Causality-Based Versioning

Kiran-Kumar Muniswamy-Reddy
and David A. Holland

Harvard School of Engineering and Applied Sciences

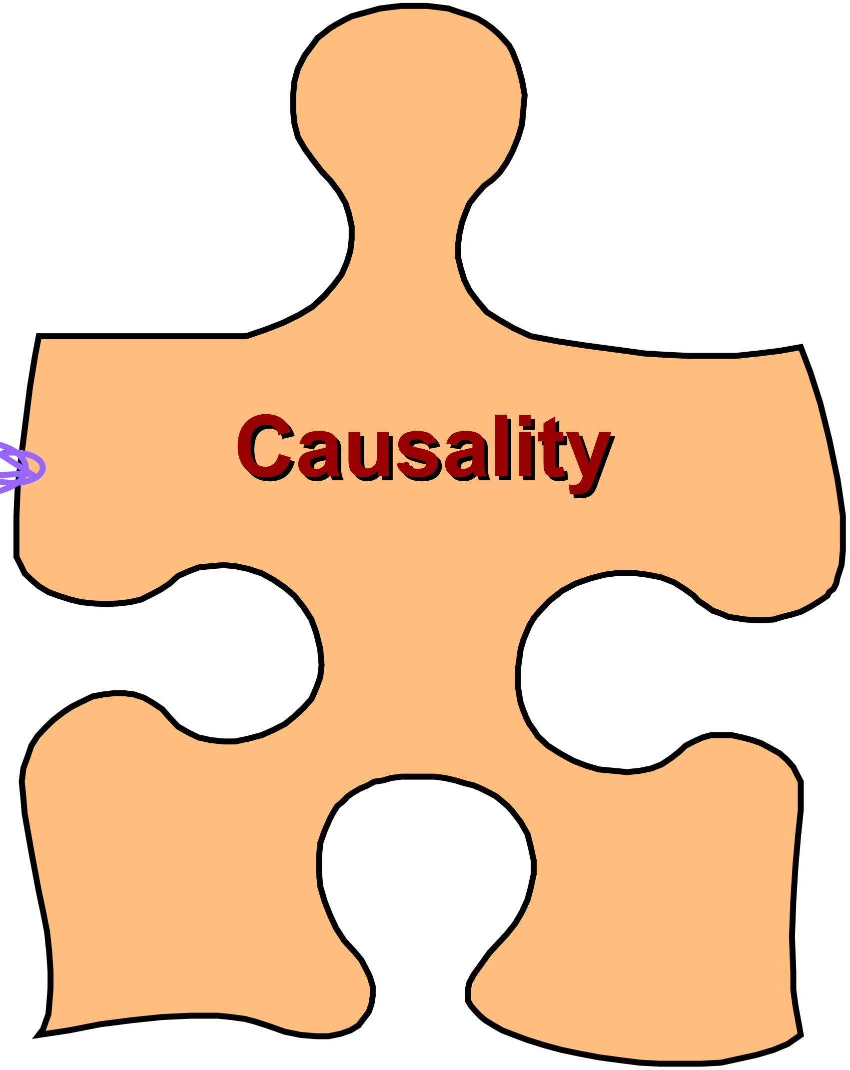
Consider this scenario

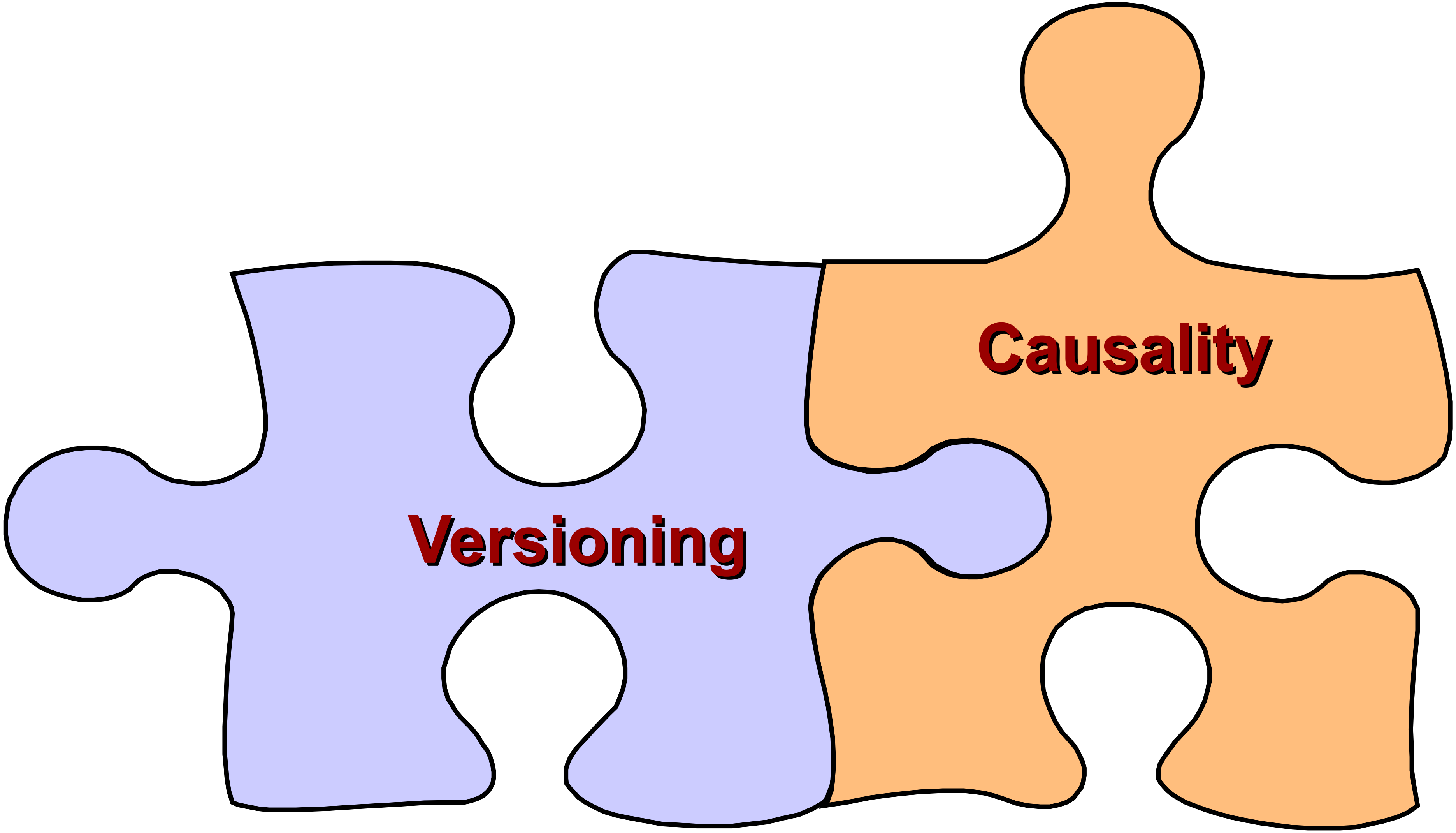
- I installed a piece of software
  - But.. that broke a few other tools!
- Uninstall not good enough
  - The config files were still corrupt


Versioning
But which files were modified?
Maintains old data to which you can recover

Tracks propagation of data and lets you find which files were modified

Too bad I don't have those old versions




Causality
Versioning

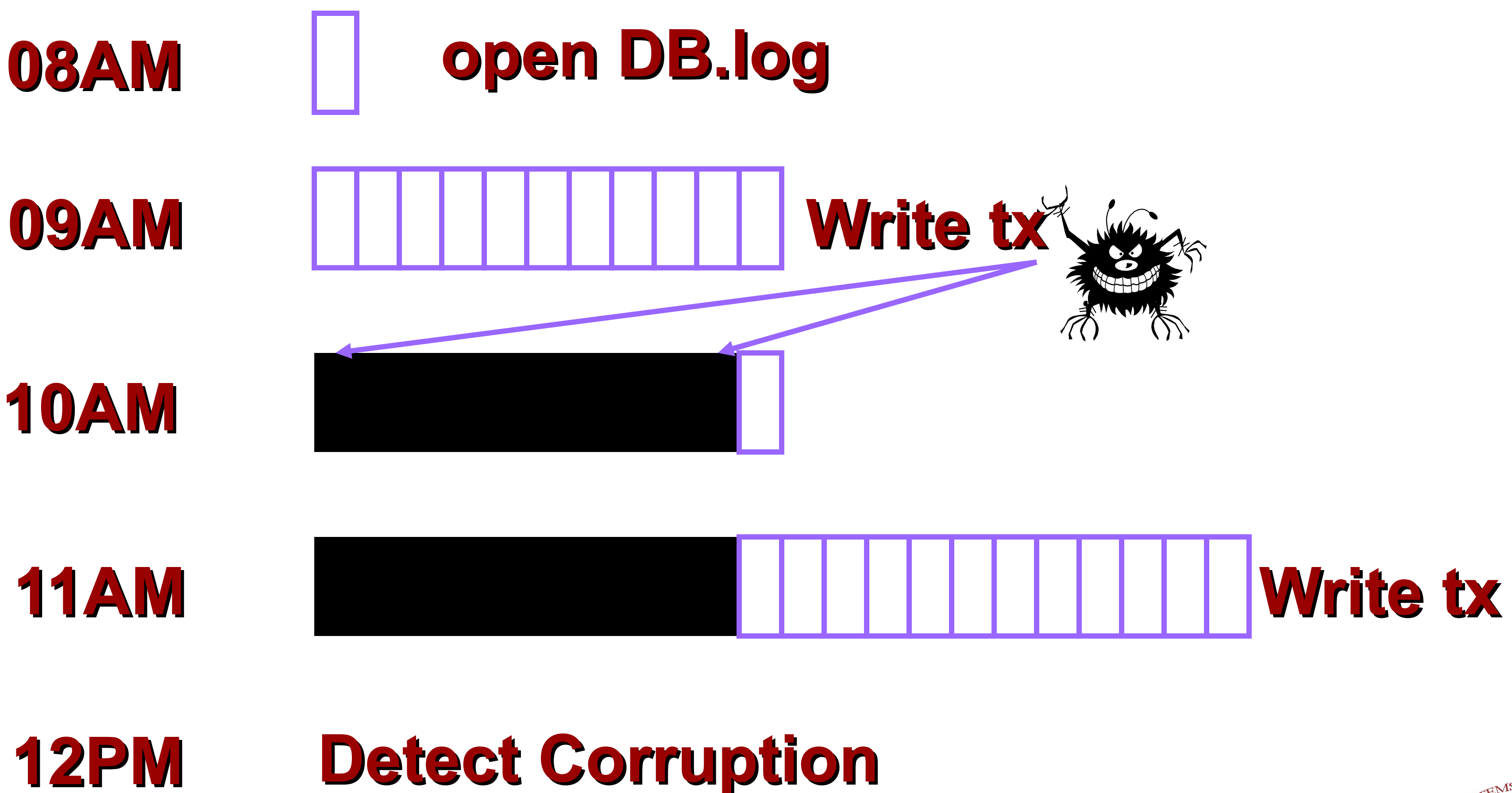
Applications of Versioning + Causality

- System Configuration Management
  - Causal data identifies files modified
  - Version data allows you to recover the files modified
- Intrusion Recovery
- IP Compliance
- Reproduce Research Results

Apache split-logfile Vulnerability

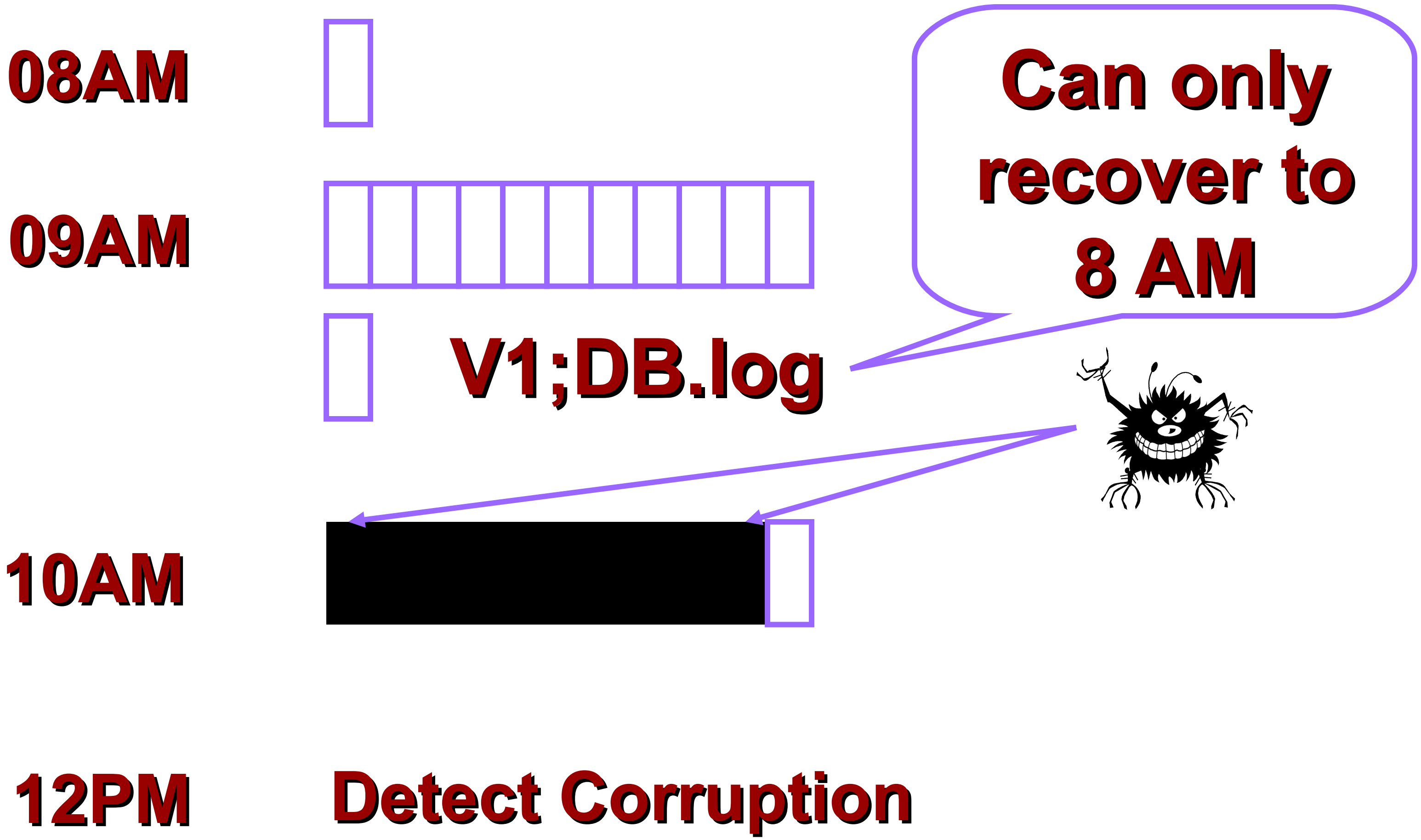
- Vulnerability in Apache 1.3
- Vulnerability allows attacker to overwrite any file with a .log extension

Scenario

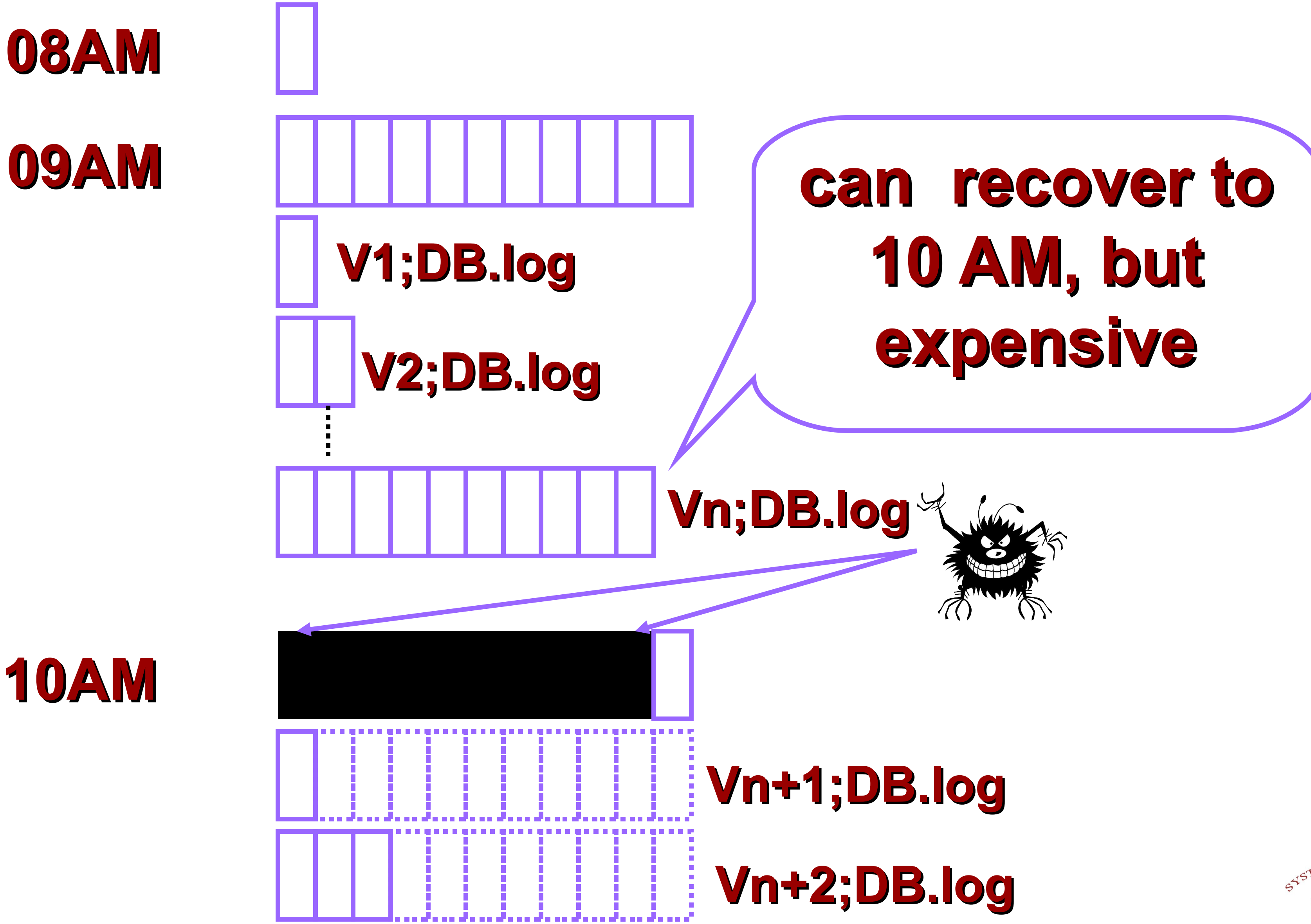


08AM
open DB.log
09AM
Write tx
10AM
11AM
Write tx
12PM
Detect Corruption

Open-close


08AM
09AM
V1;DB.log
Can only recover to 8 AM
10AM
12PM
Detect Corruption

Version-on-every write



Goal

Combine versioning and causality, taking advantage of causality information to create versions at just the right time

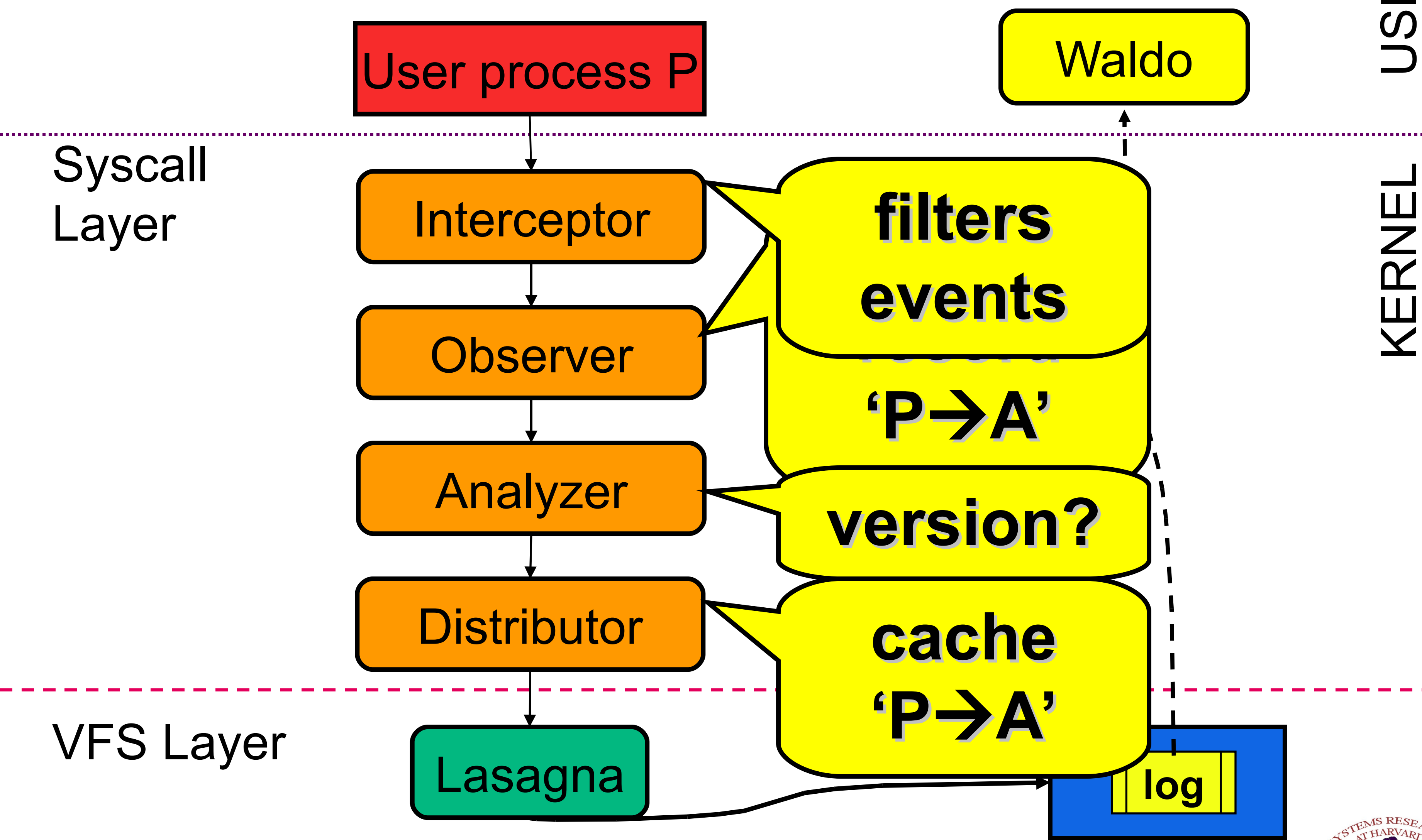
Contributions

- Two algorithms that create **useful** versions
  - Cycle Avoidance
  - Graph Finesse
- Evaluate efficacy and efficiency of these two algorithms in the context of versioning

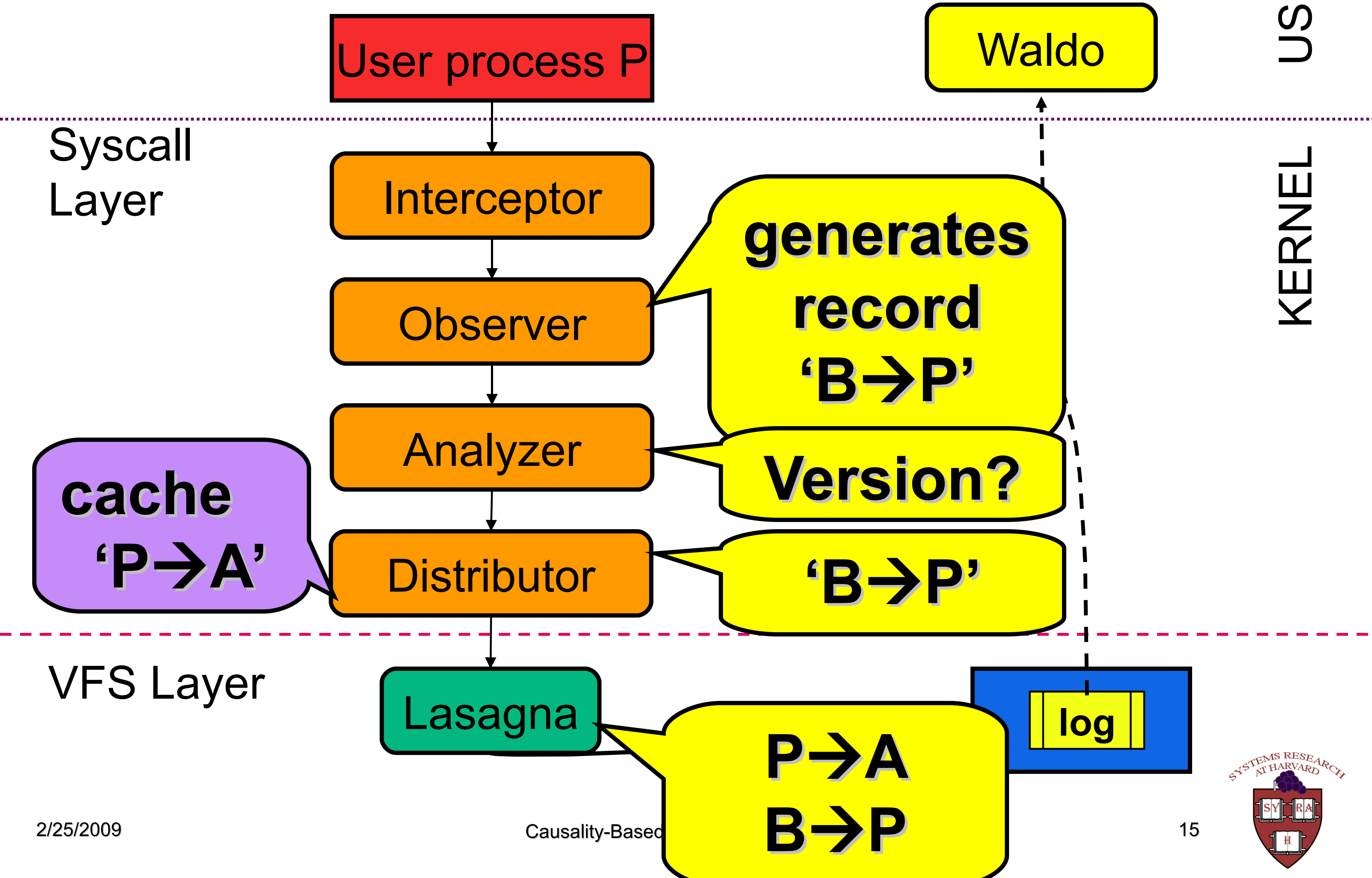
Outline

- Introduction
- **Background on PASS**
- Versioning Algorithms
- Implementation
- Evaluation
- Conclusion

PASS Architecture: P reads A



PASS Architecture: P writes B



Outline

- Introduction
- Background on PASS
- **Versioning Algorithms**
- Implementation
- Evaluation
- Conclusion

Intuition for new algorithms

- The creation of a cycle is an indicator that a version created at that instant could be useful later
- Cycles are violations of causality
  - Implies that past depends on future!

Open-Close Versioning

1. On the last close of a file, issue a “freeze” operation
   - Freeze declares end of a version
2. The next open and write triggers a new version

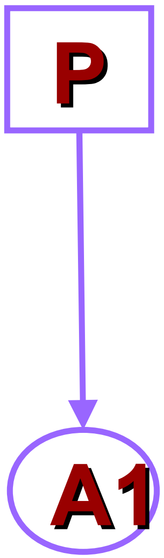
Example scenario

Time

P	Q
read A	
	read B
write B	
	write A
read A	
	read B

Each read/write is enclosed by an open and close

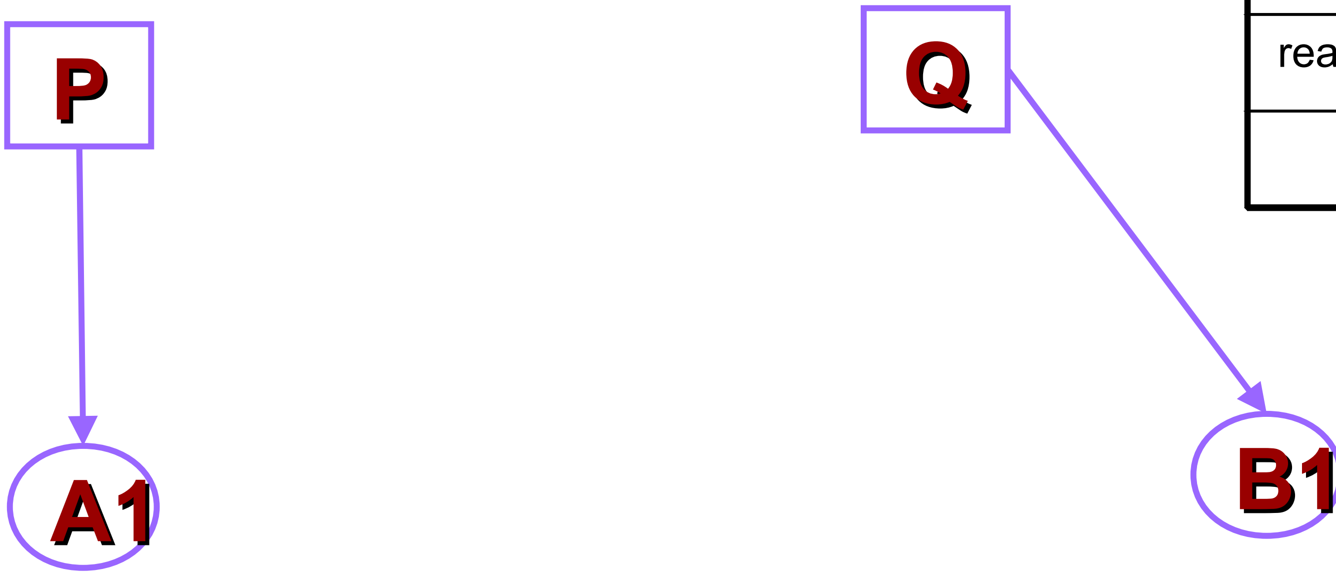
Open-Close



P	Q
read A	
	read B
write B	
	write A
read A	
	read B

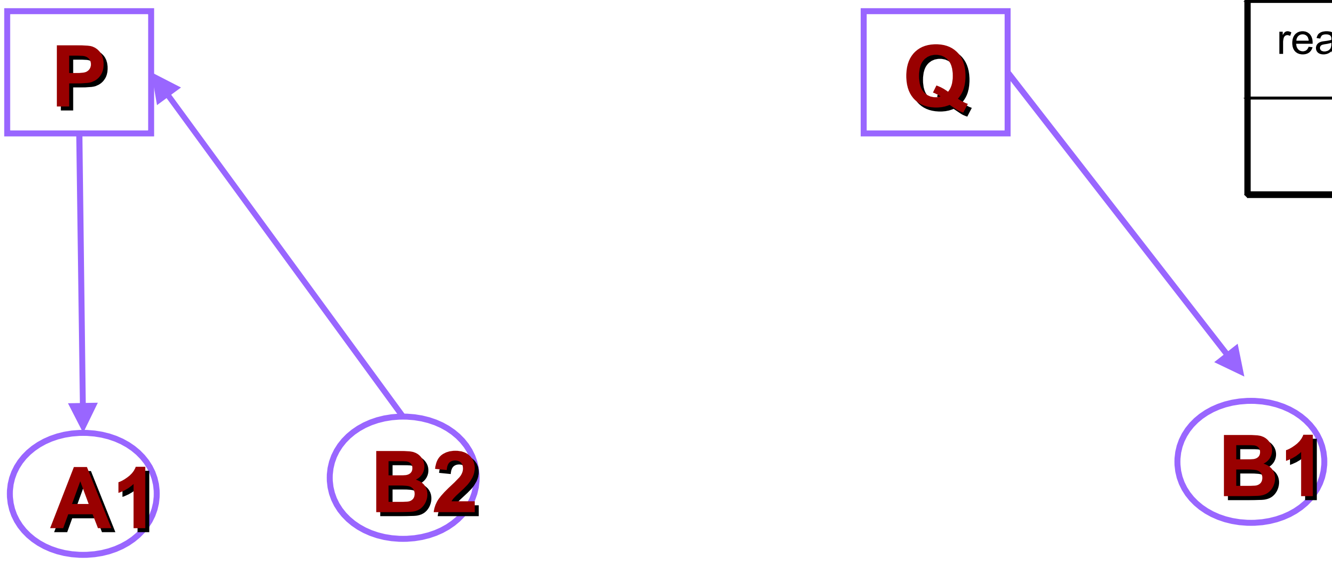
Open-Close

P	Q
read A	
	read B
write B	
	write A
read A	
	read B

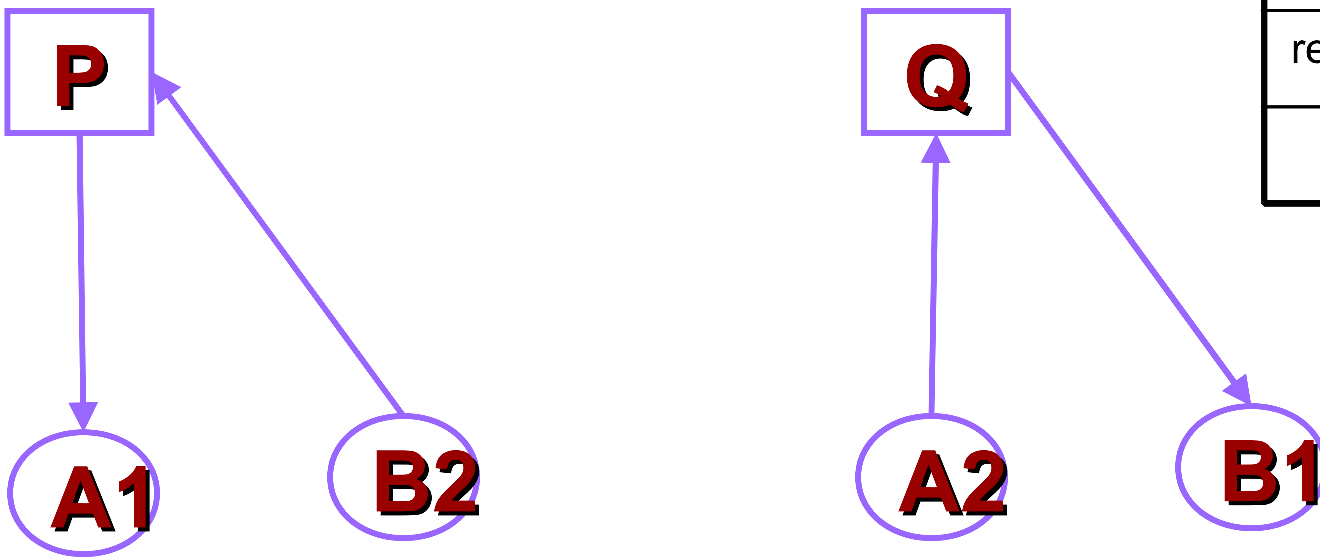


Open-Close

P	Q
read A	
	read B
write B	
	write A
read A	
	read B



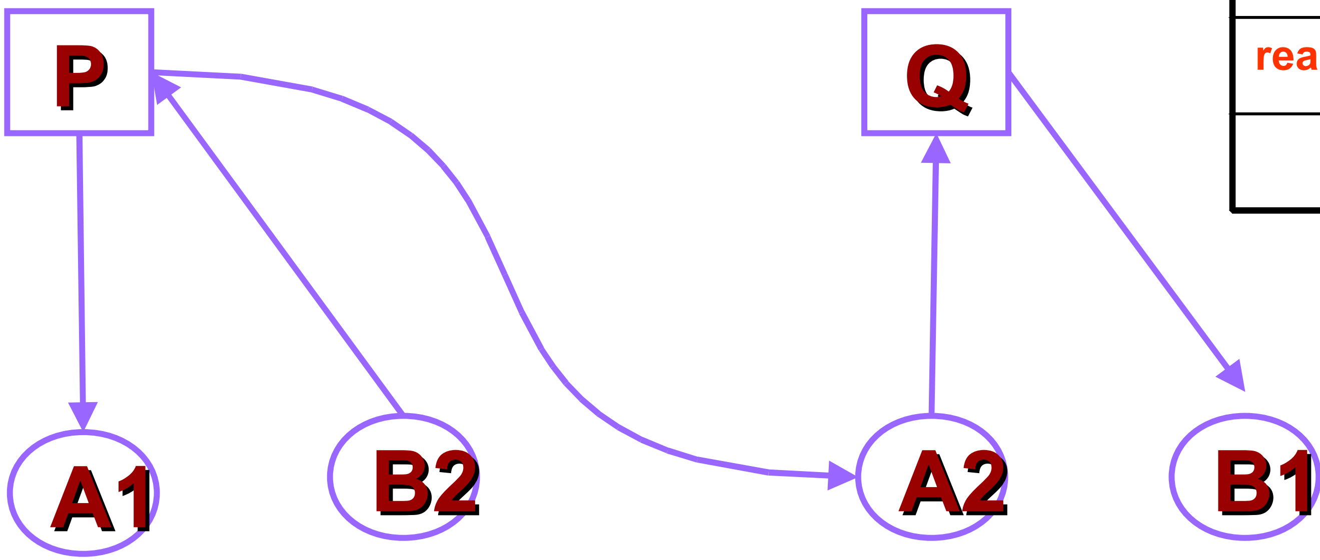
Open-Close



P	Q
read A	
	read B
write B	
	write A
read A	
	read B

Open-Close

P	Q
read A	
	read B
write B	
	write A
read A	
	read B

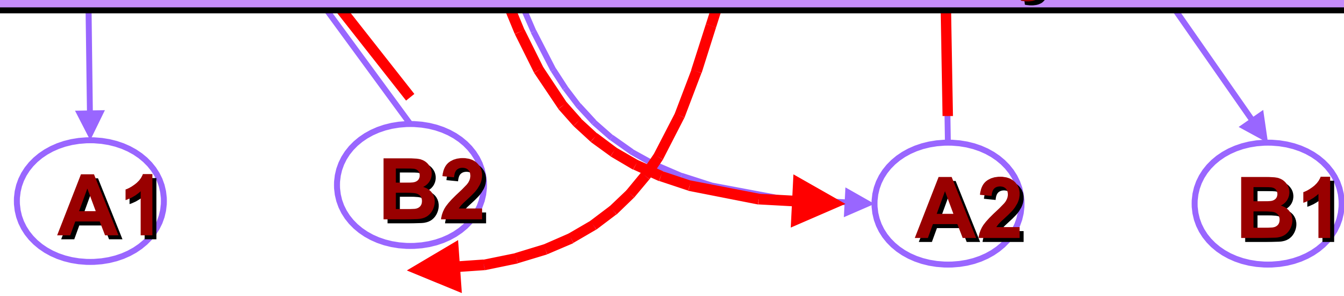


Open-Close

P	Q
read A	
	read B
write B	
	write A
	ead B

Open-Close allows cycles to happen.

Violates Causality



Version-on-every write

- Pros:
  - Preserves causality: there are no cycles
    - Every read creates a new version of the process
    - Every write creates a new version of the file
  - There are no duplicates either
- Disadvantage: most versions are unnecessary

Cycle Avoidance Algorithm

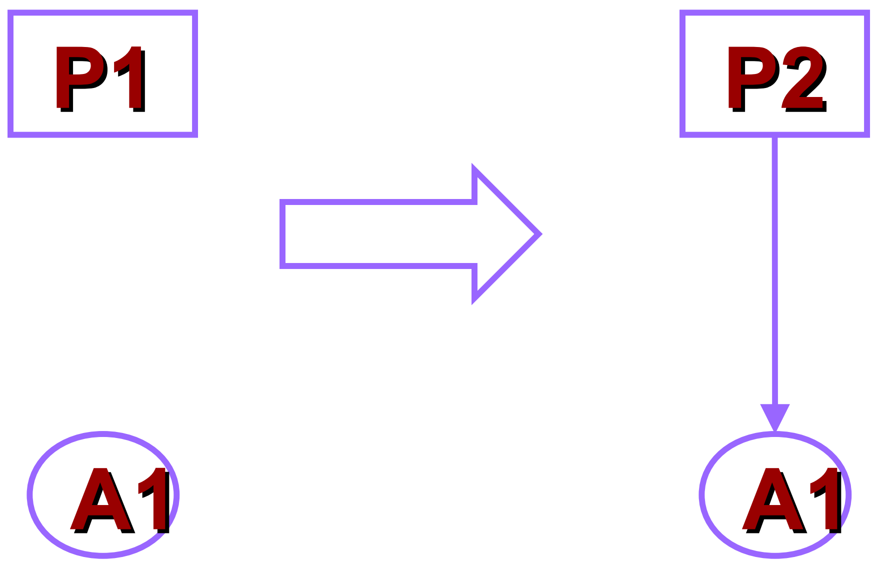
- Preserves Causality by avoiding cycles
- Uses local per-object information to make decisions
- Similar to the timestamp ordering in databases
- Intuition:

  Freeze an object when we add a dependency that does not previously exist, i.e., new causality

Cycle Avoidance Example

- On receiving record A1 → B2
  - If no B in A’s history, then freeze A
  - Else if B in A’s history, then
    - If A’s history has B2, discard record (duplicate)
    - If A’s history has B3 (version > 2), discard record
    - If A’s history has B1 (version < 2), freeze A

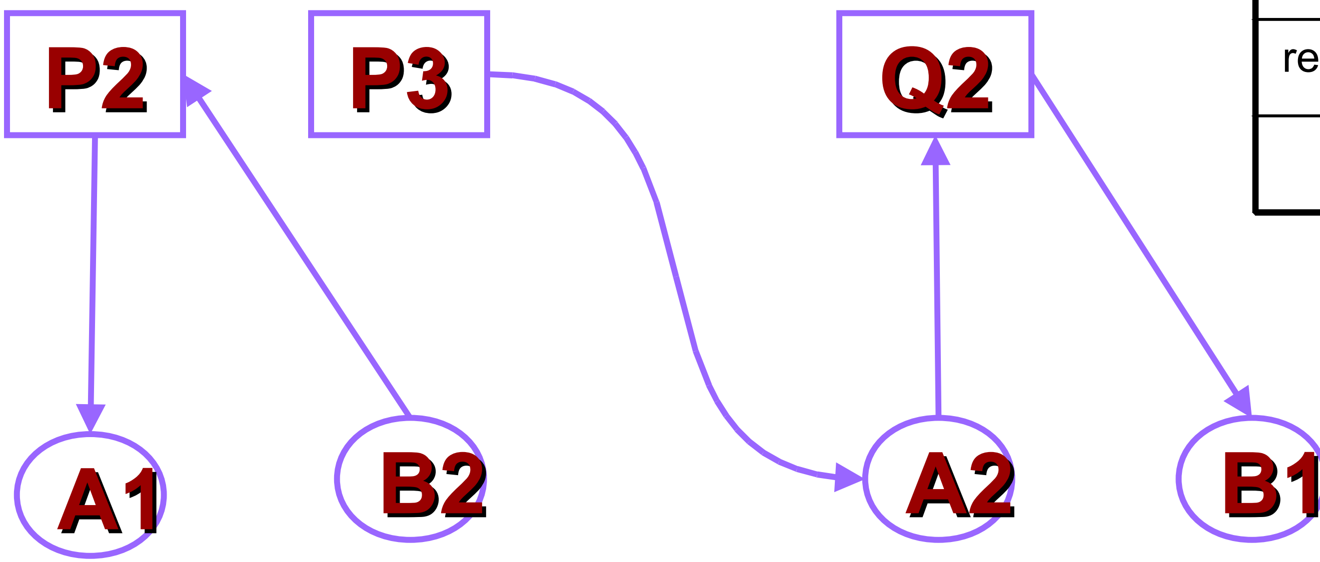
Cycle Avoidance



P	Q
read A	
	read B
write B	
	write A
read A	
	read B

Cycle Avoidance

P	Q
read A	
	read B
write B	
	write A
read A	
	read B

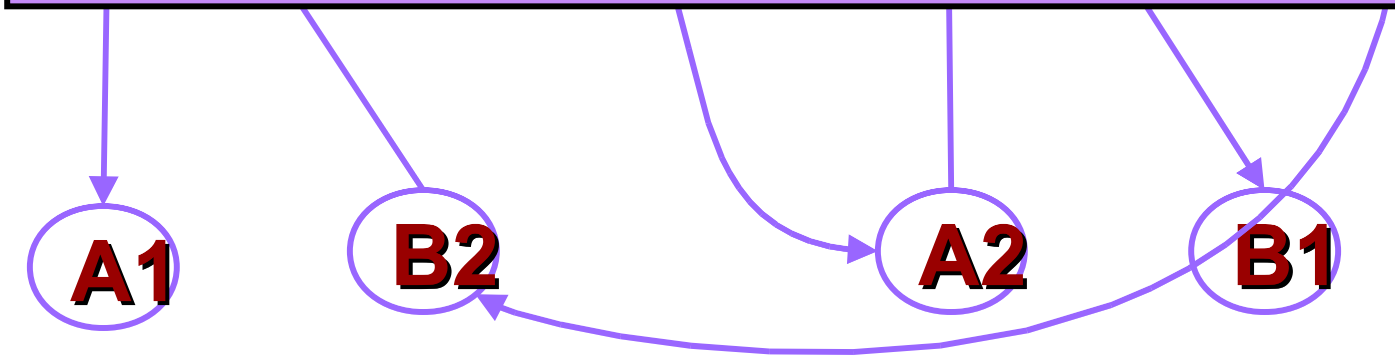


Cycle Avoidance

P	Q
read A	
	read B
write B	
	write A
read A	
	read B

Cycle-Avoidance prevents cycles,

but creates more versions



Graph Finesse Algorithm

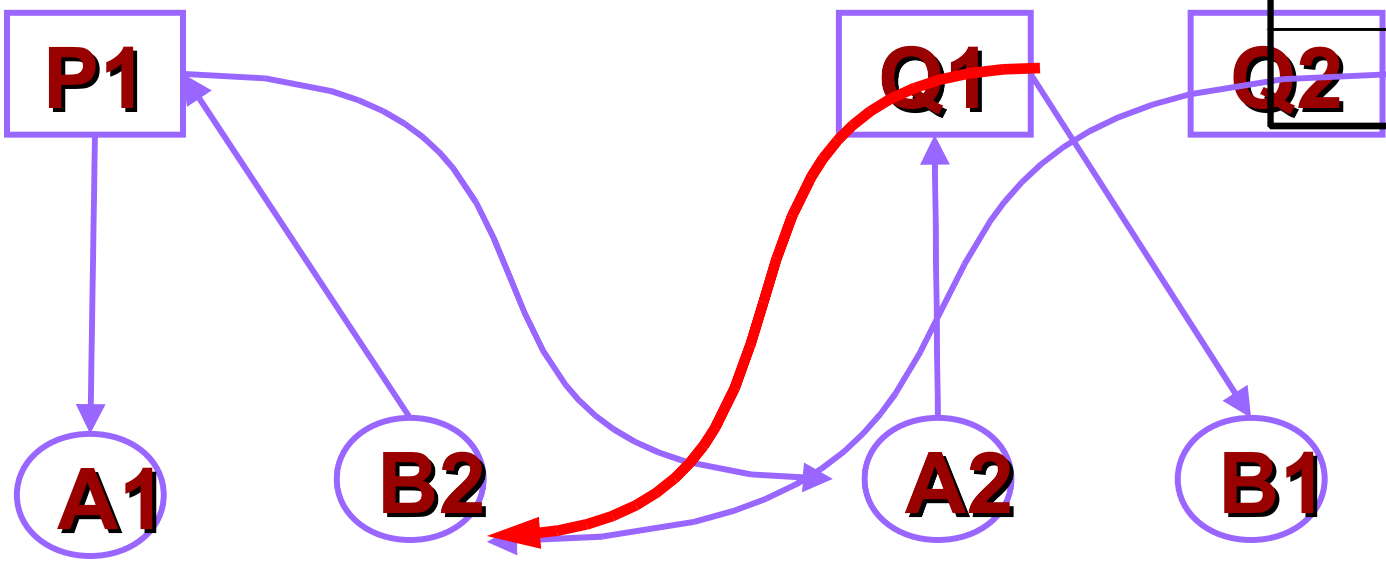
- Uses Global knowledge
- Intuition:
  - Check every new record against a global dependency graph.
  - If it forms a cycle, just freeze that one node
- Subsumes open-close algorithm

Graph Finesse Example

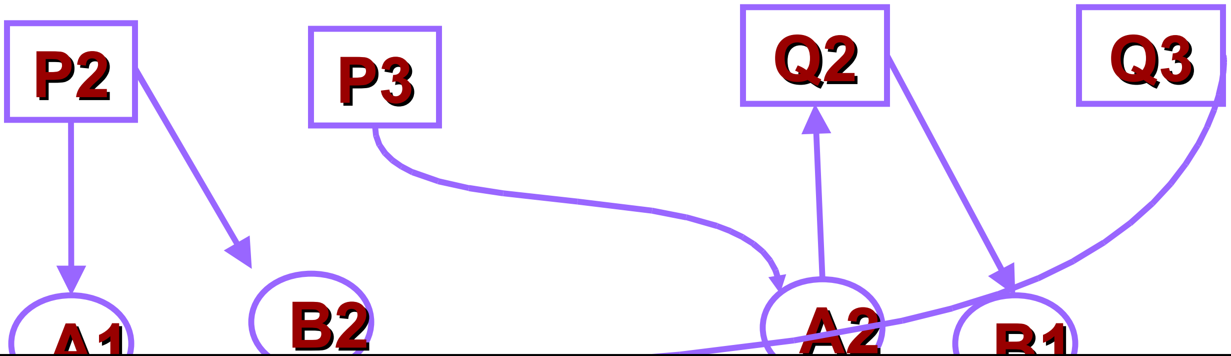
- On receiving record A1 → B2
  - If B2 is already in A's history, discard record
  - Else check for a path from B2 → A1
    - If yes, this a cycle, freeze A1 and change the record to A2→B2
    - If no cycle, add A1 → B2 to the graph

Graph Finesse

P	Q
read A	
	read B
write B	
	write A
read A	
	read B



Cycle Avoidance

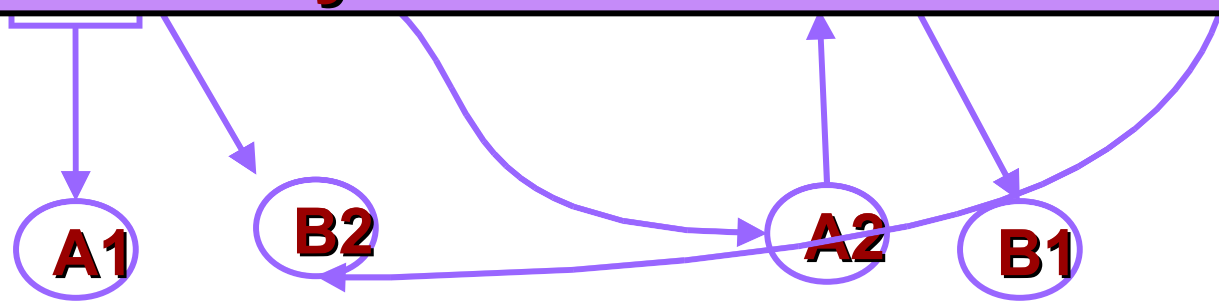


Gra

Graph Finesse prevents cycles.

But creates fewer versions than

Cycle Avoidance



Cycle Avoidance	Graph Finesse
Uses Local state	Uses Global state
Creates a few un-necessary versions	Creates fewer versions
Has lower runtime overhead	Can have high run-time overheads

Outline

- Introduction
- Background on PASS
- Versioning Algorithms
- **Implementation**
- Evaluation
- Conclusion

Implementation

- Implemented on Linux 2.6.23.17
- Lasagna is a stackable file system derived from eCryptfs
- Versioning file system
  - Redo log that keeps track of file versioning (deltas)
  - Redo log for directory modifications (deltas)

Outline

- Introduction
- Background on PASS
- Versioning Algorithms
- Implementation
- **Evaluation**
- Conclusion

Evaluation Goals

- What are the run-time overheads a user might see?
- What are the space overheads?
- How do the algorithms compare during recovery?

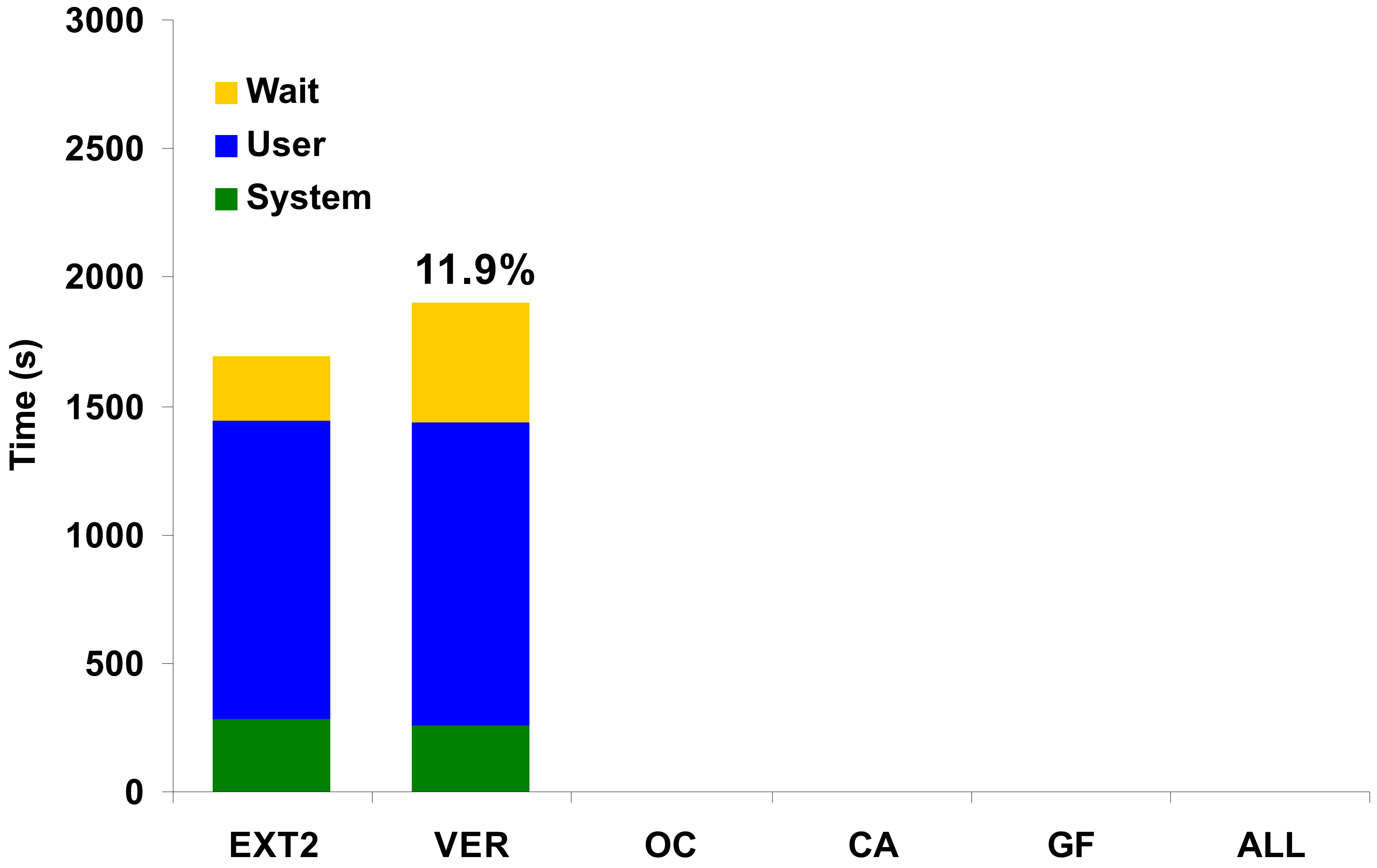
Test platform

- Linux 2.6.23.17
- 3Ghz Pentium 4
- 512MB of RAM
- 80GB 7200 RPM IDE Disk
- All results are averages of 5 runs
  - Less than 5% Std. Dev.

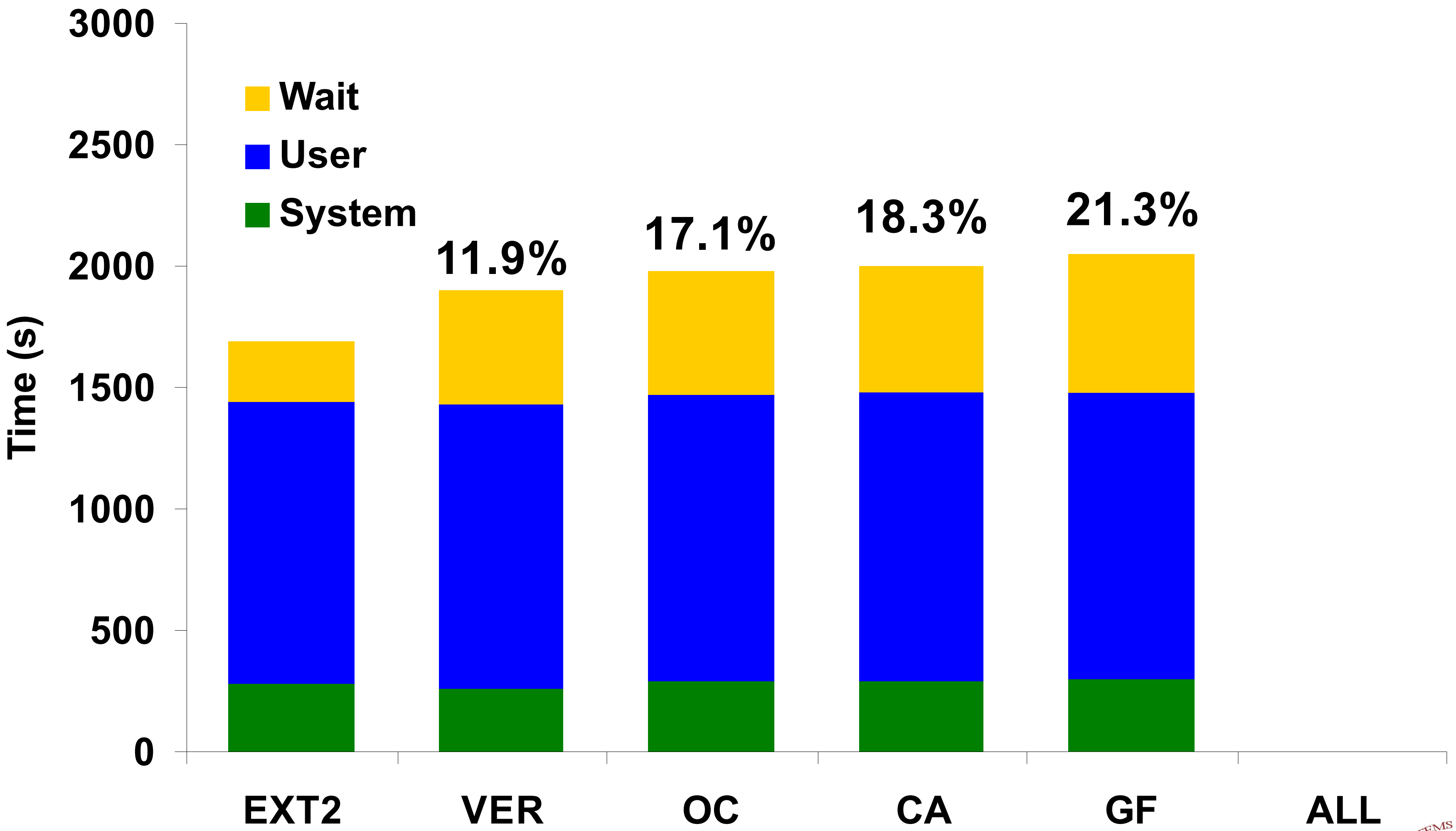
Modes

- Without causal data
  - Ext2: Baseline (Lasagna was stacked on Ext2)
  - VER: plain versioning (open-close)
- With causal data
  - OC: open-close
  - **CA: Cycle-Avoidance**
  - **GF: Graph-Finesse**
  - ALL: Version-on-every write

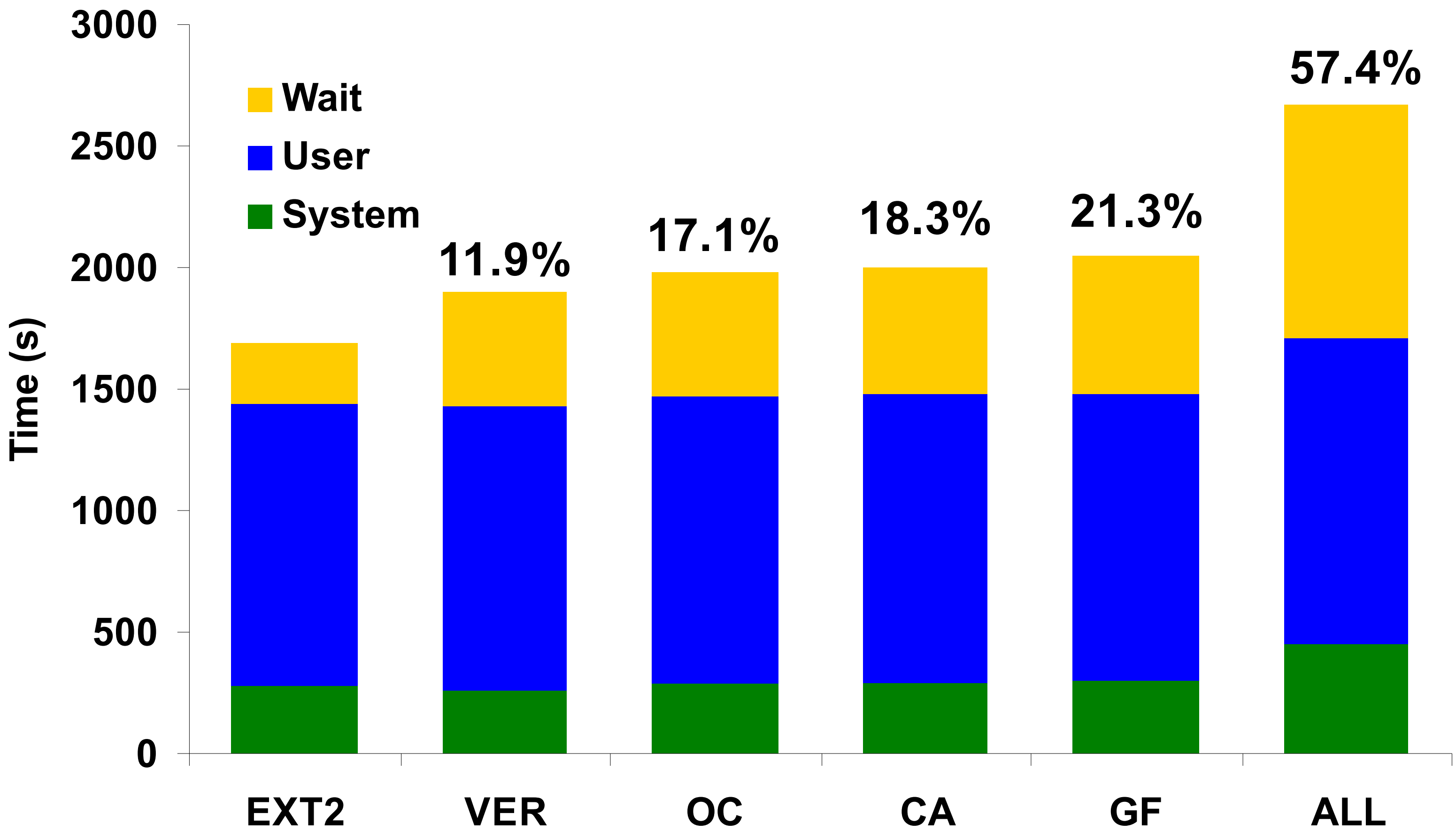
Linux Compile: Elapsed Time



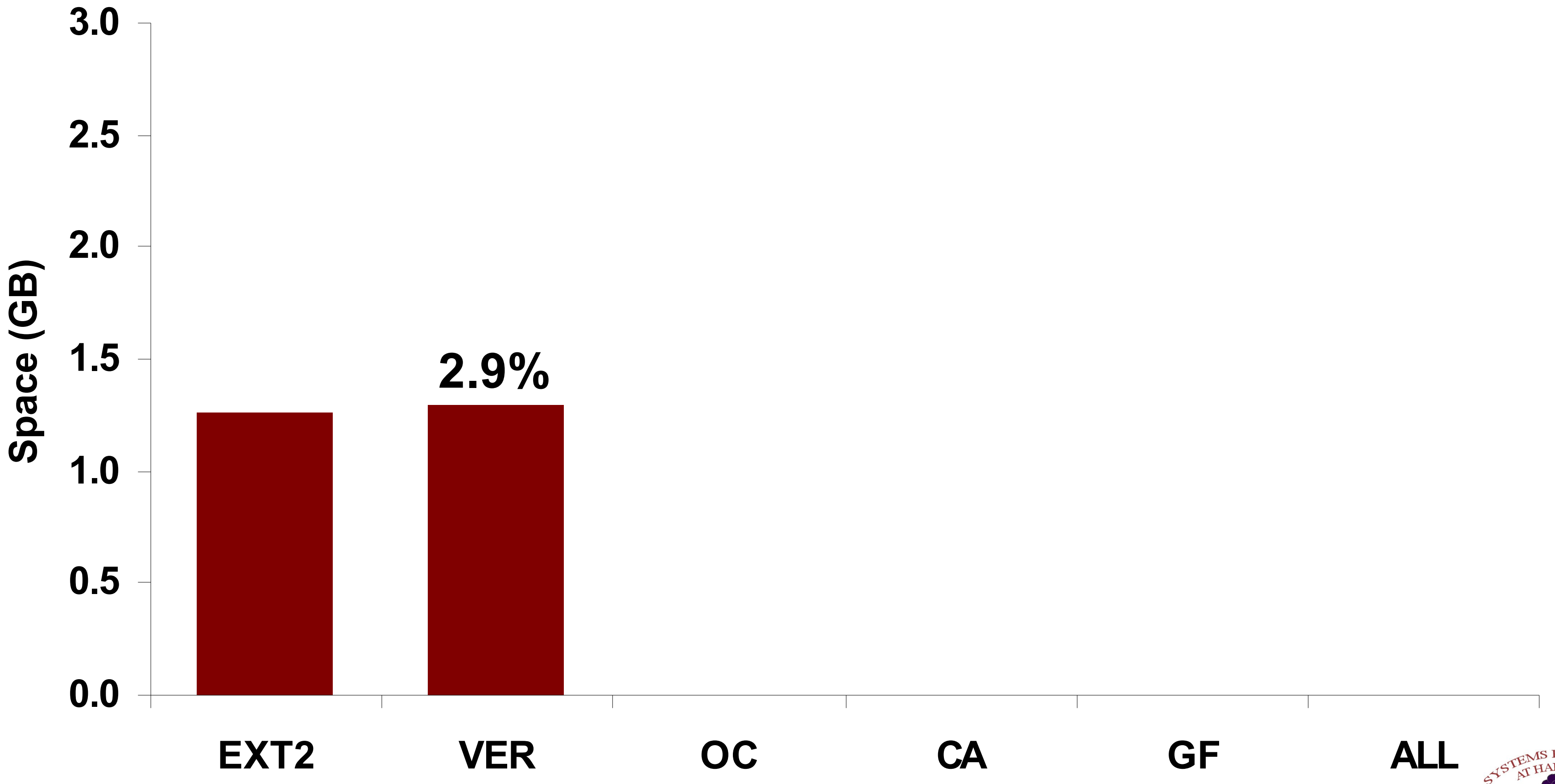
Linux Compile: Elapsed Time



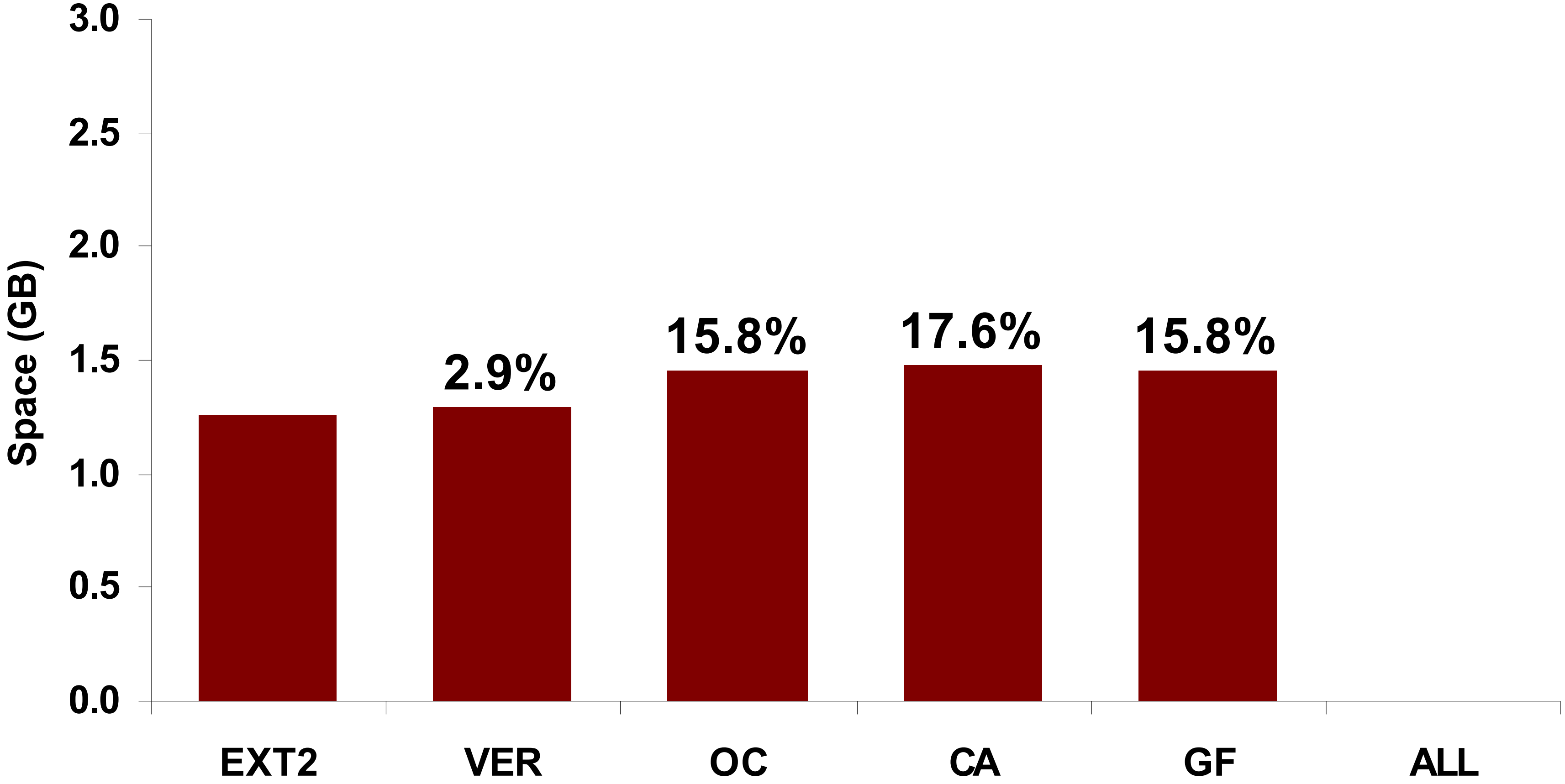
Linux Compile: Elapsed Time



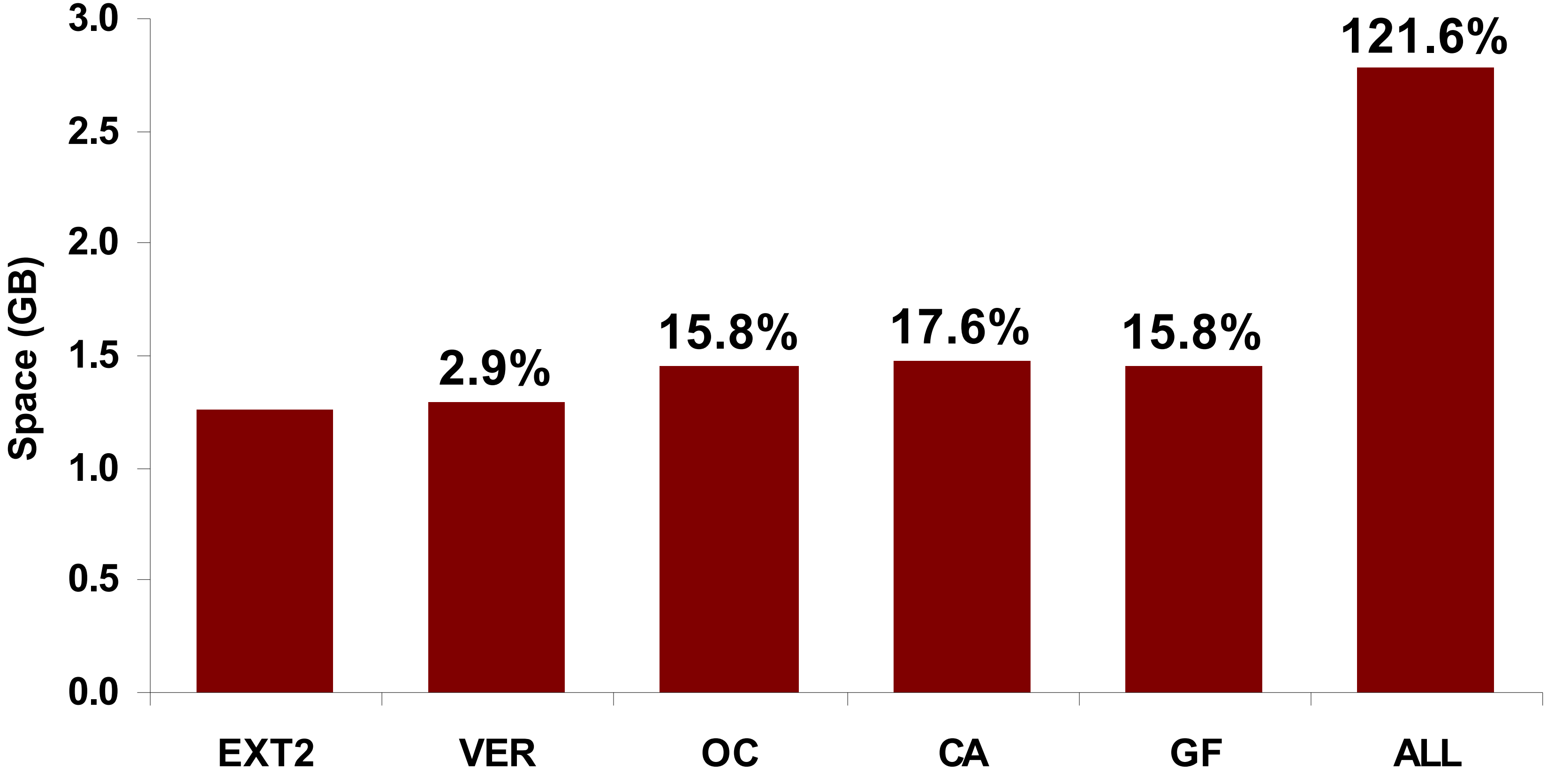
Linux Compile: Space Overheads



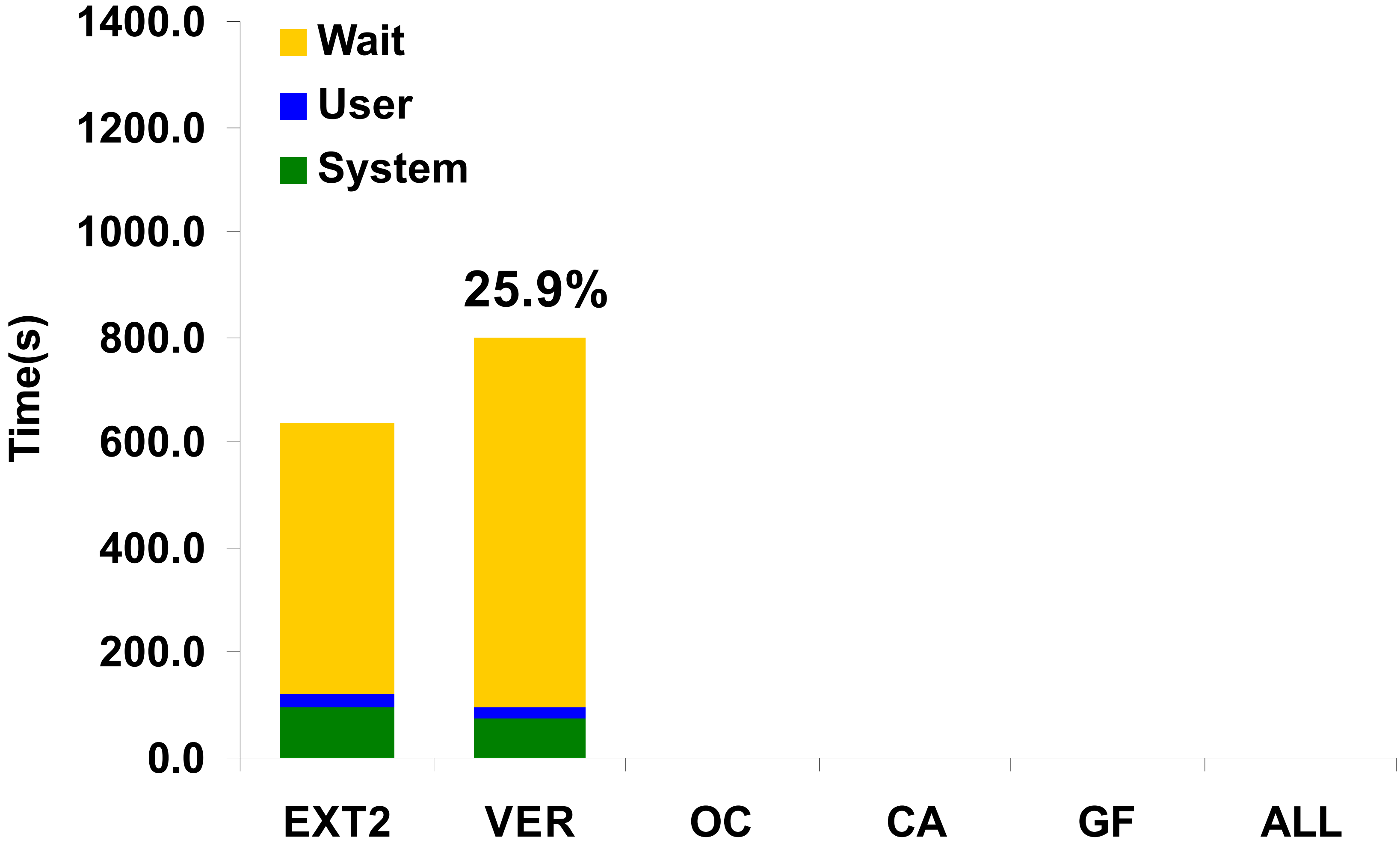
Linux Compile: Space Overheads



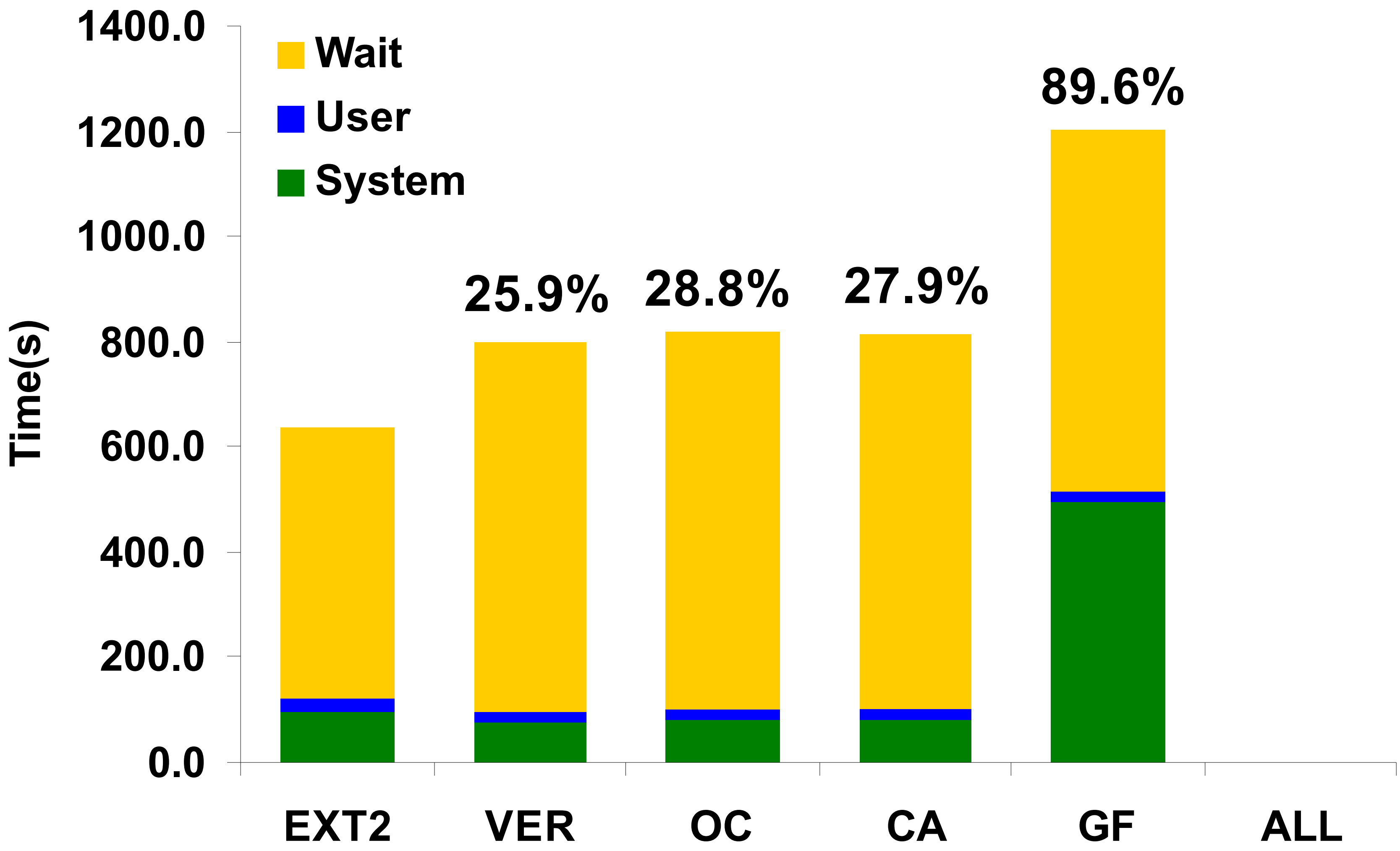
Linux Compile: Space Overheads



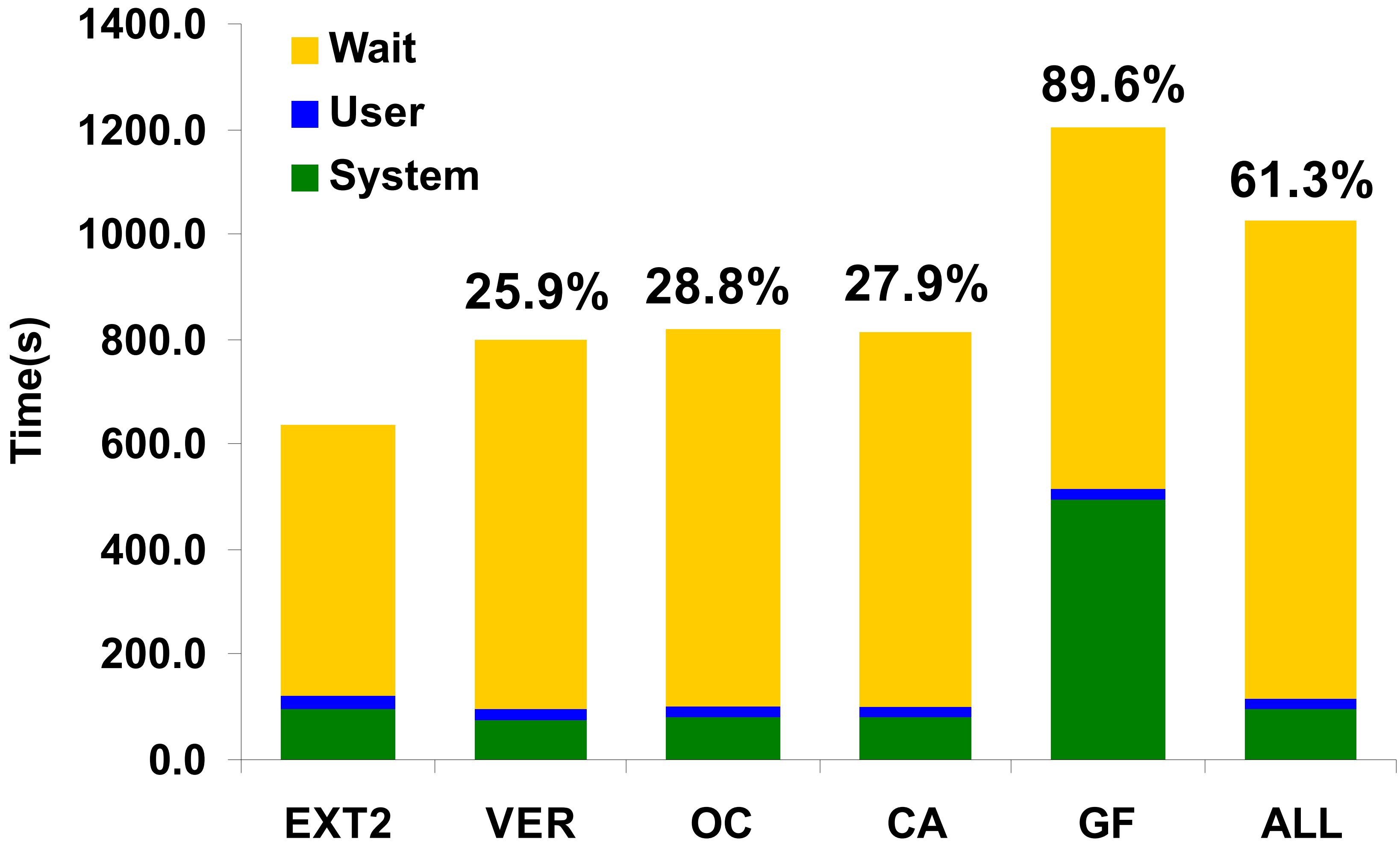
Mercurial Activity: Elapsed Time



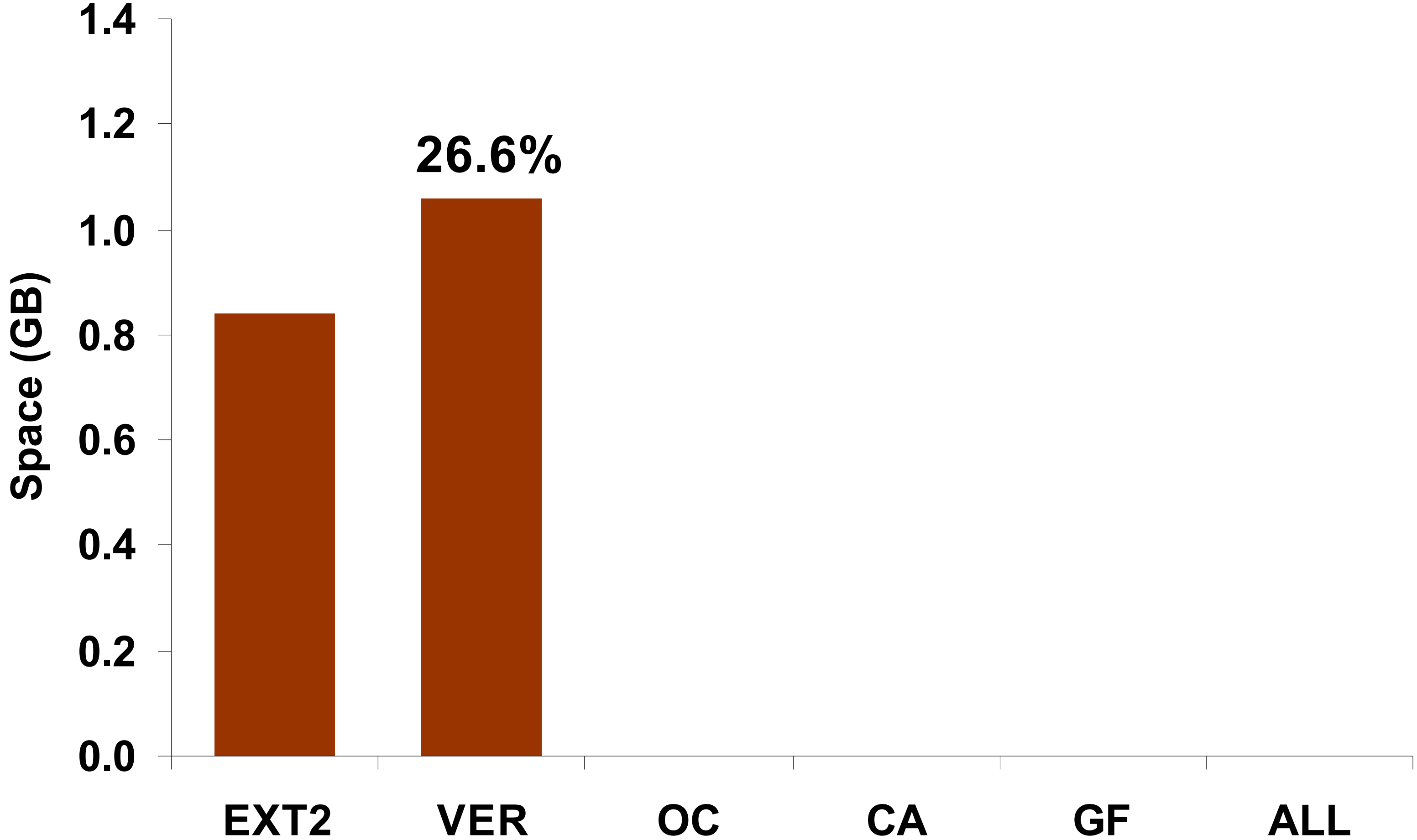
Mercurial Activity: Elapsed Time



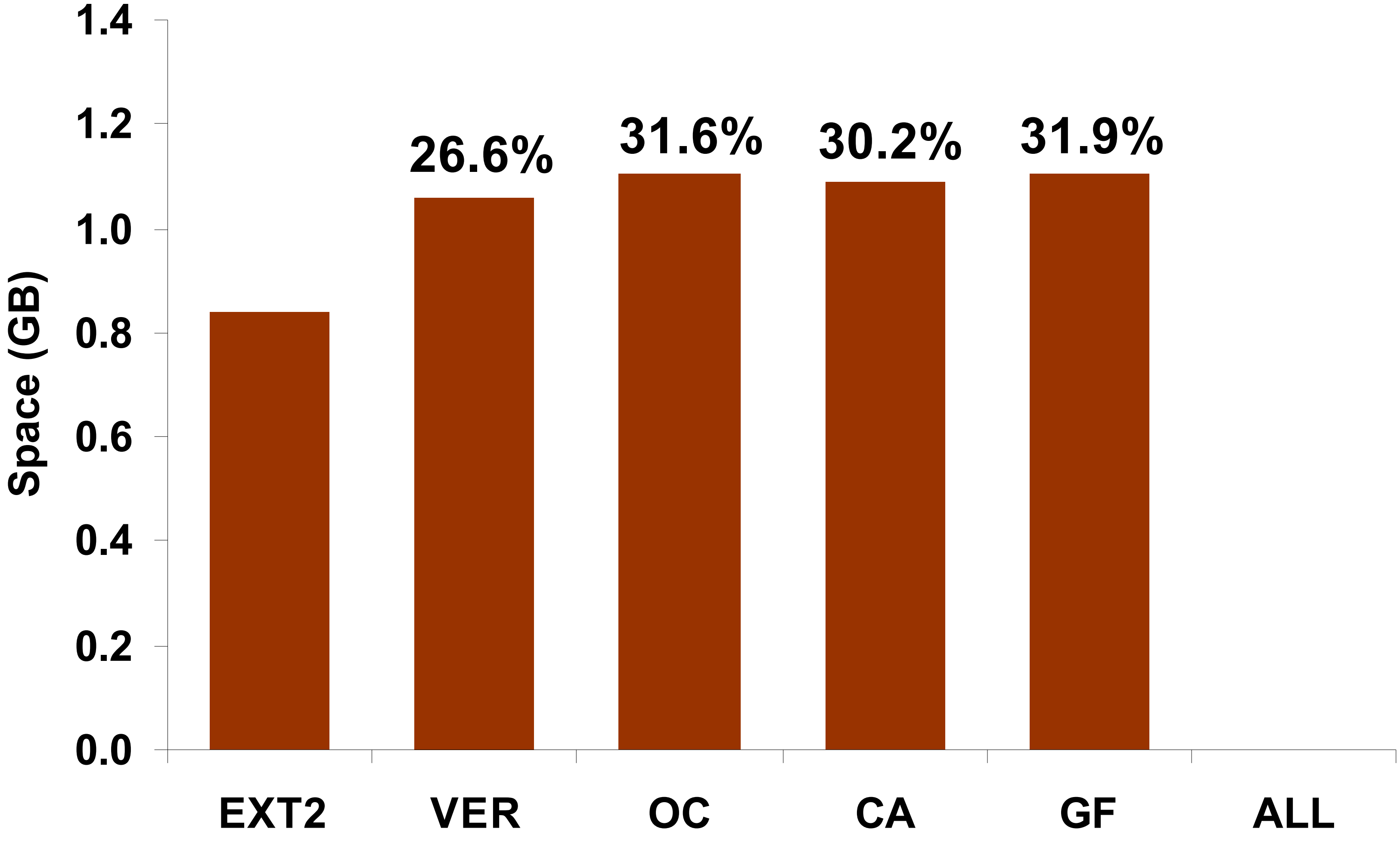
Mercurial Activity: Elapsed Time



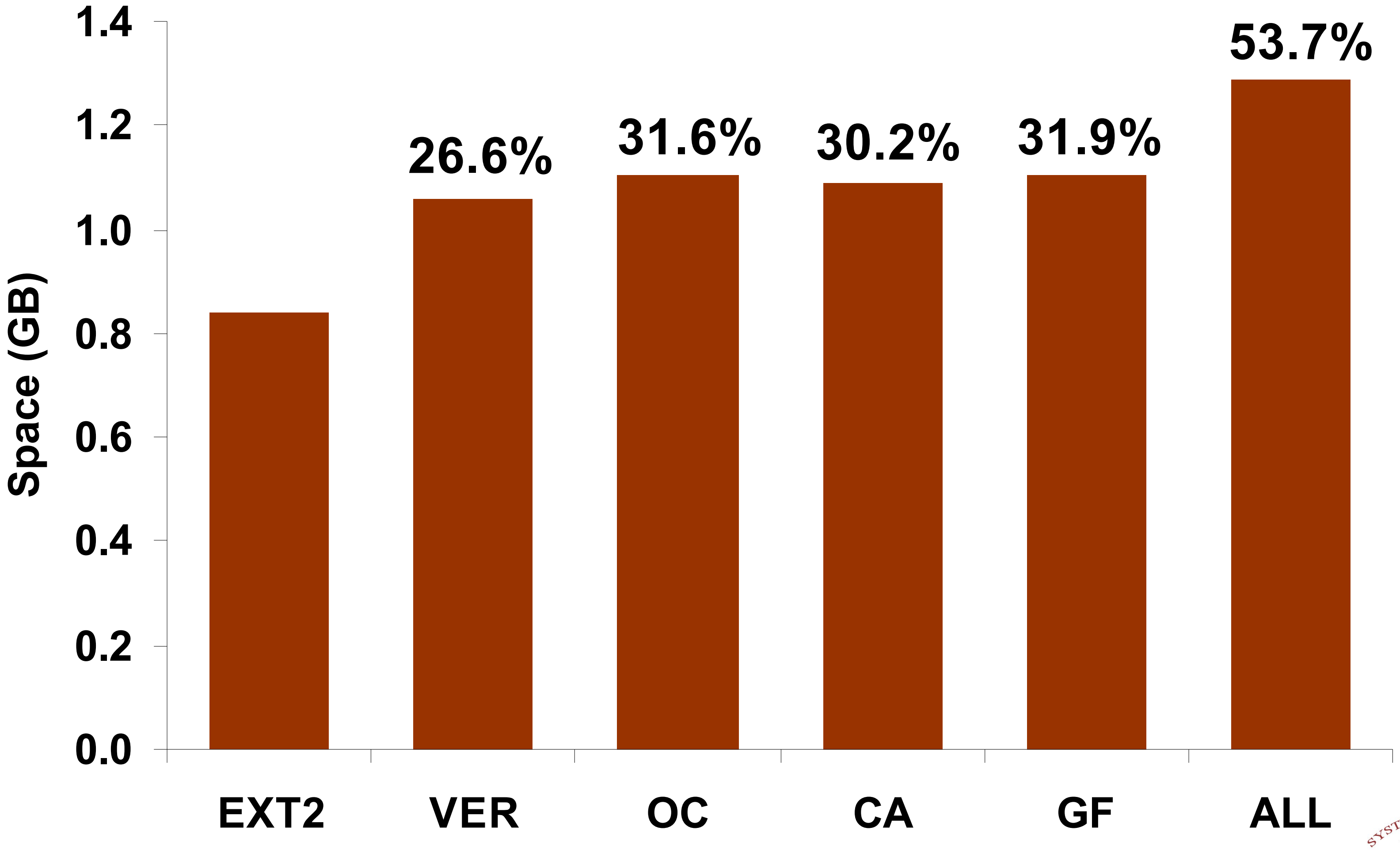
Mercurial Activity: Space Overheads



Mercurial Activity: Space Overheads



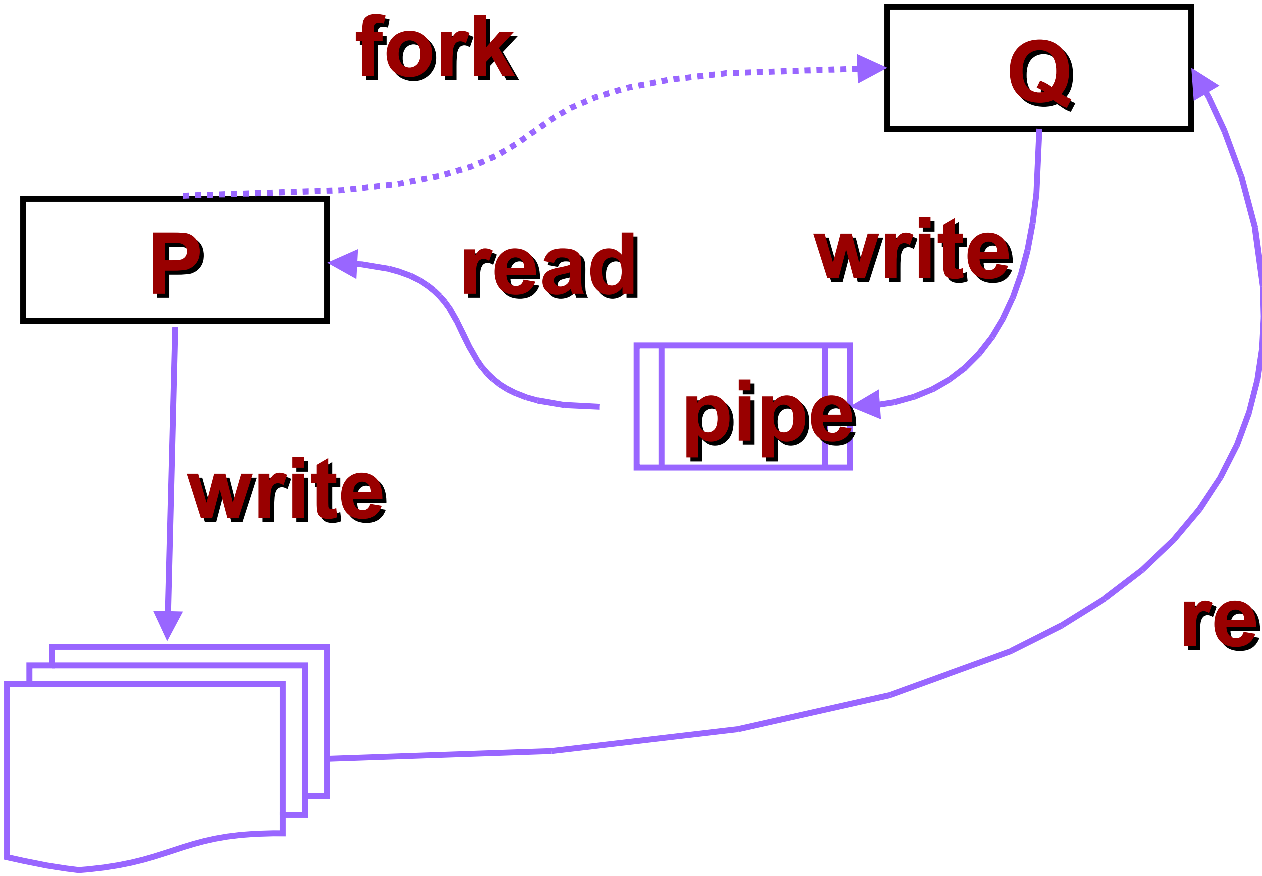
Mercurial Activity: Space Overheads



Recovery Benchmarks

- How the algorithms perform in the scenario where open close is not sufficient
- Microbenchmark
  - Models the apache split-log scenario

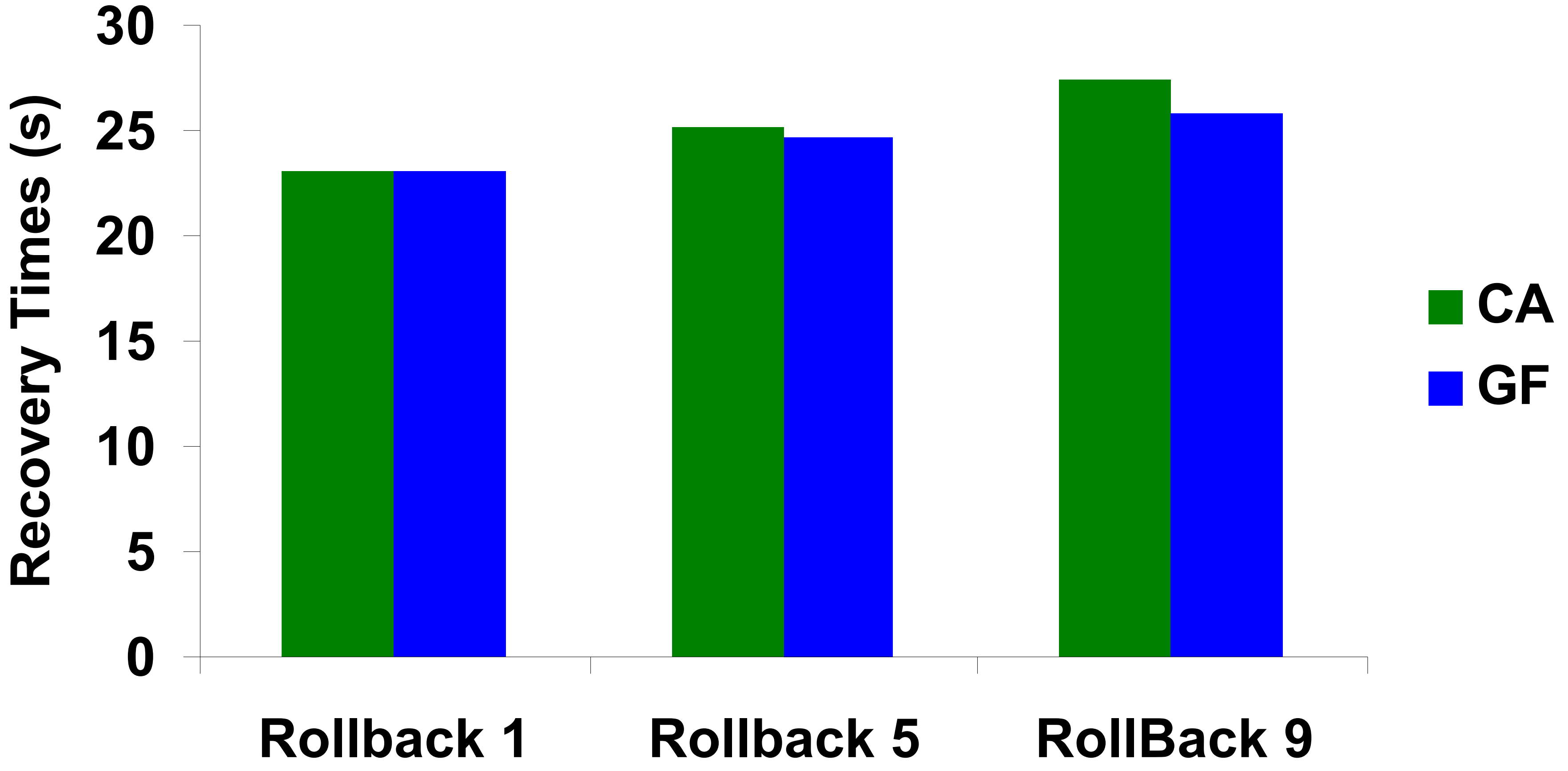
Recovery MicroBenchmark



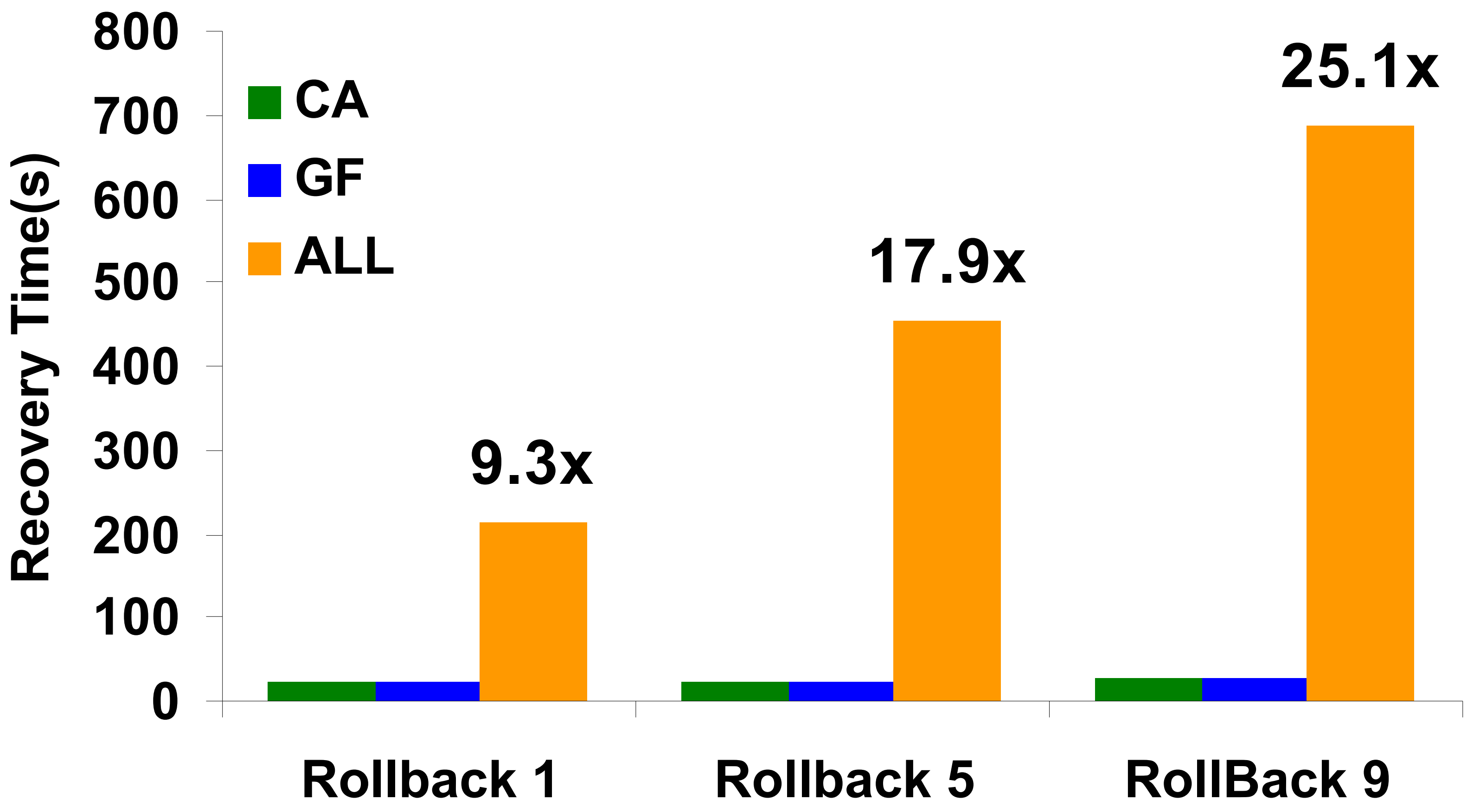
Recovery Microbenchmark: Space Util.

	Causal Data	Version Data
OC	60KB	12KB
CA	176KB	470.5MB
GF	184KB	470.5MB
ALL	76.9MB	1.97GB

Recovery Times



Recovery Times



Conclusions

- Combining Versioning and Causality enables novel functionality
- New algorithms for Causal Versioning
  - Cycle Avoidance
    - Comparable to open-close
    - May create more versions
  - Graph Finesse
    - Provides greater control on versioning
    - Can be inefficient at times

Questions?

Contact:

pass@eecs.harvard.edu

www.eecs.harvard.edu/syrah/pass