The Impact of Negative Acknowledgments in Shared Memory Scientific Applications

Mainak Chaudhuri
School of Electrical and Computer Engineering
Cornell University

Mark Heinrich
School of EECS
University of Central Florida

CSL

Introduction

Why Negative Acknowledgments (NACKs)?

- Transactions are inherently non-atomic
  - Distributed nature of directory-based cache coherence protocols
  - A transaction may involve multiple messages taking the machine into transient unstable states
  - Need for serialization to resolve races and enforce a valid total order
- Deadlock avoidance
  - Every transaction requires certain amount of resources
  - Cannot hold and wait

Need a mechanism to delay and retry transactions:
Extra network traffic and Controller occupancy

Contributions

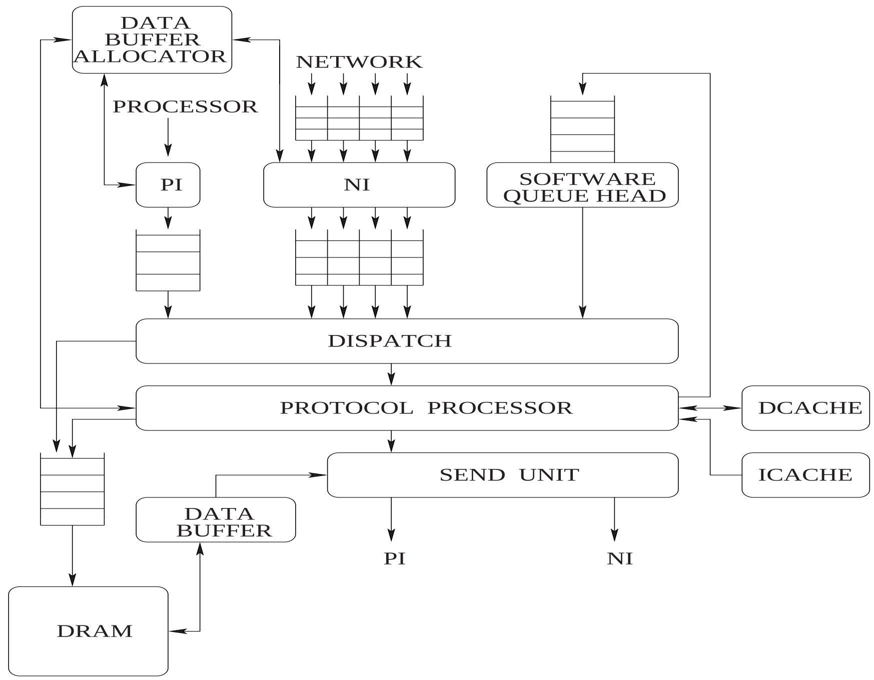
- Novel technique of request combining in the coherence protocol
- Read combining speeds up 64-node parallel execution time by
  - 6% to 93% compared to a base bitevector protocol and upto 41% compared to a modified version of Origin 2000 protocol
- An extensive quantitative analysis of NACKs on a family of previously designed as well as novel bitvector protocols

Outline

- Baseline Protocols
  - Base Bitvector
  - SGI Origin 2000
- Nack-free Protocols
  - Piranha/GS320
  - Pending Request Combining
- Evaluation
- Summary

Base Node Controller Architecture

MAGIC: Memory And General Interconnect Controller from Stanford FLASH multiprocessor [Heinrich et al, ASPLOS 1994][Kuskin et al, ISCA 1994]



Processor Interface has an Outstanding Transaction Table (OTT)

Base Bitvector Protocol

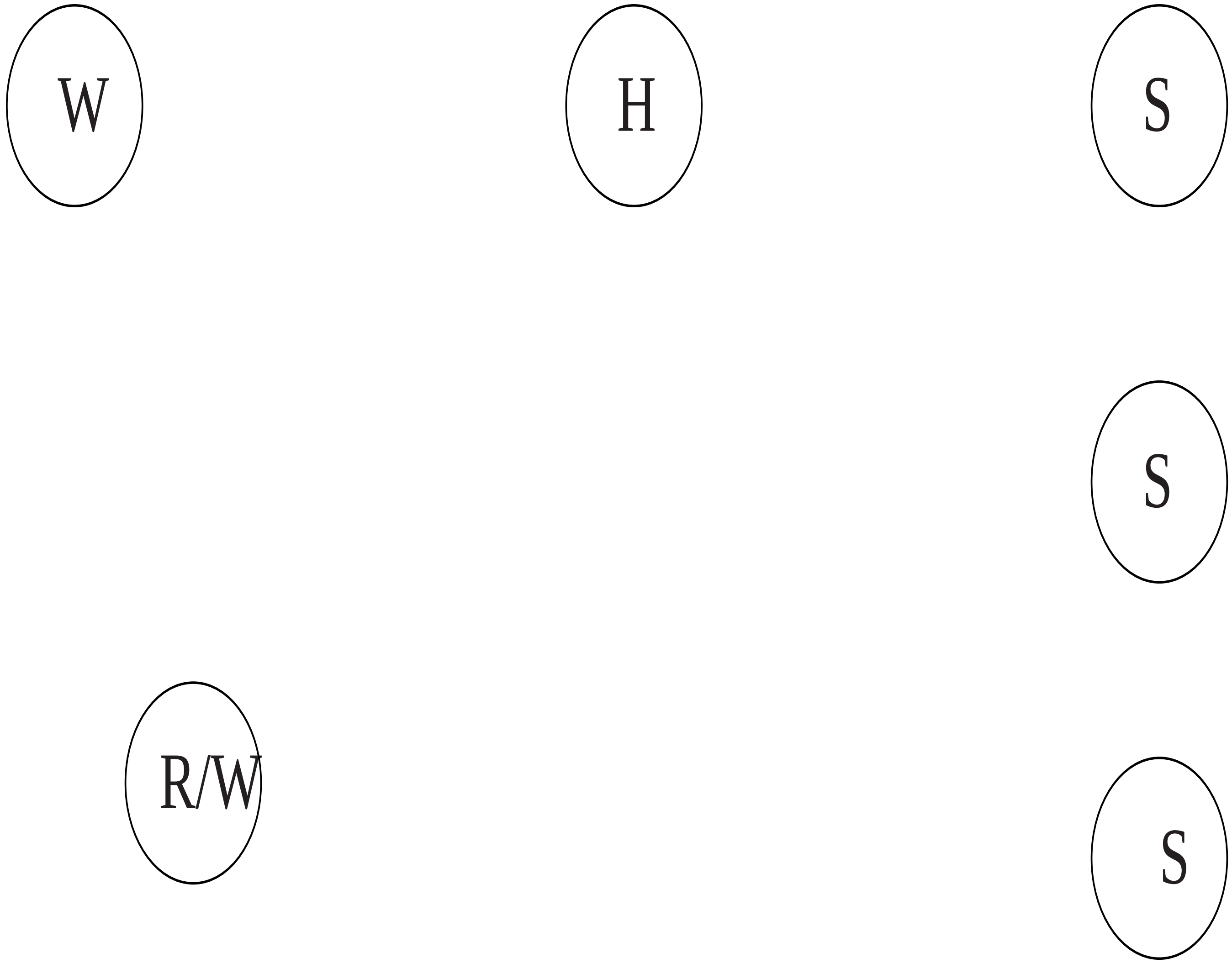
Directory Entry

- 32-bit sharer vector
- Two state bits: Pending and Dirty

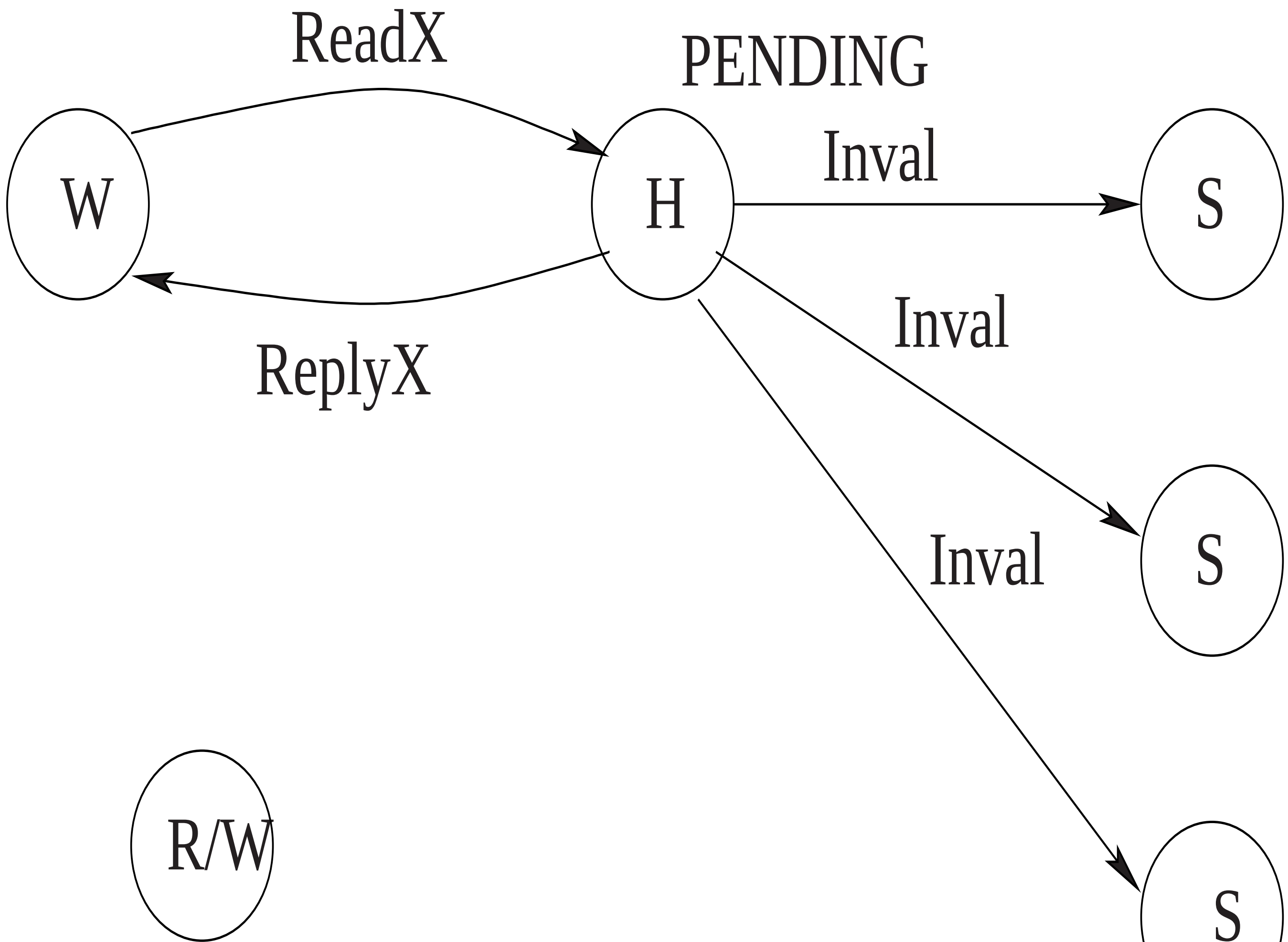
Protocol Features

- Collects Invalidation Acknowledgments at the home node
- Relaxes consistency model with eager-exclusive replies
- Generates NACKs both from the home node and the third party nodes

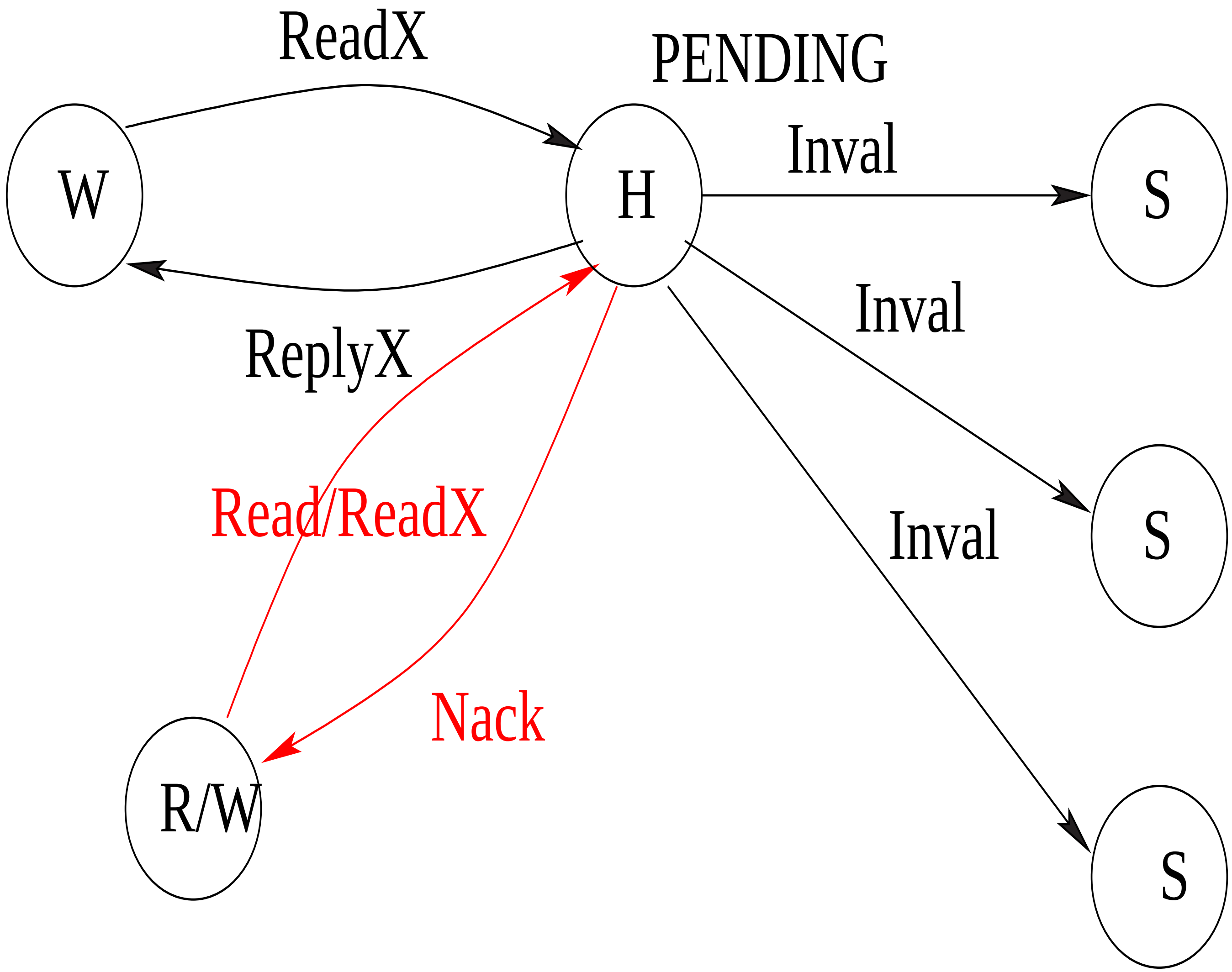
Base Bitvector: NACKs from Home (I)



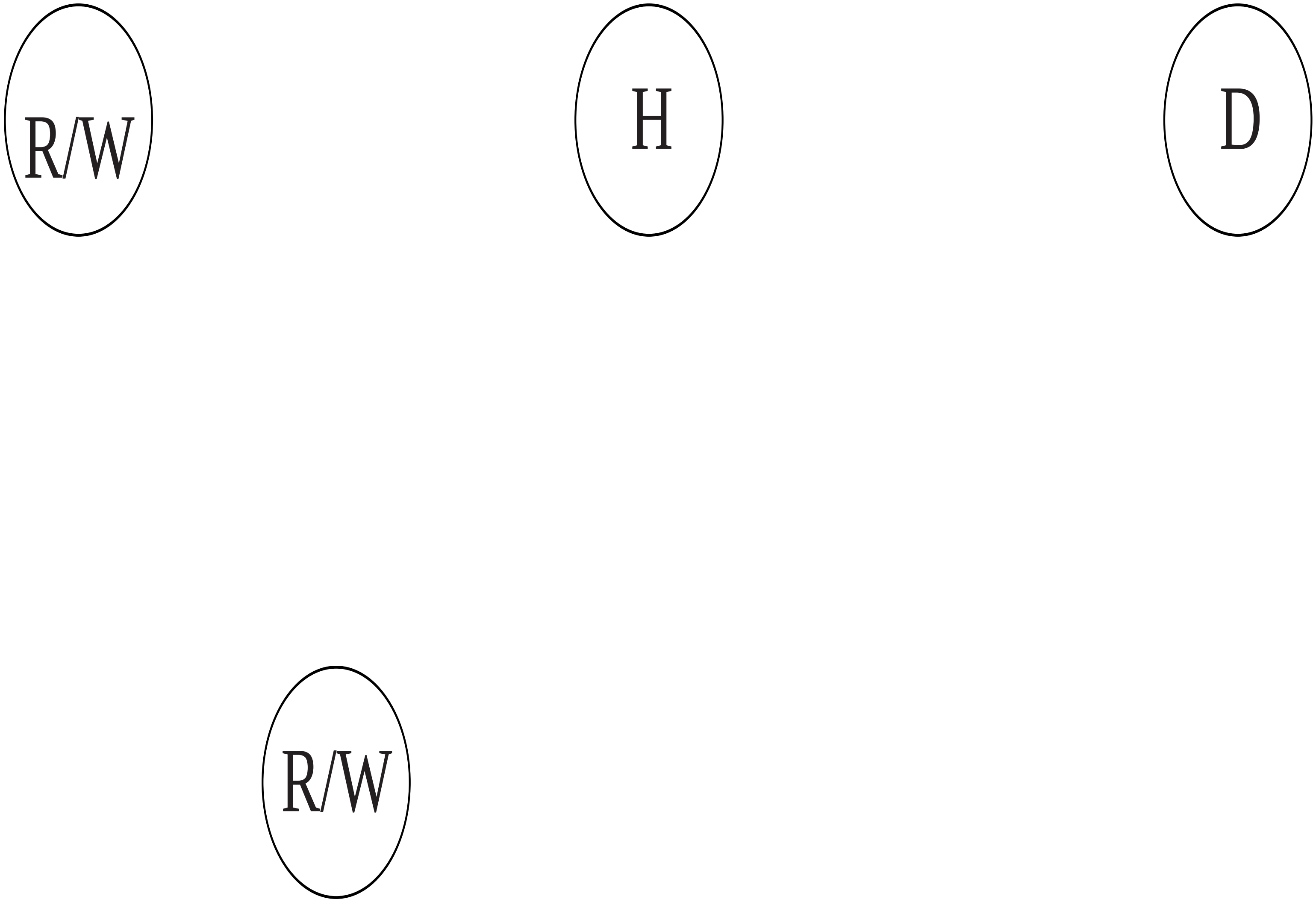
Base Bitvector: NACKs from Home (I)



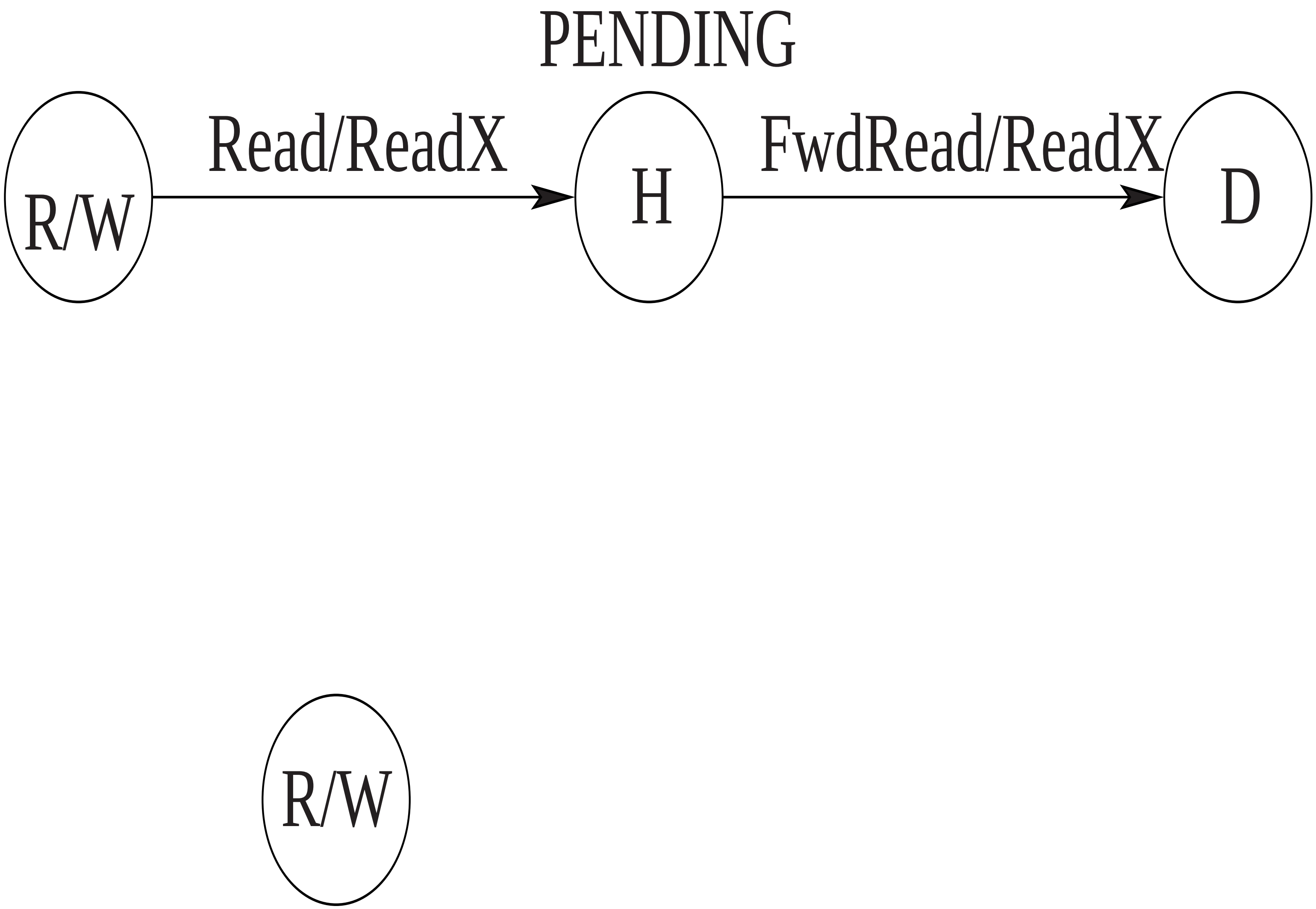
Base Bitvector: NACKs from Home (I)



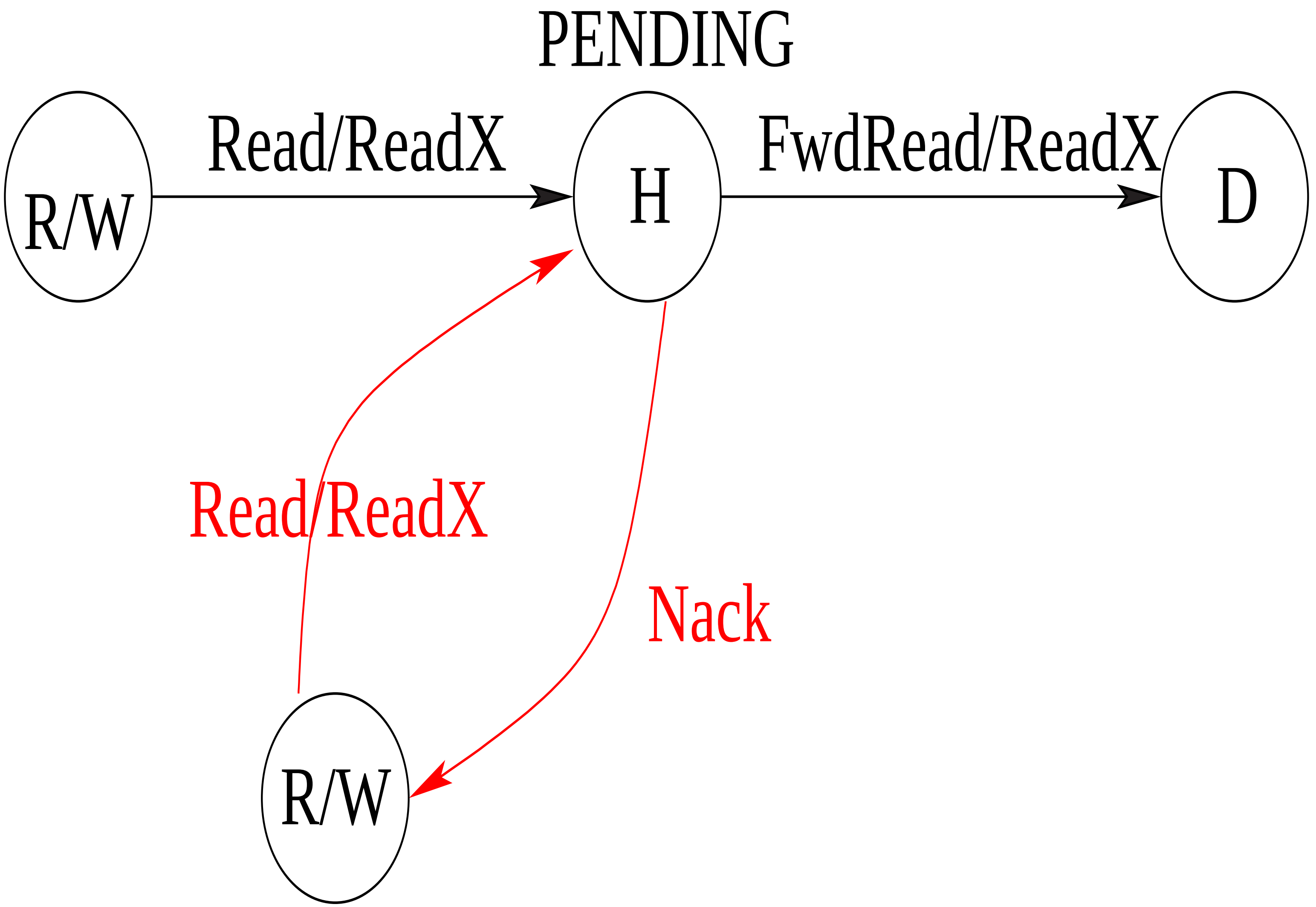
Base Bitvector: NACKs from Home (II)



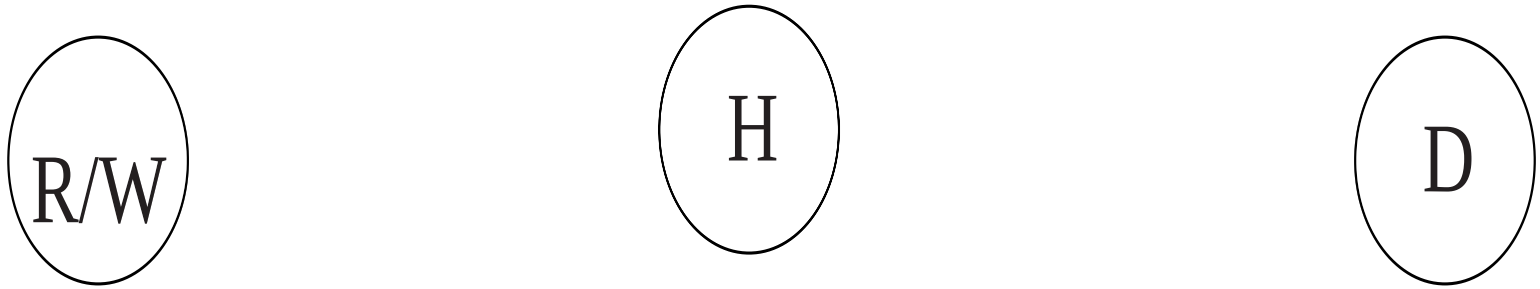
Base Bitvector: NACKs from Home (II)



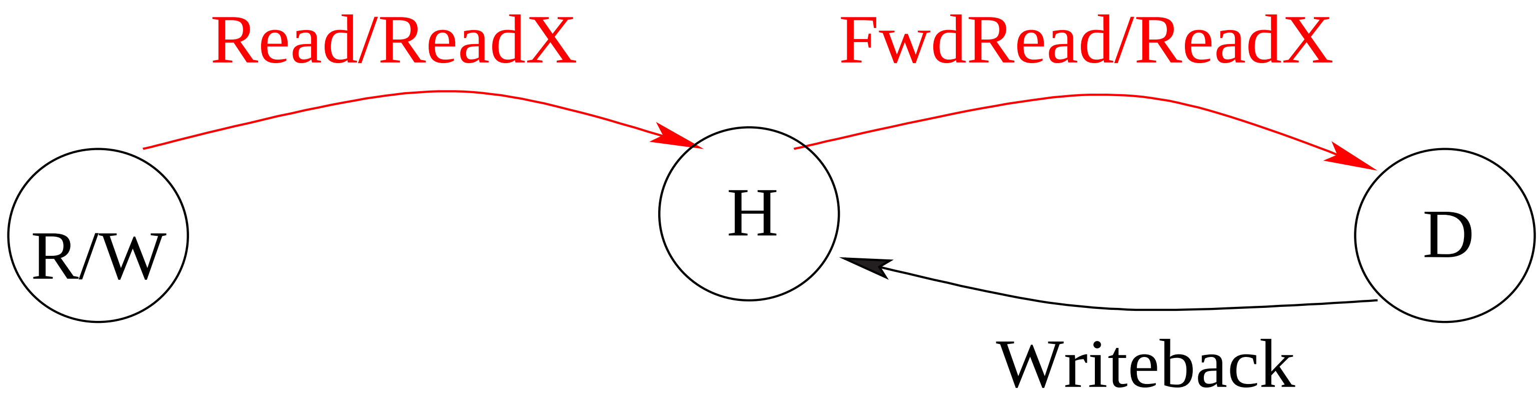
Base Bitvector: NACKs from Home (II)



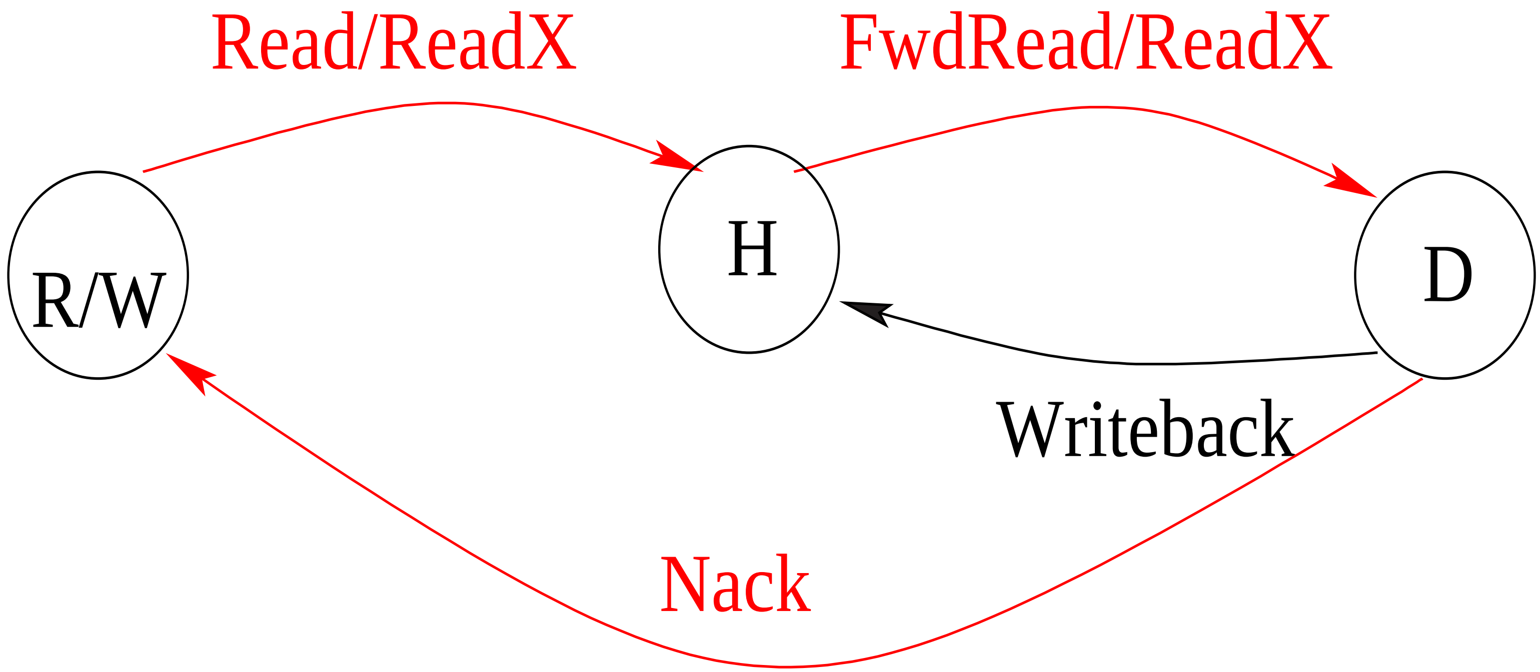
Base Bitvector: NACKs from Late Interventions



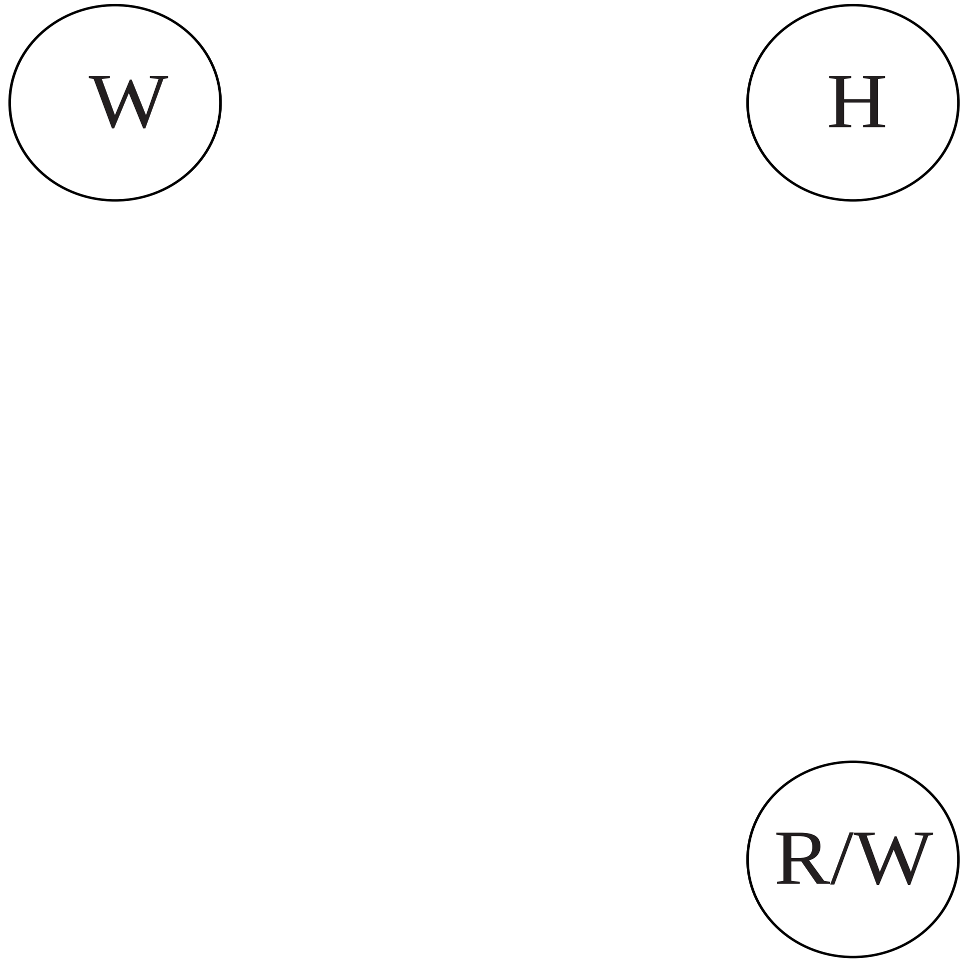
Base Bitvector: NACKs from Late Interventions



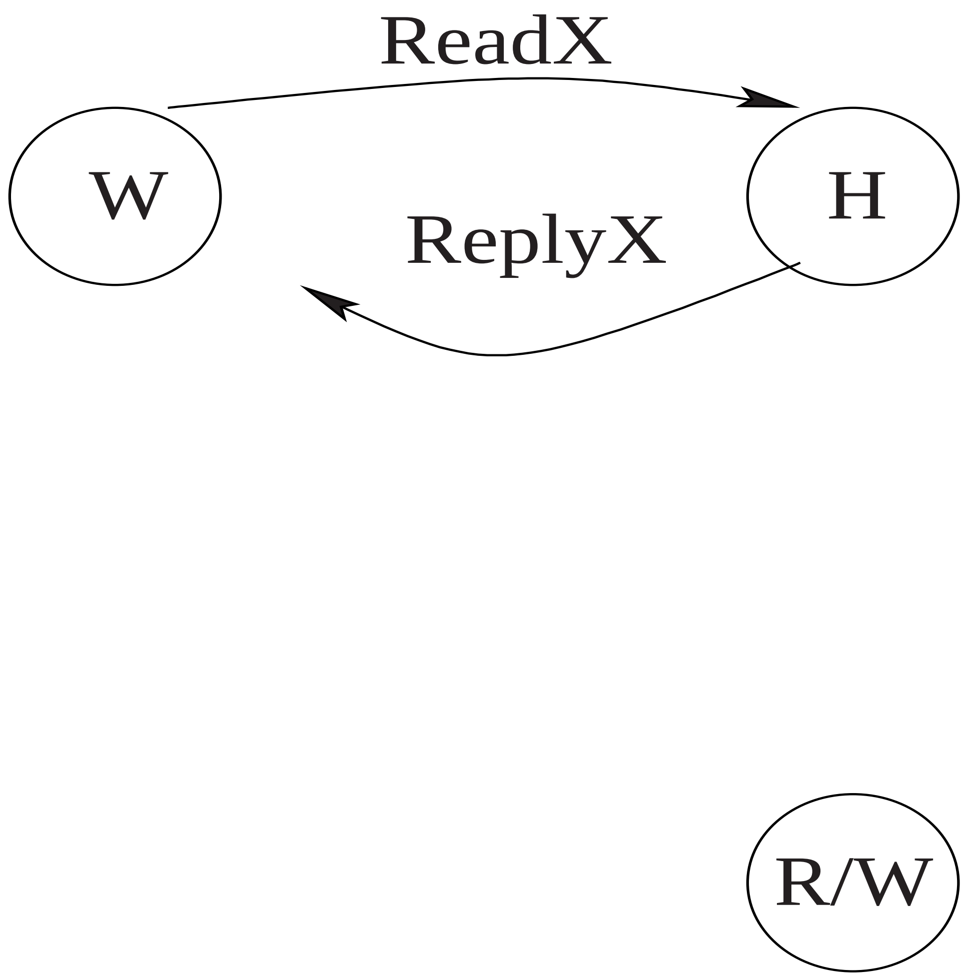
Base Bitvector: NACKs from Late Interventions



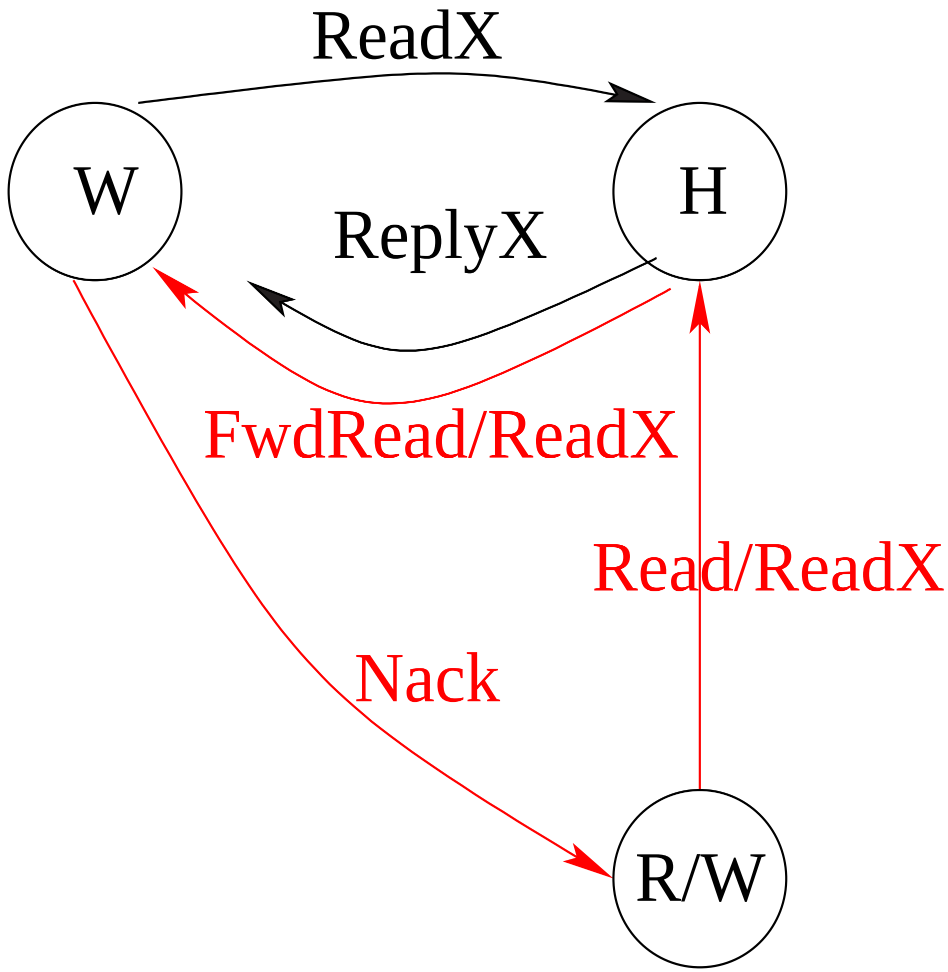
Base Bitvector: NACKs from Early Interventions



Base Bitvector: NACKs from Early Interventions



Base Bitvector: NACKs from Early Interventions



Modified SGI Origin 2000 Protocol

[Laudon and Lenoski, ISCA 1997]

Directory Entry

- 32-bit sharer vector
- Four state bits: Pending shared, Pending dirty, Dirty, Local

Protocol Features

- Collects invalidation acknowledgments at the writer
- Relaxes consistency model with eager-exclusive replies
- Generates NACKs only from the home node

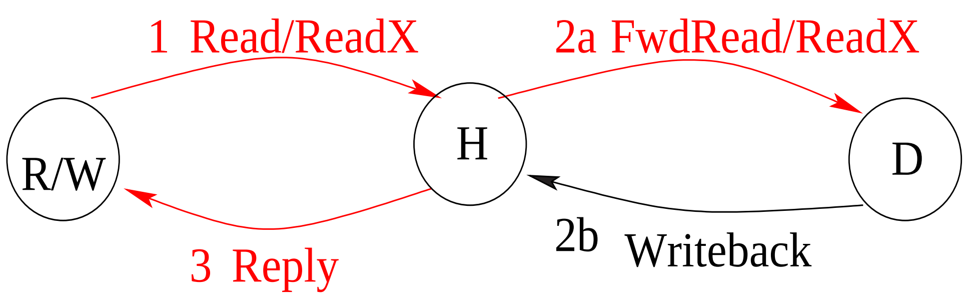
Modified SGI Origin 2000 Protocol: Nacks

Eliminating early intervention Nacks

- Buffer the intervention at the owner (in OTT) until write reply arrives

Eliminating late intervention Nacks

- Home is responsible for forwarding the writeback to the requester

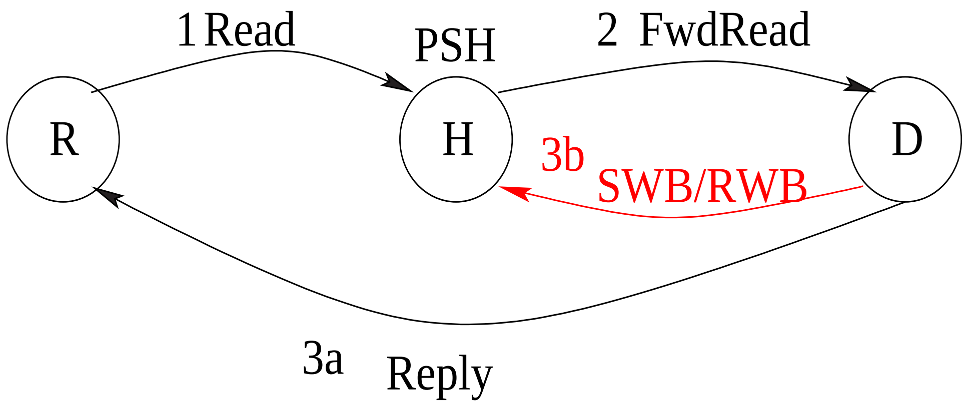


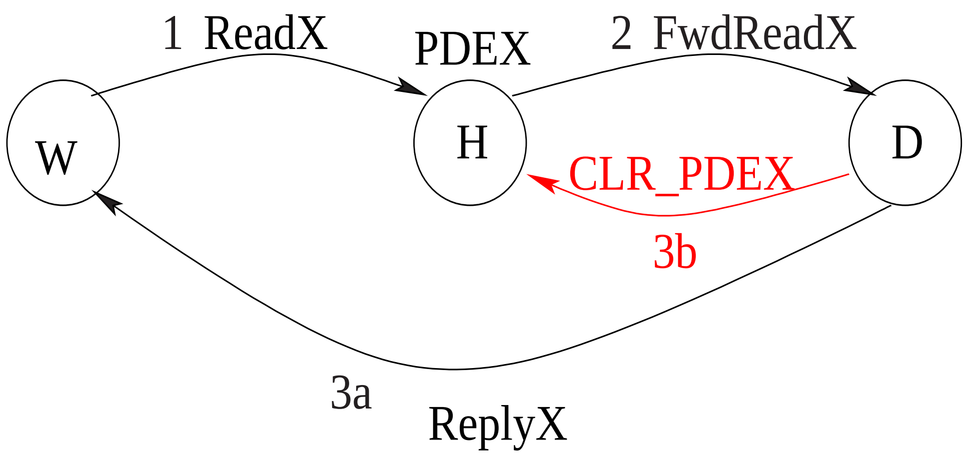
- Need mechanism to distinguish late and genuine interventions: Writeback buffer

Only home can generate Nacks if the directory entry is in one of the pending states

Eliminating the Remaining Nacks

Why do we need the pending states?





Eliminating the Remaining Nacks

Two Possibilities

- Eliminate pending states (Piranha/GS320 Protocol)
  - Accept all requests
  - Change directory entry immediately to reflect the new sharer or the owner
  - Off-load the resolution of races to the periphery (third party nodes)
- Buffer pending requests at the home node (Our Request Combining Protocol)
  - Reserve moderate buffering space in main memory
  - Any message clearing the pending state is responsible to trigger pending replies for corresponding cache lines

Piranha Inter-node Protocol

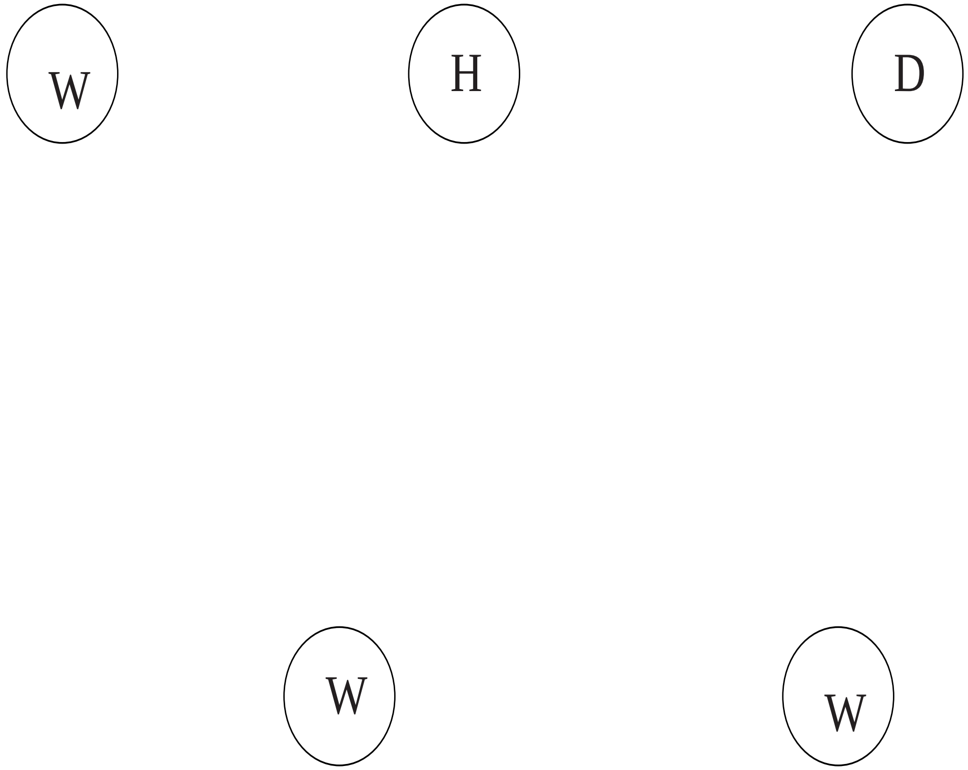
[Piranha: Barroso et al, ISCA 2000]

[GS320: Gharachorloo et al, ASPLOS 2000]

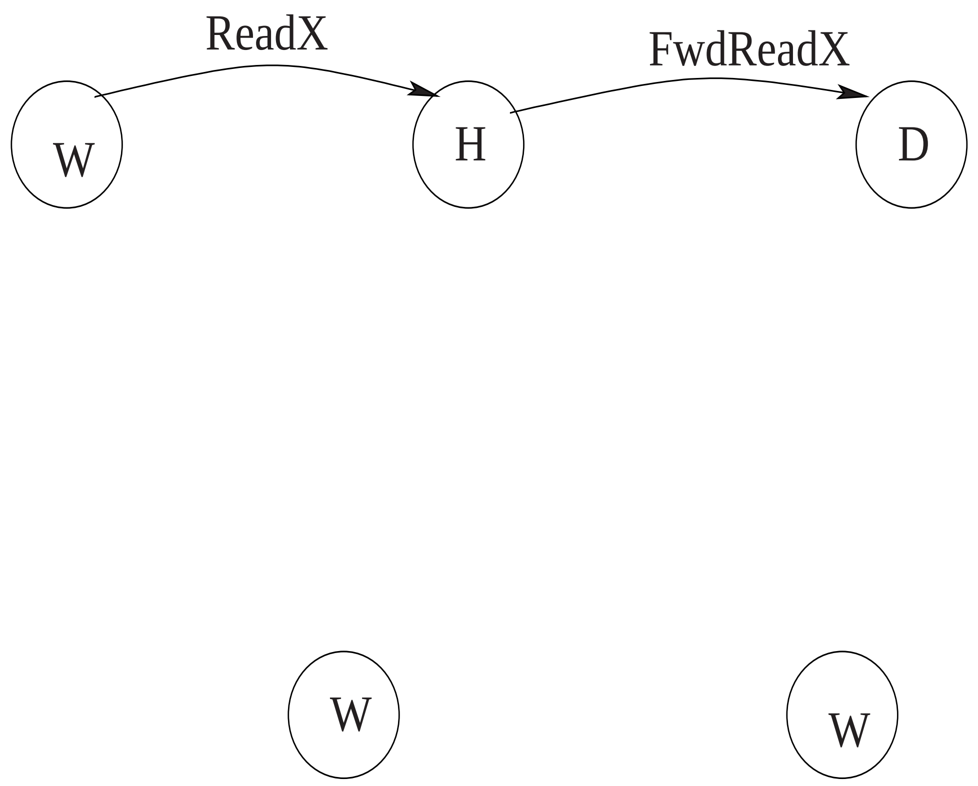
Eliminating the Pending Dirty State

- On a read exclusive or upgrade request change the owner to reflect the new owner
- Forward the intervention to the old owner if the state is dirty
- Home node expects the old owner to always be able to supply the cache line

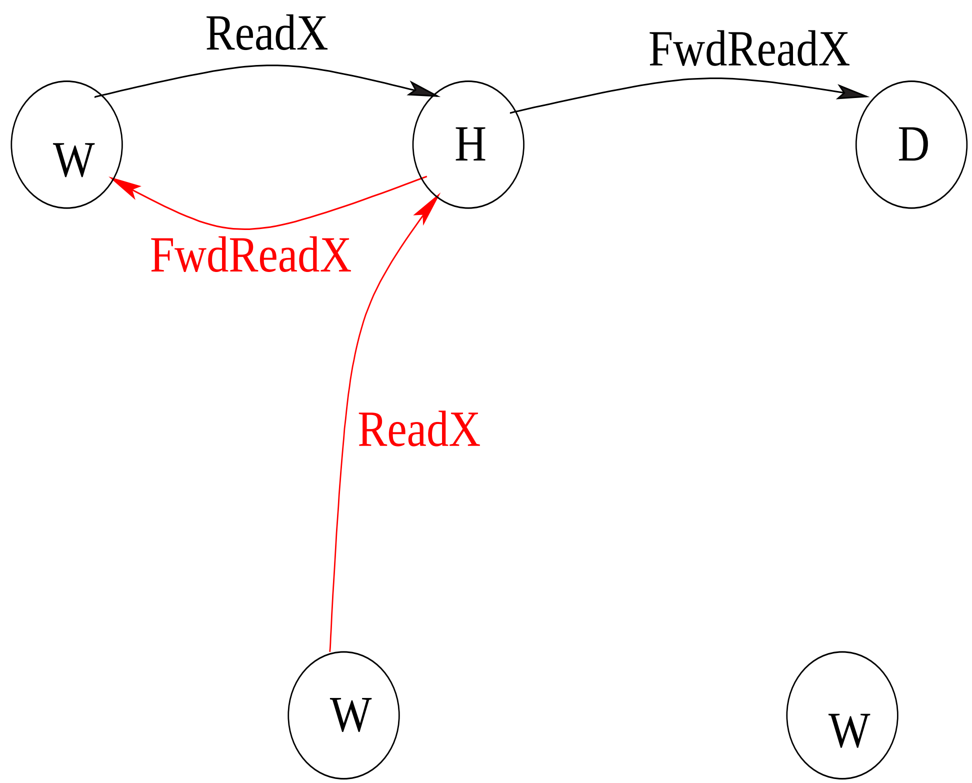
Eliminating the Pending Dirty State



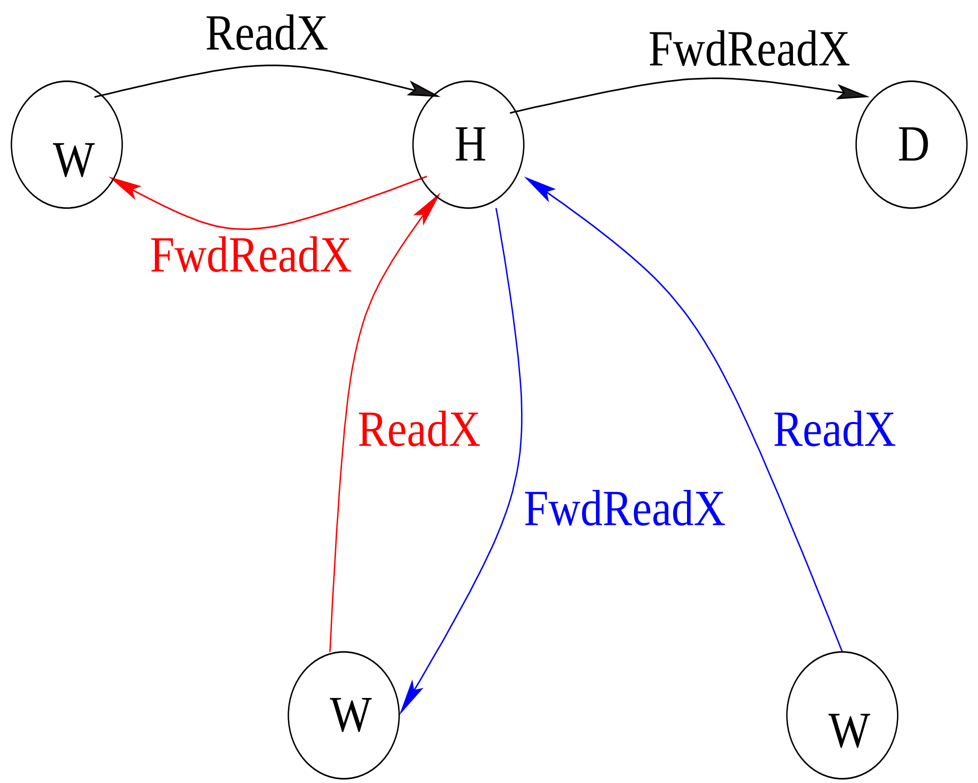
Eliminating the Pending Dirty State



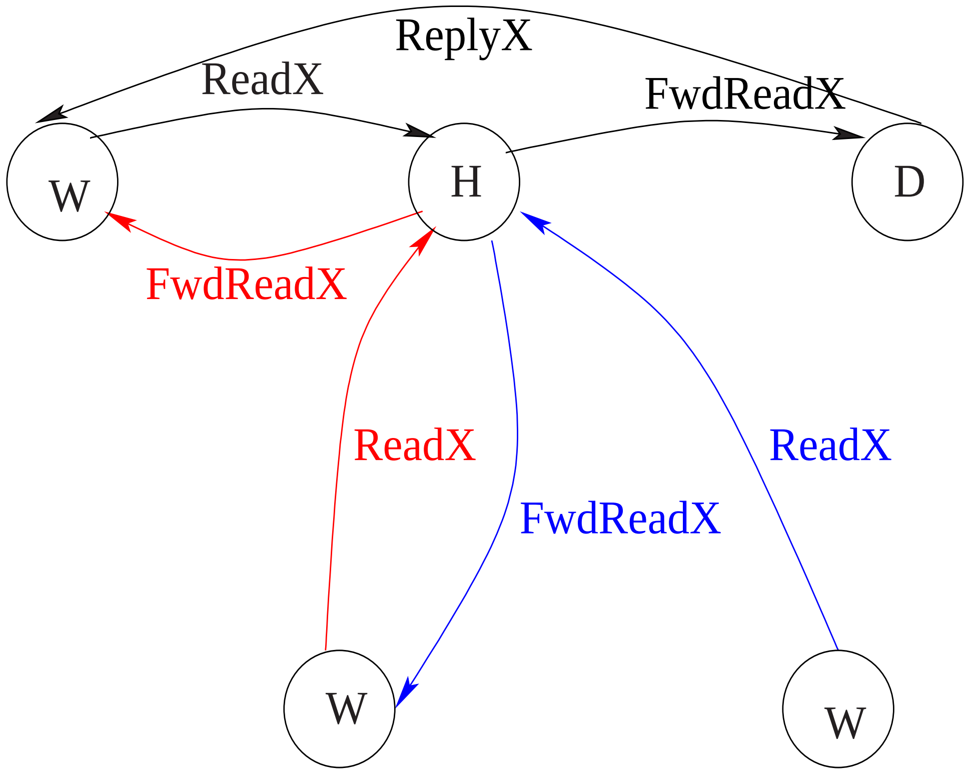
Eliminating the Pending Dirty State



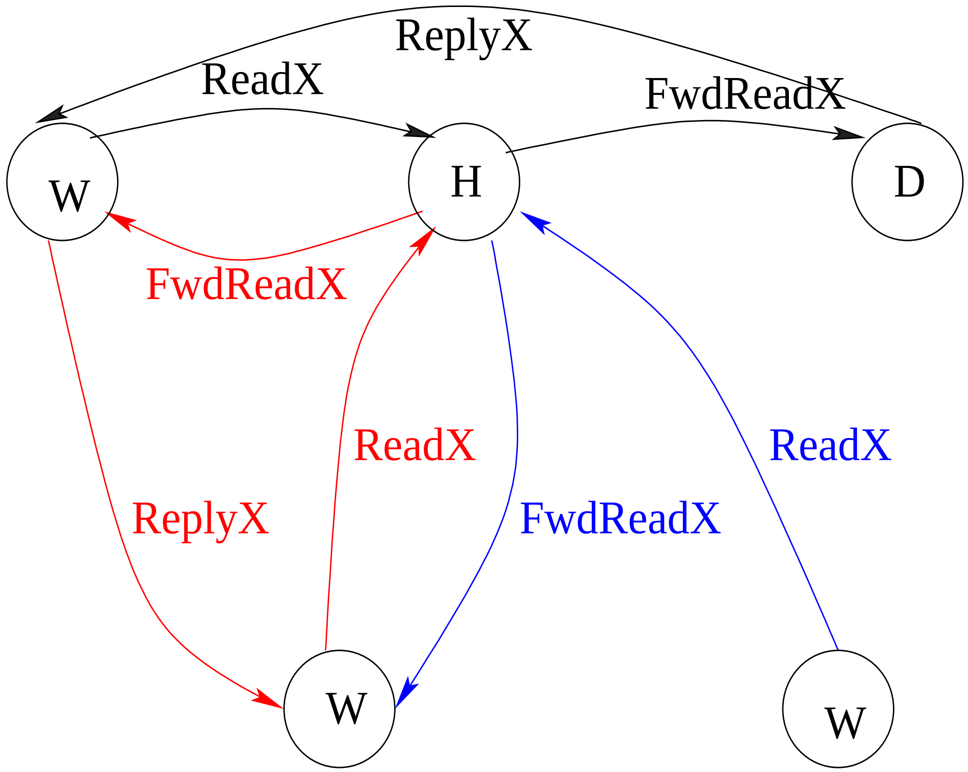
Eliminating the Pending Dirty State



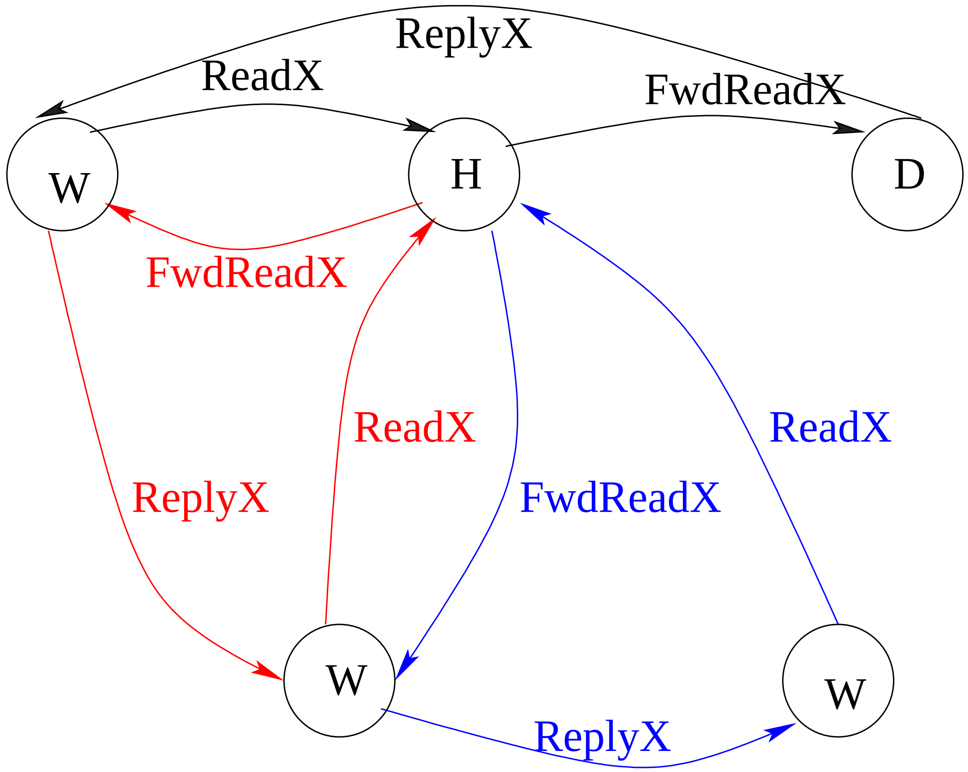
Eliminating the Pending Dirty State



Eliminating the Pending Dirty State

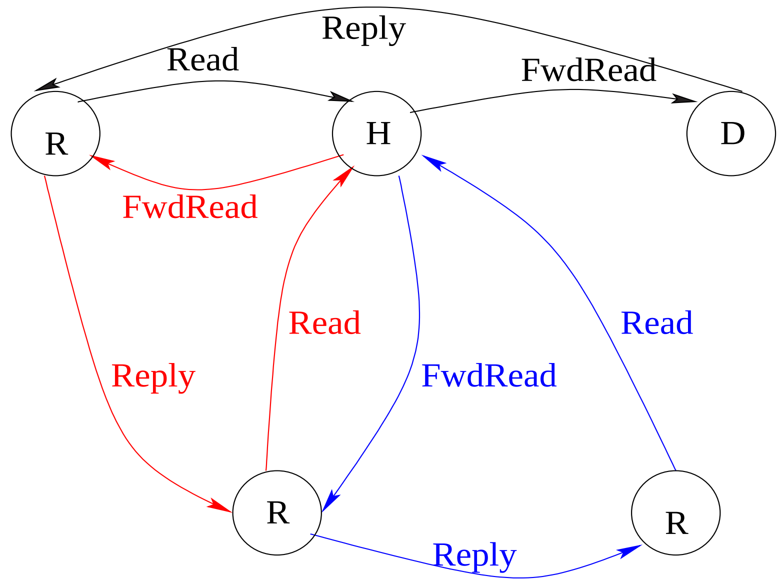


Eliminating the Pending Dirty State



- We call it Write String Forwarding (WSF)

Eliminating the Pending Shared State



- Treats forwarded read and read exclusive requests similarly
- Called Dirty Sharing (DSH)
- Many two-hop transactions are converted to three-hop transactions

Eliminating the Pending States

Changes in Node Controller

- Writeback buffer needs to hold written back data until acknowledged by home

Changes in Cache Subsystem

- Cache controller should be able to supply cache lines in shared state for intervention replies
- Cache controller needs to generate writebacks on replacing shared owned cache lines

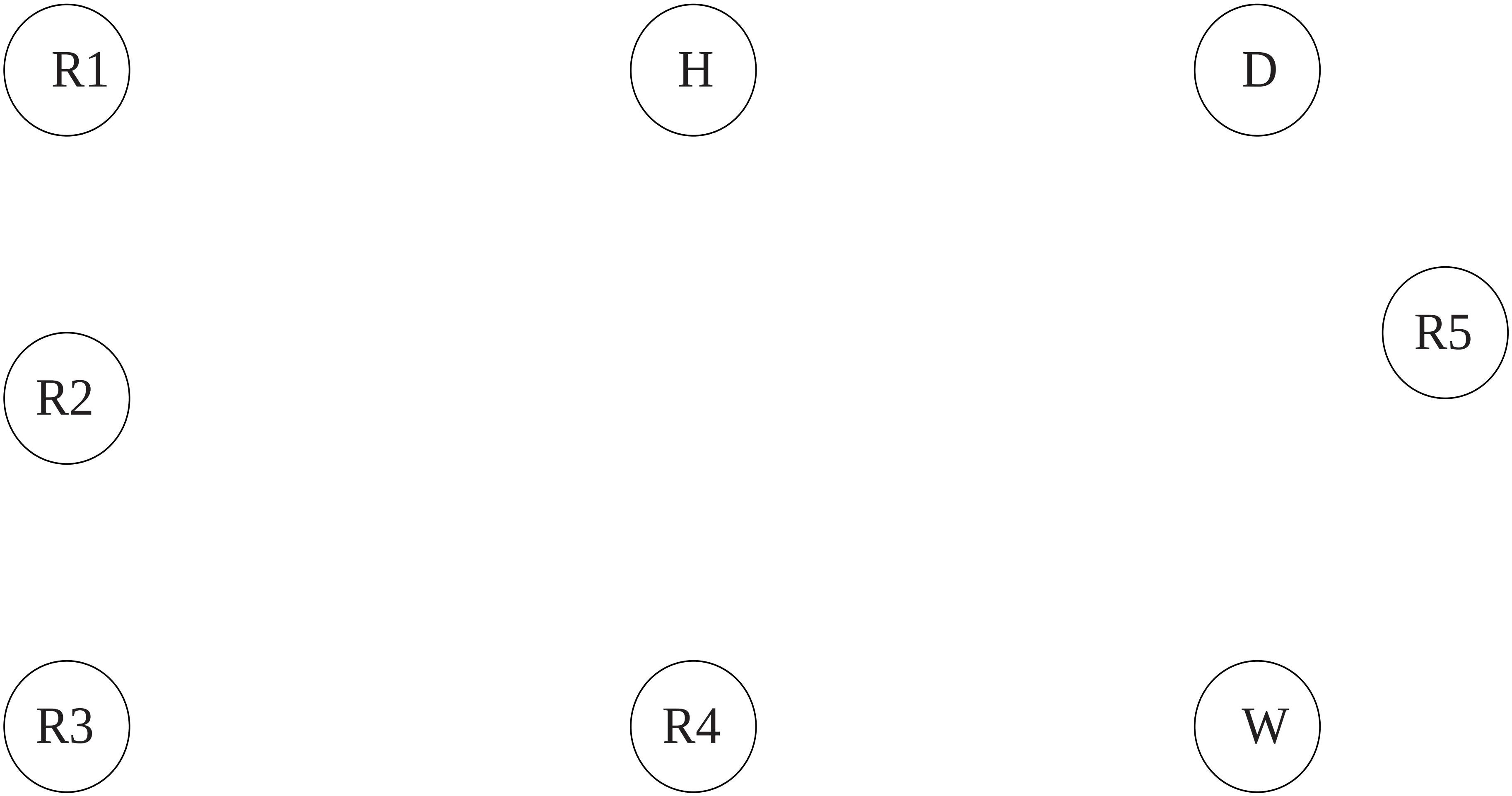
The Alternative: Buffer at the Home Node

Key observation: the order in which the home node services incoming requests does not matter

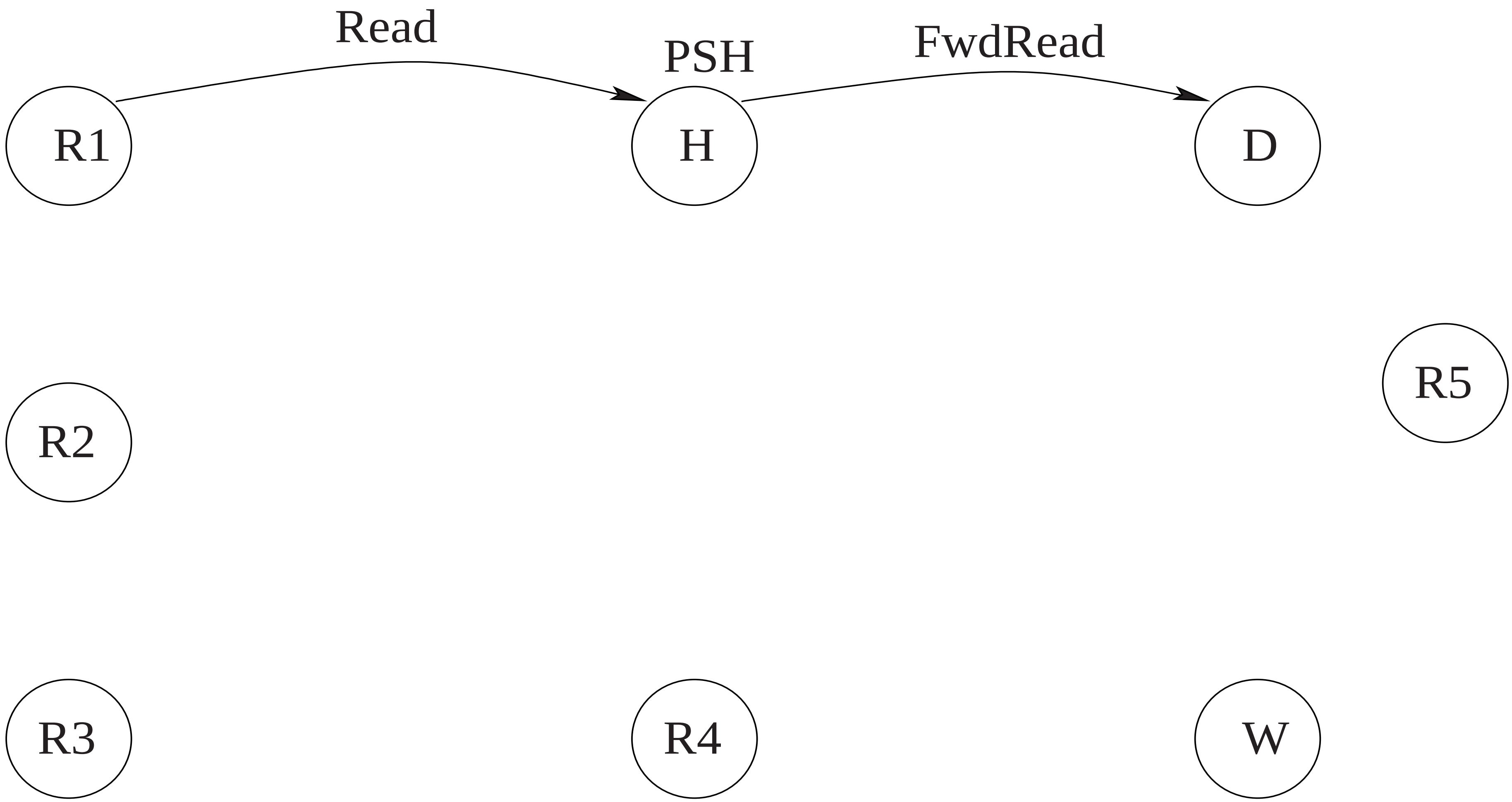
Protocol Features and Directory States

- Reserves space in main memory at boot time for two pending request lists: read and write
- Maintains states in directory entry to indicate if anything is pending for the corresponding cache line
- Maintains two entry indices (one for each list) in directory entry to indicate the start of pending request chain for the corresponding cache line
- Queues the first pending reader in the directory entry itself: favors short sharing sequences

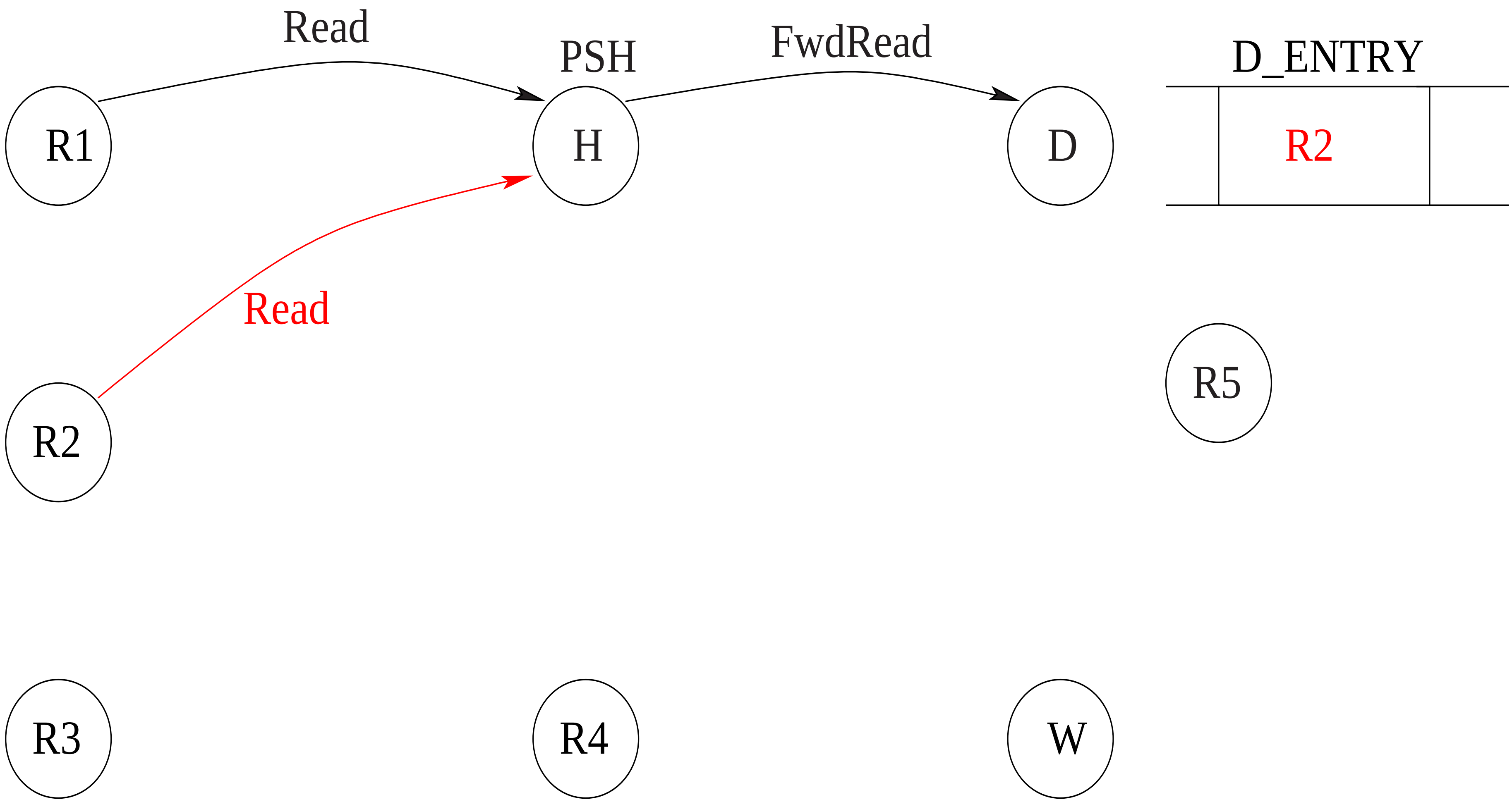
Enabling Read Combining (RComb)



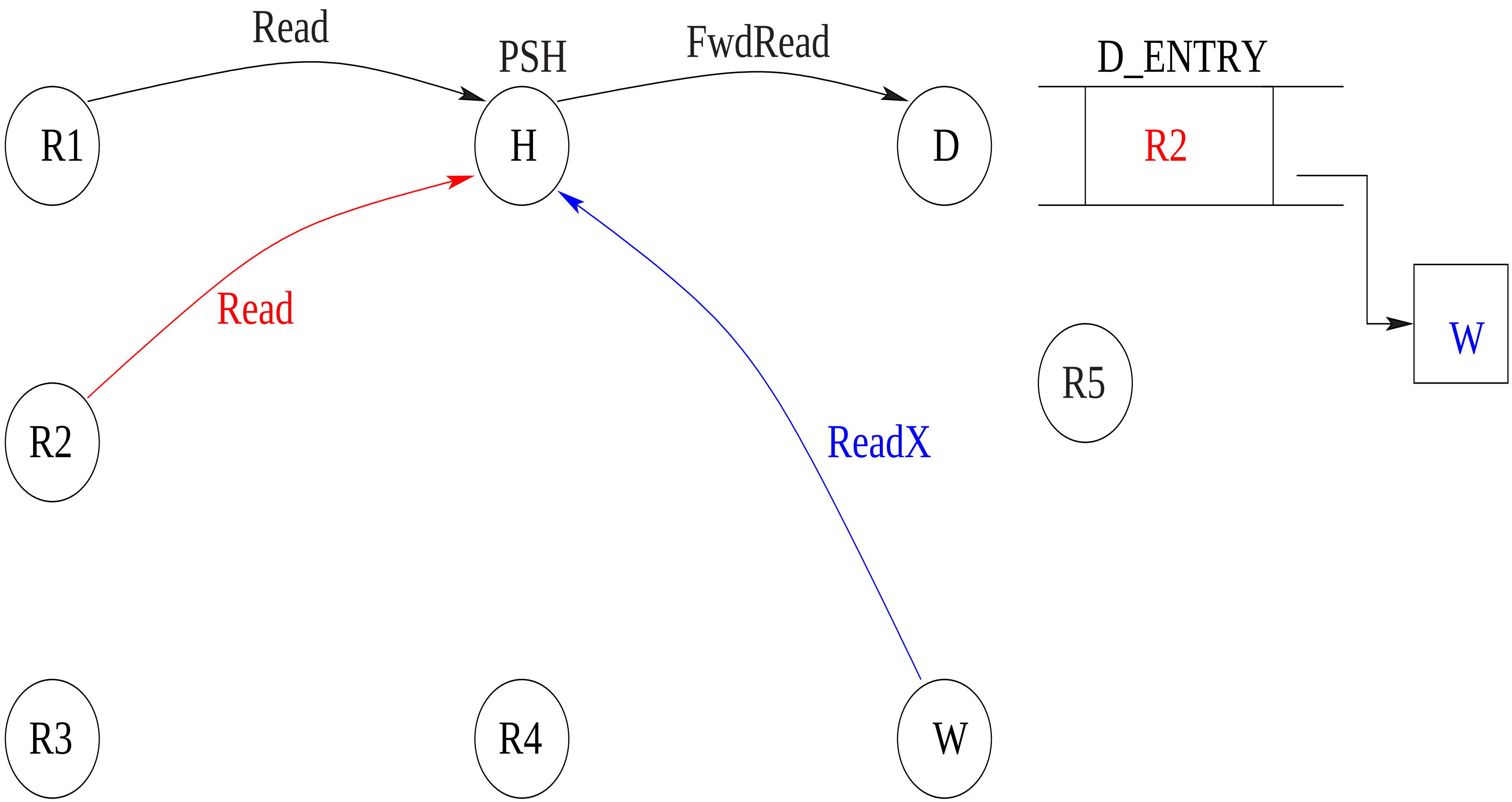
Enabling Read Combining (RComb)



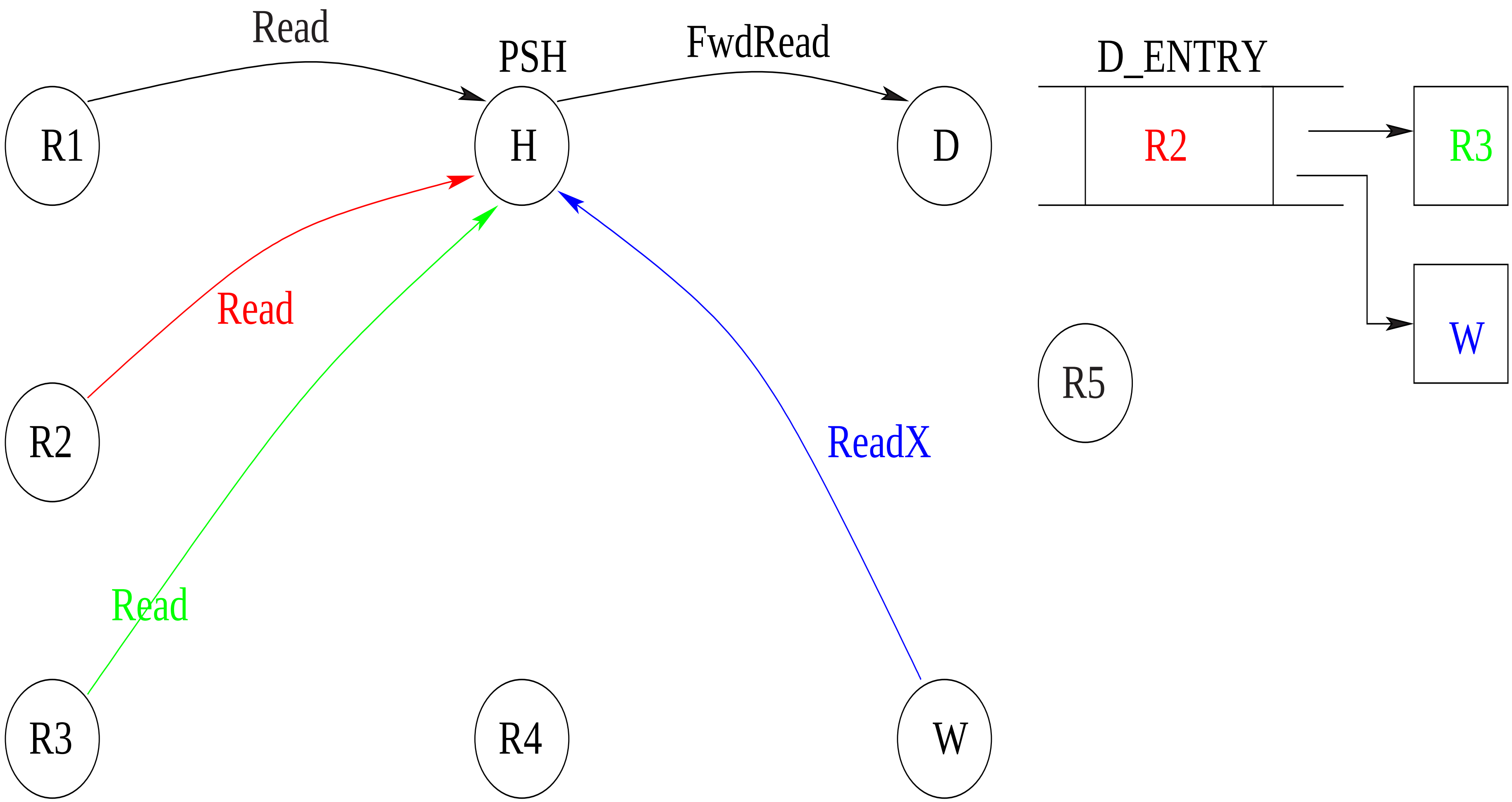
Enabling Read Combining (RComb)



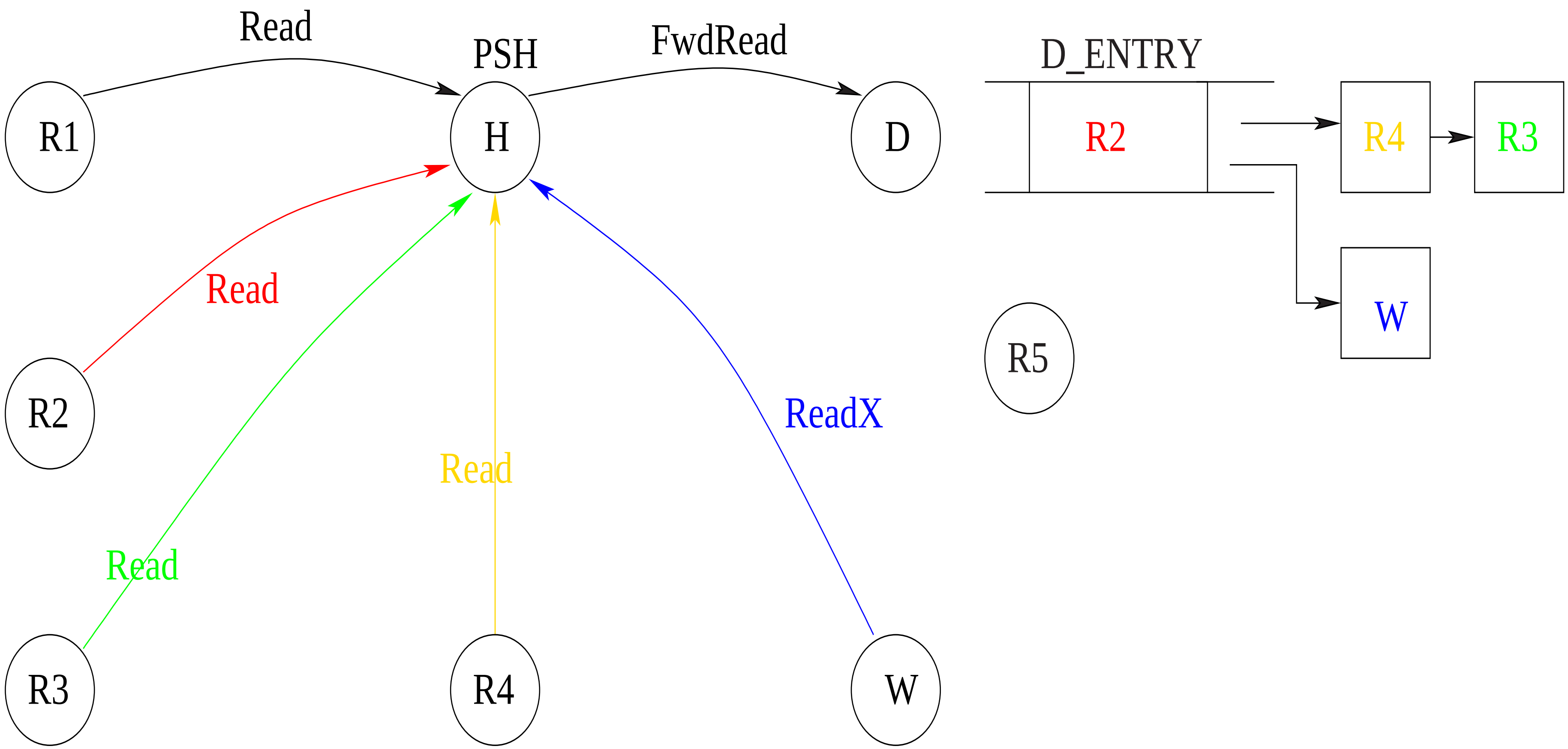
Enabling Read Combining (RComb)



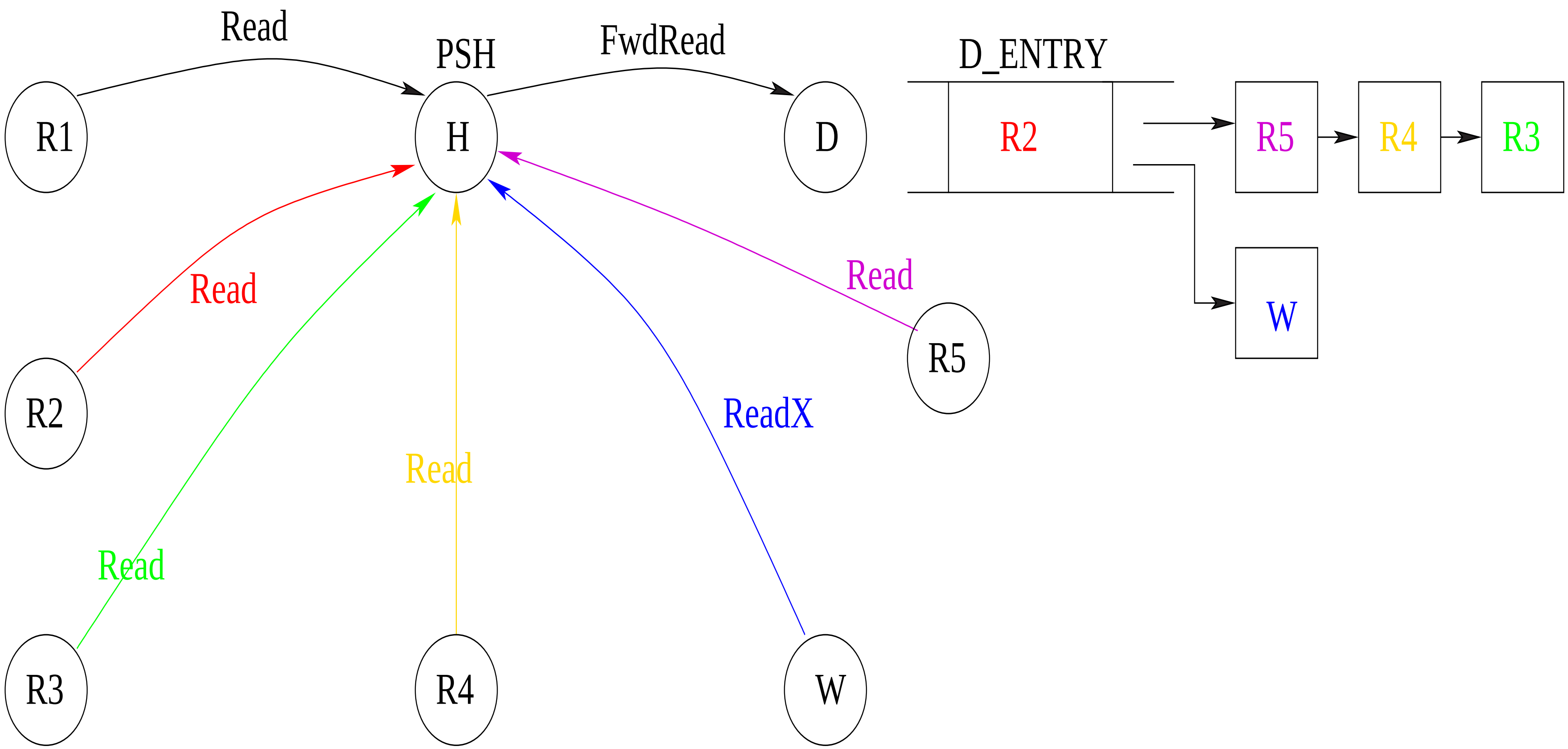
Enabling Read Combining (RComb)



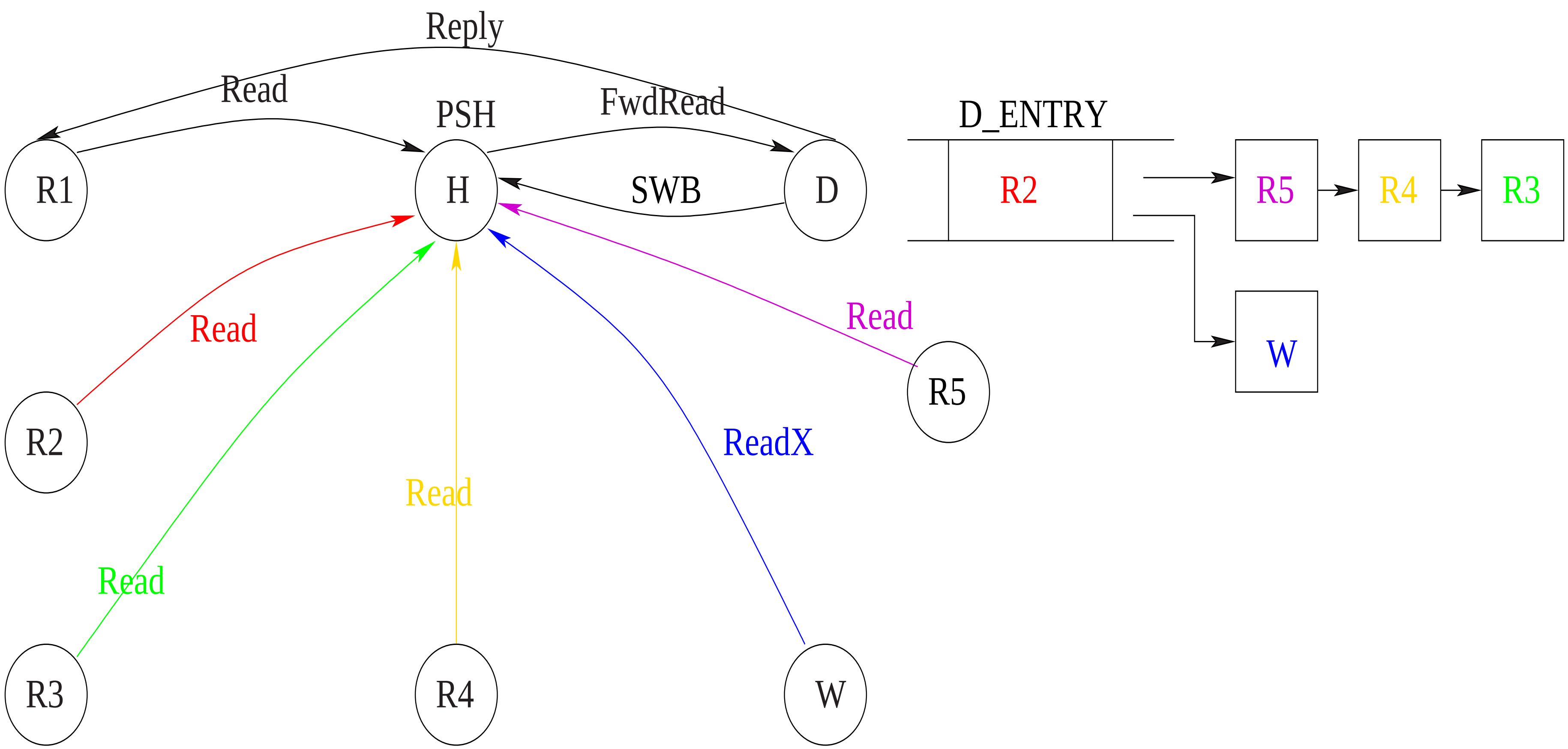
Enabling Read Combining (RComb)



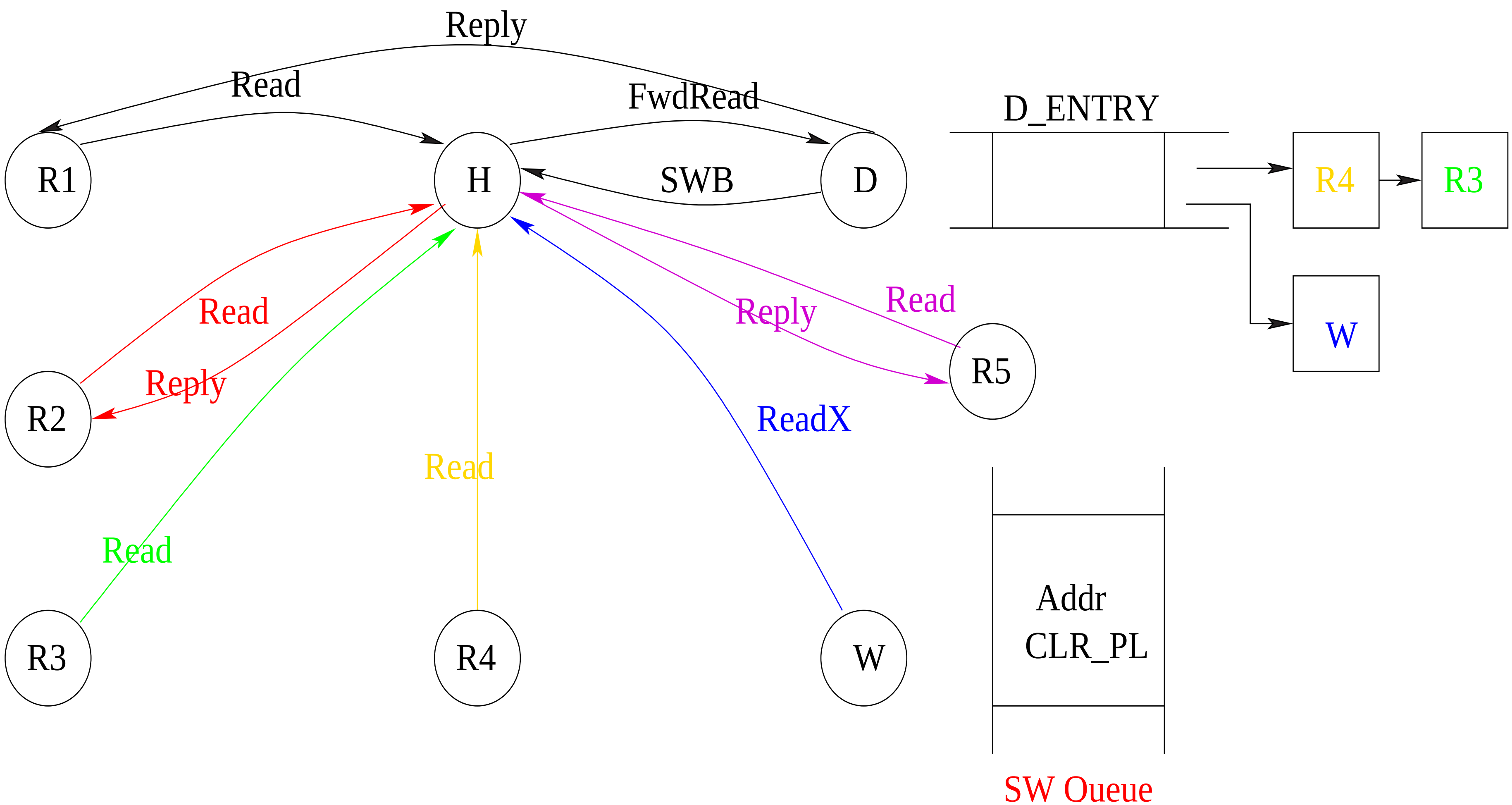
Enabling Read Combining (RComb)



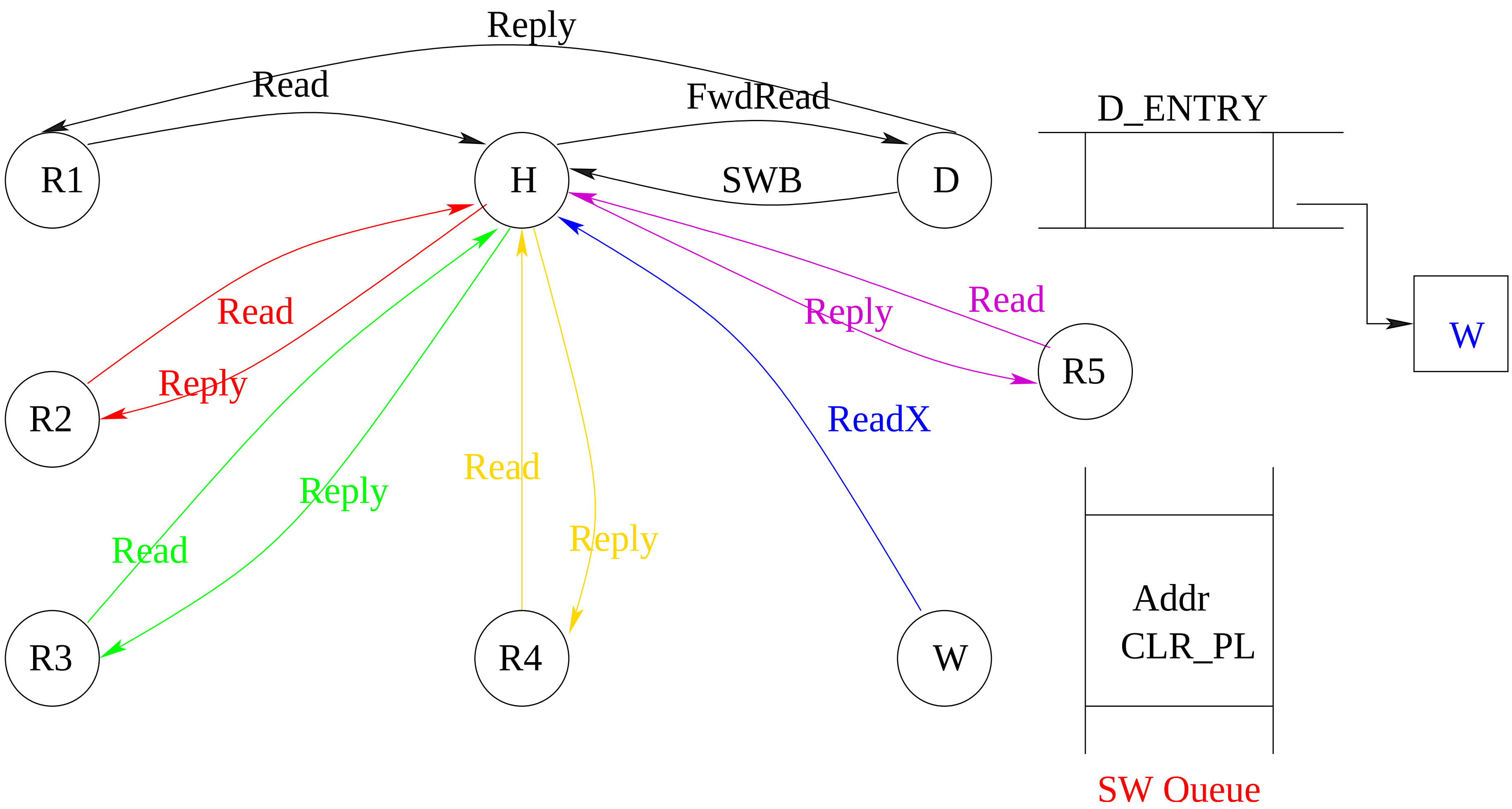
Enabling Read Combining (RComb)



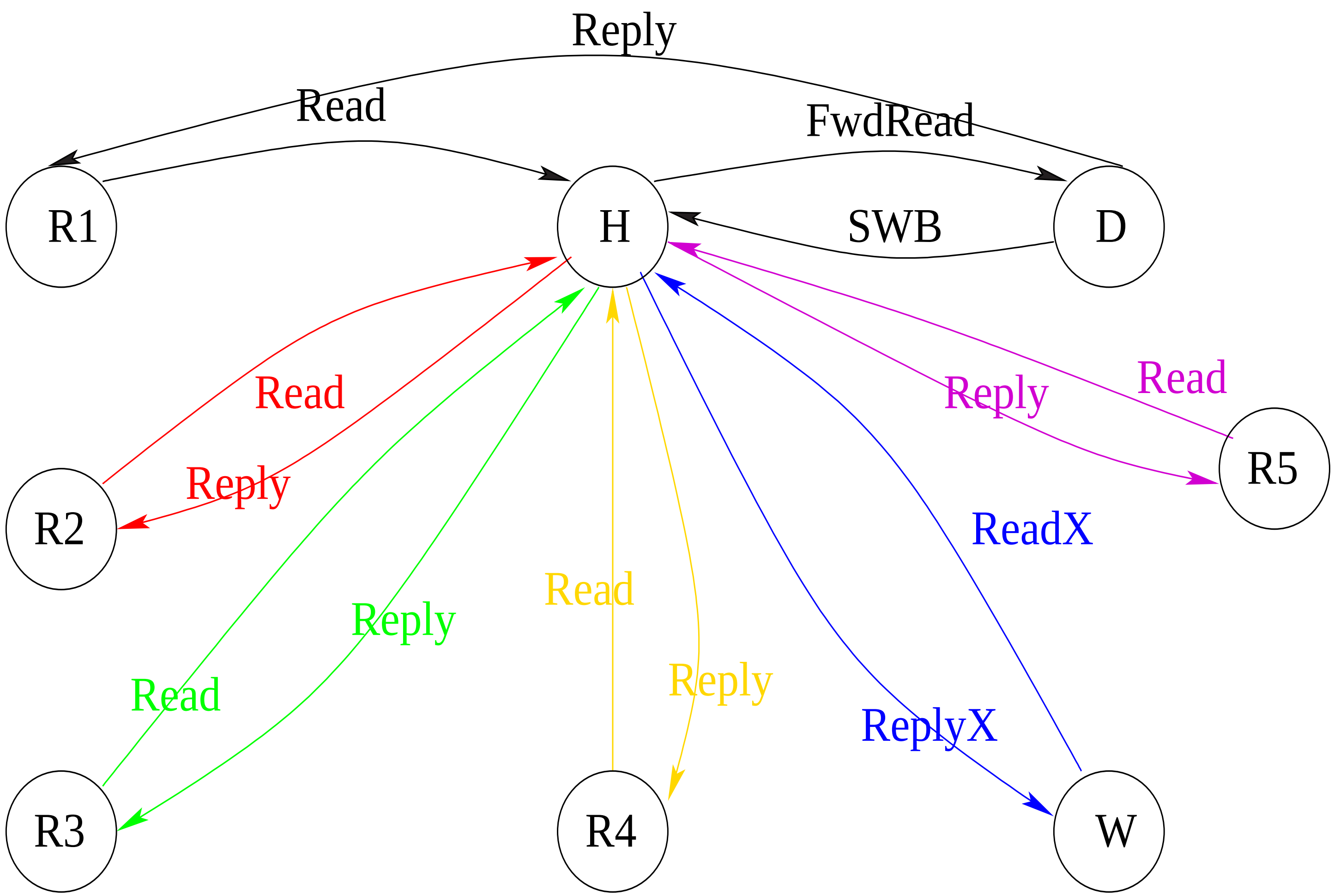
Enabling Read Combining (RComb)



Enabling Read Combining (RComb)



Enabling Read Combining (RComb)



Enabling Write Combining (RWComb+WSF)

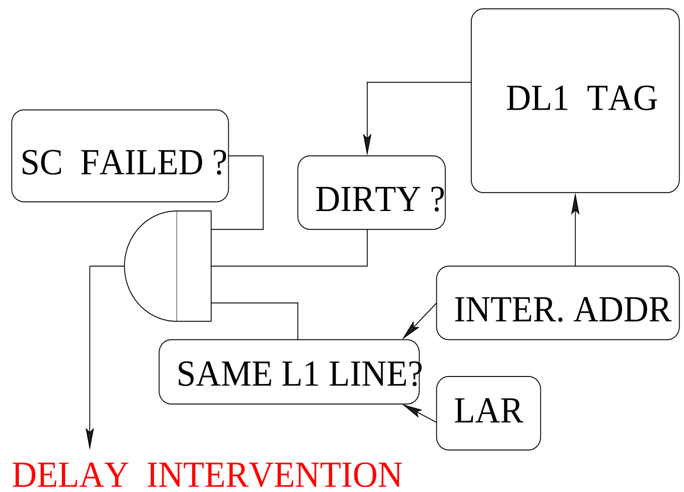
- We borrow the idea of WSF from Piranha
- Software queue handler sends out as many write interventions from the write pending chain as possible using two virtual lanes
- May hurt performance of heavily contended read-modify-writes and large critical sections

  LL; BRANCH
  INCREMENT
  SC; BRANCH
  CRITICAL SECTION
  UNLOCK
- Number of failed SC increases

Improving Read-modify-write Performance

[Similar to Rajwar et al, HPCA 2000, excluding the time-out]

Simple Changes in the L1 Cache Controller



- An LL instruction needs to unblock a pending intervention if it is looping: requires a one-bit state (No time-out)
- A SC instruction (not necessarily successful) unblocks a pending intervention

Request Combining: Overheads

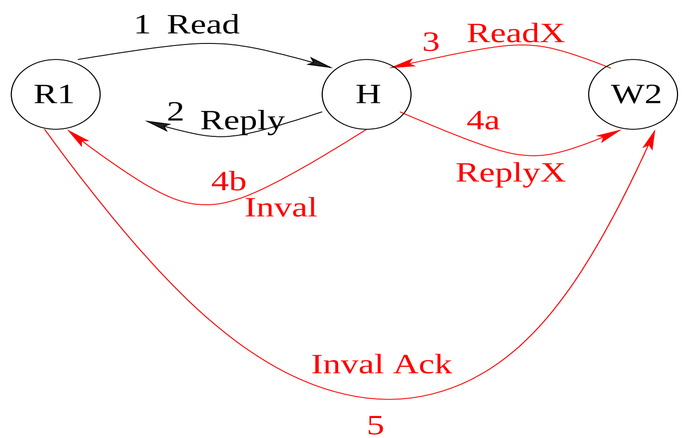
Sizing the Pending Lists

- Theoretical bound: $\mathrm{P} * \min(\mathrm{Size}_{\mathrm{MSHR}}, \mathrm{Size}_{\mathrm{OTT}})$
- In practice: contention happens for a single cache line
- Our design uses 128 entries: if fills up we revert to NACKs
- One entry: entry id (a 32 bit integer), a 64-bit vector encapsulating requester id and quad word offset of the requested physical address, a next pointer (32 bits)
- DRAM overhead per node (for 128 entries): 16*128*2 bytes i.e. 4KB

Residual Nacks

They still remain! (very small in number)

- Remaining Nacks arise from read-invalidate races



- Necessary to preserve write atomicity
- GS320 sends a marker message to filter useless invalidations (due to not having replacement hints)
  - Invalidations arriving before the marker are dropped: requires point-to-point ordering in network

Evaluated Protocols

- **BaseBV:** Base bitvector
- **OriginMod:** Modified Origin 2000 protocol
- **OriginMod+RComb:** Read combining with OriginMod
- **OriginMod+RWComb+WSF:** Read and write combining with OriginMod
- **OriginMod+RWComb+WSF+OPT:** Read and write combining with delayed intervention improvement on LL/SC
- **OriginMod+DSH+WSF(+OPT):** Dirty sharing and write string forwarding

All protocols run with coarseness of 2 (64 nodes) or 4 (128 nodes)

Evaluation

Applications from SPLASH-2

- Ocean, Barnes Hut, LU, Water, Radix-Sort, 1D FFT

Simulation Environment

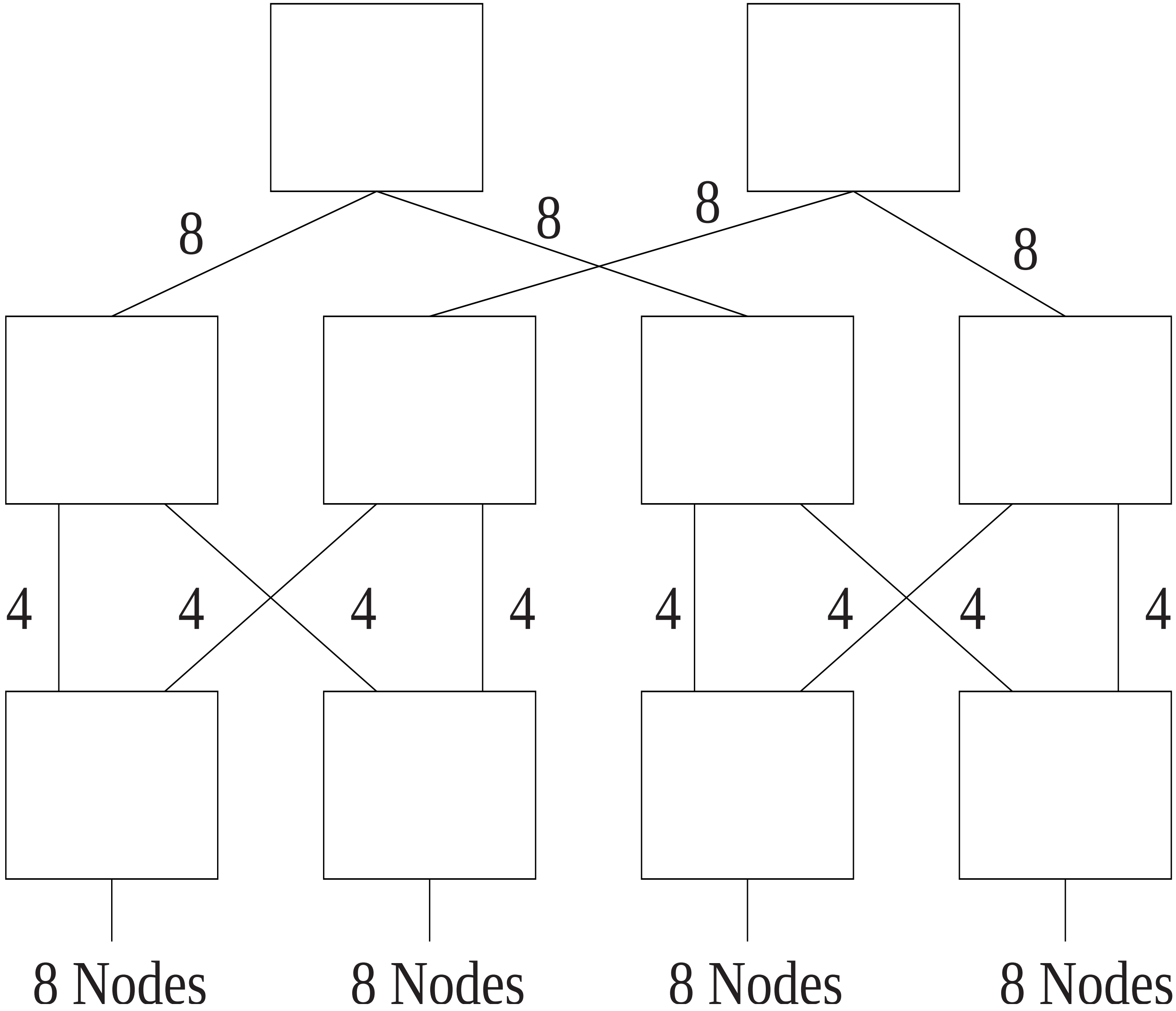
- 1GHz processor, IL1: 32KB/64B/2-way/LRU, DL1: 32KB/32B/2-way/LRU, UL2: 2MB/128B/2-way/LRU, ITLB: 8/FA/R, DTLB: 64/FA/R, Page size: 4KB
- 400MHz system clock, split transaction bus
- Memory controller: 8-entry OTT, 4-entry WBB
- Latencies: 125ns DRAM latency to the first 64 bits

Evaluation

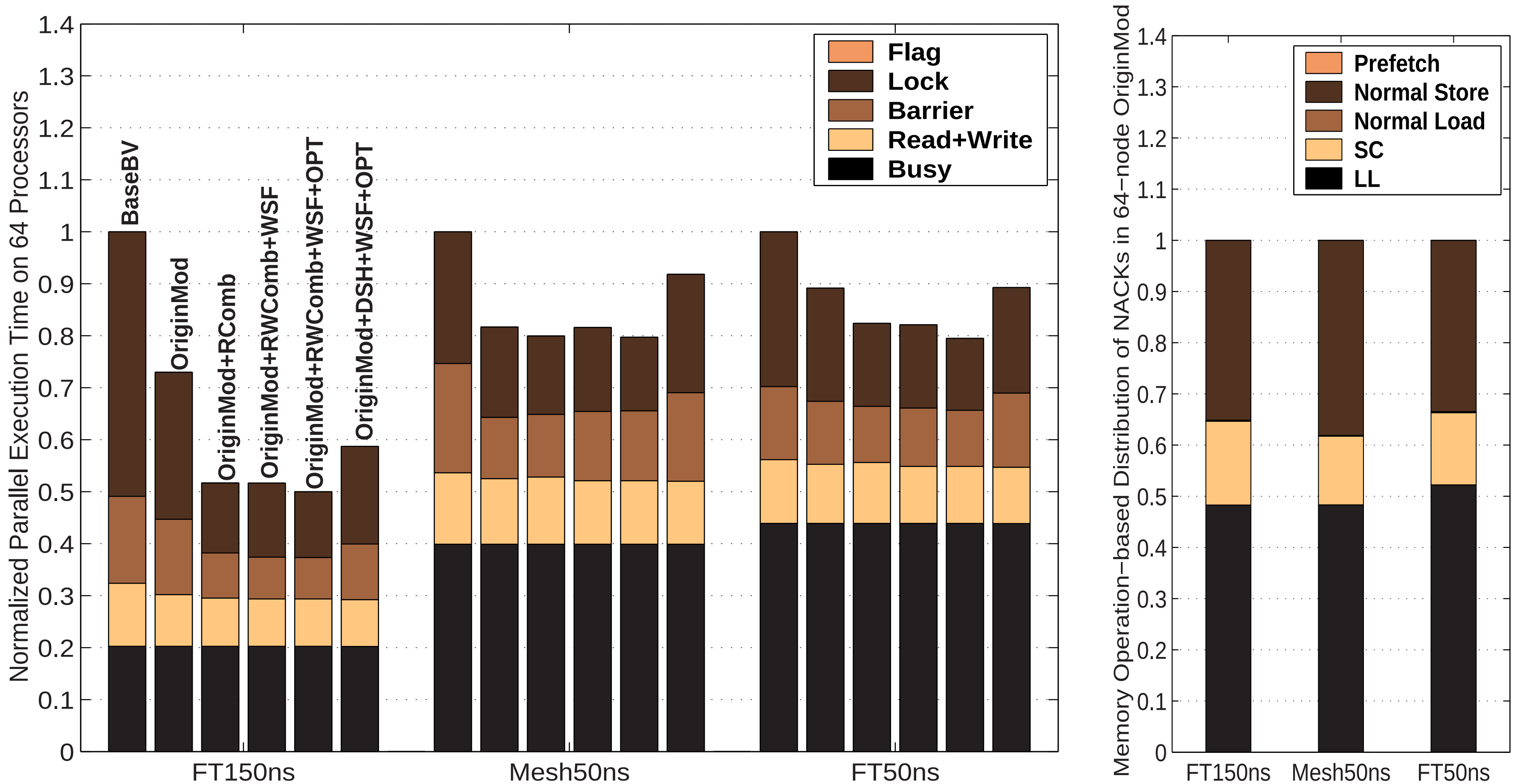
Simulation Environment

- Network configurations:
  - FT150ns/FT50ns: fat tree connected crossbars, 16-port switch, 150ns/50ns routing delay, Link BW 1GB/s
  - Mesh50ns: 2D mesh, 6-port switch, 50ns routing delay, Link BW 1GB/s

Fat Tree Connected Crossbar (32 nodes)



Water

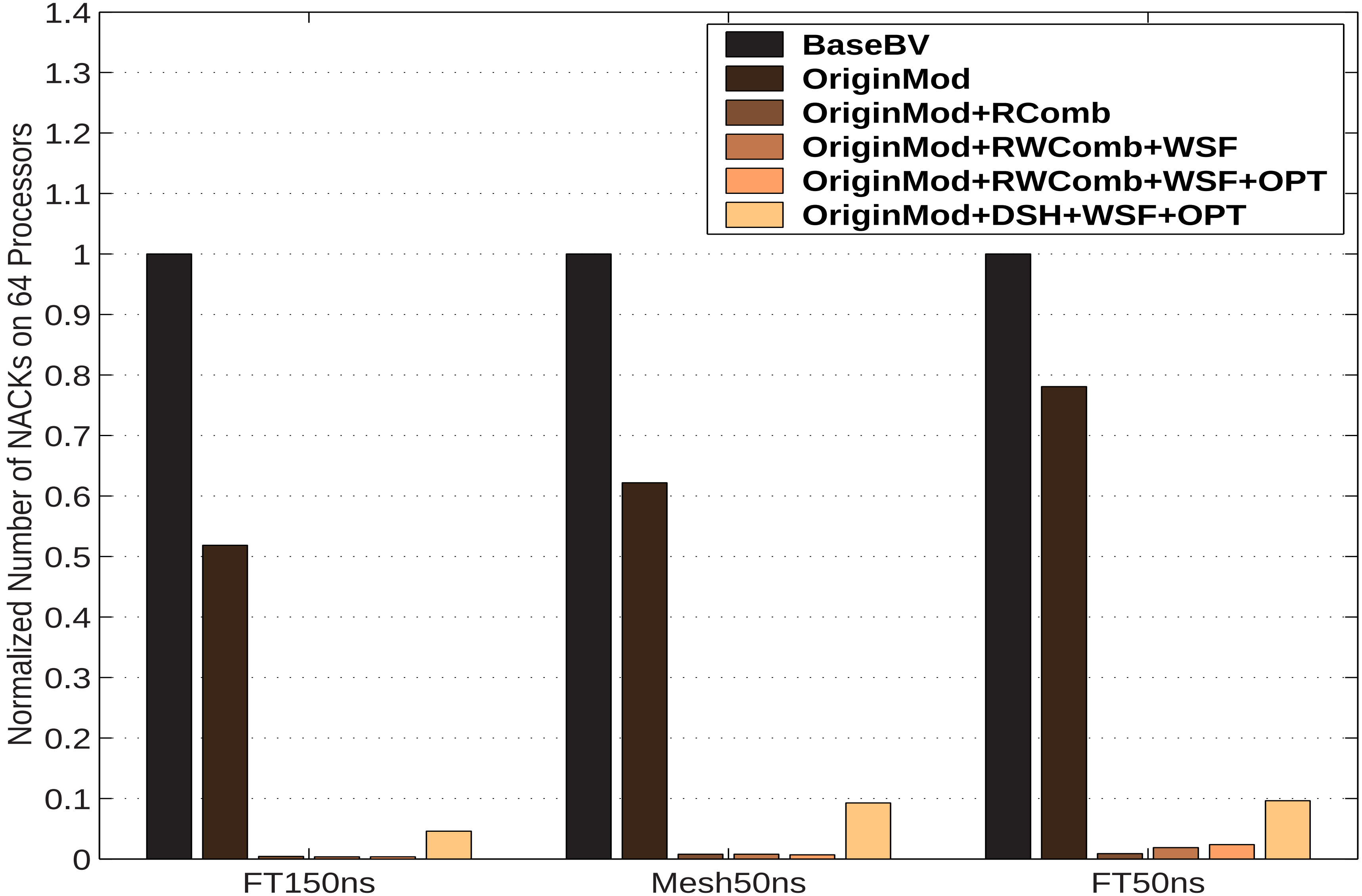


RComb is (93%, 25%, 21%) faster compared to BaseBV

(41%, 2%, 8%) faster compared to OriginMod

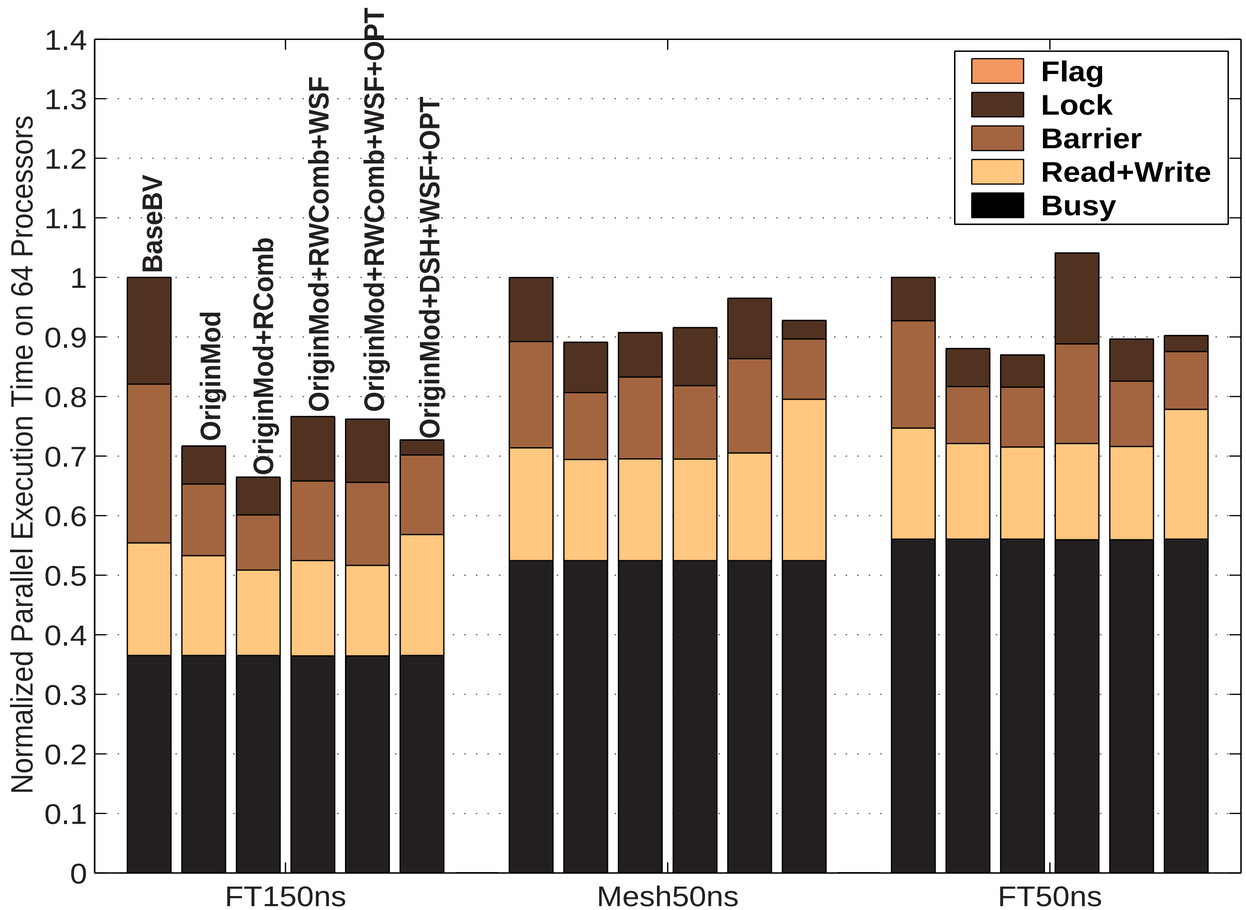
Combining in RComb: (60, 60, 60), RWComb: (39, 56, 48)

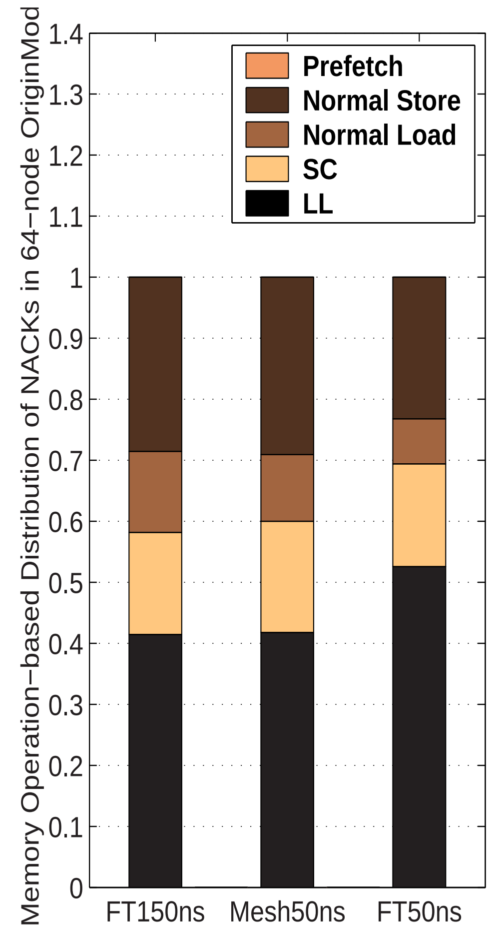
Water [Nacks]



Normalized Number of NACKs on 64 Processors
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
BaseBV
OriginMod
OriginMod+RComb
OriginMod+RWComb+WSF
OriginMod+RWComb+WSF+OPT
OriginMod+DSH+WSF+OPT
FT150ns
Mesh50ns
FT50ns

Barnes Hut with 64K Locks



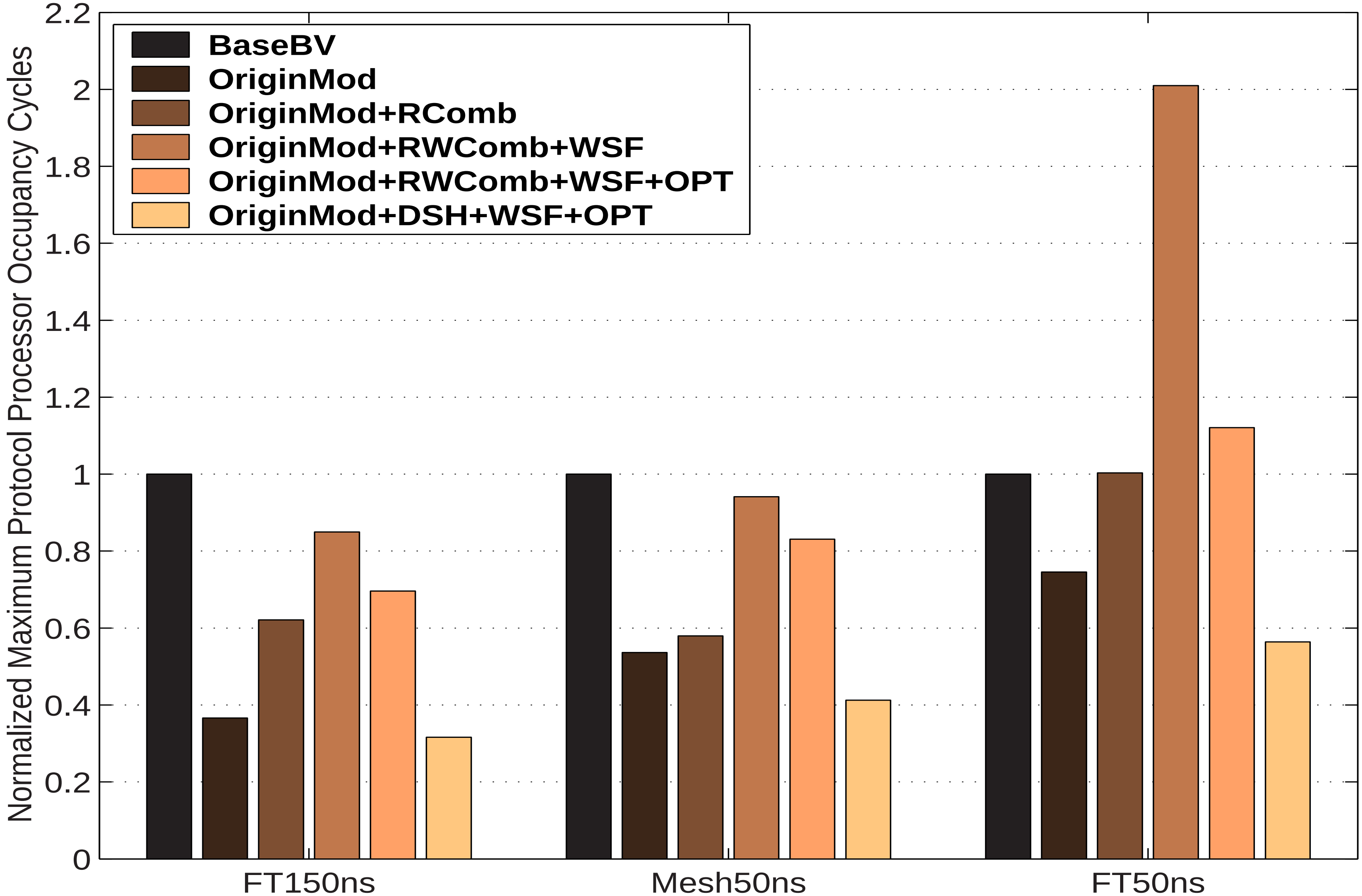


RComb is (51%, 10%, 15%) faster compared to BaseBV

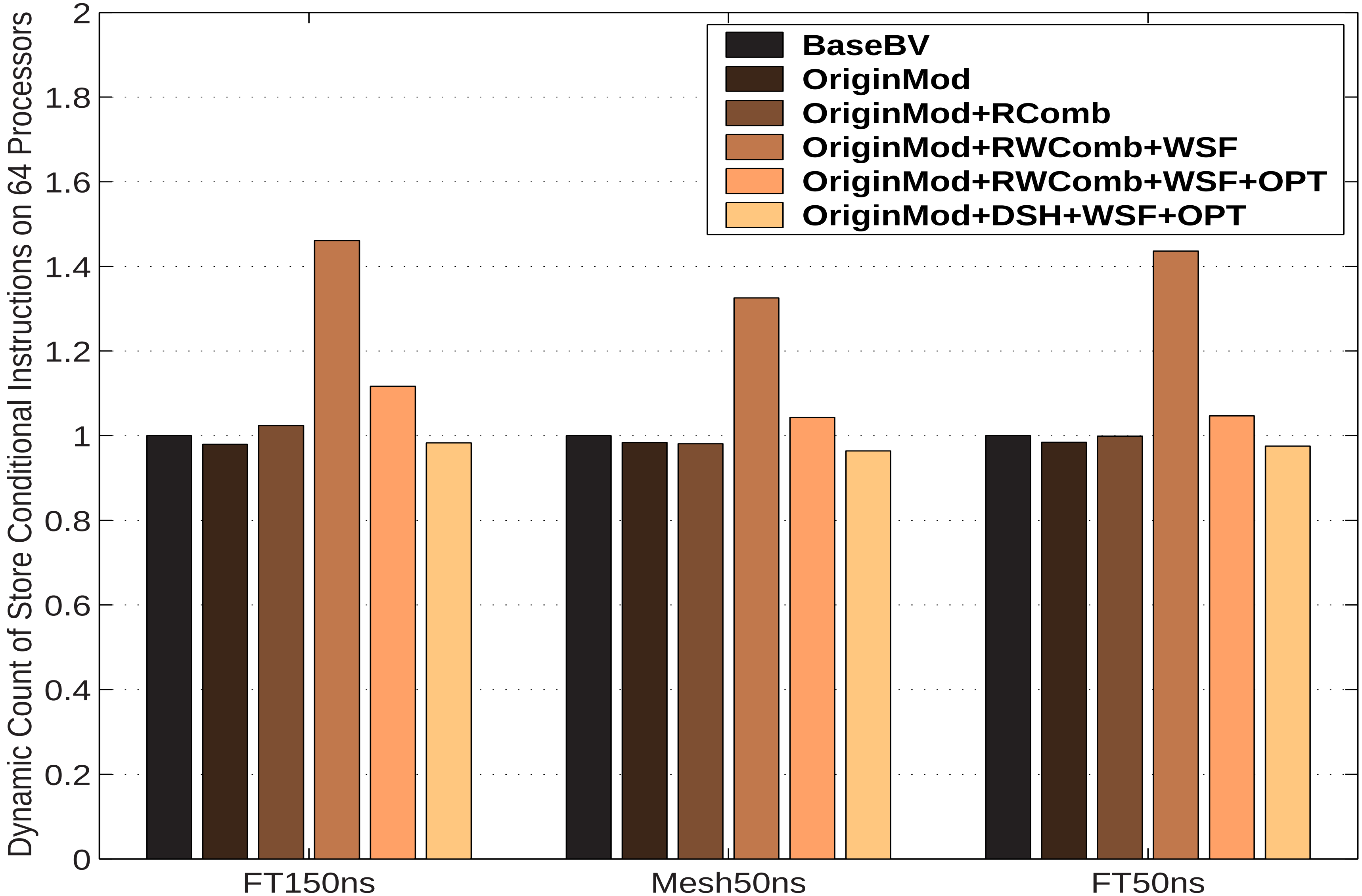
(8%, -2%, 1%) faster compared to OriginMod

Combining in RComb: (35, 40, 30), RWComb: (21, 25, 42)

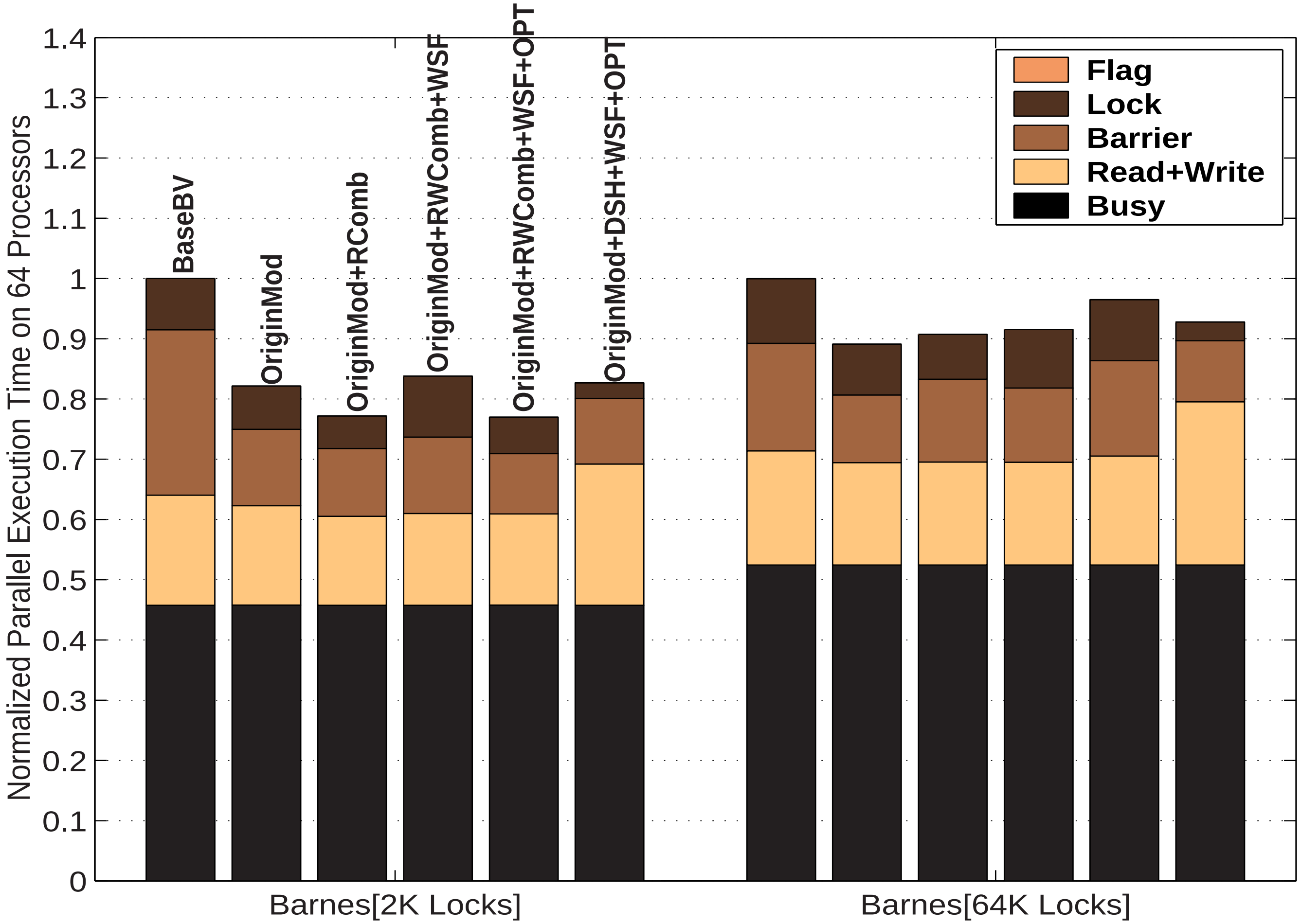
Barnes Hut with 64K Locks [Occupancy]



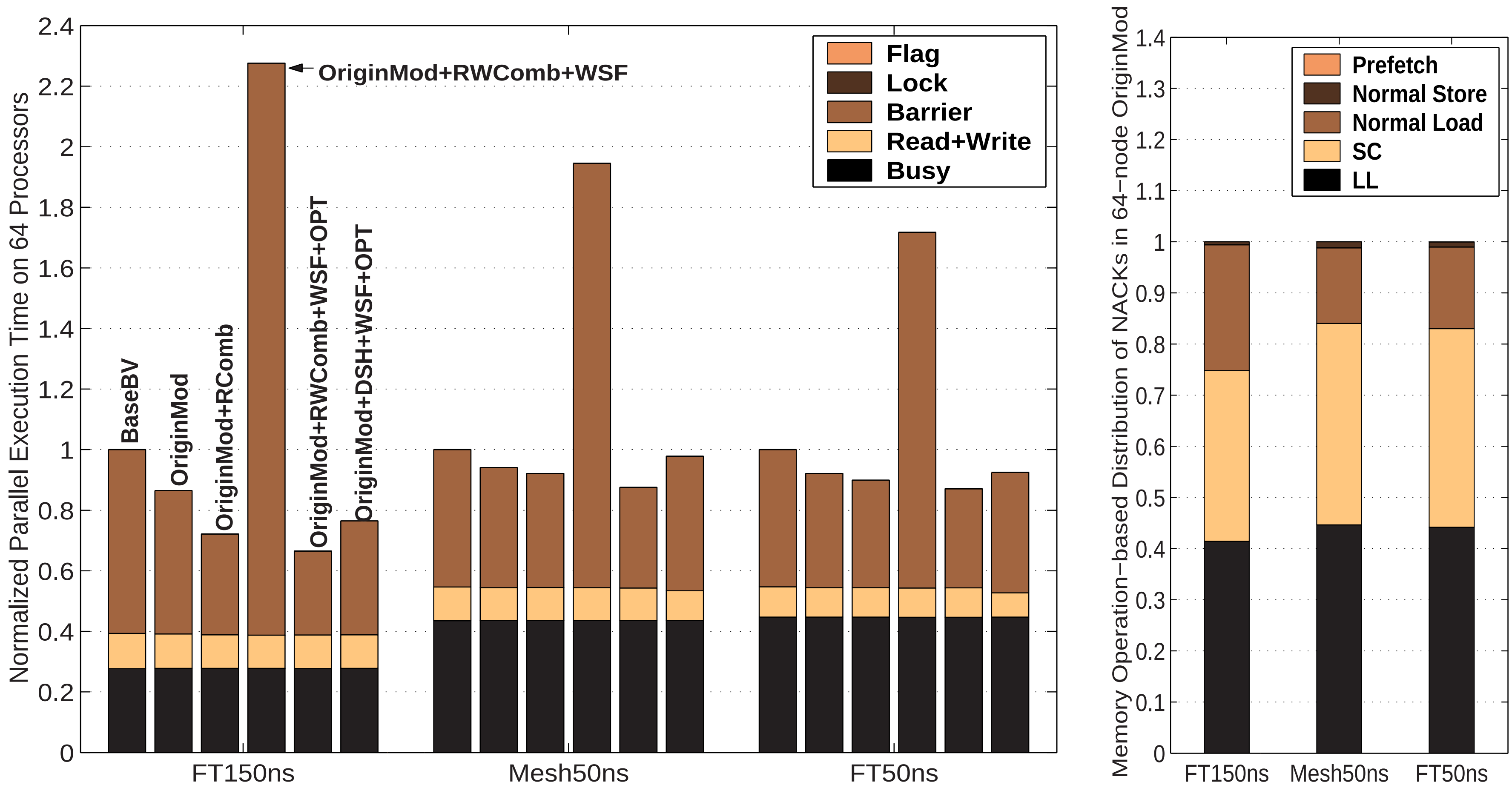
Barnes Hut with 64K Locks [SC]



Barnes Hut with 2K Locks [Mesh50ns]



Unoptimized LU

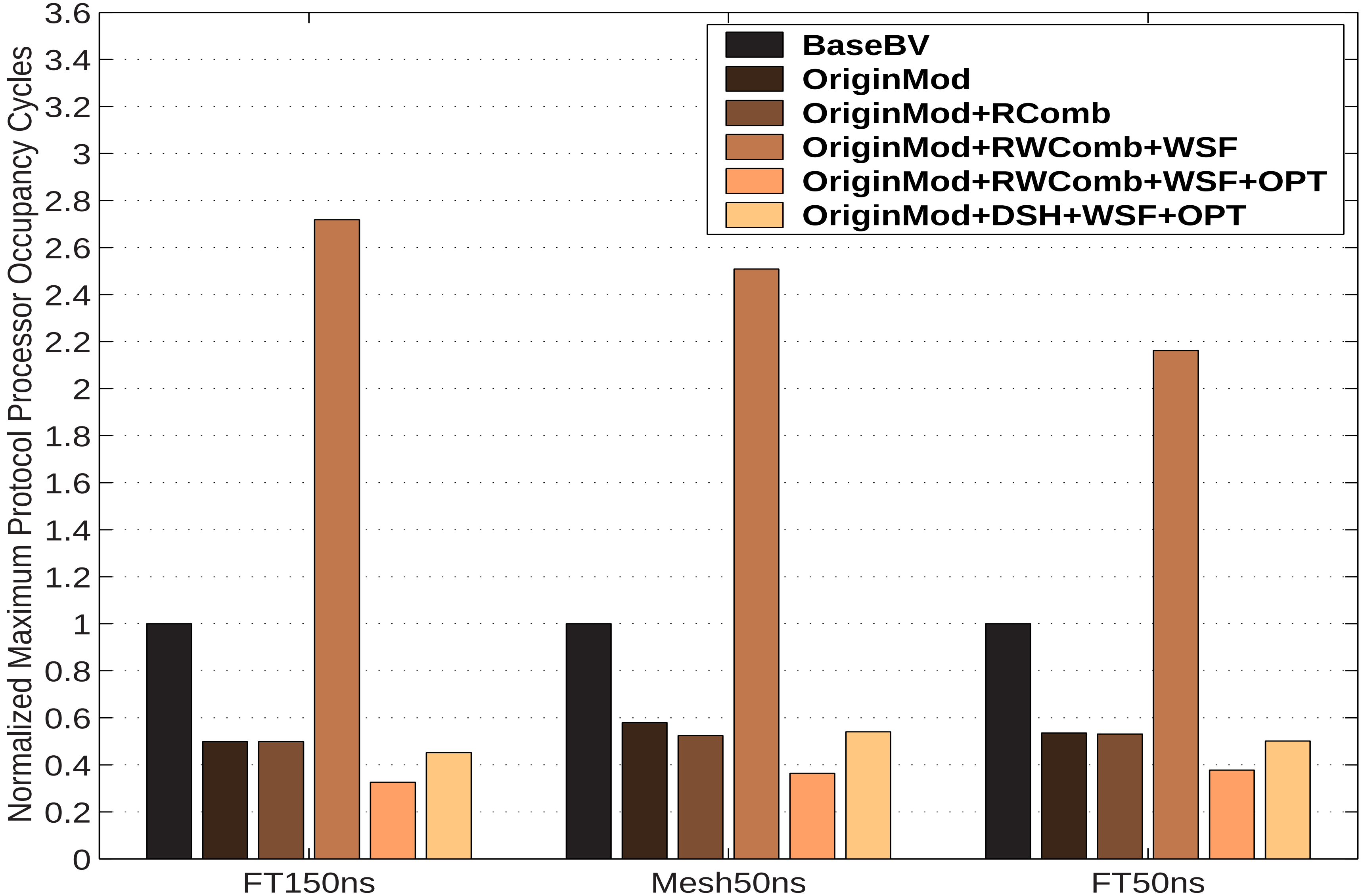


RComb is (39%, 8%, 11%) faster compared to BaseBV

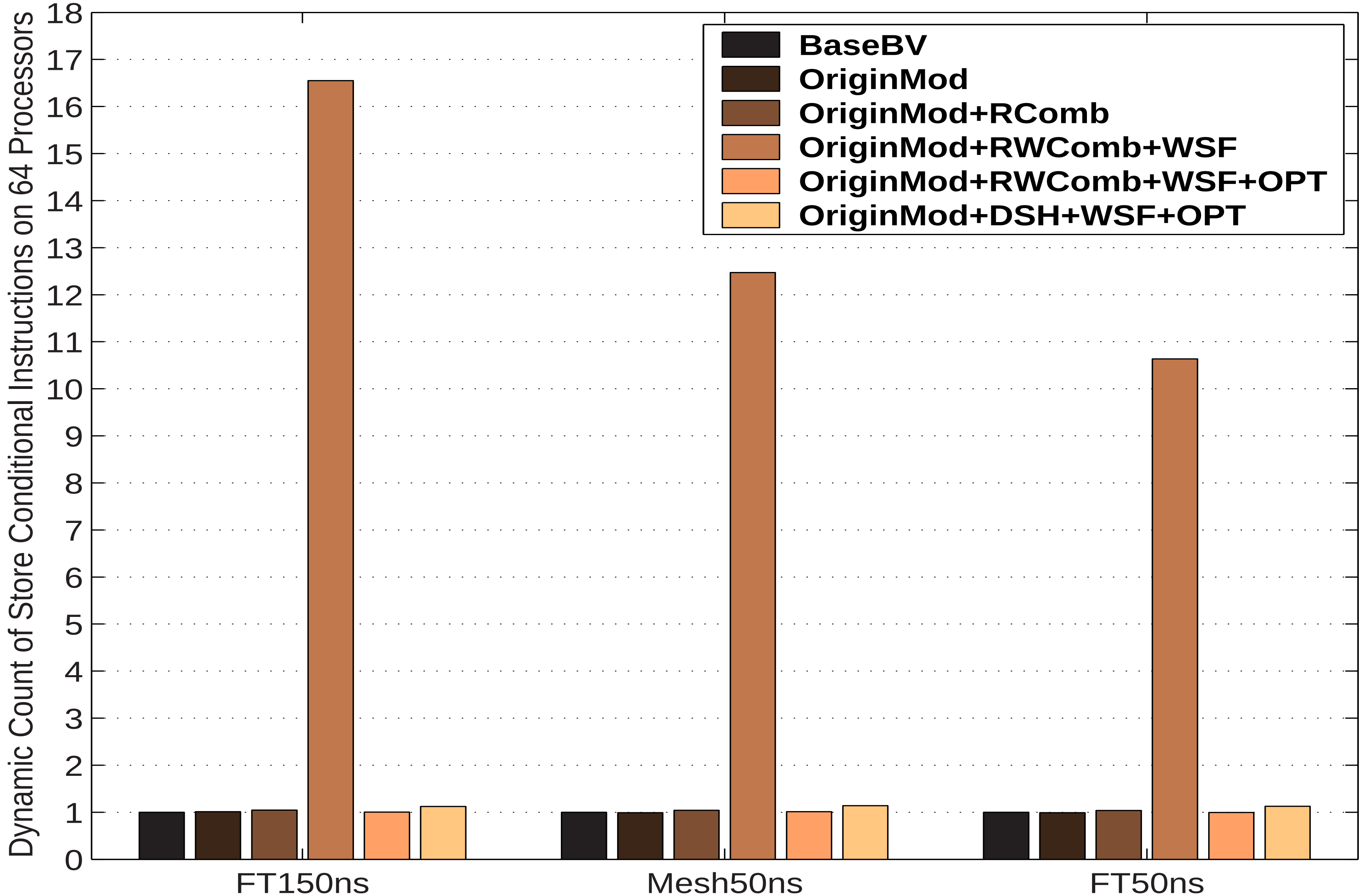
(20%, 2%, 2%) faster compared to OriginMod

Combining in RComb: (60, 56, 39), RWComb: (34, 57, 51)

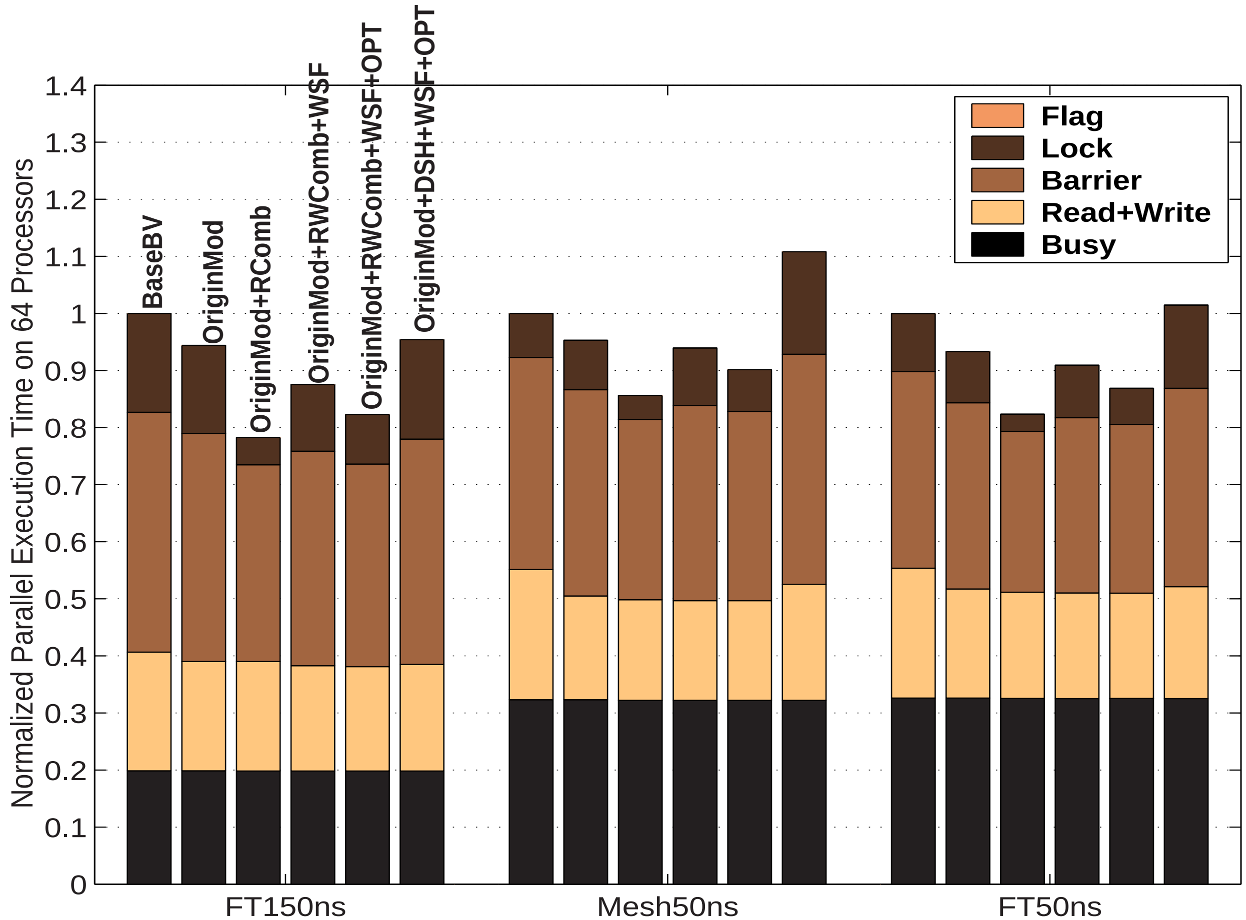
Unoptimized LU [Occupancy]

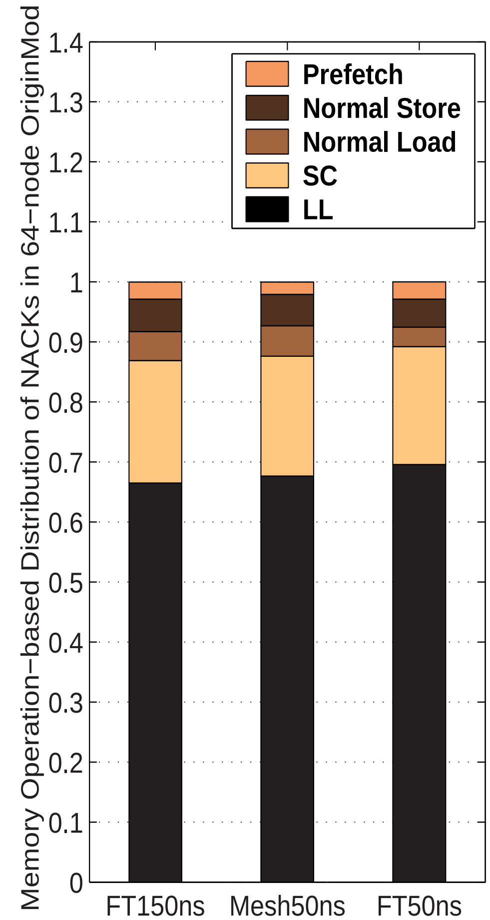


Unoptimized LU [SC]



Ocean



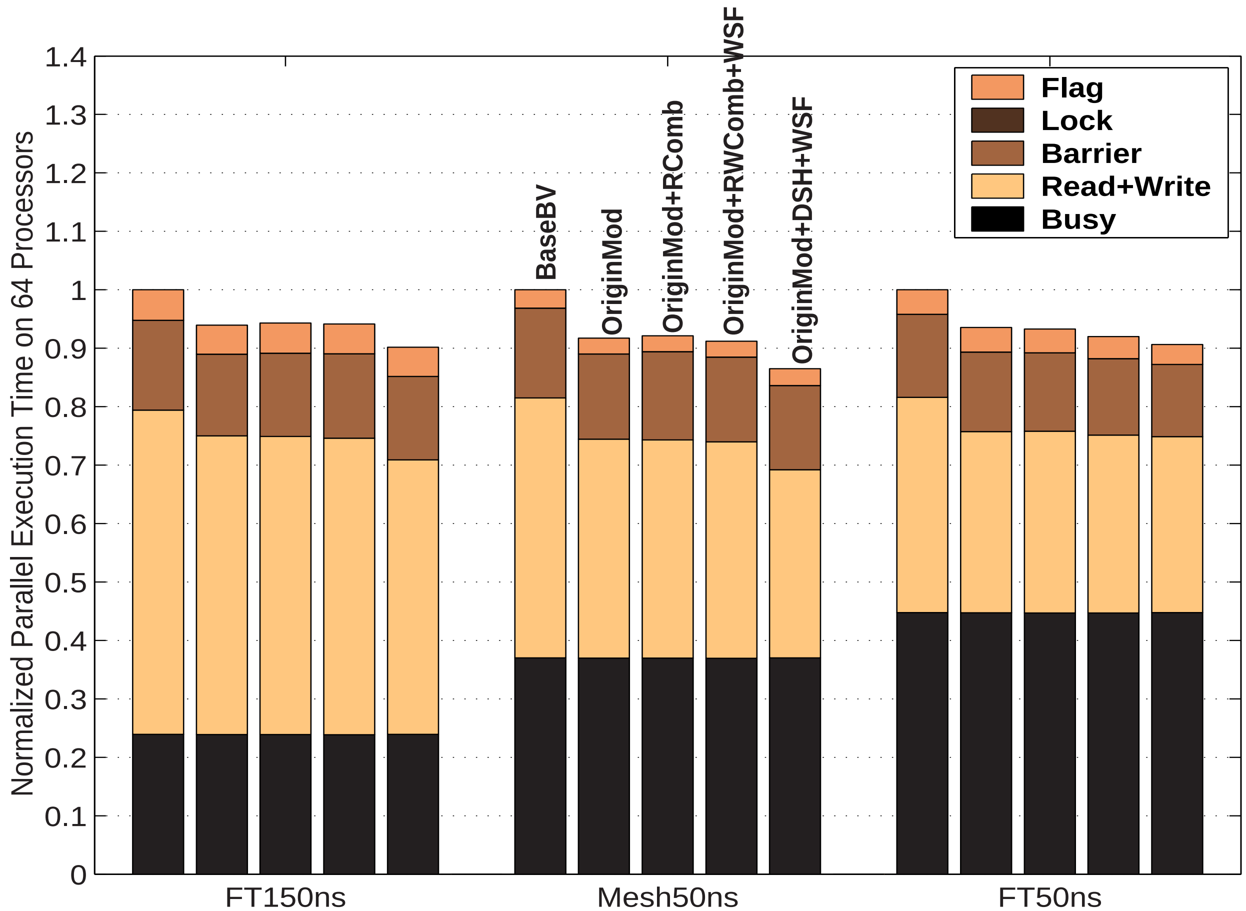


RComb is (28%, 17%, 21%) faster compared to BaseBV

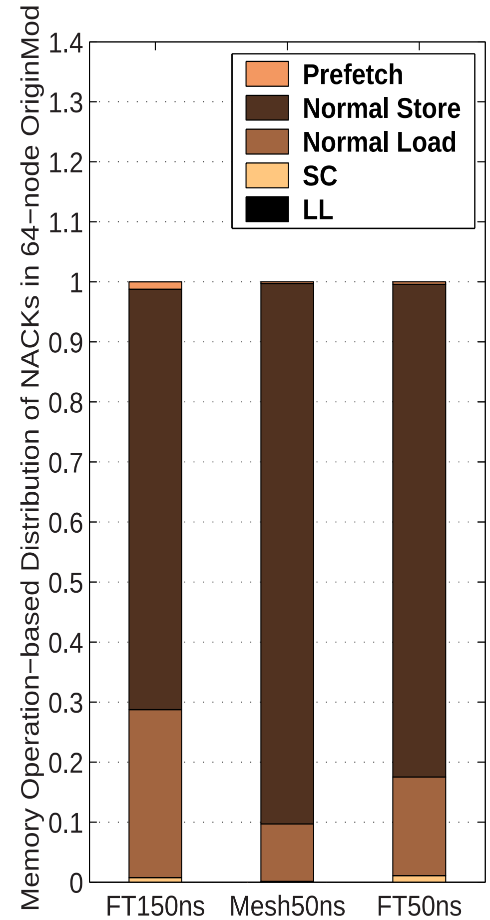
(21%, 11%, 13%) faster compared to OriginMod

Combining in RComb: (34, 45, 32), RWComb: (40, 54, 49)

Radix-Sort

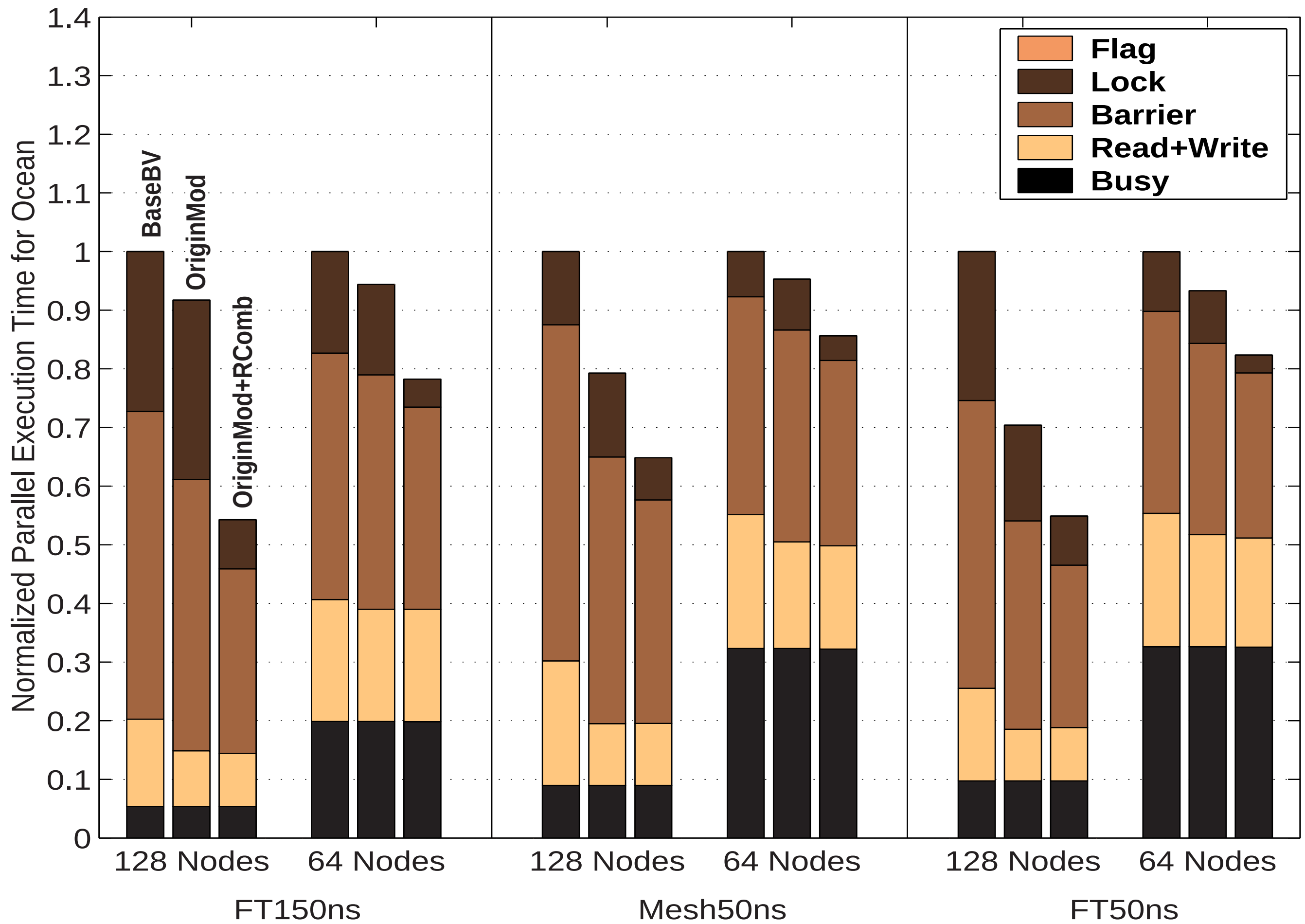


Normalized Parallel Execution Time on 64 Processors
BaseBV
OriginMod
OriginMod+RComb
OriginMod+RWComb+WSF
OriginMod+DSH+WSF
Flag
Lock
Barrier
Read+Write
Busy
FT150ns
Mesh50ns
FT50ns



Memory Operation−based Distribution of NACKs in 64−node OriginMod
Prefetch
Normal Store
Normal Load
SC
LL
FT150ns
Mesh50ns
FT50ns

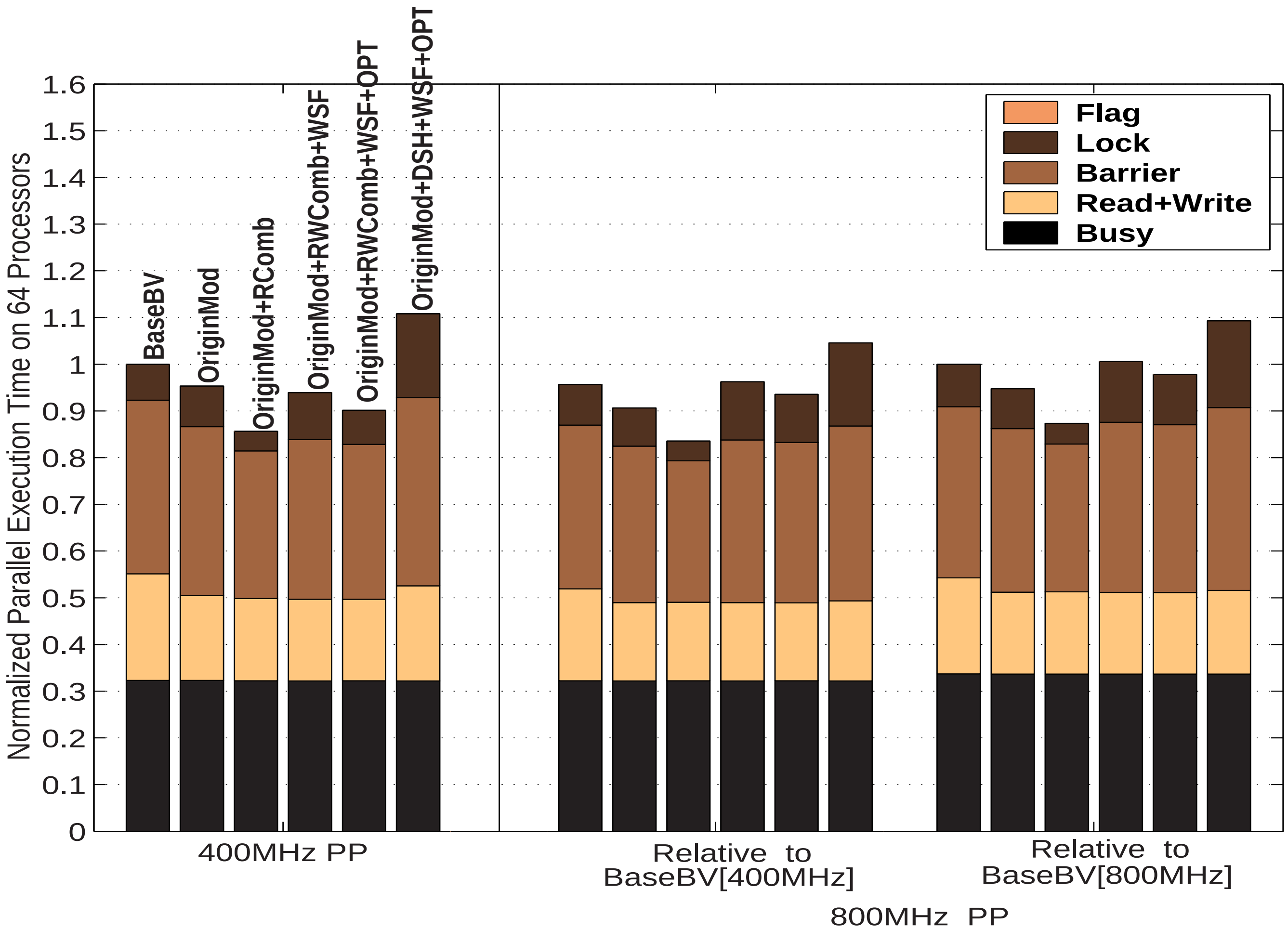
Ocean: 128 nodes



128 nodes: RComb is 69% faster compared to Origin

64 nodes: RComb is 17% faster compared to Origin

Hardwired Protocol Execution [Ocean]



Summary: The Best Protocol

Apps	FT150ns	Mesh50ns	FT50ns
Water	RComb, RWComb+WSF, RWComb+WSF+OPT	RComb, RWComb+WSF+OPT	RComb, RWComb+WSF, RWComb+WSF+OPT
Barnes (64K)	RComb	OriginMod	OriginMod, RComb
LU	RComb, RWComb+WSF+OPT	RComb, RWComb+WSF+OPT	RComb, RWComb+WSF+OPT
Ocean	RComb	RComb	RComb
Radix	DSH+WSF	DSH+WSF	DSH+WSF
FFT	OriginMod, RWComb+WSF	RComb, RWComb+WSF	RComb

Summary: Speedup of RComb

Apps	Relative to BaseBV			Relative to OriginMod		
	FT150ns	Mesh50ns	FT50ns	FT150ns	Mesh50ns	FT50ns
Water	1.93	1.25	1.21	1.41	1.02	1.08
Barnes	1.51	1.10	1.15	1.08	0.98	1.01
LU	1.39	1.08	1.11	1.20	1.02	1.02
Ocean	1.28	1.17	1.21	1.21	1.11	1.13
Radix	1.06	1.09	1.07	1.00	1.00	0.99
FFT	1.11	1.11	1.20	0.98	1.02	1.03

Conclusions

- Negative acknowledgments are important
- In general importance increases as the network gets slower and more contended
- Read combining emerges the best for majority of the cases
  - It accelerates contended read-modify-writes and large scale producer-consumer sharing
- Aggressive write forwarding may hurt performance of heavily contended read-modify-writes: requires some form of delayed intervention scheme
- Dirty sharing may hurt performance of large-scale producer-consumer sharing
- Read combining remains free of all these problems, but still improves load balance and overall performance by eliminating NACKs

The Impact of Negative Acknowledgments in Shared Memory Scientific Applications