GoJournal: a verified, concurrent, crash-safe journaling system

Tej Chajed  
MIT CSAIL

Joseph Tassarotti  
Boston College

Mark Theng  
MIT CSAIL

Ralf Jung  
MPI-SWS

Frans Kaashoek  
MIT CSAIL

Nickolai Zeldovich  
MIT CSAIL
Suppose we want to write a correct file system

**Correct:** file-system operations atomically follow specification, even on crash

**Performant:** take advantage of concurrent operations to efficiently use CPU and I/O
GoJournal gives a storage system efficient, atomic writes

GoJournal

atomic writes of multiple objects

disk
GoJournal gives a storage system efficient, atomic writes

- **GoNFS**: performant NFS server
- **GoJournal**: atomic writes of multiple objects
- **disk**
GoJournal gives a storage system efficient, atomic writes

import "github.com/mit-pdos/go-journal/jrnl"

GoNFS

performant NFS server

GoJournal

atomic writes of multiple objects

disk
GoJournal is a verified journaling system

- GoNFS: performant NFS server
- GoJournal: atomic writes of multiple objects comes with a machine-checked proof
- disk
GoJournal has a practical implementation

>95% throughput of Linux with a single client

Throughput scales with number of concurrent clients
Current approaches cannot handle a system of this complexity

Crash-safe but sequential file systems
FSCQ, Yggdrasil, VeriBetrFS

Concurrent systems
CertiKOS, AtomFS, ...
Current approaches cannot handle a system of this complexity

Crash-safe but sequential file systems
FSCQ, Yggdrasil, VeriBetrFS

Concurrent systems
CertiKOS, AtomFS, ...

Crash safety and concurrency
Perennial 1.0
Contributions

GoJournal, the first verified concurrent journal

Perennial 2.0, a new verification framework

SimpleNFS to evaluate specification

Evaluation showing GoJournal achieves good performance
Contributions

GoJournal, the first verified concurrent journal

Perennial 2.0, a new verification framework

SimpleNFS to evaluate specification

Evaluation showing GoJournal achieves good performance
GoJournal writes operations atomically to disk

// one-time init
var d Disk
jroll := OpenJroll(d)
GoJournal writes operations atomically to disk

// one-time init
var d Disk
jrnl := OpenJrnld

// copy block at 0 to 1 and 2
op := jrnl.Begin()
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
op.Commit()
GoJournal writes operations atomically to disk

concurrent operations are atomic
caller is responsible for locking

// one-time init
var d Disk
jrn := OpenJrnl(d)

// copy block at 0 to 1 and 2
op := jrnl.Begin()
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
op.Commit()
Operations can concurrently manipulate objects within a block

File system has 128-byte inodes

Sub-block access improves concurrency since caller only locks the required objects
Specification challenge: what do concurrently committed operations do?

```
op := jrnl.Begin()
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
op.Commit()
```
Sequential journaling only maintains old and next state

disk: [diagram]
mem: [diagram]

a b c
Sequential journaling only maintains old and next state

precondition

postcondition
An operation’s specification only refers to its disk footprint.

\[ 0 \mapsto b_0 \]

says block at 0 has value \( b_0 \)
An operation’s specification only refers to its disk footprint
An operation’s specification only refers to its disk footprint

```
{0 ↦ b₀, 1 ↦ -, 2 ↦ -}

op := jnl.Begin()
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
op.Commit()
```

```
{0 ↦ b₀, 1 ↦ b₀, 2 ↦ b₀}
```
An operation’s specification only refers to its disk footprint

\[
\begin{align*}
0 & \mapsto b_0 \\
1 & \mapsto - \\
2 & \mapsto - \\
\end{align*}
\]

\[
\text{op} := \text{jnl.} \text{Begin()}
\]

\[
\text{buf} := \text{op.} \text{ReadBuf(0, blockSz)}
\]

\[
\text{op.} \text{OverWrite(1, buf.Data)}
\]

\[
\text{op.} \text{OverWrite(2, buf.Data)}
\]

\[
\text{op.} \text{Commit()}
\]

\[
\begin{align*}
0 & \mapsto b_0 \\
1 & \mapsto b_0 \\
2 & \mapsto b_0 \\
\end{align*}
\]

implies any other data is not involved
Introduce assertion for operation’s view of disk

“disk points-to”

1 \mapsto \text{disk}

“operation points-to”

1 \mapsto \text{op}
Introduce assertion for operation's view of disk

“disk points-to”

1 \mapsto \text{disk}

"on-disk value of block 1"

“operation points-to”

1 \mapsto \text{op}

"op's view of block 1"
Introduce assertion for operation’s view of disk

“disk points-to”

1 \rightarrow - 
\text{disk}

lifting

1 \rightarrow - 
\text{op}

“operation points-to”

on-disk value of block 1

op’s view of block 1
Key idea: operations manipulate an in-memory view of each object
Key idea: operations manipulate an in-memory view of each object
Key idea: operations manipulate an in-memory view of each object
Key idea: operations manipulate an in-memory view of each object.

\[
\begin{align*}
\text{Key Idea:} & \quad \text{operations manipulate an in-memory view of each object} \\
\text{Diagram:} \quad & \\
\end{align*}
\]
Key idea: operations manipulate an in-memory view of each object.

\[ \text{OverWrite} \mapsto b_0 \]

\[ \text{Commit} \mapsto b_0 \]

\[ \text{ReadBuf}(0) \mapsto b_0 \]

\[ \text{crash restores original value} \]
Key idea: operations manipulate an in-memory view of each object

```plaintext
buf := op.

op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
```

```
op := jrn. Begin()
```

```
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
```

```
op.Commit()
```

```plaintext
0 ⟷ b₀
 disk
1 ⟷ --
 disk
2 ⟷ --
 disk
0 ⟷ b₀
 disk
1 ⟷ b₀
 disk
2 ⟷ b₀
 disk
```
Key idea: operations manipulate an in-memory view of each object

```
buf := op.
ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
```

```
op := jrnl.Begin()
```

```
0 ⇔ b₀
disk

1 ⇔ -
disk

2 ⇔ -
disk
```

lifting

```
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
```

```
0 ⇔ b₀
disk

1 ⇔ b₀
disk

2 ⇔ b₀
disk
```

```
op.Commit()
```

```
0 ⇔ b₀
disk

1 ⇔ b₀
disk

2 ⇔ b₀
disk
```
Key idea: operations manipulate an in-memory view of each object

\[
\begin{align*}
0 &\mapsto b_0 & 1 &\mapsto \_ & 2 &\mapsto \_ \\
\text{disk} & & \text{op} & & \text{op} & & \text{op}
\end{align*}
\]

\[
\begin{align*}
\text{op} &:= \text{jrnl.} \text{Begin()} \\
0 &\mapsto b_0 & 1 &\mapsto \_ & 2 &\mapsto \_ \\
\text{op} & & \text{op} & & \text{op} & & \text{op}
\end{align*}
\]

\[
\begin{align*}
\text{buf} &:= \text{op.} \text{ReadBuf(0, blockSz)} \\
\text{op.} \text{OverWrite(1, buf.Data)} \\
\text{op.} \text{OverWrite(2, buf.Data)}
\end{align*}
\]

\[
\begin{align*}
0 &\mapsto b_0 & 1 &\mapsto b_0 & 2 &\mapsto b_0 \\
\text{op} & & \text{op} & & \text{op} & & \text{op}
\end{align*}
\]

\[
\begin{align*}
\text{op.} \text{Commit()} \\
0 &\mapsto b_0 & 1 &\mapsto b_0 & 2 &\mapsto b_0 \\
\text{disk} & & \text{disk} & & \text{disk} & & \text{disk}
\end{align*}
\]
GoJournal has a modular implementation and proof
GoJournal has a modular implementation and proof

- GoJournal
- jrnln: crash-atomic operations
- obj: sub-block objects
- wal: write-ahead log
Write-ahead log implements the core atomicity of the journal
Writes are buffered before being logged

1. write gets buffered

Diagram showing a buffer between memory and disk.
Writes are buffered before being logged

1. Write gets buffered

2. Write gets logged
Challenge 1:
Reads can observe unstable writes

1. Write gets buffered

2. Read returns new data

________________________ system crashes here ____________________
Challenge 1:
Reads can observe unstable writes

1. write gets buffered

2. read returns new data
   —— system crashes here ——

3. read returns old data
Object layer implements sub-block object access

represents just one inode

4096 bytes
Challenge 2:
Reads and writes can proceed concurrently

 reads take subslice of whole block

2    4

4096 bytes
Challenge 2:
Reads and writes can proceed concurrently

reads take subslice of whole block

4096 bytes

concurrent writes don't affect the read
Concurrent writes are unsafe due to read-modify-write sequence
Verification techniques in Perennial 2.0

see paper for details

- Logically atomic crash specifications
- Lock-free reasoning with monotonic counters
- Lifting to specify Commit
- Crash-aware lock specification
Implementation overview

1,300 lines

GoJournal

code is available at
https://github.com/mit-pdos/go-journal
Implementation overview

1,300 lines

code is available at
https://github.com/mit-pdos/go-journal

GoJournal

20,000 lines

verification framework

Coq

Iris
Implementation overview

GoJournal

1,300 lines

goose

model of code

verification framework

20,000 lines

code is available at
https://github.com/mit-pdos/go-journal
Implementation overview

GoJournal

1,300 lines

goose

Proof of GoJournal

26,000 lines

model of code

verification framework

20,000 lines

Coq

Iris

code is available at
https://github.com/mit-pdos/go-journal
Evaluating GoNFS’s performance

Implemented GoNFS, an (unverified) NFS server, on top of GoJournal

Compare against Linux kernel NFS server exporting ext4 (with data=journal mode for fair comparison)
Experimental setup

Hardware: AWS i3.metal
36 cores at 2.3GHz, NVMe SSD

Benchmarks:
• smallfile: metadata heavy
• largefile: data heavy
• app: git clone + make

Run using Linux NFS client
Compare GoNFS throughput to Linux, running on an in-memory disk.
GoNFS gets comparable performance even with a single client

Compare GoNFS throughput to Linux, running on an in-memory disk
Run smallfile with many clients on an NVMe SSD
GoJournal allows GoNFS to scale with number of clients

- GoNFS
- Linux

Run small file with many clients on an NVMe SSD
Concurrency in the journal matters

Serial GoJournal has locks around tricky concurrent parts of WAL.
Summary

GoJournal is a verified, concurrent, crash-safe journaling system

Many concurrency challenges in verification

Demonstrate good performance with GoNFS

for followup questions you can contact Tej (tchajed@mit.edu)