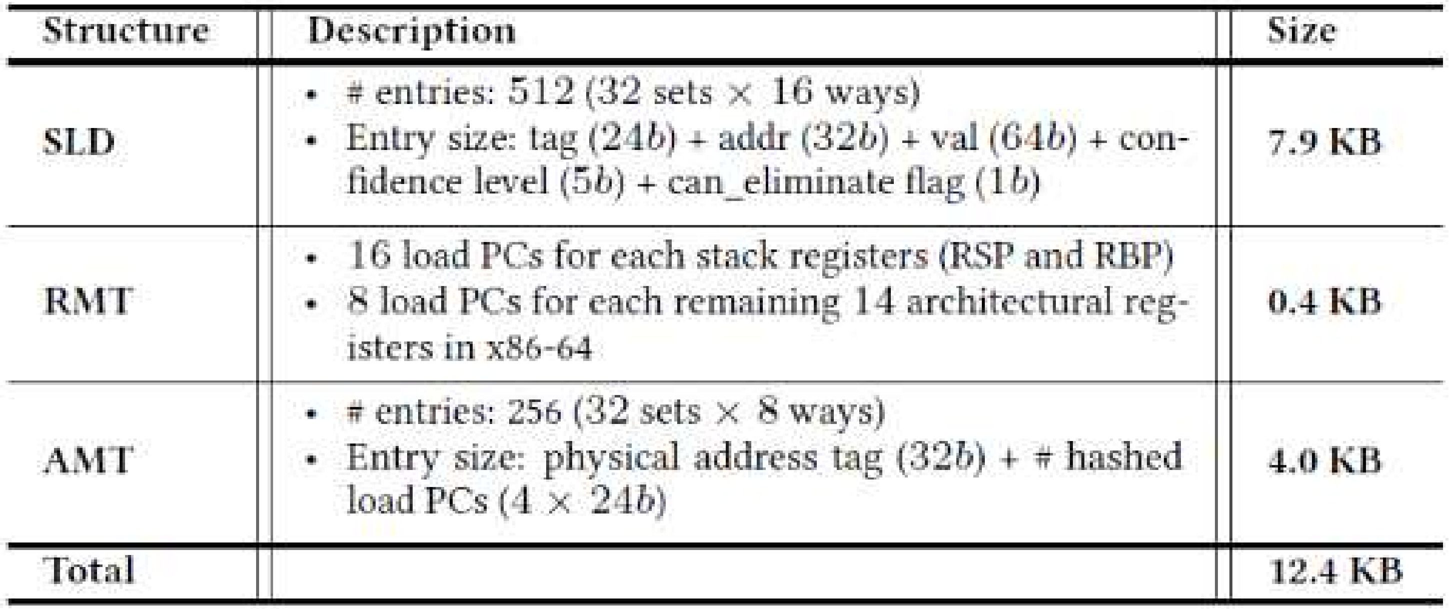
Performance evaluation between correct path instruction vs wrong path execution without recovery

## Changes in Physical Address Mapping

- Physical memory mapping change -> AMT entry is no more valid for load elimination.

- So Contsable resets all can_eliminate flag and invalidates all RMT and AMT entries when the physical memory mapping changes.

## Storage Overhead



| Structure | Description | Size |
|---|---|---|
| **SLD** | • # entries: 512 (32 sets × 16 ways)<br>• Entry size: tag ($24b$) + addr ($32b$) + val ($64b$) + confidence level ($5b$) + can_eliminate flag ($1b$) | **7.9 KB** |
| **RMT** | • 16 load PCs for each stack registers (RSP and RBP)<br>• 8 load PCs for each remaining 14 architectural registers in x86-64 | **0.4 KB** |
| **AMT** | • # entries: 256 (32 sets × 8 ways)<br>• Entry size: physical address tag ($32b$) + # hashed load PCs ($4 \times 24b$) | **4.0 KB** |
| **Total** | | **12.4 KB** |

- Main hardware: SLD, RMT, AMT.
- These requires only 12.4 KB storage per core.

## Constable vs Prior Works

- Early execute address computation
- Eliminates data fetch operation
- Memoization
- Dynamic instruction optimization

## Early Execute Address Computation

- Some prior works early of fast excute address computation of a load instruction.
- These have to execute the load instruction to verify the speculation.
- Constable safely eliminates both the address computation and the data fetch operations altogether.

## Eliminates Data Fetch Operation

- LLVP observe some static load instructions that have highly predictable load values (constant loads).
- LLVP bypasses data fetch operations of constant loads.
- But this eliminates only the data fetch operations except for address computation.

## Memoization

- Memoization caches computed results from repeated code executions.
- This enables skipping redundant operations.
- Or use PC-indexed reuse buffer to store the results of dynamic instances of every static instruction.

# Memoization vs Constable

- Storage overhead
  - Memoization requires large memoization buffer since it targets every static instruction.
  - Constable only targets loads.

- Resource dependence
  - Until the source register values are availabe, retrieval of memoized instruction output may be delayed.
  - Constable eliminate load instructions early in the pipeline.

- Multi-core Issues
  - Needs to keep coherent across memoization buffer.
  - Maintains program correctness in the out-of-order load issue.
  - Constable addresses all these problems.

## Dynamic Instruction Optimization

- Enables wide range of runtime code optimizations that removes redundant instructions.
- These learn optimizations offline per-trace basis, but Constable lerans online basis and directly to the dynamic instruction stream.
- These do not eliminate load instruction in multi-core system to maintain the coherency.

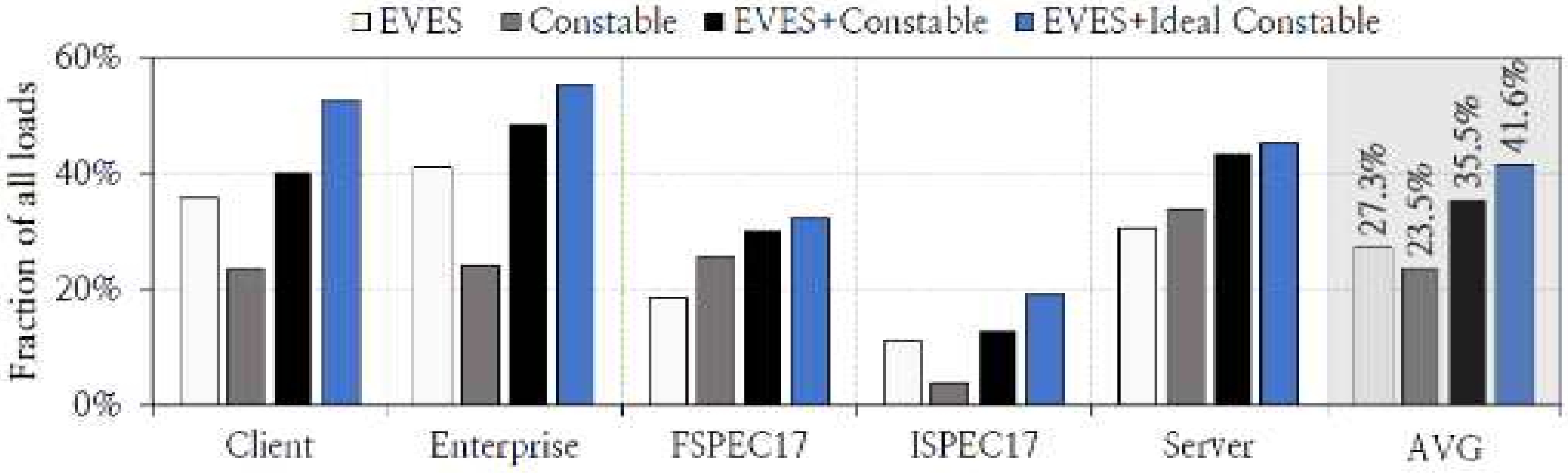
## Performance Evaluation

- noSMT and SMT Configuration
- Performance comparison with prior works
- How many load instructions could be eliminated?
- Impact on pipeline resource utilization
- Power improvement

# Methodology

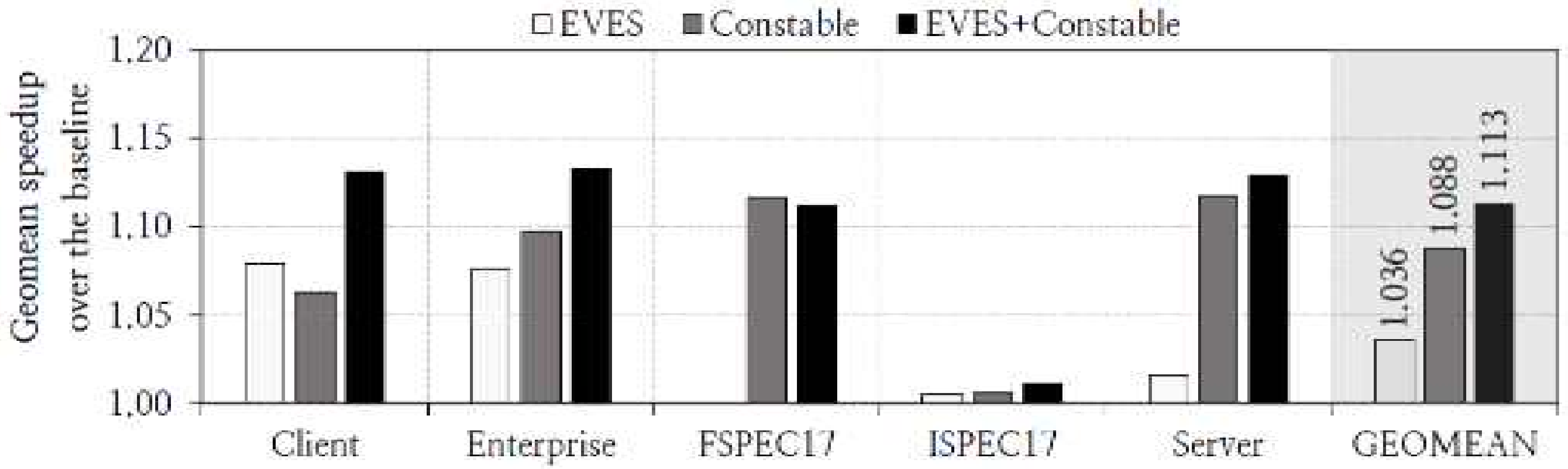
- Integrated simulator modeling 6-wide issue Out-of-Order processor.
- 90 workloads of wide variety.
- Configuration
  - noSMT
  - 2-way SMT
- Evaluation with three prior works
  - EVES : load value predictor
  - ELAR : early load address resolution
  - RFP : register file prefetching
- Golden check
  - Verify the correctness of Constable.
  - Checks load address and load data.

# Performance Improvement in noSMT



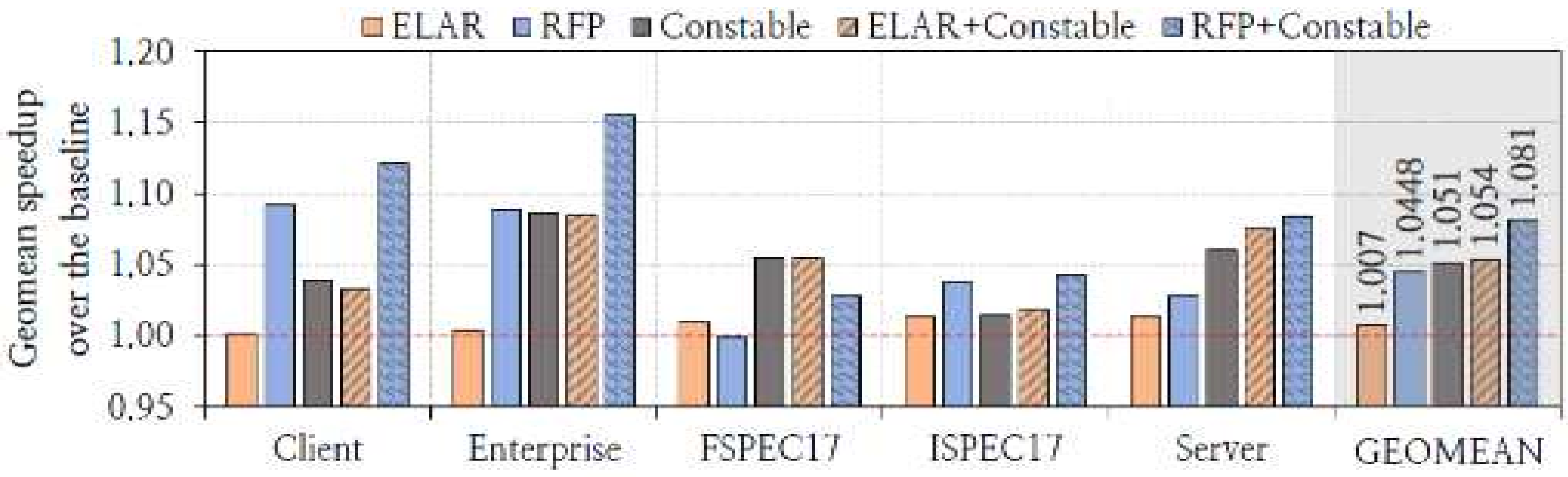
- Constable alone provides similar performance as EVES but only 1/2 of EVES' storage overhead.
- Constable combined with EVES consistently outperforms both EVES and Constable alone in every workload.

## Performance Improvement in 2-way SMT



- Constable provides better performance than non-SMT.
- Because of high resource contention in SMT.

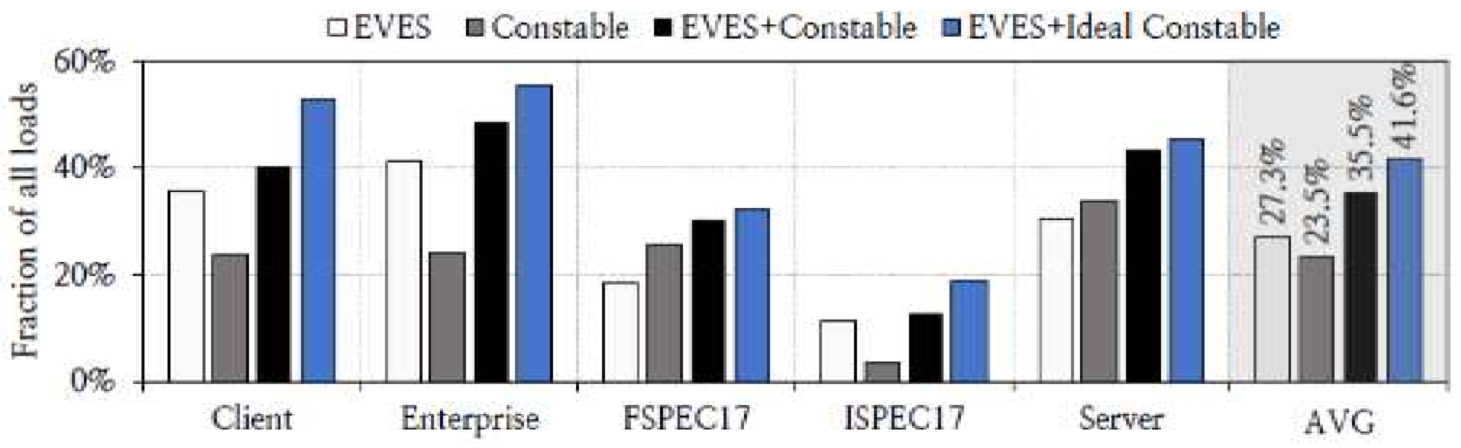
## Performance Comparison with Prior Works



- Constable alone outperforms both ELAR and RFP.

- When combined with ELAR and RFP, Constable provides more performance benefit.

# Load Coverage

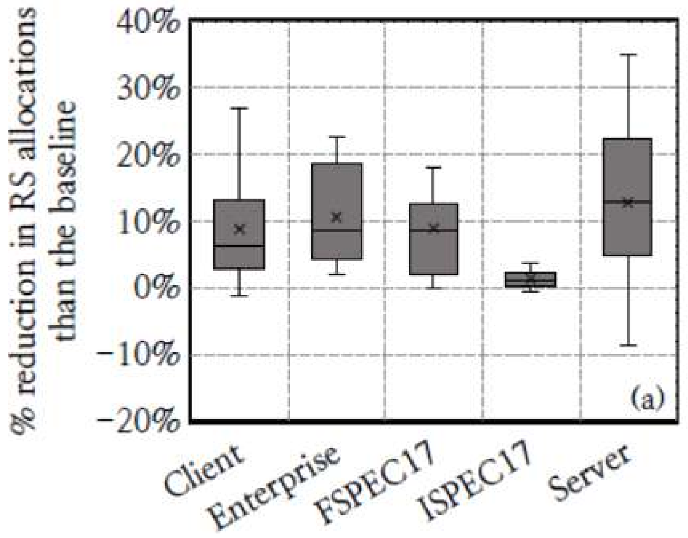


- Constable alone covers 23.5%, EVES does 27.3% since Constable targets loads which have both value and address locality, whereas EVES just targets only shows value locality.

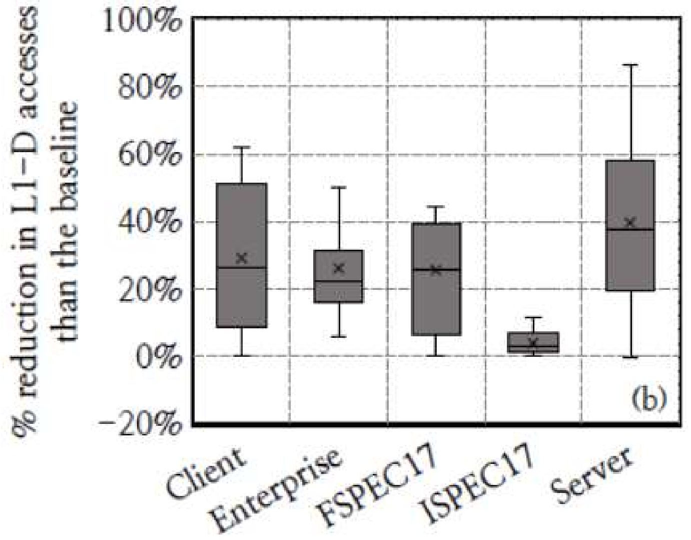
- Constable covers significant fraction of the load by itself and combined with EVES.

# Impact on Pipeline Resource Utilization

- Reduction in RS Allocation.
- Constable reduces RS allocation by 8.8% on average.



- Reduction in L1-D Access.
- Constable reduces L1-D allocation by 26.0% on average.



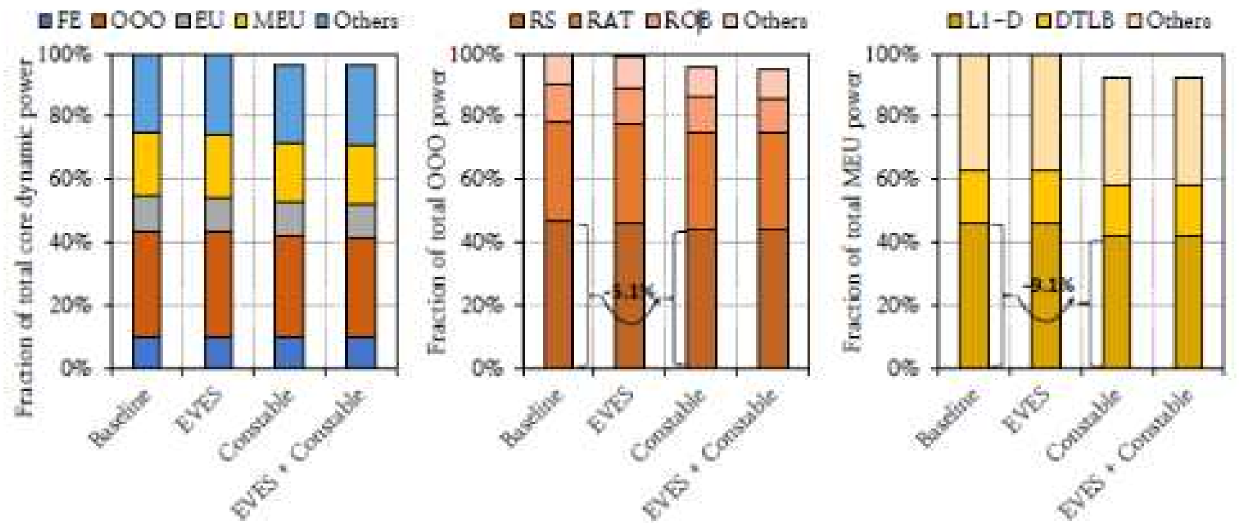
## Power Improvement

- Key units for power consumption of core
  - Front end
  - Out of order
  - Non-memory execution unit
  - Memory execution unit

- Key units for Out of Order
  - Reservation station
  - Register alias table
  - Re-order buffer

- Key units for memory execution unit
  - L1-D cache
  - Data translation look-aside buffer

# Power Improvement



- Constable reduces core power consumption by 3.4% on average over the baseline but EVES does only 0.2%.

- Constable reduces power consumed by OOO unit by 4.5% on average over the baseline because Constable significantly reduces RS allocations.

- Constable reduces power consumed by MEU unit by 7.2% on average over the baseline because MEU power reduction is dominated by L1-D cache. Constable reduces L1-D accesses.

## Summary

- Global-stable loads fetch the same data from the same address, and this cause significant ILP loss due to resource dependence.
- Constable identifies and eliminates loads that repeatedly fetch same data from same address.
- Thus, Constable mitigates data and resource dependence and even reduces the power consumption.