Fork, Explore, Commit: OS Primitives for Agentic Exploration

Cong Wang[1], Yusheng Zheng[2]

[1]Multikernel Technologies, Inc. · [2]UC Santa Cruz

Agentic OS Workshop, ASPLOS 2026

Problem: Agents Explore, But Have Side Effects

- AI agents (SWE-Agent, OpenHands, Claude Code) run shell commands, edit files, install packages → **irreversible side effects**
- Increasingly exploring **multiple paths in parallel**: Tree-of-Thoughts, Reflexion, Best-of-N
- Need to **isolate** each path, **commit** the winner, **discard** the rest

Example: An agent tries 3 bug fixes simultaneously, only the one passing tests should be committed.

What Do Agents Use Today?

Approach	Limitation
Git stashing	Misses untracked files
Temp directories	No atomic commit
Containers	Heavyweight, needs root
Per-file snapshots	Misses shell side effects

None captures all FS modifications with atomic commit/rollback. No parallel branching, no sibling invalidation.

Six Requirements for Agentic Exploration

#	Requirement	Why
R1	**Isolated parallel execution**	Concurrent paths modify same files
R2	**Atomic commit + single-winner**	Apply winner's changes, invalidate siblings
R3	**Hierarchical nesting**	Tree-of-Thoughts explores sub-variants
R4	**Complete filesystem coverage**	Must capture *all* modifications, not just tracked files
R5	**Lightweight, unprivileged, portable**	Sub-ms creation, no root, any FS (ext4, XFS, NFS...)
R6	**Process coordination**	Reliable termination, sibling isolation

No existing OS mechanism satisfies all six requirements.

Why Existing Mechanisms Fall Short

Filesystem Branching

Feature	OverlayFS	Btrfs/ZFS	DM-Snap	**BranchFS**
Portable FS	✓	✗	✓	✓
Nested branches	✗*	✓	✗	✓
Commit-to-parent	✗	✗	✗	✓
Sibling invalidation	✗	✗	✗	✓
No root	✗	✗	✗	✓

Process Management

Feature	pgrp	cgroup	PID ns	**branch()**
Reliable termination	✗	✓	✓	✓
No escape	✗	✓	✓	✓
Sibling isolation	✗	✗	✓	✓
No setup / No root	✓	✗	✗	✓
No PID 1 overhead	✓	✓	✗	✓

- **OverlayFS**: no commit semantics, requires root. **Btrfs/ZFS**: FS-specific. **DM-Snap**: O(depth) latency
- **Process groups**: escapable via `setsid()`. **Cgroups**: need setup + root. **PID ns**: PID 1 overhead
- Composing these in **userspace** creates race windows and fragile error handling

Our Solution: The Branch Context

Definition & Lifecycle

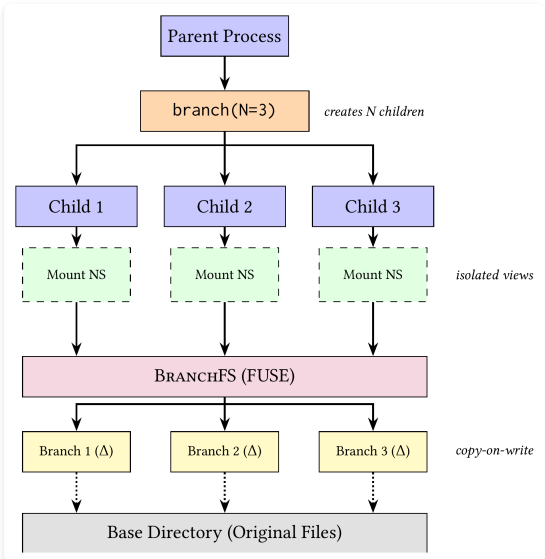
A **branch context** = CoW filesystem view (Δ_i) + confined process group

```
Fork ──► Explore ──► Commit (winner)
                └──► Abort  (losers)
```

Four Core Properties:

1. **Frozen origin**: parent read-only; no merge conflicts
2. **Parallel isolated execution**: N contexts, fully isolated
3. **First-commit-wins**: siblings auto-invalidated
4. **Nestable**: sub-contexts form exploration tree

Architecture



branch() = process coordination; **BranchFS** = filesystem CoW

BranchFS: FUSE-Based Copy-on-Write

File-Level CoW (~3,400 lines of Rust)

- First write → **copy entire file** to branch's delta layer
- Subsequent reads/writes served from **delta copy**
- Unmodified files resolved by walking the **branch chain**

Branch Chain Resolution

1. Check current branch's delta
2. Walk ancestor branches in order
3. Fall back to base directory
4. **Tombstone markers** prevent deleted files from reappearing

Key Design Properties

O(1) branch creation: just create a delta directory

No root privileges: runs entirely as a FUSE daemon

Portable: works on ext4, XFS, NFS, any FS

Commit & Abort

- **Commit**: copy delta to parent, increment **epoch counter** → invalidates all siblings
- **Abort**: discard delta layer → **near-zero cost**
- Cost proportional to **modification size**, not total FS size

The branch() Syscall

Why a Kernel Primitive?

Setting up cgroups + PID ns + mount ns + FS branches in **userspace**:

- **Multi-step** with race windows between steps
- A process can fork between cgroup creation and migration
- **Error-prone** cleanup on partial failure

`branch()` composes all atomically in a **single call** with kernel-side cleanup on failure.

Three Operations

- `BR_CREATE`: fork N children, each in its own branch
- `BR_COMMIT`: apply FS changes, terminate siblings
- `BR_ABORT`: discard changes, terminate self

Interface

```
long branch(int op,
            union branch_attr *attr,
            size_t size);
```

Composable Flags

Flag	Effect
`BR_FS`	Mount namespace + FS branch
`BR_MEMORY`	Page-table CoW
`BR_ISOLATE`	Signal/ptrace barriers
`BR_CLOSE_FDS`	Close inherited FDs

- FS-agnostic: generic `FS_IOC_BRANCH_*` ioctls
- Adding a new branching FS = implement 3 ioctls

BranchContext: Python Integration Library

7 Exploration Patterns

Pattern	Strategy
Speculate	Race N candidates, first success wins
BestOfN	Run N, commit highest-scoring
Reflexion	Sequential retry with feedback
TreeOfThoughts	Hierarchical nested branches
BeamSearch	Keep top-K at each depth
Tournament	Pairwise elimination via judge
Cascaded	Start with 1, fan out on failure

Usage Example

```
ctx = BranchContext("/workspace")

# Best-of-N: try 3 fixes, commit best
results = ctx.best_of_n(
    n=3,
    task=lambda b: try_fix(b),
    score=lambda b: run_tests(b)
)
# Winner automatically committed
```

- Each pattern manages **branch lifecycle** + **process isolation** internally
- Agent code only supplies **per-branch task logic**

github.com/multikernel/branching

Preliminary Evaluation

Branch Creation: O(1)

Base Size	Latency
100 files	292 µs
1,000 files	317 µs
10,000 files	310 µs

Independent of base size.

Commit & Abort

Mod. Size	Commit	Abort
1 KB	317 µs	315 µs
100 KB	514 µs	365 µs
1 MB	2.1 ms	890 µs

Proportional to mod size.

I/O Throughput

Mode	Read
Native	8.8 GB/s
FUSE	1.7 GB/s
Passthrough	**7.2 GB/s**

82% native with passthrough.

Agent workloads dominated by **LLM API latency** (100 ms – 10 s), BranchFS I/O overhead is **negligible**.

Hardware: AMD Ryzen 5 5500U, 8 GB DDR4, NVMe SSD. Median of 10 trials. BranchFS: ~3,400 lines of Rust, FUSE 3.

Related Work & Future Directions

Related Work

- **Containers/gVisor/Dune**: general containment, not branching
- **Speculator**: sequential speculation for distributed FS, we do parallel N-way with first-commit-wins
- **TxOS**: flat, short-lived OS transactions, we're nestable, long-lived, userspace FUSE
- **lwCs/Capsicum**: different isolation granularity

Current Limitations

- **External side effects** (network, IPC) not rolled back
- **Single-winner** only, no multi-branch merge
- File-level CoW: symlinks, hardlinks unsupported
- **BR_MEMORY** not yet implemented

Future Directions

Effect gating: buffer network/IPC until commit; agent gateways provide interposition points

Multi-branch merge: union non-overlapping changes; conflict detection

Roadmap: prototype `branch()` on Linux 6.19; BR_FS + BR_ISOLATE first

Broader use: `n_branches=1` → try-and-rollback for package mgmt, system tuning

Key Takeaways

1. **Agentic exploration** is a first-class OS concern: agents need isolated, parallel workspaces with atomic commit/rollback
2. **Branch context** = new OS abstraction with fork/explore/commit lifecycle and first-commit-wins resolution
3. **BranchFS** = working FUSE implementation: O(1) creation, no root, portable across filesystems
4. **BranchContext** = Python library with 7 ready-to-use exploration patterns
5. **branch() syscall** = proposed kernel primitive for atomic process + FS coordination

BranchFS: **github.com/multikernel/branchfs**

BranchContext: **github.com/multikernel/branching**

Thank You – Questions?

Cong Wang – cwang@multikernel.io

Yusheng Zheng – yzhen165@ucsc.edu

Agentic OS Workshop, ASPLOS 2026

BranchFS: github.com/multikernel/branchfs BranchContext: github.com/multikernel/branching