Speculative Execution In Distributed File System and External Synchrony

Edmund B.Nightingale, Kaushik Veeraraghavan
Peter Chen, Jason Flinn

Presented by Han Wang
Slides based on the SOSP and OSDI presentations

Consistency

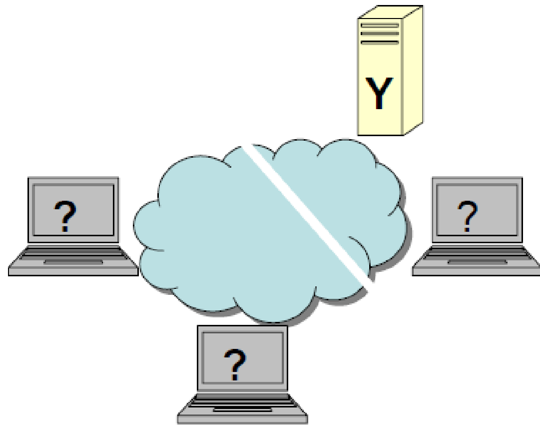
Availability

Partition Tolerance

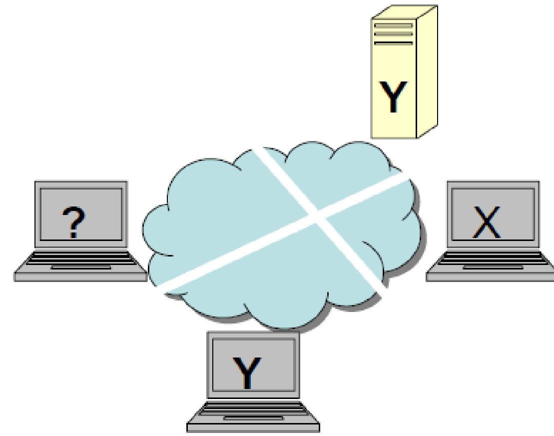
*“ … consistency, availability, and partition tolerance.
It is impossible to achieve all three. “*
-- Gilbert and Lynn, MIT

“So in reality, there are only two types of systems: CP/CA and AP”
-- Daniel Abadi, Yale

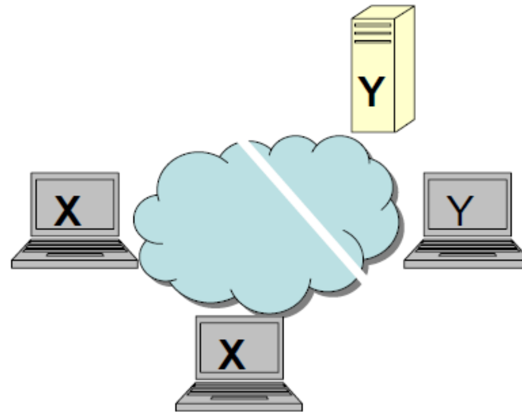
“There is no ‘free lunch’ with distributed data.”
-- Anonymous, HP



CP: Lack Availability



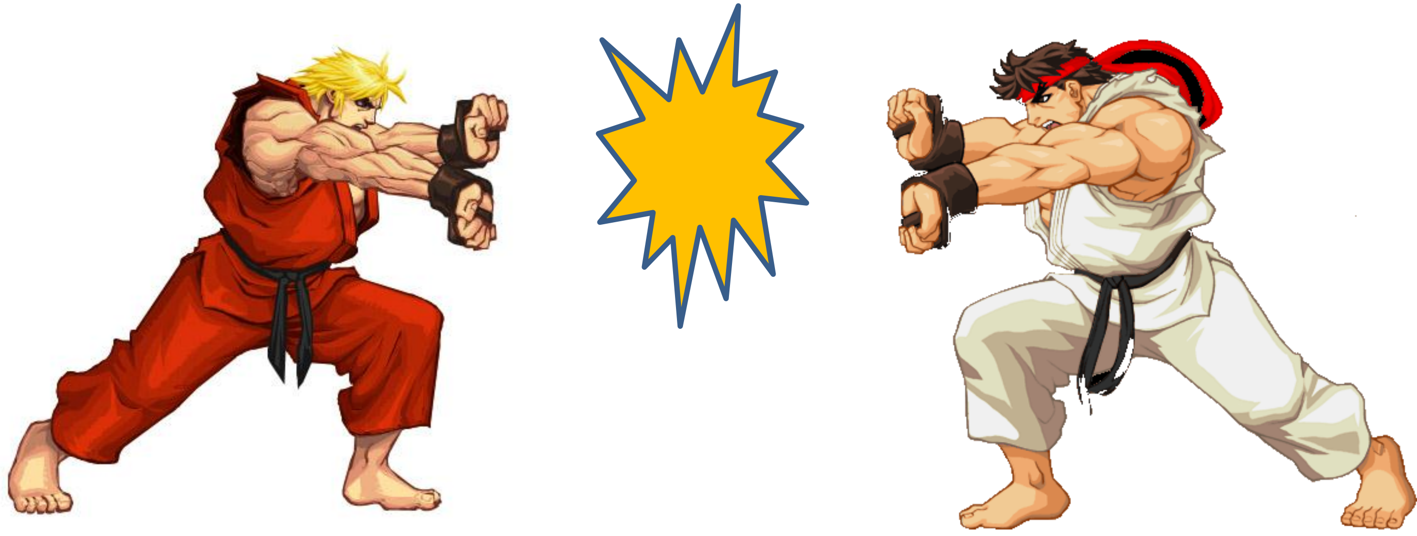
CA: Lack Partition Tolerance



AP: Lack Consistency

Synchrony

Asynchrony



Synchrony Asynchrony

synchronous abstractions:

strong reliability guarantees

but are slow

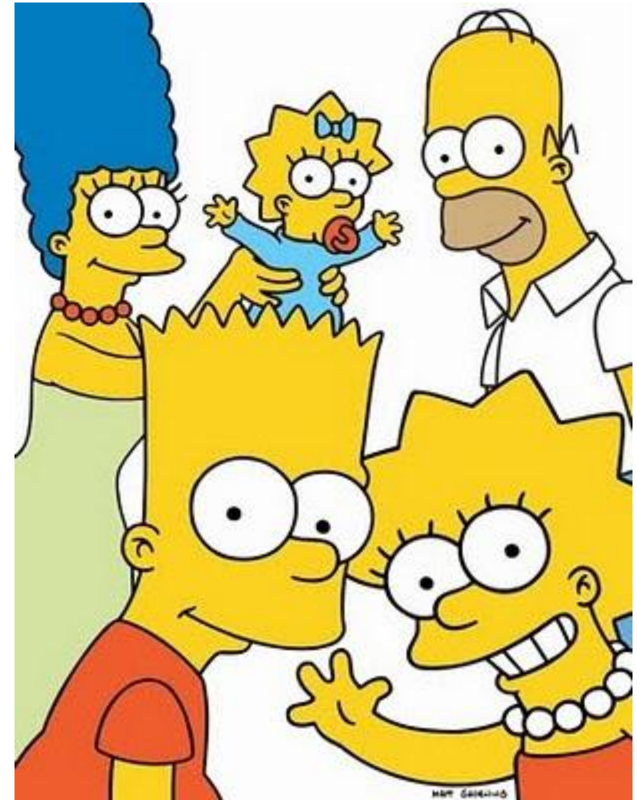
asynchronous counterparts:

relax reliability guarantees

reasonable performance

External Synchrony

- provide the reliability and simplicity of a synchronous abstraction
- approximate the performance of an asynchronous abstraction.



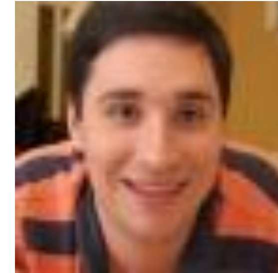
Rethink the Sync

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen and Jason Flinn

Speculative Execution in a Distributed File System

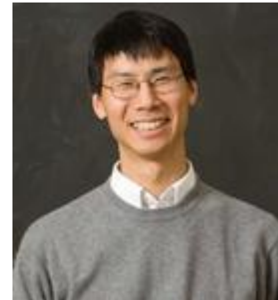
Edmund B. Nightingale, Peter M. Chen, and Jason Flinn

Authors



- Edmund B Nightingale
  - PhD from UMich (Jason Flinn)
  - Microsoft Research
  - Best Paper Award (OSDI 2006)

- Kaushik Veeraraghavan
  - PhD Student in Umich (Jason Flinn)
  - Best Paper Award (FAST 2010, ASPLOS 2011)



- Peter M Chen
  - PhD from Berkeley (David Patterson)
  - Faculty at UMich

- Jason Flinn
  - PhD from CMU (Mahadev Satyanarayanan)
  - Faculty at Umich

Rethink the Sync

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen and Jason Flinn

Speculative Execution in a Distributed File System

Edmund B. Nightingale, Peter M. Chen, and Jason Flinn

Idea

Example

Design

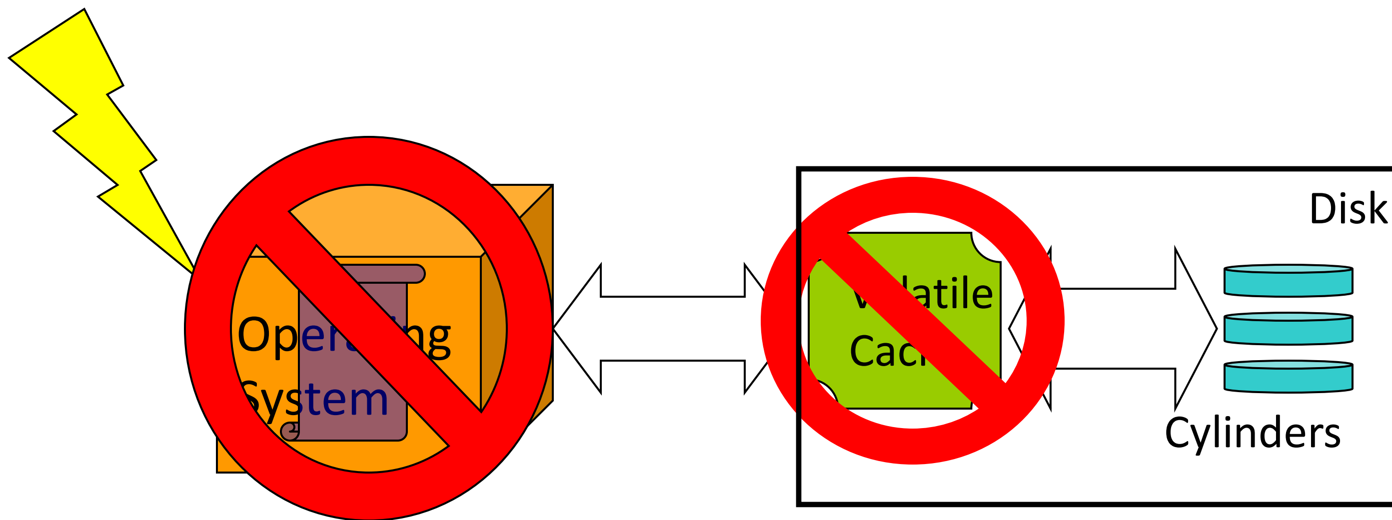
Evaluation

External Synchrony

- Question
  - How to improve both durability and performance for local file system?
- Two extremes
  - Synchronous IO
    - Easy to use
    - Guarantee ordering
  - Asynchronous IO
    - Fast

When a sync() is really async

- On sync() data written only to volatile cache
  - 10x performance penalty and data NOT safe

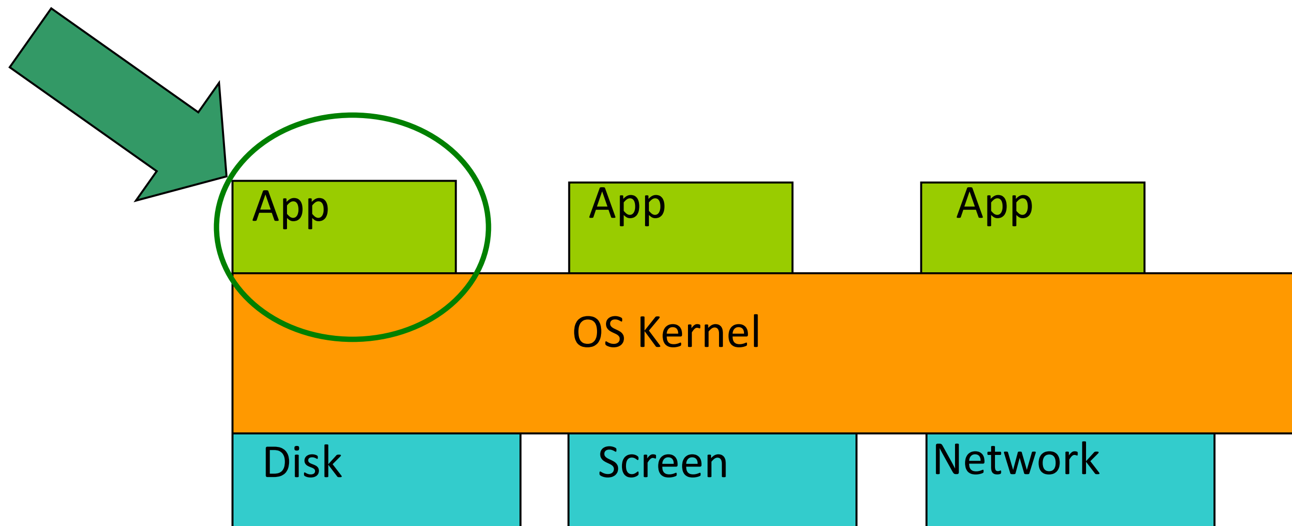


- 100x slower than asynchronous I/O if disable cache

From Nightingale's presentation

To whom are guarantees provided?

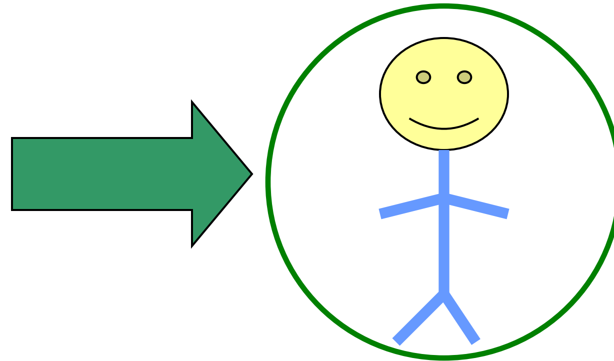
- Synchronous I/O definition:
  - Caller blocked until operation completes

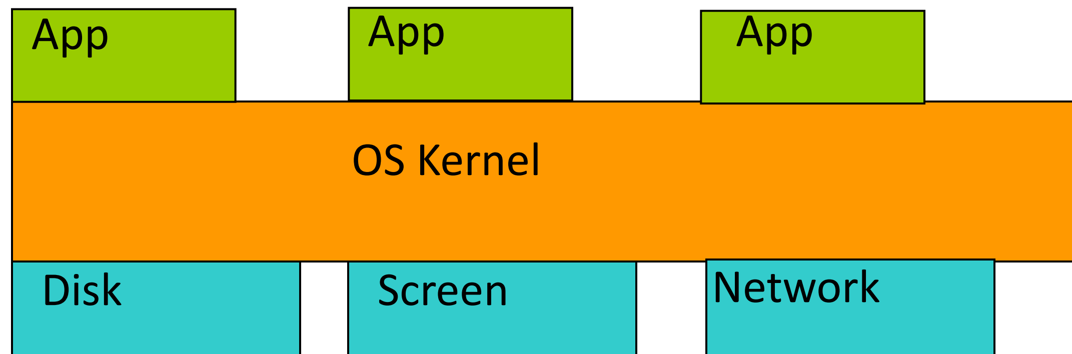


- Guarantee provided to application

From Nightingale's presentation

To whom are guarantees provided?





- Guarantee really provided to the user

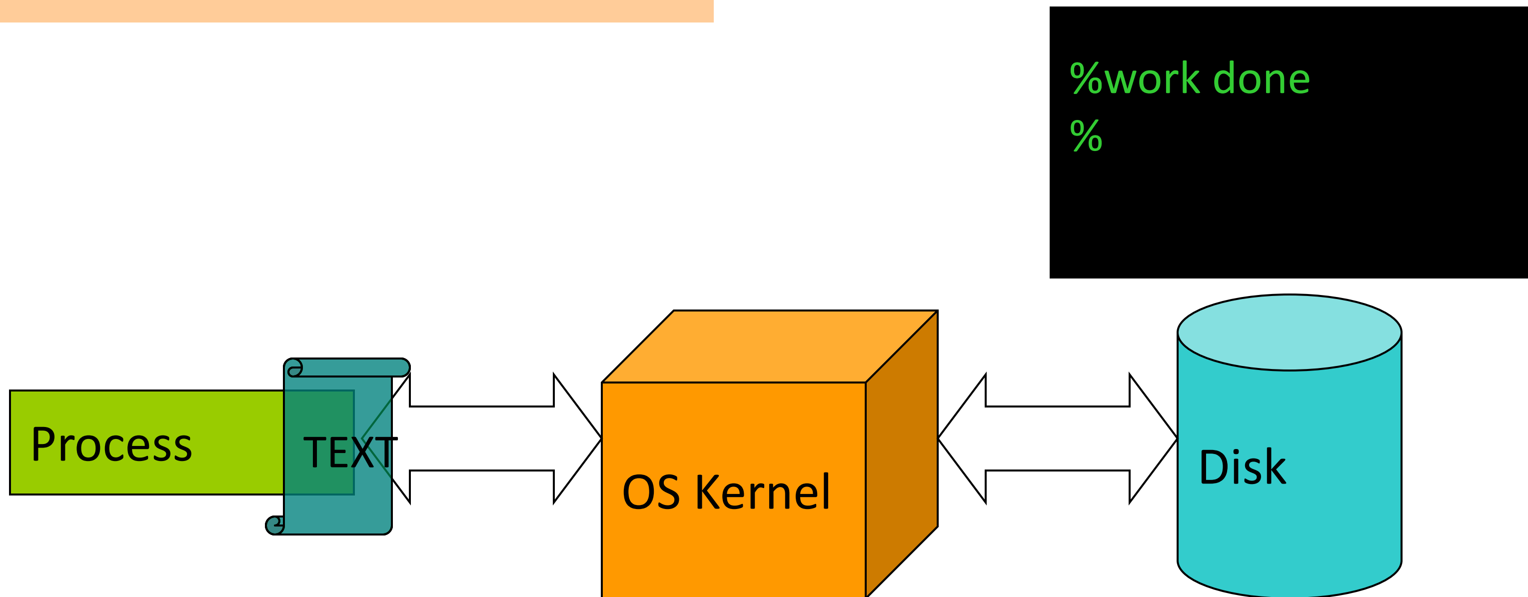
From Nightingale's presentation

Example: Synchronous I/O

```
101   write(buf_1);
102   write(buf_2);
103   print("work done");
104   foo();
```

Application blocks

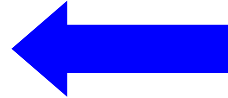
Application blocks



From Nightingale's presentation

Observing synchronous I/O

```
101   write(buf_1);
102   write(buf_2);
103   print("work done");
104   foo();
```



Depends on 1st write

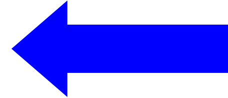
Depends on 1st & 2nd write

- Sync I/O externalizes output based on causal ordering
  - Enforces causal ordering by blocking an application
- External sync: Same causal ordering without blocking applications

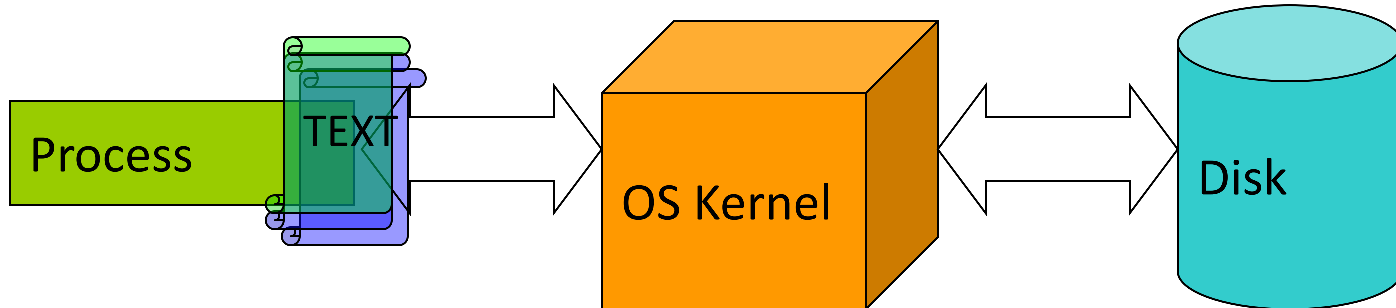
From Nightingale's presentation

Example: External synchrony

```
101   write(buf_1);
102   write(buf_2);
103   print("work done");
104   foo();
```



```
%work done
%
```



From Nightingale's presentation

External Synchrony Design Overview

- Synchrony defined by externally observable behavior.
  - I/O is externally synchronous if output cannot be distinguished from output that could be produced from synchronous I/O.
  - File system does all the same processing as for synchronous.
- Two optimizations made to improve performance.
  - Group committing is used (commits are atomic).
  - External output is buffered and processes continue execution.
- Output guaranteed to be committed every 5 seconds.

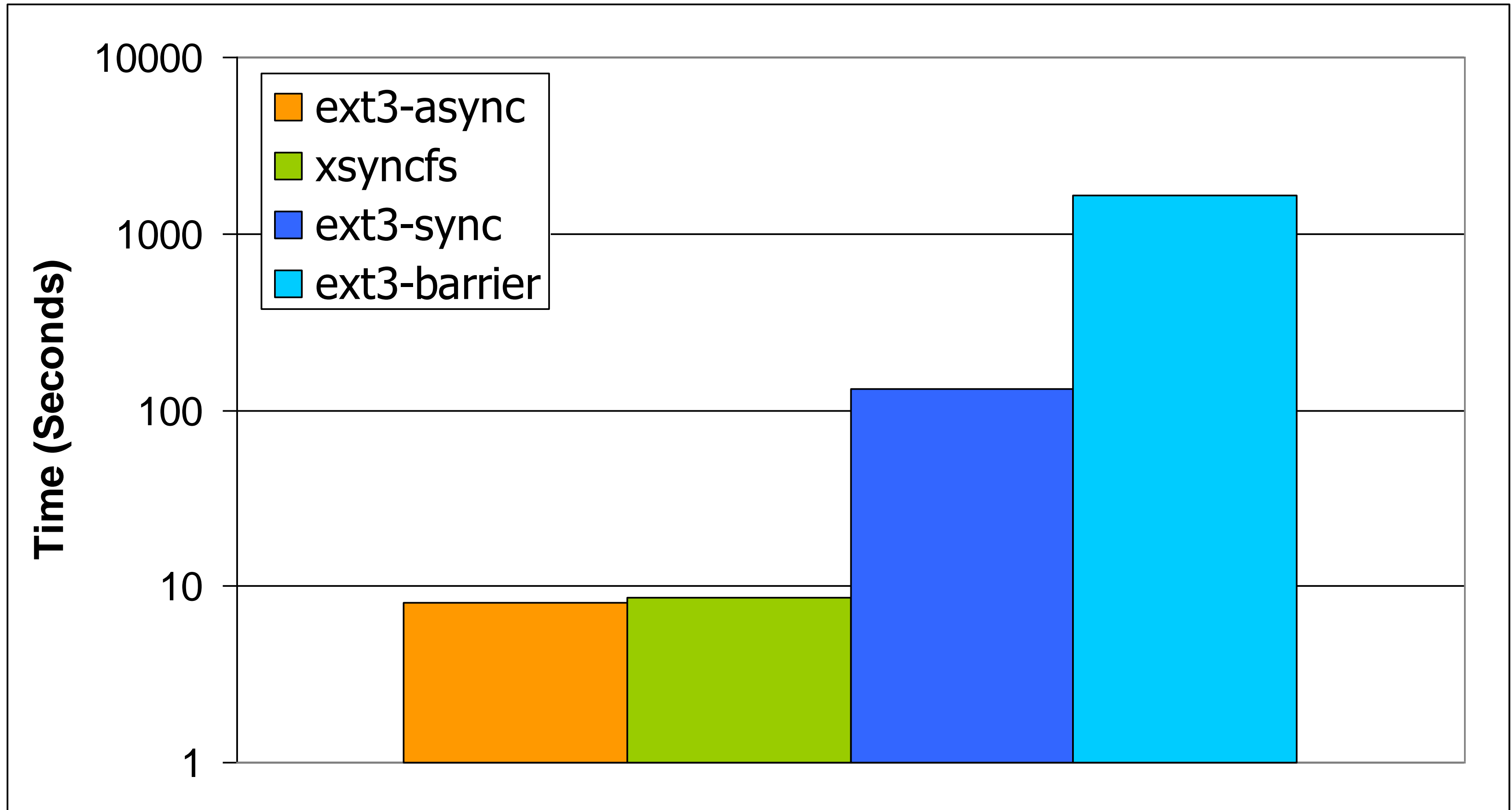
External Synchrony Implementation

- Xsyncfs leverages *Speculator* infrastructure for output buffering and dependency tracking for uncommitted state.
- *Speculator* tracks commit dependencies between processes and uncommitted file system transactions.
- ext3 operates in journaled mode.

Evaluation

- Durability
- Performance
  - IO intensive application (Postmark)
  - Application that synchronize explicitly (MySQL)
  - Network intensive, Read-heavy application (SPECweb)
  - Output-trigger commit on delay

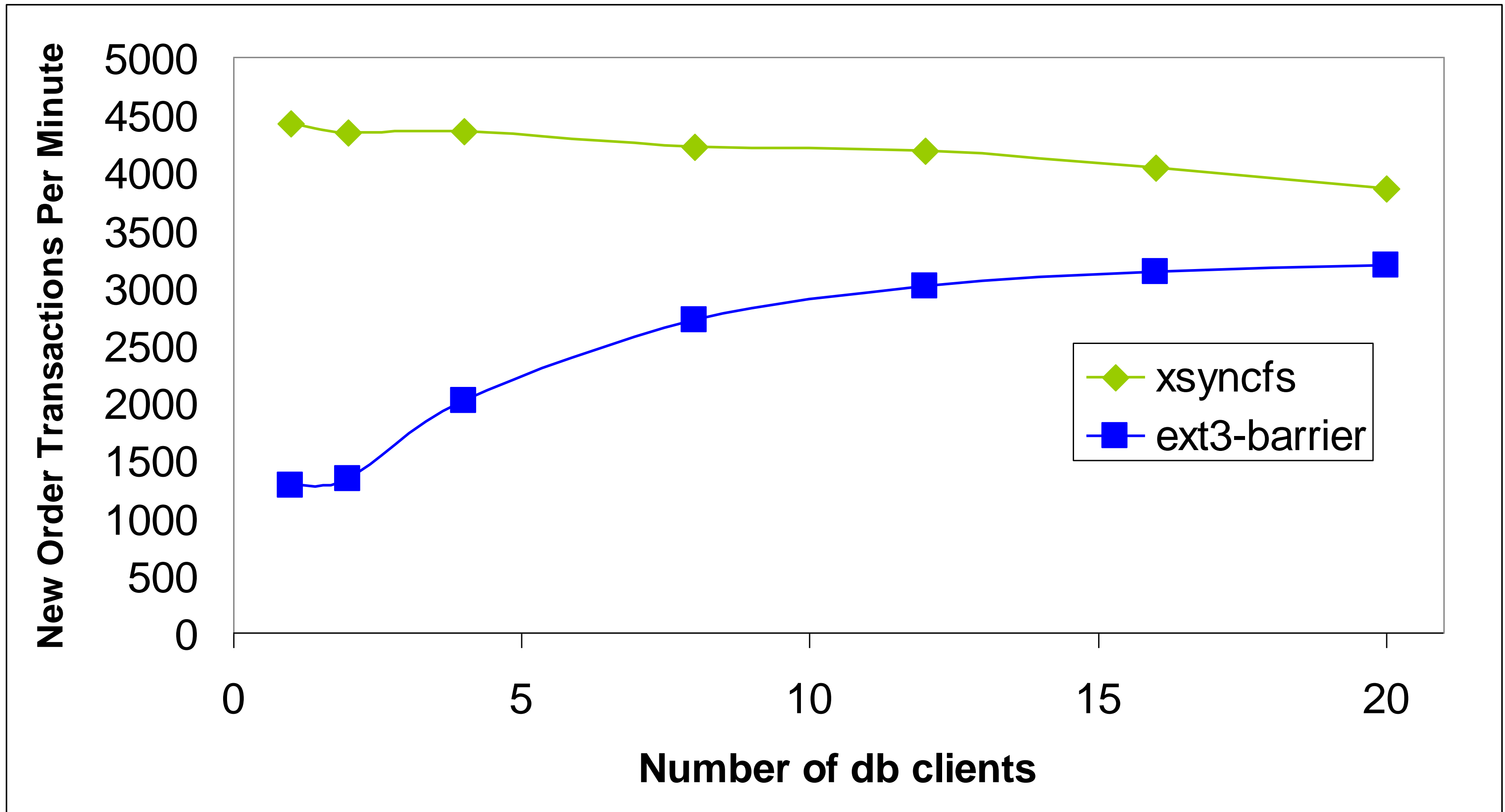
Postmark benchmark



- Xsyncfs within 7% of ext3 mounted asynchronously

From Nightingale's presentation

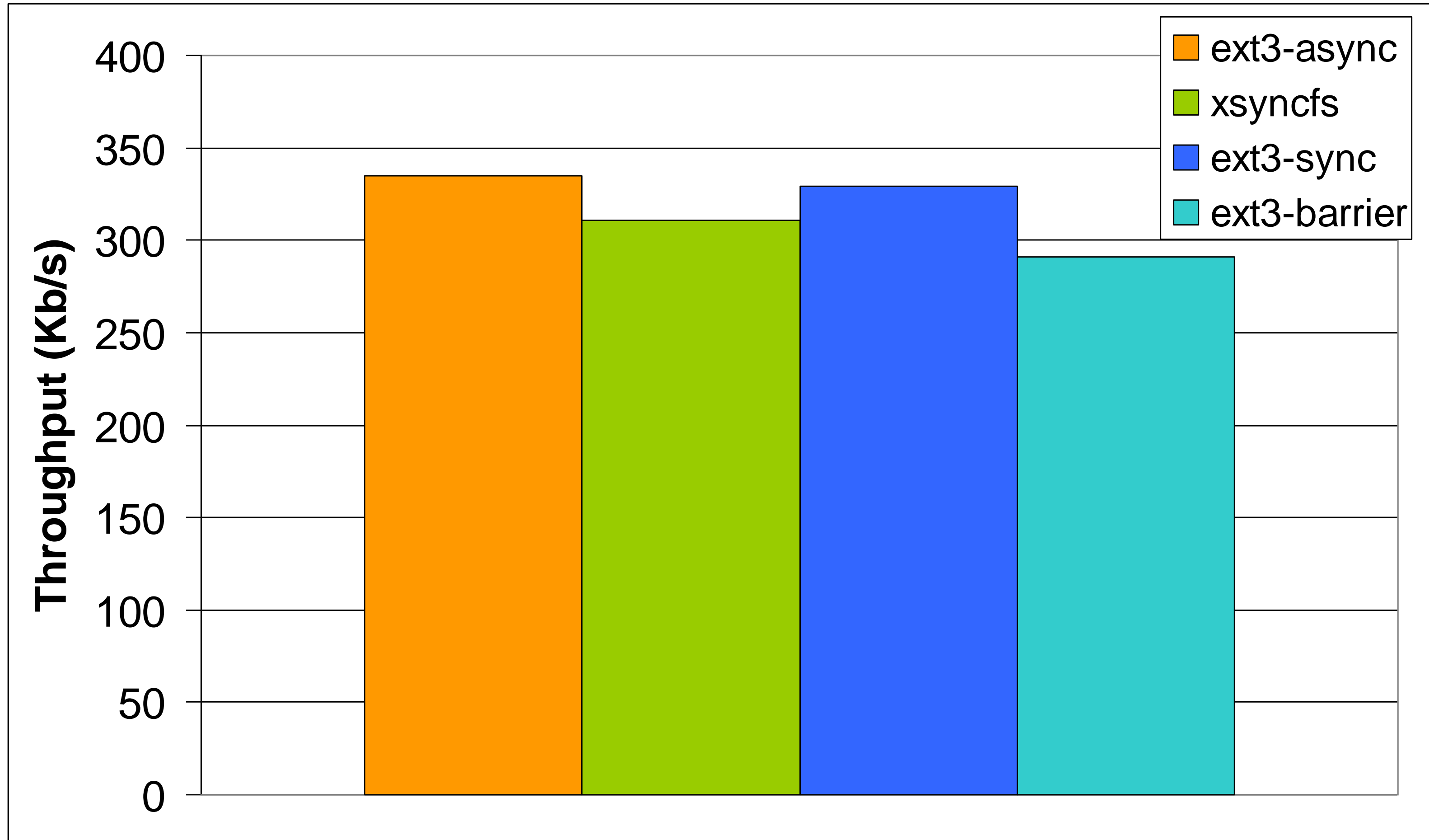
The MySQL benchmark



- Xsyncfs can group commit from a single client

From Nightingale's presentation

Specweb99 throughput



- Xsyncfs within 8% of ext3 mounted asynchronously

From Nightingale's presentation

Specweb99 latency

Request size	ext3-async	xsyncfs
0-1 KB	0.064 seconds	0.097 seconds
1-10 KB	0.150 second	0.180 seconds
10-100 KB	1.084 seconds	1.094 seconds
100-1000 KB	10.253 seconds	10.072 seconds

- Xsyncfs adds no more than 33 ms of delay

From Nightingale's presentation

Discussions

- Is the idea sound?
  - Nice idea, new idea.
- Flaws?
  - Are the experiments realistic?
- What are your take-aways from this paper?

Rethink the Sync

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen and Jason Flinn

Speculative Execution in a Distributed File System

Edmund B. Nightingale, Peter M. Chen, and Jason Flinn

Idea

Example

Design

Evaluation

Speculation Execution

- Question
  - How to improve the distributed file system performance?
- Characteristics of DFS
  - Single, coherent namespace
- Existing approach
  - Trade-off consistency for performance

The Idea

- Speculative execution
  - Hide IO latency
    - Issue multiple IO operations concurrently
  - Also improve IO throughput
    - Group commit
- For it to succeed
  - Correct
  - Efficient
  - Easy to use

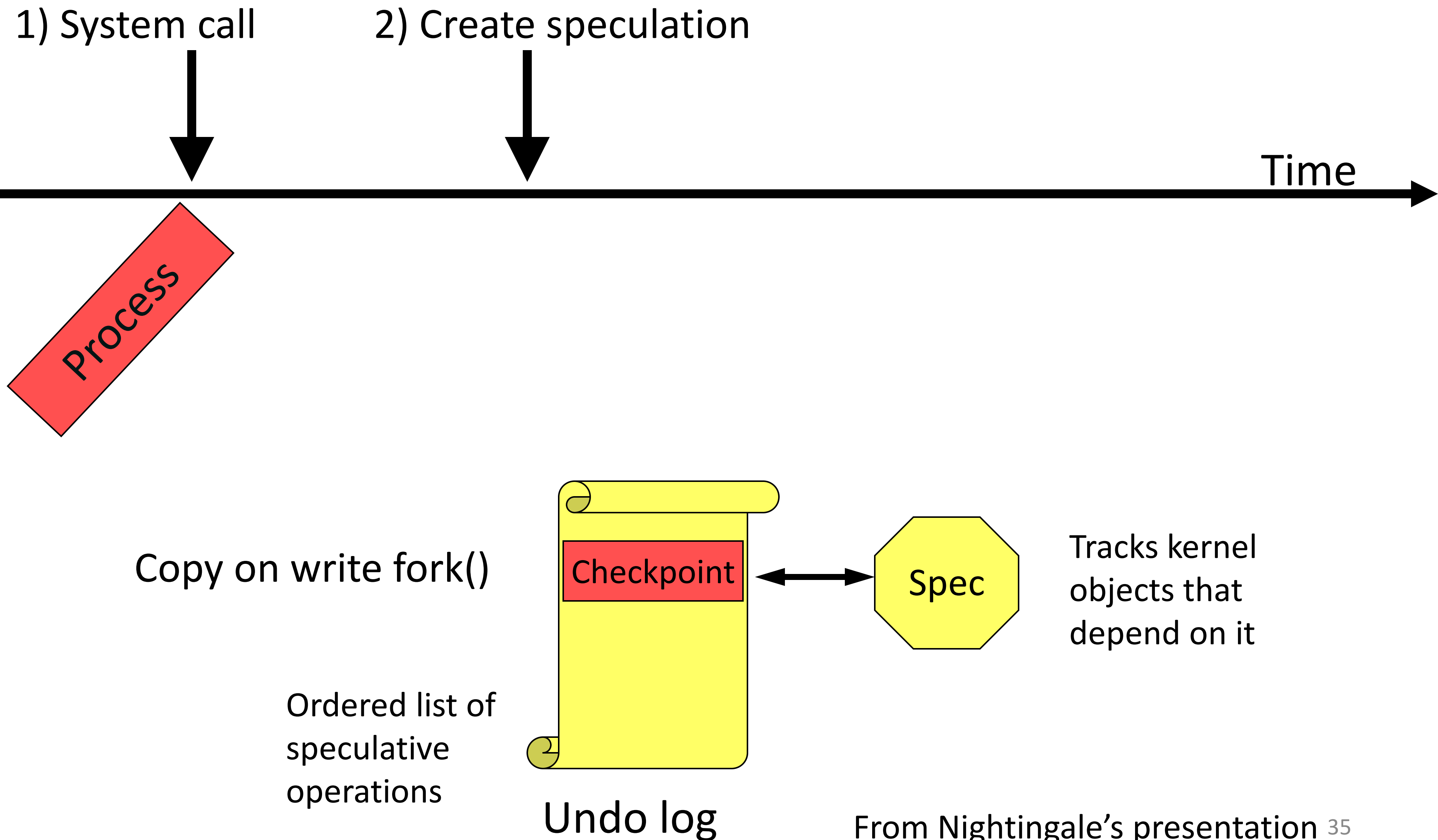
Conditions for Success of Speculations

- Results of Speculation is highly predictable
  - Concurrent updates on cached files are rare
- Checkpointing is faster than Remote I/O
  - 50us ~ 6ms (amortizable) v.s. network RTT
- Modern computers have spare resources
  - CPUs are idle for significant portions of time
  - Extra memory is available for checkpoints

Speculator Interface

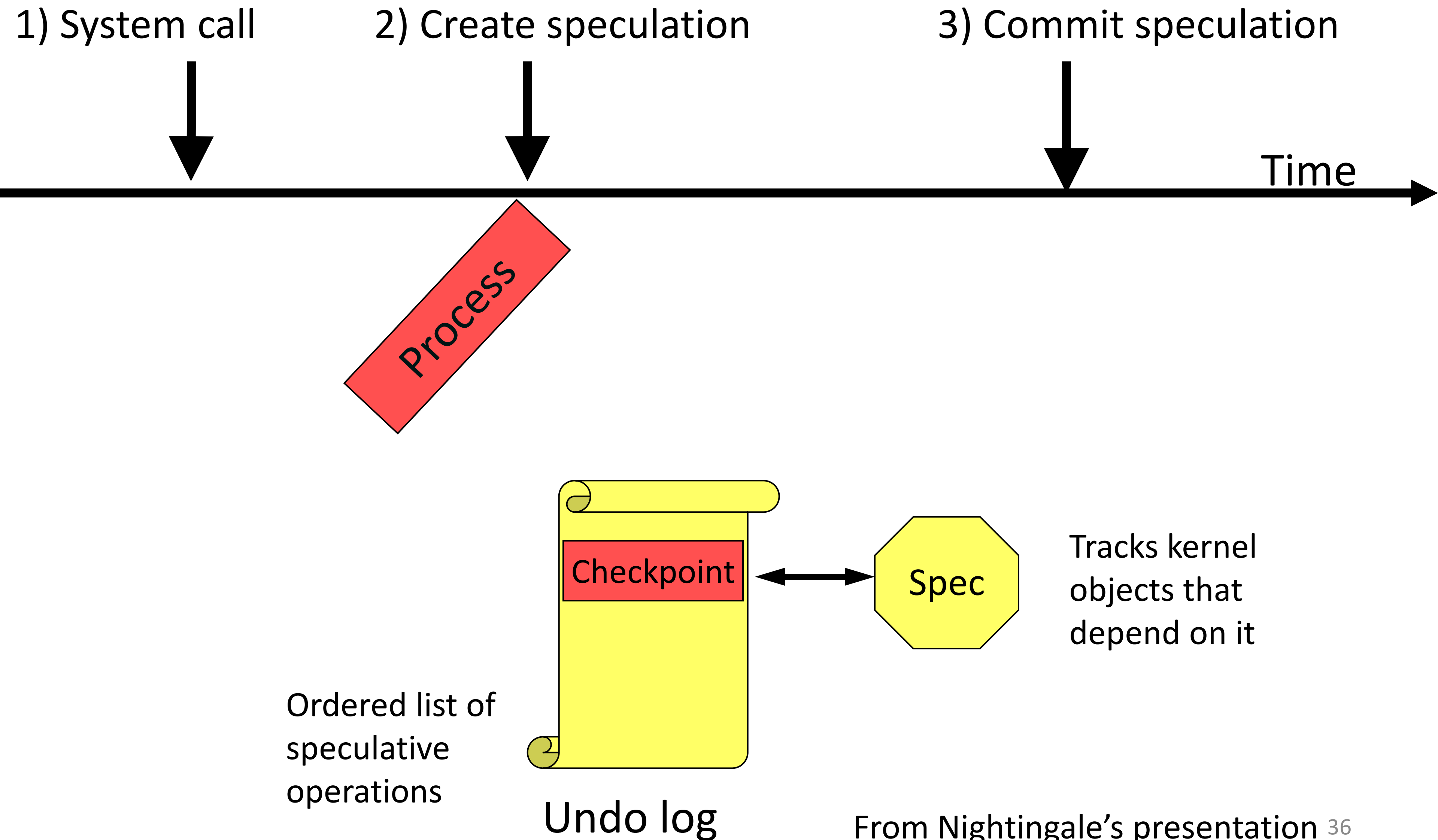
- Speculator provides a lightweight checkpoint and rollback mechanism
- Interface to encapsulate implementation details:
  - `create_speculation`
  - `commit_speculation`
  - `fail_speculation`
- Separation of policy and mechanism
  - Speculator remain ignorant on why clients speculate
  - DFS do not concern how speculation is done

Implementing Speculation



From Nightingale's presentation

Speculation Success



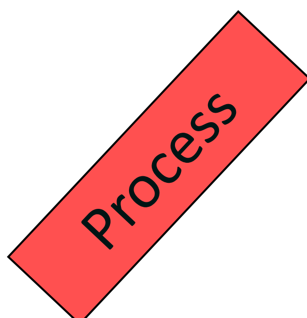
From Nightingale's presentation

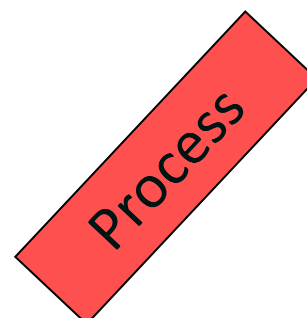
Speculation Failure

1) System call

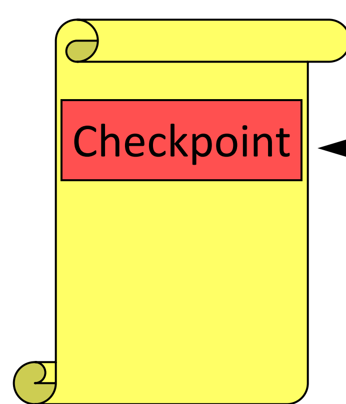
2) Create speculation

3) Fail speculation

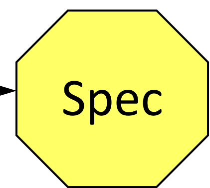




Ordered list of speculative operations



Undo log



Tracks kernel objects that depend on it

From Nightingale's presentation

Ensuring correctness

- Two invariants
  - Speculative state should never be visible to user or any external devices
  - Process should never view speculative state unless it speculatively depends on the state
    - Non-speculative process must block or become speculative when viewing speculative states
- Three ways to ensure correct executions:
  - Block
  - Buffer
  - Propagate speculations (dependencies)

Output Commits

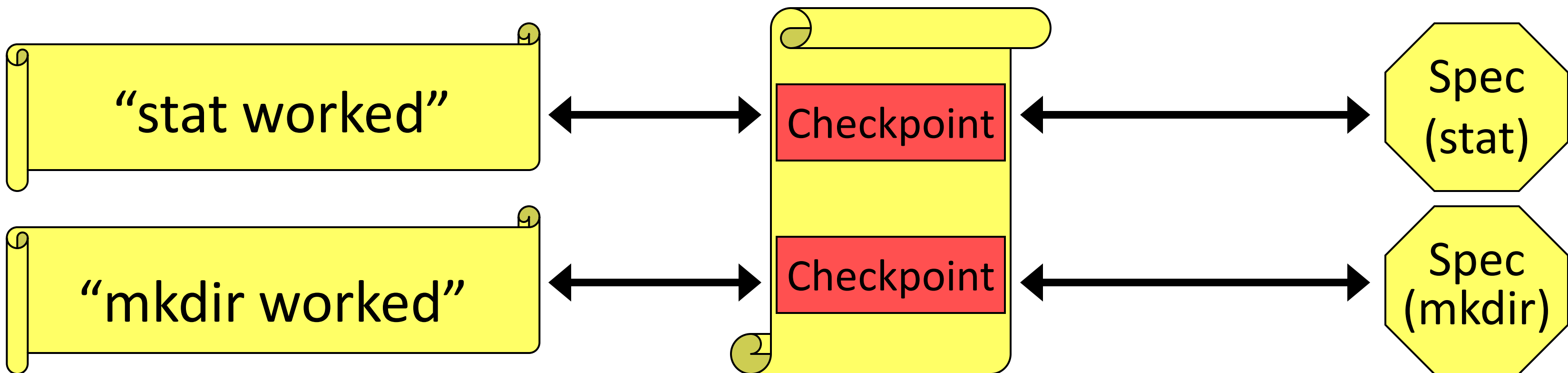
1) sys_stat

2) sys_mkdir

3) Commit speculation

Time

Process



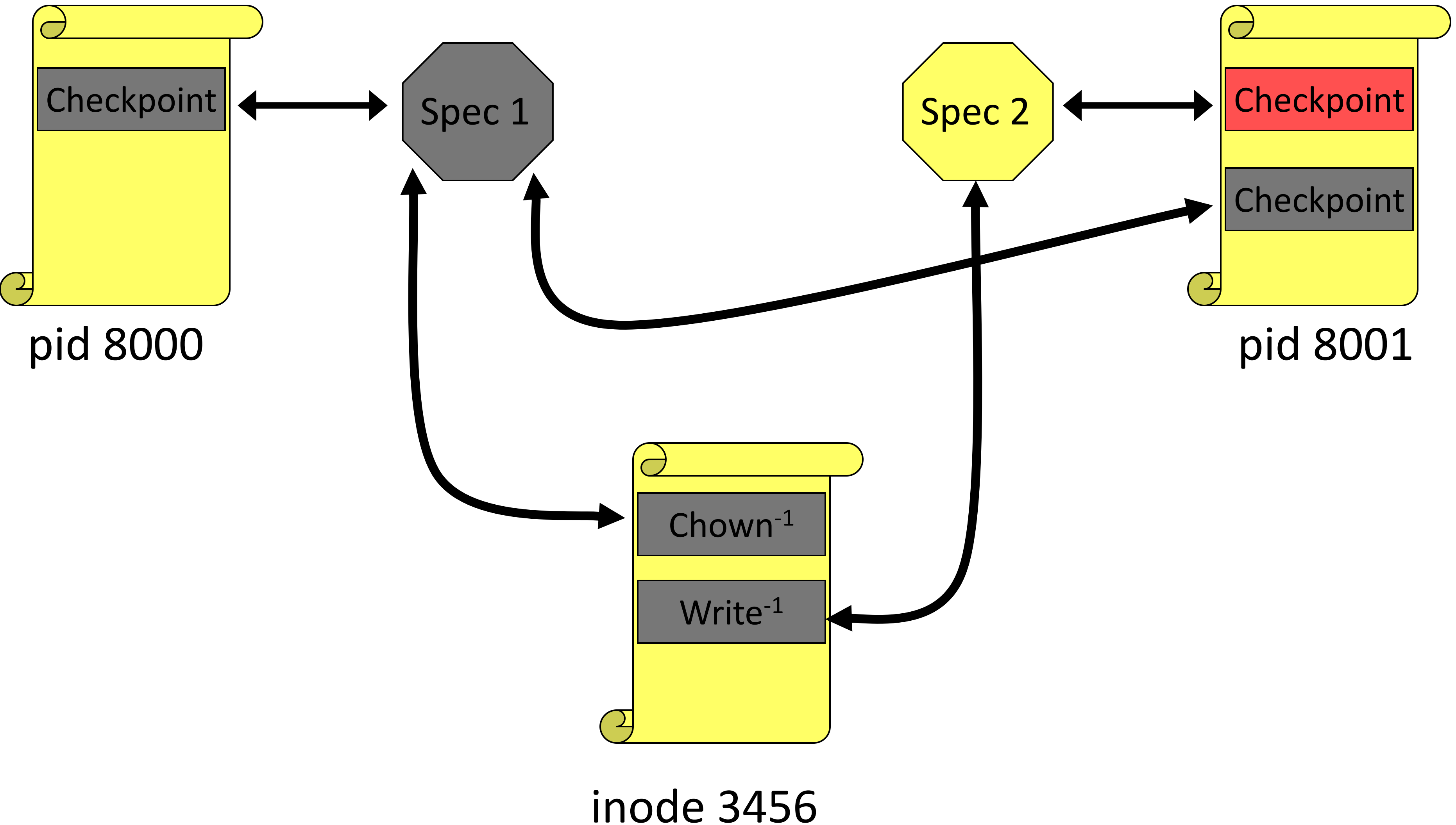
Undo log

From Nightingale’s presentation

Multi-Process Speculation

- Processes often cooperate
  - Example: “make” forks children to compile, link, etc.
  - Would block if speculation limits to one task
- Allow kernel objects to have speculative state
  - Examples: inodes, signals, pipes, Unix sockets, etc.
  - Propagate dependencies among objects
  - Objects rolled back to prior states when specs fail

Multi-Process Speculation



From Nightingale's presentation

Multi-Process Speculation

- Supports
  - Objects in distributed file system
  - Objects in local memory file system -- RAMFS
  - Modified Local ext3 file system
  - IPCs:
    - Pipes and fifos, Unix sockets, signals, fork and exits
- Does not Support
  - System V IPC, Futex, shared memory

Using Speculation

Time

Client 1	Client 2
1. cat foo > bar	
	2. cat bar

Question: What does client 2 view in ‘bar’?

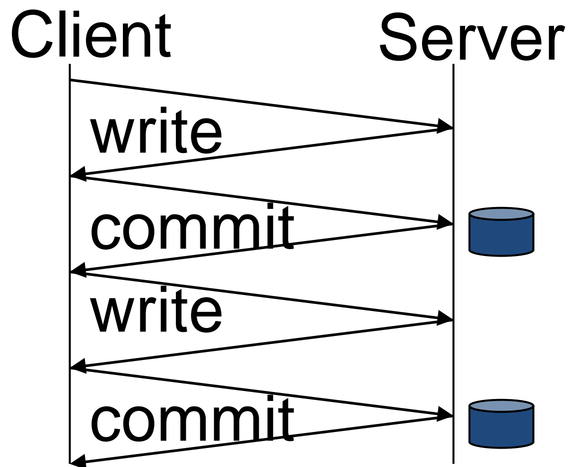
Handling Mutating Operations

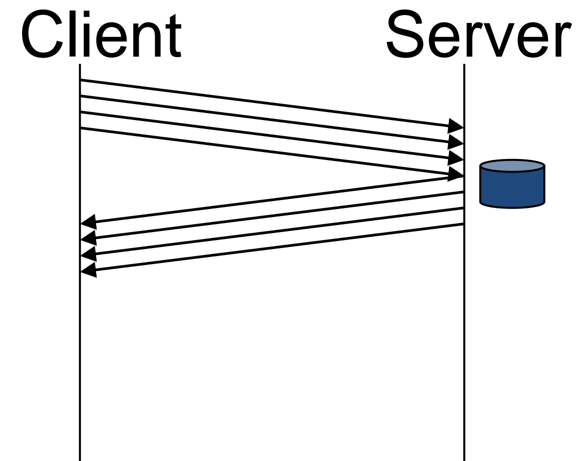
- Server permits other processes to see speculatively changed file only if cached version matches the server version
- Server must process message in the same order as clients see
- Server never store speculative data

Reproduced from Nightingale’s Presentation

Using Speculation

- Speculator makes group commit possible



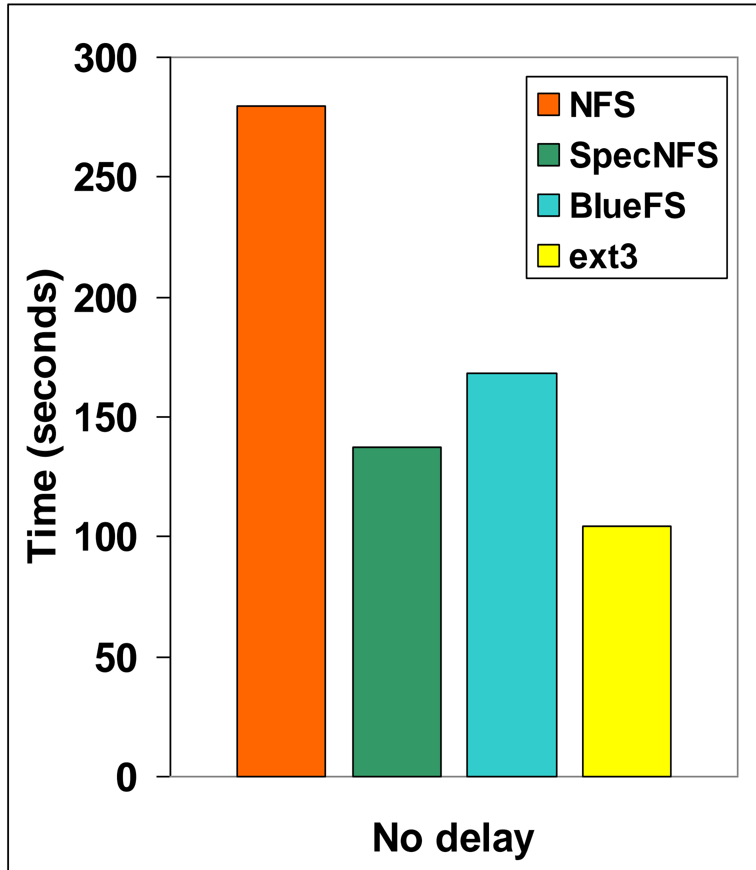


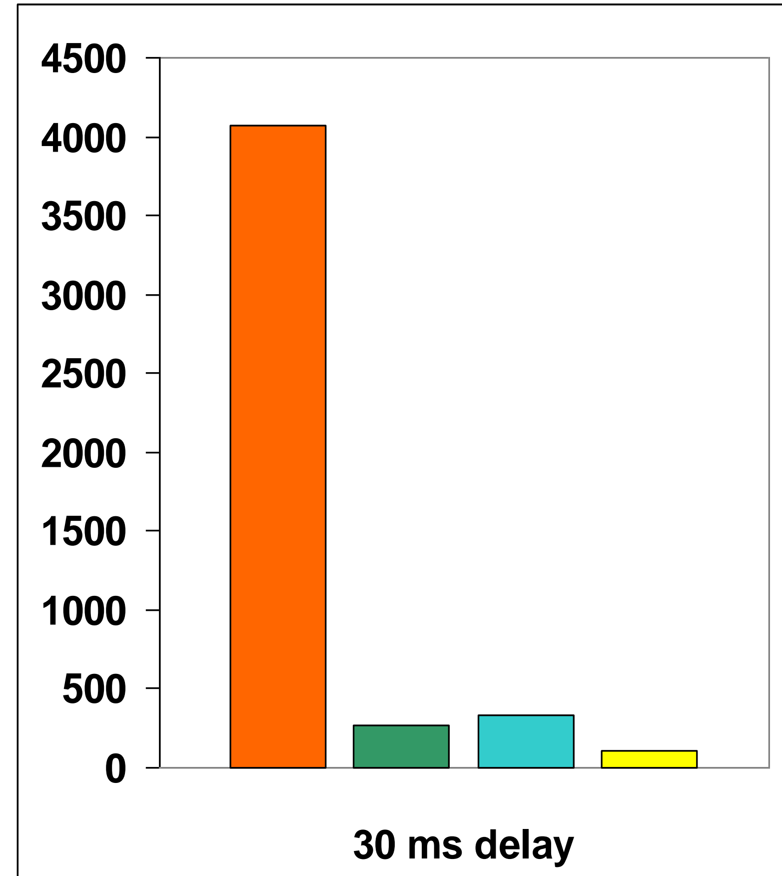
Reproduced from Nightingale’s Presentation

Evaluation: Speculative Execution

- To answer the following questions
  - Performance gain from propagating dependencies
  - Impact on performance when speculation fails
  - Impact on performance of group commit and sharing state

Apache Build

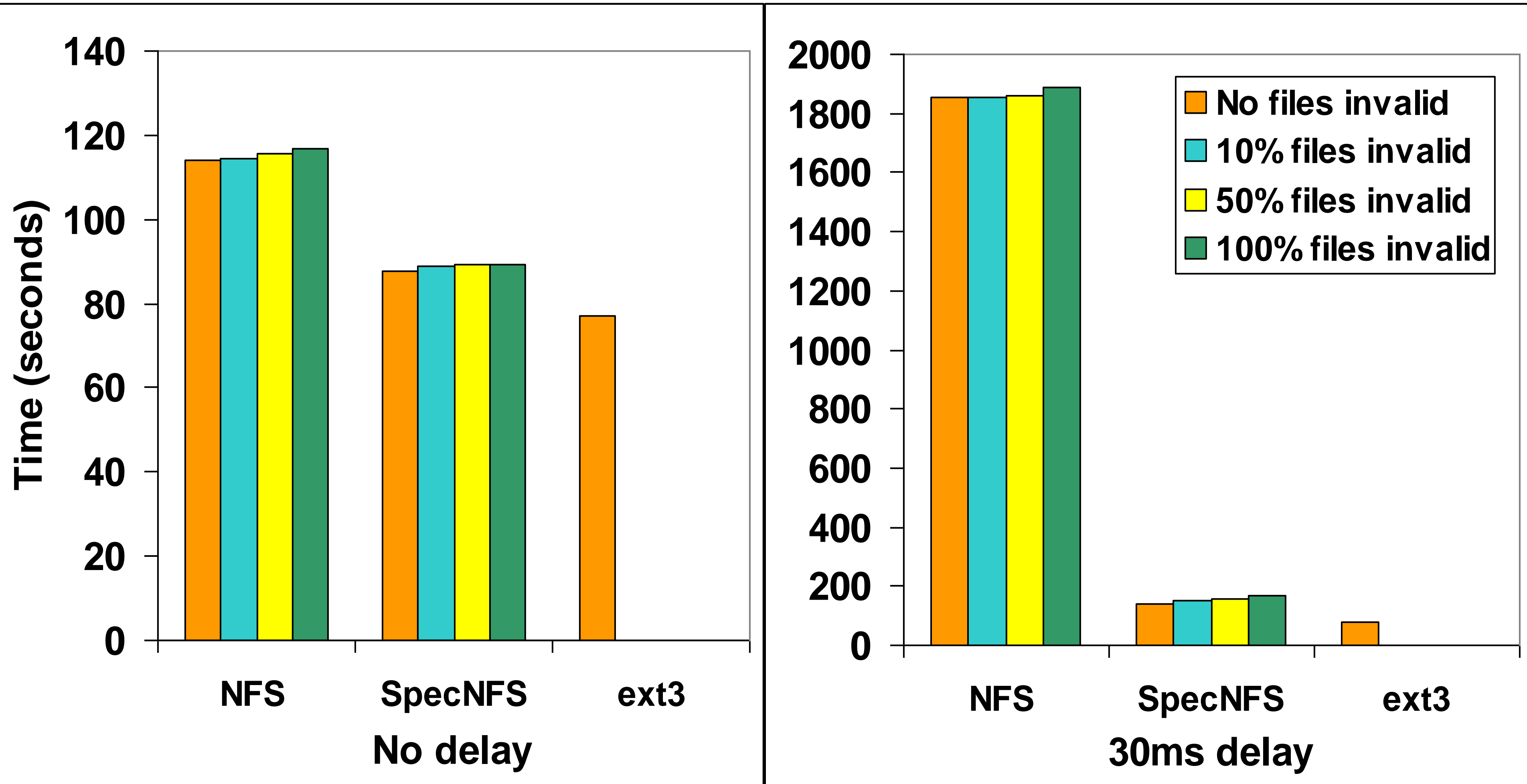




- With delays SpecNFS up to 14 times faster

From Nightingale's presentation

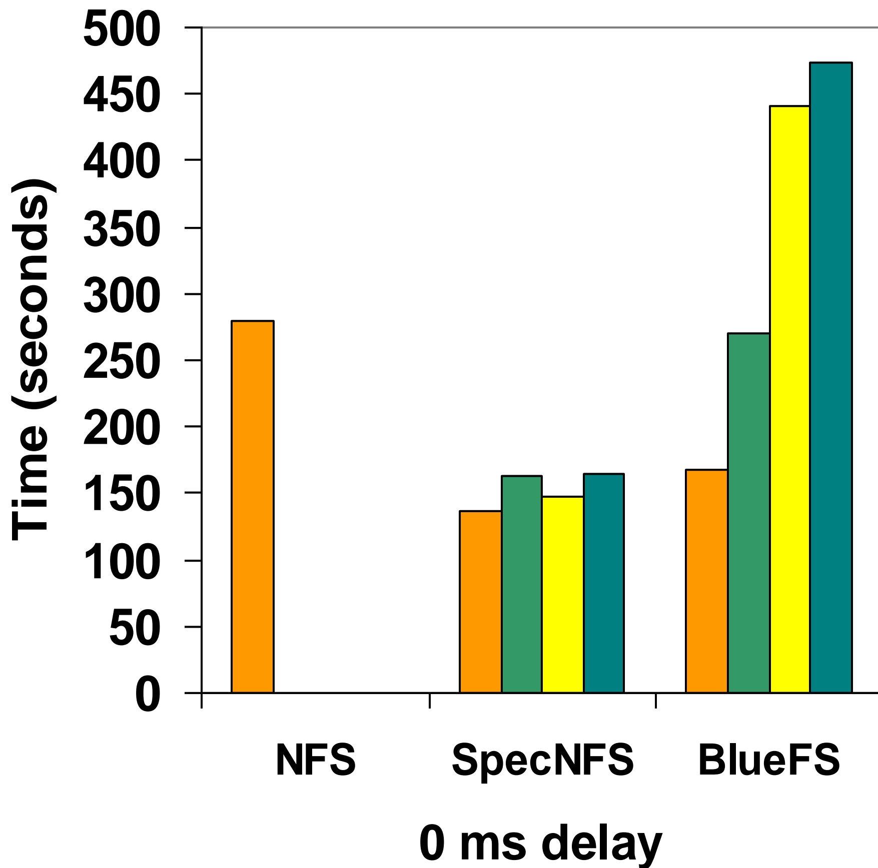
The Cost of Rollback

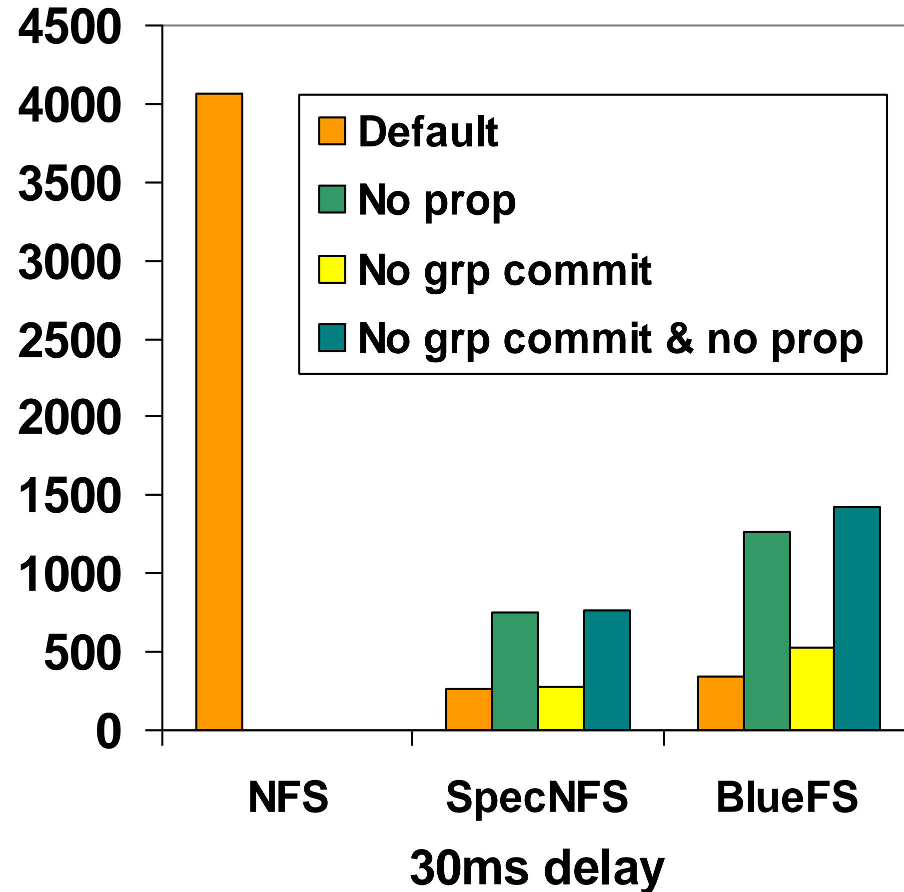


- All files out of date SpecNFS up to 11x faster

From Nightingale's presentation

Group Commit & Sharing State





From Nightingale's presentation

Discussions

- Is speculation in OS the right level of abstraction?
  - Similar Ideas:
    - Transaction and Rollback in Relational Database
    - Transactional Memory
    - Speculative Execution in OS
- What if the conditions for success do not hold?
- Portability of code
  - Code perform worse if OS does not speculate
  - What about transform source code to perform speculation?
- Why isn't this used nowadays?

Conclusions

- Performance need not be sacrified for durability
- The transaction and rollback infrastructure in OS is very useful, two good papers!
- Ideas are not new, but are generic.

Thanks!

Things they did not do

- Mechanism to prevent disk corruption when crash occurs. They used the default journaled mode.

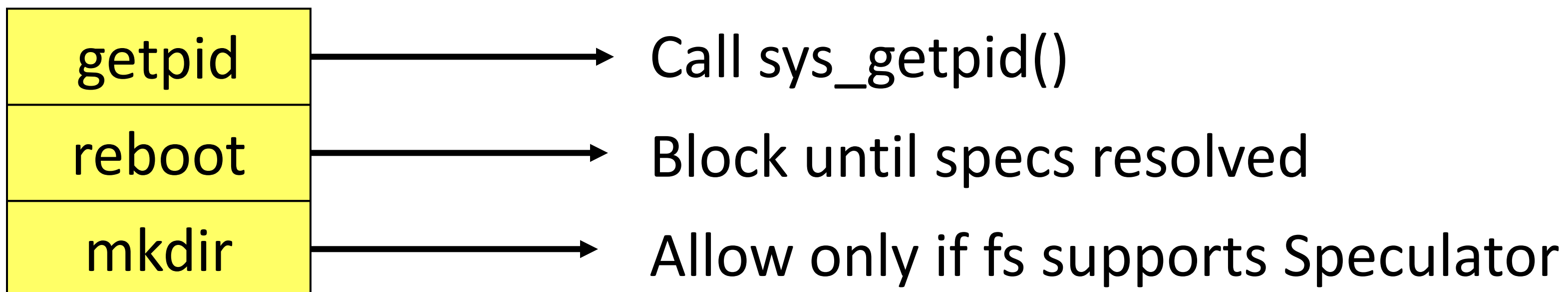
Comparison

Speculative Execution	Rethink the Sync
Synchronous IO -> Asynchronous IO	
Distributed File System	Local File System
Checkpointing	--
Pipelining Sequential IO	--
Propagate Dependencies	Propagate Dependencies
Group Commit	Group Commit
--	Output-triggered commit

Systems Calls

- Modify system call jump table
- Block calls that externalize state
  - Allow read-only calls (e.g. getpid)
  - Allow calls that modify only task state (e.g. dup2)

- File system calls -- need to dig deeper
  - Mark file systems that support Speculator



Scenario 1:

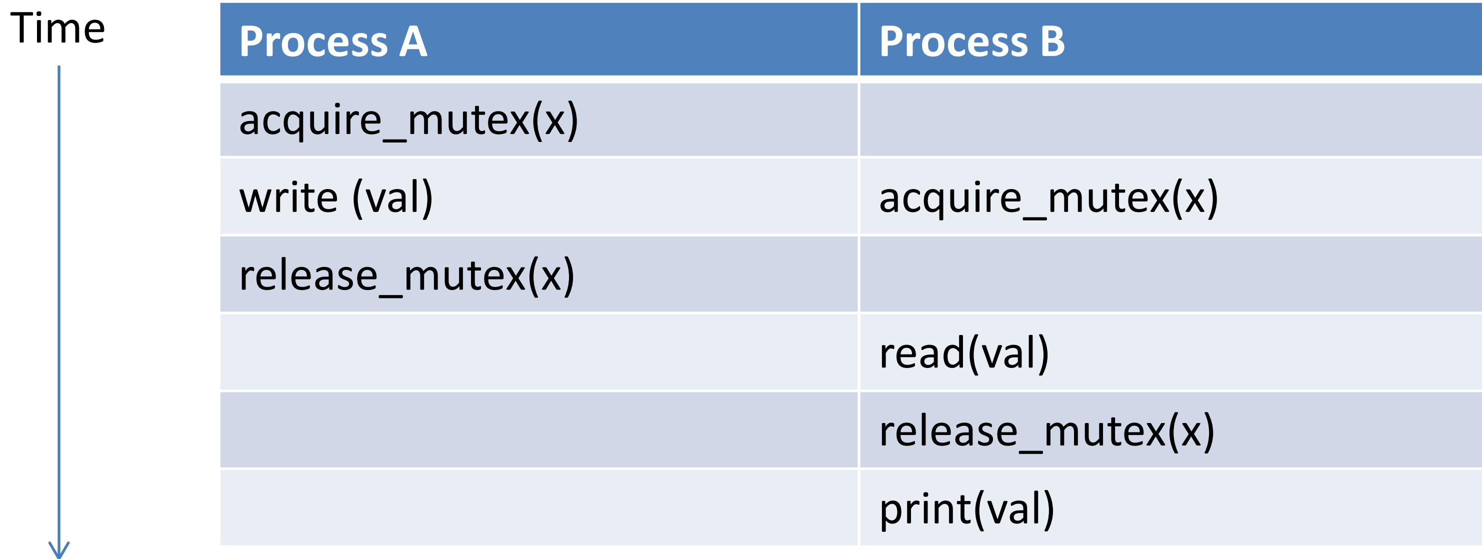
```
write ();
print ();
write ();
print ();
```

Question:

Does xsyncfs perform similarly as synchronous IO?

Source: OSDI official blog

- # Scenario 2:



Process A	Process B
acquire_mutex(x)	
write (val)	acquire_mutex(x)
release_mutex(x)	
	read(val)
	release_mutex(x)
	print(val)

Question:

Will process B fail to read (Step 4) the update by process A?

Will the print comes before the write in process A have committed?

Source: OSDI official blog