Rethink the Sync

E. Nightingale, K. Veeraraghavan, P. Chen and J. Flinn, U. Michigan.
Appeared in Proc. of OSDI 2006.
Best paper award.

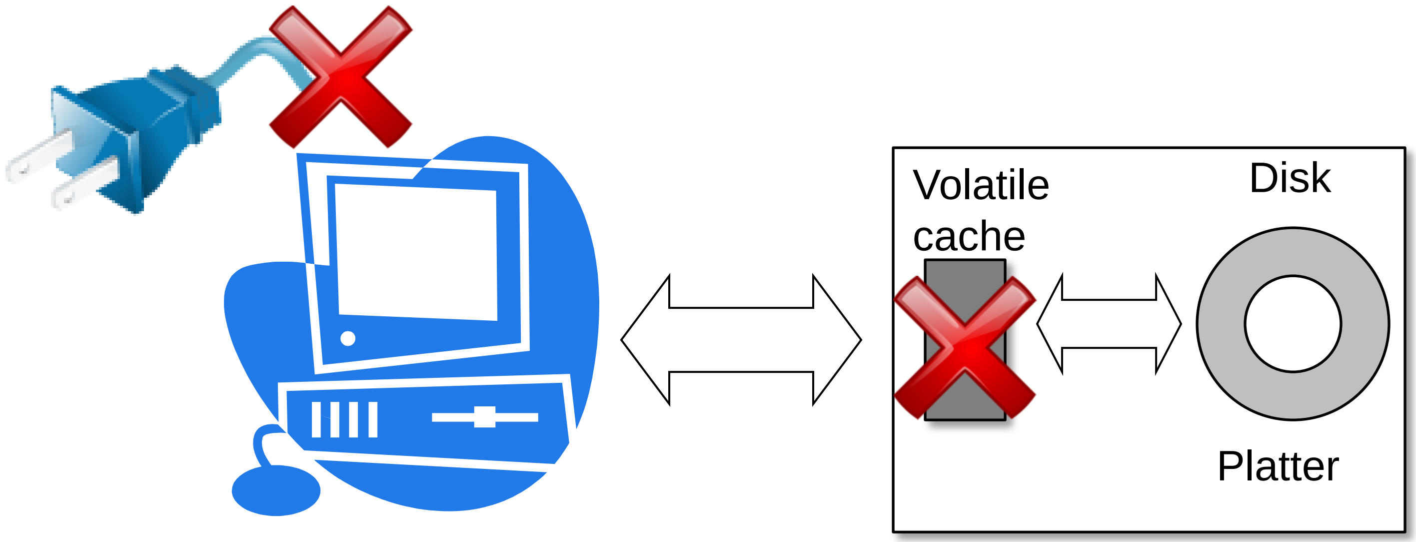
*Slides reuses some of the authors

Durability and performance in file systems

- File systems' conflicting goals
  - Durability & performance → synchronous & asynchronous
- Synchronous FS
  - Durability by blocking callers until modifications are committed to disk
  - Clean abstraction
    - What you see completing is durable and
    - Ordering is correct
  - Very slow …. 2x for disk-intensive benchmarks
- Asynchronous FS
  - Fast but not safe
    - Cost in durability and order
    - Harder programming – complicates applications that need durability or ordering guarantees

When a sync() is really async

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



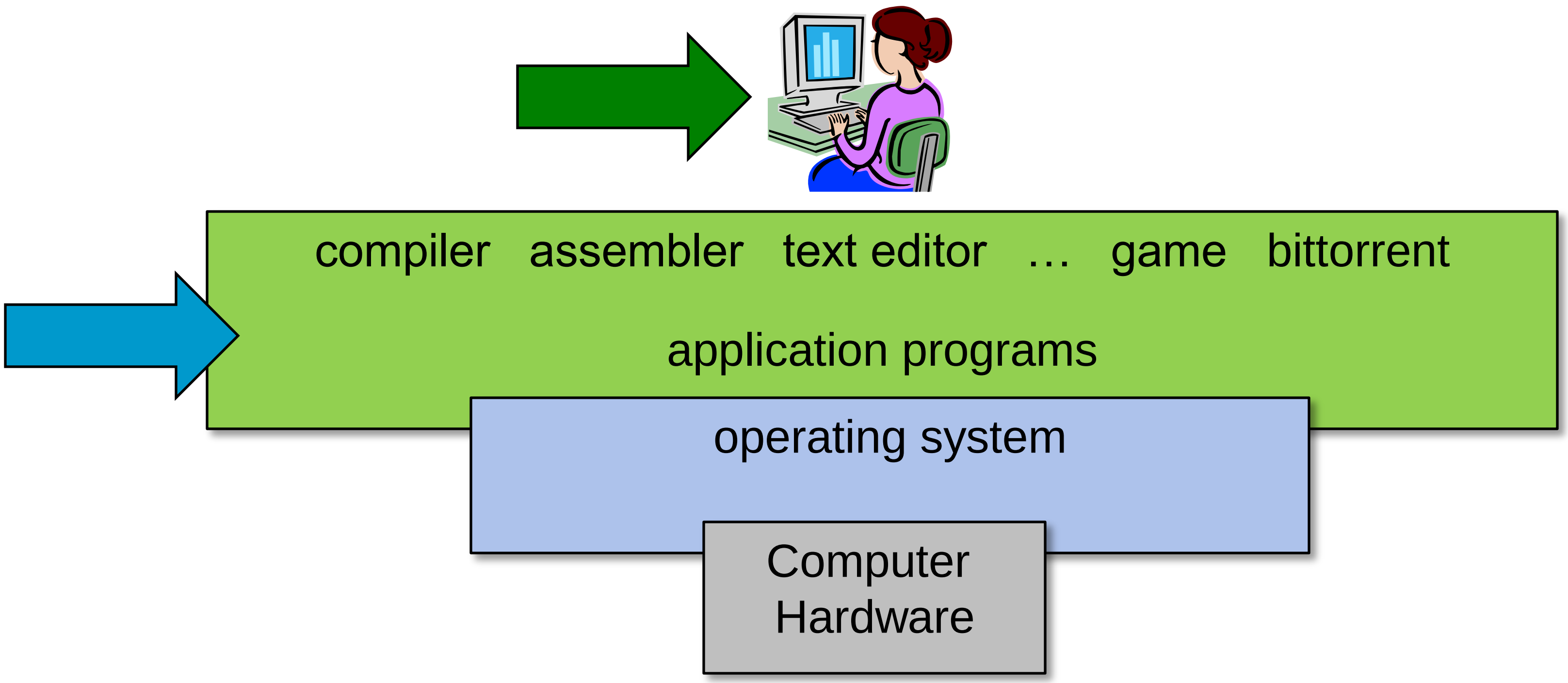
100x slower than asynchronous I/O if disable cache

Solution

- Resolving the tension with a new model for synchronous I/O
  - *External synchrony*
  - Same guarantees as synchronous I/O
  - Only 8% slower than asynchronous I/O

To whom are guarantees provided?

- Synchronous I/O
  - Defined by implementation: caller blocked until op. completes
  - An application-centric view
- Guarantee really provided to the user – users, not applications, are the true observers of the system

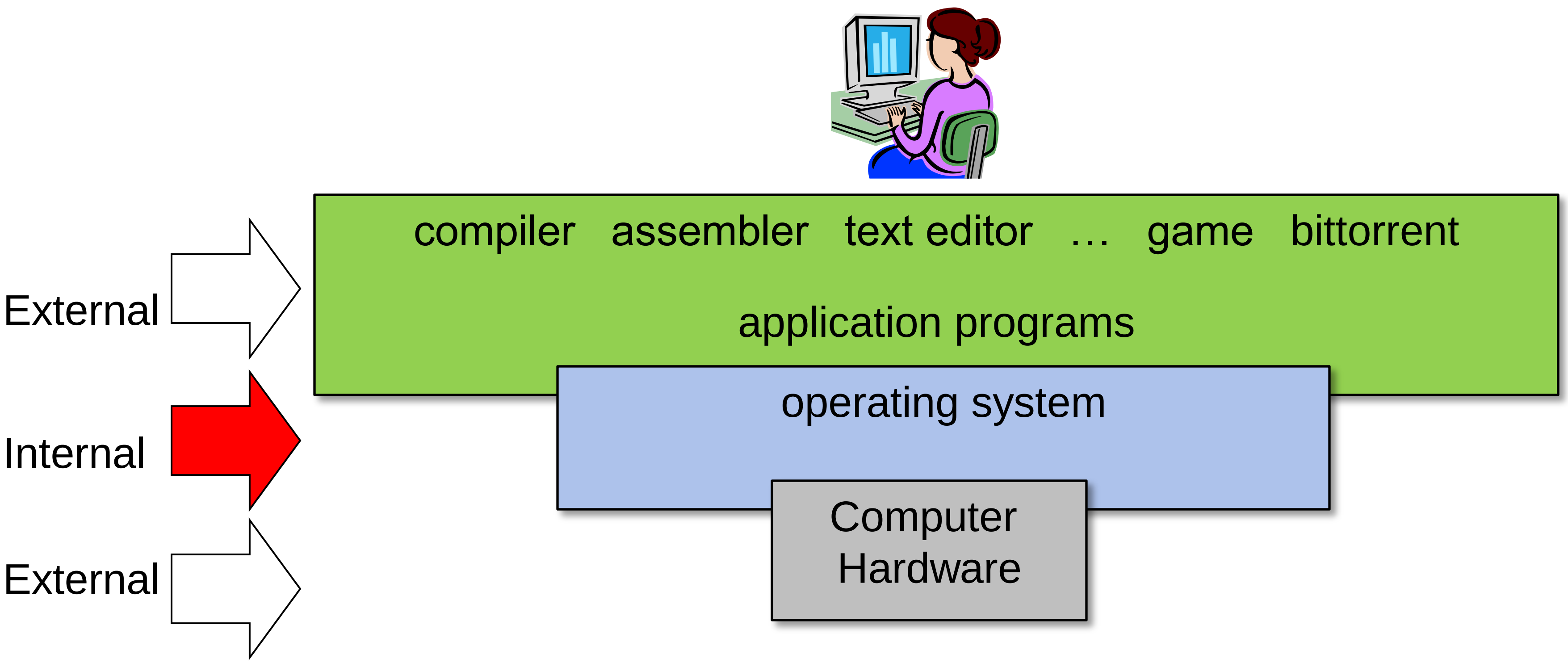


Providing the user a guarantee

- User *observes* operation has completed
  - User may examine screen, network, disk…
- Guarantee provided by synchronous I/O
  - Data durable when operation observed to complete
- To observe output it must be externally visible
  - Visible on external device
  - Application state is *not* directly observable by external entities

Why do applications block?

- Since application are external, we block on syscall



Application is internal therefore no need to block

A new model of synchronous I/O

- Provide guarantee directly to user
  - Rather than via application
- Called externally synchronous I/O
  - Indistinguishable from traditional sync I/O
  - Defined by its observable behavior – if the external output looks the same as if produced by synchronous I/O
  - Approaches speed of asynchronous I/O
- Viable because the OS control access to external devices
  - Applications can only generate external events with the OS help

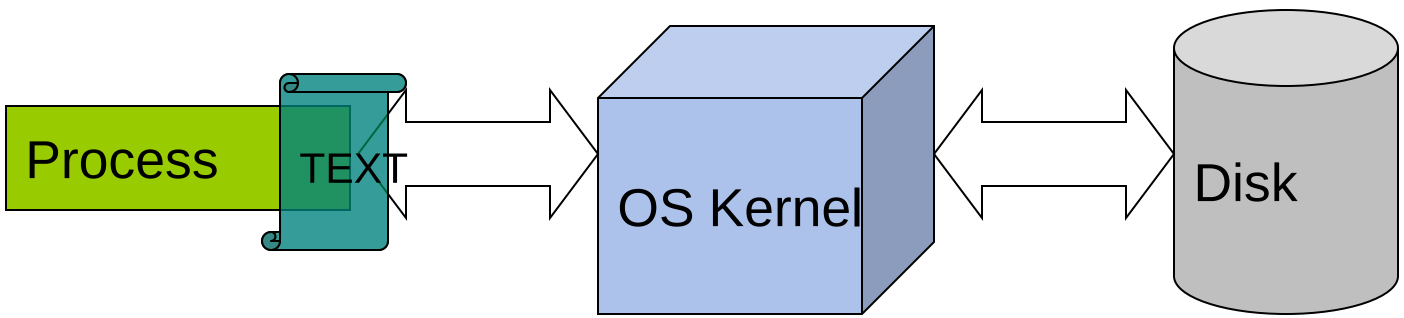
Example: Synchronous I/O

```
write(buf_1);
write(buf_2);
print("work done");
foo();
```

Application blocks

Application blocks

```
%work done
%
```



Observing synchronous I/O

```
write(buf_1);
write(buf_2);
print("work done");
foo();
```

Depends on 1st write

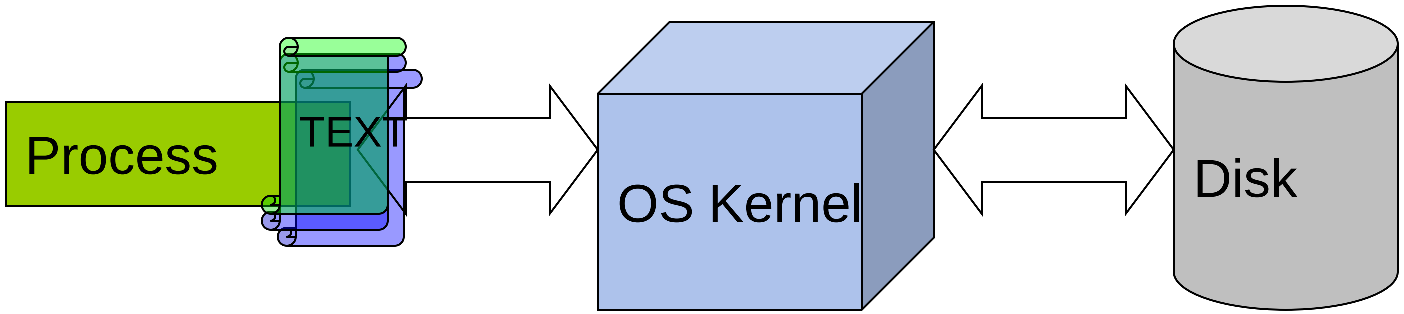
Depends on 1st & 2nd write

- Sync I/O externalizes output based on causal ordering
  - Enforces causal ordering by blocking an application
- Ext sync
  - The values of external outputs are the same
  - Outputs occur in the same causal ordering (Lamport's happens before) without blocking applications

Example: External synchrony

```
write(buf_1);
write(buf_2);
print("work done");
foo();
```

```
%work done
%
```



Tracking causal dependencies

- Applications may communicate via IPC
  - Socket, pipe, fifo etc.
- Need to propagate dependencies through IPC
- Built upon Speculator [SOSP '05]
  - Track and propagate causal dependencies
  - Buffer output to screen and network

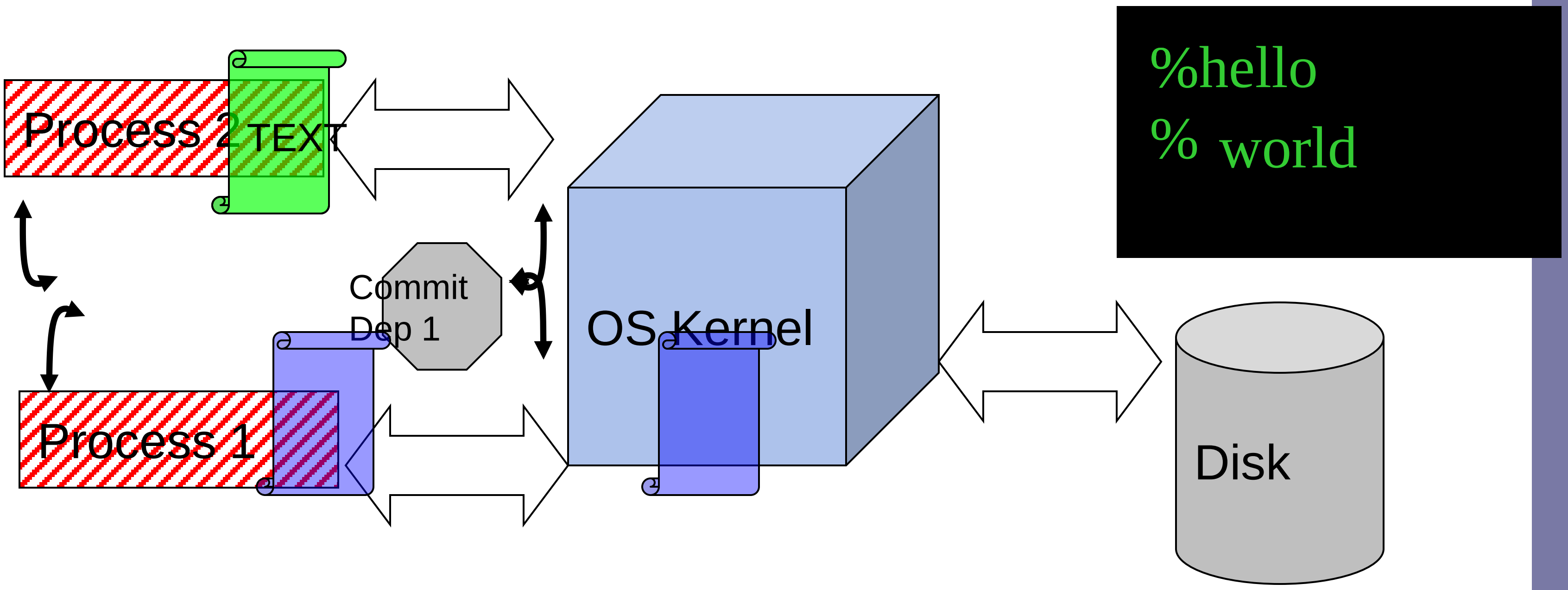
Tracking causal dependencies

Process 1

```
101   write(file1);
102   do_something();
```

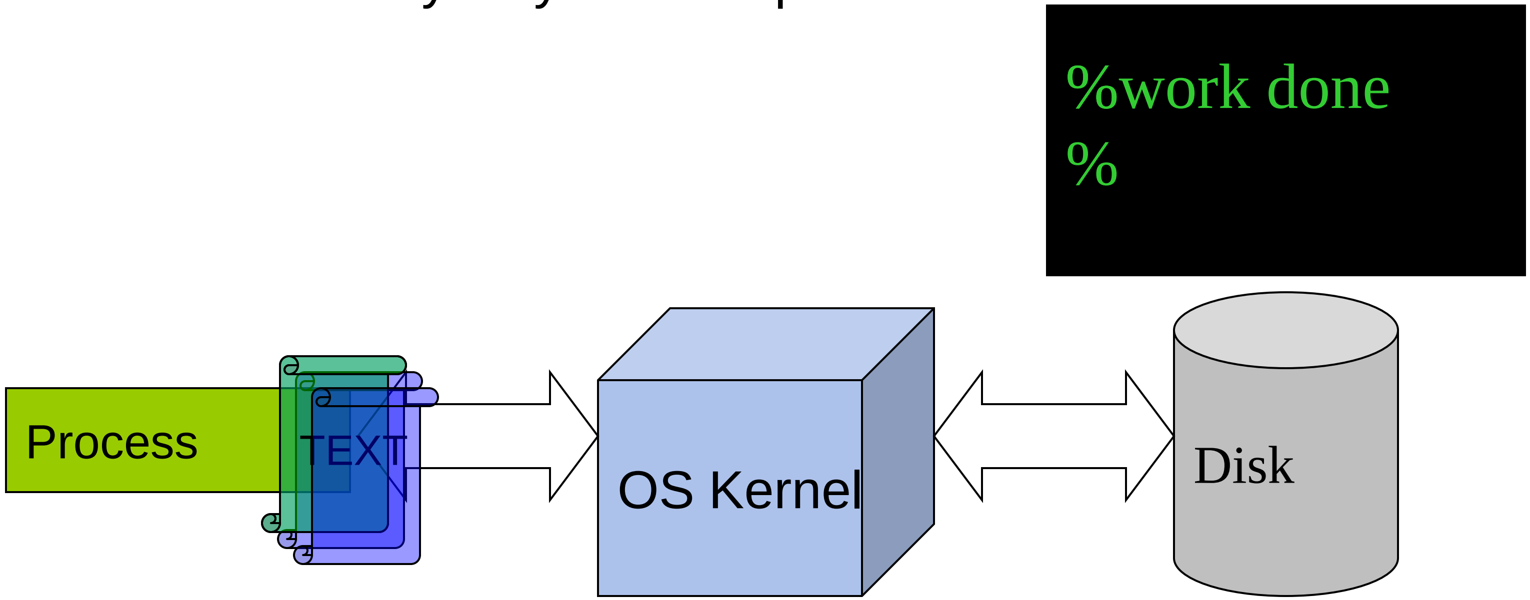
Process 2

```
101   print ("hello");
102   read(file1);
103   print("world");
```



Output triggered commits

- A well-known tradeoff between throughput & latency for group commit strategies
  - Delaying commit will improve throughput, but increase latency
- Maximize throughput until output buffered
- When output buffered, trigger commit
  - Minimize latency only when important



Limitations

- Complicates application-specific recovery from media failures – errors are not immediately obvious
- Users may have temporal expectations as to when data is committed to disk – xsyncfs avoids long waits committing every 5sec at most
- Modifications to data in two different file systems cannot be easily committed with a single disk transaction

Evaluation

- Implemented ext sync file system Xsyncfs
  - Based on the ext3 file system
  - Use journaling to preserve order of writes
  - Use write barriers to flush volatile cache
- Compare Xsyncfs to 3 other file systems
  - Default asynchronous ext3
  - Default synchronous ext3
  - Synchronous ext3 with write barriers

When is data safe?

- Local machine continuously
  - Writes to its local FS
  - Sends a UDP msg that is logged by a remote machine
- During execution, cut power
- Compare log and FS state
- Failed durability
  - Remote logs a msg for a write, but data is missing in test computer
- Failed ordering
  - State of the file afer reboot violates temporal ordering of writes (i.e. FS misses some of the previously written blocks)

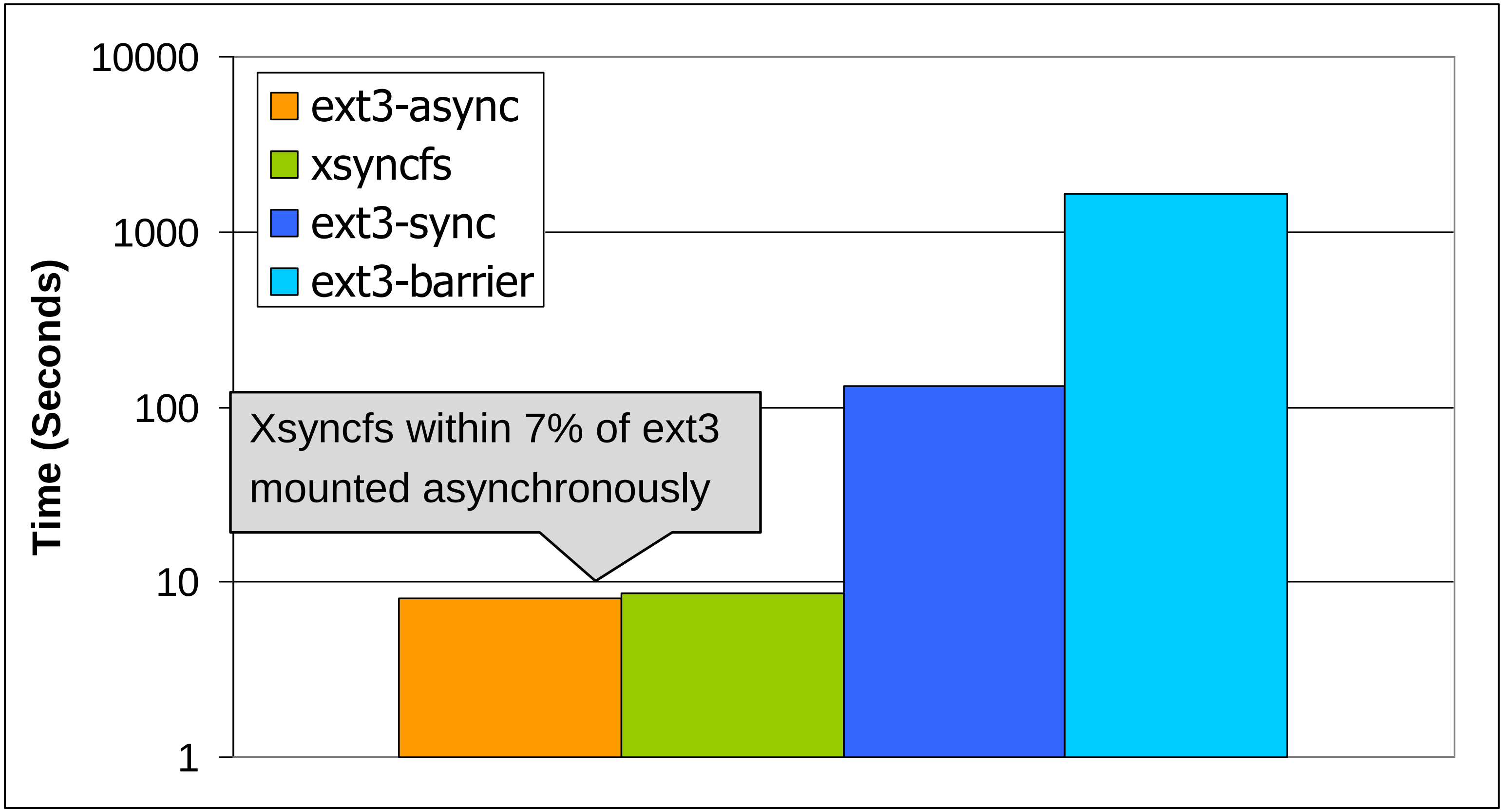
When is data safe?

- Without write barriers, ext3 does not guarantee durability
- Even with journaling, loss of power can corrupt data & metadata

File System Configuration	Data durable on write()	Data durable on fsync()
Asynchronous	No	Not on power failure
Synchronous	Not on power failure	Not on power failure
Synchronous w/ write barriers	Yes	Yes
External synchrony	Yes	Yes

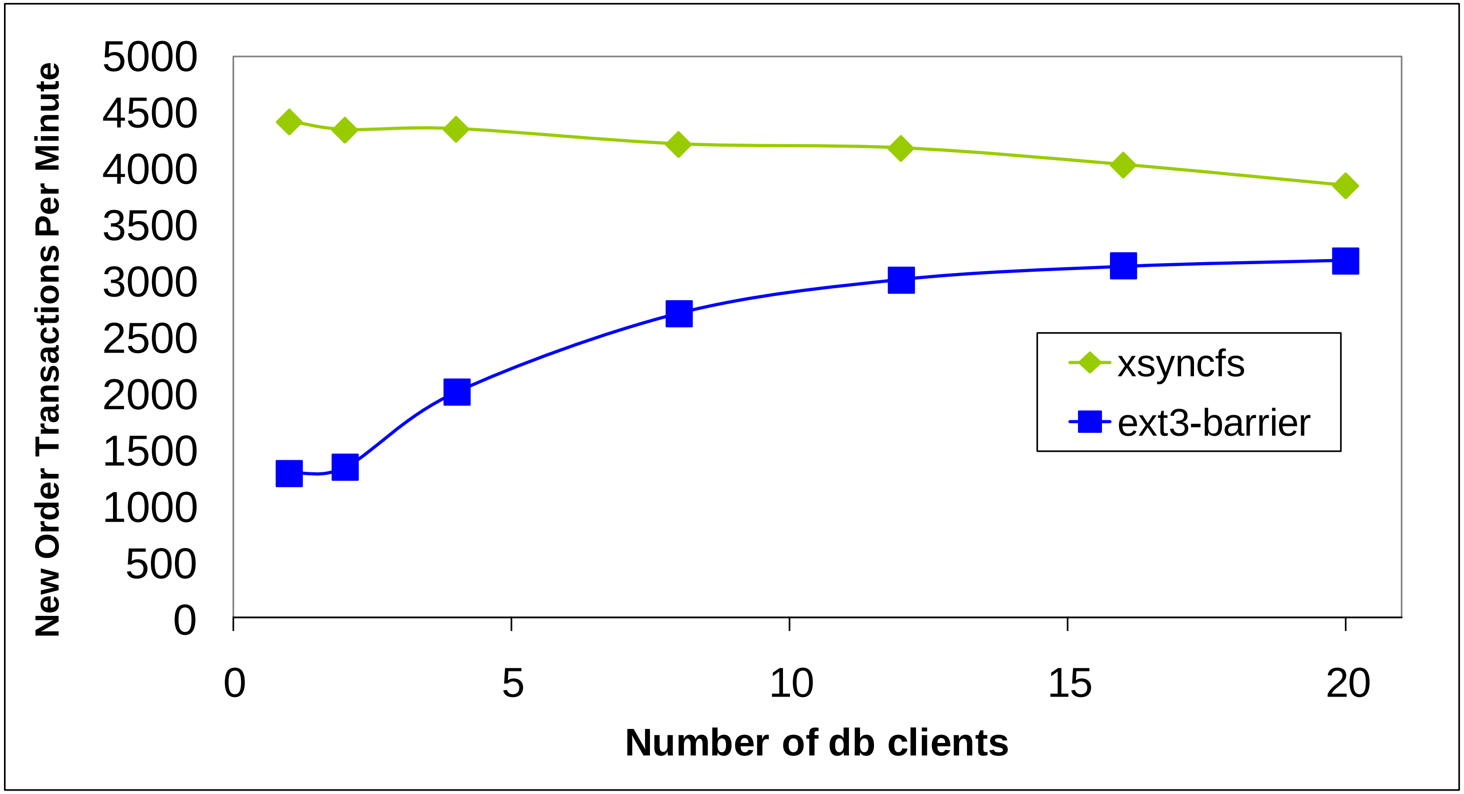
PostMark benchmark

- Replicate small file workloads seen in email, netnews, web-based commerce
- A good test of file system throughput – no output



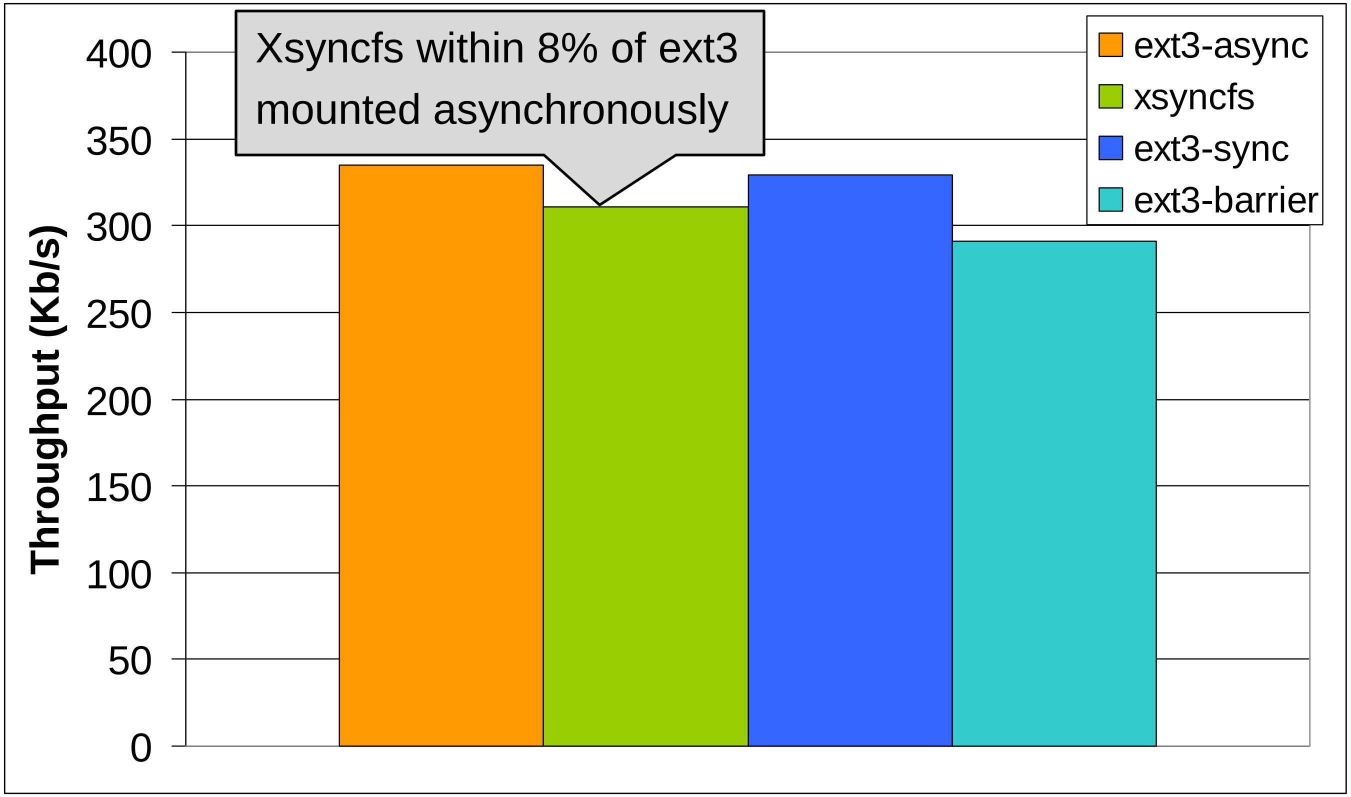
The MySQL benchmark

- How does xsyncfs compares with an application that perform its own group commit strategy?
- Use a modified version of OSDL TPC-C benchmark using MySQL



Specweb99 throughput

- Impact of external synchrony on a network-intensive application
- Clients issue a mix of http get/post requests – sending a network message externalizes state



Specweb99 latency

- Xsyncfs must buffer each message until file system data has been committed
- It adds no more than 33ms of delay (less than the 50ms perception threshold for human users)

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

Conclusion

- Hard to build simple, reliable software systems over unreliable foundations
- But, given performance trends, commodity file systems move toward relaxing durability for performance
- Reconsider who are guarantees provided to (applications or users) – Synchronous I/O can be fast
- External synchrony performs with 8% of async